GENODE

Operating System Framework 16.05

Foundations

Norman Feske

Contents

GENODE LABS

GENODE ⋙ LABS

GENODE LABS

GENODE ⫽ LABS

GENODE LABS

GENODE ⬙ LABS

GENODE LABS

1. Introduction

We are surrounded by operating systems. Each device where multiple software functions are consolidated on a single CPU employs some sort of operating system that multiplexes the physical CPU for the different functions. In our age when even mundane household items get connected to the internet, it becomes increasingly hard to find devices where this is not the case.

Our lives and our society depend on an increasing number of such devices. We have to trust them to fulfill their advertised functionality and to not perform actions that are against our interests. But are those devices trustworthy? In most cases, nobody knows that for sure. Even the device vendors are unable to guarantee the absence of vulnerabilities or hidden functions. This is not by malice. The employed commodity software stacks are simply too complex to reason about them. Software is universally known to be not perfect. So we have seemingly come to accept the common practice where vendors provide a stream of software and firmware updates that fix vulnerabilities once they become publicly known. Building moderately complex systems that are free from such issues appears to be unrealistic. Why is that?

Universal truths The past decades have provided us with enough empirical evidence about the need to be pragmatic about operating-system software. For example, high-assurance systems are known to be expensive and struggle to scale. Consequently, under cost pressure, we can live without high assurance. Security is considered as important. But at the point where the user gets bothered by it, we have to be willing to compromise. Most users would agree that guaranteed quality of service is desirable. But to attain good utilization of cheap hardware, we have to sacrifice such guarantees. Those universal truths have formed our expectations of commodity operating system software.

<div align="center">

Assurance ⟷ Scalability

</div>

In markets where vendors are held liable for the correctness of their products, physical separation provides the highest assurance for the independence and protection of different functions from each other. For example, cars contain dozens of electronic control units (ECU) that can be individually evaluated and certified. However, cost considerations call for the consolidation of multiple functions on a single ECU. At this point, separation kernels are considered to partition the hardware resources into isolated compartments. Because the isolation is only as strong as the correctness of the isolation kernel, such kernels must undergo a thorough evaluation. In the face of being liable, an oversight during the evaluation may have disastrous consequences for the vendor. Each line of code to be evaluated is an expense. Hence, separation kernels are minimized to the lowest possible complexity - up to only a few thousand lines of code.

GENODE ⋙ LABS

The low complexity of separation kernels comes at the cost of being inflexible. Because the hardware resources are partitioned at system-integration time, dynamic workloads are hard to accommodate. The rigidity of the approach stands in the way whenever the number of partitions, the assignment of resources to partitions, and the software running in the partitions have to be changed at runtime.

Even though the high level of assurance as provided by separation kernels is generally desirable, flexibility and the support for dynamic workloads is even more so. For this reason, commodity general-purpose OSes find their way into all kinds of devices except into those where vendors are held liable for the correctness of their products. The former include not only household appliances, network gear, consumer electronics, mobile devices, and certain comfort functions in vehicles but also the IT equipment of governments, smart-city appliances, and surveillance systems. To innovate quickly, vendors accept to make their products reliant on highly complex OS foundations. The trusted computing base (TCB) of all commodity general-purpose operating systems is measured in millions of lines of code. It comprises all the software components that must be trusted to not violate the interests of the user. This includes the kernel, the software executed at the system start, all background services with system privileges, and the actual application software. In contrast to separation kernels, any attempt to assess the correct functioning of the involved code is shallow at best. The trustworthiness of such a system remains uncertain to vendors and users alike. The uncertainty that comes with the staggering TCB complexity becomes a problem when such systems get connected to the internet: Is my internet router under control of a bot net? Is my mobile phone remotely manipulated to wiretap me? Is my TV spying on me when switched off? Are my sensitive documents stored on my computer prone to leakage? Faithfully, we hope the answers to those question to be no. But because it is impossible to reason about the trusted computing base of the employed operating systems, there are no answers.

Apparently, the lack of assurance must be the price to pay for the accommodation of feature-rich dynamic workloads.

Security ⟷ Ease of use

The ease of use of software systems is often perceived as diametrical to security. There are countless mundane examples: Remembering passwords of sufficient strength is annoying. Even more so is picking a dedicated password for each different purpose. Hence, users tend to become lax about choosing and updating passwords. Another example is OpenPGP. Because setting it up for secure email communication is perceived as complicated, business-sensitive information is routinely exchanged unencrypted. Yet another example is the lack of adoption of the security frameworks such as SELinux. Even though they are readily available on commodity OS distributions, com-

GENODE ⋰ LABS

prehending and defining security policies is considered as a black art, which is better left to experts.

How should an operating system strike the balance between being unusably secure and user-friendly insecure?

Utilization ⬌ Accountability

Current-generation general-purpose OSes are designed to utilize physical resources like memory, network bandwidth, computation time, and power in the best way possible. The common approach to maximize utilization is the over-provisioning of resources to processes. The OS kernel pretends the availability of an unlimited amount of resources to each process in the hope that processes will attempt to allocate and utilize as much resources as possible. Its holistic view on all processes and physical resources puts the kernel in the ideal position to balance resources between processes. For example, if physical memory becomes scarce, the kernel is able to uphold the illusion of unlimited memory by temporarily swapping the memory content of inactive processes to disk.

However, the optimization for high utilization comes at the price of indeterminism and effectively makes modern commodity OSes defenseless against denial-of-service attacks driven by applications. For example, because the network load is not accounted to individual network-using applications, a misbehaving network-heavy application is able to degrade the performance of other network applications. As another example, any GUI application is able to indirectly cause a huge memory consumption at the GUI server by creating an infinite amount of windows. If the system eventually runs out of memory, the kernel will identify the GUI server as the offender.

With the help of complex heuristics like process-behaviour-aware schedulers, the kernel tries hard to uphold the illusion of unlimited resources when under pressure. But since the physical resources are ultimately limited, this abstraction is destined to break sooner or later. If it breaks, the consequences may be fatal: In an out-of-memory situation, the last resort of the kernel is to rampage and kill arbitrary processes.

Can an operating system achieve high resource utilization while still being dependable?

Clean-slate approach Surprisingly, by disregarding the practical considerations of existing commodity operating systems, the contradictions outlined above can be resolved by a combination of the following key techniques:

Microkernels as a middle ground between separation kernels and monolithic kernels are able to accommodate dynamic workloads without unreasonably inflating the trusting computing base.

GENODE ⬈ LABS
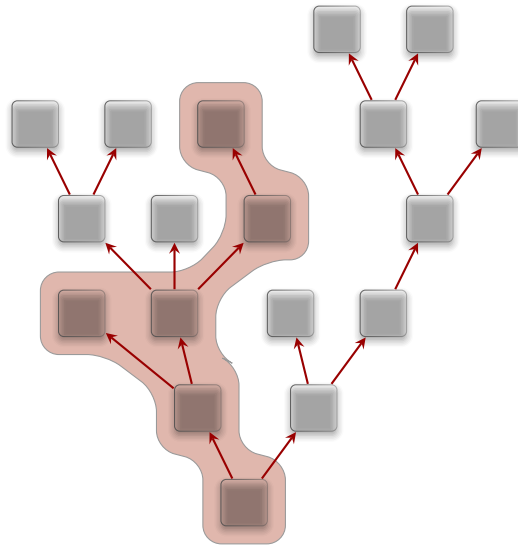
Figure 1: Application-specific trusted computing base

Capability-based security supposedly makes security easy to use by providing an intuitive way to manage authority without the need for an all-encompassing and complex global system policy.

Kernelization of software components aids the deconstruction of complex software into low-complexity security-sensitive parts and high-complexity parts. The latter no longer need to be considered as part of the trusted computing base.

Virtualization can bridge the cap between applications that expect current-generation OSes and a new operating-system design.

The management of budgets within hierarchical organizations shows how limited resources can be utilized and still be properly accounted for.

None of those techniques is new by any means. However, they have never been used as a composition of a general-purpose operating system. This is where Genode comes into the picture.

Application-specific trusted computing base A Genode system is structured as a tree of components where each component (except for the root of the tree) is owned by its parent. The notion of ownership means both responsibility and control. Being responsible for its children, the parent has to explicitly provide the resources needed by its children out of its own resources. It is also responsible to acquaint children with one another and the outside world. In return, the parent retains ultimate control over

GENODE ⋛ LABS

each of its children. As the owner of a child, it has ultimate power over the child's environment, the child's view of the system, and the lifetime of the child. Each child can, in turn, have children, which yields a recursive system structure. Figure 1 illustrates the idea.

At the root of the tree, there is a low-complexity microkernel that is always part of the TCB. The kernel is solely responsible to provide protection domains, threads of execution, and the controlled communication between protection domains. All other system functions such as device drivers, network stacks, file systems, runtime environments, virtual machines, security functions, and resource multiplexers are realized as components within the tree.

The rigid organizational structure enables the system designer to tailor the trusted computing base for each component individually. For example, by hosting a cryptographic function nearby the root of the tree, the function is exposed only to the microkernel but not to complex drivers and protocol stacks that may exist in other branches of the tree. Figure 1 illustrates the TCB of one leaf node. The TCB of the yellow component comprises the chain of parents and grandparents because it is directly or indirectly owned by them. Furthermore, the TCB comprises a service used by the component. But the right branch of tree is unrelated to the component and can thereby disregarded from the yellow component's TCB.

Trading and tracking of physical resources   Unlike traditional operating systems, Genode does not abstract from physical resources. Instead, each component has a budget of physical resources assigned by its parent. The budget allows the component to use the resources within the budget or to assign parts of its budget to its children. The usage and assignment of budgets is a deliberative decision by each component rather than a global policy of the OS kernel. Components are able to trade resource budgets along the branches of the tree. This way, components can offer services to other components without consuming their own resources. The dynamic trading of resource budgets between components allows for a high resource utilization without the over-provisioning of resources. Consequently, the system behavior remains deterministic at all times.

GENODE ⟍ LABS

1.1. Operating-system framework

The Genode OS framework is the implementation of the Genode architecture. It is a tool kit for building highly secure special-purpose operating systems. It scales from embedded systems with as little as 4 MB of memory to highly dynamic general-purpose workloads.

The system is based on a recursive structure. Each program is executed in a dedicated sandbox and gets granted only those access rights and resources that are required to fulfill its specific purpose. Programs can create and manage sub-sandboxes out of their own resources, thereby forming hierarchies where policies can be applied at each level. The framework provides mechanisms to let programs communicate with each other and trade their resources, but only in strictly-defined manners. Thanks to this rigid regime, the attack surface of security-critical functions can be reduced by orders of magnitude compared to contemporary operating systems.

The framework aligns the construction principles of microkernels with Unix philosophy. In line with Unix philosophy, Genode is a collection of small building blocks, out of which sophisticated systems can be composed. But unlike Unix, those building blocks include not only applications but also all classical OS functionalities including kernels, device drivers, file systems, and protocol stacks.

CPU architectures

Genode supports the x86 (32 and 64 bit) and ARM CPU architectures. On x86, modern architectural features such as IOMMUs and hardware virtualization can be utilized. On ARM, Genode is able to take advantage of TrustZone technology.

Kernels

Genode can be deployed on a variety of different kernels including most members of the L4 family (NOVA, Fiasco.OC, OKL4 v2.1, L4ka::Pistachio, seL4, L4/Fiasco). Furthermore, it can be used on top of the Linux kernel to attain rapid development-test cycles during development. Additionally, the framework is accompanied with a custom kernel for ARM-based platforms that has been specifically developed for Genode and thereby further reduces the complexity of the trusted computing base compared to other kernels.

Virtualization

Genode supports virtualization at different levels:

- On NOVA, faithful virtualization via VirtualBox allows the execution of unmodified guest operating systems as Genode subsystems. Alternatively, the Seoul virtual machine monitor can be used to run unmodified Linux-based guest OSes.

- On the Fiasco.OC kernel, L4Linux represents a paravirtualized Linux kernel that allows the use of regular Linux distributions as well as Android.

GENODE ⧉ LABS

- With Noux, there exists a runtime environment for Unix software such as GNU coreutils, bash, GCC, binutils, and findutils.

- On ARM, Genode can be used as TrustZone monitor.

Building blocks

There exist more than 100 ready-to-use components such as

- Device drivers for most common PC peripherals including networking, storage, display, USB, PS/2, Intel wireless, and audio output.

- Device drivers for a variety of ARM-based SoCs such as Texas Instruments OMAP4, Samsung Exynos5, and FreeScale i.MX.

- A GUI stack including a low-complexity GUI server, window management, and widget toolkits such as Qt5.

- Networking components such as TCP/IP stacks and packet-level network services.

GENODE ⬚ LABS

1.2. Licensing and commercial support

Genode is commercially supported by the German company Genode Labs GmbH, which offers trainings, development work under contract, developer support, and commercial licensing:

Genode Labs website

http://www.genode-labs.com

The framework is available under two flavours of licences: an open-source license and commercial licensing. The primary license used for the distribution of the Genode OS framework is the GNU General Public License Version 2 (GNU GPL). In short, the GNU GPL grants everybody the rights to

- Use the Genode OS framework without paying any license fee

- Freely distribute the software

- Modify the source code and distribute modified versions of the software

In return, the GNU GPL requires any modifications and derived work to be published under the same or a GPL-compatible license. For the full license text, refer to

GNU General Public License Version 2

http://genode.org/about/LICENSE

For applications that require more permissive licensing conditions than granted by the GNU GPL, Genode Labs offers the option to commercially license the technology upon request. Please write to *licensing@genode-labs.com*.

GENODE ⟋⟍ LABS

1.3. About this document

This document is split into two parts. Whereas the first part contains the textual description of the architectural and practical foundations, the second part serves as a reference of the framework's programming interface. This allows the first part to stay largely clear from implementation details. Cross-references between both parts are used to connect the conceptual level with the implementation level.

Chapter 2 provides engineering-minded readers with a practical jump start to explore the code and experiment with it. These practical steps are good to get a first impression and will hopefully provide the motivation to engage with the core part of the book, which are the Chapters 3 and 4.

Chapter 3 introduces Genode's high-level architecture by presenting the concept of capability-based security, the resource-trading mechanism, the root of the component tree, and the ways how components can collaborate without mutually trusting each other. Chapter 4 narrows the view on different types of components, namely device drivers, protocol stacks, resource multiplexers, runtime environments, and applications. The remaining part of the chapter focuses on the composition of components.

Chapter 5 substantiates Chapter 2 with all information needed to develop meaningful components. It covers the integration of 3rd-party software, the build system, the tool kit for automated testing, and the Git work flow of the regular Genode developers.

Chapter 6 addresses the system integration. After presenting Genode's holistic configuration concept, it details the usage of the init component, which bootstraps the static part of each Genode system.

Chapter 7 closes the first part with a look behind the scenes. It provides the details and the rationales behind technical decisions, explains the startup procedure of components, shows how Genode's concepts are mapped to kernel mechanisms, and documents known limitations.

The second part of the document gives an overview of the framework's C++ programming interface. The content is partially derived from the actual source code and supplemented with additional background information.

Acknowledgements and feedback   This document greatly benefited from the feedback of the community at the Genode mailing list, the wonderful team at Genode Labs, the thorough review by Adrian-Ken Rueegsegger and Reto Buerki, and several anonymous reviewers. Thanks to everyone who contributed to the effort, be it in the form of reviews, comments, moral support, or through projects commissioned to Genode Labs.

That said, feedback from you as the reader of the document is always welcome. If you identify points you would like to see improved or if you spot grammatical errors, please do not hesitate to contact the author by writing to *norman.feske@genode-labs.com* or to post your feedback to the mailing list http://genode.org/community/mailing-lists.

GENODE 🥬 LABS

Part I.
Foundations

GENODE LABS

2. Getting started

Genode can be approached from two different angles: as an operating-system architecture or as a practical tool kit. This chapter assists you with exploring Genode as the latter. After introducing the recommended development environment, it guides you through the steps needed to obtain the source code (Section 2.1), to use the tool chain (Section 2.3), to test-drive system scenarios (Section 2.4), and to create your first custom component from scratch (Section 2.5).

Recommended development environment Genode is regularly used and developed on GNU/Linux. It is recommended to use the latest long-term support (LTS) version of Ubuntu. Make sure that your installation satisfies the following requirements:

- GNU Make version 3.81 (or newer) needed by the build system,

- *libSDL-dev* needed to run system scenarios directly on Linux,

- *tclsh* and *expect* needed by test-automation and work-flow tools,

- *qemu* and *genisoimage* needed for testing system scenarios on non-Linux platforms via the Qemu emulator.

For using the entire collection of ported 3rd-party software, the following packages should be installed additionally: *byacc*, *autoconf2.64*, *autogen*, *bison*, *flex*, *g++*, *git*, *gperf*, *libxml2-utils*, *subversion*, and *xsltproc*.

Seeking help The best way to get assistance while exploring Genode is to consult the mailing list, which is the primary communication medium of regular users and developers alike. Please feel welcome to join in!

Mailing Lists

http://genode.org/community/mailing-lists

If you encounter a new bug, ambiguous documentation, or a missing feature, please consider opening a corresponding issue at the issue tracker:

Issue tracker

https://github.com/genodelabs/genode/issues

GENODE ⬛ LABS

2.1. Obtaining the source code

The centerpiece of Genode is the source code found within the official Git repository:

Source code at GitHub

> https://github.com/genodelabs/genode

To obtain the source code, clone the Git repository:

```
git clone https://github.com/genodelabs/genode.git
```

After cloning, you can find the source code within the *genode/* directory. In the following, we refer to this directory as *<genode-dir>*.

GENODE ⇘ LABS

2.2. Source-tree structure

Top-level directory   At the root of the directory tree, there is the following content:

***doc*/** Documentation in plain text format, including the release notes of all versions. *Practical hint:* The comprehensive release notes conserve most of the hands-on documentation aggregated over the lifetime of the project. When curious about a certain topic, it is often worthwhile to "grep" for the topic within the release notes to get a starting point for investigation.

***tool*/** Tools and scripts to support the build system, various boot loaders, the tool chain, and the management of 3rd-party source code. Please find more information in the *README* file contained in the subdirectory.

***repos*/** The so-called source-code repositories, which contain the actual source code of the framework components. The source code is not organized within a single source tree but multiple trees. Each tree is called a *source-code repository* and has the same principle structure. At build time, a set of source-code repositories can be selected to be incorporated into the build process. Thereby, the source-code repositories provide a coarse-grained modularization of the framework.

Repositories overview   The *<genode-dir>/repos/* directory contains the following source-code repositories.

***base*/**

> The fundamental framework interfaces as well as the platform-agnostic parts of the core component (Section 3.4).

***base-<platform>*/**   Platform-specific supplements of the *base/* repository where *<platform>* corresponds to one of the following:

> ***linux***
>
> > Linux kernel (both x86_32 and x86_64).
>
> ***nova***
>
> > NOVA microhypervisor. More information about the NOVA platform is provided by Section 7.7.
>
> ***hw***
>
> > The hw platform allows the execution of Genode on bare ARM hardware without the need for a separate kernel. The kernel functionality is included in the core component. More information about the hw platform can be found in Section 7.6.

GENODE ⧓ LABS

foc

Fiasco.OC is a modernized version of the L4/Fiasco microkernel with a completely revised kernel interface fostering capability-based security.

okl4

OKL4 kernel originally developed at Open-Kernel-Labs.

pistachio

L4ka::Pistachio kernel developed at University of Karlsruhe.

fiasco

L4/Fiasco kernel originally developed at Technische Universität Dresden.

sel4

The seL4 microkernel developed by NICTA in Sydney. The support for this kernel is highly experimental.

os/

OS components such as the init component, device drivers, and basic system services.

demo/

Various services and applications used for demonstration purposes, for example the graphical application launcher and the tutorial browser described in Section 2.4 can be found here.

hello_tutorial/

Tutorial for creating a simple client-server scenario. This repository includes documentation and the complete source code.

libports/

Ports of popular open-source libraries, most importantly the C library. Among the 3rd-party libraries are Qt5, libSDL, freetype, Python, ncurses, Mesa, and libav.

dde_linux/

Device-driver environment for executing Linux kernel subsystems as user-level components. Among the subsystems are the USB stack, the Intel wireless stack, and the TCP/IP stack.

dde_ipxe/

Device-driver environment for executing network drivers of the iPXE project.

dde_bsd/

Device-driver environment for audio drivers ported from OpenBSD.

GENODE ⚞ LABS

dde_rump/

Port of rump kernels, which are used to execute subsystems of the NetBSD kernel as user-level components. The repository contains a server that uses a rump kernel to provide various NetBSD file systems.

ports/

Ports of 3rd-party applications.

ports-<platform>/

These platform-specific source-code repositories contain software that capitalizes special features of the respective kernel platform. For the Fiasco.OC platform, `ports-foc` hosts a port of the L4Linux kernel.

gems/

Components that use both native Genode interfaces as well as features of other high-level repositories, in particular shared libraries provided by *libports/*.

GENODE 🖉 LABS

2.3. Using the build system

Genode relies on a custom tool chain, which can be downloaded at the following website:

Tool chain

> http://genode.org/download/tool-chain

Build directory   The build system never touches the source tree but generates object files, libraries, and programs in a dedicated build directory. We do not have a build directory yet. For a quick start, let us create one using the following command:

```
cd <genode-dir>
./tool/create_builddir linux_x86
```

The command creates a new build directory at `build/linux_x86`. The build directory is readily configured for using GNU/Linux as base platform.

Build configuration   Before using the build directory, it is recommended to revisit and possibly adjust the build configuration, which is located in the *etc/* subdirectory of the build directory, e. g., *build/linux_x86/etc/*. The *build.conf* file contains global build parameters, in particular the selection of source-code repositories to be incorporated. It is also a suitable place for adding global build options. For example, for enabling GNU make to use 4 CPU cores, add the following line to the *build.conf* file:

```
MAKE += -j4
```

Building components   The recipe for building a component has the form of a *target.mk* file within the *src/* directory of one of the source-code repositories. For example, the *target.mk* file of the init component is located at *<genode-dir>/repos/os/src/init/target.mk*. To build the component, execute the following command from within the build directory:

```
make init
```

The argument "init" refers to the path relative to the *src/* subdirectory. The build system determines and builds all targets found under this path in all source-code repositories. When the build the is finished, the resulting executable binary can be found in a subdirectory that matches the target's path. Additionally, the build system installs a symbolic link in the *bin/* subdirectory that points to the executable binary.

GENODE ⬚ LABS

If the specified path contains multiple *target.mk* files in different subdirectories, the build system builds all of them. For example, the following command builds all targets found within one of the *<repo>/src/drivers/* subdirectories:

```
make drivers
```

Furthermore, it is possible to specify multiple targets at once. The following command builds both the init component and the timer driver:

```
make init drivers/timer
```

GENODE 🖋 LABS

2.4. A simple system scenario

The build directory offers much more than an environment for building components.
It supports the automation of system-integration work flows, which typically include
the following steps:

1. Building a set of components,

2. Configuring the static part of a system scenario,

3. Assembling a boot directory with all ingredients needed by the scenario,

4. Creating a boot image that can be loaded onto the target platform,

5. Booting the target platform with the boot image,

6. Validating the behavior of the scenario.

The recipe for such a sequence of steps can be expressed in the form of a so-called run
script. Each run script represents a system scenario and entails all information required
to reproduce the scenario. Run scripts can reside within the *run/* subdirectory of any
source-code repository.

Genode comes with a ready-to-use run script showcasing simple graphical demo
scenario. It is located at *<genode-dir>/repos/os/run/demo.run* and can be executed from
within the build directory via:

```
make run/demo
```

The command will lookup a run script called *demo.run* in all repositories listed in
etc/build.conf. It will eventually find the run script within the *os/* repository. After com-
pleting the build of all components needed, the command will then automatically start
the scenario. Because the build directory was created for the *linux_x86* platform, the
scenario will be executed directly on the host system where each Genode component
resides in a distinct Linux process. To explore the scenario, follow the instructions given
by the graphical tutorial browser.

The terminal where the `make run/demo` command was issued displays the log out-
put of the Genode system. To cancel the execution, hit *control-c* in the terminal.

Targeting a microkernel   Whereas the ability to run system scenarios on top of Linux
allows for the convenient and rapid development of components and protocols, Gen-
ode is primarily designed for the use of microkernels. The choice of the microkernel
to use is up to the user of the framework and may depend on various factors like the
feature set, the supported hardware architectures, the license, or the development com-
munity. To execute the demo scenario directly on the NOVA microhypervisor, the fol-
lowing preparatory steps are needed:

GENODE ⬈ LABS

1. Download the 3rd-party source code of the NOVA microhypervisor

   ```
   <genode-dir>/tool/ports/prepare_port nova
   ```

   The *prepare_port* tool downloads the source code of NOVA to a subdirectory at *<genode-dir>/contrib/nova-<hash>/* where *<hash* uniquely refers to the prepared version of NOVA.

2. On real hardware, the scenario needs a framebuffer driver. The VESA driver relies on a 3rd-party x86-emulation library in order to execute the VESA BIOS code. Download the 3rd-party source code of the *x86emu* library:

   ```
   <genode-dir>/tool/ports/prepare_port x86emu
   ```

   The source code will be downloaded to *<genode-dir>/contrib/x86emu-<hash>/*.

3. Create a build directory for the *nova_x86_64* platform:

   ```
   <genode-dir>/tool/create_builddir nova_x86_64
   ```

   This step creates a new build directory at *<genode-dir>/build/nova_x86_64*.

4. Adjust the build configuration by editing the file *etc/build.conf* within the just-created build directory. Apart from enabling the parallelization of the build process as mentioned in Section 2.3, we need to incorporate the *libports* source-code repository into the build process by uncommenting the corresponding line in the configuration. Otherwise the build system would fail to build the VESA driver, which resides within *libports/*.

With those preparations in place, issue the execution of the demo run script from within the build directory:

```
cd <genode-dir>/build/nova_x86_64
make run/demo
```

This time, an instance of Qemu will be started to execute the demo scenario. The Qemu command-line arguments appear in the log output. As suggested by the arguments, the scenario is supplied to Qemu as an ISO image residing at *var/run/demo.iso*. This ISO image can not only be used with Qemu but also with a real machine. For example, creating a bootable USB stick with the system scenario is as simple as writing the ISO image onto an USB stick:

```
sudo dd if=var/run/demo.iso of=/dev/<usb-device> bs=8M conv=fsync
```

GENODE 🖉 LABS

Note that *<usb-device>* refers to the device node of an USB stick. It can be determined using the `dmesg` command after plugging-in the USB stick. For booting from the USB stick, you may need to adjust the BIOS settings of the test machine accordingly.

GENODE ⟟ LABS

2.5. Hello world

This section introduces the steps needed to create and execute a simple custom component that prints a hello-world message.

2.5.1. Using a custom source-code repository

In principle, it would be possible to add a new component to one of the existing source-code repositories found at *<genode-dir>/repos/*. However, unless the component is meant to be incorporated into upstream development of the Genode project, it is generally recommended to keep custom code separate from Genode's code base. This eases future updates to new versions of Genode and allows you to pick a revision-control system of your choice.

The new repository must appear within the *<genode-dir>/repos/* directory. This can be achieved by either hosting it as a subdirectory or by creating a symbolic link that points to an arbitrary location of your choice. For now, let us host a new source-code repository called "lab" directly within the *repos/* directory.

```
cd <genode-dir>
mkdir repos/lab
```

The lab repository will contain the source code and build rules for a single component as well as a run script for executing the component within Genode. Component source code reside in a *src/* subdirectory. By convention, the *src/* directory contains further subdirectories for hosting different types of components, in particular *server* (services and protocol stacks), *driver* (hardware-device drivers), and *app* (applications). For the hello-world component, an appropriate location would be *src/app/hello/*:

```
mkdir -p repos/lab/src/app/hello
```

2.5.2. Source code and build description

The *hello/* directory contains both the source code and the build description of the component. The main part of each component typically resides in a file called *main.cc*. Hence, for a hello-world program, we have to create the *repos/lab/src/app/hello/main.cc* file with the following content:

GENODE ⬚ LABS

```
#include <base/component.h>
#include <base/log.h>

Genode::size_t Component::stack_size() { return 64*1024; }

void Component::construct(Genode::Env &)
{
  Genode::log("Hello world");
}
```

The *base/component.h* header contains the interface each component must implement. The interface consists of two functions. The `stack_size` function dimensions the stack of the component's initial entrypoint thread. The function allows the component developer to adjust the memory usage by taking knowledge about the component's behavior into account. Whereas a stack of 64 KiB is more than enough for most components, a higher value is sensible for components that rely on deeply recursive functions or that are known to host large objects as local variables on the stack.

The `construct` function is called by the component's execution environment to initialize the component. The interface to the execution environment is passed as argument. This interface allows the application code to interact with the outside world. Normally, the implementation of the `construct` function creates an object representing the application as a `static` local variable. But the simple example above merely produces a log message. The `log` function is defined in the *base/log.h* header.

The component does not exit after the `construct` function returns. Instead, it becomes ready to respond to requests or signals originating from other components. The example above does not interact with other components though. Hence, it will just keep waiting infinitely.

Please note that there exists a recommended coding style for genuine Genode components. If you consider submitting your work to the upstream development of the project, please pay attention to these common guidelines.

Coding-style guidelines

> http://genode.org/documentation/developer-resources/coding_style

The source file *main.cc* is accompanied by a build-description file called *target.mk*. It contains the declarations for the source files, the libraries used by the component, and the name of the component. Create the file *repos/lab/src/app/hello/target.mk* with the following content:

```
TARGET = hello
SRC_CC = main.cc
LIBS  += base
```

GENODE ⟋⟍ LABS

2.5.3. Building the component

With the build-description file in place, it is time to build the new component, for example from within the *linux_x86* build directory as created in Section 2.4. To aid the build system to find the component, we have to extend the build configuration *<build-dir>/etc/build.conf* by appending the following line:

```
REPOSITORIES += $(GENODE_DIR)/repos/lab
```

By adding this line, the build system will consider our custom source-code repository. To build the component, issue the following command:

```
make app/hello
```

This step builds the base libraries of the framework, compiles the *main.cc* file and links the executable ELF binary called "hello". The result can be found in the *<build-dir>/app/hello/* subdirectory.

2.5.4. Defining a system scenario

For testing the component, we need to define a system scenario that incorporates the component. As mentioned in Section 2.4, such a description has the form of a run script. To equip the *lab* repository with a run script, we first need to create a *lab/run/* subdirectory:

```
mkdir <genode-dir>/repos/lab/run
```

Within this directory, we create the file *<genode-dir>/repos/lab/run/hello.run* with the following content:

GENODE ⫶ LABS

```
build "core init app/hello"
create_boot_directory
install_config {
 <config>
   <parent-provides>
     <service name="LOG"/>
   </parent-provides>
   <default-route>
     <any-service> <parent/> </any-service>
   </default-route>
   <start name="hello">
     <resource name="RAM" quantum="10M"/>
   </start>
 </config>
}
build_boot_image "core init hello"
append qemu_args "-nographic -m 64"
run_genode_until {Hello world.*\n} 10
```

This run script performs the following steps:

1. It builds the components core, init, and app/hello.

2. It creates a fresh boot directory at *<build-dir>/var/run/hello*. This directory contains all files that will end up in the final boot image.

3. It creates a configuration for the init component. The configuration starts the hello component as the only child of init. Session requests originating from the hello component will always be directed towards the parent of init, which is core.

4. It assembles a boot image with the executable ELF binaries core, init, and hello. The binaries are picked up from the *<build-dir>/bin/* subdirectory.

5. It instructs Qemu (if used) to disable the graphical output.

6. It triggers the execution of the system scenario and watches the log output for the given regular expression. The execution ends when the log output appears or after a timeout of 10 seconds.

The run script can be executed from within the build directory via the command:

```
make run/hello
```

After the boot output of the used kernel, the scenario will produce the following output:

GENODE ⚞ LABS

```
[init -> test-printf] Hello world

Run script execution successful.
```

The label within the brackets at the start of each line identifies the component where the message originated from. The final line is printed by the run tool after it successfully matched the log output against the regular expression specified to the run_genode_until command.

GENODE ⯈ LABS

3. Architecture

Contemporary operating systems are immensely complex to accommodate a large variety of applications on an ever diversifying spectrum of hardware platforms. Among the functionalities provided by a commodity operating system are device drivers, protocol stacks such as file systems and network protocols, the management of hardware resources, as well as the provisioning of security functions. The latter category is meant for protecting the confidentiality and integrity of information and the lifelines of critical functionality. For assessing the effectiveness of such a security function, two questions must be considered. First, what is the potential attack surface of the function? The answer to this question yields an assessment about the likelihood of a breach. Naturally, if there is a large number of potential attack vectors, the security function is at high risk. The second question is: What is the reach of a defect? If the compromised function has unlimited access to all information processed on the system, the privacy of all users may be affected. If the function is able to permanently install software, the system may become prone to back doors.

Today's widely deployed operating systems do not isolate security-critical functions from the rest of the operating system. In contrary, they are co-located with most other operating-system functionality in a single high-complexity kernel. Thereby, those functions are exposed to the other parts of the operating system. The likelihood of a security breach is as high as the likelihood of bugs in an overly complex kernel. In other words: It is certain. Moreover, once an in-kernel function has been compromised, the defect has unlimited reach throughout the system.

The Genode architecture was designed to give more assuring answers to the two questions stated. Each piece of functionality should be exposed to only those parts of the system, on which it ultimately depends. But it remains hidden from all unrelated parts. This minimizes the attack surface on individual security functions and thereby reduces the likelihood for a security breach. In the event that one part of the system gets compromised, the scope of the defect is limited to the particular fragment and its dependent parts. Unrelated functionalities remain unaffected. To realize this idea, Genode composes the system out of many components that interact with each other. Each component serves a specific role and uses well-defined interfaces to interact with its peers. For example, a network driver accesses a physical network card and provides a bidirectional stream of network packets to another component, which, in turn, may process the packets using a TCP/IP stack and a network application. Even though the network driver and the TCP/IP stack cooperate when processing network packets, they are living in separate protection domains. So a bug in one component cannot observe or corrupt the internal state of another.

Such a component-based architecture, however, raises a number of questions, which are addressed throughout this chapter. Section 3.1 explains how components can cooperate without inherently trusting each other. Section 3.2 answers the questions of who defines the relationship between components and how components become ac-

quainted with each other. An operating system ultimately acts on physical hardware resources such as memory, CPUs, and peripheral devices. Section 3.4 describes how such resources are made available to components. Section 3.5 answers the question of how a new component comes to life. The variety of relationships between components and their respective interfaces call for different communication primitives. Section 3.6 introduces Genode's inter-component communication mechanisms in detail.

GENODE LABS

3.1. Capability-based security

This section introduces the nomenclature and the general model of Genode's capability-based security concept. The Genode OS framework is not tied to one kernel but supports a variety of kernels as base platforms. On each of those base platforms, Genode uses different kernel mechanisms to implement the general model as closely as possible. Note however that not all kernels satisfy the requirements that are needed to implement the model securely. For assessing the security of a Genode-based system, the respective platform-specific implementation must be considered. Sections 7.6 and 7.7 provide details for selected kernels.

3.1.1. Capability spaces, object identities, and RPC objects

Each component lives inside a protection domain that provides an isolated execution environment.
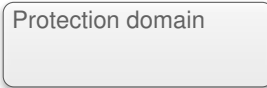
Genode provides an object-oriented way of letting components interact with each other. Analogously to object-oriented programming languages, which have the notion of objects and pointers to objects, Genode introduces the notion of RPC objects and capabilities to RPC objects.

An *RPC object* provides a remote-procedure call (RPC) interface. Similar to a regular object, an RPC object can be constructed and accessed from within the same program. But in contrast to a regular object, it can also be called from the outside of the component. What a pointer is to a regular object, a *capability* is to an RPC object. It is a token that unambiguously refers to an RPC object. In the following, we represent an RPC object as follows.
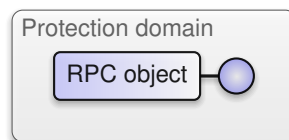
The circle represents the capability associated with the RPC object. Like a pointer to an object, that can be used to call a function of the pointed-to object, a capability can be used to call functions of its corresponding RPC object. However, there are two important differences between a capability and a pointer. First, in contrast to a pointer that can be created out of thin air (e. g., by casting an arbitrary number to a pointer), a capability cannot be created without an RPC object. At the creation time of an RPC object, Genode creates a so-called *object identity* that represents the RPC object in the kernel. Figure 2 illustrates the relationship of an RPC object and its object identity.

GENODE ⬘ LABS
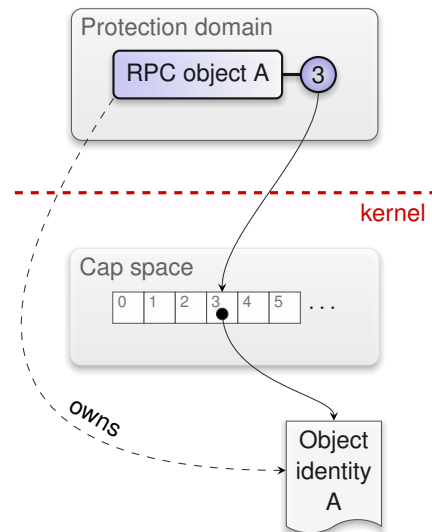
Figure 2: Relationship between an RPC object and its corresponding object identity.

For each protection domain, the kernel maintains a so-called capability space, which is a name space that is local to the protection domain. At the creation time of an RPC object, the kernel creates a corresponding object identity and lets a slot in the protection domain's capability space refer to the RPC object's identity. From the component's point of view, the RPC object A has the name 3. When interacting with the kernel, the component can use this number to refer to the RPC object A.

3.1.2. Delegation of authority and ownership

The second difference between a pointer and a capability is that a capability can be passed to different components without losing its meaning. The transfer of a capability from one protection domain to another delegates the authority to use the capability to the receiving protection domain. This operation is called *delegation* and can be performed only by the kernel. Note that the originator of the delegation does not diminish its authority by delegating a capability. It merely shares its authority with the receiving protection domain. There is no superficial notion of access rights associated with a capability. The possession of a capability ultimately enables a protection domain to use it and to delegate it further. A capability should hence be understood as an access right. Figure 3 shows the delegation of the RPC object's capability to a second protection domain and a further delegation of the capability from the second to a third protection domain. Whenever the kernel delegates a capability from one to another protection domain, it inserts a reference to the RPC object's identity into a free slot of the target's capability space. Within protection domain 2 shown in Figure 3, the RPC object can

GENODE ⧼ LABS
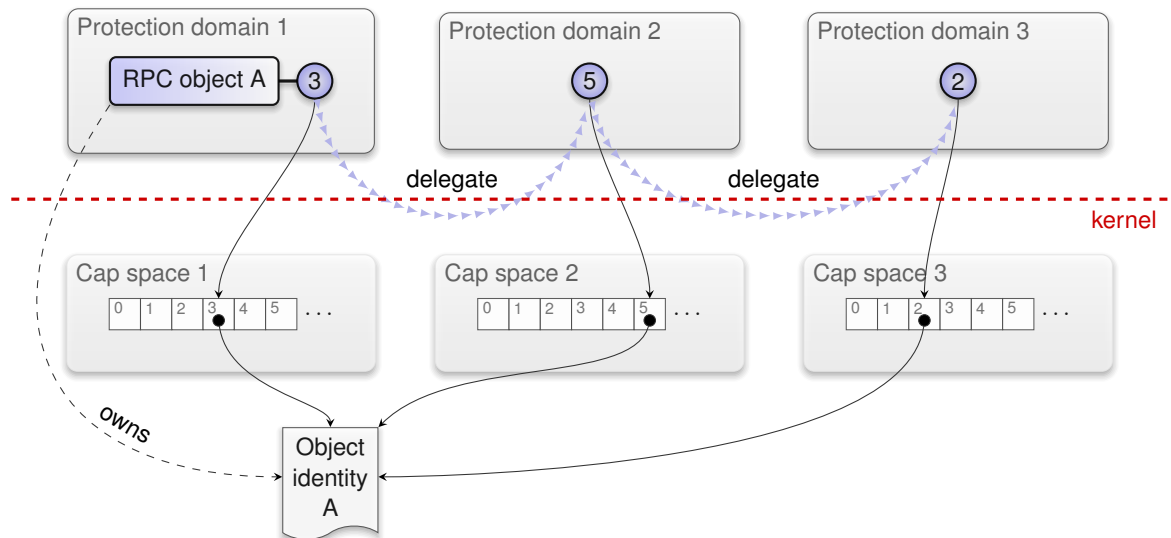
Figure 3: The transitive delegation of a capability from one protection domain to others.

be referred to by the number 5. Within protection domain 3, the same RPC object is known as 2. Note that the capability delegation does not hand over the ownership of the object identity to the target protection domain. The ownership is always retained by the protection domain that created the RPC object.

Only the owner of an RPC object is able to destroy it along with the corresponding object identity. Upon destruction of an object identity, the kernel removes all references to the vanishing object identity from all capability spaces. This effectively renders the RPC object inaccessible for all protection domains. Once the object identity for an RPC object is gone, the owner can destruct the actual RPC object.

3.1.3. Capability invocation

Capabilities enable components to call methods of RPC objects provided by different protection domains. A component that uses an RPC object plays the role of a *client* whereas a component that owns the RPC object acts in the role of a *server*. The interplay between client and server is very similar to a situation where a program calls a local function. The caller deposits the function arguments at a place where the callee will be able to pick them up and then passes control to the callee. When the callee takes over control, it obtains the function arguments, executes the function, copies the results to a place where the caller can pick them up, and finally hands back the control to the caller. In contrast to a program-local function call, however, client and server are different *threads* in their respective protection domains. The thread at the server side is called

GENODE LABS
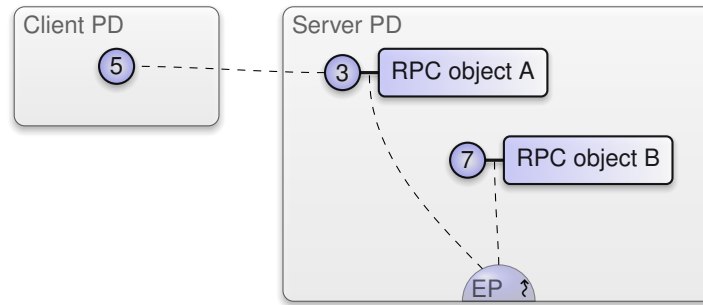
Figure 4: The RPC object A and B are associated with the server's entrypoint. A client has a
capability for A but not for B. For brevity, the kernel-protected object identities are not
depicted. Instead, the dashed line between the capabilities shows that both capabili-
ties refer to the same object identity.

entrypoint denoting the fact that it becomes active only when a call from a client enters
the protection domain or when an asynchronous notification comes in. Each compo-
nent has at least one initial entrypoint, which is created as part of the component's
execution environment.

The wiggly arrow denotes that the entrypoint is a thread. Besides being a thread that
waits for incoming requests, the entrypoint is responsible for maintaining the associ-
ation between RPC objects and their corresponding capabilities. The previous figures
illustrated this association with the link between the RPC object and its capability. In
order to become callable from the outside, an RPC object must be associated with a
concrete entrypoint. This operation results in the creation of the object's identity and
the corresponding capability. During the lifetime of the object identity, the entrypoint
maintains the association between the RPC object and its capability in a data struc-
ture called *object pool*, which allows for looking up the matching RPC object for a given
capability. Figure 4 shows a scenario where two RPC objects are associated with one
entrypoint in the protection domain of a server. The capability for the RPC object A has
been delegated to a client.

If a protection domain is in possession of a capability, each thread executed within
this protection domain can issue a call to a member function of the RPC object that is
referred to by the capability. Because this is not a normal function call but the invocation
of an object located in a different protection domain, this operation has to be provided
by the kernel. Figure 5 illustrates the interaction of the client, the kernel, and the server.
The kernel operation takes the client-local name of the invoked capability, the opcode
of the called function, and the function arguments as parameters. Upon entering the
kernel, the client's thread is blocked until it receives a response. The operation of the

GENODE 𝕃 LABS
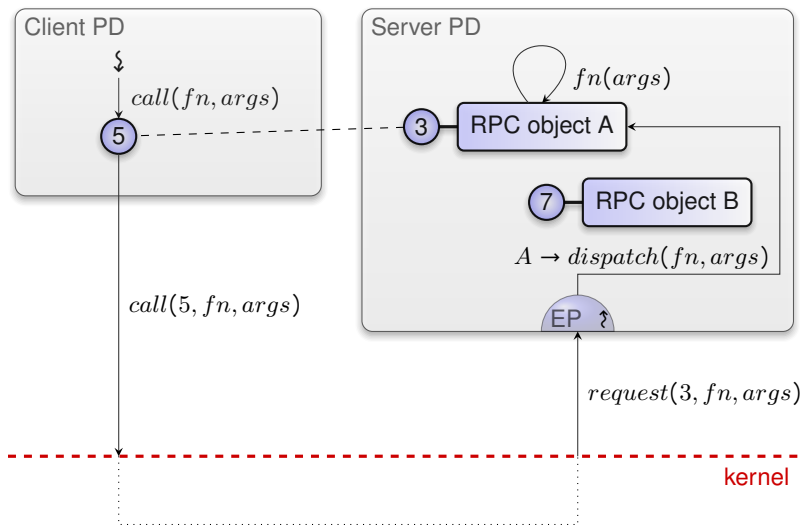
Figure 5: Control flow between client and server when the client calls a method of an RPC object.

kernel is represented by the dotted line. The kernel uses the supplied local name as an index into the client's capability space to look up the object identity, to which the capability refers. Given the object identity, the kernel is able to determine the protection domain and the corresponding entrypoint that is associated with the object identity and wakes up the entrypoint's thread with information about the incoming request. Among this information is the server-local name of the capability that was invoked. Note that the kernel has translated the client-local name to the corresponding server-local name. The capability name spaces of client and server are entirely different. The entrypoint uses this number as a key into its object pool to find the locally implemented RPC object A that belongs to the invoked capability. It then performs a method call of the so-called *dispatch* function on the RPC object. The dispatch function maps the supplied function opcode to the matching member function and calls this function with the request arguments.

The member function may produce function results. Once the RPC object's member function returns, the entrypoint thread passes the function results to the kernel by performing the kernel's *reply* operation. At this point, the server's entrypoint becomes ready for the next request. The kernel, in turn, passes the function results as return values of the original call operation to the client and wakes up the client thread.

GENODE ⩘ LABS
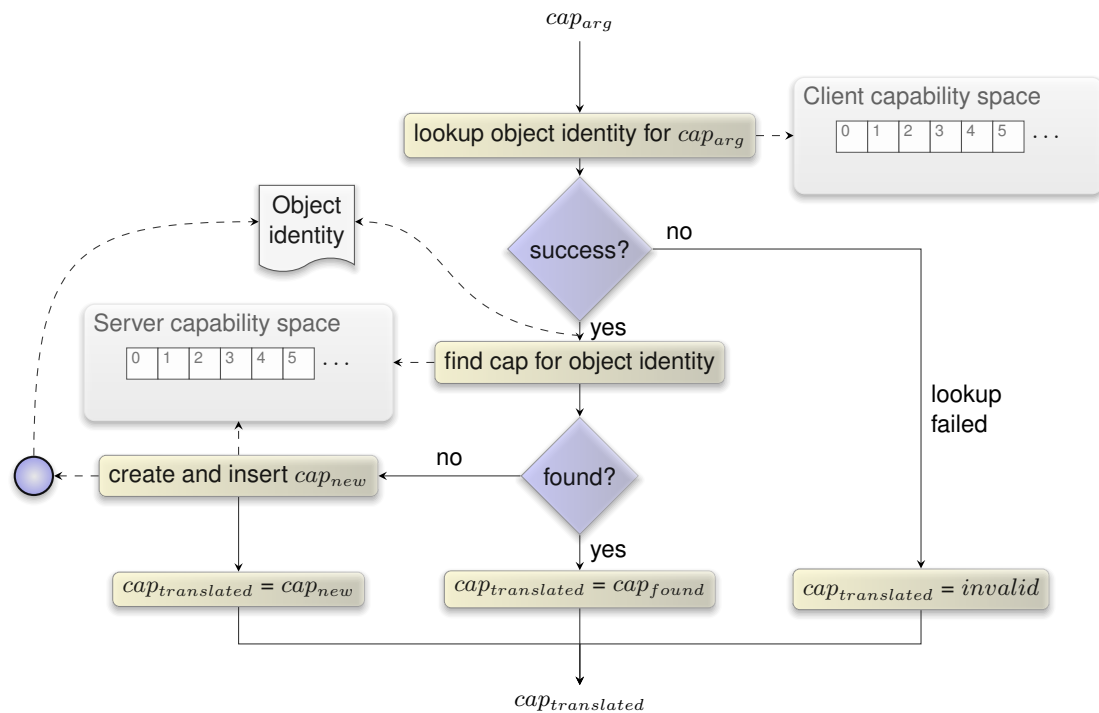
Figure 6: Procedure of delegating a capability specified as RPC argument from a client to a server.

3.1.4. Capability delegation through capability invocation

Section 3.1.2 explained that capabilities can be delegated from one protection domain to another via a kernel operation. But it left open the question of how this procedure works. The answer is the use of capabilities as RPC message payload. Similar to how a caller of a regular function can pass a pointer as an argument, a client can pass a capability as an argument to an RPC call. In fact, passing capabilities as RPC arguments or results is synonymous to delegating authority between components. If the kernel encounters a capability as an argument of a call operation, it performs the steps illustrated in Figure 6. The local names are denoted as cap, e.g., cap_{arg} is the local name of the object identity at the client side, and $cap_{translated}$ is the local name of the same object identity at the server side.

1. The kernel looks up the object identity in the capability space of the client. This lookup may fail if the client specified a number of an empty slot of its capability space. Only if the lookup succeeds is the kernel able to obtain the object identity referred to by the argument. Note that under no circumstances can the client refer to object identities, for which it has no authority because it can merely specify the object identities reachable through its capability space. For all non-empty

GENODE ⬚ LABS

slots of its capability space, the protection domain was authorized to use their referenced object identities by the means of prior delegations. If the lookup fails, the translation results in an invalid capability passed to the server.

2. Given the object identity of the argument, the kernel searches the server's capability space for a slot that refers to the object identity. Note that the term "search" does not necessarily refer to an expensive linear search. The efficiency of the operation largely depends on the kernel implementation.

3. If the server already possesses a capability to the object identity, the kernel translates the argument to the server-local name when passing it as part of the request to the server. If the server does not yet possess a capability to the argument, the kernel installs a new entry into the server's capability space. The new entry refers to the object identity of the argument. At this point, the authority over the object identity has been delegated from the client to the server.

4. The kernel passes the translated or just-created local name of the argument as part of the request to the server.

Even though the above description covered the delegation of a single capability specified as argument, it is possible to delegate more than one capability with a single RPC call. Analogously to how capabilities can be delegated from a client to a server as arguments of an RPC call, capabilities can be delegated in the other direction as part of the reply of an RPC call. The procedure in the kernel is the same in both cases.

GENODE 🖎 LABS
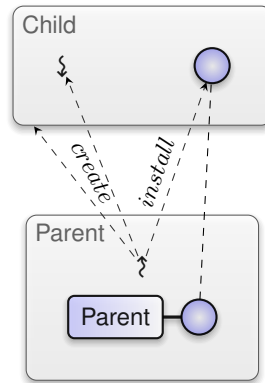
Figure 7: Initial relationship between a parent and a newly created child.

3.2. Recursive system structure

The previous section introduced capability delegation as the fundamental mechanism to share authority over RPC objects between protection domains. But in the given examples, the client was already in possession of a capability to the server's RPC object. This raises the question of how do clients get acquainted to servers?

3.2.1. Component ownership

In a Genode system, each component (except for the very first component called core) has a parent, which owns the component. The *ownership* relation between a parent and a child is two-fold.

On the one hand, ownership stands for *responsibility*. Each component requires physical resources such as the memory or in-kernel data structures that represent the component in the kernel. The parent is responsible for providing a budget of those physical resources to the child at the child's creation time but also during the child's entire lifetime. As the parent has to assign a fraction of its own physical resources to its children, it is the parent's natural interest to maintain the balance of the physical resources split between itself and each of its children. Besides being the provider of resources, the parent defines all aspects of the child's execution and serves as the child's primary point of contact for seeking acquaintances with other components.

GENODE LABS

On the other hand, ownership stands for *control*. Because the parent has created its children out of its own resources, it is in the position to exercise ultimate power over its children. This includes the decision to destruct a child at any time in order to regain the resources that were assigned to the child. But it is also in control over the relationships of the child with other components known to the parent.

Each new component is created as an empty protection domain. It is up to the parent to populate the protection domain with code and data, and to create a thread that executes the code within the protection domain. At creation time, the parent installs a single capability called *parent capability* into the new protection domain. The parent capability enables the child to perform RPC calls to the parent. The child is unaware of anything else that exists in the Genode system. It does not even know its own identity nor the identity of its parent. All it can do is issue calls to its parent using the parent capability. Figure 7 depicts the situation right after the creation of a child component. A thread in the parent component created a new protection domain and a thread residing in the protection domain. It also installed the parent capability referring to an RPC object provided by the parent. To provide the RPC object, the parent has to maintain an entrypoint. For brevity, entrypoints are not depicted in this and the following figures. Section 3.5 covers the procedure of creating a component in detail.

The ownership relation between parent and child implies that each component has to inherently trust its parent. From a child's perspective, its parent is as powerful as the kernel. Whereas the child has to trust its parent, a parent does not necessarily need to trust its children.

3.2.2. Tree of components

The parent-child relationship is not limited to a single level. Child components are free to use their resources to create further children, thereby forming a tree of components. Figure 8 shows an example scenario. The init component creates subsystems according to its configuration. In the example, it created two children, namely a GUI and a launcher. The latter allows the user to interactively create further subsystems. In the example, launcher was used to start an application.

At each position in the tree, the parent-child interface is the same. The position of a component within the tree is just a matter of composition. For example, by a mere configuration change of init, the application could be started directly by the init component and would thereby not be subjected to the launcher.

3.2.3. Services and sessions

The primary purpose of the parent interface is the establishment of communication channels between components. Any component can inform its parent about a service that it provides. In order to provide a service, a component needs to create an RPC object implementing the so-called *root interface*. The root interface offers functions for

GENODE ⤸ LABS
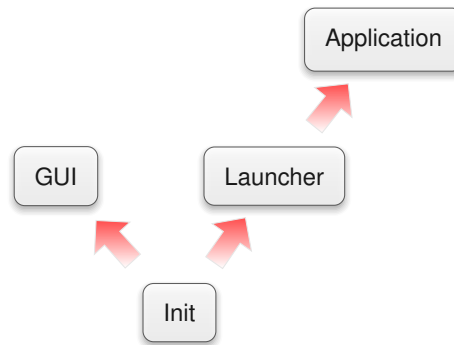
Figure 8: Example of a tree of components. The red arrow represents the ownership relation.

creating and destroying sessions of the service. Figure 9 shows a scenario where the GUI component announces its service to the init component. The announce function takes the service name and the capability for the service's root interface as arguments. Thereby, the root capability is delegated from the GUI to init.

It is up to the parent to decide what to do with the announced information. The parent may ignore the announcement or remember that the child "GUI" provides a service "GUI". A component can announce any number of services via subsequent announce calls.

The counterpart of the service announcement is the creation of a session by a client by issuing a *session* request to its parent. Figure 10 shows the scenario where the application requests a "GUI" session. Along with the session call, the client specifies the type of the service and a number of session arguments. The session arguments enable the client to inform the server about various properties of the desired session. In the example, the client informs the server that the client's window should be labeled with the name "browser". As a result of the session request, the client expects to obtain a capability to an RPC object that implements the session interface of the requested service. Such a capability is called *session capability*.

When the parent receives a session request from a child, it is free to take a policy decision on how to respond to the request. This decision is closely related to the management of resources described in Section 3.3.2. There are the following options.

Parent denies the service   The parent may deny the request and thereby prevent the child from using a particular service.

Parent provides the service   The parent could decide to implement the requested service by itself by handing out a session capability for a locally implemented RPC object to the child.

Server is another child   If the parent has received an announcement of the service from another child, it may decide to direct the session request to the other child.

GENODE ⍦ LABS
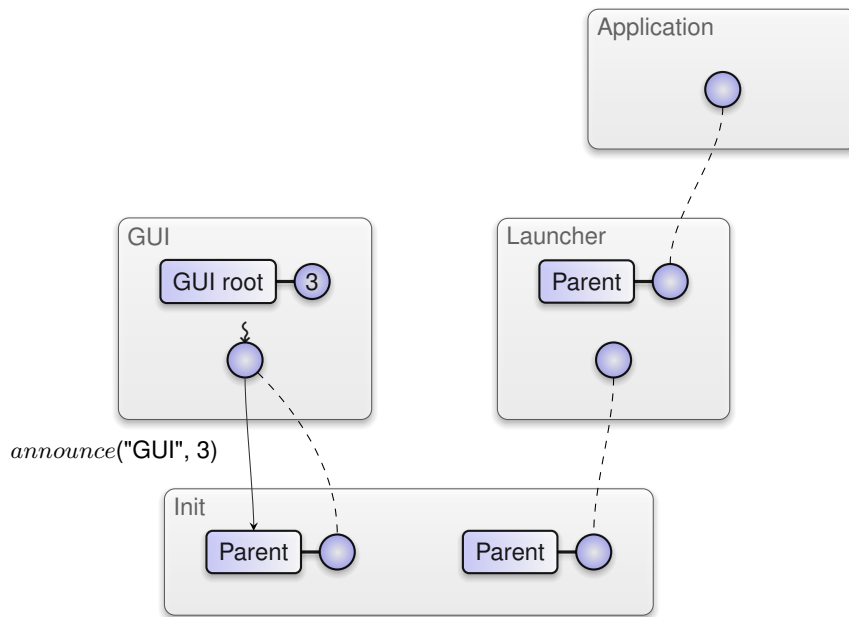
Figure 9: The GUI component announces its service to its parent using the parent interface.
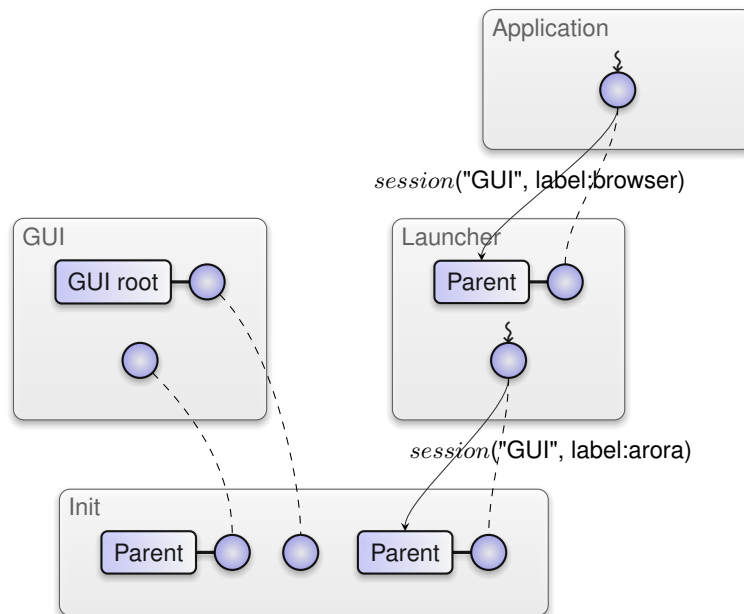
Figure 10: The application requests a GUI session using the parent interface.

46

GENODE 🖉 LABS

Forward to grandparent   The parent may decide to request a session in the name of
     its child from its own parent.

Figure 10 illustrates the latter option where the launcher responds to the application's
session request by issuing a session request to its parent, the init component. Note
that by requesting a session in the name of its child, the launcher is able to modify the
session arguments according to its policy. In the example, the launcher imposes the
use of a different label to the session. When init receives the session request from the
launcher, it is up to init to take a policy decision with the same principle options. In
fact, each component that sits in between the client and the server along the branches
of the ownership tree can impose its policy onto sessions. The routing of the session
request and the final session arguments as received by the server are the result of the
successive application of all policies along the route.

Because the GUI announced its "GUI" service beforehand, init is in possession of the
root capability, which enables it to create and destroy GUI sessions. It decides to re-
spond to the launcher's session request by triggering the GUI-session creation at the
GUI component's root interface. The GUI component responds to this request with
the creation of a new GUI session and attaches the received session arguments to the
new session. The accumulated session policy is thereby tied to the session's RPC ob-
ject. The RPC object is accompanied with its corresponding session capability, which
is delegated along the entire call chain up to the originator of the session request (Sec-
tion 3.1.2). Once the application's session request returns, the application can interact
directly with the GUI session using the session capability.

The differentiation between session creation and session use aligns two seemingly
conflicting goals with each other, namely efficiency and the application of the security
policies by potentially many components. All components on the route between client
and server are involved in the creation of the session and can thereby impose their
policies on the session. Once established, the direct communication channel between
client and server via the session capability allows for the efficient interaction between
the two components. For the actual use of the session, the intermediate components are
not on the performance-critical path.

3.2.4. Client-server relationship

Whereas the role of a component as a child is dictated by the strict ownership relation
that implies that the child has to ultimately trust its parent, the role of a component as
client or server is more diverse.

In its *role of a client* that obtained a session capability as result of a session request
from its parent, a component is unaware of the real identity of the server. It is unable
to judge the trustworthiness of the server. However, it obtained the session from its
parent, which the client ultimately trusts. Whichever session capability was handed
out by the parent, the client is not in the position to question the parent's decision.

GENODE ⬙ LABS
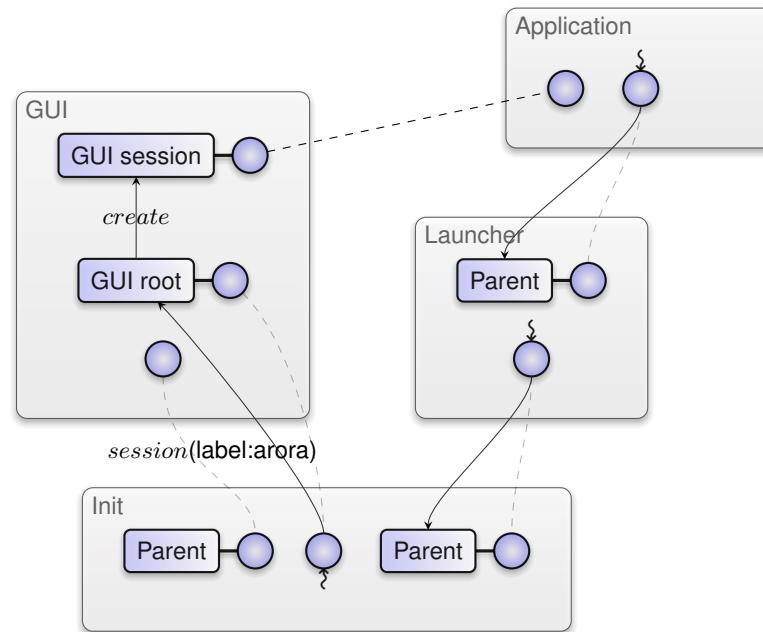
Figure 11: Session creation at the server.

Even though the integrity of the session capability can be taken for granted, the client does not need to trust the server in the same way as it trusts its parent. By invoking the capability, the client is in full control over the information it reveals to the server in the form of RPC arguments. The confidentiality and integrity of its internal state is protected. Furthermore, the invocation of a capability cannot have side effects on the client's protection domain other than the retrieval of RPC results. So the integrity of the client's internal state is protected. However, when invoking a capability, the client hands over the flow of execution to the server. The client is blocked until the server responds to the request. A misbehaving server may never respond and thereby block the client infinitely. Therefore, with respect to the liveliness of the client, the client has to trust the server. To empathize with the role of a component as a client, a capability invocation can be compared to the call of a function of an opaque 3rd-party library. When calling such a library function, the caller can never be certain to regain control. It just expects that a function returns at some point. However, in contrast to a call of a library function, a capability invocation does not put the integrity and confidentiality of the client's internal state at risk.

Servers do not trust their clients   When exercising the *role of a server*, a component should generally not trust its clients. On the contrary, from the server's perspective, clients should be expected to misbehave. This has two practical implications. First, a server is responsible for validating the arguments of incoming RPC requests. Second, a

GENODE ⫽ LABS

server should never make itself dependent on the good will of its clients. For example, a server should generally not invoke a capability obtained from one of its clients. A malicious client could have delegated a capability to a non-responding RPC object, which may block the server forever when invoked and thereby make the server unavailable for all clients. As another example, the server must always be in control over the physical memory resources used for a shared-memory interface between itself and its clients. Otherwise, if a client was in control over the used memory, it could revoke the memory from the server at any time, possibly triggering a fault at the server. The establishment of shared memory is described in detail in Section 3.6.3. Similarly to the role as client, the internal state of a server is protected from its clients with respect to integrity and confidentiality. In contrast to a client, however, the liveliness of a server is protected as well. A server never needs to wait for any response from a client. By responding to an RPC request, the server does immediately become ready to accept the next RPC request without any prior handshake with the client of the first request.

Ownership and lifetime of a session   The object identity of a session RPC object and additional RPC objects that may have been created via the session is owned by the server. So the server is in control over the lifetime of those RPC objects. The client is not in the immediate position to dictate the server when to close a session because it has no power over the server. Instead, the procedure of closing a session follows the same chain of commands as involved in the session creation. The common parent of client and server plays the role of a broker, which is trusted by both parties. From the client's perspective, closing a session is a request to its parent. The client has to accept that the response to such a request is up to the policy of the parent. The closing of a session can alternatively be initiated by all nodes of the component tree that were involved in the session creation.

From the perspective of a server that is implemented by a child, the request to close a session originates from its parent, which, as the owner of the server, represents an authority that must be ultimately obeyed. If the server complies, the object identity of the session's RPC object vanishes. Since the kernel invalidates capabilities once their associated RPC object is destroyed, all capabilities referring to the RPC object - however delegated - are implicitly revoked as a side effect. Still, a server may ignore the session-close request. In this case, the parent of a server might take steps to enforce its will by destructing the server altogether.

Trustworthiness of servers   Servers that are shared by clients of different security levels must be designed and implemented with special care. Besides the correct response to session-close requests, another consideration is the adherence to the security policy as configured by the parent. The mere fact that a server is a child of its parent does not imply that the parent won't need to trust it in some respects.

GENODE ⬚ LABS

In cases where is not viable to trust the server (e. g., because the server is based on ported software that is too complex for thorough evaluation), certain security properties such as the effectiveness of closing sessions could be enforced by a small (and thereby trustworthy) intermediate server that sits in-between the real server and the client. This intermediate server would then effectively wrap the server's session interface.

GENODE LABS

3.3. Resource trading

As introduced in Section 3.2.1, child components are created out of the resources of their respective parent components. This section describes the underlying mechanism. It first introduces the concept of RAM sessions in Section 3.3.1. Section 3.3.2 explains how RAM sessions are used to trade resources between components. The resource-trading mechanism ultimately allows servers to become resilient against client-driven resource-exhaustion attacks. However, such servers need to take special precautions that are explained in Section 3.3.3. Section 3.3.4 presents a mechanism for the dynamic balancing of resources among cooperative components.

3.3.1. Resource assignment

In general, it is the operating system's job to manage the physical resources of the machine in a way that enables multiple applications to utilize them in a safe and efficient manner. The physical resources are foremost the physical memory, the processing time of the CPUs, and devices.

The traditional approach to resource management   Traditional operating systems usually provide abstractions of physical resources to applications running on top of the operating system. For example, instead of exposing the real interface of a device to an application, a Unix kernel provides a representation of the device as a pseudo file in the virtual file system. An application interacts with the device indirectly by operating on the respective pseudo file via a device-class-specific API (ioctl operations). As another example, a traditional OS kernel provides each application with an arbitrary amount of virtual memory, which may be much larger than the available physical memory. The application's virtual memory is backed with physical memory not before the application actually uses the memory. The pretension of unlimited memory by the kernel relieves application developers from considering memory as a limited resource. On the other hand, this convenient abstraction creates problems that are extremely hard or even impossible to solve by the OS kernel.

- The amount of physical memory that is at the disposal for backing virtual memory is limited. Traditional OS kernels employ strategies to uphold the illusion of unlimited memory by swapping memory pages to disk. However, the swap space on disk is ultimately limited, too. At one point, when the physical resources are exhausted, the pretension of unlimited memory becomes a leaky abstraction and forces the kernel to take extreme decisions such as killing arbitrary processes to free up physical memory.

- Multiple applications including critical applications as well as potentially misbehaving applications share one pool of physical resources. In the presence of a

GENODE ⬚ LABS

misbehaving application that exhausts the physical memory, all applications are equally put at risk.

- Third, by granting each application the legitimate ability to consume as much memory as the application desires, applications cannot be held accountable for their consumption of physical memory. The kernel cannot distinguish a misbehaving from a well-behaving memory-demanding application.

There are several approaches to relieve those problems. For example, OS kernels that are optimized for resource utilization may employ heuristics that take the application behavior into account for parametrizing page-swapping strategies. Another example is the provisioning of a facility for pinned memory to applications. Such memory is guaranteed to be backed by physical memory. But such a facility bears the risk of allowing any application to exhaust physical memory directly. Hence, further heuristics are needed to limit the amount of pinned memory an application may use. Those counter measures and heuristics, while making the OS kernel more complex, are mere attempts to fight symptoms but unable to solve the actual problems caused by the lack of accounting. The behavior of such systems remains largely indeterministic.

As a further consequence of the abstraction from physical resources, the kernel has to entail functionality to support the abstraction. For example, for swapping memory pages to disk, the kernel has to depend on an in-kernel disk driver. For each application, whether or not it ever touches the disk, the in-kernel disk driver is part of its trusted computing base.

RAM sessions and balances   Genode does not abstract from physical resources. Instead, it solely arbitrates the access to such resources and provides means to delegate the authority over resources between components. Each low-level physical resource is represented as a dedicated service provided by the core component at the root of the component tree. The core component is described in detail in Section 3.4. The following description focuses on memory as the most prominent low-level resource managed by the operating system. Conceptually, the approach could be applied to other physical resources such as processing time, network bandwidth, disk space, or even power. However, at the time of writing, Genode employs the mechanism for memory only. Processing time is subject to the kernel's scheduling policy whereas the management of the higher-level resources such as disk space is left to the respective servers that provide those resources.

Physical memory is represented by the RAM service of core. The best way to describe the idea is to draw an analogy between the RAM service and a bank. Each RAM session corresponds to a bank account. Initially, when opening a new account, there is no balance. However, by having the authority over an existing bank account with a balance, one can transfer funds from the existing account to the new account. Naturally, such a transaction will decrease the balance of the originating account. Internally

GENODE ⬙ LABS
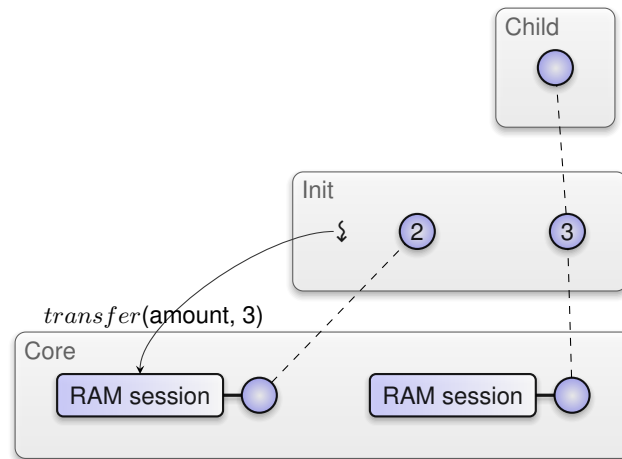
Figure 12: Init assigns a portion of its memory to a child. In addition to its own RAM session (2), init has created a second RAM session (3) designated for its child.

at the bank, the transfer does not involve any physical bank notes. The transaction is merely a change of balances of both bank accounts involved. A bank customer with the authority over a given bank account can use the value stored on the bank account to purchase physical goods while withdrawing the costs from the account. Such a withdrawal will naturally decrease the balance on the account. If the account is depleted, the bank denies the purchase attempt. Analogously to purchasing physical goods by withdrawing balances from a bank account, physical memory can be allocated from a RAM session. The balance of the RAM session is the RAM session's quota. A piece of allocated physical memory is represented by a so-called dataspace (see Section 3.4.1 for more details). A RAM dataspace is a container of physical memory that can be used for storing data.

Subdivision of budgets   Similar to a person with a bank account, each component of a Genode system has a session at core's RAM service. At boot time, the core component creates an initial RAM session with the balance set to the amount of available physical memory. This RAM session is designated for the init component, which is the first and only child of core. On request by init, core delegates the capability for this initial RAM session to the init component.

For each child component spawned by the init component, init creates a new RAM session at core. Figure 12 exemplifies this step for one child. As the result from the session creation, it obtains the capability for the new RAM session. Because it has the authority over both its own and the child's designated RAM session, it can transfer a certain amount of RAM quota from its own account to the child's account by invoking its own RAM-session capability and specifying the beneficiary's RAM-session capability as argument. Core responds to the request by atomically adjusting the quotas of
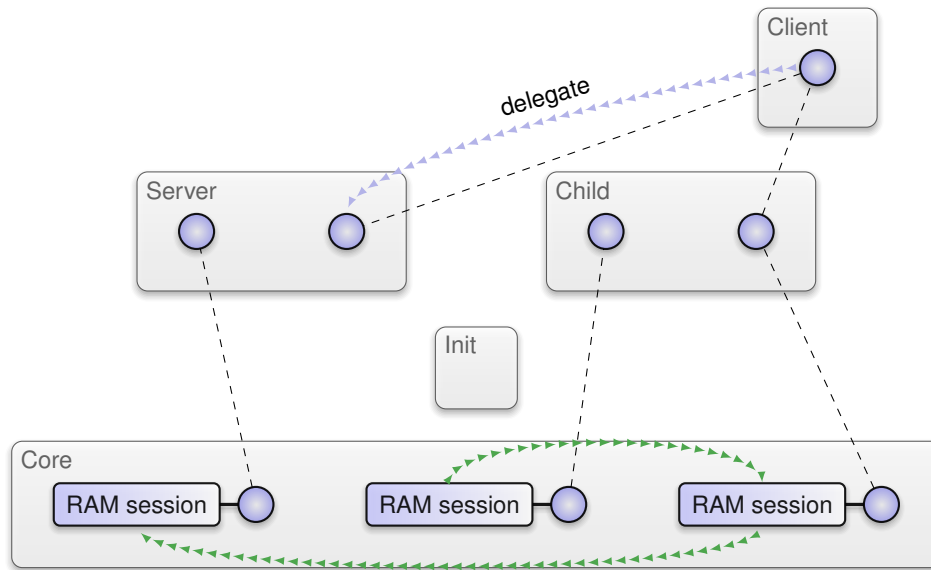
Figure 13: Memory-stealing attempt

both RAM sessions by the specified amount. In the case of init, the amount depends on init's configuration. Thereby, init explicitly splits its own RAM budget among its child components. Each child created by init can obtain the capability for its own RAM session from init via the parent interface and thereby gains the authority over the memory budget that was assigned to it. Note however, that no child has the authority over init's RAM session nor the RAM sessions of any siblings. The mechanism for distributing a given budget among multiple children works recursively. The children of init can follow the same procedure to further subdivide their budgets for spawning grandchildren.

Protection against resource stealing   A parent that created a child subsystem out of its own memory resources, expects to regain the spent resources when destructing the subsystem. For this reason, it must not be possible for a child to transfer funds to another branch of the component tree without the consent of the parent. Figure 13 illustrates an example scenario that violates this expectation. The client and server components conspire to steal memory from the child. The client was created by the child and received a portion of the child's memory budget. The client requested a session for a service that was eventually routed to the server. The client-server relationship allows the client to delegate capabilities to the server. Therefore, it is able to delegate its own RAM session capability to the server. The server, now in possession of the client's and its own RAM session capabilities, can transfer memory from the client's to its own RAM session. After this transaction, the child has no way to regain its memory

GENODE ⋙ LABS

resources because it has no authority over the server's RAM session.

To prevent such resource-stealing scenarios, Genode restricts the transfer between arbitrary RAM sessions. Each RAM session must have a reference RAM session, which can be defined only once. Transfers are permitted only between a RAM session and its reference RAM session. When creating the RAM session of a child component, the parent registers its own RAM session as the child's reference RAM session. This way, the parent becomes able to transfer budgets between its own and the child's RAM session.

RAM session destruction   When a RAM session is closed, core destroys all dataspaces that were allocated from the RAM session and transfers the RAM session's final budget to the corresponding reference RAM session.

3.3.2. Trading memory between clients and servers

An initial assignment of memory to a child is not always practical because the memory demand of a given component may be unknown at its construction time. For example, the memory needed by a GUI server over its lifetime is not known a priori but depends on the number of its clients, the number of windows on screen, or the amount of pixels that must be held at the server. In many cases, the memory usage of a server depends on the behavior of its clients. In traditional operating systems, system services like a GUI server would allocate memory on behalf of its clients. Even though the allocation was induced by a client, the server performs the allocation. The OS kernel remains unaware of the fact that the server solely needs the allocated memory for serving its client. In the presence of a misbehaving client that issues an infinite amount of requests to the server where each request triggers a server-side allocation (for example the creation of a new window), the kernel will observe the server as a resource hog. Under resource pressure, it will likely select the server to be punished. Each server that performs allocations on behalf of its clients is prone to this kind of attack. Genode solves this problem by letting clients pay for server-side allocations. Client and server may be arbitrary nodes in the component tree.

Session quotas   As described in the previous section, at the creation time of a child, the parent assigns a part of its own memory quota to the new child. Since the parent retains the RAM-session capabilities of all its children, it can issue further quota transfers back and forth between the children's RAM sessions and its own RAM session, which represents the reference account for all children. When a child requests a session at the parent interface, it can attach a fraction of its quota to the new session by specifying an amount of memory to be donated to the server as a session argument. This amount is called *session quota*. The session quota can be used by the server during the lifetime of the session. It is returned to the client when the session is closed.

When receiving a session request, the parent has to distinguish three different cases depending on its session-routing decision as described in Section 3.2.3.

GENODE 🖎 LABS

Parent provides the service   If the parent provides the requested service by itself, it first checks whether the session quota meets its need for providing the service. If so, it transfers the session quota from the requesting child's RAM session to its own RAM session. This step may fail if the child offered a session quota larger than the available quota in the child's RAM session.

Server is another child   If the parent decides to route the session request to another child, it transfers the session quota from the client's RAM session to the server's RAM session. Because the RAM sessions are not related to each other as both have the parent's RAM session as reference account, this transfer from the client to the server consists of two steps. First, the parent transfers the session quota to its own RAM session. If this step succeeded, it transfers the session quota from its own RAM session to the server's RAM session. The parent keeps track of the session quota for each session so that the quota transfers can be reverted later when closing the session. Not before the transfer of the session quota to the server's RAM session succeeded, the parent issues the actual session request at the server's root interface along with the information about the transferred session quota.

Forward to grandparent   The parent may decide to forward the session request to its own parent. In this case, the parent requests a session on behalf of its child. The grandparent neither knows nor cares about the actual origin of the request and will simply decrease the memory quota of the parent. For this reason, the parent transfers the session quota from the requesting child to its own RAM session before issuing the session request at the grandparent.

Quota transfers may fail if there is not enough budget on the originating account. In this case, the parent aborts the session creation and reflects the lack of resources as an error to the originator of the session request.

This procedure works recursively. Once the server receives the session request along with the information about the provided session quota, it can use this information to decide whether or not to provide the session under these resource conditions. It can also use the information to tailor the quality of the service according to the provided session quota. For example, a larger session quota might enable the server to use larger caches or communication buffers for the client's session.

Session upgrades   During the lifetime of a session, the initial session quota may turn out to be too scarce. Usually, the server returns such a scarcity condition as an error of operations that imply server-side allocations. The client may handle such a condition by upgrading the session quota of an existing session by issuing an upgrade request to its parent along with the targeted session capability and the additional session quota. The upgrade works analogously to the session creation. The server will receive the information about the upgrade via the root interface of the service.

Closing sessions   If a child issues a session-close request to its parent, the parent determines the corresponding server, which, depending on the route of the original session request, may be locally implemented, provided by another child, or provided by the grandparent. Once the server receives the session-close request, it is responsible for releasing all resources that were allocated from the session quota. The release of resources should revert all allocations the server has performed on behalf its client. Stressing the analogy with the bank account, the server has to sell the physical goods (i. e., RAM dataspaces) it purchased from the client's session quota to restore the balance on its RAM session. After the server has reverted all session-specific allocations, the server's RAM session is expected to have at least as much available budget as the session quota of the to-be-closed session. As a result, the session quota can be transferred back to the client.

However, a misbehaving server may fail to release those resources by malice or because of a bug. For example, the server may be unable to free a dataspace because it mistakenly used the dataspace for another client's data. Another example would be a memory leak in the server. Such misbehavior is detected on the attempt to withdraw the session quota from the server's RAM session. If the server's available RAM quota after closing a session remains lower than the session quota, the server apparently peculated memory. If the misbehaving server was locally provided by the parent, it has the full authority to not hand back the session quota to its child. If the misbehaving service was provided by the grandparent, the parent (and its whole subsystem) has to subordinate. If, however, the server was provided by another child and the child refuses to release resources, the parent's attempt to withdraw the session quota from the server's RAM session will fail. It is up to the policy of the parent to handle such a failure either by punishing the server (e. g., killing the component) or by granting more of its own quota. Generally, misbehavior is against the server's own interests. A server's best interest is to obey the parent's close request to avoid intervention.

3.3.3. Component-local heap partitioning

Components that perform memory allocations on behalf of untrusted parties must take special precautions for the component-local memory management. There are two prominent examples for such components. As discussed in Section 3.3.2, a server may be used by multiple clients that must not interfere with each other. Therefore, server-side memory allocations on behalf of a particular client must strictly be accounted to the client's session quota. Second, a parent with multiple children may need to allocate memory to perform the book keeping for the individual children, for example, maintaining the information about their open sessions and their session quotas. The parent should account those child-specific allocations to the respective children. In both cases, it is not sufficient to merely keep track of the amount of memory consumed on behalf of each untrusted party but the actual allocations must be performed on independent backing stores.

GENODE ⬆ LABS
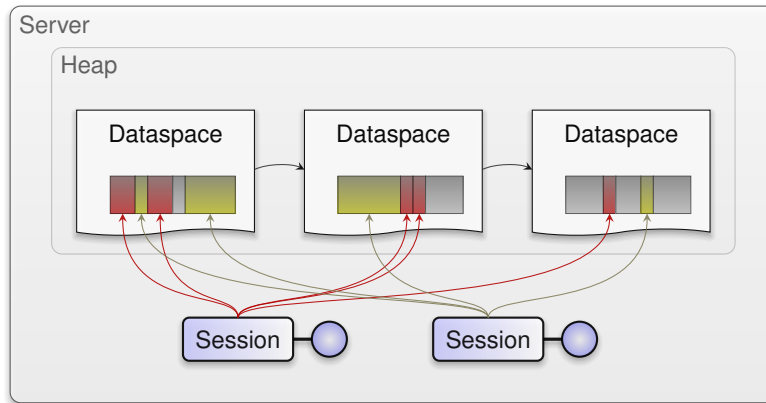
Figure 14: A server allocates anonymous memory on behalf of multiple clients from a single heap.
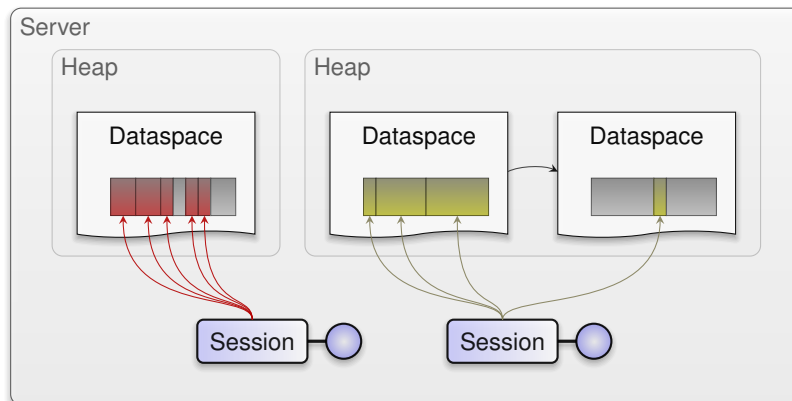
Figure 15: A server performs memory allocations from session-specific heap partitions.

Figure 14 shows a scenario where a server performs anonymous memory allocations on behalf of two session. The memory is allocated from the server's heap. Whereas allocations from the heap are of byte granularity, the heap's backing store consists of several dataspaces. Those dataspaces are allocated from the server's RAM session as needed but at a much larger granularity. As depicted in the figure, allocations from both sessions end up in the same dataspaces. This becomes a problem once one session is closed. As described in the previous section, the server's parent expects the server to release all resources that were allocated from the corresponding session quota. However, even if the server reverts all heap allocations that belong to the to-be-closed session, the server could still not release the underlying backing store because all dataspaces are still occupied with memory objects of another session. Therefore, the server becomes unable to comply with the parent's expectation.

GENODE ⚞ LABS

The solution of this problem is illustrated in Figure 15. For each session, the server maintains a separate heap partition. Each memory allocation on behalf of a client is performed from the session-specific heap partition rather than from a global heap. This way, memory objects of different sessions populate disjoint dataspaces. When closing a session, the server reverts all memory allocations from the session's heap. After freeing the session's memory objects, the heap partition becomes empty. So it can be destroyed. By destroying the heap partition, the underlying dataspaces that were used as the backing store can be properly released.

3.3.4. Dynamic resource balancing

As described in Section 3.3.1, parent components explicitly assign physical resource budgets to their children. Once assigned, the budget is at the disposal of the respective child subsystem until the subsystem gets destroyed by the parent.

However, not all components have well-defined resource demands. For example, a block cache should utilize as much memory as possible unless the memory is needed by another component. The assignment of fixed amount of memory to such a block cache cannot accommodate changes of workloads over the potentially long lifetime of the component. If dimensioned too small, there may be a lot of slack memory remaining unutilized. If dimensioned too large, the block cache would prevent other and possibly more important components to use the memory. A better alternative is to enable a component to adapt its resource use to the resource constraints of its parent. The parent interface supports this alternative with a protocol for the dynamic balancing of resources.

The resource-balancing protocol uses a combination of synchronous remote procedure calls and asynchronous notifications. Both mechanisms are described in Section 3.6. The child uses remote procedure calls to talk to its parent whereas the parent uses asynchronous notifications to signal state changes to the child. The protocol consists of two parts, which are complementary.

Resource requests By issuing a resource request to its parent, a child applies for an upgrade of its resources. The request takes the amount of desired resources as argument. A child would issue such a request if it detects scarceness of resources. A resource request returns immediately regardless of whether additional resources have been granted or not. The child may proceed working under the low resource conditions or it may block and wait for a resource-available signal from its parent. The parent may respond to this request in different ways. It may just ignore the request, possibly stalling the child. Alternatively, it may immediately transfer additional quota to the child's RAM session. Or it may take further actions to free up resources to accommodate the child. Those actions may involve long-taking operations such as the destruction of subsystems or the further propagation of resource request towards the

GENODE ⬙ LABS

root of the component tree. Once the parent has freed up enough resources to accommodate the child's request, it transfers the new resources to the child's RAM session and notifies the child by sending a resource-available signal.

Yield requests   The second part of the protocol enables the parent to express its wish for regaining resources. The parent notifies the child about this condition by sending a yield signal to the child. On the reception of such a signal, the child picks up the so-called yield request at the parent using a remote procedure call. The yield request contains the amount of resources the parent wishes to regain. It is up to the child to comply with a yield request or not. Some subsystems have meaningful ways to respond to yield requests. For example, an in-memory block cache could write back the cached information and release the memory consumed by the cache. Once the child has succeeded in freeing up resources, it reports to the parent by issuing a so-called yield response via a remote procedure call to the parent. The parent may respond to a yield response by withdrawing resources from the child's RAM session.

GENODE 🗇 LABS

3.4. Core - the root of the component tree

Core is the first user-level component, which is directly created by the kernel. It thereby represents the root of the component tree. It has access to the raw physical resources such as memory, CPUs, memory-mapped devices, interrupts, I/O ports, and boot modules. Core exposes those low-level resources as services so that they can be used by other components. For each type of resource, there exists a service in core. For example, memory resources are represented by the RAM service, interrupts are represented by the IRQ service, and CPUs are represented by the CPU service. In order to access a resource, a component has to establish a session to the corresponding service. Thereby the access to physical resources is subjected to the routing of session requests as explained in Section 3.2.3. Moreover, the resource-trading concept described in Section 3.3.2 applies to core services in the same way as for any other service.

In addition to making hardware resources available as services, core provides all prerequisites to bootstrap the component tree. These prerequisites comprise services for creating protection domains, for managing address-space layouts, and for creating object identities.

Core is almost free from policy. There are no configuration options. The only policy of core is the startup of the init process, to which core grants all available resources. Init, in turn, uses those resources to spawn further components according to its configuration.

Section 3.4.1 introduces dataspaces as containers of memory or memory-like resources. Dataspaces form the foundation for most of the core services described in the subsequent sections. The section is followed by the introduction of each individual service provided by core. In the following, a component that has established a session to such a service is called *client*. For example, a component that obtained a session to core's RAM service is a RAM client.

3.4.1. Dataspaces

A dataspace is an RPC object[1] that resides in core and represents a contiguous physical address-space region with an arbitrary size. Its base address and size are subjected to the granularity of physical pages as dictated by the memory-management unit (MMU) hardware. Typically the granularity is 4 KiB.

Dataspaces are created and managed via core's services. Because each dataspace is a distinct RPC object, the authority over the contained physical address range is represented by a capability and can thereby be delegated between components. Each component in possession of a dataspace capability can make the dataspace content visible in its local address space. Hence, by the means of delegating dataspace capabilities, components can establish shared memory.

---

[1]Interface specification in Section 8.1.3

GENODE ⬚ LABS

On Genode, only core deals with physical memory pages. All other components use dataspaces as a uniform abstraction for memory, memory-mapped I/O regions, and ROM modules.

3.4.2. Region maps

A region map[1] represents the layout of a virtual address space. The size of the virtual address space is defined at its creation time. Region maps are created implicitly as part of a PD session (Section 3.4.5) or manually via the RM service (Section 3.4.6).

Populating an address space   The concept behind region maps is a generalization of the MMU's page-table mechanism. Analogously to how a page table is populated with physical page frames, a region map is populated with dataspaces. Under the hood, core uses the MMU's page-table mechanism as a cache for region maps. The exact way of how MMU translations are installed depends on the underlying kernel and is opaque to Genode components. On most base platforms, memory mappings are established in a lazy fashion by core's page-fault resolution mechanism described in Section 7.3.3.

A region-map client in possession of a dataspace capability is able to *attach* the dataspace to the region map. Thereby the content of the dataspace becomes visible within the region map's virtual address space. When attaching a dataspace to a region map, core selects an appropriate virtual address range that is not yet populated with dataspaces. Alternatively, the client can specify a designated virtual address. It also has the option to attach a mere window of the dataspace to the region map. Furthermore, the client can specify whether the content of the dataspace should be executable or not.

The counterpart of the *attach* operation is the *detach* operation, which enables the region-map client to remove dataspaces from the region map by specifying a virtual address. Under the hood, this operation flushes the MMU mappings of the corresponding virtual address range so that the dataspace content becomes invisible.

Note that a single dataspace may be attached to any number of region maps. A dataspace may also be attached multiple times to one region map. In this case, each attach operation populates a distinct region of the virtual address space.

3.4.3. Physical memory allocation (RAM)

A RAM session[2] is a quota-bounded allocator of physical memory. At session-creation time, its quota is zero. To make the RAM session functional, it must first receive quota from another already existing RAM session, which is called the *reference account*. Once the reference account is defined, quota can be transferred back and forth between the reference account and the new RAM session.

[1]Interface specification in Section 8.4
[2]Interface specification in Section 8.5.1

GENODE ⩘ LABS

Provided that the RAM session is equipped with sufficient quota, the RAM client can allocate RAM dataspaces from the RAM session. The size of each RAM dataspace is defined by the client at the time of allocation. The location of the dataspace in physical memory is defined by core.

Each RAM dataspace is physically contiguous and can thereby be used as DMA buffer by a user-level device driver. In order to set up DMA transactions, such a device driver can request the physical address of a RAM dataspace by invoking the dataspace capability.

Closing a RAM session destroys all dataspaces allocated from the RAM session and restores the original quota. This implies that these dataspaces disappear in all components. The quota of a closed RAM session is transferred to the reference account.

The book keeping of quotas and the creation of dataspace RPC objects consumes memory within core. Core's RAM service allocates such memory from the session quota supplied by its clients. Note that this session quota is unrelated to the quota managed by the RAM session. For this reason, an allocation may fail for two different reasons. The account represented by the RAM session may be depleted. So the allocation cannot be performed unless additional quota is transferred to the account. But also, the RAM session's session quota may be depleted so that core is not able to create a new dataspace RPC object for the allocated memory. The latter condition can be resolved by upgrading the existing RAM session as detailed in Section 3.3.2.

3.4.4. Access to boot modules (ROM)

During the initial bootstrap phase of the machine, a boot loader loads the kernel's binary and additional chunks of data called *boot modules* into the physical memory. After those preparations, the boot loader passes control to the kernel. Examples of boot modules are the ELF images of the core component, the init component, the components created by init, and the configuration of the init component. Core makes each boot module available as a ROM session[1]. Because boot modules are read-only memory, they are generally called ROM modules. On session construction, the client specifies the name of the ROM module as session argument. Once created, the ROM session allows its client to obtain a ROM dataspace capability. Using this capability, the client can make the ROM module visible within its local address space. The ROM session interface is described in more detail in Section 4.5.1.

3.4.5. Protection domains (PD)

A protection domain (PD) corresponds to a unit of protection within the Genode system. Typically, there is a one-to-one relationship between a component and a PD ses-

---

[1]Interface specification in Section 8.5.2

sion[1]. Each PD consists of a virtual memory address space and a capability space (Section 3.1.1).

At the hardware-level, the CPU isolates different virtual memory address spaces via a memory-management unit. Each domain is represented by a different page directory, or an address-space ID (ASID). Genode provides an abstraction from the underlying hardware mechanism in the form of region maps as introduced in Section 3.4.2. Each PD is readily equipped with three region maps. The *address space* represents the layout of the PD's virtual memory address space, the *stack area* represents the portion of the PD's virtual address space where stacks are located, and the *linker area* is designated for dynamically linked shared objects. The stack area and linker area are attached to the address space at the component initialisation time.

The capability space is provided as a kernel mechanism. Note that not all kernels provide equally good mechanisms to implement Genode's capability model as described in Section 3.1. On kernels with support for kernel-protected object capabilities, the PD session interface allows components to create and manage kernel-protected capabilities. Initially, the PD's capability space is empty. However, the PD client can install a single capability - the parent capability - using the *assign-parent* operation at the creation time of the PD.

Core's PD service also plays the role of a broker for asynchronous notifications on kernels that lack the semantics of Genode's signalling API. The service is not used directly by components at the framework's API level but is merely an implementation artifact.

3.4.6. Region-map management (RM)

As explained in Section 3.4.5, each PD session is equipped with three region maps by default. The RM service allows components to create additional region maps manually. Such manually created region maps are also referred to as *managed dataspaces*. A managed dataspace is not backed by a range of physical addresses but its content is defined by its underlying region map. This makes region maps a generalization of nested page tables. A region-map client can obtain a dataspace capability for a given region map and use this dataspace capability in the same way as any other dataspace capability, i. e., attaching it to its local address space, or delegating it to other components.

Managed dataspaces are used in two ways. First, they allow for the manual management of portions of a component's virtual address space. For example, the so-called stack area of a protection domain is a dedicated virtual-address range preserved for stacks. Between the stacks, the virtual address space must remain empty so that stack overflows won't silently corrupt data. This is achieved by using a dedicated region map that represents the complete stack area. This region map is attached as a dataspace to the component's virtual address space. When creating a new thread along with its cor-

---

[1]Interface specification in Section 8.5.3

GENODE ⬘ LABS

responding stack, the thread's stack is not directly attached to the component's address space but to the stack area's region map. Another example is the virtual-address range managed by a dynamic linker to load shared libraries into.

The second use of managed dataspaces is the provision of on-demand-populated dataspaces. A server may hand out dataspace capabilities that are backed by region maps to its clients. Once the client has attached such a dataspace to its address space and touches it's content, the client triggers a page fault. Core responds to this page fault by blocking the client thread and delivering a notification to the server that created the managed dataspace along with the information about the fault address within the region map. The server can resolve this condition by attaching a dataspace with real backing store at the fault address, which prompts core to resume the execution of the faulted thread.

3.4.7. Processing-time allocation (CPU)

A CPU session[1] is an allocator for processing time that allows for the creation, the control, and the destruction of threads of execution. At session-construction time, the affinity of a CPU session with CPU cores can be defined via session arguments.

Once created, the session can be used to create, control, and kill threads. Each thread created via a CPU session is represented by a thread capability. The thread capability is used for subsequent thread-control operations. The most prominent thread-control operation is the *start* of the thread, which takes the thread's initial stack pointer and instruction pointer as arguments.

During the lifetime of a thread, the CPU client can retrieve and manipulate the *state* of the thread. This includes the register state as well as the execution state (whether the thread is paused or running). Those operations are primarily designated for realizing user-level debuggers.

To aid the graceful destruction of threads, the CPU client can issue a *cancel-blocking* operation, which causes the specified thread to cancel a current blocking operation such as waiting for an RPC response or the attempt to acquire a contended lock.

3.4.8. Access to device resources (IO_MEM, IO_PORT, IRQ)

Core's IO_MEM, IO_PORT, and IRQ services enable the realization of user-level device drivers as Genode components.

Memory mapped I/O (IO_MEM) An IO_MEM session[2] provides a dataspace representation for a non-memory part of the physical address space such as memory-mapped I/O regions or BIOS areas. In contrast to a memory block that is used for

---

[1]Interface specification in Section 8.5.5
[2]Interface specification in Section 8.5.6

GENODE ⩘ LABS

storing information, of which the physical location in memory is of no concern, a non-memory object has special semantics attached to its location within the physical address space. Its location is either fixed (by standard) or can be determined at runtime, for example by scanning the PCI bus for PCI resources. If the physical location of such a non-memory object is known, an IO_MEM session can be created by specifying the physical base address, the size, and the write-combining policy of the memory-mapped resource as session arguments. Once an IO_MEM session is created, the IO_MEM client can request a dataspace containing the specified physical address range.

Core hands out each physical address range only once. Session requests for ranges that intersect with physical memory are denied. Even though the granularity of memory protection is limited by the MMU page size, the IO_MEM service accepts the specification of the physical base address and size at the granularity of bytes. The rationale behind this contradiction is the unfortunate existence of platforms that host memory-mapped resources of unrelated devices on the same physical page. When driving such devices from different components, each of those components requires access to its corresponding device. So the same physical page must be handed out to multiple components. Of course, those components must be trusted to not touch any portion of the page that is unrelated to its own device.

Port I/O (IO_PORT)   For platforms that rely on I/O ports for device access, core's IO_PORT service enables the fine-grained assignment of port ranges to individual components. Each IO_PORT session[1] corresponds to the exclusive access right to a port range specified as session arguments. Core creates the new IO_PORT session only if the specified port range does not overlap with an already existing session. This ensures that each I/O port is driven by only one IO_PORT client at a time. The IO_PORT session interface resembles the physical I/O port access instructions. Reading from an I/O port can be performed via an 8-bit, 16-bit, or 32-bit access. Vice versa, there exist operations for writing to an I/O port via an 8-bit, 16-bit, or 32-bit access. The read and write operations take absolute port addresses as arguments. Core performs the I/O-port operation only if the specified port address lies within the port range of the session.

Reception of device interrupts (IRQ)   Core's IRQ service enables device-driver components to respond to device interrupts. Each IRQ session[2] corresponds to an interrupt. The physical interrupt number is specified as session argument. Each physical interrupt number can be specified by only one session. The IRQ session interface provides an operation to wait for the next interrupt. Only while the IRQ client is waiting for an interrupt, core unmasks the interrupt at the interrupt controller. Once the interrupt

---

[1]Interface specification in Section 8.5.7
[2]Interface specification in Section 8.5.8

GENODE ⪢ LABS

occurs, core wakes up the IRQ client and masks the interrupt at the interrupt controller until the driver has completed the IRQ handling and waits for the next interrupt.

3.4.9. Logging (LOG)

The LOG service is used by the lowest-level system components such as the init component for printing diagnostic output. Each LOG session[1] takes a label as session argument, which is used to prefix the output of this session. This enables developers to distinguish the output of different components with each component having a unique label. The LOG client transfers the to-be-printed characters as payload of plain RPC messages, which represents the simplest possible communication mechanism between the LOG client and core's LOG service.

3.4.10. Event tracing (TRACE)

The TRACE service provides a light-weight event-tracing facility. It is not fundamental to the architecture. However, as the service allows for the inspection and manipulation of arbitrary threads of a Genode system, TRACE sessions must not be granted to untrusted components.

---

[1]Interface specification in Section 8.5.9

GENODE 🗲 LABS
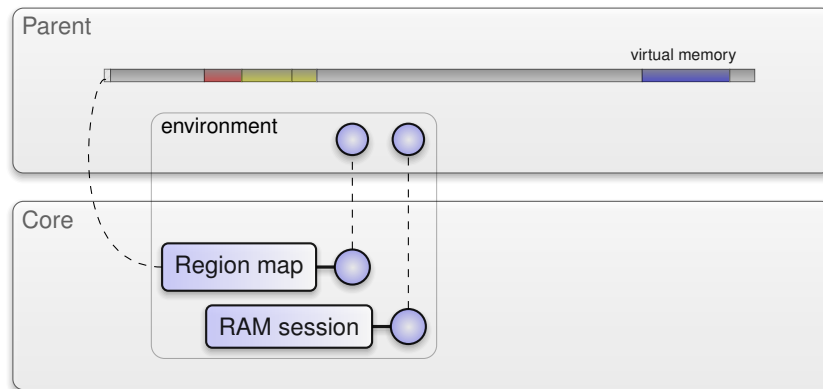
Figure 16: Starting point for creating a new component

3.5. Component creation

Each Genode component is made out of four basic ingredients:

RAM session for allocating the memory for the program's BSS segment and heap

ROM session with the executable binary

CPU session for creating the initial thread of the component

PD session representing the component's protection domain

It is the responsibility of the new component's parent to obtain those sessions. The initial situation of the parent is depicted in Figure 16. The parent's memory budget is represented by the parent's RAM (Section 3.4.3) session. The parent's virtual address space is represented by the region map contained in the parent's PD session (Section 3.4.5). Both sessions were originally created at the parent's construction time. Along with the parent's CPU session, those sessions form the parent's so-called *environment*. The address space is populated with the parent's code (shown as red), the so-called stack area that hosts the stacks (shown as blue), and presumably several RAM dataspaces for the heap, the DATA segment, and the BSS segment. Those are shown as yellow.

3.5.1. Obtaining the child's ROM and RAM sessions

The first step for creating a child component is obtaining the component's executable binary, e. g., by creating a session to a ROM service such as the one provided by core (Section 3.4.4). With the ROM session created, the parent can make the dataspace with the executable binary (i. e., an ELF binary) visible within its virtual address space by attaching the dataspace to its PD's region map. After this step, the parent is able to

GENODE ⧱ LABS
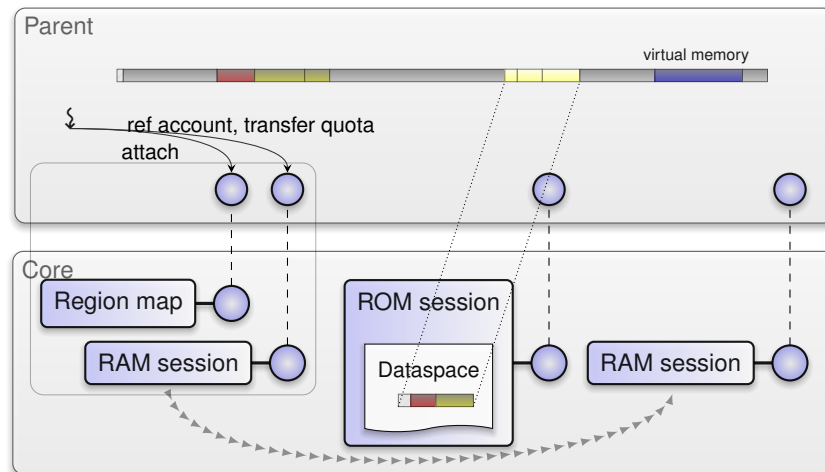
Figure 17: The parent creates the RAM session of the new child and obtains the child's executable

inspect the ELF header to determine the memory requirements for the binary's DATA and BSS segments.

The next step is the creation of the child's designated RAM session, which represents the memory budget the child will have at its disposal. The freshly created RAM session has no budget though. In order to make the RAM session usable, the parent has to transfer a portion of its own RAM quota to the child's RAM session. As explained in Section 3.3.1, the parent registers its own RAM session as the reference account for the child's RAM session in order to become able to transfer quota back and forth between both RAM sessions. Figure 17 shows the situation.

3.5.2. Constructing the child's address space

With the child's RAM session equipped with memory, the parent can construct the address space for the new child and populate it with memory allocated from the child's budget. The address-space layout is represented as a region map that is part of each PD session (Section 3.4.5). Hence, as illustrated in Figure 18, the parent has to create an PD session designated for the child. The first page of the address space is excluded such that any attempt by the child to de-reference a null pointer will cause a fault instead of silently corrupting memory. After its creation time, the child's region map is empty. It is up to the parent to populate the virtual address space with meaningful information by attaching dataspaces to the region map. The parent performs this procedure based on the information found in the ELF executable's header:

Read-only segments For each read-only segment of the ELF binary, the parent attaches the corresponding portion of the ELF dataspace to the child's address space

GENODE ⧄ LABS
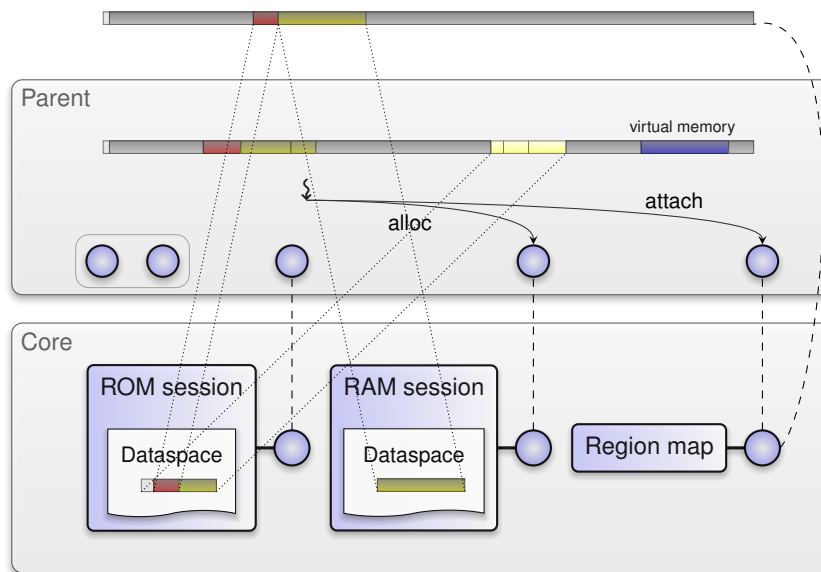
Figure 18: The parent creates and populates the virtual address space of the child using a new PD session (the parent's PD and RAM sessions are not depicted for brevity)

by invoking the attach operation on the child's region-map capability. By attaching a portion of the existing ELF dataspace to the new child's region map, no memory must be copied. If multiple instances of the same executable are created, the read-only segments of all instances refer to the same physical memory pages. If the segment contains the TEXT segment (the program code), the parent specifies a so-called executable flag to the attach operation. Core passes this flag to the respective kernel such that the corresponding page-table entries for the new components will be configured accordingly (by setting or clearing the non-executable bit in the page-table entries). Note that the propagation of this information (or the lack thereof) depends on the kernel used. Also note that not all hardware platforms distinguish executable from non-executable memory mappings.

Read-writable segments   In contrast to read-only segments, read-writable segments cannot be shared between components. Hence, each read-writable segment must be backed with a distinct copy of the segment data. The parent allocates the backing store for the copy from the child's RAM session and thereby accounts the memory consumption on behalf of the child to the child's budget. For each segment, the parent performs the following steps:

1. Allocation of a RAM dataspace from the child's RAM session. The size of the dataspace corresponds to the segment's memory size. The memory size may be higher than the size of the segment in the ELF binary (named file size). In particular, if the segment contains a DATA section followed by a BSS

GENODE ⬛ LABS

section, the file size corresponds to the size of the DATA section whereby the memory size corresponds to the sum of both sections. Core's RAM service ensures that each freshly allocated RAM dataspace is guaranteed to contain zeros. Core's RAM service returns a RAM dataspace capability as the result of the allocation operation.

2. Attachment of the RAM dataspace to the parent's virtual address space by invoking the attach operation on the parent's region map with the RAM dataspace capability as argument.

3. Copying of the segment content from the ELF binary's dataspace to the freshly allocated RAM dataspace. If the memory size of the segment is larger than the file size, no special precautions are needed as the remainder of the RAM dataspace is known to be initialized with zeros.

4. After filling the content of the segment dataspace, the parent no longer needs to access it. It can remove it from its virtual address space by invoking the detach operation on its own region map.

5. Based on the virtual segment address as found in the ELF header, the parent attaches the RAM dataspace to the child's virtual address space by invoking the attach operation on the child PD's region map with the RAM dataspace as argument.

This procedure is repeated for each segment. Note that although the above description refers to ELF executables, the underlying mechanisms used to load the executable binary are file-format agnostic.

3.5.3. Creating the initial thread

With the virtual address space of the child configured, it is time to create the component's initial thread. Analogously to the child's RAM and PD sessions, the parent creates a CPU session (Section 3.4.7) for the child. The parent may use session arguments to constrain the scheduling parameters (i. e., the priority) and the CPU affinity of the new child. Whichever session arguments are specified, the child's abilities will never exceed the parent's abilities. I.e., the child's priority is subjected to the parent's priority constraints. Once constructed, the CPU session can be used to create new threads by invoking the session's create-thread operation with the thread's designated PD as argument. Based on this association of the thread with its PD, core is able to respond to page faults triggered by the thread. The invocation of this operation results in a thread capability, which can be used to control the execution of the thread. Immediately after its creation, the thread remains inactive. In order to be executable, it first needs to be configured.

As described in Section 3.2.1, each PD has initially a single capability installed, which allows the child to communicate with its parent. Right after the creation of the PD for

GENODE ⬚ LABS
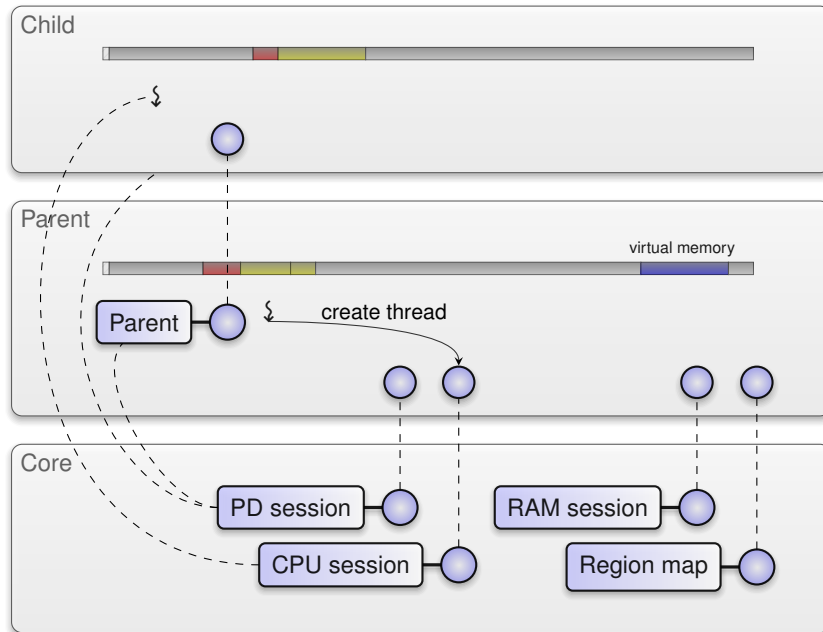
Figure 19: Creation of the child's initial thread

a new child, the parent can register a capability to a locally implemented RPC object as parent capability for the PD session. Now that the child's PD is equipped with an initial thread and a communication channel to its parent, it is the right time to kick off the execution of the thread by invoking the start operation on its thread capability. The start operation takes the initial program counter as argument, which corresponds to the program's entry-point address as found in the ELF header of the child's executable binary. Figure 19 illustrates the relationship between the PD session, the CPU session, and the parent capability. Note that neither the ROM dataspace containing the ELF binary nor the RAM dataspaces allocated during the ELF loading are visible in the parent's virtual address space any longer. After the initial loading of the ELF binary, the parent has detached those dataspaces from its own region map.

The child starts its execution at the virtual address defined by the ELF entrypoint. For regular Genode components, it points to a short assembly routine that sets up the initial stack and calls the low-level C++ startup code. This code, in turn, initializes the C++ runtime (such as the exception handling) along with the component's local Genode environment. The environment is constructed by successively requesting the component's RAM, CPU, and PD sessions from its parent. With the Genode environment in place, the startup code initializes the stack area, sets up the real stack for the initial thread within the stack area, and returns to the assembly startup code. The assembly code, in turn, switches the stack from the initial stack to the real stack and calls the program-specific C++ startup code. This code initializes the component's initial en-

GENODE ⍊ LABS

trypoint and executes all global constructors before calling the component's `construct` function. Section 7.1 describes the component-local startup procedure in detail.

73

GENODE LABS

3.6. Inter-component communication

Genode provides three principle mechanisms for inter-component communication, namely synchronous remote procedure calls (RPC), asynchronous notifications, and shared memory. Section 3.6.1 describes synchronous RPC as the most prominent one. In addition to transferring information across component boundaries, the RPC mechanism provides the means for delegating capabilities and thereby authority throughout the system.

The RPC mechanism closely resembles the semantics of a function call where the control is transferred from the caller to the callee until the function returns. As discussed in Section 3.2.4, there are situations where the provider of information does not wish to depend on the recipient to return control. Such situations are addressed by the means of an asynchronous notification mechanism explained in Section 3.6.2.

Neither synchronous RPC nor asynchronous notifications are suitable for transferring large bulks of information between components. RPC messages are strictly bound to a small size and asynchronous notifications do not carry any payload at all. This is where shared memory comes into play. By sharing memory between components, large bulks of information can be propagated without the active participation of the kernel. Section 3.6.3 explains the procedure of establishing shared memory between components.

Each of the three basic mechanisms is rarely found in isolation. Most inter-component interactions are a combination of these mechanisms. Section 3.6.4 introduces a pattern for propagating state information by combining asynchronous notifications with RPC. Section 3.6.5 shows how synchronous RPC can be combined with shared memory to transfer large bulks of information in a synchronous way. Section 3.6.6 combines asynchronous notifications with shared memory to largely decouple producers and consumers of high-throughput data streams.

GENODE ⬚ LABS
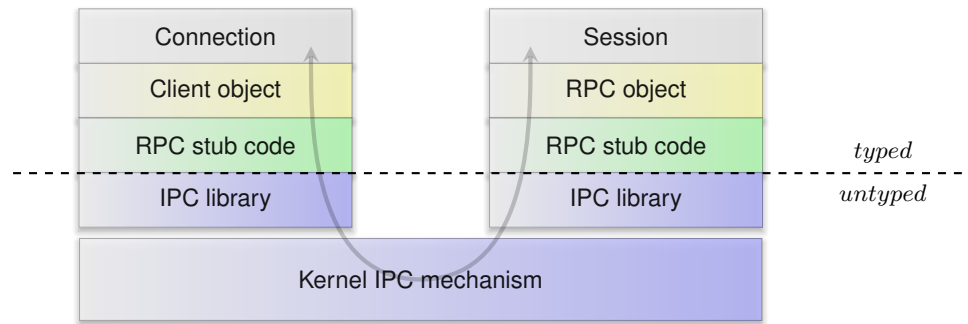
Figure 20: Layered architecture of the RPC mechanism

3.6.1. Synchronous remote procedure calls (RPC)

Section 3.1.3 introduced remote procedure calls (RPC) as Genode's fundamental mechanism to delegate authority between components. It introduced the terminology for RPC objects, capabilities, object identities, and entrypoints. It also outlined the flow of control between a client, the kernel, and a server during an RPC call. This section complements Section 3.1.3 with the information of how the mechanism presents itself at the C++ language level. It first introduces the layered structure of the RPC mechanism and the notion of *typed capabilities*. After presenting the class structure of an RPC server, it shows how those classes interact when RPC objects are created and called.

Typed capabilities   Figure 20 depicts the software layers of the RPC mechanism.

Kernel inter-process-communication (IPC) mechanism   At the lowest level, the kernel's IPC mechanism is used to transfer messages back and forth between client and server. The actual mechanism largely differs between the various kernels supported by Genode. Chapter 7 gives insights into the functioning of the IPC mechanism as used on specific kernels. Genode's capability-based security model is based on the presumption that the kernel protects *object identities* as kernel objects, allows user-level components to refer to kernel objects via capabilities, and supports the delegation of capabilities between components using the kernel's IPC mechanism. At the kernel-interface level, the kernel is not aware of language semantics like the C++ type system. From the kernel's point of view, an object identity merely exists and can be referred to, but has no type.

IPC library   The IPC library introduces a kernel-independent programming interface that is needed to implement the principle semantics of clients and servers. For each kernel supported by Genode, there exists a distinct IPC library that uses the respective kernel mechanism. The IPC library introduces the notions of untyped capabilities, message buffers, IPC clients, and IPC servers.

GENODE ⫦ LABS

An *untyped capability* is the representation of a Genode capability at the C++ language level. It consists of the local name of the referred-to object identity as well as a means to manage the lifetime of the capability, i. e., a reference counter. The exact representation of an untyped capability depends on the kernel used.

A *message buffer* is a statically sized buffer that carries the payload of an IPC message. It distinguishes two types of payload, namely raw data and capabilities. Payloads of both kinds can be simultaneously present. A message buffer can carry up to 1 KiB of raw data and up to four capabilities. Prior to issuing the kernel IPC operation, the IPC library translates the message-buffer content to the format understood by the kernel's IPC operation.

The client side of the communication channel executes an *IPC call* operation with a destination capability, a send buffer, and a receive buffer as arguments. The send buffer contains the RPC function arguments, which can comprise plain data as well as capabilities. The IPC library transfers these arguments to the server via a platform-specific kernel operation and waits for the server's response. The response is returned to the caller as new content of the receive buffer.

At the server side of the communication channel, an entrypoint thread executes the *IPC reply* and *IPC reply-and-wait* operations to interact with potentially many clients. Analogously to the client, it uses two message buffers, a receive buffer for incoming requests and a send buffer for delivering the reply of the last request. For each entrypoint, there exists an associated untyped capability that is created with the entrypoint. This capability and can be combined with an IPC client object to perform calls to the server. The *IPC reply-and-wait* operation delivers the content of the reply buffer to the last caller and then waits for a new request using a platform-specific kernel operation. Once unblocked by the kernel, it returns the arguments for the new request in the request buffer. The server does not obtain any form of client identification along with an incoming message that could be used to implement server-side access-control policies. Instead of performing access control based on a client identification in the server, access control is solely performed by the kernel on the invocation of capabilities. If a request was delivered to the server, the client has – by definition – a capability for communicating with the server and thereby the authority to perform the request.

RPC stub code   The RPC stub code complements the IPC library with the semantics of RPC interfaces and RPC functions. An RPC interface is an abstract C++ class with the declarations of the functions callable by RPC clients. Thereby each RPC interface is represented as a C++ type. The declarations are accompanied with annotations that allow the C++ compiler to generate the so-called RPC stub code on both the client side and server side. Genode uses C++ templates to generate the stub code, which avoids the crossing of a language barrier when designing RPC interfaces and alleviates the need for code-generating tools in addition to

GENODE ⧄ LABS

the compiler.

The client-side stub code translates C++ method calls to IPC-library operations. Each RPC function of an RPC interface has an associated opcode (according to the order of RPC functions). This opcode along with the method arguments are inserted into the IPC client's send buffer. Vice versa, the stub code translates the content of the IPC client's receive buffer to return values of the method invocation.

The server-side stub code implements the so-called dispatch function, which takes the IPC server's receive buffer, translates the message into a proper C++ method call, calls the corresponding server-side function of the RPC interface, and translates the function results into the IPC server's send buffer.

RPC object and client object   Thanks to the RPC stub code, the server-side implementation of an RPC object comes down to the implementation of the abstract interface of the corresponding RPC interface. When an RPC object is associated with an entrypoint, the entrypoint creates a unique capability for the given RPC object. RPC objects are typed with their corresponding RPC interface. This C++ type information is propagated to the corresponding capabilities. For example, when associating an RPC object that implements the LOG-session interface with an entrypoint, the resulting capability is a LOG-session capability.

This capability represents the authority to invoke the functions of the RPC object. On the client side, the client object plays the role of a proxy of the RPC object within the client's component. Thereby, the client becomes able to interact with the RPC object in a natural manner.

Sessions and connections   Section 3.2.3 introduced sessions between client and server components as the basic building blocks of system composition. At the server side each session is represented by an RPC object that implements the session interface. At the client side, an open session is represented by a *connection* object. The connection object encapsulates the session arguments and also represents a client object to interact with the session.

As depicted in Figure 20, capabilities are associated with types on all levels above the IPC library. Because the IPC library is solely used by the RPC stub code but not at the framework's API level, capabilities appear as being C++ type safe, even across component boundaries. Each RPC interface implicitly defines a corresponding capability type. Figure 21 shows the inheritance graph of Genode's most fundamental capability types.

Server-side class structure   Figure 22 gives on overview of the C++ classes that are involved at the server side of the RPC mechanism. As described in Section 3.1.3, each entrypoint maintains a so-called object pool. The object pool contains references to RPC objects associated with the entrypoint. When receiving an RPC request along with

GENODE ⟋ LABS
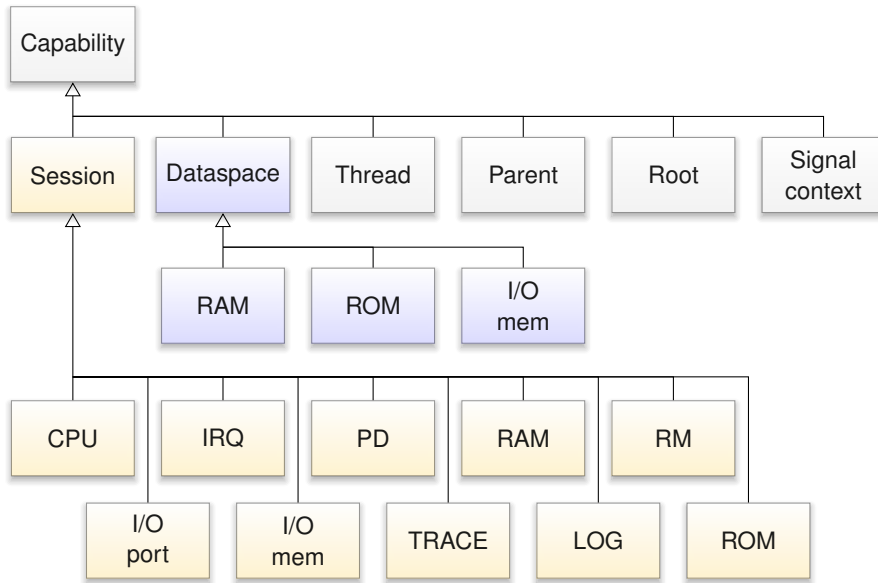
Figure 21: Fundamental capability types
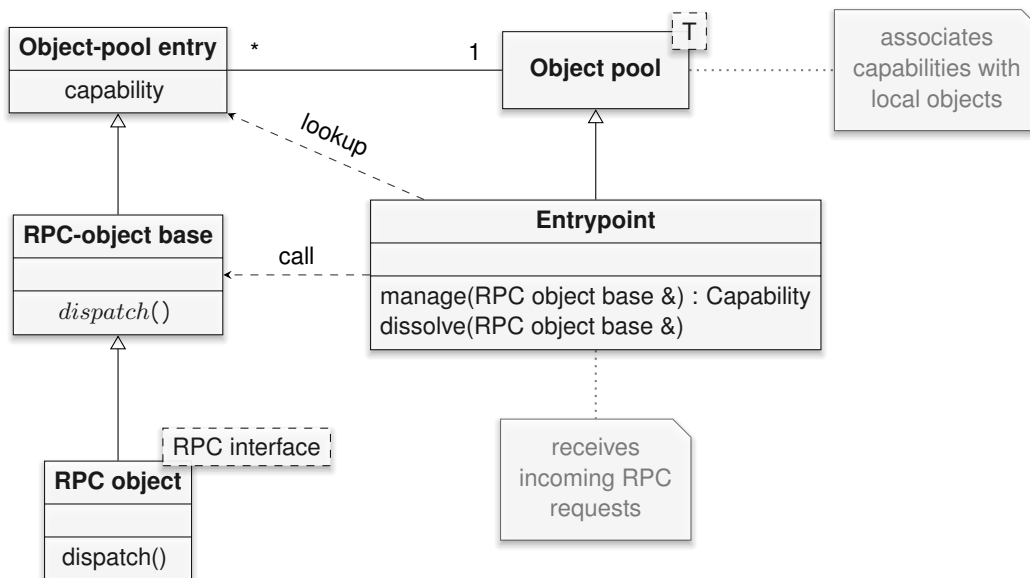
Figure 22: Server-side structure of the RPC mechanism

GENODE LABS

the local name of the invoked object identity, the entrypoint uses the object pool to lookup the corresponding RPC object. As seen in the figure, the RPC object is a class template parametrized with its RPC interface. When instantiated, the dispatch function is generated by the C++ compiler according to the RPC interface.

GENODE LABS
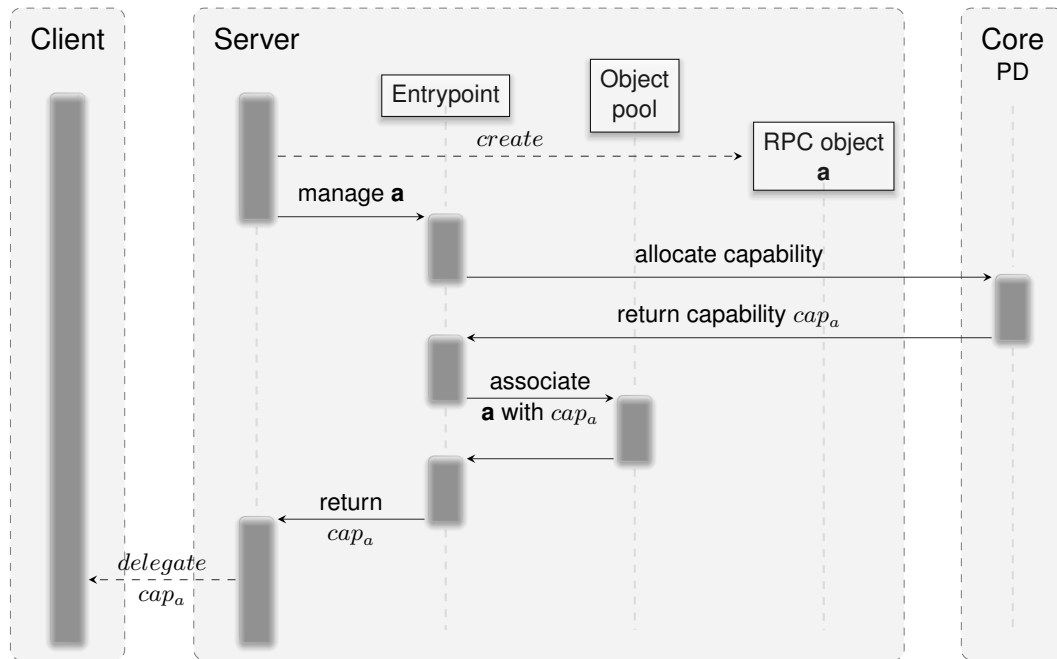
Figure 23: Creation of a new RPC object

RPC-object creation Figure 23 shows the procedure of creating a new RPC object. The server component has already created an entrypoint, which, in turn, created its corresponding object pool.

1. The server component creates an instance of an RPC object. "RPC object" denotes an object that inherits the RPC object class template typed with the RPC interface and that implements the virtual functions of this interface. By inheriting the RPC object class template, it gets equipped with a dispatch function for the given RPC interface.

   Note that a single entrypoint can be used to manage any number of RPC objects of arbitrary types.

2. The server component associates the RPC object with the entrypoint by calling the entrypoint's *manage* function with the RPC object as argument. The entrypoint responds to this call by allocating a new object identity using a session to core's PD service (Section 3.4.5). For allocating the new object identity, the entrypoint specifies the untyped capability of its IPC server as argument. Core's PD service responds to the request by instructing the kernel to create a new object identity associated with the untyped capability. Thereby, the kernel creates a new capability that is derived from the untyped capability. When invoked, the derived capability refers to the same IPC server as the original untyped capability. But it

GENODE ⦚ LABS
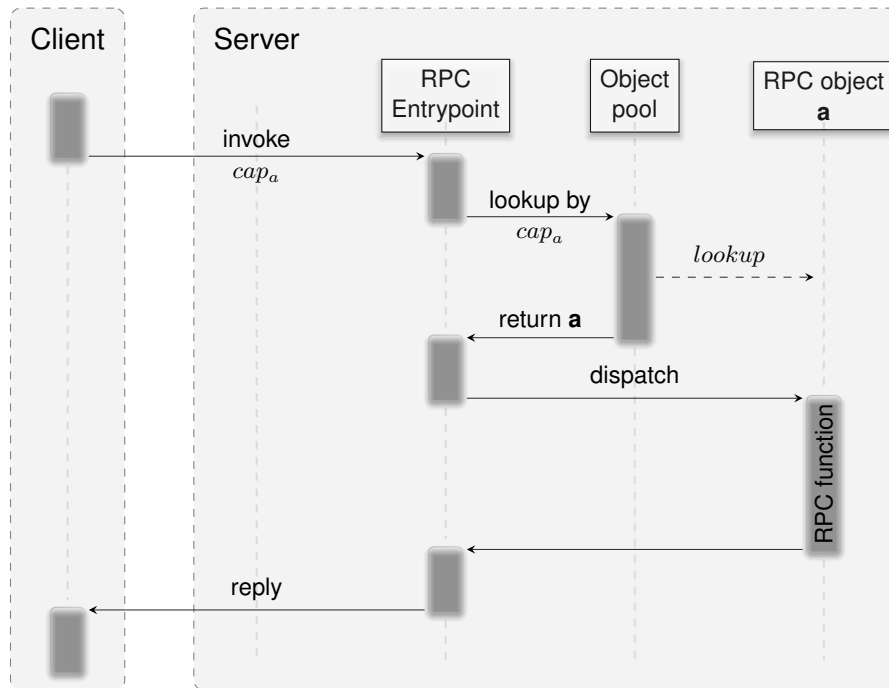
Figure 24: Invocation of an RPC object

represents a distinct object identity. The IPC server retrieves the local name of this object identity when called via the derived capability. The entrypoint stores the association of the derived capability with the RPC object in the object pool.

3. The entrypoint hands out the derived capability as return value of the manage function. At this step, the derived capability is converted into a typed capability with its type corresponding to the type of the RPC object that was specified as argument. This way, the link between the types of the RPC object and the corresponding capability is preserved at the C++ language level.

4. The server delegates the capability to another component, e. g., as payload of a remote procedure call. At this point, the client receives the authority to call the RPC object.

RPC-object invocation   Figure 24 shows the flow of execution when a client calls an RPC object by invoking a capability.

1. The client invokes the given capability using an instance of an RPC client object, which uses the IPC library to invoke the kernel's IPC mechanism. The kernel delivers the request to the IPC server that belongs to the invoked capability and

GENODE ⩘ LABS

wakes up the corresponding entrypoint. On reception of the request, the entry-
point obtains the local name of the invoked object identity.

2. The entrypoint uses the local name of the invoked object identity as a key into its
   object pool to look up the matching RPC object. If the lookup fails, the entrypoint
   replies with an error.

3. If the matching RPC object was found, the entrypoint calls the RPC object's dis-
   patch method. This method is implemented by the server-side stub code. It con-
   verts the content of the receive buffer of the IPC server to a method call. I.e., it
   obtains the opcode of the RPC function from the receive buffer to decide which
   method to call, and supplies the arguments according to the definition in the RPC
   interface.

4. On the return of the RPC function, the RPC stub code populates the send buffer
   of the IPC server with the function results and invokes the kernel's reply opera-
   tion via the IPC library. Thereby, the entrypoint becomes ready to serve the next
   request.

5. When delivering the reply to the client, the kernel resumes the execution of the
   client, which can pick up the results of the RPC call.

GENODE ⋙ LABS

3.6.2. Asynchronous notifications

The synchronous RPC mechanism described in the previous section is not sufficient to cover all forms of inter-component interactions. It shows its limitations in the following situations.

Waiting for multiple conditions

In principle, the RPC mechanism can be used by an RPC client to block for a condition at a server. For example, a timer server could provide a blocking sleep function that, when called by a client, blocks the client for a certain amount of time. However, if the client wanted to respond to multiple conditions such as a timeout, incoming user input, and network activity, it would need to spawn one thread for each condition where each thread would block for a different condition. If one condition triggers, the respective thread would resume its execution and respond to the condition. However, because all threads could potentially be woken up independently from each other – as their execution depends only on their respective condition – they need to synchronize access to shared state. Consequently, components that need to respond to multiple conditions would not only waste threads but also suffer from synchronization overhead.

At the server side, the approach of blocking RPC calls is equally bad in the presence of multiple clients. For example, a timer service with the above outlined blocking interface would need to spawn one thread per client.

Signaling events to untrusted parties

With merely synchronous RPC, a server cannot deliver sporadic events to its clients. If the server wanted to inform one of its clients about such an event, it would need to act as a client itself by performing an RPC call to its own client. However, by performing an RPC call, the caller passes the control of execution to the callee. In the case of a server that serves multiple clients, it would put the availability of the server at the discretion of all its clients, which is unacceptable.

A similar situation is the interplay between a parent and a child where the parent does not trust its child but still wishes to propagate sporadic events to the child.

The solution to those problems is the use of asynchronous notifications, also named signals. Figure 25 shows the interplay between two components. The component labeled as signal handler responds to potentially many external conditions propagated as signals. The component labeled as signal producer triggers a condition. Note that both can be arbitrary components.

GENODE ⬙ LABS
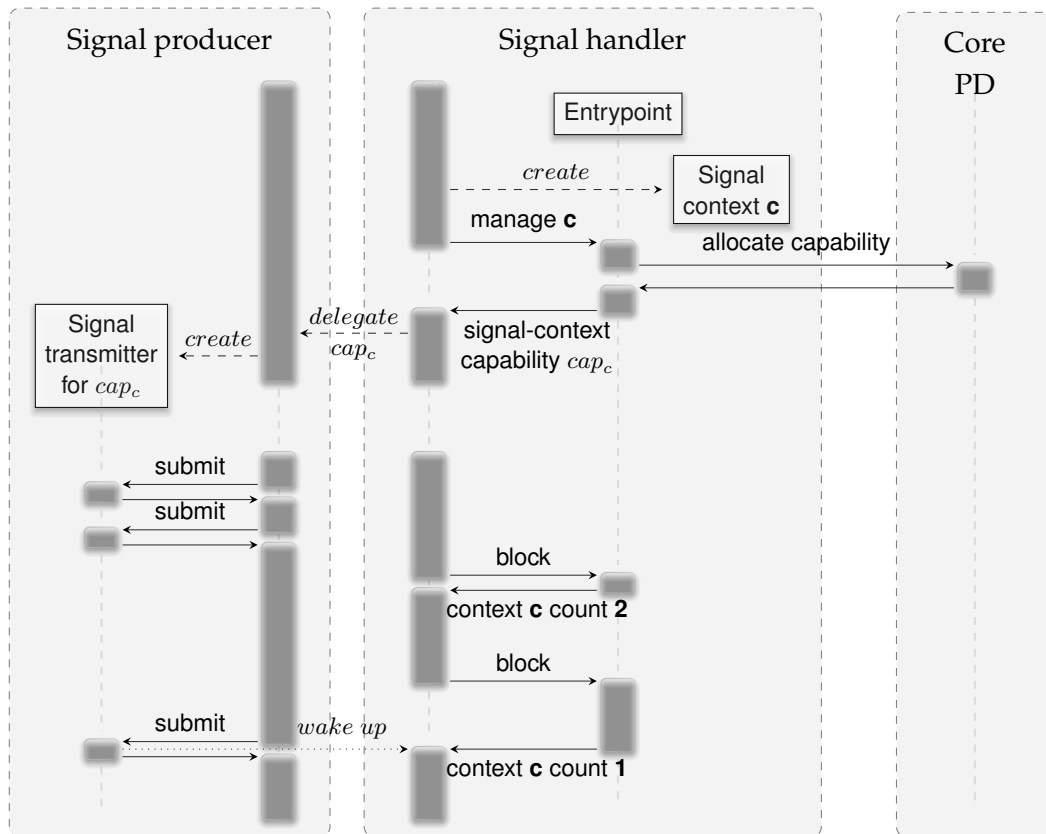
Figure 25: Interplay between signal producer and signal handler

Signal-context creation and delegation   The upper part of Figure 25 depicts the steps needed by a signal handler to become able to receive asynchronous notifications.

1. Each Genode component is equipped with at least one initial entrypoint that responds to incoming RPC requests as well as asynchronous notifications. Similar to how it can handle requests for an arbitrary number of RPC objects, it can receive signals from many different sources. Within the signal-handler component, each source is represented as a so-called *signal context*. A component that needs to respond to multiple conditions creates one signal context for each condition. In the figure, a signal context "c" is created.

2. The signal-handler component associates the signal context with its entrypoint via the manage method. Analogous to the way how RPC objects are associated with entrypoints, the manage method returns a capability for the signal context. Under the hood, the entrypoint uses core's PD service to create this kind of capability.

GENODE ⚑ LABS

3. As for regular capabilities, a signal-context capability can be delegated to other components. Thereby, the authority to trigger signals for the associated context is delegated.

Triggering signals The lower part of Figure 25 illustrates the use of a signal-context capability by the signal producer.

1. Now in possession of the signal-context capability, the signal producer creates a so-called *signal transmitter* for the capability. The signal transmitter can be used to trigger a signal by calling the *submit* method. This method returns immediately. In contrast to a remote procedure call, the submission of a signal is a fire-and-forget operation.

2. At the time when the signal producer submitted the first signal, the signal handler is not yet ready to handle them. It is still busy with other things. Once the signal handler becomes ready to receive a new signal, the pending signal is delivered, which triggers the execution of the corresponding signal-handler method. Note that signals are not buffered. If signals are triggered at a high rate, multiple signals may result in only a single execution of the signal handler. For this reason, the handler cannot infer the number of events from the number of signal-handler invocations. In situations where such information is needed, the signal handler must retrieve it via another mechanism such as an RPC call to query the most current status of the server that produced the signals.

3. After handling the first batch of signals, the signal handler component blocks and becomes ready for another signal or RPC request. This time, no signals are immediately pending. After a while, however, the signal producer submits another signal, which eventually triggers another execution of the signal handler.

In contrast to remote procedure calls, signals carry no payload. If signals carried any payload, this payload would need to be buffered somewhere. Regardless of where this information is buffered, the buffer could overrun if signals are submitted at a higher rate than handled. There might be two approaches to deal with this situation. The first option would be to drop the payload once the buffer overruns, which would make the mechanism indeterministic, which is hardly desirable. The second option would be to sacrifice the fire-and-forget semantics at the producer side, blocking the producer when the buffer is full. However, this approach would put the liveliness of the producer at the whim of the signal handler. Consequently, signals are void of any payload.

GENODE ⚞ LABS
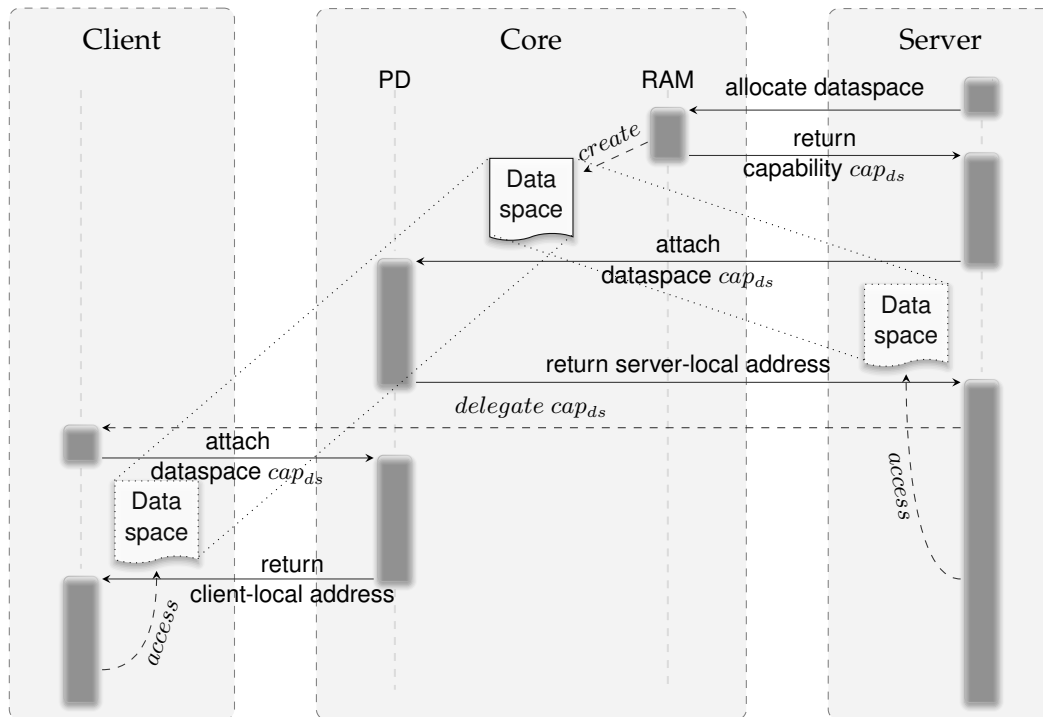
Figure 26: Establishing shared memory between client and server. The server interacts with core's RAM service. Both client and server interact with the region maps of their respective PD sessions at core.

3.6.3. Shared memory

By sharing memory between components, large amounts of information can be propagated across protection-domain boundaries without the active involvement of the kernel.

Sharing memory between components raises a number of questions. First, Section 3.3 explained that physical memory resources must be explicitly assigned to components either by their respective parents or by the means of resource trading. This raises the question of which component is bound to pay for the memory shared between multiple components. Second, unlike traditional operating systems where different programs can refer to globally visible files and thereby establish shared memory by mapping a prior-agreed file into their respective virtual memory spaces, Genode does not have a global name space. How do components refer to the to-be-shared piece of memory? Figure 26 answers these questions showing the sequence of shared-memory establishment between a server and its client. The diagram depicts a client, core, and a server. The notion of a client-server relationship is intrinsic for the shared-memory mechanism. When establishing shared memory between components, the component's roles as client and server must be clearly defined.

GENODE ⟋ LABS

1. The server interacts with core's RAM service to allocate a new RAM dataspace. Because the server uses its own RAM session for that allocation, the dataspace is paid for by the server. At first glance, this seems contradictory to the principle that clients should have to pay for using services as discussed in Section 3.3.2. However, this is not the case. By establishing the client-server relationship, the client has transferred a budget of RAM to the server via the session-quota mechanism. So the client already paid for the memory. Still, it is the server's responsibility to limit the size of the allocation to the client's session quota.

   Because the server allocates the dataspace, it is the owner of the dataspace. Hence, the lifetime of the dataspace is controlled by the server.

   Core's RAM service returns a dataspace capability as the result of the allocation.

2. The server makes the content of the dataspace visible in its virtual address space by attaching the dataspace within the region map of its PD session. The server refers to the dataspace via the dataspace capability as returned from the prior allocation. When attaching the dataspace to the server's region map, core's PD service maps the dataspace content at a suitable virtual-address range that is not occupied with existing mappings and returns the base address of the occupied range to the server. Using this base address and the known dataspace size, the server can safely access the dataspace content by reading from or writing to its virtual memory.

3. The server delegates the authority to use the dataspace to the client. This delegation can happen in different ways, e. g., the client could request the dataspace capability via an RPC function at the server. But the delegation could also involve further components that transitively delegate the dataspace capability. Therefore, the delegation operation is depicted as a dashed line.

4. Once the client has obtained the dataspace capability, it can use the region map of its own PD session to make the dataspace content visible in its address space. Note that even though both client and server use core's PD service, each component uses a different session. Analogous to the server, the client receives a client-local address within its virtual address space as the result of the attach operation.

5. After the client has attached the dataspace within its region map, both client and server can access the shared memory using their respective virtual addresses.

In contrast to the server, the client is not in control over the lifetime of the dataspace. In principle, the server, as the owner of the dataspace, could free the dataspace at its RAM session at any time and thereby revoke the corresponding memory mappings in all components that attached the dataspace. The client has to trust the server with respect to its liveliness, which is consistent with the discussion in Section 3.2.4. A well-behaving server should tie the lifetime of a shared-memory dataspace to the lifetime

GENODE ⋟ LABS

of the client session. When the server frees the dataspace at its RAM session, core implicitly detaches the dataspace from all region maps. Thereby the dataspace will become inaccessible to the client.

3.6.4. Asynchronous state propagation

In many cases, the mere information that a signal occurred is insufficient to handle the signal in a meaningful manner. For example, a component that registers a timeout handler at a timer server will eventually receive a timeout. But in order to handle the timeout properly, it needs to know the actual time. The time could not be delivered along with the timeout because signals cannot carry any payload. But the timeout handler may issue a subsequent RPC call to the timer server for requesting the time.

Another example of this combination of asynchronous notifications and remote procedure calls is the resource-balancing protocol described in Section 3.3.4.

3.6.5. Synchronous bulk transfer

The synchronous RPC mechanism described in Section 3.6.1 enables components to exchange information via a kernel operation. In contrast to shared memory, the kernel plays an active role by copying information (and delegating capabilities) between the communication partners. Most kernels impose a restriction onto the maximum message size. To comply with all kernels supported by Genode, RPC messages must not exceed a size of 1 KiB. In principle, larger payloads could be transferred as a sequence of RPCs. But since each RPC implies the costs of two context switches, this approach is not suitable for transferring large bulks of data. But by combining synchronous RPC with shared memory, these costs can be mitigated.

Figure 27 shows the procedure of transferring large bulk data using shared memory as a communication buffer while using synchronous RPCs for arbitrating the use of the buffer. The upper half of the figure depicts the setup phase that needs to be performed only once. The lower half exemplifies an operation where the client transfers a large amount of data to the server, which processes the data before transferring a large amount of data back to the client.

1. At session-creation time, the server allocates the dataspace, which represents the designated communication buffer. The steps resemble those described in Section 3.6.3. The server uses session quota provided by the client for the allocation. This way, the client is able to aid the dimensioning of the dataspace by supplying an appropriate amount of session quota to the server. Since the server performed the allocation, the server is in control of the lifetime of the dataspace.

2. After the client established a session to the server, it initially queries the dataspace capability from the server using a synchronous RPC and attaches the dataspace
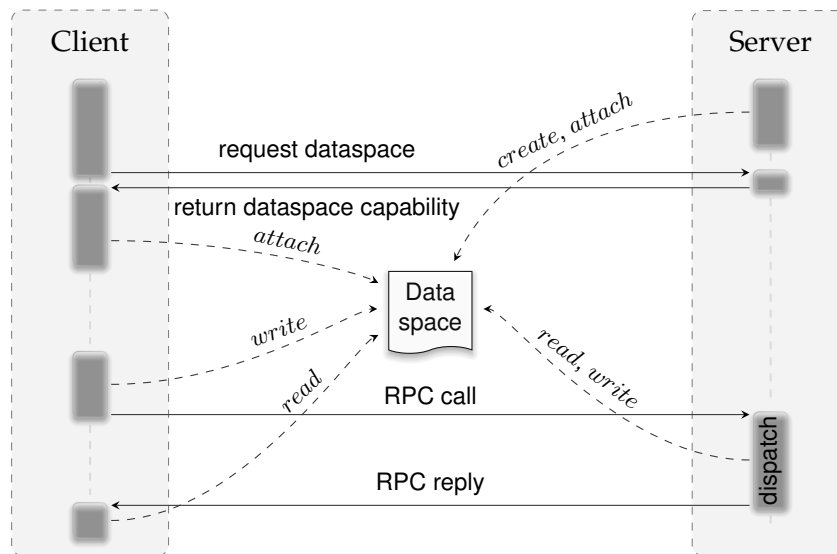
Figure 27: Transferring bulk data by combining synchronous RPC with shared memory

to its own address space. After this step, both client and server can read and write the shared communication buffer.

3. Initially the client plays the role of the user of the dataspace. The client writes the bulk data into the dataspace. Naturally, the maximum amount of data is limited by the dataspace size.

4. The client performs an RPC call to the server. Thereby, it hands over the role of the dataspace user to the server. Note that this handover is not enforced. The client's PD retains the right to access the dataspace, i. e., by another thread running in the same PD.

5. On reception of the RPC, the server becomes active. It reads and processes the bulk data, and writes its results to the dataspace. The server must not assume to be the exclusive user of the dataspace. A misbehaving client may change the buffer content at any time. Therefore, the server must take appropriate precautions. In particular, if the data must be validated at the server side, the server must copy the data from the shared dataspace to a private buffer before validating and using it.

6. Once the server has finished processing the data and written the results to the dataspace, it replies to the RPC. Thereby, it hands back the role of the user of the dataspace to the client.

7. The client resumes its execution with the return of the RPC call, and can read the result of the server-side operation from the dataspace.

GENODE ⫽ LABS

shared between source and sink
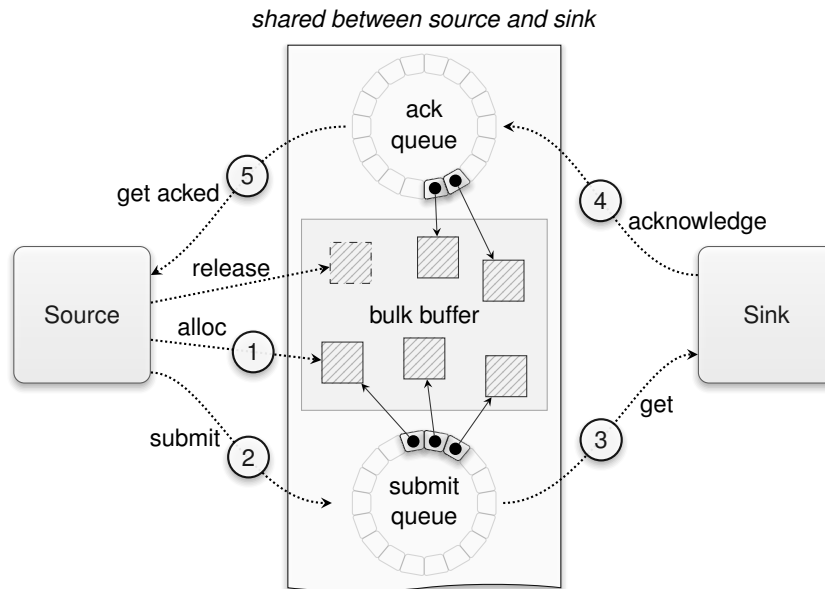
Figure 28: Life cycle of a data packet transmitted over the packet-stream interface

The RPC call may be used for carrying control information. For example, the client may provide the amount of data to process, or the server may provide the amount of data produced.

3.6.6. Asynchronous bulk transfer - packet streams

The packet-stream interface complements the facilities for the synchronous data transfer described in Sections 3.6.1 and 3.6.5 with a mechanism that carries payload over a shared memory block and employs an asynchronous data-flow protocol. It is designed for large bulk payloads such as network traffic, block-device data, video frames, and USB URB payloads.

As illustrated in Figure 28, the communication buffer consists of three parts: a submit queue, an acknowledgement queue, and a bulk buffer. The submit queue contains packets generated by the source to be processed by the sink. The acknowledgement queue contains packets that are processed and acknowledged by the sink. The bulk buffer contains the actual payload. The assignment of packets to bulk-buffer regions is performed by the source.

A packet is represented by a packet descriptor that refers to a portion of the bulk buffer and contains additional control information. Such control information may include an opcode and further arguments interpreted at the sink to perform an operation on the supplied packet data. Either the source or the sink is in charge of handling a given packet at a given time. At the points 1, 2, and 5, the packet is owned by the

GENODE LABS

source. At the points 3 and 4, the packet is owned by the sink. Putting a packet descriptor in the submit or acknowledgement queue represents a handover of responsibility. The life cycle of a single packet looks as follows:

1. The source allocates a region of the bulk buffer for storing the packet payload *(packet alloc)*. It then requests the local pointer to the payload *(packet content)* and fills the packet with data.

2. The source submits the packet to the submit queue *(submit packet)*.

3. The sink requests a packet from the submit queue *(get packet)*, determines the local pointer to the payload *(packet content)*, and processes the contained data.

4. After having finished the processing of the packet, the sink acknowledges the packet *(acknowledge packet)*, placing the packet into the acknowledgement queue.

5. The source reads the packet from the acknowledgement queue and releases the packet *(release packet)*. Thereby, the region of the bulk buffer that was used by the packet becomes marked as free.

This protocol has four corner cases that are handled by signals:

Submit queue is full when the source is trying to submit a new packet. In this case, the source blocks and waits for the sink to remove packets from the submit queue. If the sink observes such a condition (when it attempts to get a packet from a full submit queue), it delivers a *ready-to-submit* signal to wake up the source.

Submit queue is empty when the sink tries to obtain a packet from an empty submit queue, it may block. If the source places a packet into an empty submit queue, it delivers a *packet-avail* signal to wake up the sink.

Acknowledgement queue is full when the sink tries to acknowledge a packet at a saturated acknowledgement queue, the sink needs to wait until the source removes an acknowledged packet from the acknowledgement queue. The source notifies the sink about this condition by delivering a *ready-to-ack* signal. On reception of the signal, the sink wakes up and proceeds to submit packets into the acknowledgement queue.

Acknowledgement queue is empty when the source tries to obtain an acknowledged packet *(get acked packet)* from an empty acknowledgement queue. In this case, the source may block until the sink places another acknowledged packet into the empty acknowledgement queue and delivers an *ack-avail* signal.

If bidirectional data exchange between a client and a server is desired, there are two approaches:

GENODE ⬙ LABS

One stream of operations If data transfers in either direction are triggered by the
client only, a single packet stream where the client acts as the source and the server
represents the sink can accommodate transfers in both directions. For example,
the block session interface (Section 4.5.8) represents read and write requests as
packet descriptors. The allocation of the operation's read or write buffer within
the bulk buffer is performed by the client, being the source of the stream of oper-
ations. For write operations, the client populates the write buffer with the to-be-
written information before submitting the packet. When the server processes the
incoming packets, it distinguishes the read and write operations using the control
information given in the packet descriptor. For a write operation, it processes the
information contained in the packet. For a read operation, it populates the packet
with new information before acknowledging the packet.

Two streams of data If data transfers in both directions can be triggered indepen-
dently from client and server, two packet streams can be used. For example, the
NIC session interface (Section 4.5.11) uses one packet stream for ingoing and one
packet stream for outgoing network traffic. For outgoing traffic, the client plays
the role of the source. For incoming traffic, the server (such as a NIC driver) is the
source.

GENODE ⬚ LABS

4. Components

The architecture introduced in Chapter 3 clears the way to compose sophisticated systems out of many building blocks. Each building block is represented by an individual component that resides in a dedicated protection domain and interacts with other components in a well-defined manner. Those components do not merely represent applications but all typical operating-system functionalities.

Components can come in a large variety of shape and form. Compared to a monolithic operating-system kernel, a component-based operating system challenges the system designer by enlarging the design space with the decision of the functional scope of each component and thereby the granularity of componentization. This decision depends on several factors:

Security The smaller a component, the lower the risk for bugs and vulnerabilities. The more rigid a component's interfaces, the smaller its attack surface becomes. Hence, the security of a complex system function can potentially be vastly improved by splitting it into a low-complexity component that encapsulates the security-critical part and a high-complexity component that is uncritical for security.

Performance The split of functionality into multiple components introduces inter-component communication and thereby context-switch overhead. If a functionality is known to be performance critical, such a split should clearly be motivated by a benefit for security.

Reusability Componentization can be pursued to improve reusability while sometimes disregarding performance considerations at the same time. However, reusability can also be achieved by moving functionality into libraries that can easily be reused by linking them directly against library-using components. By using a dynamic linker, linking can even happen at run time, which yields the same flexibility as the use of multiple distinct components. Therefore, the split of functionality into multiple components for the sole sake of modularization has to be questioned.

Sections 4.1, 4.2, 4.3, and 4.4 aid the navigation within the componentization design space by discussing the different roles a component can play within a Genode system. This can be the role of a device driver, protocol stack, resource multiplexer, runtime environment, and that of an application. By distinguishing those roles, it becomes possible to assess the possible security implications of each individual component.

The versatility of a component-based system does not come from the existence of many components alone. Even more important is the composability of components. Components can be combined only if their interfaces match. To maximize composability, the number of interfaces throughout the system should be as low as possible, and

GENODE LABS

all interfaces should be largely orthogonal to each other. Section 4.5 reviews Genode's common session interfaces.

Components can be used in different ways depending on their configuration and their position within the component tree. Section 4.6 explains how a component obtains and processes its configuration. Section 4.7 discusses the most prominent options of composing components.
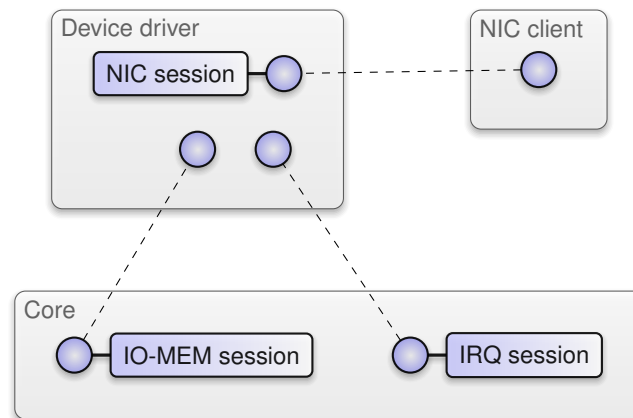
Figure 29: A network device driver provides a NIC service to a single client and uses core's
IO-MEM and IRQ services to interact with the physical network adaptor.

4.1. Device drivers

A device driver translates a device interface to a Genode session interface. Figure 29
illustrates the typical role of a device driver.

The device interface is defined by the device vendor and typically comprises the
driving of state machines of the device, the notification of device-related events via in-
terrupts, and the means to transfer data from and to the device. A device-driver compo-
nent accesses the device interface via sessions to the core services IO_MEM, IO_PORT,
and IRQ as described in Section 3.4.8.

In general, a physical device cannot safely be driven by multiple users at the same
time. If multiple users accessed one device concurrently, the device state would eventu-
ally become inconsistent. A device driver should not attempt to multiplex a hardware
device. Instead, to keep its complexity low, it should act as a server that serves only a
single client per physical device. Whereas a device driver for a simple device usually
accepts only one client, a device driver for a complex device with multiple sub devices
(such as a USB driver) may hand out each sub device to a different client.

A device driver should be largely void of built-in policy. If it merely translates the
interface of a single device to a session interface, there is not much room for policy
anyway. If, however, a device driver hands out multiple sub devices to different clients,
the assignment of sub devices to clients must be subjected to a policy. In this case, the
device driver should obtain policy information from its configuration as provided by
the driver's parent.

GENODE LABS

4.1.1. Platform driver

There are three problems that are fundamentally important for running an operating system on modern hardware but that lie outside the scope of an ordinary device driver because they affect the platform as a whole rather than a single device. Those problems are the enumeration of devices, the discovery of interrupt routing, and the initial setup of the platform.

Problem 1: Device enumeration Modern hardware platforms are rather complex and vary a lot. For example, the devices attached to the PCI bus of a PC are usually not known at the build time of the system but need to be discovered at run time. Technically, each individual device driver could probe its respective device at the PCI bus. But in the presence of multiple drivers, this approach would hardly work. First, the configuration interface of the PCI bus is a device itself. The concurrent access to the PCI configuration interface by multiple drivers would ultimately yield undefined behaviour. Second, for being able to interact directly with the PCI configuration interface, each driver would need to carry with it the functionality to interact with PCI.

Problem 2: Interrupt routing On PC platforms with multiple processors, the use of legacy interrupts as provided by the Intel 8259 programmable interrupt controller (PIC) is not suitable because there is no way to express the assignment of interrupts to CPUs. To overcome the limitations of the PIC, Intel introduced the Advanced Programmable Interrupt Controller (APIC). The APIC, however, comes with a different name space for interrupt numbers, which creates an inconsistency between the numbers provided by the PCI configuration (interrupt lines) and interrupt numbers as understood by the APIC. The assignment of legacy interrupts to APIC interrupts is provided by the Advanced Configuration and Power Interface (ACPI) tables. Consequently, in order to support multi-processor PC platforms, the operating system needs to interpret those tables. Within a component-based system, we need to answer the question of which component is responsible to interpret the ACPI tables and how this information is applied to individual device drivers.

Problem 3: Initial hardware setup In embedded systems, the interaction of the SoC (system on chip) with its surrounding peripheral hardware is often not fixed in hardware but rather a configuration issue. For example, the power supply and clocks of certain peripherals may be enabled by speaking an I2C protocol with a separate power-management chip. Also, the direction and polarity of the general-purpose I/O pins depends largely on the way how the SoC is used. Naturally, such hardware setup steps could be performed by the kernel. But this would require the kernel to become aware of potentially complex platform intrinsics.

GENODE ⩘ LABS

Central platform driver   The natural solution to these problems is the introduction of a so-called platform driver, which encapsulates the peculiarities outlined above. On PC platforms, the role of the platform driver is executed by the ACPI driver. The ACPI driver provides an interface to the PCI bus in the form of a PCI service. Device drivers obtain the information about PCI devices by creating a PCI session at the ACPI driver. Furthermore, the ACPI driver provides an IRQ service that transparently applies the interrupt routing based on the information provided by the ACPI tables. Furthermore, the ACPI driver provides the means to allocate DMA buffers, which is further explained in Section 4.1.3.

On ARM platforms, the corresponding component is named platform driver and provides a so-called platform service. Because of the large variety of ARM-based SoCs, the session interface for this service differs from platform to platform.

4.1.2. Interrupt handling

Most device drivers need to respond to sporadic events produced by the device and propagated to the CPU as interrupts. In Genode, a device-driver component obtains device interrupts via core's IRQ service introduced in Section 3.4.8. On PC platforms, device drivers usually do not use core's IRQ service directly but rather use the IRQ service provided by the platform driver (Section 4.1.1).

4.1.3. Direct memory access (DMA) transactions

Devices that need to transfer large amounts of data usually support a means to issue data transfers from and to the system's physical memory without the active participation of the CPU. Such transfers are called *direct memory access (DMA) transactions*. DMA transactions relieve the CPU from actively copying data between device registers and memory, optimize the throughput of the system bus by the effective use of burst transfers, and may even be used to establish direct data paths between devices. However, the benefits of DMA come at the risk of corrupting the physical memory by misguided DMA transactions. Because those DMA-capable devices can issue bus requests that target the physical memory directly while not involving the CPU altogether, such requests are naturally not subjected to the virtual-memory mechanism implemented in the CPU in the form of a memory-management unit (MMU). Figure 30 illustrates the problem. From the device's point of view, there is just physical memory. Hence, if a driver sets up a DMA transaction, e. g., if a disk driver wants to read a block from the disk, it programs the memory-mapped registers of the device with the address and size of a physical-memory buffer where it expects to receive the data. If the driver lives in a user-level component, as is the case for a Genode-based system, it still needs to know the physical address of the DMA buffer to program the device correctly. Unfortunately, there is nothing to prevent the driver from specifying any physical address to the device. A malicious driver could misuse the device to read and manipulate all

GENODE 🐘 LABS
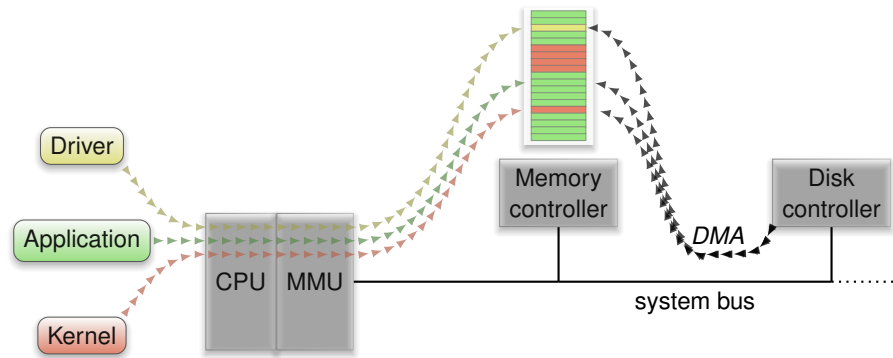
Figure 30: The MMU restricts the access of physical memory pages by different components according to their virtual address spaces. However, direct memory accesses issued by the disk controller are not subjected to the MMU. The disk controller can access the entirety of the physical memory present in the system.

parts of the physical memory, including the kernel. Consequently, device drivers and devices should ideally be trustworthy. However, there are several scenarios where this is ultimately not the case.

Scenario 1: Direct device assignment to virtual machines   When hosting virtual machines as Genode components, the direct assignment of a physical device such as a USB controller, a GPU, or a dedicated network card to the guest OS running in the virtual machine can be useful in two ways. First, if the guest OS is the sole user of the device, direct assignment of the device maximizes the I/O performance of the guest OS using the device. Second, the guest OS may be equipped with a proprietary device driver that is not present as a Genode component otherwise. In this case, the guest OS may be used as a runtime that executes the device driver, and thus, provides a driver interface to the Genode world. In both cases the guest OS should not be considered as trustworthy. On the contrary, it bears the risk of subverting the isolation between components. A misbehaving guest OS could issue DMA requests referring to the physical memory used by other components or even the kernel, and thereby break out of its virtual machine.

Scenario 2: Firmware-driven attacks   Modern peripherals such as wireless LAN adaptors, network cards, or GPUs employ firmware executed on the peripheral device. This firmware is executed on a microcontroller on the device, and is thereby not subjected to the policy of the normal operating system. Such firmware may either be built-in by the device vendor, or is loaded by the device driver at initialization time of the device. In both cases, the firmware tends to be a black box that remains obscure with the exception of the device vendor. Hidden functionality or vulnerabilities might be

GENODE ⬊ LABS

present in it. By the means of DMA transactions, such firmware has unlimited access to the system. For example, a back door implemented in the firmware of a network adaptor could look for special network packets to activate and control arbitrary spyware. Because malware embedded in the firmware of the device can neither be detected nor controlled by the operating system, both monolithic and microkernel-based operating systems are powerless against such attacks.

Scenario 3: Bus-level attacks The previous examples misuse a DMA-capable device as a proxy to drive an attack. However, the system bus can be attacked directly with no hardware tinkering at all. There are ready-to-exploit interfaces that are featured on most PC systems. For example, most laptops come with PCMCIA / Express-Card slots, which allow expansion cards to access the system bus. Furthermore, serial bus interfaces, e. g., IEEE 1394 (Firewire), enable connected devices to indirectly access the system bus via the peripheral bus controller. If the bus controller allows the device to issue direct system bus requests by default, a connected device becomes able to gain control over the whole system.

DMA transactions in component-based systems Direct memory access (DMA) of devices looks like the Achilles heel of component-based operating systems. The most compelling argument in favor of componentization is that by encapsulating each system component within a dedicated user-level address space, the system as a whole becomes more robust and secure compared to a monolithic operating-system kernel. In the event that one component fails due to a bug or an attack, other components remain unaffected. The prime example for such buggy components are, however, device drivers. By empirical evidence, those remain the most prominent trouble makers in today's operating systems, which suggests that the DMA loophole renders the approach of component-based systems largely ineffective. However, there are three counter arguments to this observation.

First, by encapsulating each driver in a dedicated address space, classes of bugs that are unrelated to DMA remain confined in the driver component. In practice most driver-related problems stem from issues like memory leaks, synchronization problems, deadlocks, flawed driver logic, wrong state machines, or incorrect device-initialization sequences. For those classes of problems, the benefits of isolating the driver in a dedicated component still apply.

Second, executing a driver largely isolated from other operating-system code minimizes the attack surface onto the driver. If the driver interface is rigidly small and well-defined, it is hard to compromise the driver by exploiting its interface.

Third, modern PC hardware has closed the DMA loophole by incorporating so-called IOMMUs into the system. As depicted in Figure 31, the IOMMU sits between the physical memory and the system bus where the devices are attached to. So each DMA request has to go through the IOMMU, which is not only able to arbitrate the access of DMA

99

GENODE ⟋⟍ LABS
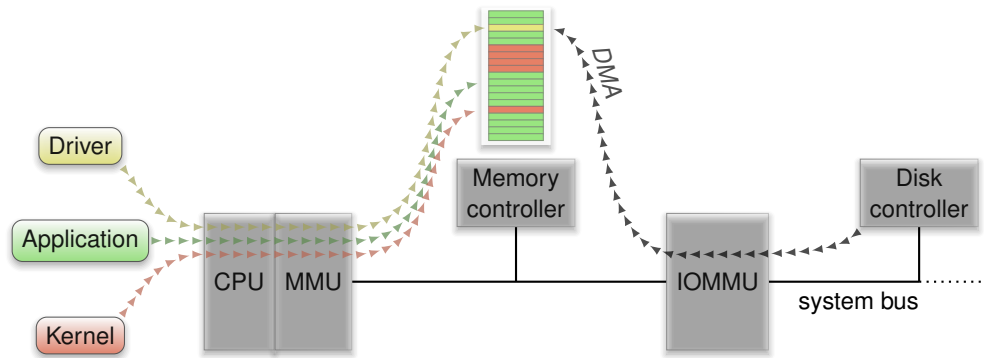
Figure 31: An IOMMU arbitrates and virtualizes DMA accesses issued by a device to the RAM. Only if a valid IOMMU mapping exists for a given DMA access, the memory access is performed.

requests to the RAM but is also able to virtualize the address space per device. Similar to how an MMU confines each process running on the CPU within a distinct virtual address space, the IOMMU is able to confine each device within a dedicated virtual address space. To tell the different devices apart, the IOMMU uses the PCI device's bus-device-function triplet as unique identification.

With an IOMMU in place, the operating system can effectively limit the scope of actions the given device can execute on the system. I.e., by restricting all accesses originating from a particular PCI device to the DMA buffers used for the communication, the operating system becomes able to detect and prevent any unintended bus accesses initiated by the device.

When executed on the NOVA kernel, Genode subjects all DMA transactions to the IOMMU, if present. Section 7.7.7 discusses the use of IOMMUs in more depth.

GENODE ⋙ LABS
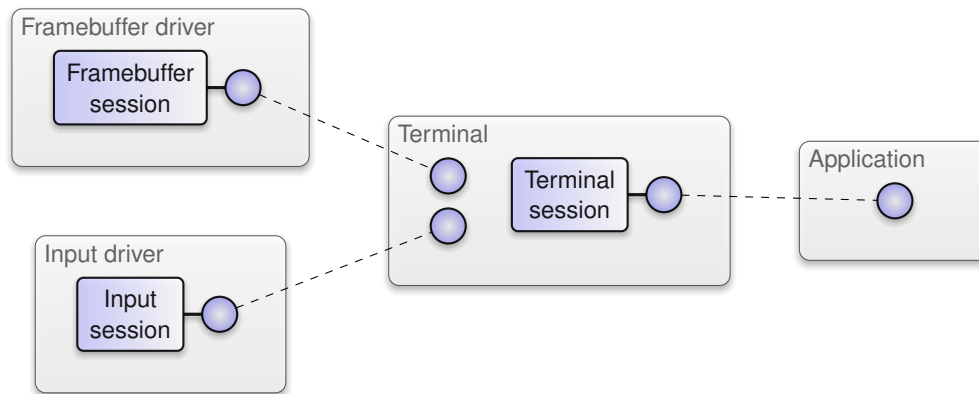
Figure 32: Example of a protocol stack. The terminal provides the translation between the terminal-session interface (on the right) and the driver interfaces (on the left).

4.2. Protocol stacks

A protocol stack *translates* one session interface to another (or the same) session interface. For example, a terminal component may provide a command-line application with a service for obtaining textual user input and for printing text. To implement this service, the terminal uses an input session and a framebuffer session. Figure 32 depicts the relationship between the terminal, its client application, and the used drivers. For realizing the output of a stream of characters on screen, it implements a parser for escape sequences, maintains a state machine for the virtual terminal, and renders the pixel representation of characters onto the framebuffer. For the provisioning of textual user input, it responds to key presses reported by the input session, maintains the state of modifier keys, and applies a keyboard layout to the stream of incoming events. When viewed from the outside of the component, the terminal translates a terminal session to a framebuffer session as well as an input session.

Similar to a device driver, a protocol stack typically serves a single client. In contrast to device drivers, however, protocol stacks are not bound to physical devices. Therefore, a protocol stack can be instantiated any number of times. For example, if multiple terminals are needed, one terminal component could be instantiated per terminal. Because each terminal uses an independent instance of the protocol stack, a bug in the protocol stack of one terminal does not affect any other terminal. However complex the implementation of the protocol stack may be, it is not prone to leaking information to another terminal because it is connected to a single client only. The leakage of information is constrained to interfaces used by the individual instance. Hence, in cases like this, the protocol-stack component is suitable for hosting highly complex untrusted code if such code cannot be avoided.

Note that the example above cannot be generalized for all protocol stacks. There are protocol stacks that are critical for the confidentiality of information. For example, an

GENODE 🐌 LABS

in-band encryption component may translate plain-text network traffic to encrypted network traffic designated to be transported over a public network. Even though the component is a protocol stack, it may still be prone to leaking unencrypted information to the public network.

Whereas protocol stacks are not necessarily critical for integrity and confidentiality, they are almost universally critical for availability.

GENODE LABS
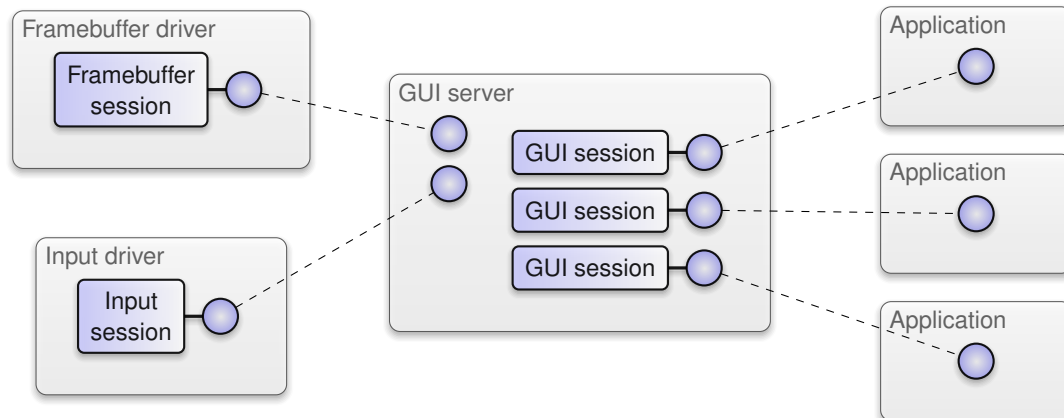
Figure 33: A GUI server multiplexes the physical framebuffer and input devices among multiple applications.

4.3. Resource multiplexers

A resource multiplexer transforms one resource into a number of virtual resources. A resource is typically a session to a device driver. For example, a NIC-switch component may use one NIC session to a NIC driver as uplink and, in turn, provide a NIC service where each session represents a virtual NIC. Another example is a GUI server as depicted in Figure 33, which enables multiple applications to share the same physical framebuffer and input devices by presenting each client in a window or a virtual console.

In contrast to a typical device driver or protocol stack that serves only a single client, a resource multiplexer is shared by potentially many clients. In the presence of untrusted clients besides security-critical clients, a resource multiplexer ultimately becomes a so-called *multi-level* component. This term denotes that the component is cross-cutting the security levels of all its clients. This has the following ramifications.

Covert channels   Because the component is a shared resource that is accessed by clients of different security levels, it must maintain the strict isolation between its clients unless explicitly configured otherwise. Hence, the component's client interface as well as the internal structure must be designed to prevent the leakage of information across clients. I.e., two clients must never share the same namespace of server-side objects if such a namespace can be modified by the clients. For example, a window server that hands out global window IDs to its clients is prone to unintended information leakage because one client could observe the allocation of window IDs by another client. The ID allocation could be misused as a covert channel that circumvents security policies. In the same line, a resource multiplexer is prone to timing channels if the operations provided via its client interface depends on the behavior of other clients. For this reason, blocking RPC

103

calls should be avoided because the duration of a blocking operation may reveal information about the internal state such as the presence of other clients of the resource multiplexer.

Complexity is dangerous   As a resource multiplexer is shared by clients of different security levels, the same considerations apply as for the OS kernel: high complexity poses a major risk for bugs. Such bugs may, in turn, result in the unintended flow of information between clients or degrade the quality of service for all clients. Hence, in terms of complexity, resource multiplexers must be as simple as possible.

Denial of service   The exposure of a resource multiplexer to untrusted and even malicious clients makes it a potential target for denial-of-service attacks. Some operations provided by the resource multiplexer may require the allocation of memory. For example, a GUI server may need memory for the book keeping of each window created by its clients. If the resource multiplexer performed such allocations from its own memory budget, a malicious client could trigger the exhaustion of server-side memory by creating new windows in an infinite loop. To mitigate this category of problems, a resource multiplexer should perform memory allocations exclusively from client-provided resources, i. e., using the session quota as provided by each client at session-creation time. Section 3.3 describes Genode's resource-trading mechanism in detail. In particular, resource multiplexers should employ heap partitioning as explained in Section 3.3.3.

Avoiding built-in policies   A resource multiplexer can be understood as a microkernel for a higher-level resource. Whereas a microkernel multiplexes or arbitrates the CPU and memory between multiple components, a resource multiplexer does the same for sessions. Hence, the principles for constructing microkernels equally apply for resource multiplexers. In the line of those principles, a resource multiplexer should ideally implement sole mechanisms but should be void of built-in policy.

Enforcement of policy   Instead of providing a built-in policy, a resource multiplexer obtains policy information from its configuration as supplied by its parent. The resource multiplexer must enforce the given policy. Otherwise, the security policy expressed in the configuration remains ineffective.

GENODE ⦚ LABS
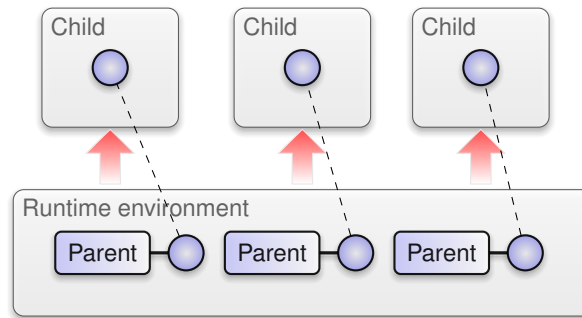
Figure 34: A runtime environment manages multiple child components.

4.4. Runtime environments and applications

The component types discussed in the previous sections have in common that they deliberately lack built-in policy but act according to a policy supplied by their respective parents by the means of configuration. This raises the question where those policies should come from. The answer comes in the form of runtime environments and applications.

A *runtime environment* as depicted in Figure 34 is a component that hosts child components. As explained in the Sections 3.2 and 3.3, it is thereby able to exercise control over its children but is also responsible to manage the children's resources. A runtime environment controls its children in three ways:

Session routing   It is up to the runtime environment to decide how to route session requests originating from a child. The routing of sessions is discussed in Section 3.2.3.

Configuration   Each child obtains its configuration from its parent in the form of a ROM session as described in Section 4.6. Using this mechanism, the runtime environment is able to feed policy information to its children. Of course, in order to make the policy effective, the respective child has to interpret and enforce the configuration accordingly.

Lifetime   The lifetime of a child ultimately depends on its parent. Hence, a runtime environment can destroy and possibly restart child components at any time.

With regard to the management of child resources, a runtime environment can employ a large variety of policies using two principal approaches:

Quota management   Using the resource-trading mechanisms introduced in Section 3.3, the runtime environment can assign resources to each child individually. Moreover, if a child supports the dynamic rebalancing protocol described in Section 3.3.4, the runtime environment may even change those assignments over the lifetime of its children.

105

GENODE ⋙ LABS

Interposing services Because the runtime environment controls the session routing
of each child, it is principally able to interpose the child's use of any service in-
cluding those normally provided by core such as RAM (Section 3.4.3), PD (Sec-
tion 3.4.5), and CPU (Section 3.4.7). The runtime environment may provide a
locally implemented version of those session interfaces instead of routing session
requests directly towards the core component. Internally, each session of such a
local service may create a session to the real core service, thereby effectively wrap-
ping core's sessions. This way, the runtime environment can not only observe the
interaction of its child with core services but also implement custom resource-
management strategies, for example, sharing one single budget among multiple
children.

Canonical examples of runtime environments are the init component that applies a
policy according to its configuration, the noux runtime that presents itself as a Unix
kernel to its children, a debugger that interposes all core services for the debugging
target, or a virtual machine monitor.

A typical *application* is a leaf node in the component tree that merely uses services. In
practice, however, the boundary between applications and runtime environments can
be blurry. As illustrated in Section 4.7, Genode fosters the internal split of applications
into several components, thereby forming *multi-component applications*. From the out-
side, such a multi-component application appears as a leaf node of the component tree
but internally, it employs an additional level of componentization by executing portions
of its functionality in separate child components. The primary incentive behind this ap-
proach is the sandboxing of untrusted application functionality. For example, a video
player may execute the video codec within a separate child component so that a bug in
the complex video codec will not compromise the entire video-player application.

GENODE ⬗ LABS

4.5. Common session interfaces

The core services described in Section 3.4 principally enable the creation of a recursively structured system. However, their scope is limited to the few low-level resources provided by core, namely processing time, memory, and low-level device resources. Device drivers (Section 4.1) and protocol stacks (Section 4.2) transform those low-level resources into higher-level resources. Analogously to how core's low-level resources are represented by the session interfaces of core's services, higher-level resources are represented by the session interfaces provided by device drivers and protocol stacks. In principle, each device driver could introduce a custom session interface representing the particular device. But as discussed in the introduction of Chapter 4, a low number of orthogonal session interfaces is desirable to maximize the composability of components. This section introduces the common session interfaces that are used throughout Genode.

4.5.1. Read-only memory (ROM)

The ROM session interface makes a piece of data in the form of a dataspace available to the client.

Session creation   At session-creation time, the client specifies the name of a ROM module as session argument. One server may hand out different ROM modules depending on the name specified. Once a ROM session has been created, the client can request the capability of the dataspace that contains the ROM module. Using this capability and the region map of the client's PD session, the client can attach the ROM module to its local address space and thereby access the information. The client is expected to merely read the data, hence the name of the interface.

ROM module updates   In contrast to the intuitive assumption that read-only data is immutable, ROM modules may mutate during the lifetime of the session. The server may update the content of the ROM module with new versions. However, the server does not do so without the consent of the client. The protocol between client and server consists of the following steps.

1. The client registers a signal handler at the server to indicate that it is interested in receiving updates of the ROM module.

2. If the server has a new version of the ROM module, it does not immediately change the dataspace shared with the client. Instead, it maintains the new version separately and informs the client by submitting a signal to the client's signal handler.

GENODE ⋛ LABS

3. The client continues working with the original version of the dataspace. Once it receives the signal from the server, it may decide to update the dataspace by calling the *update* function at the server.

4. The server responds to the update request. If the new version fits into the existing dataspace, the server copies the content of the new version into the existing dataspace and returns this condition with the reply of the update call. Thereby, the ROM session interface employs synchronous bulk transfers as described in Section 3.6.5.

5. The client evaluates the result of the update call. If the new version did fit into the existing dataspace, the update is complete at this point. However, if the new version is larger than the existing dataspace, the client requests a new dataspace from the server.

6. Upon reception of the dataspace request, the server destroys the original dataspace (thereby making it invisible to the client), and returns the new version of the ROM module as a freshly allocated dataspace.

7. The client attaches the new dataspace capability to its local address space to access the new version.

The protocol is designed in such a way that neither the client nor the server need to support updates. A server with no support for updating ROM modules such as core's ROM service simply ignores the registration of a signal handler by a client. A client that is not able to cope with ROM-module updates never requests the dataspace twice.

However, if both client and server support the update protocol, the ROM session interface provides a means to propagate large state changes from the server to the client in a transactional way. In the common case where the new version of a ROM module fits into the same dataspace as the old version, the update does not require any memory mappings to be changed.

Use cases   The ROM session interface is used wherever data shall be accessed in a memory mapped fashion.

- Boot time data comes in the form of the ROM sessions provided by core's ROM service. On some kernels, core exports kernel-specific information such as the kernel version in the form of special ROM modules.

- If an executable binary is provided as a ROM module, the binary's text segment can be attached directly to the address space of a new process (Section 3.5). So multiple instances of the same component effectively share the same text segment. The same holds true for shared libraries. For this reason, executable binaries and shared libraries are requested in the form of ROM sessions.

GENODE ⬕ LABS

- Components obtain their configuration by requesting a ROM session for the "config" ROM module at their respective parent (Section 4.6). This way, configuration information can be propagated using a simple interface with no need for a file system. Furthermore, the update mechanism allows the parent to dynamically change the configuration of a component during its lifetime.

- As described in Section 4.7.5, multi-component applications may obtain data models in the form of ROM sessions. In such scenarios, the ROM session's update mechanism is used to propagate model updates in a transactional way.

4.5.2. Report

The report session interface allows a client to report its internal state to the outside using synchronous bulk transfers (Section 3.6.5).

Session creation   At session-creation time, the client specifies a label and a buffer size. The label aids the routing of the session request but may also be used to select a policy at the report server. The buffer size determines the size of the dataspace shared between the report server and its client.

Use cases

- Components may use report sessions to export their internal state for monitoring purposes or for propagating exceptional events.

- Device drivers may report information about detected devices or other resources. For example, a bus driver may report a list of devices attached on the bus, or a wireless driver may report the list of available networks.

- In multi-component applications, components that provide data models to other components may use the report-session interface to propagate model updates.

4.5.3. Terminal and UART

The terminal session interface provides a bi-directional communication channel between client and server using synchronous bulk transfers (Section 3.6.5). It is primarily meant to be used for textual interfaces but may also be used to transfer other serial streams of data.

The interface uses the two RPC functions *read* and *write* to arbitrate the access to a shared-memory communication buffer between client and server as described in Section 3.6.5. The read function never blocks. When called, it copies new input into the communication buffer and returns the number of new characters. If there is no new input, it returns 0. To avoid the need to poll for new input at the client side, the client can

GENODE ⋛ LABS

register a signal handler that gets notified upon the arrival of new input. The write function takes the number of to-be-written characters as argument. The server responds to this function by processing the specified amount of characters from the communication buffer.

Besides the actual read and write operations, the terminal supports the querying of the number of new available input events (without reading it) and the terminal size in rows and columns.

Session creation   At session-creation time, the terminal session may not be ready to use. For example, a TCP terminal session needs an established TCP connection first. In such a situation, the use of the terminal session by a particular client must be deferred until the session becomes ready. Delaying the session creation at the server side is not an option because this would render the server's entry point unavailable for all other clients until the TCP connection is ready. Instead, the client blocks until the server delivers a `connected` signal. This signal is emitted when the session becomes ready to use. The client waits for this signal right after creating the session.

Use cases

- Device drivers that provide streams of characters in either direction.

- A graphical terminal.

- Transfer of streams of data over TCP (using the TCP terminal).

- Writing streams of data to a file (using a file terminal).

- User input and output of traditional command-line based software, i. e., programs executed in the noux runtime environment.

- Multiplexing of multiple textual user interfaces (using the terminal-mux component).

- Headless operation and management of subsystems (using the CLI monitor).

UART   The UART session interface complements the terminal session interface with additional control functions, e. g., for setting the baud rate. Because UART sessions are compatible to terminal sessions, a UART device driver can be used as both UART server and terminal server.

4.5.4. Input

The input session interface is used to communicate low-level user-input events from the server to the client using synchronous bulk transfers (Section 3.6.5). Such an event can be of one of the following types:

GENODE ⚞ LABS

press or release of a button or key. Each physical button (such as a mouse button) or key (such as a key on a keyboard) is represented by a unique value. At the input-session level, key events are reported as raw hardware events. They are reported without a keyboard layout applied and without any interpretation of meta keys (like shift, alt, and control). This gives the client the flexibility to handle arbitrary combinations of keys.

relative motion of pointer devices such as a mouse. Such events are generated by device drivers.

absolute motion of pointer devices such as a touch screen or graphics tablet. Furthermore absolute motion events are generated by virtual input devices such as the input session provided by a GUI server.

wheel motion of scroll wheels in vertical and horizontal directions.

focus of the session. Focus events are artificially generated by servers to indicate a gained or lost keyboard focus of the client. The client may respond to such an event by changing its graphical representation accordingly.

leave of the pointer position. Leave events are artificially generated by servers to indicate a lost pointer focus.

Use cases

- Drivers for user-input devices play the roles of input servers.

- Providing user input from a GUI server to its clients, e. g., the interface of the nitpicker GUI server provides an input session as part of the server's interface.

- Merging multiple streams of user input into one stream (using an input merger).

- Virtual input devices can be realized as input servers that generate artificial input events.

4.5.5. Framebuffer

The framebuffer session interface allows a client to supply pixel data to a framebuffer server such as a framebuffer driver or a virtual framebuffer provided by a GUI server. The client obtains access to the framebuffer as a dataspace, which is shared between client and server. The client may update the pixels within the dataspace at any time. Once a part of the framebuffer has been updated, the client informs the server by calling a *refresh* RPC function. Thereby, the framebuffer session interface employs a synchronous bulk transfer mechanism (Section 3.6.5).

GENODE 🔺 LABS

Session creation   In general, the screen mode is defined by the framebuffer server, not the client. The mode may be constrained by the physical capabilities of the hardware or depend on the driver configuration. Some framebuffer servers, however, may take a suggestion by the client into account. At session-creation time, the client may specify a preferred mode as session argument. Once the session is constructed, however, the client must request the actually used mode via the *mode* RPC function.

Screen-mode changes   The session interface supports dynamic screen-mode changes during the lifetime of the session using the following protocol:

1. The client may register a signal handler using the *mode_sigh* RPC function. This handler gets notified in the event of server-side mode changes.

2. Similarly to the transactional protocol used for updating ROM modules (Section 4.5.1), the dataspace shared between client and server stays intact until the client acknowledges the mode change by calling the *mode* RPC function.

3. The server responds to the *mode* function by applying the new mode and returns the corresponding mode information to the client. This step may destroy the old framebuffer dataspace.

4. The client requests a new version of the frambuffer dataspace by calling the *dataspace* RPC function and attaches the dataspace to its local address space. Note that each subsequent call of the dataspace RPC function may result in the replacement of the existing dataspace by a new dataspace. Hence, calling *dataspace* twice may invalidate the dataspace returned from the first call.

Frame-rate synchronization   To enable framebuffer clients to synchronize their operations with the display frequency, a client can register a handler for receiving display-synchronization events as asynchronous notifications (Section 3.6.2).

Use cases

- Framebuffer device drivers are represented as framebuffer servers.

- A virtual framebuffer may provide both the framebuffer and input session interfaces by presenting a window on screen. The resizing of the window may be reflected to the client as screen-mode changes.

- A filter component requests a framebuffer session and, in turn, provides a framebuffer session to a client. This way, pixel transformations can be applied to pixels produced by a client without extending the client.

GENODE ⬚ LABS
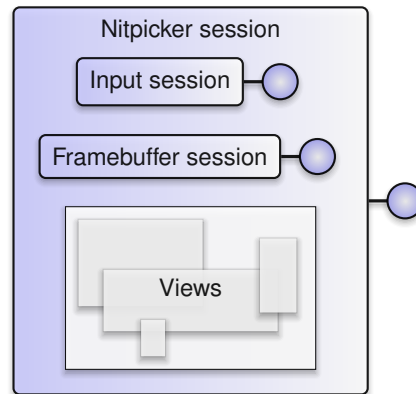
Figure 35: A nitpicker session aggregates a framebuffer session, an input session, and a session-local view stack.

4.5.6. Nitpicker GUI

The nitpicker session interface combines an input session and a framebuffer session into a single session (Figure 35). Furthermore, it supplements the framebuffer session with the notion of views, which allows the creation of flexible multi-window user interfaces. The interface is generally suited for resource multiplexers of the framebuffer and input sessions. A view is a rectangular area on screen that displays a portion of the client's virtual framebuffer. The position, size, and viewport of each view is defined by the client. Views can overlap, thereby creating a view stack. The stacking order of the views of one client can be freely defined by the client.

The size of the virtual framebuffer can be freely defined by the client but the required backing store must be provided in the form of session quota. Clients may request the screen mode of the physical framebuffer and are able to register a signal handler for mode changes of the physical framebuffer. This way, nitpicker clients are able to adapt themselves to changing screen resolutions.

Use cases

- The nitpicker GUI server allows multiple GUI applications to share a pair of a physical framebuffer session and an input session in a secure way.

- A window manager implementing the nitpicker session interface may represent each view as a window with window decorations and a placement policy. The resizing of a window by the user is reflected to the client as a screen-mode change.

- A loader (Section 4.5.14) virtualizes the nitpicker session interface for the loaded subsystem.

GENODE ⩘ LABS

4.5.7. Platform

The platform session interface (on ARM-based devices) and the PCI session interface (on x86-based machines) provide the client with access to the devices present on the hardware platform. See Section 4.1.1 for more information on the role of platform drivers.

4.5.8. Block

The block session interface allows a client to access a storage server at the block level. The interface is based on a packet stream (Section 3.6.6). Each packet represents a block-access command, which can be either read or write. Thanks to the use of the packet-stream mechanism, the client can issue multiple commands at once and thereby hide access latencies by submitting batches of block requests. The server acknowledges each packet after completing the corresponding block-command operation.

The packet-stream interface for submitting commands is complemented by the *info* RPC function for querying the properties of the block device, i. e., the supported operations, the block size, and the block count. Furthermore, a client can call the *sync* RPC function to flush caches at the block server.

Session creation   At session-creation time, the client can dimension the size of the communication buffer as session argument. The server allocates the shared communication buffer from the session quota.

Use cases

- Block-device drivers implement the block-session interface.

- The part-block component requests a single block session, parses a partition table, and hands out each partition as a separate block session to its clients. There can be one client for each partition.

- File-system servers use block sessions as their back end.

4.5.9. Regulator

The regulator session represents an adjustable value in the hardware platform. Examples are runtime-configurable frequencies and voltages. The interface is a plain RPC interface.

4.5.10. Timer

The timer session interface provides a client with a session-local time source. A client can use it to schedule timeouts that are delivered as signals to a previously registered

GENODE ⩘ LABS

signal handler. Furthermore, the client can request the elapsed number of milliseconds since the creation of the timer session.

4.5.11. NIC

A NIC session represents a network interface that operates at network-packet level. Each session employs two independent packet streams (Section 3.6.6), one for receiving network packets and one for transmitting network packets. Furthermore, the client can query the MAC address of the network interface.

Session creation   At session-creation time, the communication buffers of both packet streams are dimensioned via session arguments. The communication buffers are allocated by the server using the session quota provided by the client.

Use cases

- Network drivers are represented as NIC servers.

- A NIC switch uses one NIC session connected to a NIC driver, and provides multiple virtual NIC interfaces to its clients by managing a custom name space of virtual MAC addresses.

- A TCP/IP stack uses a NIC session as back end.

4.5.12. Audio output

The audio output interface allows for the transfer of audio data from the client to the server. One session corresponds to one channel. I.e., for stereo output, two audio-out sessions are required.

Session construction   At session-construction time, the client specifies the type of channel (e. g., front left) as session argument.

Interface design   For the output of streamed audio data, a codec typically decodes a relatively large portion of an audio stream and submits the sample data to a mixer. The mixer, in turn, mixes the samples of multiple sources and forwards the result to the audio driver. The codec, the mixer, and the audio driver are separate components. By using large buffer sizes between them, there is only very little context-switching overhead. Also, the driver can submit large buffers of sample data to the sound device without any further intervention needed. In contrast, sporadic sounds are used to inform the user about an immediate event. An example is the acoustic feedback to certain user input in games. The user ultimately expects that such sounds are played back without much latency. Otherwise the interactive experience would suffer. Hence,

GENODE ⬭ LABS

shared between source and sink
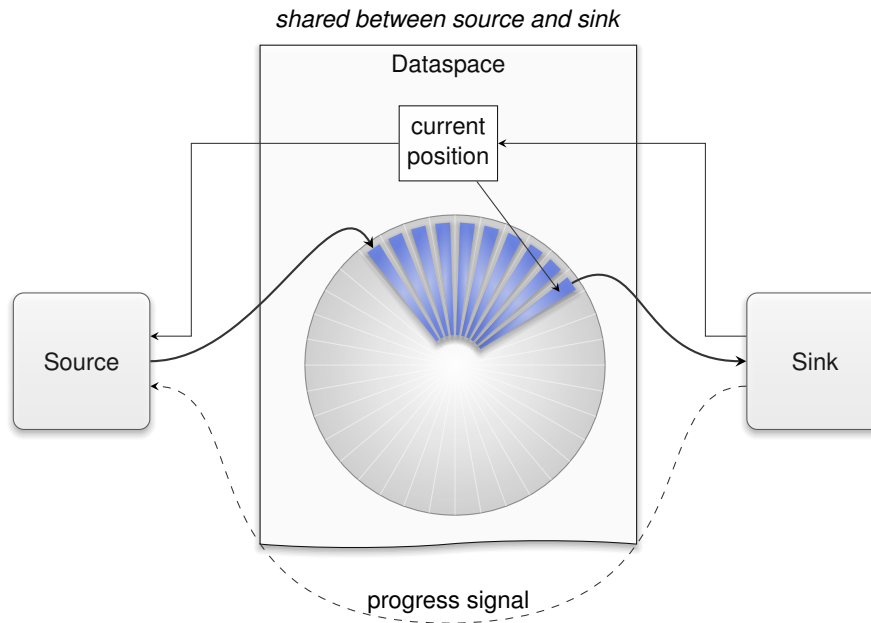
Figure 36: The time-driven audio-out session interface uses shared memory to transfer audio frames and propagate progress information.

using large buffers between the audio source, the mixer, and the driver is not an option. The audio-out session interface was specifically designed to accommodate both corner cases of audio output.

Similarly to the packet-stream mechanism described in Section 3.6.6, the audio-out session interface depicted in Figure 36 employs a combination of shared memory and asynchronous notifications. However, in contrast to the packet-stream mechanism, it has no notion of ownership of packets. When using the normal packet-stream protocol, either the source or the sink is in charge of handling a given packet at a given time, not both. The audio-out session interface weakens this notion of ownership by letting the source update once submitted audio frames even after submitting them. If there are solely continuous streams of audio arriving at the mixer, the mixer can mix those large batches of audio samples at once and pass the result to the driver.

Now, if a sporadic sound comes in, the mixer checks the current output position reported by the audio driver, and re-mixes those portions that haven't been played back yet by incorporating the sporadic sound. So the buffer consumed by the driver gets updated with new data.

Besides the way of how packets are populated with data, the second major difference to the packet-stream mechanism is its time-triggered mode of operation. The driver produces periodic signals that indicate the completeness of a played-back audio packet. This signal triggers the mixer to become active, which in turn serves as a time base for

116

GENODE ⋙ LABS
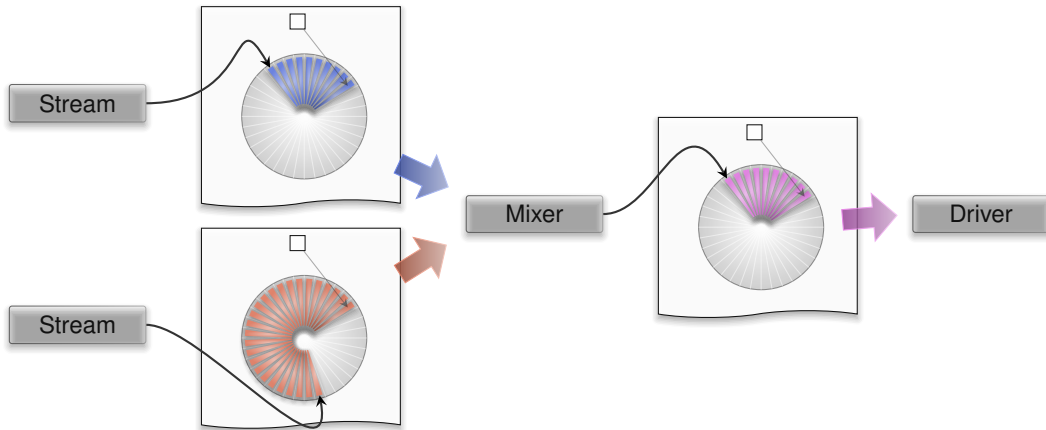
Figure 37: The mixer processes batches of incoming audio frames from multiple sources.
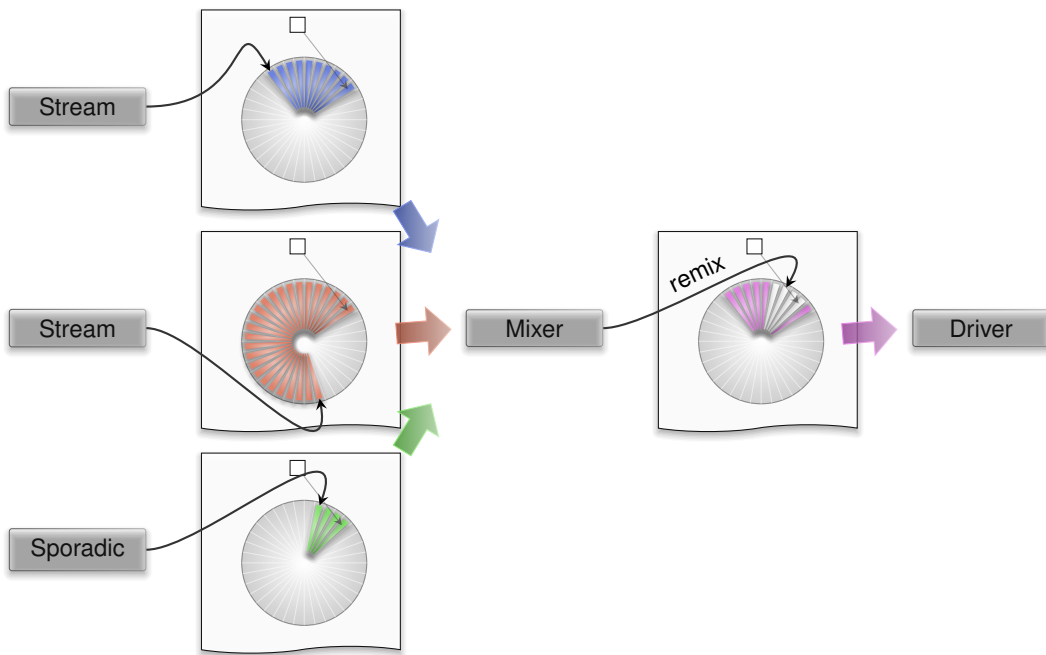
Figure 38: A sporadic occurring sound prompts the mixer to remix packets that were already submitted in the output queue.

GENODE LABS

its clients. The current playback position is denoted alongside the sample data as a field in the memory buffer shared between source and sink.

Use cases

- The audio-out session interface is provided by audio drivers.

- An audio mixer combines incoming audio streams of multiple clients into one audio stream transferred to an audio driver.

4.5.13. File system

The file-system session interface provides the client with a storage facility at the file and directory-level. Compared to the block session interface (Section 4.5.8), it operates on a higher abstraction level that is suited for multiplexing the storage device among multiple clients. Similar to the block session, the file-system session employs a single packet stream interface (Section 3.6.6) for issuing read and write operations. This way, read and write requests can be processed in batches and even out of order.

In contrast to read and write operations that carry potentially large amounts of payload, the directory functions provided by the file-system session interface are synchronous RPC functions. Those functions are used for opening, creating, renaming, moving, deleting, and querying files, directories and symbolic links.

The directory functions are complemented with an interface for receiving notifications upon file or directory changes using asynchronous notifications.

Use cases

- A file-system operates on a block session to provide file-system sessions to its clients.

- A RAM file system keeps the directory structure and files in memory and provides file-system sessions to multiple clients. Each session may be restricted in different ways (such as the root directory as visible by the respective client, or the permission to write). Thereby the clients can communicate using the RAM file system as a shared storage facility but are subjected to an information-flow policy.

- A file-system component may play the role of a filter that transparently encrypts the content of the files of its client and stores the encrypted files at another file-system server.

- A pseudo file system may use the file-system interface as an hierarchic control interface. For example, a trace file system provides a pseudo file system as a front end to interact with core's TRACE service.

GENODE ⁄⁄ LABS

4.5.14. Loader

The loader session interface allows clients to dynamically create Genode subsystems to be hosted as children of a loader service. In contrast to a component that is spawning a new subsystem as an immediate child, a loader client has very limited control over the spawned subsystem. It can merely define the binaries and configuration to start, define the position where the loaded subsystem will appear on screen, and kill the subsystem. But it is not able to interfere with the operation of the subsystem during its lifetime.

Session creation   At session-creation time, the client defines the amount of memory to be used for the new subsystem as session quota. Once the session is established, the client equips the loader session with ROM modules that will be presented to the loaded subsystem. From the perspective of the subsystem, those ROM modules can be requested in the form of ROM sessions from its parent.

Visual integration of the subsystem   The loaded subsystem may implement a graphical user interface by creating a nitpicker session (Section 4.5.6). The loader responds to such a session request by providing a locally implemented session. The loader subordinates the nitpicker session of the loaded subsystem to a nitpicker view (called parent view) defined by the loader client. The loader client can use the loader session interface to position the view relative to the parent-view position. Thereby, the graphical user interface of the loaded subsystem can be seamlessly integrated with the user interface of the loader client.

Use case   The most illustrative use case is the execution of web-browser plugins where neither the browser trusts the plugin nor the plugin trusts the browser (Section 4.7.4).

GENODE ⧸ LABS

4.6. Component configuration

By convention, each component obtains its configuration in the form of a ROM module named "config". The ROM session for this ROM module is provided by the parent of the component. For example, for the init component, which is the immediate child of core, its "config" ROM module is provided by core's ROM service. Init, in turn, provides a different config ROM module to each of its children by a locally implemented ROM service per child.

4.6.1. Configuration format

In principle, being a mere ROM module, a component configuration can come in an arbitrary format. However, throughout Genode, there exists the convention to use XML as syntax and wrap the configuration within a `<config>` node. The definition of sub nodes of the configuration depends on the respective component.

4.6.2. Server-side policy selection

Servers that serve multiple clients may apply a different policy to each client. In general, the policy may be defined by the session arguments aggregated on the route of the session request as explained in Section 3.2.3. However, in the usual case, the policy is dictated by the common parent of client and server. In this case, the parent may propagate its policy as the server's configuration and deliver a textual label as session argument for each session requested at the server. The configuration contains a list of policies whereas the session label is used as a key to select the policy from the list. For example, the following snippet configures a RAM file system with different policies.

```
<config>
  <!-- constrain sessions according to their labels -->
  <policy label="noux -> root" root="/" />
  <policy label="noux -> home" root="/home/user" />
  <policy label="noux -> tmp"  root="/tmp" writeable="yes" />
</config>
```

Each time a session is created, the server matches the supplied session label against the configured policies. Only if a policy matches, the parameters of the matching policy come into effect. The way how the session label is matched against the policies depends on the implementation of the server. However, by convention, servers usually select the policy depending on the attributes `label`, `label_prefix`, and `label_suffix`. If present, the `label` attribute must perfectly match the session label whereby the suffix and prefix counterparts allow for partially matching the session label. If multiple `<policy>` nodes match at the server side, the most specific policy is selected. Exact matches are considered as most specific, prefixes as less specific, and suffixes as least

GENODE 🖎 LABS

specific. If multiple prefixes or suffixes match, the longest is considered as the most specific. If multiple policies have the same label, the selection is undefined. This is a configuration error.

4.6.3. Dynamic component reconfiguration at runtime

As described in Section 4.5.1, a ROM module can be updated during the lifetime of the ROM session. This principally enables a parent to dynamically reconfigure a child component without the need to restart it. If a component supports its dynamic reconfiguration, it installs a signal handler at its "config" ROM session. Each time, the configuration changes, the component will receive a signal. It responds to such a signal by obtaining the new version of the ROM module using the steps described in Section 4.5.1 and applying the new configuration.

GENODE LABS

4.7. Component composition

Genode provides a playground for combining components in many different ways. The best composition of components often depends on the goal of the system integrator. Among possible goals are the ease of use for the end user, the cost-efficient reuse of existing software, and good application performance. However, the most prominent goal is the mitigation of security risks. This section presents composition techniques that leverage Genode's architecture to dramatically reduce the trusted computing base of applications and to solve rather complicated problems in surprisingly easy ways.

The figures presented throughout this section use a simpler nomenclature than the previous sections. A component is depicted as box. Parent-child relationships are represented as light-gray arrows. A session between a client and a server is illustrated by a dashed arrow pointing to the server.
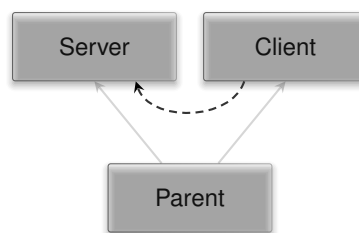
4.7.1. Sandboxing

The functionality of existing applications and libraries is often worth reusing or economically downright infeasible to reimplement. Examples are PDF rendering engines, libraries that support commonly used video and audio codecs, or libraries that decode hundreds of image formats.

However, code of such rich functionality is inherently complex and must be assumed to contain security flaws. This is empirically evidenced by the never ending stream of security exploits targeting the decoders of data formats. But even in the absence of bugs, the processing of data by third-party libraries may have unintended side effects. For example, a PDF file may contain code that accesses the file system, which the user of a PDF reader may not expect. By linking such a third-party library to a security-critical application, the application's security is seemingly traded against the functional value that the library offers.

Fortunately, Genode's architecture principally allows every component to encapsulate untrusted functionality in child components. So instead of directly linking a third-party library to an application, the application executes the library code in a dedicated sub component. By imposing a strict session-routing policy onto the component, the untrusted code is restricted to its sandbox. Figure 39 shows a video player as a practical example of this approach.

The video player uses the nitpicker GUI server to present a user interface with the graphical controls of the player. Furthermore, it has access to a media file containing

122

GENODE ⚝ LABS
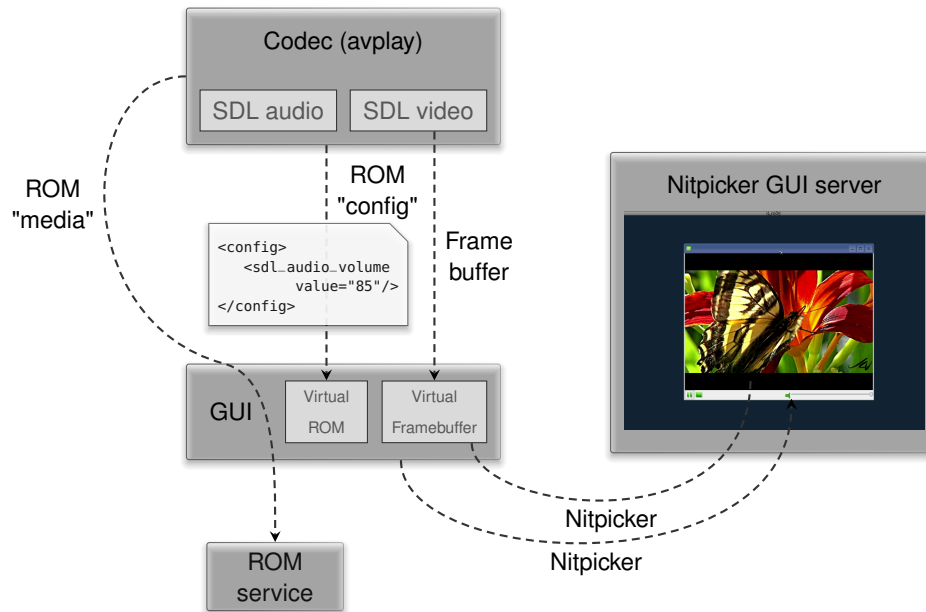
Figure 39: A video player executes the video and audio codecs inside a dedicated sandbox.

video and audio data. Instead of linking the media-codec library (libav) directly to the video-player application, it executes the codec as a child component. Thereby the application effectively restricts the execution environment of the codec to only those resources that are needed by the codec. Those resources are the media file that is handed out to the codec as a ROM module, a facility to output video frames in the form of a framebuffer session, and a facility to output an audio stream in the form of an audio-out session.

In order to reuse as much code as possible, the video player executes an existing example application called avplay that comes with the codec library as child component. The avplay example uses libSDL as back end for video and audio output and responds to a few keyboard shortcuts for controlling the video playback such as pausing the video. Because there exists a Genode version of libSDL, avplay can be executed as a Genode component with no modifications. This version of libSDL requests a framebuffer session (Section 4.5.5) and an audio-out session (Section 4.5.12) to perform the video and audio output. To handle user input, libSDL opens an input session (Section 4.5.4). Furthermore, it opens a ROM session for obtaining a configuration. This configuration parametrizes the audio back end of libSDL. Because avplay is a child of the video-player application, all those session requests are directed to the application. It is entirely up to the application how to respond to those requests. For accommodating the request for a frambuffer session, the application creates a second nitpicker session, configures a virtual framebuffer, and embeds this virtual framebuffer into its GUI. It keeps the nitpicker session capability for itself and merely hands out the virtual

GENODE ⫸ LABS

framebuffer's session capability to avplay. For accommodating the request for the input session, it hands out a capability to a locally-implemented input session. Using this input session, it becomes able to supply artificial input events to avplay. For example, when the user clicks on the play button of the application's GUI, the application would submit a sequence of press and release events to the input sessions, which appear to avplay as the keyboard shortcut for starting the playback. To let the user adjust the audio parameters of libSDL during playback, the video-player application dynamically changes the avplay configuration using the mechanism described in Section 4.6.3. As a response to a configuration update, libSDL's audio back end picks up the changed configuration parameters and adjusts the audio playback accordingly.

By sandboxing avplay as a child component of the video player, a bug in the video or audio codecs can no longer compromise the application. The execution environment of avplay is tailored to the needs of the codec. In particular, it does not allow the codec to access any files or the network. In the worst case, if avplay becomes corrupted, the possible damage is restricted to producing wrong video or audio frames but a corrupted codec can neither access any of the user's data nor can it communicate to the outside world.

4.7.2. Component-level and OS-level virtualization

The sandboxing technique presented in the previous section tailors the execution environment of untrusted third-party code by applying an application-specific policy to all session requests originating from the untrusted code. However, the tailoring of the execution environment by the parent can even go a step further by providing the all-encompassing virtualization of all services used by the child, including core's services such as RAM, PD, and CPU. This way, the parent can not just tailor the execution environment of a child but completely define all aspects of the child's execution. This clears the way for introducing custom operating-system interfaces at any position within the component tree, or for monitoring the behavior of subsystems.

Introducing a custom OS interface By implementing all session interfaces normally provided by core, a runtime environment becomes able to handle all low-level interactions of the child with core. This includes the allocation of memory using the RAM service, the spawning and controlling of threads using the CPU service, and the management of the child's address space using the PD service.

The noux runtime illustrated in Figure 40 is the canonical example of this approach. It appears as a Unix kernel to its children and thereby enables the use of Unix software on top of Genode. Normally, several aspects of Unix would contradict with Genode's architecture:

- The Unix system-call interface supports files and sockets as first-level citizens.

- There is no global virtual file system in Genode.

GENODE ⟋ LABS
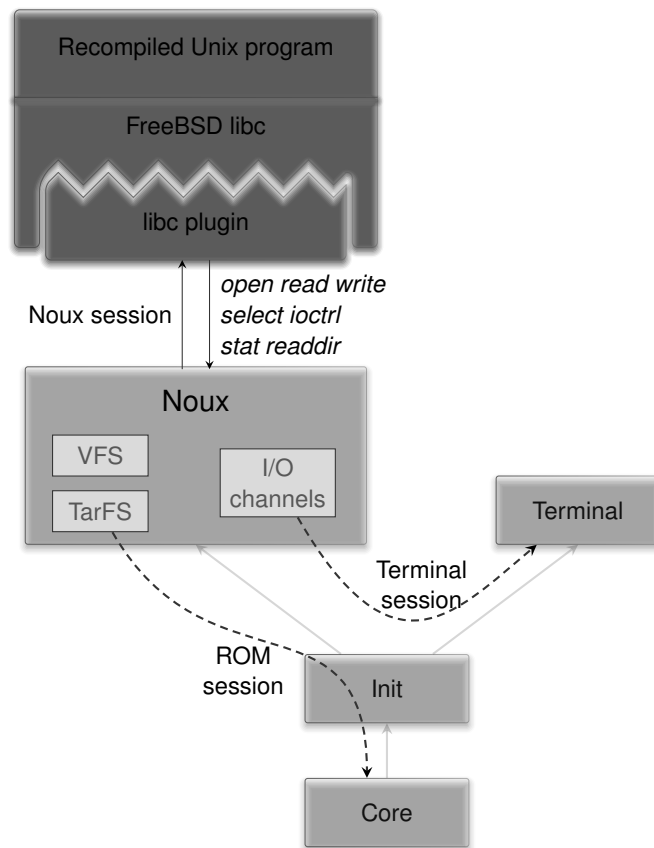
Figure 40: The Noux runtime provides a Unix-like interface to its children.

GENODE LABS
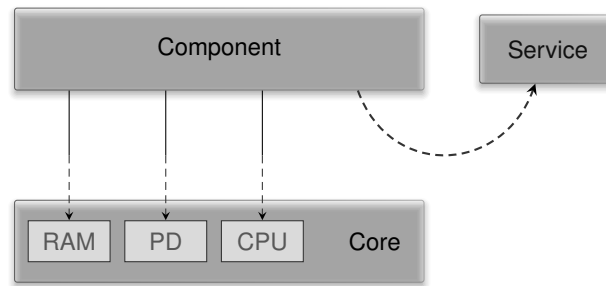
Figure 41: Each Genode component is created out of basic resources provided by core.

- Any Unix process can allocate memory as needed. There is no necessity for explicit assignment of memory resources to Unix processes.

- Processes are created by forking existing processes. The new process inherits the roles (in the form of open file descriptors) of the forking process.

Noux resolves these contradictions by providing the interfaces of core's low-level services alongside a custom RPC interface. By providing a custom noux session interface to its children, noux can accommodate all kinds of abstractions including the notion of files and sockets. Noux maintains a virtual file system that appears to be global among all the children of the noux instance. Since noux handles all the children's interaction with the RAM service, it can hand out memory allocations from a pool of memory shared among all children. Finally, because noux observes all the interactions of each child with the PD service, it is able to replay the address-space layout of an existing process to a new process when fork is called.

Monitoring the behavior of subsystems   Besides hosting arbitrary OS personalities as a subsystem, the interception of core's services allows for the all-encompassing monitoring of subsystems without the need for special support in the kernel. This is useful for failsafe monitoring or for user-level debugging.

As described in Section 3.5, any Genode component is created out of low-level resources in the form of sessions provided by core. Those sessions include at least a RAM session, a CPU session, and a PD session as depicted in Figure 41. In addition to those low-level sessions, the component may interact with sessions provided by other components.

For debugging a component, a debugger would need a way to inspect the internal state of the component. As the complete internal state is usually known by the OS kernel only, the traditional approach to user-level debugging is the introduction of a debugging interface into the kernel. For example, Linux has the ptrace mechanism and several microkernels of the L4 family come with built-in kernel debuggers. Such a debugging interface, however, introduces security risks. Besides increasing the com-

126
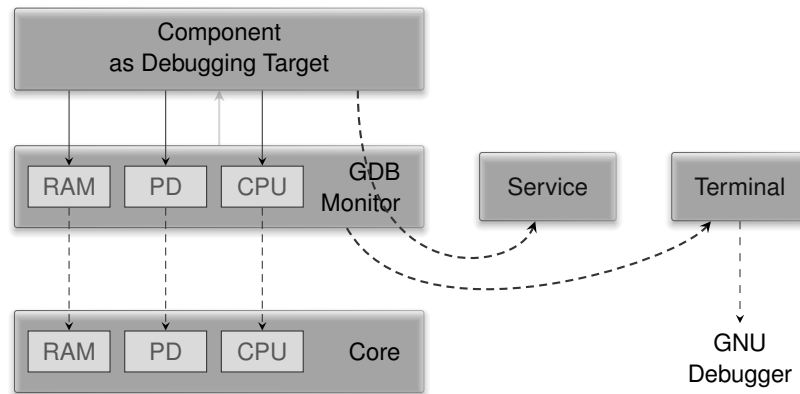
Figure 42: By intercepting all sessions to core's services, a debug monitor obtains insights into the internal state of its child component. The debug monitor, in turn, is controlled from a remote debugger.

plexity of the kernel, access to the kernel's debugging mechanisms needs to be strictly subjected to a security policy. Otherwise any program could use those mechanisms to inspect or manipulate other programs. Most L4 kernels usually exclude debugging features in production builds altogether.

In a Genode system, the component's internal state is represented in the form of core sessions. Hence, by intercepting those sessions of a child, a parent can monitor all interactions of the child with core and thereby record the child's internal state. Figure 42 shows a scenario where a debug monitor executes a component (debugging target) as a child while intercepting all sessions to core's services. The interception is performed by providing custom implementations of core's session interfaces as locally implemented services. Under the hood, the local services realize their functionality using actual core sessions. But by sitting in the middle between the debugging target and core, the debug monitor can observe the target's internal state including the memory content, the virtual address-space layout, and the state of all threads running inside the component. Furthermore, since the debug monitor is in possession of all the session capabilities of the debugging target, it can *manipulate* it in arbitrary ways. For example, it can change thread states (e. g., pausing the execution or enable single-stepping) and modify the memory content (e. g., inserting breakpoint instructions). The figure shows that those debugging features can be remotely controlled over a terminal connection.

Using this form of component-level virtualization, a problem that used to require special kernel additions in traditional operating systems can be solved via Genode's regular interfaces. Furthermore, Figure 43 shows that by combining the solution with OS-level virtualization, the connection to a remote debugger can actually be routed to an on-target instance of the debugger, thereby enabling on-target debugging.

GENODE ⫛ LABS
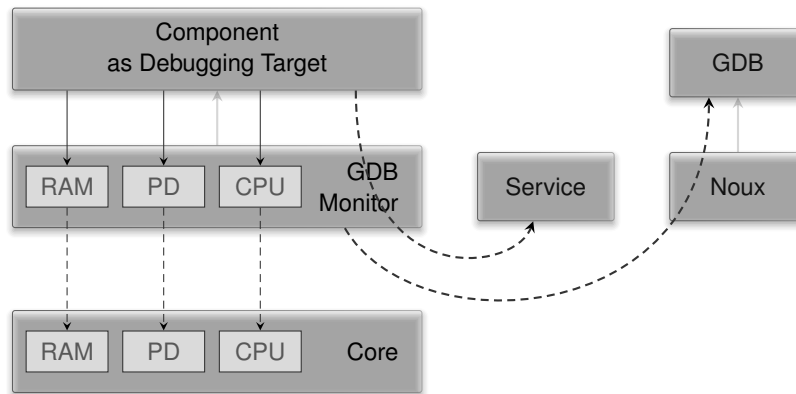
Figure 43: The GNU debugger is executed within a dedicated noux instance, thereby providing an on-target debugging facility.

4.7.3. Interposing individual services

The design of Genode's fundamental services, in particular resource multiplexers, is guided by the principle of minimalism. Because such components are security critical, complexity must be avoided. Functionality is added to such components only if it cannot be provided outside the component.

However, components like the nitpicker GUI server are often confronted with feature requests. For example, users may want to move a window on screen by dragging the window's title bar. Because nitpicker has no notion of windows or title bars, such functionality is not supported. Instead, nitpicker moves the burden to implement window decorations to its clients. However, this approach sacrifices functionality that is taken for granted on modern graphical user interfaces. For example, the user may want to switch the application focus using a keyboard shortcut or perform window operations and the interactions with virtual desktops in a consistent way. If each application implemented the functionality of virtual desktops individually, the result would hardly be usable. For this reason, it is tempting to move window-management functionality into the GUI server and to accept the violation of the minimalism principle.

The nitpicker GUI server is not the only service challenged by feature requests. The problem is present even at the lowest-level services provided by core. Core's region-map mechanism is used to manage the virtual address spaces of components via their respective PD sessions. When a dataspace is attached to a region map, the region map picks a suitable virtual address range where the dataspace will be made visible in the virtual address space. The allocation strategy depends on several factors such as alignment constraints and the address range that fits best. But eventually, it is deterministic. This contradicts the common wisdom that address spaces shall be randomized. Hence core's PD service is challenged with the request for adding address-space randomization as a feature. Unfortunately, the addition of such a feature into core raises two

GENODE LABS
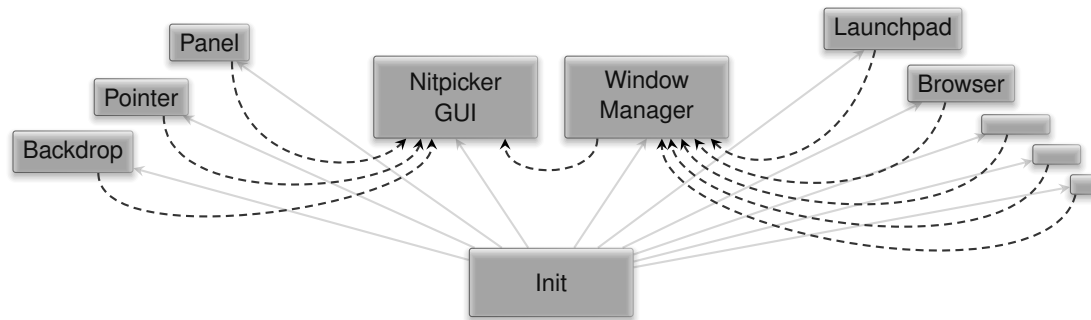
Figure 44: The nitpicker GUI accompanied with a window manager that interposes the nit-
picker session interface for the applications on the right. The applications on the
left are still able to use nitpicker directly and thereby avoid the complexity added by
the window manager.

issues. First, core would need to have a source of good random numbers. But core does
not contain any device drivers where to draw entropy from. With weak entropy, the
randomization might be not random enough. In this case, the pretension of a security
mechanism that is actually ineffective may be worse than not having it in the first place.
Second, the feature would certainly increase the complexity of core. This is acceptable
for components that potentially benefit from the added feature, such as outward-facing
network applications. But the complexity eventually becomes part of the TCB of all
components including those that do not benefit from the feature.

The solution to those kind of problems is the enrichment of existing servers by inter-
posing their sessions. Figure 44 shows a window manager implemented as a separate
component outside of nitpicker. Both the nitpicker GUI server and the window man-
ager provide the nitpicker session interface. But the window manager enriches the
semantics of the interface by adding window decorations and a window-layout policy.
Under the hood, the window manager uses the real nitpicker GUI server to implement
its service. From the application's point of view, the use of either service is transparent.
Security-critical applications can still be routed directly to the nitpicker GUI server. So
the complexity of the window manager comes into effect only for those applications
that use it.

The same approach can be applied to the address-space randomization problem. A
component with access to good random numbers may provide a randomized version
of core's PD service. Outward-facing components can benefit from this security feature
by having their PD session requests routed to this component instead of core.

4.7.4. Ceding the parenthood

When using a shell to manage subsystems, the complexity of the shell naturally be-
comes a security risk. A shell can be a text-command interpreter, a graphical desktop
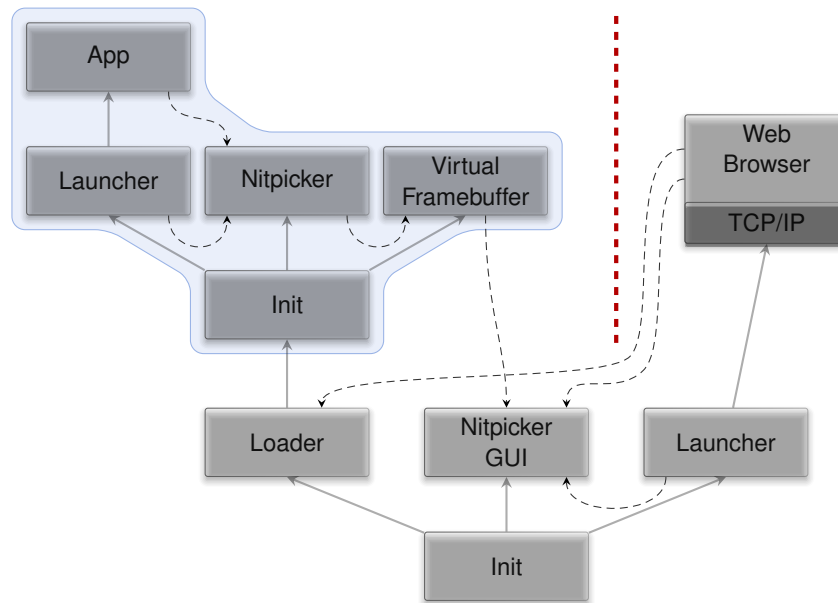
Figure 45: A web browser spawns a plugin by ceding the parenthood of the plugin to the trusted
loader service.

shell, a web browser that launches subsystems as plugins, or a web server that pro-
vides a remote administration interface. What all those kinds of shells have in common
is that they contain an enormous amount of complexity that can be attributed to conve-
nience. For example, a textual shell usually depends on libreadline, ncurses, or similar
libraries to provide a command history and to deal with the peculiarities of virtual text
terminals. A graphical desktop shell is even worse because it usually depends on a
highly complex widget toolkit, not to mention using a web browser as a shell. Un-
fortunately, the functionality provided by these programs cannot be dismissed as it is
expected by the user. But the high complexity of the convenience functions funda-
mentally contradicts the security-critical role of the shell as the common parent of all
spawned subsystems. If the shell gets compromised, all the spawned subsystems will
suffer.

The risk of such convoluted shells can be mitigated by moving the parent role for
the started subsystems to another component, namely a loader service. In contrast
to the shell, which should be regarded as untrusted due it its complexity, the loader
is a small component that is orders of magnitude less complex. Figure 45 shows a
scenario where a web browser is used as a shell to spawn a Genode subsystem. Instead
of spawning the subsystem as the child of the browser, the browser creates a loader
session. Using the loader-session interface described in Section 4.5.14, it can initially
import the to-be-executed subsystem into the loader session and kick off the execution
of the subsystem. However, once the subsystem is running, the browser can no longer

GENODE 🖎 LABS

interfere with the subsystem's operation. So security-sensitive information processed within the loaded subsystem are no longer exposed to the browser. Still, the lifetime of the loaded subsystem depends on the browser. If it decides to close the loader session, the loader will destroy the corresponding subsystem.

By ceding the parenthood to a trusted component, the risks stemming from the complexity of various kinds of shells can be mitigated.

4.7.5. Publishing and subscribing

All the mechanisms for transferring data between components presented in Section 3.6 have in common that data is transferred in a peer-to-peer fashion. A client transfers data to a server or vice versa. However, there are situations where such a close coupling of both ends of communication is not desired. In multicast scenarios, the producer of information desires to propagate information without the need to interact (or even depend on a handshake) with each individual recipient. Specifically, a component might want to publish status information about itself that might be useful for other components. For example, a wireless-networking driver may report the list of detected wireless networks along with their respective SSIDs and reception qualities such that a GUI component can pick up the information and present it to the user. Each time, the driver detects a change in the ether, it wants to publish an updated version of the list. Such a scenario could principally be addressed by introducing a use-case-specific session interface, i. e., a "wlan-list" session. But this approach has two disadvantages.

1. It forces the wireless driver to play an additional server role. Instead of pushing information anytime at the discretion of the driver, the driver has to actively support the pulling of information from the wlan-list client. This is arguably more complex.

2. The wlan-list session interface ultimately depends on the capabilities of the driver implementation. If an alternative wireless driver is able to supplement the list with further details, the wlan-list session interface of the alternative driver might look different. As a consequence, the approach is likely to introduce many special-purpose session interfaces. This contradicts with the goal to promote the composability of components as stated at the beginning of Section 4.5.

As an alternative to introducing special-purpose session interfaces for addressing the scenarios outlined above, two existing session interfaces can be combined, namely ROM and report.

Report-ROM server   The *report-rom* server is both a ROM service and a report service. It acts as an information broker between information providers (clients of the report service) and information consumers (clients of the ROM service).

GENODE ⬙ LABS

To propagate its internal state to the outside, a component creates a report session. From the client's perspective, the posting of information via the report session's *submit* function is a fire-and-forget operation, similar to the submission of a signal. But in contrast to a signal, which cannot carry any payload, a report is accompanied with arbitrary data. For the example above, the wireless driver would create a report session. Each time, the list of networks changes, it would submit an updated list as a report to the report-ROM server.

The report-ROM server stores incoming reports in a database using the client's session label as key. Therefore, the wireless driver's report will end up in the database under the name of the driver component. If one component wishes to post reports of different kinds, it can do so by extending the session label by a component-provided label suffix supplied as session-construction argument (Section 4.5.2). The memory needed as the backing store for the report at the report-ROM server is accounted to the report client via the session-quota mechanism described in Section 3.3.2.

In its role of a ROM service, the report-ROM server hands out the reports stored in its database as ROM modules. The association of reports with ROM sessions is based on the session label of the ROM client. The configuration of the report-ROM server contains a list of policies as introduced in Section 4.6.2. Each policy entry is accompanied with a corresponding key into the report database.

When a new report comes in, all ROM clients that are associated with the report are informed via a ROM-update signal (Section 4.5.1). Each client can individually respond to the signal by following the ROM-module update procedure and thereby obtain the new version of the report. From the client's perspective, the origin of the information is opaque. It cannot decide whether the ROM module is provided by the report-ROM server or an arbitrary other ROM service.

Coming back to the wireless-driver example, the use of the report-ROM server effectively decouples the GUI application from the wireless driver. This has the following benefits:

- The application can be developed and tested with an arbitrary ROM server supplying an artificially created list of networks.

- There is no need for the introduction of a special-purpose session interface between both components.

- The wireless driver can post state updates in an intuitive fire-and-forget way without playing an additional server role.

- The wireless driver can be restarted without affecting the application.

Poly-instantiation of the report-ROM mechanism The report-ROM server is a canonical example of a protocol stack (Section 4.2). It performs a translation between the report-session interface and the ROM-session interface. Being a protocol stack, it

GENODE ⫸ LABS

can be instantiated any number of times. It is up to the system integrator whether to use one instance for gathering the reports of many report clients, or to instantiate multiple report-ROM servers. Taken to the extreme, one report-ROM server could be instantiated per report client. The routing of ROM-session requests restricts the access of the ROM clients to the different instances. Even in the event that the report-ROM server is compromised, the policy for the information flows between the producers and consumers of information stays in effect.

4.7.6. Enslaving services

In the scenarios described in the previous sections, the relationships between clients and servers have been one of the following:

- The client is a sibling of the server within the component tree, or

- The client is a child of a parent that provides a locally-implemented service to its child.

However, the Genode architecture allows for a third option: The parent can be a client of its own child. Given the discussion in Section 3.2.4, this arrangement looks counter-intuitive at first because the discussion concluded that a client has to trust the server with respect to the client's liveliness. Here, a call to the server would be synonymous to a call to the child. Even though the parent is the owner of the child, it would make itself dependent on the child, which is generally against the interest of the parent.

That said, there is a plausible case where the parent's trust in a child is justified: If the parent uses an existing component like a 3rd-party library. When calling code of a 3rd-party library, the caller implicitly agrees to yield control to the library and trusts the called function to return at some point. The call of a service that is provided by a child corresponds to such a library call.

By providing the option to host a server as a child component, Genode's architecture facilitates the use of arbitrary server components in a library-like fashion. Because the server performs a useful function but is owned by its client, it is called *slave*. An application may aggregate existing protocol-stack components as slaves without the need to incorporate the code of the protocol stacks into the application. For example, by enslaving the report-ROM server introduced in Section 4.7.5, an application becomes able to use it as a local publisher-subscriber mechanism. Another example would be an application that aggregates an instance of the nitpicker GUI server for the sole purpose of composing an image out of several source images. When started, the nitpicker slave requests a framebuffer and an input session. The application responds to these requests by handing out locally-implemented sessions so that the output of the nitpicker slave becomes visible to the application. To perform the image composition, the application creates a nitpicker session for each source image and supplies the image data to the virtual framebuffer of the respective session. After configuring nitpicker views according

GENODE ⚞ LABS

to the desired layout of the final image, the application obtains the composed image from nitpicker's framebuffer.

Note that by calling the slave, the parent does not need to trust the slave with respect to the integrity and confidentiality of its internal state (see the discussion in Section 3.2.4). By performing the call, only the liveliness of the parent is potentially affected. If not trusting the slave to return control once called, the parent may take special precautions: A watchdog thread inside the parent could monitor the progress of the slave and cancel the call after the expiration of a timeout.

GENODE LABS

5. Development

The Genode OS framework is accompanied by a scalable build system and tooling infrastructure that is designed for the creation of highly modular and portable systems software. Understanding the underlying concepts is important for leveraging the full potential of the framework. This chapter complements Chapter 2 with the explanation of the coarse-grained source-tree structure (Section 5.1), the integration of 3rd-party software (Section 5.2), the build system (Section 5.3), and system-integration tools (Section 5.4). Furthermore, it describes the project's development process in Section 5.5.

GENODE 🖉 LABS

5.1. Source-code repositories

As briefly introduced in Section 2.2, Genode's source tree is organized in the form of several source-code repositories. This coarse-grained modularization of the source code has the following benefits:

- Source codes of different concerns remain well separated. For example, the platform-specific code for each base platform is located in a dedicated *base-<platform>* repository.

- Different abstraction levels and features of the system can be maintained in different source-code repositories. Whereby the source code contained in the *os* repository is free from any dependency from 3rd-party software, the components hosted in the *libports* repository are free to use foreign code.

- Custom developments and experimental features can be hosted in dedicated source-code repositories, which do not interfere with Genode's source tree. Such a custom repository can be managed independently from Genode using arbitrary revision-control systems.

The build-directory configuration defines the set of repositories to incorporate into the build process. At build time, the build system overlays the directory structures of all selected repositories to form a single logical source tree. The selection of source-code repositories ultimately defines the view of the build system on the source tree.

Note that the order of the repositories as configured in the build configuration (in *etc/build.conf*) is important. Front-most repositories shadow subsequent repositories. This makes the repository mechanism a powerful tool for tweaking existing repositories: By adding a custom repository in front of another one, customized versions of single files (e. g., header files or target description files) can be supplied to the build system without changing the original repository.

Each source-code repository has the principle structure shown in Table 1.

GENODE ⟋⟍ LABS

Directory	Description
doc/	Documentation, specific for the repository
etc/	Default configuration for the build process
mk/	Build-system supplements
include/	Globally visible header files
src/	Source codes and target build descriptions
lib/mk/	Library build descriptions
lib/import/	Library import descriptions
ports/	Port descriptions of 3rd-party software

Table 1: Structure of a source-code repository. Depending on the repository, only a subset of those directories may be present.

GENODE LABS

5.2. Integration of 3rd-party software

Downloaded 3rd-party source code resides outside of the actual repository at the central *<genode-dir>/contrib/* directory. This structure has the following benefits over hosting 3rd-party source code along with Genode's genuine source code:

- Working with grep within the repositories works very efficient because downloaded and extracted 3rd-party code is not in the way. Such code resides next to the repositories.

- Storing all build directories and downloaded 3rd-party source code somewhere outside the Genode source tree, e. g., on different disk partitions, can be easily accomplished by creating symbolic links for the *build/* and *contrib/* directories.

The *contrib/* directory is managed using the tools at *<genode-dir>/tool/ports/*.

Obtain a list of available ports

```
tool/ports/list
```

Download and install a port

```
tool/ports/prepare_port <port-name>
```

The *prepare_port* tool scans all repositories under *repos/* for the specified port and installs the port into *contrib/*. Each version of an installed port resides in a dedicated subdirectory within the *contrib/* directory. The port-specific directory is called port directory. It is named *<port-name>-<fingerprint>*. The *<fingerprint>* uniquely identifies the version of the port (it is a SHA1 hash of the ingredients of the port). If two versions of the same port are installed, each of them will have a different fingerprint. So they end up in different directories.

Within a source-code repository, a port is represented by two files, a *<port-name>.port* and a *<port-name>.hash* file. Both files reside at the *ports/* subdirectory of the corresponding repository. The *<port-name>.port* file is the port description, which declares the ingredients of the port, e. g., the archives to download and the patches to apply. The *<port-name>.hash* file contains the fingerprint of the corresponding port description, thereby uniquely identifying a version of the port as expected by the checked-out Genode version.

For step-by-step instructions on how to add a port using the mechanism, please refer to the porting guide:

Genode Porting Guide

http://genode.org/documentation/developer-resources/porting

GENODE ⟋ LABS

5.3. Build system

5.3.1. Build directories

The build system is supposed to never touch the source tree. The procedure of building components and integrating them into system scenarios is performed within a distinct *build directory*. One build directory targets a specific kernel and hardware platform. Because the source tree is decoupled from the build directory, one source tree can have many different build directories associated, each targeted at a different platform.

The recommended way for creating a build directory is the use of the *create_builddir* tool located at *<genode-dir>/tool/*. The tool prints usage information along with a list of supported base platforms when started without arguments. For creating a new build directory, one of the listed target platforms must be specified. Furthermore, the location of the new build directory can optionally be specified via the `BUILD_DIR=` argument. For example:

```
cd <genode-dir>
./tool/create_builddir linux_x86 BUILD_DIR=/tmp/build.linux_x86
```

This command creates a new build directory for the Linux/x86 platform at */tmp/build.linux_x86/*. For the basic operations available from within the build directory, please refer to Section 2.3.

Configuration Each build directory contains a *Makefile*, which is a symbolic link to *tool/builddir/build.mk*. The makefile is the front end of the build system and not supposed to be edited. Besides the makefile, there is an *etc/* subdirectory that contains the build-directory configuration. For most platforms, there exists merely a single *build.conf* file, which defines the source-code repositories to be incorporated into the build process along with the parameters for the run tool explained in Section 5.4.1.

The selection of source-code repositories is defined by the REPOSITORIES declaration, which contains a list of directories. The *etc/build.conf* file as found in a freshly created build directory is preconfigured to select the source-code repositories *base-<platform>*, *base*, *os*, and *demo*. There are a number of commented-out lines that can be uncommented for enabling additional repositories.

Cleaning To remove all but kernel-related generated files, use

```
make clean
```

To remove all generated files, use

```
make cleanall
```

GENODE ⬈ LABS

Both `clean` and `cleanall` won't remove any files from the *bin/* subdirectory. This makes the *bin/* a safe place for files that are unrelated to the build process, yet are required for the integration stage, e. g., binary data.

Controlling the verbosity   To understand the inner workings of the build process in more detail, you can tell the build system to display each directory change by specifying

```
make VERBOSE_DIR=
```

If you are interested in the arguments that are passed to each invocation of `make`, you can make them visible via

```
make VERBOSE_MK=
```

Furthermore, you can observe each single shell-command invocation by specifying

```
make VERBOSE=
```

Of course, you can combine these verboseness toggles for maximizing the noise.

5.3.2. Target descriptions

Each build target is represented by a corresponding *target.mk* file within the *src/* subdirectory of a source-code repository. This file declares the name of the target, the source codes to be incorporated into the target, and the libraries the target depends on. The build system evaluates target descriptions using *make*. Hence, the syntax corresponds to the syntax of makefiles and the principle functionality of make is available for *target.mk* files. For example, it is possible to define custom rules as done in Section 5.3.5.

Target declarations

TARGET is the name of the binary to be created. This is the only **mandatory variable** to be defined in each *target.mk* file.

LIBS is the list of libraries that are used by the target.

SRC_CC contains the list of `.cc` source files. The default search location for source codes is the directory where the *target.mk* file resides.

SRC_C contains the list of `.c` source files.

SRC_S contains the list of assembly `.s` source files.

GENODE ⩘ LABS

SRC_BIN contains binary data files to be linked to the target.

INC_DIR is the list of include search locations. Directories should always be appended by using +=.

REQUIRES expresses the requirements that must be satisfied in order to build the target. More details about the underlying mechanism is provided by Section 5.3.4.

CC_OPT contains additional compiler options to be used for .c as well as for .cc files.

CC_CXX_OPT contains additional compiler options to be used for the C++ compiler only.

CC_C_OPT contains additional compiler options to be used for the C compiler only.

EXT_OBJECTS is a list of external objects or libraries. This declaration is merely used for interfacing Genode with legacy software components.

Specifying search locations When specifying search locations for header files via the INC_DIR variable or for source files via vpath, the use of relative pathnames is illegal. Instead, the following variables can be used to reference locations within the source-code repository where the target resides:

REP_DIR is the base directory of the target's source-code repository. Normally, specifying locations relative to the base of the repository is rarely used by *target.mk* files but needed by library descriptions.

PRG_DIR is the directory where the *target.mk* file resides. This variable is always to be used when specifying a relative path.

$(call select_from_repositories,path/relative/to/repo) This function returns the absolute path for the given repository-relative path by looking at all source-code repositories in their configured order. Hereby, it is possible to access files or directories that are outside the target's source-code repository.

$(call select_from_ports,<port-name>) This function returns the absolute path for the *contrib* directory of the specified *<port-name>*. The contrib directory is located at *<genode-dir>/contrib/<port-name>-<fingerprint>* whereby *<fingerprint>* uniquely identifies the version of the port as expected by the current state of the Genode source tree.

5.3.3. Library descriptions

In contrast to target descriptions that are scattered across the whole source tree, library descriptions are located at the central place *lib/mk*. Each library corresponds to a *<libname>.mk* file. The base of the description file is the name of the library. Therefore,

GENODE 〽 LABS

no `TARGET` variable needs to be defined. The location of source-code files is usually defined relative to `$(REP_DIR)`. Library-description files support the following additional declaration:

SHARED_LIB = yes declares that the library should be built as a shared object rather than a static library. The resulting object will be called *<libname>.lib.so*.

5.3.4. Platform specifications

Building components for different platforms likely implicates that portions of code are tied to certain aspects of the target platform. For example, target platforms may differ in the following respects:

- The API of the used kernel,

- The hardware architecture such as x86, ARMv7,

- Certain hardware facilities such as a custom device, or

- Other considerations such as software license requirements.

Each of those aspects may influence the build process in different ways. The build system provides a generic mechanism to steer the build process according to such aspects. Each aspect is represented by a tag called *spec value*. Any platform targeted by Genode can be characterized by a set of such spec values.

The **developer** of a software component knows the constraints of his software and thus specifies these requirements in the build-description file of the component. The **system integrator** defines the platform the software will be built for by specifying the targeted platform in the SPECS declaration in the build directory's *etc/specs.conf* file. In addition to the (optional) *etc/specs.conf* file within the build directory, the build system incorporates the first *etc/specs.conf* file found in the enabled repositories. For example, for a `linux_x86` build directory, the *base-linux/etc/specs.conf* file is used by default. The build directory's `specs.conf` file can still be used to extend the SPECS declarations, for example to enable special features.

Each *<spec>* in the SPECS variable instructs the build system to

- Include the make-rules of a corresponding *base/mk/spec/<specname>.mk* file. This enables the customization of the build process for each platform.

- Search for *<libname>.mk* files in the *lib/mk/spec/<specname>/* subdirectory. This way, alternative implementations of one and the same library interface can be selected depending on the platform specification.

Before a target or library gets built, the build system checks if the REQUIRES entries of the build description file are satisfied by entries of the SPECS variable. The compilation is executed only if each entry in the REQUIRES variable is present in the SPECS variable as supplied by the build directory configuration.

GENODE ⚡ LABS

5.3.5. Building tools to be executed on the host platform

Sometimes, software requires custom tools that are used to generate source code or other ingredients for the build process, for example IDL compilers. Such tools won't be executed on top of Genode but on the host platform during the build process. Hence, they must be compiled with the tool chain installed on the host, not the Genode tool chain.

The build system accommodates the building of such host tools as a side effect of building a library or a target. Even though it is possible to add the tool-compilation step to a regular build description file, it is recommended to introduce a dedicated pseudo library for building such tools. This way, the rules for building host tools are kept separate from rules that refer to regular targets. By convention, the pseudo library should be named *<package>_host_tools* and the host tools should be built at *<build-dir>/tool/<package>/* where *<package>* refers to the name of the software package the tool belongs to, e. g., qt5 or mupdf. To build a tool named *<tool>*, the pseudo library contains a custom make rule like the following:

```
$(BUILD_BASE_DIR)/tool/<package>/<tool>:
    $(MSG_BUILD)$(notdir $@)
    $(VERBOSE)mkdir -p $(dir $@)
    $(VERBOSE)...build commands...
```

To let the build system trigger the rule, add the custom target to the `HOST_TOOLS` variable:

```
HOST_TOOLS += $(BUILD_BASE_DIR)/tool/<package>/<tool>
```

Once the pseudo library for building the host tools is in place, it can be referenced by each target or library that relies on the respective tools via the `LIBS` declaration. The tool can be invoked by referring to `$(BUILD_BASE_DIR)/tool/<package>/tool`.

For an example of using custom host tools, please refer to the mupdf package found within the libports repository. During the build of the mupdf library, two custom tools fontdump and cmapdump are invoked. The tools are built via the *lib/mk/mupdf_host_tools.mk* library description file. The actual mupdf library (*lib/mk/mupdf.mk*) has the pseudo library `mupdf_host_tools` listed in its `LIBS` declaration and refers to the tools relative to `$(BUILD_BASE_DIR)`.

5.3.6. Building 3rd-party software

The source code of 3rd-party software is managed by the mechanism presented in Section 5.2. Once prepared, such source codes resides in a subdirectory of *<genode-dir>/contrib/*.

GENODE LABS

If the build system encounters a target that incorporates ported source code (that is, a build-description file that calls the `select_from_ports` function), it looks up the respective *<port-name>.hash* file in the repositories as specified in the build configuration. The fingerprint found in the hash file is used to construct the path to the port directory under *contrib/*. If that lookup fails, a meaningful error is printed. Any number of versions of the same port can be installed at the same time. I.e., when switching Git branches that use different versions of the same port, the build system automatically finds the right port version as expected by the currently active branch.

GENODE LABS

5.4. System integration and automated testing

Genode's portability across kernels and hardware platforms is one of the prime features of the framework. However, each kernel or hardware platform requires different considerations when it comes to system configuration, integration, and booting. When using a particular kernel, profound knowledge about the boot concept and the kernel-specific tools is required. To streamline the testing of system scenarios across the many different supported kernels and hardware platforms, the framework is equipped with tools that relieve the system integrator from these peculiarities.

5.4.1. Run tool

The centerpiece of the system-integration infrastructure is the so-called run tool. Directed by a script (run script), it performs all the steps necessary to test a system scenario. Those steps are:

1. **Building** the components of a scenario

2. **Configuration** of the init component

3. Assembly of the **boot directory**

4. Creation of the **boot image**

5. **Powering-on** the test machine

6. **Loading** of the boot image

7. Capturing the **LOG output**

8. **Validation** of the scenario's behavior

9. **Powering-off** the test machine

Each of those steps depends on various parameters such as the used kernel, the hardware platform used to execute the scenario, the way the test hardware is connected to the test infrastructure (e. g., UART, AMT, JTAG, network), the way the test hardware is powered or reseted, or the way of how the scenario is loaded into the test hardware. To accommodate the variety of combinations of these parameters, the run tool consists of an extensible library of modules. The selection and configuration of the modules is expressed in the run-tool configuration. The following types of modules exist:

boot-dir modules These modules contain the functionality to populate the boot directory and are specific to each kernel. It is mandatory to always include the module corresponding to the used kernel.

(the available modules are: linux, hw, okl4, fiasco, pistachio, nova, sel4, foc)

GENODE ⧄ LABS

image modules   These modules are used to wrap up all components used by the run script in a specific format and thereby prepare them for execution. Depending on the used kernel, different formats can be used. With these modules, the creation of ISO and disk images is also handled.

(the available modules are: uboot, disk, iso)

load modules   These modules handle the way the components are transfered to the target system. Depending on the used kernel there are various options to pass on the components. For example, loading from TFTP or via JTAG is handled by the modules of this category.

(the available modules are: tftp, jtag, fastboot)

log modules   These modules handle how the output of a currently executed run script is captured.

(the available modules are: qemu, linux, serial, amt)

power_on modules   These modules are used for bringing the target system into a defined state, e. g., by starting or rebooting the system.

(the available modules are: qemu, linux, softreset, powerplug, amt)

power_off modules   These modules are used for turning the target system off after the execution of a run script.

(the available modules are: powerplug)

Each module has the form of a script snippet located under the *tool/run/<step>/* directory where *<step>* is a subdirectory named after the module type. Further instructions about the use of each module (e. g., additional configuration arguments) can be found in the form of comments inside the respective script snippets. Thanks to this modular structure, an extension of the tool kit comes down to adding a file at the corresponding module-type subdirectory. This way, custom work flows (such as tunneling JTAG over SSH) can be accommodated fairly easily.

5.4.2. Run-tool configuration examples

To execute a run script, a combination of modules may be used. The combination is controlled via the RUN_OPT declaration contained in the build directory's *etc/build.conf* file. The following examples illustrate the selection and configuration of different run modules:

Executing NOVA in Qemu

GENODE ⬙ LABS

```
RUN_OPT = --include boot_dir/nova \
          --include power_on/qemu --include log/qemu --include image/iso
```

By including `boot_dir/nova`, the run tool assembles a boot directory equipped with a boot loader and a boot-loader configuration that is able to bootstrap the NOVA kernel. The combination of the modules `power_on/qemu` and `log/qemu` prompts the run tool to spawn the Qemu emulator with the generated boot image and fetch the log output of the emulated machine from its virtual comport. The specification of `image/iso` tells the run tool to use a bootable ISO image as a boot medium as opposed to a disk image.

Executing NOVA on a real x86 machine using AMT   The following example uses Intel's advanced management technology (AMT) to remotely reset a physical target machine (`power_on/amt`) and capture the serial output over network (`log/amt`). In contrast to the example above, the system scenario is supplied via TFTP (`load/tftp`). Note that the example requires a working network-boot setup including a TFTP server, a DHCP server, and a PXE boot loader.

```
RUN_OPT = --include boot_dir/nova \
          --include power_on/amt \
                --power-on-amt-host 10.23.42.13 \
                --power-on-amt-password 'foo!' \
          --include load/tftp \
                --load-tftp-base-dir /var/lib/tftpboot \
                --load-tftp-offset-dir /x86 \
          --include log/amt \
                --log-amt-host 10.23.42.13 \
                --log-amt-password 'foo!'
```

If the test machine has a comport connection to the machine where the run tool is executed, the `log/serial` module may be used instead of `log/amt`:

```
 --include log/serial --log-serial-cmd 'picocom -b 115200 /dev/ttyUSB0'
```

Executing base-hw on a Raspberry Pi   The following example boots a system scenario based on the base-hw kernel on a Raspberry Pi that is powered via a USB-controllable power plug. The Raspberry Pi is connected to a JTAG debugger, which is used to load the system image onto the device.

GENODE LABS

```
RUN_OPT = --include boot_dir/hw \
          --include power_on/powerplug \
                --power-on-powerplug-ip 10.23.42.5 \
                --power-on-powerplug-user admin \
                --power-on-powerplug-password secret \
                --power-on-powerplug-port 1 \
          --include power_off/powerplug \
                --power-off-powerplug-ip 10.23.42.5 \
                --power-off-powerplug-user admin \
                --power-off-powerplug-password secret \
                --power-off-powerplug-port 1 \
          --include load/jtag \
          --load-jtag-debugger \
              /usr/share/openocd/scripts/interface/flyswatter2.cfg \
          --load-jtag-board \
              /usr/share/openocd/scripts/interface/raspberrypi.cfg \
          --include log/serial \
                --log-serial-cmd 'picocom -b 115200 /dev/ttyUSB0'
```

5.4.3. Meaningful default behaviour

The `create_builddir` tool introduced in Section 2.3 equips a freshly created build directory with a meaningful default configuration that depends on the selected platform. For example, when creating a build directory for the Linux base platform, RUN_OPT is initially defined as

```
RUN_OPT = --include boot_dir/linux \
          --include power_on/linux --include log/linux
```

5.4.4. Run scripts

Using run scripts, complete system scenarios can be described in a concise and kernel-independent way. As described in Section 2.4, a run script can be used to integrate and test-drive the scenario directly from the build directory. The best way to get acquainted with the concept is by reviewing the run script for the hello-world example presented in Section 2.5.4. It performs the following steps:

1. Building the components needed for the system using the `build` command. This command instructs the build system to compile the targets listed in the brace block. It has the same effect as manually invoking `make` with the specified argument from within the build directory.

GENODE ⬘ LABS

2. Creating a new boot directory using the `create_boot_directory` command. The integration of the scenario is performed in a dedicated directory at *<build-dir>/var/run/<run-script-name>/*. When the run script is finished, this boot directory will contain all components of the final system.

3. Installing the configuration for the init component into the boot directory using the `install_config` command. The argument to this command will be written to a file called `config` within the boot directory. It will eventually be loaded as boot module and made available by core's ROM service to the init component. The configuration of init is explained in Chapter 6.

4. Creating a bootable system image using the `build_boot_image` command. This command copies the specified list of files from the *<build-dir>/bin/* directory to the boot directory and executes the steps needed to transform the content of the boot directory into a bootable form. Under the hood, the run tool invokes the run-module types *boot_dir* and *boot_image*. Depending on the run-tool configuration, this form may be an ISO image, a disk image, or a bootable ELF image.

5. Executing the system image using the `run_genode_until` command. Depending on the run-tool configuration, the system image is executed using an emulator or a physical machine. Under the hood, this step invokes the run modules of the types *power_on*, *load*, *log*, and *power_off*. For most platforms, Qemu is used by default. On Linux, the scenario is executed by starting core directly from the boot directory. The `run_genode_until` command takes a regular expression as argument. If the log output of the scenario matches the specified pattern, the `run_genode_until` command returns. If specifying `forever` as argument, this command will never return. If a regular expression is specified, an additional argument determines a timeout in seconds. If the regular expression does not match until the timeout is reached, the run script will abort.

After the successful completion of a run script, the run tool prints the message "Run script execution successful.".

Note that the *hello.run* script does not contain kernel-specific information. Therefore it can be executed from the build directory of any base platform via the command `make run/hello`. When invoking `make` with an argument of the form `run/<run-script>`, the build system searches all repositories for a run script with the specified name. The run script must be located in one of the repositories' *run/* subdirectories and have the file extension `.run`.

5.4.5. The run mechanism explained

The run tool is based on *expect*, which is an extension of the Tcl scripting language that allows for the scripting of interactive command-line-based programs. When the

GENODE ⟋⟍ LABS

user invokes a run script via *make run/<run-script>*, the build system invokes the run tool at *<genode-dir>/tool/run/run* with the run script and the content of the RUN_OPT definition as arguments. The run tool is an expect script that has no other purpose than defining several commands used by run scripts and including the run modules as specified by the run-tool configuration. Whereas *tool/run/run* provides the generic commands, the run modules under *tool/run/<module>/* contain all the peculiarities of the various kernels and boot strategies. The run modules thereby document precisely how the integration and boot concept works for each kernel platform.

Run modules Each module consist of an expect source file located in one of the existing directories of a category. It is named implicitly by its location and the name of the source file, e. g. *image/iso* is the name of the image module that creates an ISO image. The source file contains one mandatory function:

```
run_<module> { <module-args> }
```

The function is called if the step is executed by the run tool. If its execution was successful, it returns true and otherwise false. Certain modules may also call exit on failure.

A module may have arguments, which are - by convention - prefixed with the name of the module, e. g., power_on/amt has an argument called -power-on-amt-host. By convention, the modules contain accessor functions for argument values. For example, the function power_on_amt_host in the run module *power_on/amt* returns the value supplied to the argument -power-on-amt-host. Thereby, a run script can access the value of such arguments in a defined way by calling power_on_amt_host. Also, arguments without a value are treated similarly. For example, for querying the presence of the argument -image-uboot-no-gzip, the run module *run/image/uboot* provides the corresponding function image_uboot_use_no_gzip. In addition to these functions, a module may have additional public functions. Those functions may be used by run scripts or other modules. To enable a run script or module to query the presence of another module, the run tool provides the function have_include. For example, the presence of the *load/tftp* module can be checked by calling have_include with the argument "load/tftp".

5.4.6. Using run scripts to implement test cases

Because run scripts are actually expect scripts, the whole arsenal of language features of the Tcl scripting language is available to them. This turns run scripts into powerful tools for the automated execution of test cases. A good example is the run script at *repos/lib-ports/run/lwip.run*, which tests the lwIP stack by running a simple Genode-based HTTP server on the test machine. It fetches and validates a HTML page from this server. The run script makes use of a regular expression as argument to the run_genode_until

command to detect the state when the web server becomes ready, subsequently executes the `lynx` shell command to fetch the web site, and employs Tcl's support for regular expressions to validate the result. The run script works across all platforms that have network support. To accommodate a high diversity of platforms, parts of the run script depend on the *spec* values as defined for the build directory. The spec values are probed via the `have_spec` function. Depending on the probed spec values, the run script uses the `append_if` and `lappend_if` commands to conditionally assemble the init configuration and the list of boot modules.

To use the run mechanism efficiently, a basic understanding of the Tcl scripting language is required. Furthermore the functions provided by *tool/run/run* and the run modules at *tool/run/* should be studied.

5.4.7. Automated testing across base platforms

To execute one or multiple test cases on more than one base platform, there exists a dedicated tool at *tool/autopilot*. Its primary purpose is the nightly execution of test cases. The tool takes a list of platforms and of run scripts as arguments and executes each run script on each platform. The build directory for each platform is created at */tmp/autopilot.<username>/<platform>* and the output of each run script is written to a file called *<platform>.<run-script>.log*. On stderr, autopilot prints the statistics about whether or not each run script executed successfully on each platform. If at least one run script failed, autopilot returns a non-zero exit code, which makes it straight forward to include autopilot into an automated build-and-test environment.

GENODE LABS

5.5. Git flow

The official Genode Git repository is available at the project's GitHub site:

GitHub project

>   https://github.com/genodelabs/genode

5.5.1. Master and staging

The official Git repository has two branches "master" and "staging".

Master branch The master branch is the recommended branch for users of the framework. It is known to have passed quality tests. The existing history of this branch is fixed and will never change.

Staging branch The staging branch contains the commits that are scheduled for inclusion into the master branch. However, before changes are merged into the master branch, they are subjected to quality-assurance measures conducted by Genode Labs. Those measures include the successful building of the framework for all base platforms and the passing of automated tests. After changes enter the staging branch, those quality-assurance measures are expected to fail. If so, the changes are successively refined by a series of *fixup* commits. Each fixup commit should refer to the commit it is refining using a commit message as follows:

```
fixup "<commit message of the refined commit>"
```

If the fixup is non-trivial, change the "fixup" prefix to "squash" and add a more elaborative description to the commit message.

Once the staging branch passes the quality-assurance measures, the Genode maintainers tidy-up the history of the staging branch by merging all fixup commits with their respective original commit. The resulting commits are then merged on top of the master branch and the staging branch is reset to the new master branch.

Note that the staging branch is volatile. In contrast to the master branch, its history is not stable. Hence, it should not be used to base developments on.

Release version The version number of a Genode release refers to the release date. The two-digit major number corresponds to the last two digits of the year and the two-digit minor number corresponds to the month. For example, "15.02".

Each Genode release represents a snapshot of the master branch taken at release time. It is complemented by the following commits:

GENODE LABS

- "Release notes for version <version>" containing the release documentation in the form of a text file at *doc/release_notes*,

- "News item for Genode <version>" containing the release announcement as published at the *genode.org* website,

- "Version: <version>" with the adaptation of the *VERSION* file.

The latter commit is tagged with the version number. The tag is signed by one of the mainline developers.

5.5.2. Development practice

Each developer maintains a fork of Genode's Git repository. To facilitate close collaboration with the developer community, it is recommended to host the fork on GitHub. Open a GitHub account, use GitHub's web interface to create a new fork, and follow the steps given by GitHub to fetch the cloned repository to your development machine.

In the following, we refer to the official Genode repository as "genodelabs/genode". To conveniently follow the project's mainline development, it is recommended to register the official repository as a "remote" in your Git repository:

```
git remote add genodelabs https://github.com/genodelabs/genode.git
```

Once, the official repository is known to your clone, you can fetch new official revisions via

```
git fetch genodelabs
```

Topic branches   As a rule of thumb, every line of development has a corresponding topic in the issue tracker. This is the place where the developers discuss and review the ongoing work. Hence, when starting a new line of development, the first step should be the creation of a new topic.

Issue tracker

https://github.com/genodelabs/genode/issues

The new topic should be accompanied with a short description about the motivation behind the line of work and the taken approach. The second step is the creation of a dedicated topic branch in the developer's fork of Genode's Git repository.

```
git checkout -b issue<number> genodelabs/master
```

GENODE 🖉 LABS

The new topic branch should be based on the most current *genodelabs/master* branch. This eases the later integration of the topic branch into the mainline development.

While working on a topic branch, it is recommended to commit many small intermediate steps. This is useful to keep track of the line of thoughts during development. This history is regarded as volatile. That is, it is not set in stone. Hence, you as developer do not have to spend too much thoughts on the commits during the actual development.

Once the work on the topic is completed and the topic branch is going to get integrated into the mainline development, the developer curates the topic-branch history so that a short and well-arranged sequence of commits remains. This step is usually performed by interactively editing the topic-branch history via the `git rebase -i` command. In many cases, the entire topic branch can be squashed into a single commit. The goal behind this curating step is to let the mainline history document the progress at a level of detail that is meaningful for the users of the framework. The mainline history should satisfy the following:

- The relationship of a commit with an issue at the issue tracker should be visible. For this reason, GitHub's annotations "Issue #n" and "Fixed #n" are added to the commit messages.

- Revisiting the history between Genode releases should clearly reveal the changes that potentially interest the users. I.e., when writing the quarterly release notes, the Genode developers go through the history and base the release-notes documentation on the information contained in the commit messages. This works best if each topic is comprised by a few commits with meaningful descriptions. This becomes hard if the history contains too many details.

- Each commit should represent a kind of "transaction" that can be reviewed independently without knowing too much context. This is hardly possible if intermediate steps that subsequently touch the same code are present as individual commits.

- It should be easy to selectively revert individual topics/features using git revert (e. g., when trouble-shooting). This is simple when each topic is represented by one or just a few commits.

Coding conventions   Genode's source code follows time-tested conventions regarding the coding style and code pattern, which are important to follow. The coding style is described in the following document:

Coding-style Guidelines

http://genode.org/documentation/developer-resources/coding_style

GENODE ⩘ LABS

Writing a commit message   Commit messages should adhere the following conven-tion. The first line summarizes the commit using not more than 50 characters. This line will be displayed by various tools. So it should express the basic topic and eventually refer to an issue. For example:

```
Add sanity checks in tool/tool_chain, fix #62
```

If the patch refers to an existing issue, add a reference to the corresponding issue. If not, please consider opening an issue first. In the case the patch is supposed to close an existing issue, add this information using GitHub's conventions, e. g., by stating "Fix #45" in your commit message, the issue will be closed automatically, by stating "Issue #45", the commit will be displayed in the stream of discussion of the corresponding issue.

After a blank line, a description of the patch follows. The description should consider the following questions:

- Why is the patch needed?

- How does the patch achieve the goal?

- What are known consequences of this patch? Will it break API compatibility, or produce a follow-up issue?

Reconsider the documentation related to your patch: If the commit message contains important information not present in the source code, this information should bet-ter be placed into the code or the accompanied documentation (e. g., in the form of a README file).

GENODE LABS

6. System configuration

There are manifold principal approaches to configure different aspects of an operating system and the applications running on top. At the lowest level, there exists the opportunity to pass configuration information to the boot loader. This information may be evaluated directly by the boot loader or passed to the booted system. As an example for the former, some boot loaders allow for setting up a graphics mode depending on its configuration. Hence, the graphics mode to be used by the OS could be defined right at this early stage of booting. More prominently, however, is the mere passing of configuration information to the booted OS, e. g., in the form of a kernel command line or as command-line arguments to boot modules. The OS interprets boot-loader-provided data structures (i. e., multiboot info structures) to obtain such information. Most kernels interpret certain configuration arguments passed via this mechanism. At the OS-initialization level, before any drivers are functioning, the OS behavior is typically governed by configuration information provided along with the kernel image, i. e., an initial file-system image (initrd). On Linux-based systems, this information comes in the form of configuration files and init scripts located at well-known locations within the initial file-system image. Higher up the software stack, configuration becomes an even more diverse topic. I.e., the runtime behavior of a GNU/Linux-based system is defined by a conglomerate of configuration files, daemons and their respective command-line arguments, environment variables, collections of symlinks, and plenty of heuristics.

The diversity and complexity of configuration mechanisms, however, is problematic for high-assurance computing. To attain a high level of assurance, Genode's architecture must be complemented by a low-complexity yet scalable configuration concept. The design of this concept takes the following considerations into account:

Uniformity across platforms To be applicable across a variety of kernels and hardware platforms, the configuration mechanism must not rely on a particular kernel or boot loader. Even though boot loaders for x86-based machines usually support the multiboot specification and thereby the ability to supplement boot modules with additional command lines, boot loaders on ARM-based platforms generally lack this ability. Furthermore, even if a multiboot compliant boot loader is used, the kernel - once started - must provide a way to reflect the boot information to the system on top, which is not the case for most microkernels.

Low complexity The configuration mechanism is an intrinsic part of each component. Hence, it affects the trusted computing base of every Genode-based system. For this reason, the mechanism must be easy to understand and implementable without the need for complex underlying OS infrastructure. As a negative example, the provision of configuration files via a file system would require each Genode-based system to support the notion of a file system and to define the naming of configuration files.

GENODE 𝄞 LABS

Expressiveness Passing configuration information as command-line arguments to components at their creation time seems like a natural way to avoid the complexity of a file-based configuration mechanism. However, whereas command-line arguments are the tried and tested way for supplying program arguments in a concise way, the expressiveness of the approach is limited. In particular, it is ill-suited for expressing structured information as often found in configurations. Being a component-based system, Genode requires a way to express relationships between components, which lends itself to the use of a structural representation.

Common syntax The requirement of a low-complexity mechanism mandates a common syntax across components. Otherwise, each component would need to come with a custom parser. Each of those parsers would eventually inflate the complexity of the trusted computing base. In contrast, a common syntax that is both expressive and simple to parse helps to avoid such redundancies by using a single parser implementation across all components.

Least privilege Being the guiding motive behind Genode's architecture, the principle of least privilege needs to be applied to the access of configuration information. Each component needs to be able to access its own configuration but must not observe configuration information concerning unrelated components. A system-global registry of configurations or even a global namespace of keys for such a database would violate this principle.

Accommodation of dynamic workloads Supplying configuration information at the construction time of a component is not sufficient for long-living components, whose behavior might need to be adapted at runtime. For example, the assignment of resources to the clients of a resource multiplexer might change over the lifetime of the resource multiplexer. Hence, the configuration concept should provide a means to update the configuration information of a component after it has been constructed.

```
<config>
  <parent-provides> ... </parent-provides>
  <default-route> ... </default-route>
  ...
  <start name="nitpicker">
    ...
  </start>
  <start name="launchpad">
    ...
    <config>
      <launcher name="Virtualbox">
        <binary name="init"/>
        <config>
          <parent-provides> ... </parent-provides>
          <default-route>
            <any-service> <any-child/> <parent/> </any-service>
          </default-route>
          <start name="nit_fb">
            <resource name="RAM" quantum="6M"/>
            <config xpos="400" ypos="270" width="300" height="200" />
            <provides>
              <service name="Input"/>
              <service name="Framebuffer"/>
            </provides>
          </start>
          <start name="virtualbox">
            <resource name="RAM" quantum="1G"/>
            <config vbox_file="test.vbox" vm_name="TestVM">
              ...
            </config>
          </start>
        </config>
      </launcher>
    </config>
  </start>
</config>
```

Figure 46: Nested system configuration

6.1. Nested configuration concept

Genode's configuration concept is based on the ROM session interface described in Section 4.5.1. In contrast to a file-system interface, the ROM session interface, is extremely simple. The client of a ROM service specifies the requested ROM module by its name as known by the client. There is neither a way to query a list of available ROM modules, nor are ROM modules organized in a hierarchic name space.

The ROM session interface is implemented by core's ROM service to make boot modules available to other components. Those boot modules comprise the executable binaries of the init component as well as those of the components created by init. Furthermore, a ROM module called "config" contains the configuration of the init process in XML format. To obtain its configuration, init requests a ROM session for the ROM module "config" from its parent, which is core. Figure 46 shows an example of such a

GENODE ⤳ LABS
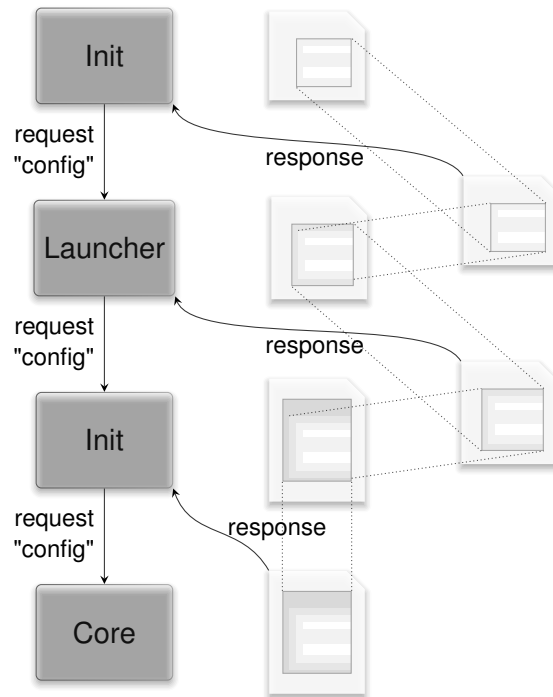
Figure 47: Successive interception of "config" ROM requests

config ROM module.

The config ROM module uses XML as syntax, which supports the expression of arbitrary structural data while being simple to parse. I.e., Genode's XML parser comes in the form of a single header file with less than 400 lines of code. Init's configuration is contained within a single `<config>` node.

Each component started by init obtains its configuration by requesting a ROM module named "config" from its parent. Init responds to this request by handing out a locally-provided ROM session. Instead of handing out the "config" ROM module as obtained from core, it creates a new dataspace that solely contains the portion of init's config ROM module that refers to the respective child. Analogously to init's configuration, each child's configuration has the form of a single `<config>` node. This works recursively. From each component's perspective, including the init component, the mechanism for obtaining its configuration is identical – it obtains a ROM session for a ROM module named "config" from its parent. The parent interposes the ROM session request as described in Section 4.7.3. Figure 47 shows the successive interposing of "config" ROM requests according to the example configuration given in Figure 46. At each level, the information structure within the `<config>` node can be different. Besides following the convention that a configuration has the form of a single `<config>` node, each component can introduce arbitrary custom tags and attributes.

GENODE ⊿ LABS

Besides being simple, the use of the ROM session interface for supplying configuration information has the benefit of supporting dynamic configuration updates over the lifetime of the config ROM session. Section 4.5.1 describes the update protocol between client and server of a ROM session. This way, the configuration of long-living components can be dynamically changed.

GENODE LABS

6.2. The init component

The init component plays a special role within Genode's component tree. It gets started directly by core, gets assigned all physical resources, and controls the execution of all subsequent component nodes, which can be further instances of init. Init's policy is driven by an XML-based configuration, which declares a number of children, their relationships, and resource assignments.

6.2.1. Session routing

At the parent-child interface, there are two operations that are subject to policy decisions of the parent: the child announcing a service and the child requesting a service. If a child announces a service, it is up to the parent to decide if and how to make this service accessible to its other children. When a child requests a service, the parent may deny the session request, delegate the request to its own parent, implement the requested service locally, or open a session at one of its other children. This decision may depend on the service requested or the session-construction arguments provided by the child. Apart from assigning resources to children, the central element of the policy implemented in the parent is a set of rules to route session requests. Therefore, init's configuration concept is laid out around child components and the routing of session requests originating from those components. The mechanism is best illustrated by an example:

```
<config>
  <parent-provides>
    <service name="LOG"/>
  </parent-provides>
  <start name="timer">
    <resource name="RAM" quantum="1M"/>
    <provides> <service name="Timer"/> </provides>
  </start>
  <start name="test-timer">
    <resource name="RAM" quantum="1M"/>
    <route>
      <service name="Timer">  <child name="timer"/> </service>
      <service name="LOG">    <parent/>             </service>
    </route>
  </start>
</config>
```

First, there is the declaration of services provided by the parent of the configured init instance. In this case, we declare that the parent provides a a LOG service. For each child to start, there is a `<start>` node describing resource assignments, declaring

GENODE ⟋ LABS

services provided by the child, and holding a routing table for session requests originating from the child. The first child is called "timer" and implements the "Timer" service. The second component called "test-timer" is a client of the timer service. In its routing table, we see that requests for "Timer" sessions are routed to the "timer" child whereas requests for "LOG" sessions are routed to init's parent. Per-child service routing rules provide a flexible way to express arbitrary client-server relationships. For example, service requests may be transparently mediated through special policy components acting upon session-construction arguments. There might be multiple children implementing the same service, each targeted by different routing tables. If there exists no valid route to a requested service, the service is denied. In the example above, the routing tables act effectively as a white list of services the child is allowed to use.

Routing based on session labels   Access-control policies in Genode systems are based on session labels. When a server receives a new session request, the session label is passed along with the request.

A session label is a string that is assembled by the components that are involved with routing the session request from the client along the branches of the component tree to the server. The client may specify the least significant part of the label by itself. This part gives the parent a hint for routing the request. For example, a client may create two file-system sessions, one labeled with "home" and one labeled with "bin". The parent may take this information into account and route the individual requests to different file-system servers. The label is successively superseded (prefixed) by additional parts along the chain of components on the route of the session request. The first part of the label is the most significant part as it is imposed by the component in the intermediate proximity of the server. The last part is the least trusted part of the label because it originated from the client. Once the session request arrives at the server, the server takes the session label as the key to select a server-side policy as described in Section 4.6.2.

In most cases, routing decisions are simply based on the type of the requested sessions. However, by equipping `<service>` nodes with the following attributes, it is possible to take session labels as a criterion for the routing of session requests into account.

`label="<string>"`   The session label must perfectly match the specified string.

`label_prefix="<string>"`   The first part of the label must match the specified string.

`label_suffix="<string>"`   The last part of the label must match the specified string.

If no attributes are present, the route matches. The attributes can be combined. If any of the specified attributes mismatch, the route is neglected. If multiple `<service>` nodes match in init's routing configuration, the first matching rule is taken. So the order of the nodes is important.

GENODE ⬰ LABS

Wildcards In practice, usage scenarios become more complex than the basic example, increasing the size of routing tables. Furthermore, in many practical cases, multiple children may use the same set of services and require duplicated routing tables within the configuration. In particular during development, the elaborative specification of routing tables tend to become an inconvenience. To alleviate this problem, there are two mechanisms, namely wildcards and a default route. Instead of specifying a list of individual service routes targeting the same destination, the wildcard `<any-service>` becomes handy. For example, instead of specifying

```
<route>
  <service name="ROM"> <parent/> </service>
  <service name="RAM"> <parent/> </service>
  <service name="PD">  <parent/> </service>
  <service name="CPU"> <parent/> </service>
</route>
```

the following shortform can be used:

```
<route>
  <any-service> <parent/> </any-service>
</route>
```

The latter version is not as strict as the first one because it permits the child to create sessions at the parent, which were not white listed in the elaborative version. Therefore, the use of wildcards is discouraged for configuring untrusted components. Wildcards and explicit routes may be combined as illustrated by the following example:

```
<route>
  <service name="LOG"> <child name="nitlog"/> </service>
  <any-service>        <parent/>               </any-service>
</route>
```

The routing table is processed starting with the first entry. If the route matches the service request, it is taken, otherwise the remaining routing-table entries are visited. This way, the explicit service route of "LOG" sessions to the "nitlog" child shadows the LOG service provided by the parent.

To allow a child to use services provided by arbitrary other children, there is a further wildcard called `<any-child>`. Using this wildcard, such a policy can be expressed as follows:

```
<route>
  <any-service> <parent/>    </any-service>
  <any-service> <any-child/> </any-service>
</route>
```

GENODE ⟋ LABS

This rule would delegate all session requests referring to one of the parent's services to the parent. If no parent service matches the session request, the request is routed to any child providing the service. The rule can be further abbreviated to:

```
<route>
  <any-service> <parent/> <any-child/> </any-service>
</route>
```

Init detects potential ambiguities caused by multiple children providing the same service. In this case, the ambiguity must be resolved using an explicit route preceding the wildcards.

Default routing   To reduce the need to specify the same routing table for many children in one configuration, there is a `<default-route>` mechanism. The default route is declared within the `<config>` node and used for each `<start>` entry with no `<route>` node. In particular during development, the default route becomes handy to keep the configuration tidy and neat.

The combination of explicit routes and wildcards is designed to scale well from being convenient to use during development towards being highly secure at deployment time. If only explicit rules are present in the configuration, the permitted relationships between all processes are explicitly defined and can be easily verified.

6.2.2. Resource quota saturation

If a specified resource (i. e., RAM quota) exceeds the available resources, all available resources are assigned completely to the child. This makes it possible to assign all remaining resources to the last child by simply specifying an overly large quantum.

6.2.3. Handing out slack resources

Resources may remain unused after the creation of all children if the quantum of available resources is higher than the sum of the quotas assigned to each child. Init makes such slack memory available to its children via the resource-request protocol described in Section 3.3.4. Slack memory is handed out on a first-come first-served basis.

6.2.4. Multiple instantiation of a single ELF binary

Each `<start>` node requires a unique `name` attribute. By default, the value of this attribute is used as ROM module name for obtaining the ELF binary from the parent. If multiple instances of a component with the same ELF binary are needed, the binary name can be explicitly specified using a `<binary>` sub node of the `<start>` node:

GENODE ⫷ LABS

```
<binary name="filename"/>
```

This way, a unique child name can be defined independently from the binary name.

6.2.5. Nested configuration

Each <start> node can host a <config> sub node. As described in Section 6.1, the content of this sub node is provided to the child when a ROM session for the module name "config" is requested. Thereby, arbitrary configuration parameters can be passed to the child. For example, the following configuration starts timer-test within an init instance within another init instance. To show the flexibility of init's service routing facility, the "Timer" session of the second-level timer-test child is routed to the timer service started at the first-level init instance.

GENODE ⋙ LABS

```
<config>
  <parent-provides>
    <service name="LOG"/>
    <service name="ROM"/>
    <service name="RAM"/>
    <service name="CPU"/>
    <service name="PD"/>
  </parent-provides>
  <start name="timer">
    <resource name="RAM" quantum="1M"/>
    <provides><service name="Timer"/></provides>
    <route>
      <any-service> <parent/> </any-service>
    </route>
  </start>
  <start name="init">
    <resource name="RAM" quantum="1M"/>
    <config>
      <parent-provides>
        <service name="Timer"/>
        <service name="LOG"/>
      </parent-provides>
      <start name="test-timer">
        <resource name="RAM" quantum="1M"/>
        <route>
          <any-service> <parent/> </any-service>
        </route>
      </start>
    </config>
    <route>
      <service name="Timer"> <child name="timer"/> </service>
      <any-service>            <parent/>             </any-service>
    </route>
  </start>
</config>
```

The services ROM, RAM, CPU, and PD are required by the second-level init instance to create the timer-test component. As illustrated by this example, the use of nested configurations enables the construction of arbitrarily complex component trees via a single configuration.

As an alternative to specifying all nested configurations in a single configuration, sub configurations can be placed in separate ROM modules specified via the configfile node. For example:

166

GENODE ⩘ LABS
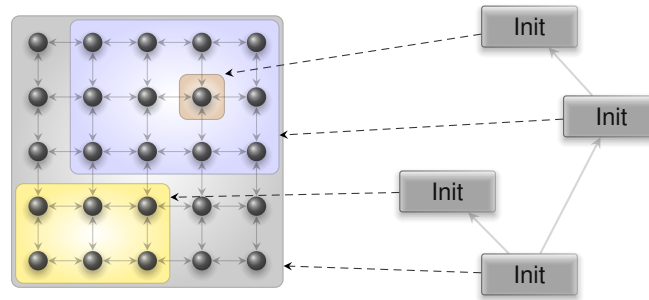
Figure 48: Successive virtualization of CPU affinity spaces by nested instances of init

```
<start name="nitpicker">
  <resource name="RAM" quantum="1M"/>
  <configfile name="nitpicker.config"/>
</start>
```

6.2.6. Assigning subsystems to CPUs

Most multi-processor (MP) systems have topologies that can be represented on a two-dimensional coordinate system. CPU nodes close to each other are expected to have closer relationship than distant nodes. In a large MP system, it is natural to assign clusters of closely related nodes to a given workload. As described in Section 3.2, Genode's architecture is based on a strictly hierarchic organizational structure. Thereby, it lends itself to the idea of applying this successive virtualization of resources to the problem of clustering CPU nodes.

Each component within the component tree has a component-local view on a so-called *affinity space*, which is a two-dimensional coordinate space. If the component creates a new subsystem, it can assign a portion of its own affinity space to the new subsystem by imposing a rectangular affinity location to the subsystem's CPU session. Figure 48 illustrates the idea.

Following from the expression of affinities as a rectangular location within a component-local affinity space, the assignment of subsystems to CPU nodes consists of two parts: the definition of the affinity space dimensions as used for the init instance, and the association of subsystems with affinity locations relative to the affinity space. The affinity space is configured as a sub node of the <config> node. For example, the following declaration describes an affinity space of 4x2:

GENODE ⫸ LABS

```
<config>
  ...
  <affinity-space width="4" height="2" />
  ...
</config>
```

Subsystems can be constrained to parts of the affinity space using the `<affinity>` sub node of a `<start>` entry:

```
<config>
  ...
  <start name="loader">
    <affinity xpos="0" ypos="1" width="2" height="1" />
    ...
  </start>
  ...
</config>
```

As illustrated by this example, the numbers used in the declarations for this instance of init are not directly related to physical CPUs. If the machine has merely two cores, init's affinity space would be mapped to the range 0,1 of physical CPUs. However, in a machine with 16x16 CPUs, the loader would obtain 8x8 CPUs with the upper-left CPU at position (4,0).

6.2.7. Priority support

The number of CPU priorities to be distinguished by init can be specified with the `prio_levels` attribute of the `<config>` node. The value must be a power of two. By default, no priorities are used. To assign a priority to a child process, a priority value can be specified as `priority` attribute of the corresponding `<start>` node. Valid priority values lie in the range of -prio_levels + 1 (maximum priority degradation) to 0 (no priority degradation).

6.2.8. Init verbosity

To ease debugging, init can be instructed to print diverse status information as LOG output. To enable the verbose mode, assign the value "yes" to the `verbose` attribute of the `<config>` node.

GENODE ⨯ LABS

7. Under the hood

This chapter gives insight into the inner workings of the Genode OS framework. In particular, it explains how the concepts explained in Chapter 3 are realized on different kernels and hardware platforms.

GENODE LABS

7.1. Component-local startup code and linker scripts

All Genode components including core rely on the same startup code, which is roughly outlined at the end of Section 3.5. This section revisits the required steps in more detail and refers to the corresponding points in the source code. Furthermore, it provides background information about the linkage of components, which is closely related to the startup code.

7.1.1. Linker scripts

Under the hood, the Genode build system uses three different linker scripts located at *repos/base/src/ld/*:

genode.ld is used for statically linked components, including core,

genode_dyn.ld is used for dynamically linked components, i. e., components that are linked against at least one shared library,

genode_rel.ld is used for shared libraries.

Additionally, there exists a special linker script for the dynamic linker (Section 7.5).

Each program image generated by the linker generally consists of three parts, which appear consecutively in the component's virtual memory.

1. A read-only "text" part contains sections for code, read-only data, and the list of global constructors and destructors.

   The startup code is placed in a dedicated section `.text.crt0`, which appears right at the start of the segment. Thereby the link address of the component is known to correspond to the ELF entrypoint (the first instruction of the assembly startup code). This is useful when converting the ELF image of the base-hw version of core into a raw binary. Such a raw binary can be loaded directly into the memory of the target platform without the need for an ELF loader.

   The mechanisms for generating the list of constructors and destructors differ between CPU architecture and are defined by the architecture's ABI. On x86, the lists are represented by `.ctors.*` and `.dtors.*`. On ARM, the information about global constructors is represented by `.init_array` and there is no visible information about global destructors.

2. A read-writable "data" part that is pre-populated with data.

3. A read-writable "bss" part that is not physically present in the binary but known to be zero-initialized when the ELF image is loaded.

GENODE ⬙ LABS

The link address is not defined in the linker script but specified as linker argument. The default link address is specified in a platform-specific spec file, e. g., *repos/base-nova/mk/spec/nova.mk* for the NOVA platform. Components that need to organize their virtual address space in a special way (e. g., a virtual machine monitor that co-locates the guest-physical address space with its virtual address space) may specify link addresses that differ from the default one by overriding the LD_TEXT_ADDR value.

ELF entry point   As defined at the start of the linker script via the ENTRY directive, the ELF entrypoint is the function _start. This function is located at the very beginning of the .text.crt0 section. See the Section 7.1.2 for more details.

Symbols defined by the linker script   The following symbols are defined by the linker script and used by the base framework.

_prog_img_beg, _prog_img_data, _prog_img_end   Those symbols mark the start of the "text" part, the start of the "data" part (the end of the "text" part), and the end of the "bss" part. They are used by core to exclude those virtual memory ranges from the core's virtual-memory allocator (core-region allocator).

_parent_cap, _parent_cap_thread_id, _parent_cap_local_name   Those symbols are located at the beginning of the "data" part. During the ELF loading of a new component, the parent writes information about the parent capability to this location (the start of the first read-writable ELF segment). See the corresponding code in the Loaded_executable constructor in *base/src/lib/base/child_process.cc*. The use of the information depends on the base platform. E.g., on a platform where a capability is represented by a tuple of a global thread ID and an object ID such as OKL4 and L4ka::Pistachio, the information is taken as verbatim values. On platforms that fully support capability-based security without the use of any form of a global name to represent a capability, the information remains unused. Here, the parent capability is represented by the same known local name in all components.

Even though the linker scripts are used across all base platforms, they contain a few platform-specific supplements that are needed to support the respective kernel ABIs. For example, the definition of the symbol __l4sys_invoke_indirect is needed only on the Fiasco.OC platform and is unused on the other base platforms. Please refer to the comments in the linker script for further explanations.

7.1.2. Startup code

The execution of the initial thread of a new component starts at the ELF entry point, which corresponds to the _start function. This is an assembly function defined in

GENODE ⬥ LABS

repos/base/src/lib/startup/spec/<arch>/crt0.s where *<arch>* is the CPU architecture (x86_32, x86_64, or ARM).

Assembly startup code The assembly startup code is position-independent code (PIC). Because the Genode base libraries are linked against both statically-linked and dynamically linked executables, they have to be compiled as PIC code. To be consistent with the base libraries, the startup code needs to be position-independent, too.

The code performs the following steps:

1. Saving the initial state of certain CPU registers. Depending on the used kernel, these registers carry information from the kernel to the core component. More details about this information are provided by Section 7.3.1. The initial register values are saved in global variables named `_initial_<register>`. The global variables are located in the BSS segment. Note that those variables are used solely by core.

2. Setting up the initial stack. Before the assembly code can call any higher-level C function, the stack pointer must be initialized to point to the top of a valid stack. The initial stack is located in the BSS section and referred to by the symbol `_stack_high`. However, having a stack located within the BSS section is dangerous. If it overflows (e. g., by declaring large local variables, or by recursive function calls), the stack would silently overwrite parts of the BSS and DATA sections located below the lower stack boundary. For prior known code, the stack can be dimensioned to a reasonable size. But for arbitrary application code, no assumption about the stack usage can be made. For this reason, the initial stack cannot be used for the entire lifetime of the component. Before any component-specific code is called, the stack needs to be relocated to another area of the virtual address space where the lower bound of the stack is guarded by empty pages. When using such a "real" stack, a stack overflow will produce a page fault, which can be handled or at least immediately detected. The initial stack is solely used to perform the steps required to set up the real stack. Because those steps are the same for all components, the usage of the initial stack is bounded.

3. Because the startup code is used by statically linked components as well as the dynamic linker, the startup immediately calls the `init_rtld` hook function. For regular components, the function does not do anything. The default implementation in *init_main_thread.cc* at *repos/base/src/lib/startup/* is a weak function. The dynamic linker provides a non-weak implementation, which allows the linker to perform initial relocations of itself very early at the dynamic linker's startup.

4. By calling the `init_main_thread` function defined in *repos/base/src/lib/startup/init_main_thread.cc*, the assembly code triggers the execution of all the steps needed for the creation

GENODE 🏋 LABS

of the real stack. The function is implemented in C++, uses the initial stack, and
returns the address of the real stack.

5. With the new stack pointer returned by `init_main_thread`, the assembly startup
   code is able to switch the stack pointer from the initial stack to the real stack. From
   this point on, stack overflows cannot easily corrupt any data.

6. With the real stack in place, the assembly code finally passes the control over to
   the C++ startup code provided by the `_main` function.

Initialization of the real stack along with the Genode environment   As mentioned
above, the assembly code calls the `init_main_thread` function (located in *repos/base/s-
rc/lib/startup/init_main_thread.cc*) for setting up the real stack for the program. For plac-
ing a stack in a dedicated portion of the component's virtual address space, the function
needs to overcome two principle problems:

- It needs to obtain the backing store used for the stack, i. e., allocating a dataspace
  from the component's RAM session as initialized by the parent.

- It needs to preserve a portion of its virtual address space for placing the stack and
  make the allocated memory visible within this portion.

In order to solve both problems, the function needs to obtain capabilities for its RAM
session and PD session from its parent. This comes down to the need to perform RPC
calls. First, for requesting the PD and RAM session capabilities from the parent, and
second, for invoking the session capabilities to perform the RAM allocation and region-
map attach operations.

   The RPC mechanism is based on C++. In particular, the mechanism supports the
propagation of C++ exceptions across RPC interfaces. Hence, before being able to per-
form RPC calls, the program must initialize the C++ runtime including the exception-
handling support. The initialization of the C++ runtime, in turn, requires support for
dynamically allocating memory. Hence, a heap must be available. This chain of depen-
dencies ultimately results in the need to construct the entire Genode environment as a
side effect of initializing the real stack of the program.

   During the construction of the Genode environment, the program requests its own
RAM, CPU, and PD sessions from its parent.

   With the environment constructed, the program is able to interact with its own PD
and RAM sessions and can principally realize the initialization of the real stack. How-
ever, instead of merely allocating a new RAM dataspace and attaching the dataspace to
the address space of the PD session, a so-called stack area is used. The stack area is a
secondary region map that is attached as a dataspace to the component's address-space
region map. This way, virtual-memory allocations within the stack area can be man-
aged manually. I.e., the spaces between the stacks of different threads are guaranteed

GENODE 〳 LABS

to remain free from any attached dataspaces. The stack area of a component is created as part of the component's PD session. The environment initialization code requests its region-map capability via `Pd_session::stack_area` and attaches it as a managed dataspace to the component's address space.

Component-dependent startup code   With the Genode environment constructed and the initial stack switched to a proper stack located in the stack area, the component-dependent startup code of the `_main` function in *repos/base/src/lib/startup/_main.cc* can be executed. This code is responsible for calling the global constructors of the program before calling the program's main function.

In accordance to the established signature of the `main` function, taking an argument list and an environment as arguments, the startup code supplies these arguments but uses dummy default values. However, since the values are taken from the global variables `genode_argv`, `genode_argc`, and `genode_envp`, a global constructor is able to override the default values.

The startup code in `_main.cc` is accompanied with support for *atexit* handling. The atexit mechanism allows for the registration of handlers to be called at the exit of the program. It is provided in the form of a POSIX API by the C runtime. But it is also used by the compiler to schedule the execution of the destructors of function-local static objects. For the latter reason, the atexit mechanism cannot be merely provided by the (optional) C runtime but must be supported by the base library.

GENODE 🎗 LABS

7.2. C++ runtime

Genode is implemented in C++ and relies on all C++ features required to use the language in its idiomatic way. This includes the use of exceptions and runtime-type information.

7.2.1. Rationale behind using exceptions

Compared to return-based error handling as prominently used in C programs, the C++ exception mechanism is much more complex. In particular, it requires the use of a C++ runtime library that is called as a back-end by the exception handling code and generated by the compiler. This library contains the functionality needed to unwind the stack and a mechanism for obtaining runtime type information (RTTI). The C++ runtime libraries that come with common tool chains, in turn, rely on a C library for performing dynamic memory allocations, string operations, and I/O operations. Consequently, C++ programs that rely on exceptions and RTTI implicitly depend on a C library. For this reason, the use of those C++ features is universally disregarded for low-level operating-system code that usually does not run in an environment where a complete C library is available.

In principle, C++ can be used without exceptions and RTTI (by passing the arguments `-fno-exceptions` and `-fno-rtti` to GCC). However, without those features, it is hardly possible to use the language as designed.

For example, when the operator `new` is used, it performs two steps: Allocating the memory needed to hold the to-be-created object and calling the constructor of the object with the return value of the allocation as `this` pointer. In the event that the memory allocation fails, the only way for the allocator to propagate the out-of-memory condition is throwing an exception. If such an exception is not thrown, the constructor would be called with a null as `this` pointer.

Another example is the handling of errors during the construction of an object. The object construction may consist of several consecutive steps such as the construction of base classes and aggregated objects. If one of those steps fails, the construction of the overall object remains incomplete. This condition must be propagated to the code that issued the object construction. There are two principle approaches:

1. The error condition can be kept as an attribute in the object. After constructing the object, the user of the object may detect the error condition by requesting the attribute value. However, this approach is plagued by the following problems.

   First, the failure of one step may cause subsequent steps to fail as well. In the worst case, if the failed step initializes a pointer that is passed to subsequent steps, the subsequent steps may use an uninitialized pointer. Consequently, the error condition must eventually be propagated to subsequent steps, which, in turn, need to be implemented in a defensive way.

GENODE ⚖ LABS

Second, if the construction failed, the object exists but it is inconsistent. In the worst case, if the user of the object misses to check for the successful construction, it will perform operations on an inconsistent object. But even in the good case, where the user detects the incomplete construction and decides to immediately destruct the object, the destruction is error prone. The already performed steps may have had side effects such as resource allocations. So it is important to revert all the successful steps by invoking their respective destructors. However, when destructing the object, the destructors of the incomplete steps are also called. Consequently, such destructors need to be implemented in a defensive manner to accommodate this situation.

Third, objects cannot have references that depend on potentially failing construction steps. In contrast to a pointer that may be marked as uninitialized by being a null pointer, a reference is, by definition, initialized once it exists. Consequently, the result of such a step can never be passed as reference to subsequent steps. Pointers must be used.

Fourth, the mere existence of incompletely constructed objects introduces many variants of possible failures that need to be considered in the code. There may be many different stages of incompleteness. Because of the third problem, every time a construction step takes the result of a previous step as an argument, it explicitly has to consider the error case. This, in turn, tremendously inflates the test space of the code.

Furthermore, there needs to be a convention of how the completion of an object is indicated. All programmers have to learn and follow the convention.

2. The error condition triggers an exception. Thereby, the object construction immediately stops at the erroneous step. Subsequent steps are not executed at all. Furthermore, while unwinding the stack, the exception mechanism reverts all already completed steps by calling their respective destructors. Consequently, the construction of an object can be considered as a transaction. If it succeeds, the object is known to be completely constructed. If it fails, the object immediately ceases to exist.

Thanks to the transactional semantics of the second variant, the state space for potential error conditions (and thereby the test space) remains small. Also, the second variant facilitates the use of references as class members, which can be safely passed as arguments to subsequent constructors. When receiving such a reference as argument (as opposed to a pointer), no validity checks are needed. Consequently, by using exceptions, the robustness of object-oriented code (i. e., code that relies on C++ constructors) can be greatly improved over code that avoids exceptions.

GENODE ⚞ LABS

7.2.2. Bare-metal C++ runtime

Acknowledging the rationale given in the previous section, there is still the problem of the complexity added by the exception mechanism. For Genode, the complexity of the trusted computing base is a fundamental metric. The C++ exception mechanism with its dependency to the C library arguably adds significant complexity. The code complexity of a C library exceeds the complexity of the fundamental components (such as the kernel, core, and init) by an order of magnitude. Making the fundamental components depend on such a C library would jeopardize one of Genode's most valuable assets, which is its low complexity.

To enable the use of C++ exceptions and runtime type information but avoid the incorporation of an entire C library into the trusted computing base, Genode comes with a customized C++ runtime that does not depend on a C library. The C++ runtime libraries are provided by the tool chain. To build those libraries without a C library, a libc emulation header (*tool/libgcc_libc_stub.h*) is used instead of the interface of a real C library. The emulation header contains only those definitions and declarations needed by the C++ runtime. The resulting libraries contain references to (some of) the symbols present in the emulation header. Those symbols are provided by Genode's C++ support code (*repos/base/src/lib/cxx*). The implementation of those functions is specifically tied to the usage patterns of the C++ runtime. Several of the functions are mere dummies.

Unfortunately, the interface used by the C++ runtime does not reside in a specific namespace but it is rather a subset of the POSIX API. When linking a real C library to a Genode component, the symbols present in the C library would collide with the symbols present in Genode's C++ support code. For this reason, the C++ runtime (of the compiler) and Genode's C++ support code are wrapped in a single library (*repos/base/lib/mk/cxx.mk*) in a way that all POSIX functions remain hidden. All the references of the C++ runtime are resolved by the C++ support code, both wrapped in the cxx library. To the outside, the cxx library solely exports the CXA ABI as required by the compiler.

GENODE ⬙ LABS

7.3. Interaction of core with the underlying kernel

Core is the root of the component tree. It is initialized and started directly by the underlying kernel and has two purposes. First, it makes the low-level physical resources of the machine available to other components in the form of services. These resources are physical memory, processing time, device resources, initial boot modules, and protection mechanisms (such as the MMU, IOMMU, and virtualization extensions). It thereby hides the peculiarities of the used kernel behind an API that is uniform across all kernels supported by Genode. Core's second purpose is the creation of the init component by using its own services and following the steps described in Section 3.5.

Even though core is executed in user mode, its role as the root of the component tree makes it as critical as the kernel. It just happens to be executed in a different processor mode. Whereas regular components solely interact with the kernel when performing inter-component communication, core interplays with the kernel more intensely. The following subsections go into detail about this interplay.

The description tries to be general across the various kernels supported by Genode. Note, however, that a particular kernel may deviate from the general description.

7.3.1. Bootstrapping and allocator setup

At boot time, the kernel passes information about the physical resources and the initial system state to core. Even though the mechanism and format of this information varies from kernel to kernel, it generally covers the following aspects:

- A list of free physical memory ranges

- A list of the physical memory locations of the boot modules along with their respective names

- The number of available CPUs

- All information needed to enable the initial thread to perform kernel operations

Core's allocators   Core's kernel-specific platform initialization code (*core/platform.cc*) uses this information to initialize the allocators used for keeping track of physical resources. Those allocators are:

RAM allocator contains the ranges of the available physical memory

I/O memory allocator contains the physical address ranges of unused memory-mapped I/O resources. In general, all ranges not initially present in the RAM allocator are considered to be I/O memory.

I/O port allocator contains the I/O ports on x86-based platforms that are currently not in use. This allocator is initialized with the entire I/O port range of 0 to 0xffff.

GENODE ⋙ LABS

IRQ allocator contains the IRQs that are associated with IRQ sessions. This allocator is initialized with the entirety of the available IRQ numbers.

Core-region allocator contains the virtual memory regions of core that are not in use.

The RAM allocator and core-region allocator are subsumed in the so-called core-memory allocator. In addition to aggregating both allocators, the core-memory allocator allows for the allocation of core-local virtual-memory regions that can be used for holding core-local objects. Each region allocated from the core-memory allocator has to satisfy three conditions:

1. It must be backed by a physical memory range (as allocated from the RAM allocator)

2. It must have assigned a core-local virtual memory range (as allocated from the core-region allocator)

3. The physical-memory range must have the same size as the virtual-memory range

4. The virtual memory range must be mapped to the physical memory range using the MMU

Internally, the core-memory allocator maintains a so-called mapped-memory allocator that contains ranges of ready-to-use core-local memory. If a new allocation exceeds the available capacity, the core-memory allocator expands its capacity by allocating a new physical memory region from the RAM allocator, allocating a new core-virtual memory region from the core-region allocator, and installing a mapping from the virtual region to the physical region.

All memory allocations mentioned above are performed at the granularity of physical pages, i. e., 4 KiB.

The core-memory allocator is expanded on demand but never shrunk. This makes it unsuitable for allocating objects on behalf of core's clients because allocations could not be reverted when closing a session. It is solely used for dynamic memory allocations at startup (e. g., the memory needed for keeping the information about the boot modules), and for keeping meta data for the allocators themselves.

7.3.2. Kernel-object creation

Kernel objects are objects maintained within the kernel and used by the kernel. The exact notion of what a kernel object represents depends on the actual kernel as the various kernels differ with respect to the abstractions they provide. Typical kernel objects are threads and protection domains. Some kernels have kernel objects for memory mappings while others provide page tables as kernel objects. Whereas some kernels represent scheduling parameters as distinct kernel objects, others subsume scheduling

GENODE ⬈ LABS

parameters to threads. What all kernel objects have in common, though, is that they consume kernel memory. Most kernels of the L4 family preserve a fixed pool of memory for the allocation of kernel objects.

If an arbitrary component were able to perform a kernel operation that triggers the creation of a kernel object, the memory consumption of the kernel would depend on the good behavior of all components. A misbehaving component may exhaust the kernel memory.

To counter this problem, on Genode, only core triggers the creation of kernel objects and thereby guards the consumption of kernel memory. Note, however, that not all kernels are able to prevent the creation of kernel objects outside of core.

7.3.3. Page-fault handling

Each time a thread within the Genode system triggers a page fault, the kernel reflects the page fault along with the fault information as a message to the user-level page-fault handler residing in core. The fault information comprises the identity and instruction pointer of the faulted thread, the page-fault address, and the fault type (read, write, execute). The page-fault handler represents each thread as a so-called *pager object*, which encapsulates the subset of the thread's interface that is needed to handle page faults. For handling the page fault, the page-fault handler first looks up the pager object that belongs to the faulting thread's identity, analogously to how an RPC entry-point looks up the RPC object for an incoming RPC request. Given the pager object, the fault is handled by calling the `pager` function with the fault information as argument. This function is implemented by the so-called `Rm_client` (*repos/base/src/core/region_map_component.cc*), which represents the association of the pager object with its virtual address space (region map). Given the context information about the region map of the thread's PD, the `pager` function looks up the region within the region map, on which the page fault occurred. The lookup results in one of the following three cases:

Region is populated with a dataspace If a dataspace is attached at the fault address, the backing store of the dataspace is determined. Depending on the kernel, the backing store may be a physical page, a core-local page, or another reference to a physical memory page. The pager function then installs a memory mapping from the virtual page where the fault occurred to the corresponding part of the backing store.

Region is populated with a managed dataspace If the fault occurred within a region where a managed dataspace is attached, the fault handling is forwarded to the region map that represents the managed dataspace.

Region is empty If no dataspace could be found at the fault address, the fault cannot be resolved. In this case, core submits an region-map-fault signal to the region map where the fault occurred. This way, the region-map client has the chance to

GENODE ⋛ LABS

detect and possibly respond to the fault. Once the signal handler receives a fault signal, it is able to query the fault address from the region map. As a response to the fault, the region-map client may attach a dataspace at this address. This attach operation, in turn, will prompt core to wake up the thread (or multiple threads) that faulted within the attached region. Unless a dataspace is attached at the page-fault address, the faulting thread remains blocked. If no signal handler for region-map faults is registered for the region map, core prints a diagnostic message and blocks the faulting thread forever.

To optimize the TLB footprint and the use of kernel memory, region maps do not merely operate at the granularity of memory pages but on address ranges whose size and alignment are arbitrary power-of-two values (at least as large as the size of the smallest physical page). The source and destinations of memory mappings may span many pages. This way, depending on the kernel and the architecture, multiple pages may be mapped at once, or large page-table mappings can be used.

GENODE 🖎 LABS

7.4. Asynchronous notification mechanism

Section 3.6.2 introduces asynchronous notifications (signals) as one of the fundamental inter-component communication mechanisms. The description covers the semantics of the mechanism but the question of how the mechanism relates to core and the underlying kernel remains unanswered. This section complements Section 3.6.2 with those implementation details.

Most kernels do not directly support the semantics of asynchronous notifications as presented in Section 3.6.2. As a reminder, the mechanism has the following features:

- The authority for triggering a signal is represented by a signal-context capability, which can be delegated via the common capability-delegation mechanism described in Section 3.1.4.

- The submission of a signal is a fire-and-forget operation. The signal producer is never blocked.

- On the reception of a signal, the signal handler can obtain the context to which the signal refers. This way, it is able to distinguish different sources of events.

- A signal receiver can wait or poll for potentially many signal contexts. The number of signal contexts associated with a single signal receiver is not limited.

The gap between this feature set and the mechanisms provided by the underlying kernel is bridged by core as part of the PD service. This service plays the role of a proxy between the producers and receivers of signals. Each component that interacts with signals has a session to this service.

Within core, a signal context is represented as an RPC object. The RPC object maintains a counter of signals pending for this context. Signal contexts can be created and destroyed by the clients of the PD service using the *alloc_context* and *free_context* RPC functions. Upon the creation of a signal context, the PD client can specify an integer value called *imprint* with a client-local meaning. Later, on the reception of signals, the imprint value is delivered along with the signal to enable the client to tell the contexts of the incoming signals apart. As a result of the allocation of a new signal context, the client obtains a signal-context capability. This capability can be delegated to other components using the regular capability-delegation mechanism.

Signal submission   A component in possession of a signal-context capability is able to trigger signals using the *submit* function of its PD session. The submit function takes the signal context capability of the targeted context and a counter value as arguments. The capability as supplied to the submit function does not need to originate from the called session. It may have been created and delegated by another component. Note that even though a signal context is an RPC object, the submission of a signal is not realized as an invocation of this object. The signal-context capability is merely used

GENODE ⧫ LABS

as an RPC function argument. This design accounts for the fact that signal-context capabilities may originate from untrusted peers as is the case for servers that deliver asynchronous notifications to their clients. A client of such a server supplies a signal-context capability as argument to one of the server's RPC functions. An example is the input session interface (Section 4.5.4) that allows the client to get notified when new user input becomes available. A malicious client may specify a capability that was not created via core's PD service but that instead refers to an RPC object local to the client. If the submit function was an RPC function of the signal context, the server's call of the submit RPC function would eventually invoke the RPC object of the client. This would put the client in a position where it may block the server indefinitely and thereby make the server unavailable to all clients. In contrast to the untrusted signal-context capability, the PD session of a signal producer is by definition trusted. So it is safe to invoke the submit RPC function with the signal-context capability as argument. In the case where an invalid signal-context capability is delegated to the signal producer, core will fail to look up a signal context for the given capability and omit the signal.

Signal reception For receiving signals, a component needs a way to obtain information about pending signals from core. This involves two steps: First, the component needs a way to block until signals are available. Second, if a signal is pending, the component needs a way to determine the signal context and the signal receiver associated with the signal and wake up the thread that blocks the `Signal_receiver::block_for_signal` API function.

Both problems are solved by a dedicated thread that is spawned during component startup. This signal thread blocks at core's PD service for incoming signals. The blocking operation is not directly performed on the PD session but on a decoupled RPC object called *signal source*. In contrast to the PD session interface that is kernel agnostic, the underlying kernel mechanism used for blocking the signal thread at the signal source depends on the used base platform.

The signal-source RPC object implements an RPC interface, on which the PD client issues a blocking *wait_for_signal* RPC function. This function blocks as long as no signal that refers to the session's signal contexts is pending. If the function returns, the return value contains the imprint that was assigned to the signal context at its creation and the number of signals pending for this context. On most base platforms, the implementation of the blocking RPC interface is realized by processing RPC requests and responses out of order to enable one entrypoint in core to serve all signal sources. Core uses a dedicated entrypoint for the signal-source handling to decouple the delivery of signals from potentially long-taking operations of the other core services.

Given the imprint value returned by the signal source, the signal thread determines the signal context and signal receiver that belongs to the pending signal (using a data structure called `Signal_context_registry`) and locally submits the signal to the signal-receiver object. This, in turn, unblocks the `Signal_receiver::block_for_signal`

GENODE ⋙ LABS

function at the API level.

GENODE LABS

7.5. Dynamic linker

The dynamic linker is a mechanism for loading ELF binaries that are dynamically-linked against shared libraries.

7.5.1. Building dynamically-linked programs

The build system automatically decides whether a program is linked statically or dynamically depending on the use of shared libraries. If the target is linked against at least one shared library, the resulting ELF image is a dynamically-linked program. Each dynamically-linked program is implicitly linked against the dynamic linker. Because the dynamic linker contains all the base libraries and the C++ runtime, the program itself must be void of those symbols. Otherwise two ambiguous versions of the base libraries would be present on program startup. This invariant is enforced by the build system by stripping all base libraries from the linking step of a dynamically-linked program. The base libraries are defined in *repos/base/mk/base-libs.mk*.

The entrypoint of a dynamically-linked program is the `main` function.

7.5.2. Startup of dynamically-linked programs

When creating a new component, the parent first detects whether the to-be-loaded ELF binary represents a statically-linked program or a dynamically-linked program by inspecting the ELF binary's program-header information (see *repos/base/src/lib/base/elf_binary.cc*). If the program is statically linked, the parent follows the procedure as described in Section 3.5. If the program is dynamically linked, the parent remembers the dataspace of the program's ELF image but starts the ELF image of the dynamic linker instead.

The dynamic linker is a regular Genode component that follows the startup procedure described in Section 7.1.2. However, because of its hybrid nature, it needs to take special precautions before using any data that contains relocations. Because the dynamic linker is a shared library, it contains data relocations. Even though the linker's code is position independent and can principally be loaded to an arbitrary address, global data objects may contain pointers to other global data objects or code. For example, vtable entries contain pointers to code. Those pointers must be relocated depending on the load address of the binary. This step is performed by the `init_rtld` hook function, which was already mentioned in Section 7.1.2. Global data objects must not be used before calling this function. For this reason, `init_rtld` is called at the earliest possible time directly from the assembly startup code. Apart from the call of this hook function, the startup of the dynamic linker is the same as for statically-linked programs.

The main function of the dynamic linker obtains the binary of the actual dynamically-linked program by requesting a ROM session for the module "binary". The parent responds to this request by handing out a locally-provided ROM session that contains the dataspace of the actual program. Once the linker has obtained the dataspace containing

GENODE ⋙ LABS

the dynamically-linked program, it loads the program and all required shared libraries. The dynamic linker requests each shared library as a ROM session from its parent.

After completing the loading of all ELF objects, the dynamic linker calls the entry-point of the loaded binary, which is the program's `main` function. Note that the symbol `main` remains ambiguous as both the dynamic linker and the loaded program have a main function.

7.5.3. Address-space management

To load the binary and the associated shared libraries, the linker does not directly attach dataspaces to its address space. Instead, it manages a dedicated part of the component's virtual address space called *linker area* manually. The linker area is a region map that is created as part of a PD session. The dynamic linker attaches the linker area as a managed dataspace to its address space. This way, the linker can precisely control the layout within the virtual-address range covered by the managed dataspace. This control is needed because the loading of an ELF object does not correspond to an atomic attachment of a single dataspace but it involves consecutive attach operations for multiple dataspaces, one for each ELF segment. When attaching one segment, the linker must make sure that there is enough space beyond the segment to host the next segment. The use of a managed dataspace allows the linker to manually allocate large-enough portions of virtual memory and populate them in multiple steps.

GENODE ⩘ LABS

7.6. Execution on bare hardware (base-hw)

The code specific to the base-hw platform is located within the *repos/base-hw/* directory. In the following description, unless explicitly stated otherwise, all paths are relative to this directory.

In contrast to classical L4 microkernels where Genode's core process runs as user-level roottask on top of the kernel, base-hw executes Genode's core directly on the hardware with no distinct kernel underneath. Core and the kernel are melted into one hybrid kernel/userland program. Only a few code paths are executed in privileged mode but most code runs in user mode. This design has several benefits. First, the kernel part becomes much simpler. For example, there are no allocators needed in the kernel part because allocators are managed by the user-level part of core. Second, base-hw side-steps long-standing hard kernel-level problems, in particular the management of kernel resources. For the allocation of kernel objects, the hybrid core/kernel can employ Genode's user-level resource trading concepts as described in Section 3.3. Finally and most importantly, merging the kernel with roottask removes a lot of redundancies between both programs. Traditionally, both kernel and roottask perform the book keeping of physical-resource allocations and the existence of kernel objects such as address spaces and threads. In base-hw, those data structures exist only once. The complexity of the combined kernel/core is significantly lower than the sum of the complexities of a traditional self-sufficient kernel and a distinct roottask on top. This way, base-hw helps to make Genode's TCB less complex.

The following subsections detail the problems that base-hw had to address to become a self-sufficient base platform for Genode.

7.6.1. Bootstrapping of base-hw

A Genode-based system consists of potentially many boot modules. But boot loaders on ARM platforms usually merely support the loading of a single system image. Hence, base-hw requires a concept for merging boot modules together with the core/kernel into a single image.

System-image assembly   In contrast to other base platforms where core is a self-sufficient program, on the base-hw platform, core is actually built as a library. The library description file is specific for each platform and located at *lib/mk/spec/<pf>/core.mk* where *<pf>* corresponds to the hardware platform used. It includes the platform-agnostic *lib/mk/core.inc* file. The library contains everything core needs (including the C++ runtime, the kernel code, and the user-level core code) except for the following symbols:

_boot_modules_headers_begin and _boot_modules_headers_end   Between those symbols, core expects an array of boot-module header structures. A boot-module

GENODE ⬚ LABS

header contains the name, core-local address, and size of a boot module. This meta data is used by core's initialization code in *src/core/platform.cc* to populate the ROM service with modules.

_boot_modules_binaries_begin and _boot_modules_binaries_end Between those symbols, core expects the actual module data. This range is outside the core image (beyond _prog_img_end). In contrast to the boot-module headers, the modules reside in a separate section that remains unmapped within core's virtual address space. Only when access to a boot module is required by core (i. e., the ELF binary of init during the creation of the init component), core makes the module visible within its virtual address space.

Making the boot modules invisible to core has two benefits. The integrity of the boot modules does not depend on core. Even in the presence of a bug in core, the boot modules cannot be accidentally overwritten. Second, there are no page-table entries needed to map the modules into the virtual address space of core. This is particularly beneficial when using large boot modules such as a complete disk image. If base-hw incorporated such a large module into the core image, page-table entries for the entire disk image would need to be allocated at the initialization time of core.

These symbols are defined in an assembly file called *boot_modules.s*. If building core stand-alone, the final linking stage combines the core library with the dummy *boot_modules.s* file located at *src/core/boot_modules.s*. But when using the run tool (Section 5.4.1) to integrate a bootable system image, the run tool dynamically generates a version of *boot_modules.s* depending on the boot modules listed in the run script and repeats the final linking stage of core by combining the core library with the generated *boot_modules.s* file. The generated file is placed at *<build-dir>/var/run/<scenario>/* and incorporates the boot modules using the assembler's .incbin directive. The result of the final linking stage is an executable ELF binary that contains both core and the boot modules.

Startup of the base-hw kernel Core on base-hw uses Genode's regular linker script. Like any regular Genode component, its execution starts at the _start symbol. But unlike a regular component, core is started by the boot loader as a kernel in privileged mode. Instead of directly following the startup procedure described in Section 7.1.2, base-hw uses custom startup code that initializes the kernel part of core first. The startup code is located at *src/core/spec/arm/crt0.s*. It eventually calls the kernel initialization code in *src/core/kernel/kernel.cc*. Core's regular C++ startup code (the _main function) is executed by the first user-level thread created by the kernel (see the thread setup in the init_kernel_mp_primary function).

7.6.2. Kernel entry and exit

The execution model of the kernel can be roughly characterized as a single-stack kernel. In contrast to traditional L4 kernels that maintain one kernel thread per user thread, the base-hw kernel is a mere state machine that never blocks in the kernel. State transitions are triggered by user-level threads that enter the kernel via a system call, by device interrupts, or by a CPU exception. Once entered, the kernel applies the state change depending on the event that caused the kernel entry, and leaves the kernel to the user land. The transition between user and kernel mode depends on the revision of the ARM architecture. For ARMv7, the corresponding code is located at *src/core/spec/arm_v7/mode_transition.s*.

7.6.3. Interrupt handling and preemptive multi-threading

In order to respond to interrupts, base-hw has to contain a driver for the interrupt controller. ARM-based SoCs greatly differ with respect to the used interrupt controllers. The interrupt-controller driver for a particular SoC can be found at *src/core/include/spec/<spec>/pic.h* and the corresponding *src/core/spec/<spec>/pic.cc* where *<spec>* refers to a particular platform (e. g., imx53) or an IP block that is is used across different platforms (e. g., arm_gic for ARM's generic interrupt controller). Each of the drivers implement the same interface. When building core, the build system uses the build-spec mechanism explained in Section 5.3 to incorporate the single driver needed for the targeted SoC.

To support preemptive multi-threading, base-hw requires a hardware timer. The timer is programmed with the timeslice length of the currently executed thread. Once the programed timeout elapses, the timer device generates an interrupt that is handled by the kernel. Similarly to interrupt controllers, there exist a variety of different timer devices on ARM-based SoCs. Therefore, base-hw contains different timer drivers. The timer drivers are located at *src/core/include/spec/<spec>/timer.h* where *<spec>* refers to the timer variant.

The in-kernel handler of the timer interrupt invokes the thread scheduler (*src/core/include/kernel/cpu_scheduler.h*). The scheduler maintains a list of so-called scheduling contexts where each context refers to a thread. Each time the kernel is entered, the scheduler is updated with the passed duration. When updated, it takes a scheduling decision by making the next to-be-executed thread the head of the list. At kernel exit, the control is passed to the user-level thread that corresponds to the head of the scheduler list.

7.6.4. Split kernel interface

The system-call interface of the base-hw kernel is split into two parts. One part is usable by all components and solely contains system calls for inter-component communication and thread synchronization. The definition of this interface is located at

GENODE ⟋⟍ LABS

include/kernel/thread.h. The second part is exposed only to core. It supplements the public interface with operations for the creation, the management, and the destruction of kernel objects. The definition of the core-private interface is located at *src/core/include/kernel/core_interface.h*.

The distinction between both parts of the kernel interface is enforced by the function `Thread::_call` in *src/core/kernel/thread.cc*.

7.6.5. Public part of the kernel interface

Threads do not run independently but interact with each other via synchronous inter-component communication as detailed in Section 3.6. Within base-hw, this mechanism is referred to as IPC (for inter-process communication). To allow threads to perform calls to other threads or to receive RPC requests, the kernel interface is equipped with system calls for performing IPC (*send_request_msg*, *await_request_msg*, *send_reply_msg*). To keep the kernel as simple as possible, IPC is performed using so-called user-level thread-control blocks (UTCB). Each thread has a corresponding memory page that is always mapped in the kernel. This UTCB page is used to carry IPC payload. The largely simplified procedure of transferring a message is as follows. (In reality, the state space is more complex because the receiver may not be in a blocking state when the sender issues the message)

1. The user-level sender marshals its payload into its UTCB and invokes the kernel,

2. The kernel transfers the payload from the sender's UTCB to the receiver's UTCB and schedules the receiver,

3. The receiver retrieves the incoming message from its UTCB.

Because all UTCBs are always mapped in the kernel, no page faults can occur during the second step. This way, the flow of execution within the kernel becomes predictable and always returns to the user land.

In addition to IPC, threads interact via the synchronization primitives provided by the Genode API. To implement these portions of the API, the kernel provides system calls for managing the execution control of threads (*pause_current_thread*, *resume_local_thread*, *yield_thread*).

To support asynchronous notifications as described in Section 3.6.2, the kernel provides system calls for the submission and reception of signals (*await_signal*, *signal_pending*, *submit_signal*, and *ack_signal*) as well as the life-time management of signal contexts (*kill_signal_context*). In contrast to other base platforms, Genode's signal API is directly supported by the kernel so that the propagation of signals does not require any interaction with core's PD service. However, the creation of signal contexts is arbitrated by the PD service. This way, the kernel objects needed for the signalling mechanism are accounted to the corresponding clients of the PD service.

GENODE ⬙ LABS

The kernel provides an interface to make the kernel's scheduling timer available as time source to the userland. Using this interface, user-level components can bind signal contexts to timeouts (*timeout*) and follow the progress of time (*timeout_age_us* and *timeout_max_us*).

7.6.6. Core-private part of the kernel interface

The core-private part of the kernel interface allows the user-level part of core to perform privileged operations. Note that even though the kernel and core are executed in different CPU modes (privileged mode and user mode), both parts share a single address space and ultimately trust each other. The kernel is regarded a mere support library of core that executes those functions that can only be executed in the privileged CPU mode. In particular, the kernel does not perform any allocation. Instead, the allocation of kernel objects is performed as an interplay of core and the kernel.

1. Core allocates physical memory from its physical-memory allocator. Most kernel-object allocations are performed in the context of one of core's services. Hence, those allocations can be properly accounted to a session quota (Section 3.3). This way, kernel objects allocated on behalf of core's clients are "paid for" by those clients.

2. Core allocates virtual memory to make the allocated physical memory visible within core and the kernel.

3. Core invokes the kernel to construct the kernel object at the location specified by core. This kernel invocation is actually a system call that enters the kernel via the kernel-entry path.

4. The kernel initializes the kernel object at the virtual address specified by core and returns to core via the kernel-exit path.

The core-private kernel interface consists of the following operations:

- The creation and destruction of protection domains (*new_pd*, *update_pd*, *delete_pd*), invoked by the PD service

- The creation, manipulation, and destruction of threads (*new_thread*, *start_thread*, *resume_thread*, *thread_quota*, *pause_thread*, *delete_thread*, *access_thread_regs*, and *route_thread_event*), used by the CPU service and the core-specific back end of the Genode::Thread API

- The creation and destruction of signal receivers and signal contexts (*new_signal_receiver*, *delete_signal_receiver*, *new_signal_context*, and *delete_signal_context*), invoked by the PD service

- The creation and destruction of kernel-protected object identities (*new_obj*, *delete_obj*)

GENODE ⤳ LABS

7.6.7. Scheduler of the base-hw kernel

CPU scheduling in traditional L4 microkernels is based on static priorities. The scheduler always picks the runnable thread with highest priority for execution. If multiple threads share one priority, the kernel schedules those threads in a round-robin fashion. Whereas being pretty fast and easy to implement, this scheme has disadvantages: First, there is no way to prevent high-prioritized threads from starving lower-prioritized ones. Second, CPU time cannot be granted to threads and passed between them by the means of quota. To cope with these problems without much loss of performance, base-hw employs a custom scheduler that deviates from the traditional approach.

The base-hw scheduler introduces the distinction between high-throughput-oriented scheduling contexts - called *fills* - and low-latency-oriented scheduling contexts - called *claims*. Examples for typical fills would be the processing of a compiler job or the rendering computations of a sophisticated graphics program. They shall obtain as much CPU time as the system can spare but there is no demand for a high responsiveness. In contrast, an example for the claim category would be a typical GUI-software stack covering the control flow from user-input drivers through a chain of GUI components to the drivers of the graphical output. Another example is a user-level device driver that must quickly respond to sporadic interrupts but is otherwise untrusted. The low latency of such components is a key factor for usability and quality of service. Besides introducing the distinction between claim and fill scheduling contexts, base-hw introduces the notion of a so-called *super period*, which is a multiple of typical scheduling time slices, e. g., one second. The entire super period corresponds to 100% of the CPU time of one CPU. Portions of it can be assigned to scheduling contexts. A CPU quota thereby corresponds to a percentage of the super period.

At the beginning of a super period, each claim has its full amount of assigned CPU quota. The priority defines the absolute scheduling order within the super period among those claims that are active and have quota left. As long as there exist such claims, the scheduler stays in the claim mode and the quota of the scheduled claims decreases. At the end of a super period, the quota of all claims is replenished to the initial value. Every time the scheduler can't find an active claim with CPU-quota left, it switches to the fill mode. Fills are scheduled in a simple round-robin fashion with identical time slices. The proceeding of the super period doesn't affect the scheduling order and time-slices of this mode. The concept of quota and priority that is implemented through the claim mode aligns nicely with Genode's way of hierarchical resource management: Through CPU sessions, each process becomes able to assign portions of its CPU time and subranges of its priority band to its children without knowing the global meaning of CPU time or priority.

GENODE ⋙ LABS

7.6.8. Sparsely populated core address space

Even though core has the authority over all physical memory, it has no immediate access to the physical pages. Whenever core requires access to a physical memory page, it first has to explicitly map the physical page into its own virtual memory space. This way, the virtual address space of core stays clean from any data of other components. Even in the presence of a bug in core (e.g., a dangling pointer), information cannot accidentally leak between different protection domains because the virtual memory of the other components is not visible to core.

7.6.9. Multi-processor support of base-hw

On uniprocessor systems, the base-hw kernel is single-threaded. Its execution model corresponds to a mere state machine. On SMP systems, it maintains one kernel thread and one scheduler per CPU core. Access to kernel objects gets fully serialized by one global spin lock that is acquired when entering the kernel and released when leaving the kernel. This keeps the use of multiple cores transparent to the kernel model, which greatly simplifies the code compared to traditional L4 microkernels. Given that the kernel is a simple state machine providing lightweight non-blocking operations, there is little contention for the global kernel lock. Even though this claim may not hold up when scaling to a large number of cores, current ARM-based platforms can be accommodated well.

Cross-CPU inter-component communication Regarding synchronous and asynchronous inter-processor communication - thanks to the global kernel lock - there is no semantic difference to the uniprocessor case. The only difference is that on a multiprocessor system, one processor may change the schedule of another processor by unblocking one of its threads (e.g., when an RPC call is received by a server that resides on a different CPU as the client). This condition may rescind the current scheduling choice of the other processor. To avoid lags in this case, the kernel lets the unaware target processor trap into an inter-processor interrupt (IPI). The targeted processor can react to the IPI by taking the decision to schedule the receiving thread. As the IPI sender does not have to wait for an answer, the sending and receiving CPUs remain largely decoupled. There is no need for a complex IPI protocol between sender and receiver.

TLB shootdown With respect to the synchronization of core-local hardware, there are two different situations to deal with. Some hardware components like most ARM caches and branch predictors implement their own coherence protocol and thus need adaption in terms of configuration only. Others, like the TLBs lack this feature. When for instance a page table entry gets invalid, the TLB invalidation of the affected entries must be performed locally by each core. To signal the necessity of TLB maintenance

GENODE ⚞ LABS

work, an IPI is sent to all other cores. Once all cores have completed the cleaning, the thread that invoked the TLB invalidation resumes its execution.

7.6.10. Asynchronous notifications on base-hw

The base-hw platform improves the mechanism described in Section 7.4 by introducing signal receivers and signal contexts as first-class kernel objects. Core's PD service is merely used to arbitrate the creation and destruction of those kernel objects but it does not play the role of a signal-delivery proxy. Instead, signals are communicated directly by using the public kernel operations *await_signal*, *signal_pending*, *submit_signal*, and *ack_signal*.

GENODE ⟋ LABS

7.7. Execution on the NOVA microhypervisor (base-nova)

NOVA is a so-called microhypervisor, denoting the combination of microkernel and a virtualization platform (hypervisor). It is a high-performance microkernel for the x86 architecture. In contrast to other microkernels, it has been designed for hardware-based virtualization via user-level virtual-machine monitors. In line with Genode's architecture, NOVA's kernel interface is based on capability-based security. Hence, the kernel fully supports the model of a Genode kernel as described in Section 3.1.

NOVA website

   http://hypervisor.org

NOVA kernel-interface specification

   https://github.com/udosteinberg/NOVA/raw/master/doc/specification.pdf

7.7.1. Integration of NOVA with Genode

The NOVA kernel is available via Genode's ports mechanism detailed in Section 5.2. The port description is located at *repos/base-nova/ports/nova.port*.

Building the NOVA kernel   Even though NOVA is a third-party kernel with a custom build system, the kernel is built directly by the Genode build system. NOVA's build system remains unused.

   From within a Genode build directory configured for one of the nova_x86_32 or nova_x86_64 platforms, the kernel can be built via

```
make kernel
```

   The build description for the kernel is located at *repos/base-nova/src/kernel/target.mk*.

System-call bindings   NOVA is not accompanied with bindings to its kernel interface. There only is a description of the kernel interface in the form of the kernel specification available. For this reason, Genode maintains the kernel bindings for NOVA within the Genode source tree. The bindings are located at *repos/base-nova/include/* in the subdirectories *nova/*, *spec/32bit/nova/*, and *spec/64bit/nova/*.

7.7.2. Bootstrapping of a NOVA-based system

After finishing its initialization, the kernel starts the second boot module, the first being the kernel itself, as root task. The root task is Genode's core. The virtual address space of core contains the text and data segments of core, the UTCB of the initial execution context (EC), and the hypervisor info page (HIP). Details about the HIP are provided in Section 6 of the NOVA specification.

GENODE ⫸ LABS

BSS section of core   The kernel's ELF loader does not support the concept of a BSS segment. It simply maps the physical pages of core's text and data segments into the virtual memory of core but does not allocate any additional physical pages for backing the BSS. For this reason, the NOVA version of core does not use the *genode.ld* linker script as described in Section 7.1.1 but the linker script located at *repos/base-nova/src/core/core.ld*. This version hosts the BSS section within the data segment. Thereby, the BSS is physically present in the core binary in the form of zero-initialized data.

Initial information provided by NOVA to core   The kernel passes a pointer to the HIP to core as the initial value of the ESP register. Genode's startup code saves this value in the global variable `_initial_sp` (Section 7.1.2).

7.7.3. Log output on modern PC hardware

Because transmitting information over legacy comports does not require complex device drivers, serial output over comports is still the predominant way to output low-level system logs like kernel messages or the output of core's LOG service.

Unfortunately, most modern PCs lack dedicated comports. This leaves two options to obtain low-level system logs.

1. The use of vendor-specific platform-management features such as Intel VPro / Intel Advanced Management Technology (AMT) or Intel Platform Management Interface (IPMI). These platform features are able to emulate a legacy comport and provide the serial output over the network. Unfortunately, those solutions are not uniform across different vendors, difficult to use, and tend to be unreliable.

2. The use of a PCI card or an Express Card that provides a physical comport. When using such a device, the added comport appears as PCI I/O resource. Because the device interface is compatible to the legacy comports, no special drivers are needed.

The latter option allows the retrieval of low-level system logs on hardware that lacks special management features. In contrast to the legacy comports, however, it has the minor disadvantage that the location of the device's I/O resources is not known beforehand. The I/O port range of the comport depends on the device-enumeration procedure of the BIOS. To enable the kernel to output information over this comport, the kernel must be configured with the I/O port range as assigned by the BIOS on the specific machine. One kernel binary cannot simply be used across different machines.

The Bender chain boot loader   To alleviate the need to adapt the kernel configuration to the used comport hardware, the bender chain boot loader can be used.

GENODE ⬳ LABS

Bender is part of the MORBO tools

https://github.com/TUD-OS/morbo

Instead of starting the NOVA hypervisor directly, the multi-boot-compliant boot loader (such as GRUB) starts bender as the kernel. All remaining boot modules including the real kernel have already been loaded into memory by the original boot loader. Bender scans the PCI bus for a comport device. If such a device is found (e. g., an Express Card), it writes the information about the device's I/O port range to a known offset within the BIOS data area (BDA).

After the comport-device probing is finished, bender passes control to the next boot module, which is the real kernel. The comport device driver of the kernel does not use a hard-coded I/O port range for the comport but looks up the comport location in the BDA. The use of bender is optional. When not used, the BDA always contains the I/O port range of the legacy comport 1.

The Genode source tree contains a pre-compiled binary of bender at *tool/boot/bender*. This binary is automatically incorporated into boot images for the NOVA base platform when the run tool (Section 5.4.1) is used.

7.7.4. Relation of NOVA's kernel objects to Genode's core services

For the terminology of NOVA's kernel objects, refer to the NOVA specification mentioned in the introduction of Section 7.7. A brief glossary for the terminology used in the remainder of this section is given in table 2.

NOVA term	
PD	Protection domain
EC	Execution context (thread)
SC	Scheduling context
HIP	Hypervisor information page
IDC	Inter-domain call (RPC call)
portal	communication endpoint

Table 2: Glossary of NOVA's terminology

NOVA capabilities are not Genode capabilities   Both NOVA and Genode use the term "capability". However, the term does not have the same meaning in both contexts. A Genode capability refers to an RPC object or a signal context. In the context of NOVA, a capability refers to a NOVA kernel object. To avoid confusing both meanings of the term, Genode refers to NOVA's term as "capability selector", or simply "selector". A Genode signal context capability corresponds to a NOVA semaphore, all other Genode capabilities correspond to NOVA portals.

GENODE ⬙ LABS

PD service   A PD session corresponds to a NOVA PD.

A Genode capability being a NOVA portal has a defined IP and an associated local EC (the Genode entrypoint). The invocation of a such a Genode capability is an IDC call to a portal. A Genode capability is delegated by passing its corresponding portal or semaphore selector as IDC argument.

Page faults are handled as explained in Section 7.7.5. Each memory mapping installed in a component implicitly triggers the allocation of a node in the kernel's mapping database.

CPU service   NOVA distinguishes between so-called global ECs and local ECs. A global EC can be equipped with CPU time by associating it with an SC. It can perform IDC calls but it cannot receive IDC calls. In contrast to a global EC, a local EC is able to receive IDC calls but it has no CPU time. A local EC is not executed before it is called by another EC.

A regular Genode thread is a global EC. A Genode entrypoint is a local EC. Core distinguishes both cases based on the instruction-pointer (IP) argument of the CPU session's start function. For a local EC, the IP is set to zero.

RAM and IO_MEM services   Core's RAM and IO_MEM allocators are initialized based on the information found in NOVA's HIP.

ROM service   Core's ROM service provides all boot modules as ROM modules. Additionally, a copy of NOVA's HIP is provided as a ROM module named "hypervisor_info_page".

IRQ service   NOVA represents each interrupt as a semaphore created by the kernel. By registration of a Genode signal context capability via the sigh method of the Irq_session interface, the semaphore of the signal context capability is bound to the interrupt semaphore. Genode signals and NOVA semaphores are handled as described in 7.7.6.

Upon the initial IRQ session's *ack_irq* call, a NOVA semaphore-down operation is issued within core on the interrupt semaphore, which implicitly unmasks the interrupt at the CPU. When the interrupt occurs, the kernel masks the interrupt at the CPU and performs the semaphore-up operation on the IRQ's semaphore. Thereby, the chained semaphore, which is the beforehand registered Genode signal context, is triggered and the interrupt is delivered as Genode signal. The interrupt gets acknowledged and unmasked by calling the IRQ session's *ack_irq* method.

GENODE ⋙ LABS

7.7.5. Page-fault handling on NOVA

On NOVA, each EC has a pre-defined range of portal selectors. For each type of exception, the range has a dedicated portal that is entered in the event of an exception. The page-fault portal of a Genode thread is defined at the creation time of the thread and points to a pager EC per CPU within core. Hence, for each CPU, a pager EC in core pages all Genode threads running on the same CPU.

The operation of pager ECs   When an EC triggers a page fault, the faulting EC implicitly performs an IDC call to its pager. The IDC message contains the fault information. On NOVA, there is a one-to-one relationship between a pager EC and Genode's pager object. For resolving the page fault, core follows the procedure described in 7.3.3. If the lookup for a dataspace within the faulter's region map succeeds, core establishes a memory mapping into the EC's PD by sending a so-called map item as reply to the page fault message. In the case where the region lookup within the thread's corresponding region map fails, the pager EC blocks on a semaphore. Because the page-fault message remains unanswered, the faulting thread is effectively put on halt. In the event that the fault is resolved by a region-map client as described in the paragraph "Region is empty" of Section 7.3.3, the blocking on the semaphore gets released and the pager EC is able to reply to the original page-fault message. However, the reply does not immediately establish a memory mapping. Instead, the faulter will immediately trigger another fault at the same address. This time, however, the region lookup succeeds.

Mapping database   NOVA tracks memory mappings in a data structure called *mapping database* and has the notion of the delegation of memory mappings (rather than the delegation of memory access). Memory access can be delegated only if the originator of the delegation has a mapping. Core is the only exception because it can establish mappings originating from the physical memory space. Because mappings can be delegated transitively between PDs, the mapping database is a tree where each node denotes the delegation of a mapping. The tree is maintained in order to enable the kernel to rescind the authority. When a mapping is revoked, the kernel implicitly cancels all transitive mappings that originated from the revoked node.

7.7.6. Asynchronous notifications on NOVA

To support asynchronous notifications as described in Section 3.6.2, we extended the NOVA kernel semaphores to support signalling via chained NOVA semaphores. This extension enables the creation of kernel semaphores with a per-semaphore value, which can be bound to another kernel semaphore. Each bound semaphore corresponds to a Genode signal context. The per-semaphore value is used to distinguish different sources of signals.

GENODE ⬙ LABS

On this base platform, the blocking of the signal thread at the signal source is realized by using a kernel semaphore shared by the PD session and the PD client. All chained semaphores (Signal contexts) are bound to this semaphore. When first issuing a *wait-for-signal* operation at the signal source, the client requests a capability selector for the shared semaphore *(repos/base-nova/include/signal_session/source_client.h)*. It then performs a *down* operation on this semaphore to block.

If a signal sender issues a submit operation on a Genode signal capability, then a regular NOVA kernel semaphore-up syscall is used. If the kernel detects that the used semaphore is chained to another semaphore, the up operation is delegated to the one received during the initial *wait-for-signal* operation of the signal receiving thread.

In contrast to other base platforms, Genode's signal API is supported by the kernel so that the propagation of signals does not require any interaction with core's PD service. However, the creation of signal contexts is arbitrated by the PD service.

7.7.7. IOMMU support

As discussed in Section 4.1.3, misbehaving device drivers may exploit DMA transactions to circumvent their component boundaries. When executing Genode on the NOVA microhypervisor, however, bus-master DMA is subjected to the IOMMU.

The NOVA kernel applies a subset of the (MMU) address space of a protection domain to the (IOMMU) address space of a device. So the device's address space can be managed in the same way as one normally manages the address space of a PD. The only missing link is the assignment of device address spaces to PDs. This link is provided by the dedicated system call *assign_pci* that takes a PD capability selector and a device identifier as arguments. The PD capability selector represents the authorization over the protection domain, which is going to be targeted by DMA transactions. The device identifier is a virtual address where the extended PCI configuration space of the device is mapped in the specified PD. Only if a user-level device driver has access to the extended PCI configuration space of the device, is it able to get the assignment in place.

To make NOVA's IOMMU support available to Genode, the ACPI driver has the ability to lookup the extended PCI configuration space region for all devices and reports it via a Genode ROM. The platform driver on x86 evaluates the reported ROM and uses the information to obtain transparently for platform clients (device drivers) the extended PCI configuration space per device. The platform driver uses a NOVA-specific extension (*assign_pci*) to the PD session interface to associate a PCI device with a protection domain.

Even though these mechanisms combined should in theory suffice to let drivers operate with the IOMMU enabled, in practice, the situation is a bit more complicated. Because NOVA uses the same virtual-to-physical mappings for the device as it uses for the process, the DMA addresses the driver needs to supply to the device must be virtual addresses rather than physical addresses. Consequently, to be able to make a

GENODE ⟋ LABS

device driver usable on systems without IOMMU as well as on systems with IOMMU, the driver needs to become IOMMU-aware and distinguish both cases. This is an unfortunate consequence of the otherwise elegant mechanism provided by NOVA. To relieve the device drivers from worrying about both cases, Genode decouples the virtual address space of the device from the virtual address space of the driver. The former address space is represented by a Genode component called *device PD*. Its sole purpose is to hold mappings of DMA buffers that are accessible by the associated device. By using one-to-one physical-to-virtual mappings for those buffers within the device PD, each device PD contains a subset of the physical address space. The platform driver performs the assignment of device PDs to PCI devices. If a device driver intends to use DMA, it allocates a new DMA buffer for a specific PCI device at the platform driver. The platform driver responds to such a request by allocating a RAM dataspace at core, attaching it to the device PD using the dataspace's physical address as virtual address, and by handing out the dataspace capability to the client. If the driver requests the physical address of the dataspace, the address returned will be a valid virtual address in the associated device PD. This design implies that a device driver must allocate DMA buffers at the platform driver (specifying the PCI device the buffer is intended for) instead of using core's RAM service to allocate buffers anonymously.

7.7.8. Genode-specific modifications of the NOVA kernel

NOVA is not ready to be used as a Genode base platform as is. This section compiles the modifications that were needed to meet the functional requirements of the framework. All modifications are maintained at the following repository:

Genode's version of NOVA

> https://github.com/alex-ab/NOVA.git

The repository contains a separate branch for each version of NOVA that has been used by Genode. When preparing the NOVA port using the port description at *repos/base-nova/ports/nova.port*, the NOVA branch that matches the used Genode version is checked out automatically. The port description refers to a specific commit ID. The commit history of each branch within the NOVA repository corresponds to the history of the original NOVA kernel followed by a series of Genode-specific commits. Each time NOVA is updated, a new branch is created and all Genode-specific commits are rebased on top of the history of the new NOVA version. This way, the differences between the original NOVA kernel and the Genode version remain clearly documented. The Genode-specific modifications solve the following problems:

Destruction of kernel objects

> NOVA does not support the destruction of kernel objects. I.e., PDs and ECs can be created but not destroyed. With Genode being a dynamic system, kernel-object destruction is a mandatory feature.

GENODE 🗲 LABS

Inter-processor IDC

On NOVA, only local ECs can receive IDC calls. Furthermore each local EC is bound to a particular CPU (hence the name "local EC"). Consequently, synchronous inter-component communication via IDC calls is possible only between ECs that both reside on the same CPU but can never cross CPU boundaries. Unfortunately, IDC is the only mechanism for the delegation of capabilities. Consequently, authority cannot be delegated between subsystems that reside on different CPUs. For Genode, this scheme is too rigid.

Therefore, the Genode version of NOVA introduces inter-CPU IDC calls. When calling an EC on another CPU, the kernel creates a temporary EC and SC on the target CPU as a representative of the caller. The calling EC is blocked. The temporary EC uses the same UTCB as the calling EC. Thereby, the original IDC message is effectively transferred from one CPU to the other. The temporary EC then performs a local IDC to the destination EC using NOVA's existing IDC mechanism. Once the temporary EC receives the reply (with the reply message contained in the caller's UTCB), the kernel destroys the temporary EC and SC and unblocks the caller EC.

Support for priority-inheriting spinlocks

Genode's lock mechanism relies on a yielding spinlock for protecting the lock meta data. On most base platforms, there exists the invariant that all threads of one component share the same CPU priority. So priority inversion within a component cannot occur. NOVA breaks this invariant because the scheduling parameters (SC) are passed along IDC call chains. Consequently, when a client calls a server, the SCs of both client and server reside within the server. These SCs may have different priorities. The use of a naive spinlock for synchronization will produce priority inversion problems. The kernel has been extended with the mechanisms needed to support the implementation of priority-inheriting spinlocks in userland.

Combination of capability delegation and translation

As described in Section 3.1.4, there are two cases when a capability is specified as an RPC argument. The callee may already have a capability referring to the specified object identity. In this case, the callee expects to receive the corresponding local name of the object identity. In the other case, when the callee does not yet have a capability for the object identity, it obtains a new local name that refers to the delegated capability.

NOVA does not support this mechanism per se. When specifying a capability selector as map item for an IDC call, the caller has to specify whether a new mapping should be created or the translation of the local names should be performed by the kernel. However, in the general case, this question is not decidable by the

GENODE 🎓 LABS

caller. Hence, NOVA had to be changed to take the decision depending on the existence of a valid translation for the specified capability selector.

Support for deferred page-fault resolution

With the original version of NOVA, the maximum number of threads is limited by core's stack area: NOVA's page-fault handling protocol works completely synchronously. When a page fault occurs, the faulting EC enters its page-fault portal and thereby activates the corresponding pager EC in core. If the pager's lookup for a matching dataspace within the faulter's region map succeeds, the page fault is resolved by delegating a memory mapping as the reply to the page-fault IDC call. However, if a page fault occurs on a managed dataspace, the pager cannot resolve it immediately. The resolution must be delayed until the region-map fault handler (outside of core) responds to the fault signal. In order to enable core to serve page faults of other threads in the meantime, each thread has its dedicated pager EC in core.

Each pager EC, in turn, consumes a slot in the stack area within core. Since core's stack area is limited, the maximum number of ECs within core is limited too. Because one core EC is needed as pager for each thread outside of core, the available stacks within core become a limited resource shared by all CPU-session clients. Because each Genode component is a client of core's CPU service, this bounded resource is effectively shared among all components. Consequently, the allocation of threads on NOVA's version of core represents a possible covert storage channel.

To avoid the downsides described above, we extended the NOVA IPC reply system call to specify an optional semaphore capability selector. The NOVA kernel validates the capability selector and blocks the faulting thread in the semaphore. The faulted thread remains blocked even after the pager has replied to the fault message. But the pager immediately becomes available for other page-fault requests. With this change, it suffices to maintain only one pager thread per CPU for all client threads.

The benefits are manifold. First, the base-nova implementation converges more closely to other Genode base platforms. Second, core can not run out of threads anymore as the number of threads in core is fixed for a given setup. And the third benefit is that the helping mechanism of NOVA can be leveraged for concurrently faulting threads.

Remote revocation of memory mappings In the original version of NOVA, roottask must retain mappings to all memory used throughout the system. In order to be able to delegate a mapping to another PD as response of a page fault, it must possess a local mapping of the physical page. Otherwise, it would not be able to revoke the mapping later on because the kernel expects roottask's mapping node

GENODE LABS

as a proof of the authorization for the revocation of the mapping. Consequently, even though roottask never touches memory handed out to other components, it needs to have memory mappings with full access rights installed within its virtual address space.

To relieve Genode's roottask (core) from the need to keep local mappings for all memory handed out to other components and thereby let core benefit from a sparsely populated address space as described in Section 7.6.8 for base-hw, we changed the kernel's revoke operation to take a PD selector and a virtual address within the targeted PD as argument. By presenting the PD selector as a token of authorization over the entire PD, we do no longer need core-locally installed mappings as the proof of authorization. Hence, memory mappings can always be installed directly from the physical address space to the target PD.

Support for write-combined access to memory-mapped I/O resources The original version of NOVA is not able to benefit from write combining because the kernel interface does not allow the userland to specify cacheability attributes for memory mappings. To achieve good throughput to the framebuffer, write combining is crucial. Hence, we extended the kernel interface to allow the userland to propagate cacheability attributes to the page-table entries of memory mappings and set up the x86 page attribute table (PAT) with a configuration for write combining.

Support for the virtualization of 64-bit guest operating systems The original version of NOVA supports 32-bit guest operations only. We enhanced the kernel to also support 64-bit guests.

Resource quotas for kernel resources The NOVA kernel lacks the ability to adopt the kernel memory pool to the behavior of the userland. The kernel memory pool has a fixed size, which cannot be changed at runtime. Even though we have not removed this principal limitation, we extended the kernel with the ability to subject kernel-memory allocations to a userlevel policy at the granularity of PDs. Each kernel operation that consumes kernel memory is accounted to a PD whereas each PD has a limited quota of kernel memory. This measure prevents arbitrary userland programs to bring down the entire system by exhausting the kernel memory. The reach of damage is limited to the respective PD.

Asynchronous notification mechanism We extended the NOVA kernel semaphores to support signalling via chained NOVA semaphores. This extension enables the creation of kernel semaphores with a per-semaphore value, which can be bound to another kernel semaphore. Each bound semaphore corresponds to a Genode signal context. The per-semaphore value is used to distinguish different sources of signals. Now, a signal sender issues a submit operation on a Genode signal capability via a regular NOVA semaphore-up syscall. If the kernel detects that the

GENODE ⋙ LABS

used semaphore is chained to another semaphore, the up operation is delegated to the chained one. If a thread is blocked, it gets woken up directly and the per-semaphore value of the bound semaphore gets delivered. In case no thread is currently blocked, the signal is stored and delivered as soon as a thread issues the next semaphore-down operation.

Chaining semaphores is an operation that is limited to a single level, which avoids attacks targeting endless loops in the kernel. The creation of such signals can solely be performed if the issuer has a NOVA PD capability with the semaphore-create permission set. On Genode, this effectively reserves the operation to core. Furthermore, our solution preserves the invariant of the original NOVA kernel that a thread may be blocked in only one semaphore at a time.

Interrupt delivery We applied the same principle of the asynchronous notification extension to the delivery of interrupts by the NOVA kernel. Interrupts are delivered as ordinary Genode signals, which alleviate of the need for one thread per interrupt as required by the original NOVA kernel. The interrupt gets directly delivered to the address space of the driver in case of a Message Signalled Interrupt (MSI), or in case of a shared interrupt, to the x86 platform driver.

7.7.9. Known limitations of NOVA

This section summarizes the known limitations of NOVA and the NOVA version of core.

Fixed amount of kernel memory NOVA allocates kernel objects out of a memory pool of a fixed size. The pool is dimensioned in the kernel's linker script *nova/src/hypervisor.ld* (at the symbol `_mempool_f`).

Bounded number of object capabilities within core For each capability created via core's PD service, core allocates the corresponding NOVA portal or NOVA semaphore and maintains the capability selector during the lifetime of the associated object identity. Each allocation of a capability via core's PD service consumes one entry in core's capability space. Because the space is bounded, clients of the service could misuse core's capability space as covert storage channel.

Non-executable (NX) bit NOVA does not support the NX bit.

GENODE ⫟ LABS

Part II.
Reference

GENODE LABS

8. Functional specification

This chapter complements the architectural information of Chapters 3 and 4 with a thorough description of the framework's C++ programming interface. The material is partially generated from the source code, specifically the public header files of the *base* and *os* source-code repositories. The location of the headers in either both repositories depends on role of the respective header. Only if the interface is fundamentally required by the core component or by the base framework itself, it is contained in the base repository. Otherwise, the header is found in the *os* repository.

Scope of the API   The Genode API covers everything needed by Genode to be self-sufficient without any dependency on 3rd-party code. It does not even depend on a C runtime. It is possible to create components that solely use the raw Genode API, as is the case for the ones hosted within the *repos/os/* repository. Because such components do not rely on any library code except for the low-complexity Genode base libraries, they are relatively easy to evaluate. That said, the API must not be mistaken as an application-level API. It does not present an alternative to established application-level libraries like the standard C++ library. For example, even though the framework contains a few utilities for processing strings, those utilities do merely exist to support the use cases present in Genode.

General conventions   The functional specification adheres to the following general conventions:

- Static member functions are called *class functions*. All other member functions are called *methods*.

- Both structs and classes are referred to as *classes*. There is no distinction between both at the API level.

- Constant functions with no argument and a return value are called *accessors*. Technically, they are "getters" but since Genode does not have any "setters", accessors are synonymous to "getters". Because those functions are so simple, they are listed in the class overview and are not described separately.

- Classes that merely inherit other classes are called *sub types*. In contrast to the use of a typedef, such classes are not mere name aliases but represent a new type. The type is compatible to the base class but not vice versa. For example, exception types are sub types of the Exception class.

- Namespace-wide information is presented with a light yellow background.

- Each non-trivial class is presented in the form of a class diagram on a blue-shaded background. The diagram shows the public base classes and the list of public

GENODE ⬢ LABS

class functions and methods. If the class is a template, the template arguments are annotated at the top-right of corner of the class. If the class represents an RPC interface, an information box is attached. The methods listed in the diagram can be used as hyperlinks to further descriptions if available.

- Method descriptions are presented on a green-shaded background, global functions are presented on an orange-shaded background.

- The directory path presented under "Header" sections is a hyperlink to the corresponding source code at GitHub.

GENODE LABS

8.1. API primitives

8.1.1. Capability types

As described in Section 3.1, inter-component communication is based on capabilities. A capability refers to a system-wide unique object identity and can be delegated among components. At API level, each capability is associated with the type of the RPC interface the capability refers to - similar to how a C++ reference refers to the type of a specific C++ object.

Genode	**Capability**	
Capability referring to a specific RPC interface		class template

Untyped_capability

RPC_INTERFACE

Capability

Capability(...)
Capability()

Template argument

RPC_INTERFACE	**typename**
	Class containing the rpc interface declaration

Header

repos/base/include/base/capability.h

Genode	Capability	**Capability**
		constructor template

Template argument

FROM_RPC_INTERFACE	**typename**

Argument

cap	**Capability<FROM_RPC_INTERFACE> const &**

Details

This implicit constructor checks at compile time for the compatibility of the source and target capability types. The construction is performed only if the target capability type is identical to or a base type of the source capability type.

GENODE LABS

Genode › Capability › **Capability**

Default constructor creates invalid capability   constructor

8.1.2. Sessions and connections

Servers provide their services over session-based communication channels. A Session type is defined as an abstract interface inherited from the `Session` base class.

Genode ⟩ **Session**	
Base class of session interfaces	class
`Session`	
Header	
repos/base/include/session/session.h	
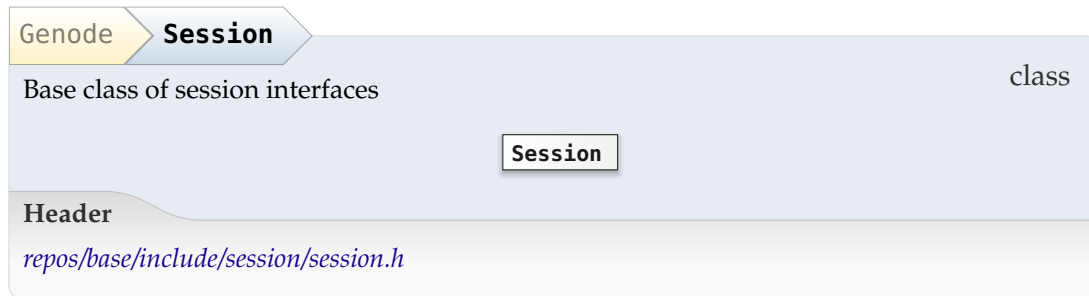
Each session interface has to provide an implementation of the following class function that returns the name of the service as constant string.

```
static const char *service_name();
```

This function is used by the framework for the announcement of the service's root interface at the component's parent. The string returned by this function corresponds to the service name as used in the system configuration (Section 6).

The interaction of a client with a server involves the definition of session-construction arguments, the request of the session creation via its parent, the initialization of the matching RPC-client stub code with the received session capability, the actual use of the session interface, and the closure of the session. The `Connection` template class provides a way to greatly simplify the handling of session arguments, session creation, and destruction on the client side. By implementing a service-specific connection class inherited from `Connection`, session arguments become plain constructor arguments, session functions can be called directly on the `Connection` object, and the session gets properly closed when destructing the `Connection`.

GENODE ⚎ LABS

Genode   **Connection**

Representation of an open connection to a service

class template

Noncopyable

SESSION_TYPE

Connection

Connection(...)
~Connection()
cap() : Capability<SESSION_TYPE>
on_destruction(...)
session(...) : Capability<SESSION_TYPE>
session(...) : Capability<SESSION_TYPE>

Template argument

SESSION_TYPE   **typename**

Header

repos/base/include/base/connection.h

Genode   Connection   **Connection**

constructor

Arguments

env   **Env &**

cap   **Capability**<**SESSION_TYPE**>
      Session capability

od   **On_destruction**
      Session policy applied when destructing the connection
      Default is CLOSE

Details

The op argument defines whether the session should automatically be closed by the destructor of the connection (CLOSE), or the session should stay open (KEEP_OPEN). The latter is useful in situations where the creator a connection merely passes the session capability of the connection to another party but never invokes any of the session's RPC functions.

GENODE LABS

Genode > Connection > **cap**

const method

Return value

Capability<SESSION_TYPE>   Session capability

Genode > Connection > **on_destruction**

method

Define session policy

Argument

od   **On_destruction**

Genode > Connection > **session**

method

Issue session request to the parent

Arguments

parent   **Parent &**

format_args **const char ***

-            **...**

Return value

Capability<SESSION_TYPE>

Genode > Connection > **session**

method

Issue session request to the parent

Arguments

parent   **Parent &**

affinity   **Affinity const &**

format_args **char const ***

-            **...**

Return value

Capability<SESSION_TYPE>

GENODE LABS

8.1.3. Dataspace interface

The dataspace abstraction described in Section 3.4.1 is the fundamental API primitive for representing a container of memory as provided by core's RAM, IO_MEM, or ROM services. Each dataspace is referenced by a capability that can be passed among components. Each component with the capability to a dataspace can access the dataspace's content by attaching the dataspace to the region map of its PD session. In addition to the use as arguments for region-map operations, dataspaces provide the following interface.

Genode › **Dataspace**

Dataspace interface   class

```
            Dataspace
  ────────────────────────────
  ~Dataspace()
  size() : size_t                ····· RPC interface
  phys_addr() : addr_t
  writable() : bool
```

Header

repos/base/include/dataspace/dataspace.h

Genode › Dataspace › *size*

Request size of dataspace   pure virtual method

Return value

  size_t

Genode › Dataspace › *phys_addr*

Request base address in physical address space   pure virtual method

Return value

  addr_t

Genode › Dataspace › *writable*

                                                     pure virtual method
Return value

  bool   True if dataspace is writable

214

GENODE LABS

GENODE LABS

Attached dataspace   As a utility for handling the common case where a dataspace is attached to the component's address space as long as a certain object (like a session) exists, an instance of an `Attached_dataspace` can be hosted as a member variable. When destructed, the dataspace will be automatically detached from the component's address space.

Genode **Attached_dataspace**

class

Utility to attach a dataspace to the local address space

Attached_dataspace
Attached_dataspace(...)
~Attached_dataspace()
cap() : Dataspace_capability
local_addr() : T *
local_addr() : T const *
size() : size_t
invalidate()

Header

repos/base/include/base/attached_dataspace.h

Genode Attached_dataspace **Attached_dataspace**

constructor

Arguments

rm   **Region_map &**

ds   **Dataspace_capability**

Exceptions

Rm_session::Attach_failed
Invalid_dataspace

Genode Attached_dataspace **cap**

const method

Return value

Dataspace_capability   Capability of the used dataspace

GENODE LABS

Genode	Attached_dataspace	**local_addr**	method template

Request local address

Template argument

  T   **typename**

Return value

  T ∗

Details

This is a template to avoid inconvenient casts at the caller. A newly attached dataspace is untyped memory anyway.

Genode	Attached_dataspace	**local_addr**	const method template

Template argument

  T   **typename**

Return value

  T const ∗

Genode	Attached_dataspace	**size**	const method

Return value

  size_t   Size

Genode	Attached_dataspace	**invalidate**	method

Forget dataspace, thereby skipping the detachment on destruction

Details

This method can be called if the the dataspace is known to be physically destroyed, e. g., because the session where the dataspace originated from was closed. In this case, core will already have removed the memory mappings of the dataspace. So we have to omit the detach operation in ~Attached_dataspace.

GENODE LABS

8.2. Component execution environment

Each component has to provide an implementation of the `Component` interface as illustrated by the example given in Section 2.5.2.

Component   namespace

Hook functions for bootstrapping the component

Functions

- `stack_size() : Genode::size_t`
- `construct(...)`

Header

repos/base/include/base/component.h

Component   **stack_size**   global function

Return value

`Genode::size_t`   Stack size of the component's initial entrypoint

Component   **construct**   global function

Construct component

Argument

env   **Env &**
      Interface to the component's execution environment

Genode   namespace

Hook functions for bootstrapping the component

Header

repos/base/include/base/component.h

8.2.1. Interface to the component's environment

As described in Section 3.5, each component is a composition of a protection domain (PD session), a memory budget (RAM session), and a CPU session, from which the main thread is created. These sessions form the *environment* of the component, which is

GENODE ⚒ LABS

represented by the Env interface class. The environment is provided to the component as argument to the Component::construct function.

Genode ⟩ **Env**

Component environment

class

Env
parent() : Parent &
ram() : Ram_session &
cpu() : Cpu_session &
rm() : Region_map &
pd() : Pd_session &
ep() : Entrypoint &
ram_session_cap() : Ram_session_capability
cpu_session_cap() : Cpu_session_capability

Header

repos/base/include/base/env.h

Genode ⟩ Env ⟩ *parent*

pure virtual method

Return value

Parent &

Genode ⟩ Env ⟩ *ram*

RAM session of the component

pure virtual method

Return value

Ram_session &

Details

The RAM Session represents a budget of memory (quota) that is available to the component. This budget can be used to allocate RAM dataspaces.

GENODE LABS

Genode > Env > *cpu*

CPU session of the component   pure virtual method

Return value

`Cpu_session &`

Details

This session is used to create the threads of the component.

Genode > Env > *rm*

Region map of the component's address space   pure virtual method

Return value

`Region_map &`

Genode > Env > *pd*

PD session of the component as created by the parent   pure virtual method

Return value

`Pd_session &`

Genode > Env > *ep*

Entrypoint for handling RPC requests and signals   pure virtual method

Return value

`Entrypoint &`

Genode > Env > *ram_session_cap*

pure virtual method

Return value

`Ram_session_capability`

Genode > Env > *cpu_session_cap*

pure virtual method

Return value

`Cpu_session_capability`

GENODE LABS

221

8.2.2. Parent interface

At its creation time, the only communication partner of a component is its immediate parent. The parent can be reached via interface returned by `Env::parent()`.

Genode	**Parent**		class

Parent interface

```
                          Parent

              ~Parent()
              exit(...)
              announce(...)
              announce(...)
              session(...) : Session_capability
              session(...) : Capability<SESSION_TYPE>    ......... RPC interface
              upgrade(...)
              close(...)
              main_thread_cap() : Thread_capability
              resource_avail_sigh(...)
              resource_request(...)
              yield_sigh(...)
              yield_request() : Resource_args
              yield_response()
```

Header

repos/base/include/parent/parent.h

Genode	Parent	*exit*

Tell parent to exit the program   pure virtual method

Argument

exit_value   **int**

Genode	Parent	*announce*

Announce service to the parent   pure virtual method

Arguments

service_name   **Service_name const &**

service_root   **Root_capability**

GENODE LABS

Genode 〉 Parent 〉 **announce**

Announce service to the parent   method template

Template argument

ROOT_INTERFACE   **typename**

Argument

service_root   **Capability<ROOT_INTERFACE> const &**
                Root capability

Details

The type of the specified service_root capability match with an interface that provides a Session_type type (i. e., a Typed_root interface). This Session_type is expected to host a class function called service_name returning the name of the provided interface as null-terminated string.

Genode 〉 Parent 〉 *session*

Create session to a service   pure virtual method

Arguments

service_name   **Service_name const &**
                Name of the requested interface

args   **Session_args const &**
                Session constructor arguments

affinity   **Affinity const &**
                Preferred cpu affinity for the session
                Default is Affinity()

Exceptions

Service_denied   Parent denies session request
Quota_exceeded   Our own quota does not suffice for the creation of the new session
Unavailable

Return value

Session_capability   Untyped capability to new session

Details

The use of this method is discouraged. Please use the type safe session() template instead.

GENODE 〳 LABS

Genode ⟩ Parent ⟩ **session**

Create session to a service   method template

Template argument

SESSION_TYPE   **typename**
                Session interface type

Arguments

args   **Session_args const &**
            Session constructor arguments

affinity   **Affinity const &**
            Preferred cpu affinity for the session
            Default is Affinity()

Exceptions

Service_denied   Parent denies session request
Quota_exceeded   Our own quota does not suffice for the creation of the new session
Unavailable

Return value

Capability<**SESSION_TYPE**>   Capability to new session

Genode ⟩ Parent ⟩ *upgrade*

Transfer our quota to the server that provides the specified ses-   pure virtual method
sion

Arguments

to_session   **Session_capability**
              Recipient session

args   **Upgrade_args const &**
              Description of the amount of quota to transfer

Exception

Quota_exceeded   Quota could not be transferred

Details

The args argument has the same principle format as the args argument of the session
operation. The error case indicates that there is not enough unused quota on the source
side.

GENODE LABS

Genode › Parent › *close*

pure virtual method

Close session

Argument

session   **Session_capability**

Genode › Parent › *main_thread_cap*

pure virtual const method

Provide thread_cap of main thread

Return value

Thread_capability

Genode › Parent › *resource_avail_sigh*

pure virtual method

Register signal handler for resource notifications

Argument

sigh   **Signal_context_capability**

Genode › Parent › *resource_request*

pure virtual method

Request additional resources

Argument

args   **Resource_args const &**

Details

By invoking this method, a process is able to inform its parent about the need for additional resources. The argument string contains a resource description in the same format as used for session-construction arguments. In particular, for requesting additional RAM quota, the argument looks like "ram_quota=<amount>" where amount is the amount of additional resources expected from the parent. If the parent complies with the request, it submits a resource-available signal to the handler registered via resource_avail_sigh(). On the reception of such a signal, the process can re-evaluate its resource quota and resume execution.

GENODE LABS

Genode > Parent > *yield_sigh*

Register signal handler for resource yield notifications   pure virtual method

Argument

sigh   **Signal_context_capability**

Details

Using the yield signal, the parent is able to inform the process about its wish to regain resources.

Genode > Parent > *yield_request*

Obtain information about the amount of resources to free   pure virtual method

Return value

Resource_args

Details

The amount of resources returned by this method is the goal set by the parent. It is not commanded but merely meant as a friendly beg to cooperate. The process is not obligated to comply. If the process decides to take action to free resources, it can inform its parent about the availability of freed up resources by calling yield_response().

Genode > Parent > *yield_response*

Notify the parent about a response to a yield request   pure virtual method

GENODE LABS

8.3. Entrypoint

An entrypoint is a thread that is able respond to RPC requests and signals. Each component has at least one initial entrypoint that is created as part of the component's environment. It can be accessed via the `Env::ep()` method.

The `Entrypoint::manage` and `Entrypoint::dissolve` methods are used to associate respectively disassociate signal handlers and RPC objects with the entrypoint. Under the hood, those operations interact with the component's PD session in order to bind the component's signal handlers and RPC objects to capabilities.

Note that the current version of the `Entrypoint` utilizes the former 'Rpc_entrypoint' and `Signal_receiver` APIs. The entrypoint is designated to eventually replace those APIs. Hence, application code should no longer use the `Rpc_entrypoint` and `Signal_receiver` directly.

Genode ⟩ **Entrypoint**

Entrypoint for serving RPC requests and dispatching signals   class

```
                         Entrypoint

  Entrypoint(...)
  manage(...) : Capability<RPC_INTERFACE>
  dissolve(...)
  manage(...) : Signal_context_capability
  dissolve(...)
  rpc_ep() : Rpc_entrypoint &
  schedule_suspend()
```

Header

repos/base/include/base/entrypoint.h

Genode ⟩ Entrypoint ⟩ **Entrypoint**

                                                                                     constructor

Arguments

env	**Env &**
stack_size	**size_t**
name	**char const ***

GENODE LABS
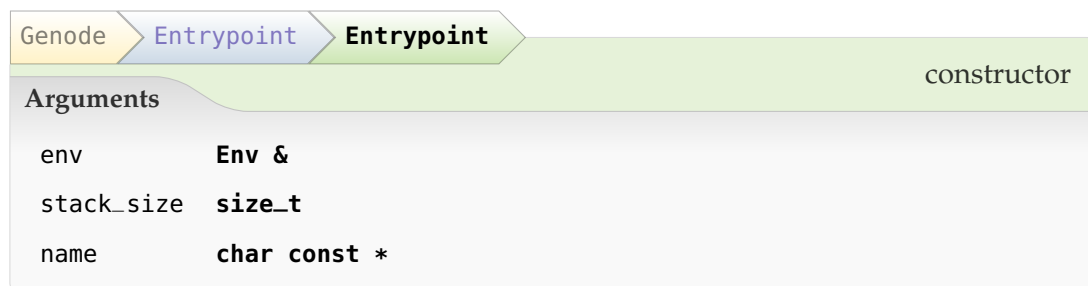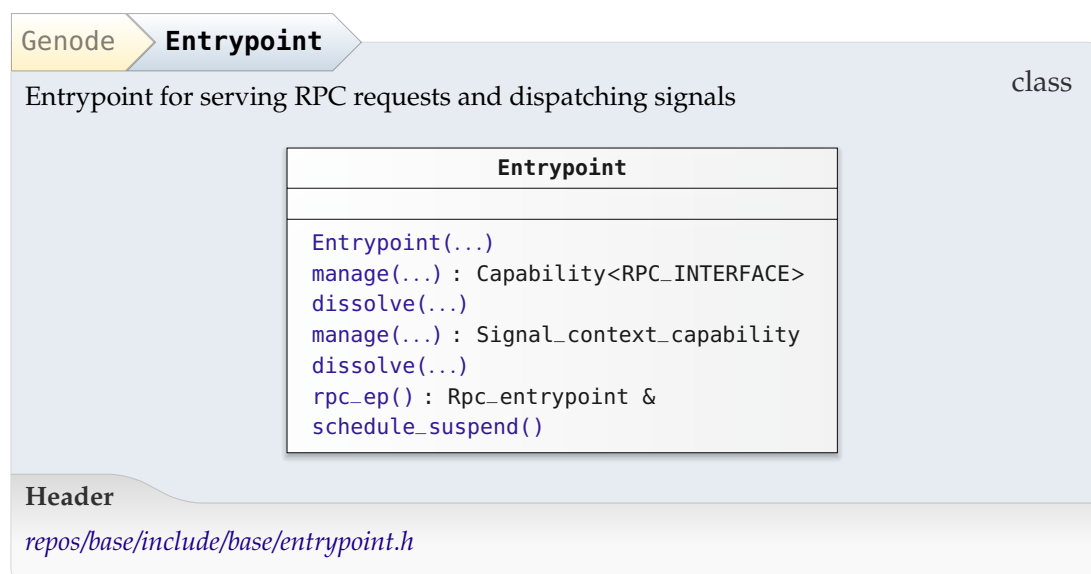
Genode › Entrypoint › **manage**

Associate RPC object with the entry point   method template

Template arguments

RPC_INTERFACE	**typename**
RPC_SERVER	**typename**

Argument

obj   **Rpc_object**<**RPC_INTERFACE, RPC_SERVER**> **&**

Return value

Capability<**RPC_INTERFACE**>

Genode › Entrypoint › **dissolve**

Dissolve RPC object from entry point   method template

Template arguments

RPC_INTERFACE	**typename**
RPC_SERVER	**typename**

Argument

obj   **Rpc_object**<**RPC_INTERFACE, RPC_SERVER**> **&**

Genode › Entrypoint › **manage**

Associate signal dispatcher with entry point   method

Argument

-    **Signal_dispatcher_base &**

Return value

Signal_context_capability

GENODE LABS

Genode ⟩ Entrypoint ⟩ **dissolve**

method

Disassociate signal dispatcher from entry point

Argument

- **Signal_dispatcher_base &**

Genode ⟩ Entrypoint ⟩ **rpc_ep**

method

Return value

Rpc_entrypoint &   Rpc entrypoint

Genode ⟩ Entrypoint ⟩ **schedule_suspend**

method

Trigger a suspend-resume cycle in the entrypoint

Details

suspended is called after the entrypoint entered the safe suspend state, while 'resumed is called when the entrypoint is fully functional again.

GENODE ⟋ LABS

8.4. Region-map interface

A region map represents a (portion of) a virtual memory address space (Section 3.4.2). By attaching dataspaces to a region map, the content of those dataspaces becomes visible in the component's virtual address space. Furthermore, a region map can be used as a dataspace. Thereby, the nesting of region maps becomes possible. This allows a component to manage portions of its own virtual address space manually as done for the stack area and linker area (Section 8.5.3). The use of region maps as so-called managed dataspaces makes it even possible to manage portions of the virtual address space of other components (Section 8.5.4).

GENODE LABS

Genode **Region_map**

Region map interface   class

Region_map
attach(...) : Local_addr
attach_at(...) : Local_addr
attach_executable(...) : Local_addr
detach(...)
fault_handler(...)
state() : State
dataspace() : Dataspace_capability

RPC interface

Header

repos/base/include/region_map/region_map.h

Genode Region_map ***attach***

Map dataspace into local address space   pure virtual method

Arguments

ds   **Dataspace_capability**
                 Capability of dataspace to map

size   **size_t**
                 Size of the locally mapped region default (0) is the whole dataspace
                 Default is `0`

offset   **off_t**
                 Start at offset in dataspace (page-aligned)
                 Default is `0`

use_local_addr   **bool**
                 If set to true, attach the dataspace at the specified `local_addr`
                 Default is `false`

local_addr   **Local_addr**
                 Local destination address
                 Default is `(void *)0`

executable   **bool**
                 If the mapping should be executable
                 Default is `false`

Exceptions

Attach_failed   If dataspace or offset is invalid, or on region conflict
Out_of_metadata   If meta-data backing store is exhausted

Return value

Local_addr   Local address of mapped dataspace

231

Details

GENODE LABS

Genode	Region_map	**attach_at**		method

Shortcut for attaching a dataspace at a predefined local address

Arguments

ds	**Dataspace_capability**
local_addr	**addr_t**
size	**size_t** *Default is 0*
offset	**off_t** *Default is 0*

Return value

Local_addr

Genode	Region_map	**attach_executable**		method

Shortcut for attaching a dataspace executable at a predefined local address

Arguments

ds	**Dataspace_capability**
local_addr	**addr_t**
size	**size_t** *Default is 0*
offset	**off_t** *Default is 0*

Return value

Local_addr

Genode	Region_map	*detach*		pure virtual method

Remove region from local address space

Argument

local_addr	**Local_addr**

GENODE LABS

Genode	Region_map	*fault_handler*

Register signal handler for region-manager faults

pure virtual method

Argument

handler **Signal_context_capability**

Details

On Linux, this signal is never delivered because page-fault handling is performed by the Linux kernel. On microkernel platforms, unresolvable page faults (traditionally called segmentation fault) will result in the delivery of the signal.

Genode	Region_map	*state*

Request current state of region map

pure virtual method

Return value

State

Genode	Region_map	*dataspace*

pure virtual method

Return value

Dataspace_capability   Dataspace representation of region map

8.5. Session interfaces of the base API

8.5.1. RAM session interface

The RAM service (Section 3.4.3) provides access to physical memory. Each RAM session corresponds to a memory budget with a bounded quota. From this budget, the RAM-session client can allocate memory blocks in the form of dataspaces.

Genode	namespace
RAM session interface	

Types	
`Ram_dataspace`	is a subtype of `Dataspace`
`Ram_dataspace_capability`	is defined as `Capability<Ram_dataspace>`

Header
repos/base/include/ram_session/ram_session.h

GENODE LABS

Genode **Ram_session**

RAM session interface

class

Session

Ram_session
service_name() : const char *
~Ram_session()
alloc(...) : Ram_dataspace_capability
free(...)
ref_account(...) : int
transfer_quota(...) : int
quota() : size_t
used() : size_t
avail() : size_t

RPC interface

Header

repos/base/include/ram_session/ram_session.h

Genode Ram_session **service_name**

class function

Return value

const char *

GENODE LABS
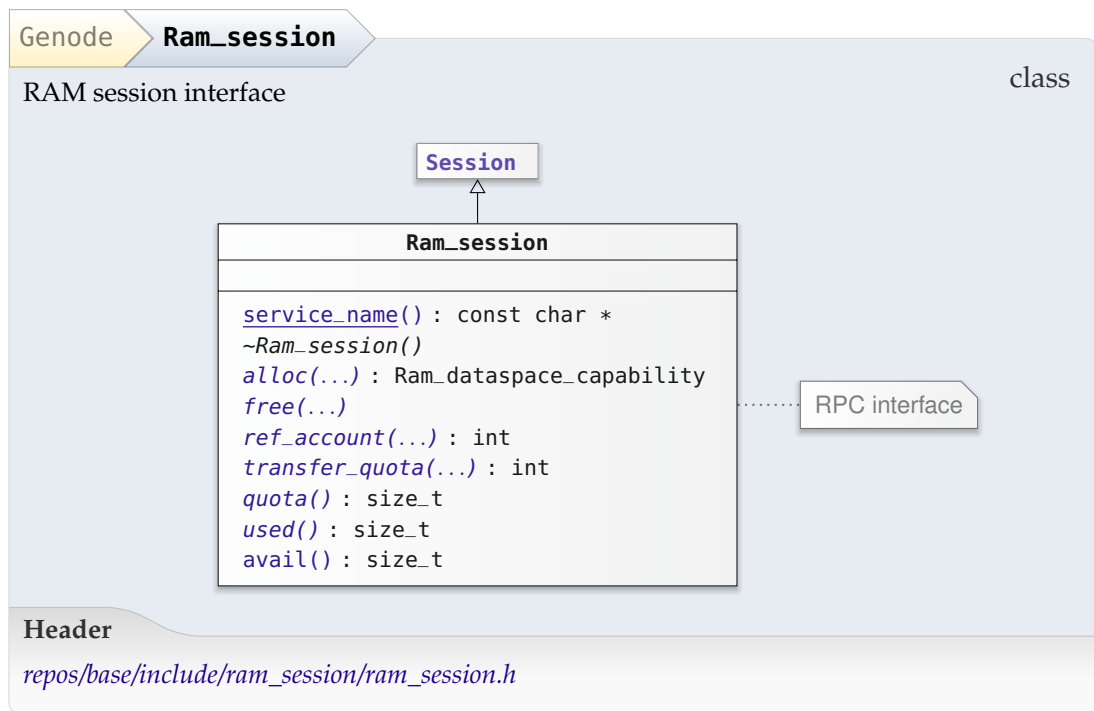
Genode	Ram_session	*alloc*	

Allocate RAM dataspace

pure virtual method

Arguments

size **size_t**
Size of ram dataspace

cached **Cache_attribute**
Selects cacheability attributes of the memory, uncached memory, i. e., for dma buffers
Default is CACHED

Exceptions

Quota_exceeded
Out_of_metadata

Return value

Ram_dataspace_capability Capability to new ram dataspace

Genode	Ram_session	*free*	

Free RAM dataspace

pure virtual method

Argument

ds **Ram_dataspace_capability**
Dataspace capability as returned by alloc

Genode	Ram_session	*ref_account*	

Define reference account for the RAM session

pure virtual method

Argument

ram_session **Ram_session_capability**
Reference account

Return value

int 0 on success

Details

Each RAM session requires another RAM session as reference account to transfer quota to and from. The reference account can be defined only once.

GENODE LABS

Genode	Ram_session	*transfer_quota*	

Transfer quota to another RAM session

pure virtual method

Arguments

ram_session **Ram_session_capability**
Receiver of quota donation

amount **size_t**
Amount of quota to donate

Return value

int 0 on success

Details

Quota can only be transfered if the specified RAM session is either the reference account for this session or vice versa.

Genode	Ram_session	*quota*

pure virtual method

Return value

size_t Current quota limit

Genode	Ram_session	*used*

pure virtual method

Return value

size_t Used quota

Genode	Ram_session	**avail**

method

Return value

size_t Amount of available quota

237

GENODE LABS

Genode > **Ram_connection**

Connection to RAM service

class

Connection <Ram_session>   Ram_session_client

Ram_connection

Ram_connection(...)

Header

repos/base/include/ram_session/connection.h

Genode > Ram_connection > **Ram_connection**

constructor

Arguments

env	**Env &**
label	**const char *** Session label *Default is* `""`
phys_start	**unsigned long** *Default is* `0UL`
phys_size	**unsigned long** *Default is* `0UL`

Attached RAM dataspace The instantiation of an `Attached_ram_dataspace` object
subsumes the tasks of allocating a dataspaces from the component's RAM session and
attaching the dataspace to the component's region map. Furthermore, the reverse op-
erations are performed during the destruction of an `Attached_ram_dataspace` object.

GENODE LABS

Genode ⟩ **Attached_ram_dataspace**

class

Utility to allocate and locally attach a RAM dataspace

Attached_ram_dataspace
Attached_ram_dataspace(...)
~Attached_ram_dataspace()
cap() : Ram_dataspace_capability
local_addr() : T *
size() : size_t
swap(...)
realloc(...)

Header

repos/base/include/base/attached_ram_dataspace.h

Genode ⟩ Attached_ram_dataspace ⟩ **Attached_ram_dataspace**

constructor

Arguments

ram	**Ram_session &**
rm	**Region_map &**
size	**size_t**
cached	**Cache_attribute**
	Default is CACHED

Exceptions

Ram_session::Alloc_failed
Rm_session::Attach_failed

Genode ⟩ Attached_ram_dataspace ⟩ **cap**

const method

Return value

Ram_dataspace_capability   Capability of the used ram dataspace

GENODE LABS

Genode	Attached_ram_dataspace	**local_addr**

Request local address — const method template

Template argument

T   **typename**

Return value

T *

Details

This is a template to avoid inconvenient casts at the caller. A newly allocated RAM dataspace is untyped memory anyway.

Genode	Attached_ram_dataspace	**size**

const method

Return value

size_t   Size

Genode	Attached_ram_dataspace	**swap**

method

Argument

other   **Attached_ram_dataspace &**

Genode	Attached_ram_dataspace	**realloc**

Re-allocate dataspace with a new size — method

Arguments

ram_session   **Ram_session** *

new_size   **size_t**

Details

The content of the original dataspace is not retained.

GENODE LABS

8.5.2. ROM session interface

The ROM service (Section 4.5.1) provides access to ROM modules, e. g., binary data loaded by the boot loader (core's ROM service described in Section 3.4.4). Each session refers to one ROM module. The module's data is provided to the client in the form of a dataspace (Section 3.4.1).

Genode

ROM session interface

Types

   Rom_dataspace   is a subtype of `Dataspace`

   Rom_dataspace_capability   is defined as `Capability<Rom_dataspace>`

Header

repos/base/include/rom_session/rom_session.h

namespace

Genode   **Rom_session**

                                             class

            Session

```
              ┌─────────────────────────────────────┐
              │            Rom_session              │
              ├─────────────────────────────────────┤
              ├─────────────────────────────────────┤
              │ service_name() : const char *       │
              │ ~Rom_session()                      │
              │ dataspace() : Rom_dataspace_capability │····· RPC interface
              │ update() : bool                     │
              │ sigh(...)                           │
              └─────────────────────────────────────┘
```

Header

repos/base/include/rom_session/rom_session.h

Genode   Rom_session   **service_name**

Return value

  const char *

class function

GENODE ⋙ LABS

Genode   Rom_session   *dataspace*

Request dataspace containing the ROM session data

pure virtual method

Return value

Rom_dataspace_capability   Capability to rom dataspace

Details

The capability may be invalid.

Consecutive calls of this method are not guaranteed to return the same dataspace as dynamic ROM sessions may update the ROM data during the lifetime of the session. When calling the method, the server may destroy the old dataspace and replace it with a new one containing the updated data. Hence, prior calling this method, the client should make sure to detach the previously requested dataspace from its local address space.

Genode   Rom_session   *update*

Update ROM dataspace content

virtual method

Return value

bool   True if the existing dataspace contains up-to-date content, or false if a new dataspace must be requested via the dataspace method

Details

This method is an optimization for use cases where ROM dataspaces are updated at a high rate. In such cases, requesting a new dataspace for each update induces a large overhead because memory mappings must be revoked and updated (e. g., handling the page faults referring to the dataspace). If the updated content fits in the existing dataspace, those costly operations can be omitted.

When this method is called, the server may replace the dataspace content with new data.

GENODE ⟋⟍ LABS

Genode › Rom_session › *sigh*

Register signal handler to be notified of ROM data changes

pure virtual method

Argument

sigh   **Signal_context_capability**

Details

The ROM session interface allows for the implementation of ROM services that dynamically update the data exported as ROM dataspace during the lifetime of the session. This is useful in scenarios where this data is generated rather than originating from a static file, for example to update a program's configuration at runtime.

By installing a signal handler using the sigh() method, the client will receive a notification each time the data changes at the server. From the client's perspective, the original data contained in the currently used dataspace remains unchanged until the client calls dataspace() the next time.

GENODE LABS

```
Genode    Rom_connection                                        class

Connection to ROM file service

        Connection <Rom_session>    Rom_session_client

                        Rom_connection

                        Rom_connection(...)

Header

repos/base/include/rom_session/connection.h
```

```
Genode    Rom_connection    Rom_connection

                                                    constructor
Arguments

  env         Env &

  module_name const char *
              Name of rom module

  label       const char *
              Initial session label
              Default is 0

Exception

  Rom_connection_failed
```

Attached ROM dataspace   By instantiating an `Attached_rom_dataspace` object, a ROM module is requested and made visible within the component's address space in a single step.

To accommodate the common use of a ROM session as provider of configuration data (Section 4.6) or an XML-formatted data model in a publisher-subscriber scenario 4.7.5, the `Attached_rom_dataspace` provides a convenient way to retrieve its content as an XML node via the `xml` method. The method always returns a valid `Xml_node` even in the event where the dataspace is invalid or contains no XML. In such cases, the returned XML node is `<empty/>`. This relieves the caller from handling exceptions that may occur during XML parsing.

GENODE ⟋⟋ LABS

Genode › **Attached_rom_dataspace**

class

Utility to open a ROM session and locally attach its content

Attached_rom_dataspace
Attached_rom_dataspace(...)
cap() : Dataspace_capability
local_addr() : T *
local_addr() : T const *
size() : size_t
sigh(...)
update()
valid() : bool
xml() : Xml_node

Accessor

size **size_t**

Header

repos/base/include/base/attached_rom_dataspace.h

Genode › Attached_rom_dataspace › **Attached_rom_dataspace**

constructor

Arguments

env **Env &**

name **char const ***

Exceptions

Rom_connection::Rom_connection_failed
Rm_session::Attach_failed

Genode › Attached_rom_dataspace › **cap**

const method

Return value

Dataspace_capability Capability of the used dataspace

GENODE LABS

Genode 〉 Attached_rom_dataspace 〉 **local_addr**

method template

Template argument

T **typename**

Return value

T *

Genode 〉 Attached_rom_dataspace 〉 **local_addr**

const method template

Template argument

T **typename**

Return value

T const *

Genode 〉 Attached_rom_dataspace 〉 **sigh**

method

Register signal handler for ROM module changes

Argument

sigh **Signal_context_capability**

Genode 〉 Attached_rom_dataspace 〉 **update**

method

Update ROM module content, re-attach if needed

Genode 〉 Attached_rom_dataspace 〉 **valid**

const method

Return value

bool True of content is present

GENODE LABS

Genode	Attached_rom_dataspace	xml

const method

Return value

Xml_node   Dataspace content as xml node

Details

This method always returns a valid XML node. It never throws an exception. If the dataspace is invalid or does not contain properly formatted XML, the returned XML node has the form "<empty/>".

GENODE LABS

8.5.3. PD session interface

The protection-domain (PD) service (Section 3.4.5) enables the creation of address spaces that are isolated from each other. Each PD session corresponds to a protection domain. Each PD session is equipped with three region maps, one representing the PD's virtual address space, one for the component's stack area, and one for the component's linker area designated for shared objects. Of those region maps, only the virtual address space is needed by application code directly. The stack area and linker area are used by the framework internally. The PD service is rarely needed by applications directly but it is internally.

Besides representing the component's virtual address space, a PD session also entails the component's capability space. It thereby enables the component to create globally unique RPC-object capabilities and signal-context capabilities.

Once allocated from a PD session, a capability can be associated with a local object or a signal handler. It can be passed to another protection domain via an RPC call whereby the receiving protection domain obtains the right to send messages to the associated object or to trigger the associated signal handler respectively. The capability-management operations are not used directly by components at the API level but are used indirectly via the RPC mechanism described in Section 8.14 and the signalling API described in Section 8.13.

GENODE ⤳ LABS

Genode   **Pd_session**

Protection domain (PD) session interface   class

Session

```
                        Pd_session

    service_name() : const char *
    ~Pd_session()
    assign_parent(...)
    assign_pci(...) : bool
    alloc_signal_source() : Signal_source_capability
    free_signal_source(...)
    alloc_context(...) : Capability<Signal_context>
    free_context(...)
    submit(...)
    alloc_rpc_cap(...) : Native_capability
    free_rpc_cap(...)
    address_space() : Capability<Region_map>
    stack_area() : Capability<Region_map>
    linker_area() : Capability<Region_map>
    native_pd() : Capability<Native_pd>
```

RPC interface

Header

repos/base/include/pd_session/pd_session.h

Genode   Pd_session   **service_name**

Return value   class function

 const char *

Genode   Pd_session   *assign_parent*

Assign parent to protection domain   pure virtual method

Argument

 parent   **Capability**<**Parent**>
          Capability of parent interface

GENODE LABS

Genode 〉 Pd_session 〉 *assign_pci*

Assign PCI device to PD   pure virtual method

Arguments

  pci_config_memory_address   **addr_t**

  bdf   **uint16_t**

Return value

  bool

Details

The specified address has to refer to the locally mapped PCI configuration space of the device.
This function is solely used on the NOVA kernel.

Genode 〉 Pd_session 〉 *alloc_signal_source*

Create a new signal source   pure virtual method

Exception

  Out_of_metadata

Return value

  Signal_source_capability   A cap that acts as reference to the created source

Details

The signal source provides an interface to wait for incoming signals.

Genode 〉 Pd_session 〉 *free_signal_source*

Free a signal source   pure virtual method

Argument

  cap   **Signal_source_capability**
        Capability of the signal source to destroy

Genode ❯ Pd_session ❯ *alloc_context*

Allocate signal context   pure virtual method

Arguments

source   **Signal_source_capability**
          Signal source that shall provide the new context

imprint   **unsigned long**
          Opaque value that gets delivered with signals originating from the allocated
          signal-context capability

Exceptions

Out_of_metadata
Invalid_signal_source

Return value

Capability<**Signal_context**>   New signal-context capability

---

Genode ❯ Pd_session ❯ *free_context*

Free signal-context   pure virtual method

Argument

cap   **Capability**<**Signal_context**>
      Capability of signal-context to release

GENODE LABS

Genode	Pd_session	*submit*

Submit signals to the specified signal context **pure virtual method**

Arguments

context **Capability<Signal_context>**
Signal destination

cnt **unsigned**
Number of signals to submit at once
Default is **1**

Details

The context argument does not necessarily belong to this PD session. Normally, it is a capability obtained from a potentially untrusted component. Because we cannot trust this capability, signals are not submitted by invoking cap directly but by using it as argument to our trusted PD-session interface. Otherwise, a potential signal receiver could supply a capability with a blocking interface to compromise the nonblocking behaviour of the signal submission.

Genode	Pd_session	*alloc_rpc_cap*

Allocate new RPC-object capability **pure virtual method**

Argument

ep **Native_capability**
Entry point that will use this capability

Exception

Out_of_metadata If meta-data backing store is exhausted

Return value

Native_capability New rpc capability

Genode	Pd_session	*free_rpc_cap*

Free RPC-object capability **pure virtual method**

Argument

cap **Native_capability**
Capability to free

GENODE LABS

Genode 〉 Pd_session 〉 **_address_space_**

pure virtual method

Return value

  Capability<Region_map>   Region map of the pd's virtual address space

Genode 〉 Pd_session 〉 **_stack_area_**

pure virtual method

Return value

  Capability<Region_map>   Region map of the pd's stack area

Genode 〉 Pd_session 〉 **_linker_area_**

pure virtual method

Return value

  Capability<Region_map>   Region map of the pd's linker area

Genode 〉 Pd_session 〉 **_native_pd_**

pure virtual method

Return value

  Capability<Native_pd>   Capability to kernel-specific pd operations

GENODE LABS

| Genode | **Pd_connection** | | class |

Connection to PD service

Connection <Pd_session>   Pd_session_client

Pd_connection

Pd_connection(...)

Header

repos/base/include/pd_session/connection.h

| Genode | Pd_connection | **Pd_connection** | constructor |

Arguments

env   **Env &**

label   **char const ***
       Session label
       Default is " "

GENODE LABS
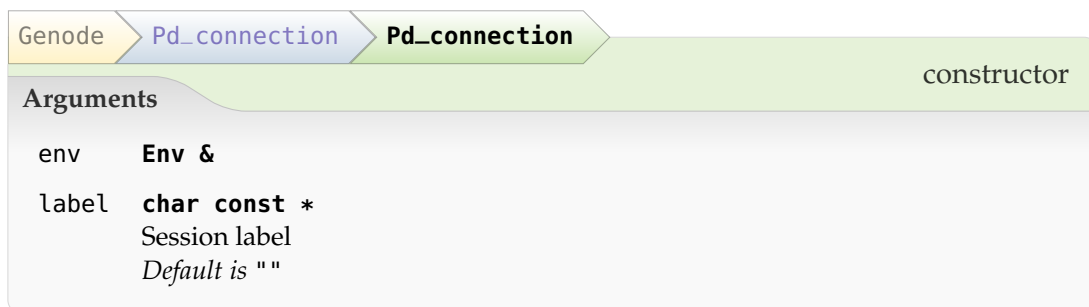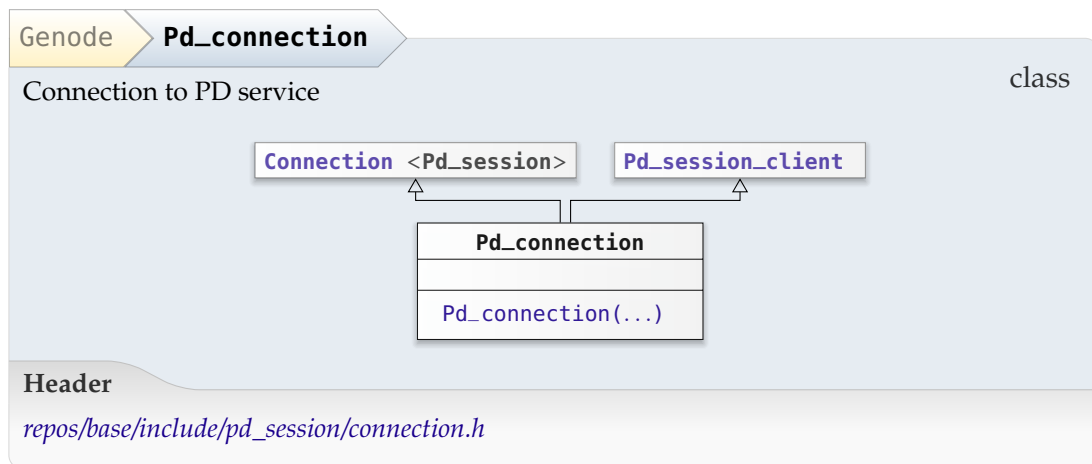
8.5.4. RM session interface

The region-map (RM) service (Section 3.4.6) allows for the manual creation of region maps that represent (portions of) virtual address spaces. Components can use this service to manage the virtual memory layout of so-called managed dataspaces manually. For example, it allows a dataspace provider to populate the content of dataspaces on demand, depending on the access patterns produced by the dataspace user.

Note that the RM service is not available on all base platforms. In particular, it is not supported on Linux.

Genode	**Rm_session**		class

Region-map session interface

```
                            ┌──────────┐
                            │ Session  │
                            └──────────┘
                                 △
         ┌───────────────────────────────────────┐
         │              Rm_session               │
         ├───────────────────────────────────────┤
         ├───────────────────────────────────────┤
         │  service_name() : const char *        │ ┄┄┄┄ RPC interface
         │  create(...) : Capability<Region_map>  │
         │  destroy(...)                          │
         └───────────────────────────────────────┘
```

Header

repos/base/include/rm_session/rm_session.h

Genode	Rm_session	**service_name**	class function

Return value

const char *

GENODE LABS
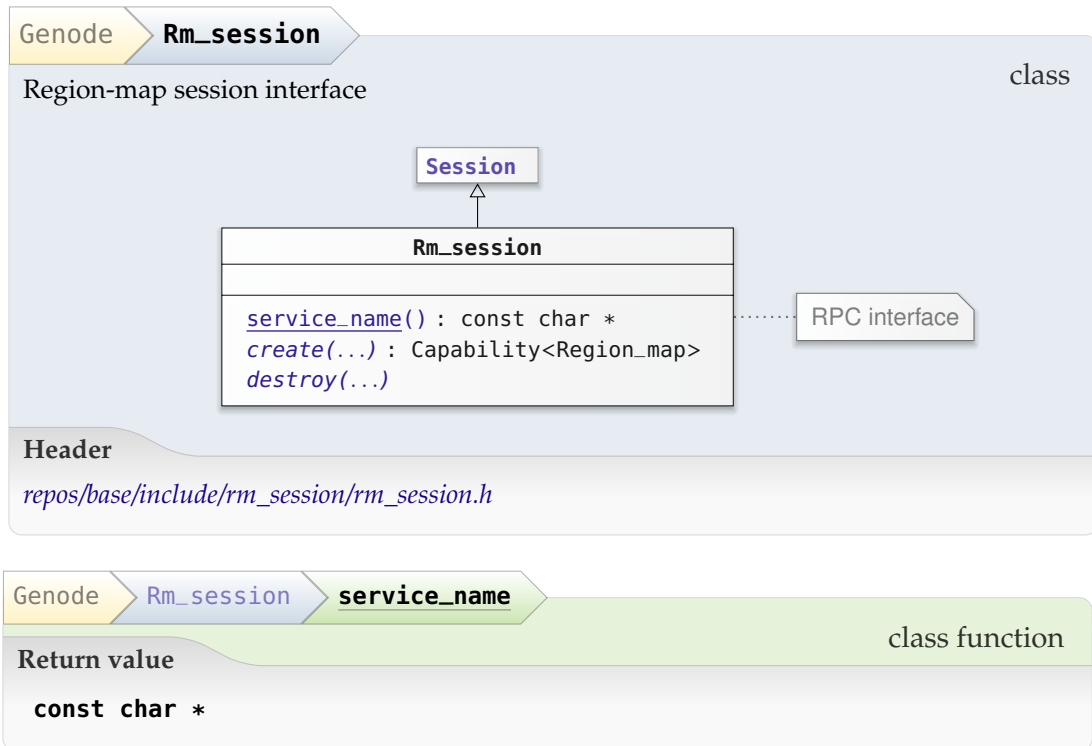
Genode 〉 Rm_session 〉 *create*

Create region map   pure virtual method

Argument

  size   **size_t**
         Upper bound of region map

Exception

  Out_of_metadata

Return value

  Capability<**Region_map**>   Region-map capability

Genode 〉 Rm_session 〉 *destroy*

Destroy region map   pure virtual method

Argument

  -   **Capability**<**Region_map**>

Genode 〉 **Rm_connection**

Connection to RM service   class

Connection <Rm_session>   **Rm_session_client**

                    Rm_connection

                    Rm_connection(...)

Header

repos/base/include/rm_session/connection.h

Genode 〉 Rm_connection 〉 **Rm_connection**

                                                    constructor

Argument

  env   **Env &**

GENODE ⚖ LABS

8.5.5. CPU session interface

The CPU service (Section 3.4.7) provides a facility for creating and managing threads. A CPU session corresponds to a CPU-time allocator, from which multiple threads can be allocated.

Genode **Cpu_session**

class

Session

```
Cpu_session

service_name() : const char *
~Cpu_session()
create_thread(...) : Thread_capability
kill_thread(...)
exception_sigh(...)
affinity_space() : Affinity::Space
scale_priority(...) : unsigned
trace_control() : Dataspace_capability
ref_account(...) : int
transfer_quota(...) : int
quota() : Quota
quota_lim_upscale(...) : size_t
quota_lim_downscale(...) : size_t
native_cpu() : Capability<Native_cpu>
```

RPC interface

Header

repos/base/include/cpu_session/cpu_session.h

Genode Cpu_session **service_name**

class function

Return value

  const char *

GENODE ⬙ LABS

Genode ⟩ Cpu_session ⟩ *create_thread*

Create a new thread | pure virtual method

Arguments

pd **Capability<Pd_session>**
Protection domain where the thread will be executed

name **Name const &**
Name for the thread

affinity **Location**
Cpu affinity, referring to the session-local affinity space

weight **Weight**
Cpu quota that shall be granted to the thread

utcb **addr_t**
Base of the utcb that will be used by the thread
Default is 0

Exceptions

Thread_creation_failed
Out_of_metadata
Quota_exceeded

Return value

Thread_capability Capability representing the new thread

Genode ⟩ Cpu_session ⟩ *kill_thread*

Kill an existing thread | pure virtual method

Argument

thread **Thread_capability**
Capability of the thread to kill

GENODE LABS

Genode > Cpu_session > *exception_sigh*

Register default signal handler for exceptions

pure virtual method

Argument

- **Signal_context_capability**

Details

This handler is used for all threads that have no explicitly installed exception handler. On Linux, this exception is delivered when the process triggers a SIGCHLD. On other platforms, this exception is delivered on the occurrence of CPU exceptions such as division by zero.

Genode > Cpu_session > *affinity_space*

pure virtual const method

Return value

Affinity::Space   Affinity space of cpu nodes available to the cpu session

Details

The dimension of the affinity space as returned by this method represent the physical CPUs that are available.

Genode > Cpu_session > **scale_priority**

Translate generic priority value to kernel-specific priority levels

class function

Arguments

pf_prio_limit   **unsigned**
                 Maximum priority used for the kernel, must be power of 2

prio   **unsigned**
                 Generic priority value as used by the cpu session interface

inverse   **bool**
                 Order of platform priorities, if true `pf_prio_limit` corresponds to the highest priority, otherwise it refers to the lowest priority.
                 Default is `true`

Return value

unsigned   Platform-specific priority value

GENODE LABS

Genode 〉 Cpu_session 〉 *trace_control*

Request trace control dataspace

pure virtual method

Return value

Dataspace_capability

Details

The trace-control dataspace is used to propagate tracing control information from core to the threads of a CPU session.
The trace-control dataspace is accounted to the CPU session.

Genode 〉 Cpu_session 〉 *ref_account*

Define reference account for the CPU session

pure virtual method

Argument

cpu_session   **Cpu_session_capability**
Reference account

Return value

int   0 on success

Details

Each CPU session requires another CPU session as reference account to transfer quota to and from. The reference account can be defined only once.

Genode 〉 Cpu_session 〉 *transfer_quota*

Transfer quota to another CPU session

pure virtual method

Arguments

cpu_session   **Cpu_session_capability**
Receiver of quota donation

amount   **size_t**
Percentage of the session quota scaled up to the quota_limit space

Return value

int   0 on success

Details

Quota can only be transfered if the specified CPU session is either the reference account for this session or vice versa.

GENODE LABS

Genode   Cpu_session   *quota*

pure virtual method

Return value

Quota   Quota configuration of the session

Genode   Cpu_session   **quota_lim_upscale**

Scale up value from its space with limit to the QUOTA_LIMIT space class function template

Template argument

T   **typename**
   Default is size_t

Arguments

value   **size_t const**

limit   **size_t const**

Return value

size_t

Genode   Cpu_session   **quota_lim_downscale**

Scale down value from the QUOTA_LIMIT space to a space with limit class function template

Template argument

T   **typename**
   Default is size_t

Arguments

value   **size_t const**

limit   **size_t const**

Return value

size_t

GENODE LABS

Genode	Cpu_session	*native_cpu*	

pure virtual method

Return value

Capability<Native_cpu>   Capability to kernel-specific cpu operations

Genode	**Cpu_connection**

class

Connection to CPU service

Connection <**Cpu_session**>   **Cpu_session_client**

Cpu_connection

Cpu_connection(...)

Header

repos/base/include/cpu_session/connection.h

Genode	Cpu_connection	**Cpu_connection**

constructor

Arguments

env   **Env &**

label   **const char ***
           Initial session label
           Default is ""

priority   **long**
           Designated priority of all threads created with this cpu session
           Default is DEFAULT_PRIORITY

affinity   **Affinity const &**
           Default is Affinity()

GENODE LABS

Genode ⟩ **Affinity**

Affinity to CPU nodes

class

```
                          Affinity

              Affinity(...)
              Affinity()
              space() : Space
              location() : Location
              scale_to(...) : Location
```

Accessors

| space | **Space** |
| location | **Location** |

Details

The entity of CPU nodes is expected to form a grid where the Euclidean distance between nodes roughly correlate to the locality of their respective resources. Closely interacting processes are supposed to perform best when using nodes close to each other. To allow a relatively simple specification of such constraints, the affinity of a subsystem (e. g., a process) to CPU nodes is expressed as a rectangle within the grid of available CPU nodes. The dimensions of the grid are represented by `Affinity::Space`. The rectangle within the grid is represented by `Affinity::Location`.

Header

repos/base/include/base/affinity.h

Genode ⟩ Affinity ⟩ **Affinity**

constructor

Arguments

| space | **Space const &** |
| location | **Location const &** |

GENODE ⟋⟍ LABS

Genode Affinity **scale_to**

const method

Argument

space **Space const &**

Return value

Location Location scaled to specified affinity space
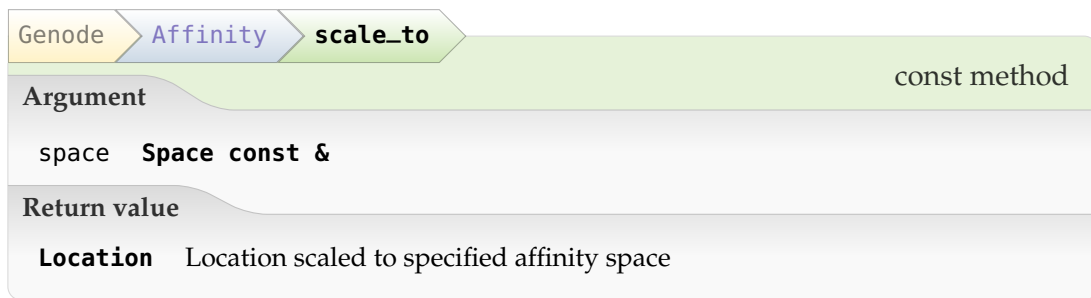
Once created, a thread is referred to via a thread capability. This capability allows for the destruction of the thread via the CPU session, and provides the `Cpu_thread` RPC interface to operate on the thread.

Genode `Cpu_thread`

CPU thread interface

class

Cpu_thread
utcb() : Dataspace_capability *start(...)* *pause()* *resume()* *cancel_blocking()* *state()* : Thread_state *state(...)* *exception_sigh(...)* *single_step(...)* *affinity(...)* *trace_control_index()* : unsigned *trace_buffer()* : Dataspace_capability *trace_policy()* : Dataspace_capability

RPC interface

Header

repos/base/include/cpu_thread/cpu_thread.h

Genode Cpu_thread *utcb*

Get dataspace of the thread's user-level thread-control block (UTCB)

pure virtual method

Return value

Dataspace_capability

Genode Cpu_thread *start*

Modify instruction and stack pointer of thread - start the thread

pure virtual method

Arguments

ip **addr_t**
    Initial instruction pointer

sp **addr_t**
    Initial stack pointer

265

GENODE LABS

Genode ⟩ Cpu_thread ⟩ *pause*

Pause the thread   pure virtual method

Details

After calling this method, the execution of the thread can be continued by calling `resume`.

Genode ⟩ Cpu_thread ⟩ *resume*

Resume the thread   pure virtual method

Genode ⟩ Cpu_thread ⟩ *cancel_blocking*

Cancel a currently blocking operation   pure virtual method

Genode ⟩ Cpu_thread ⟩ *state*

Get the current thread state   pure virtual method

Exception

State_access_failed

Return value

Thread_state   State of the targeted thread

Genode ⟩ Cpu_thread ⟩ *state*

Override the current thread state   pure virtual method

Argument

state   **Thread_state const &**
         State that shall be applied

Exception

State_access_failed

GENODE LABS

Genode	Cpu_thread	**exception_sigh**

Register signal handler for exceptions of the thread

pure virtual method

Argument

handler **Signal_context_capability**

Details

On Linux, this exception is delivered when the process triggers a SIGCHLD. On other platforms, this exception is delivered on the occurrence of CPU exceptions such as division by zero.

Genode	Cpu_thread	**single_step**

Enable/disable single stepping

pure virtual method

Argument

enabled **bool**
         True = enable single-step mode; false = disable

Details

Since this method is currently supported by a small number of platforms, we provide a default implementation

Genode	Cpu_thread	**affinity**

Define affinity of thread to one or multiple CPU nodes

pure virtual method

Argument

location **Location**
          Location within the affinity space of the thread's cpu session

Details

In the normal case, a thread is assigned to a single CPU. Specifying more than one CPU node is supposed to principally allow a CPU service to balance the load of threads among multiple CPUs.

GENODE 👁 LABS

Genode › Cpu_thread › **trace_control_index**

Request index within the trace control block of the thread's CPU session

pure virtual method

Return value

unsigned

Details

The trace control dataspace contains the control blocks for all threads of the CPU session. Each thread gets assigned a different index by the CPU service.

Genode › Cpu_thread › **trace_buffer**

Request trace buffer for the thread

pure virtual method

Return value

Dataspace_capability

Details

The trace buffer is not accounted to the CPU session. It is owned by a TRACE session.

Genode › Cpu_thread › **trace_policy**

Request trace policy

pure virtual method

Return value

Dataspace_capability

Details

The trace policy buffer is not accounted to the CPU session. It is owned by a TRACE session.

GENODE LABS

8.5.6. IO_MEM session interface

The IO_MEM service (Section 3.4.8) enables user-level device drivers to obtain memory-mapped device resources in the form of dataspaces. Each IO_MEM session corresponds to the reservation of a physical address range, for which a dataspace is provided to the client. The user-level device driver can make the device resource visible in its address space by attaching the dataspace to the region map of its own PD session.

Genode   namespace

I/O-memory session interface

Types

    Io_mem_dataspace   is a subtype of `Dataspace`

    Io_mem_dataspace_capability   is defined as `Capability<Io_mem_dataspace>`

Header

repos/base/include/io_mem_session/io_mem_session.h

Genode   **Io_mem_session**   class

                              Session

Io_mem_session
service_name() : const char *
~Io_mem_session()
dataspace() : Io_mem_dataspace_capability

                                                RPC interface

Header

repos/base/include/io_mem_session/io_mem_session.h

Genode   Io_mem_session   **service_name**

                                       class function

Return value

  const char ∗

GENODE LABS

Genode	Io_mem_session	*dataspace*	
Request dataspace containing the IO_MEM session data			pure virtual method

Return value

   Io_mem_dataspace_capability   Capability to io_mem dataspace (may be invalid)

Genode	**Io_mem_connection**	
Connection to I/O-memory service		class

Connection <Io_mem_session>   **Io_mem_session_client**

> **Io_mem_connection**
>
> Io_mem_connection(...)

Header

repos/base/include/io_mem_session/connection.h

Genode	Io_mem_connection	**Io_mem_connection**	
			constructor

Arguments

env	**Env &**
base	**addr_t** Physical base address of memory-mapped i/o resource
size	**size_t** Size memory-mapped i/o resource
write_combined	**bool** Enable write-combined access to i/o memory *Default is* `false`

Attached IO_MEM dataspace   An instance of an `Attached_io_mem_dataspace` represents a locally mapped memory-mapped I/O range.

GENODE LABS

Genode ⟩ **Attached_io_mem_dataspace**   class

Request and locally attach a memory-mapped I/O resource

Attached_io_mem_dataspace
Attached_io_mem_dataspace(...)
~Attached_io_mem_dataspace()
cap() : Io_mem_dataspace_capability
local_addr() : T *

Details

This class is a wrapper for a typical sequence of operations performed by device drivers to access memory-mapped device resources. Its sole purpose is to avoid duplicated code.

Header

repos/base/include/base/attached_io_mem_dataspace.h

Genode ⟩ Attached_io_mem_dataspace ⟩ **Attached_io_mem_dataspace**

constructor

Arguments

env	**Env &**
base	**addr_t**
	Base address of memory-mapped i/o resource
size	**size_t**
	Size of resource
write_combined	**bool**
	Enable write combining for the resource
	Default is `false`

Exceptions

Parent::Service_denied
Parent::Quota_exceeded
Parent::Unavailable
Rm_session::Attach_failed

GENODE LABS

Genode Attached_io_mem_dataspace **cap**

method

Return value

Io_mem_dataspace_capability   Capability of the used ram dataspace

Genode Attached_io_mem_dataspace **local_addr**

method template

Request local address

Template argument

T   **typename**

Return value

T *

Details

This is a template to avoid inconvenient casts at the caller. A newly allocated I/O MEM
dataspace is untyped memory anyway.

GENODE ⬙ LABS

8.5.7. IO_PORT session interface

On the x86 architecture, the IO_PORT service (Section 3.4.8) provides access to device I/O ports via an RPC interface. Each IO_PORT session corresponds to the access right to a port range.

Genode ⟩ **Io_port_session**

I/O-port session interface
class

Session

Io_port_session
service_name() : const char * ~Io_port_session() *inb(...)* : unsigned char *inw(...)* : unsigned short *inl(...)* : unsigned *outb(...)* *outw(...)* *outl(...)*

RPC interface

Header

repos/base/include/io_port_session/io_port_session.h

Genode ⟩ Io_port_session ⟩ **service_name**

Return value
class function

const char *

Genode ⟩ Io_port_session ⟩ *inb*

Read byte (8 bit)
pure virtual method

Argument

address **unsigned short**
Physical i/o port address

Return value

unsigned char   Value read from port

GENODE ⚞ LABS

Genode ⟩ Io_port_session ⟩ *inw*

Read word (16 bit)   pure virtual method

Argument

  address   **unsigned short**
            Physical i/o port address

Return value

  unsigned short   Value read from port

Genode ⟩ Io_port_session ⟩ *inl*

Read double word (32 bit)   pure virtual method

Argument

  address   **unsigned short**
            Physical i/o port address

Return value

  unsigned   Value read from port

Genode ⟩ Io_port_session ⟩ *outb*

Write byte (8 bit)   pure virtual method

Arguments

  address   **unsigned short**
            Physical i/o port address

  value   **unsigned char**
            Value to write to port

Genode ⟩ Io_port_session ⟩ *outw*

Write word (16 bit)   pure virtual method

Arguments

  address   **unsigned short**
            Physical i/o port address

  value   **unsigned short**
            Value to write to port

GENODE LABS

Genode ⟩ Io_port_session ⟩ **outl**

Write double word (32 bit)   pure virtual method

Arguments

address   **unsigned short**
           Physical i/o port address

value   **unsigned**
           Value to write to port

Genode ⟩ **Io_port_connection**

Connection to I/O-port service   class

Connection <Io_port_session>   **Io_port_session_client**

              Io_port_connection

              Io_port_connection(...)

Header

repos/base/include/io_port_session/connection.h

Genode ⟩ Io_port_connection ⟩ **Io_port_connection**

                                                            constructor
Arguments

env   **Env &**

base   **unsigned**
        Base address of port range

size   **unsigned**
        Size of port range

GENODE LABS

8.5.8. IRQ session interface

The IRQ service (Section 3.4.8) enables user-level device drivers to serve device interrupts. Each IRQ session corresponds to an associated interrupt line.

Genode · **Irq_connection**

Connection to IRQ service   class

Connection <Irq_session>   Irq_session_client

Irq_connection

Irq_connection(...)

Header

repos/base/include/irq_session/connection.h

Genode · Irq_connection · **Irq_connection**

constructor

Arguments

env	**Env &**
irq	**unsigned** Physical interrupt number
-	**TRIGGER_UNCHANGED**
-	**POLARITY_UNCHANGED**
device_config_phys	**addr_t** *Default is 0*

GENODE ⬟ LABS

8.5.9. LOG session interface

For low-level debugging, core provides a simple LOG service (Section 3.4.9), which enables clients to print textual messages. In the LOG output, each message is tagged with the label of the corresponding client.

Genode **Log_session**

Log text output session interface   class

Session

Log_session
service_name() : const char * ⌐··········⌐ RPC interface
~Log_session()
write(...) : size_t

Header

repos/base/include/log_session/log_session.h

Genode Log_session **service_name**

Return value   class function

const char ∗

Genode Log_session *write*

Output null-terminated string   pure virtual method

Argument

string **String const &**

Return value

size_t   Number of written characters

277

GENODE ⧸⧸ LABS

Genode   **Log_connection**

Connection to LOG service

class

Connection <Log_session>   Log_session_client

Log_connection

Log_connection(...)

Header

repos/base/include/log_session/connection.h

Genode   Log_connection   **Log_connection**

constructor

Argument

env   **Env &**

GENODE LABS

8.6. OS-level session interfaces

8.6.1. Report session interface

Report sessions (Section 4.5.2) allow a component to propagate internal state to the outside. The most prominent use case is the realization of publisher-subscriber communication schemes as discussed in Section 4.7.5.

```
Report       Session

Report session interface                                        class


                        Session

              Report::Session

              service_name() : const char *
              dataspace() : Dataspace_capability    ......  RPC interface
              submit(...)
              response_sigh(...)
              obtain_response() : size_t

Header

repos/os/include/report_session/report_session.h
```

```
Report     Session     service_name

Return value                                           class function

  const char *
```

```
Report     Session     dataspace

Request the dataspace used to carry reports and responses   pure virtual method

Return value

  Dataspace_capability
```

GENODE LABS

Report	Session	*submit*	

Submit data that is currently contained in the dataspace as report   pure virtual method

Argument

length   **size_t**
         Length of report in bytes

Details

While this method is called, the information in the dataspace must not be modified by the client.

Report	Session	*response_sigh*	

Install signal handler for response notifications   pure virtual method

Argument

-    **Signal_context_capability**

Report	Session	*obtain_response*	

Request a response from the recipient of reports   pure virtual method

Return value

size_t   Length of response in bytes

Details

By calling this method, the client expects that the server will replace the content of the dataspace with new information.

GENODE LABS

Report	**Connection**		

Connection to Report service

class

Connection <Session>	Session_client

Report::Connection
Connection(...)

Header

repos/os/include/report_session/connection.h

Report	Connection	**Connection**

constructor

Arguments

env	**Env &**
label	**char const ***
buffer_size	**size_t**
	Default is 4096

GENODE LABS
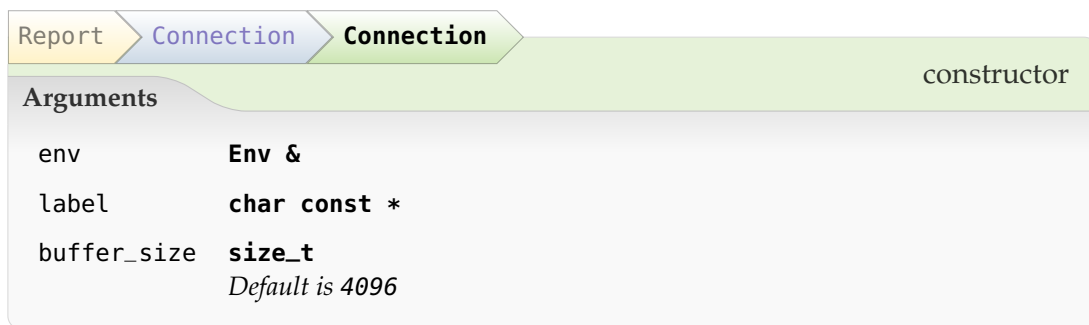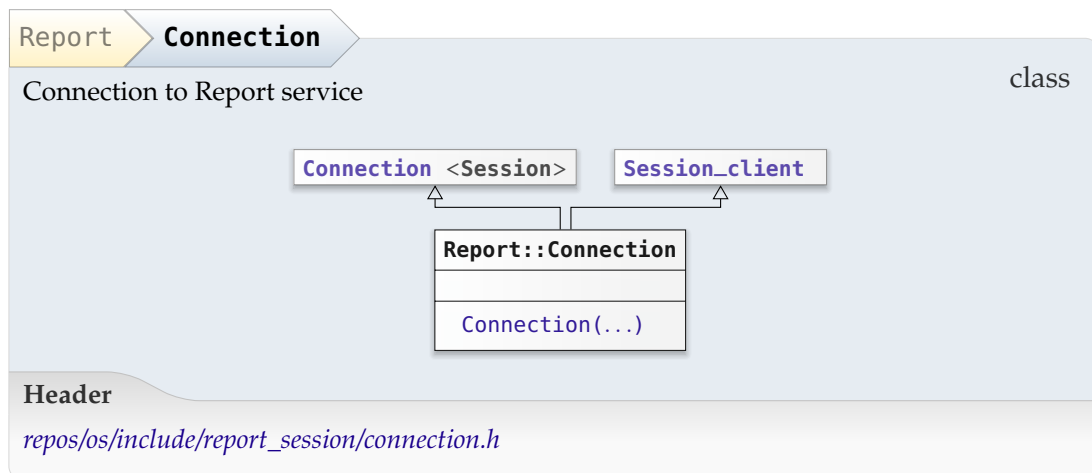
The client-side Reporter is a convenient front end for the use of a report connection to propagate XML-formed data.

Genode > **Reporter**		class

Utility for status reporting

```
                    Reporter
        ────────────────────────────
        Reporter(...)
        enabled(...)
        enabled() : bool
        name() : Name
        clear()
        report(...)
```

Accessor

  name   **Name**

Header

repos/os/include/os/reporter.h

Genode > Reporter > **Reporter**	constructor

Arguments

xml_name	**char const ***
label	**char const ***
	Default is nullptr
buffer_size	**size_t**
	Default is 4096

Genode > Reporter > **enabled**	method

Enable or disable reporting

Argument

  enabled **bool**

282

GENODE LABS

Genode > Reporter > **enabled**

const method

Return value

bool   True if reporter is enabled

Genode > Reporter > **clear**

method

Clear report buffer

Genode > Reporter > **report**

method

Report data buffer

Arguments

data   **void const ***
          Data buffer

length   **size_t**
          Number of bytes to report

283

GENODE LABS

8.6.2. Terminal and UART session interfaces

A terminal session (Section 4.5.3) is a bi-directional communication channel. The
UART session interface supplements the terminal session interface with a facility to
parametrize UART configuration parameters

```
Terminal  Session                                                        class
          Terminal session interface

                                 Session

                           Terminal::Session

                        service_name() : const char *
                        size() : Size
                        avail() : bool                    RPC interface
                        read(...) : Genode::size_t
                        write(...) : Genode::size_t
                        connected_sigh(...)
                        read_avail_sigh(...)

          Header

          repos/os/include/terminal_session/terminal_session.h
```

```
Terminal  Session  service_name
                                                              class function
          Return value

            const char *
```

```
Terminal  Session  size
                                                           pure virtual method
          Return value

            Size    Terminal size
```

```
Terminal  Session  avail
                                                           pure virtual method
          Return value

            bool    True of one or more characters are available for reading
```

GENODE LABS

Terminal   Session   *read*

pure virtual method

Read characters from terminal

Arguments

buf   **void ∗**

buf_size   **size_t**

Return value

Genode::size_t

Terminal   Session   *write*

pure virtual method

Write characters to terminal

Arguments

buf   **void const ∗**

num_bytes   **size_t**

Return value

Genode::size_t

Terminal   Session   *connected_sigh*

pure virtual method

Register signal handler to be informed about the established connection

Argument

cap   **Signal_context_capability**

Details

This method is used for a simple startup protocol of terminal sessions. At session-creation time, the terminal session may not be ready to use. For example, a TCP terminal session needs an established TCP connection first. However, we do not want to let the session-creation block on the server side because this would render the servers entrypoint unavailable for all other clients until the TCP connection is ready. Instead, we deliver a connected signal to the client emitted when the session becomes ready to use. The Terminal::Connection waits for this signal at construction time.

GENODE LABS

Terminal	Session	***read_avail_sigh***

Register signal handler to be informed about ready-to-read characters

pure virtual method

Argument

cap   **Signal_context_capability**

Terminal	**Connection**

Connection to Terminal service

class

Connection <Session>   **Session_client**

Terminal::Connection

Connection(...)

Header

repos/os/include/terminal_session/connection.h

Terminal	Connection	**Connection**

constructor

Arguments

env   **Env &**

label   **char const ***
        Default is ""

GENODE LABS

Uart ⟩ **Session**

UART session interface   class

Terminal::Session

Uart::Session
-----
service_name() : const char *   ⋯⋯ RPC interface
baud_rate(...)

Header

repos/os/include/uart_session/uart_session.h

Uart ⟩ Session ⟩ **service_name**

Return value   class function

const char *

Uart ⟩ Session ⟩ ***baud_rate***

Set baud rate   pure virtual method

Argument

bits_per_second   **size_t**

GENODE ⚞ LABS

Uart · **Connection**

Connection to UART service

class

Connection <Session>   Session_client

Uart::Connection

Connection(...)

Header

repos/os/include/uart_session/connection.h

Uart · Connection · **Connection**

constructor

Argument

env   **Env &**

GENODE LABS

8.6.3. Input session interface

An input session (Section 4.5.4) represents a stream of user-input events.

```
┌─────────────────────────────────────────────────────────────────────┐
│ ┌Input ╲ Session ╲                                              class │
│ Input session interface                                               │
│                                                                       │
│                          ┌─Session─┐                                  │
│                          └────△────┘                                  │
│                               │                                       │
│              ┌────────────────────────────────────┐                  │
│              │          Input::Session             │                  │
│              ├────────────────────────────────────┤                  │
│              ├────────────────────────────────────┤                  │
│              │ service_name() : const char *       │   ┌ RPC interface│
│              │ ~Session()                          │···│              │
│              │ dataspace() : Genode::Dataspace_capability              │
│              │ pending() : bool                    │                  │
│              │ flush() : int                       │                  │
│              │ sigh(...)                           │                  │
│              └────────────────────────────────────┘                  │
│                                                                       │
│ ┌Header                                                               │
│ repos/os/include/input_session/input_session.h                        │
└─────────────────────────────────────────────────────────────────────┘
```

```
┌─────────────────────────────────────────────────────────────────────┐
│ Input ╲ Session ╲ service_name ╲                                      │
│                                                       class function  │
│ ┌Return value                                                         │
│   const char *                                                        │
└─────────────────────────────────────────────────────────────────────┘
```

```
┌─────────────────────────────────────────────────────────────────────┐
│ Input ╲ Session ╲ dataspace ╲                                         │
│                                                  pure virtual method  │
│ ┌Return value                                                         │
│   Genode::Dataspace_capability   Capability to event buffer dataspace │
└─────────────────────────────────────────────────────────────────────┘
```

```
┌─────────────────────────────────────────────────────────────────────┐
│ Input ╲ Session ╲ pending ╲                                           │
│ Request input state                          pure virtual const method│
│ ┌Return value                                                         │
│   bool   True if new events are available                             │
└─────────────────────────────────────────────────────────────────────┘
```

GENODE LABS

Input Session *flush*

Flush pending events to event buffer   pure virtual method

Return value

  int   Number of flushed events

Input Session *sigh*

Register signal handler to be notified on arrival of new input   pure virtual method

Argument

  -   **Signal_context_capability**

Input **Connection**

Connection to input service   class

```
Connection <Session>        Session_client

        Input::Connection

        Connection(...)
```

Header

repos/os/include/input_session/connection.h

Input Connection **Connection**

                                                                              constructor

Argument

  env   **Env &**

GENODE LABS

8.6.4. Framebuffer session interface

Framebuffer ❭ **Mode**

Framebuffer mode info as returned by `Framebuffer::Session::mode()`

class

```
Framebuffer::Mode
```

Header

repos/os/include/framebuffer_session/framebuffer_session.h

Framebuffer ❭ **Session**

class

Session

```
             Framebuffer::Session

  service_name() : const char *
  ~Session()
  dataspace() : Genode::Dataspace_capability   ......... RPC interface
  mode() : Mode
  mode_sigh(...)
  refresh(...)
  sync_sigh(...)
```

Header

repos/os/include/framebuffer_session/framebuffer_session.h

Framebuffer ❭ Session ❭ **service_name**

class function

Return value

const char *

GENODE LABS

Framebuffer	Session	*dataspace*

Request dataspace representing the logical frame buffer
pure virtual method

Return value

   Genode::Dataspace_capability

Details

By calling this method, the framebuffer client enables the server to reallocate the frame-buffer dataspace (e. g., on mode changes). Hence, prior calling this method, the client should make sure to have detached the previously requested dataspace from its local address space.

Framebuffer	Session	*mode*

Request display-mode properties of the framebuffer ready to be
pure virtual const method
obtained via the `dataspace()` method

Return value

   Mode

Framebuffer	Session	*mode_sigh*

Register signal handler to be notified on mode changes
pure virtual method

Argument

  sigh   **Signal_context_capability**

Details

The framebuffer server may support changing the display mode on the fly. For example, a virtual framebuffer presented in a window may get resized according to the window dimensions. By installing a signal handler for mode changes, the framebuffer client can respond to such changes. The new mode can be obtained using the `mode()` method. However, from the client's perspective, the original mode stays in effect until the it calls `dataspace()` again.

GENODE ⋙ LABS

Framebuffer Session *refresh*

Flush specified pixel region

pure virtual method

Arguments

x **int**

y **int**

w **int**

h **int**

Framebuffer Session *sync_sigh*

Register signal handler for refresh synchronization

pure virtual method

Argument

- **Signal_context_capability**

GENODE LABS

Framebuffer **Connection**

Connection to frame-buffer service

class

Connection <**Session**>   **Session_client**

Framebuffer::Connection

Connection(...)

Header

repos/os/include/framebuffer_session/connection.h
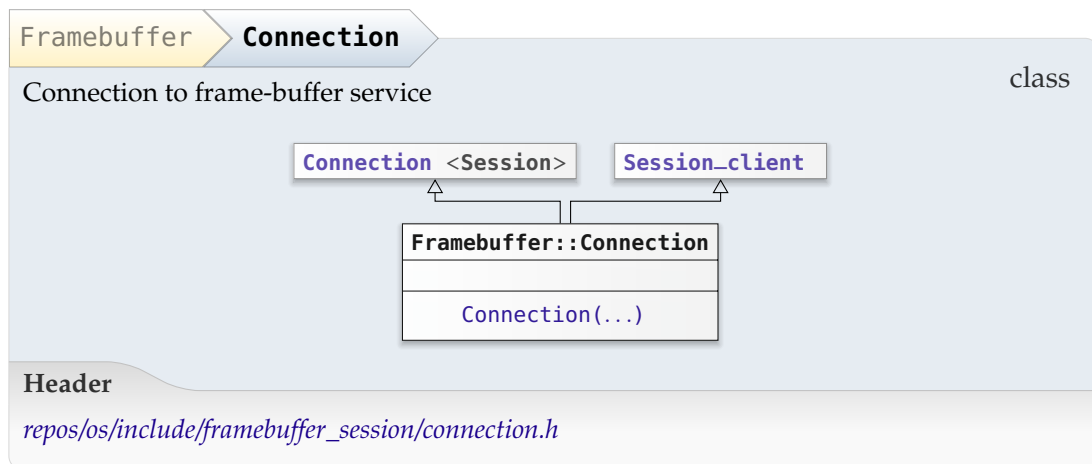
Framebuffer Connection **Connection**

constructor

Arguments

env   **Env &**

mode   **Mode**
       Desired size and pixel format

Details

The specified values are not enforced. After creating the session, you should validate the actual frame-buffer attributes by calling the info method of the frame-buffer interface.

GENODE LABS

8.6.5. Nitpicker session interface

Nitpicker	**Session**		class

Nitpicker session interface

Session

Nitpicker::Session

service_name() : const char *
~Session()
framebuffer_session() : Framebuffer::Session_capability
input_session() : Input::Session_capability
create_view(...) : View_handle
destroy_view(...)
view_handle(...) : View_handle
view_capability(...) : View_capability
release_view_handle(...)
command_dataspace() : Genode::Dataspace_capability
execute()
mode() : Framebuffer::Mode
mode_sigh(...)
buffer(...)
focus(...)
session_control(...)
ram_quota(...) : size_t

RPC interface

Header

repos/os/include/nitpicker_session/nitpicker_session.h

Nitpicker	Session	**service_name**	class function

Return value

  const char *

Nitpicker	Session	*framebuffer_session*	pure virtual method

Request framebuffer sub-session

Return value

  Framebuffer::Session_capability

GENODE 〽 LABS

Nitpicker > Session > *input_session*

Request input sub-session

pure virtual method

Return value

Input::Session_capability

Nitpicker > Session > *create_view*

Create a new view at the buffer

pure virtual method

Argument

parent   **View_handle**
          Parent view
          Default is View_handle()

Exception

Invalid_handle

Return value

View_handle   Handle for new view

Details

The parent argument allows the client to use the location of an existing view as the coordinate origin for the to-be-created view. If an invalid handle is specified (default), the view will be a top-level view.

Nitpicker > Session > *destroy_view*

Destroy view

pure virtual method

Argument

-    **View_handle**

GENODE LABS

Nitpicker	Session	*view_handle*

pure virtual method

Arguments

-	**View_capability**

handle	**View_handle**

Designated view handle to be assigned to the imported view. by default, a new handle will be allocated.
Default is `View_handle()`

Exception

Out_of_metadata

Return value

View_handle   Session-local handle for the specified view

Details

The handle returned by this method can be used to issue commands via the `execute` method.

Nitpicker	Session	*view_capability*

Request view capability for a given handle

pure virtual method

Argument

-	**View_handle**

Return value

View_capability

Details

The returned view capability can be used to share the view with another session.

Nitpicker	Session	*release_view_handle*

Release session-local view handle

pure virtual method

Argument

-	**View_handle**

297

GENODE LABS

Nitpicker › Session › *command_dataspace*

Request dataspace used for issuing view commands to nitpicker   pure virtual method

Return value

  Genode::Dataspace_capability

Nitpicker › Session › *execute*

Execution batch of commands contained in the command dataspace   pure virtual method

---

Nitpicker › Session › *mode*

                                         pure virtual method

Return value

  Framebuffer::Mode   Physical screen mode

Nitpicker › Session › *mode_sigh*

Register signal handler to be notified about mode changes   pure virtual method

Argument

  -   **Signal_context_capability**

Nitpicker › Session › *buffer*

Define dimensions of virtual framebuffer   pure virtual method

Arguments

  mode   **Mode**

  use_alpha   **bool**

Exception

  Out_of_metadata   Session quota does not suffice for specified buffer dimensions

GENODE ⌇ LABS

Nitpicker	Session	*focus*

Set focused session — pure virtual method

Argument

focused	**Capability**<**Session**>

Details

Normally, the focused session is defined by the user by clicking on a view. The `focus` method allows a client to set the focus without user action. However, the change of the focus is performed only is the currently focused session belongs to a child or the same process as the called session. This relationship is checked by comparing the session labels of the currently focused session and the caller. This way, a common parent can manage the focus among its child processes. But a session cannot steal the focus from an unrelated session.

Nitpicker	Session	*session_control*

Perform control operation on one or multiple sessions — virtual method

Arguments

label	**Label**
operation	**Session_control**

Details

The `label` is used to select the sessions, on which the `operation` is performed. Nitpicker creates a selector string by concatenating the caller's session label with the supplied `label` argument. A session is selected if its label starts with the selector string. Thereby, the operation is limited to the caller session or any child session of the caller.

Nitpicker	Session	**ram_quota**

class function

Arguments

mode	**Mode**
use_alpha	**bool**

Return value

size_t	Number of bytes needed for virtual framebuffer of specified size

GENODE LABS

Nitpicker   **Connection**

Connection to Nitpicker service   class

Connection <Session>   **Session_client**

Nitpicker::Connection

Header

repos/os/include/nitpicker_session/connection.h

GENODE LABS
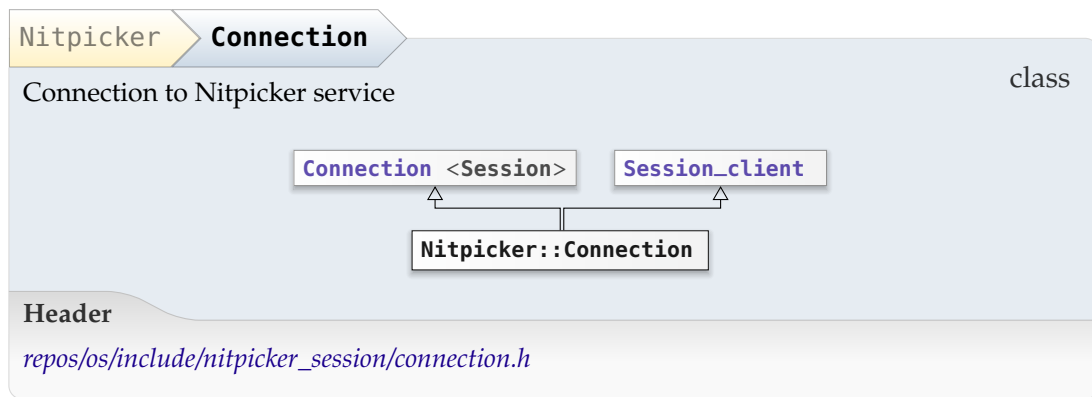
8.6.6. Platform session interface

Platform ⟩ **Device**

PCI-device interface   class

```
                    Platform::Abstract_device
                    ┌──────────────────────────────────┐
                    │        Platform::Device          │
                    ├──────────────────────────────────┤
                    ├──────────────────────────────────┤
                    │ ~Device()                        │
                    │ bus_address(...)                 │
                    │ vendor_id() : unsigned short     │
                    │ device_id() : unsigned short     │        RPC interface
                    │ class_code() : unsigned          │
                    │ resource(...) : Resource         │
                    │ config_read(...) : unsigned      │
                    │ config_write(...)                │
                    │ io_port(...) : Genode::Io_port_session_capability │
                    │ base_class() : unsigned          │
                    │ sub_class() : unsigned           │
                    │ phys_bar_to_virt(...) : Genode::uint8_t │
                    └──────────────────────────────────┘
```

Header

repos/os/include/spec/x86/platform_device/platform_device.h

Platform ⟩ Device ⟩ **bus_address**

Arguments   pure virtual method

  bus   **unsigned char** *

  dev   **unsigned char** *

  fn   **unsigned char** *

Platform ⟩ Device ⟩ **vendor_id**

                                                           pure virtual method
Return value

  unsigned short   Vendor id obtained from the pci config space

GENODE ⚙ LABS

Platform ⟩ Device ⟩ *device_id*

pure virtual method

Return value

unsigned short   Device id obtained from the pci config space

Platform ⟩ Device ⟩ *class_code*

pure virtual method

Return value

unsigned   Device class code from the pci config space

Platform ⟩ Device ⟩ *resource*

Query PCI-resource information

pure virtual method

Argument

resource_id   **int**
           Index of according pci resource of the device

Return value

Resource   Resource description

Platform ⟩ Device ⟩ *config_read*

Read configuration space

pure virtual method

Arguments

address   **unsigned char**

size   **Access_size**

Return value

unsigned

GENODE ⋙ LABS

Platform ⟩ Device ⟩ **config_write**

Write configuration space
pure virtual method

Arguments

address **unsigned char**

value **unsigned**

size **Access_size**

Platform ⟩ Device ⟩ **io_port**

Query Io_port of specified bar
pure virtual method

Argument

id **uint8_t**
Index of according pci resource of the device

Return value

Genode::Io_port_session_capability

Platform ⟩ Device ⟩ **base_class**

method

Return value

unsigned

Platform ⟩ Device ⟩ **sub_class**

method

Return value

unsigned

303

GENODE LABS

Platform ⟩ Device ⟩ **phys_bar_to_virt**

method

Convenience method to translate a PCI physical BAR id to a Genode virtual one usable with the io_port and io_mem methods. The virtual id is solely valid for the specific BAR type.

Argument

phys_bar **uint8_t**

Return value

Genode::uint8_t

Platform ⟩ **Session**

class

Platform session interface

Session

△
|
Platform::Session

service_name() : const char *
~Session()
first_device(...) : Device_capability
next_device(...) : Device_capability
release_device(...)
device(...) : Device_capability
alloc_dma_buffer(...) : Genode::Ram_dataspace_capability
free_dma_buffer(...)

RPC interface

Header

repos/os/include/spec/x86/platform_session/platform_session.h

---

Platform ⟩ Session ⟩ **service_name**

class function

Return value

const char *

GENODE ⋙ LABS

Platform	Session	*first_device*	

Find first accessible device   pure virtual method

Arguments

- **unsigned**

- **unsigned**

Return value

Device_capability

Platform	Session	*next_device*	

Find next accessible device   pure virtual method

Arguments

prev_device   **Device_capability**
               Previous device

-              **unsigned**

-              **unsigned**

Return value

Device_capability

Details

The prev_device argument is used to iterate through all devices.

Platform	Session	*release_device*	

Free server-internal data structures representing the device   pure virtual method

Argument

device   **Device_capability**

Details

Use this method to relax the heap partition of your PCI session.

GENODE LABS

Platform ⟩ Session ⟩ ***device***

Provide non-PCI device known by unique name.

pure virtual method

Argument

string   **String const &**

Return value

Device_capability

Platform ⟩ Session ⟩ ***alloc_dma_buffer***

Allocate memory suitable for DMA.

pure virtual method

Argument

-    **size_t**

Return value

Genode::Ram_dataspace_capability

Platform ⟩ Session ⟩ ***free_dma_buffer***

Free previously allocated DMA memory

pure virtual method

Argument

-    **Ram_dataspace_capability**

GENODE LABS

8.6.7. Block session interface

Block ⟩ **Packet_descriptor**

Represents an block-operation request   class

```
                              Packet_descriptor

                      ┌─────────────────────────────┐
                      │    Block::Packet_descriptor  │
                      ├─────────────────────────────┤
                      │                              │
                      ├─────────────────────────────┤
                      │ Packet_descriptor(...)       │
                      │ Packet_descriptor(...)       │
                      │ operation() : Opcode         │
                      │ block_number() : sector_t    │
                      │ block_count() : Genode::size_t│
                      │ succeeded() : bool           │
                      │ succeeded(...)               │
                      └─────────────────────────────┘
```

Accessors

operation	**Opcode**
block_number	**sector_t**
block_count	**Genode::size_t**
succeeded	**bool**

Details

The data associated with the Packet_descriptor is either the data read from or written to the block indicated by its number.

Header

repos/os/include/block_session/block_session.h

Block ⟩ Packet_descriptor ⟩ **Packet_descriptor**

                                                                                   constructor
Arguments

offset	**off_t** *Default is 0*
size	**size_t** *Default is 0*

GENODE LABS

Block 〉 Packet_descriptor 〉 **Packet_descriptor**

constructor

Arguments

p	**Packet_descriptor**
op	**Opcode**
blk_nr	**sector_t**
blk_count	**size_t**
	Default is **1**

Block 〉 Packet_descriptor 〉 **succeeded**

method

Argument

b	**bool**

Block 〉 **Session**

class

Session

```
          Block::Session

  service_name() : const char *
  ~Session()                          RPC interface
  info(...)
  sync()
  tx_channel() : Tx *
  tx() : Tx::Source *
```

Header

repos/os/include/block_session/block_session.h

Block 〉 Session 〉 **service_name**

class function

Return value

const char *

GENODE LABS

Block ⟩ Session ⟩ *info*

Request information about the metrics of the block device   pure virtual method

Arguments

blk_count   **sector_t ***
             Will contain total number of blocks

blk_size   **size_t ***
             Will contain total size in bytes

ops   **Operations ***
             Supported operations

Block ⟩ Session ⟩ *sync*

Synchronize with block device, like ensuring data to be written   pure virtual method

Block ⟩ Session ⟩ *tx_channel*

Request packet-transmission channel   virtual method

Return value

 Tx *

Block ⟩ Session ⟩ *tx*

Request client-side packet-stream interface of tx channel   virtual method

Return value

 Tx::Source *

GENODE LABS

Block	**Connection**		class

Connection to block service

Connection <Session>   **Session_client**

Block::Connection

Connection(...)

Header

repos/os/include/block_session/connection.h

Block	Connection	**Connection**

constructor

Arguments

env   **Env &**

tx_block_alloc   **Range_allocator ***

tx_buf_size   **size_t**
               Size of transmission buffer in bytes
               Default is 128∗1024

label   **const char ***
               Default is ""

GENODE LABS
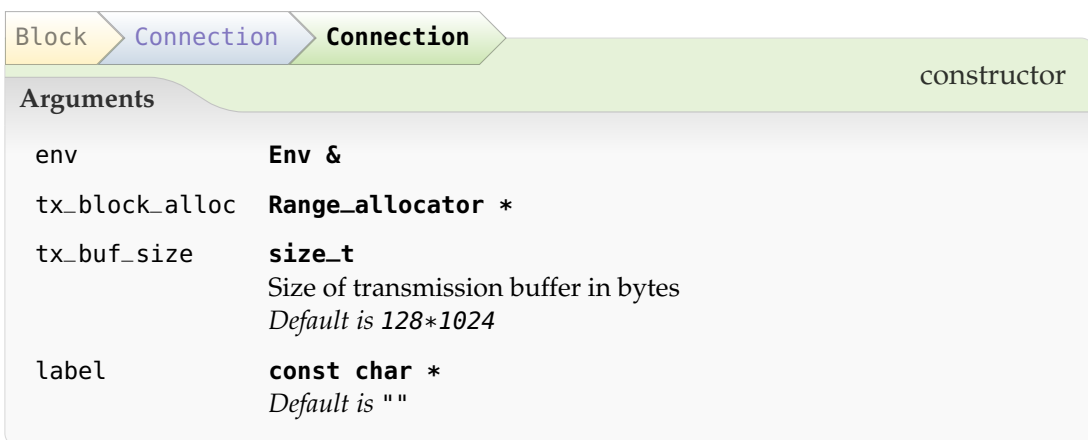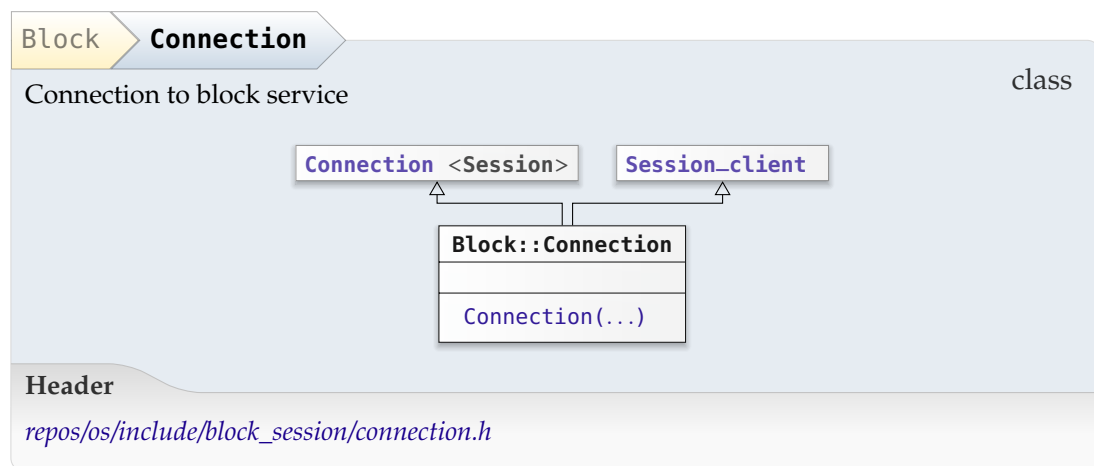
8.6.8. Regulator session interface

Regulator	Session			class

Abstract regulator session interface.

```
                          ┌──────────┐
                          │ Session  │
                          └──────────┘
                               △
              ┌────────────────┴──────────────────┐
              │        Regulator::Session          │
              ├────────────────────────────────────┤
              ├────────────────────────────────────┤
              │ service_name() : const char *      │
              │ ~Session()                         │ ·········  RPC interface
              │ level(...)                         │
              │ level() : unsigned long            │
              │ state(...)                         │
              │ state() : bool                     │
              └────────────────────────────────────┘
```

Header

repos/os/include/regulator_session/regulator_session.h

Regulator	Session	**service_name**	class function

Return value

const char *

Regulator	Session	*level*	pure virtual method

Set regulator specific level

Argument

level   **unsigned long**

Regulator	Session	*level*	pure virtual method

Return value

unsigned long   Current regulator level

GENODE LABS

Regulator > Session > **state**

pure virtual method

Enable/disable regulator

Argument

enable   **bool**

Regulator > Session > **state**

pure virtual method

Return value

bool   Whether regulator is enabled or not

Regulator > **Connection**

class

Connection to regulator service

Connection <Session>   **Session_client**

Regulator::Connection

Connection(...)
Connection(...)

Header

repos/os/include/regulator_session/connection.h

Regulator > Connection > **Connection**

constructor

Arguments

env   **Env &**

regulator   **Regulator_id**
             Identifier for the specific regulator

label   **const char ***
             String identifier of the client
             Default is " "

GENODE LABS

Regulator Connection **Connection**

constructor

Arguments

regulator   **Regulator_id**
            Identifier for the specific regulator

label   **const char ***
            String identifier of the client
            Default is `""`

313

GENODE LABS
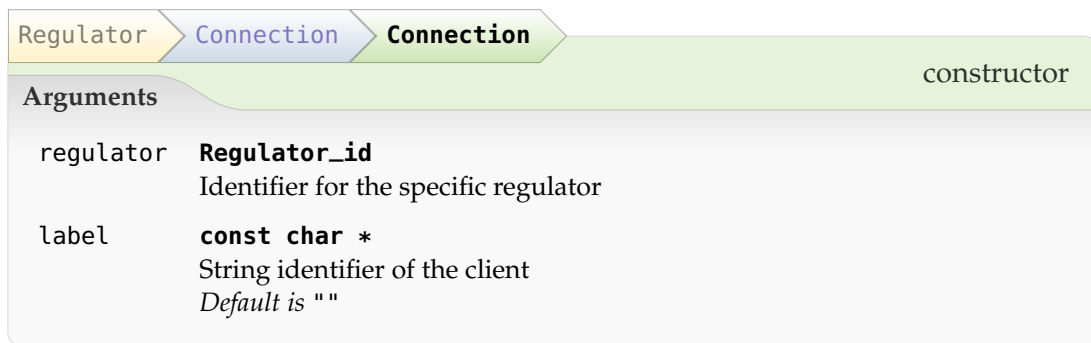
8.6.9. Timer session interface

Timer **Session**

Timer session interface   class

Session

Timer::Session
service_name() : const char *
~Session()
trigger_once(...)
trigger_periodic(...)
sigh(...)
elapsed_ms() : unsigned long
msleep(...)
usleep(...)

RPC interface

Header

repos/os/include/timer_session/timer_session.h

Timer   Session   **service_name**

Return value   class function

  const char *

Timer   Session   *trigger_once*

Program single timeout (relative from now in microseconds)   pure virtual method

Argument

 us   **unsigned**

314

GENODE ⬙ LABS

`Timer` `Session` ***trigger_periodic***

Program periodic timeout (in microseconds)

pure virtual method

Argument

us   **unsigned**

Details

The first period will be triggered after us at the latest, but it might be triggered earlier as well.

`Timer` `Session` ***sigh***

Register timeout signal handler

pure virtual method

Argument

sigh   **Signal_context_capability**

`Timer` `Session` ***elapsed_ms***

pure virtual const method

Return value

unsigned long   Number of elapsed milliseconds since session creation

`Timer` `Session` ***msleep***

Client-side convenience method for sleeping the specified number of milliseconds

pure virtual method

Argument

ms   **unsigned**

`Timer` `Session` ***usleep***

Client-side convenience method for sleeping the specified number of microseconds

pure virtual method

Argument

us   **unsigned**

GENODE LABS

Timer **Connection**

Connection to timer service   class

Connection <Session>   Session_client

```
Timer::Connection

Connection(...)
~Connection()
sigh(...)
usleep(...)
msleep(...)
```

Header

repos/os/include/timer_session/connection.h

Timer Connection **Connection**

                                                                    constructor
Argument

 env **Env &**

Timer Connection **sigh**

                                                                        method
Argument

 sigh **Signal_context_capability**

Timer Connection **usleep**

                                                                        method
Argument

 us **unsigned**

Timer Connection **msleep**

                                                                        method
Argument

 ms **unsigned**

GENODE LABS

8.6.10. NIC session interface

Nic ⟩ **Mac_address**

class

`Nic::Mac_address`

Header

repos/os/include/nic_session/nic_session.h

Nic ⟩ **Session**

class

Session

```
          Nic::Session

service_name() : const char *
~Session()
mac_address() : Mac_address
tx_channel() : Tx *
rx_channel() : Rx *
tx() : Tx::Source *
rx() : Rx::Sink *
link_state() : bool
link_state_sigh(...)
```

RPC interface

Header

repos/os/include/nic_session/nic_session.h

Nic ⟩ Session ⟩ **service_name**

class function

Return value

const char *

Nic ⟩ Session ⟩ *mac_address*

pure virtual method

Request MAC address of network adapter

Return value

Mac_address

GENODE LABS

Nic Session *tx_channel*

Request packet-transmission channel

virtual method

Return value

`Tx *`

Nic Session *rx_channel*

Request packet-reception channel

virtual method

Return value

`Rx *`

Nic Session *tx*

Request client-side packet-stream interface of tx channel

virtual method

Return value

`Tx::Source *`

Nic Session *rx*

Request client-side packet-stream interface of rx channel

virtual method

Return value

`Rx::Sink *`

Nic Session *link_state*

Request current link state of network adapter (true means link detected)

pure virtual method

Return value

`bool`

GENODE LABS

Nic	Session	***link_state_sigh***

Register signal handler for link state changes

pure virtual method

Argument

  sigh   **Signal_context_capability**

Nic	**Connection**

Connection to NIC service

class

Connection <Session>   **Session_client**

Nic::Connection

Connection(...)

Header

repos/os/include/nic_session/connection.h

Nic	Connection	**Connection**

constructor

Arguments

  env   **Env &**

  tx_block_alloc **Range_allocator ***

  tx_buf_size   **size_t**
                 Size of transmission buffer in bytes

  rx_buf_size   **size_t**
                 Size of reception buffer in bytes

  label   **char const ***
                 Default is ""

GENODE LABS

8.6.11. Audio-out session interface

Audio_out	**Session**	

Audio_out session base

class

```
                              Session
                                 △
                                 ┊
              ┌─────────────────────────────────────────┐
              │            Audio_out::Session            │
              ├─────────────────────────────────────────┤
              ├─────────────────────────────────────────┤
              │  service_name() : const char *           │
              │  stream() : Stream *                     │
              │  start()                                 │
              │  stop()                                  │
              │  progress_sigh(...)                      │
              │  alloc_sigh(...)                         │
              │  data_avail_sigh() : Genode::Signal_context_capability │
              └─────────────────────────────────────────┘
```

RPC interface

Header

repos/os/include/audio_out_session/audio_out_session.h

Audio_out	Session	**service_name**

class function

Return value

const char *

Audio_out	Session	**stream**

const method

Return value

Stream *   Stream of this session, see stream above

Audio_out	Session	*start*

Start playback (alloc and submit packets after calling start)   pure virtual method

---

Audio_out	Session	*stop*

Stop playback   pure virtual method

GENODE LABS

Audio_out > Session > **progress_sigh**

The progress signal is sent from the server to the client if a   pure virtual method
packet has been played.

Argument

  sigh   **Signal_context_capability**

Details

See: client.h, connection.h

Audio_out > Session > **alloc_sigh**

The alloc signal is sent from the server to the client when the   pure virtual method
stream queue leaves the full state.

Argument

  sigh   **Signal_context_capability**

Details

See: client.h, connection.h

Audio_out > Session > **data_avail_sigh**

The data_avail signal is sent from the client to the server if the   pure virtual method
stream queue leaves the empty state.

Return value

  Genode::Signal_context_capability

GENODE LABS

Audio_out 〉 **Connection**

Connection to audio service

class

Connection <Session>   Audio_out::Session_client

Audio_out::Connection

Connection(...)

Header

repos/os/include/audio_out_session/connection.h

Audio_out 〉 Connection 〉 **Connection**

constructor

Arguments

env	**Env &**
channel	**char const *** Channel identifier (e. g., "front left")
alloc_signal	**bool** Install `alloc_signal`, the client may then use `wait_for_alloc` when the stream is full *Default is* `true`
progress_signal	**bool** Install progress signal, the client may then call `wait_for_progress`, which is sent when the server processed one or more packets *Default is* `false`

GENODE ⫽ LABS

8.6.12. File-system session interface

The file-system session (Section 4.5.13) interface provides to store and retrieve data in the form of files organized in a hierarchic directory structure. Directory operations are performed via a synchronous RPC interface whereas the actual read and write operations are performed asynchronously using a packet stream.

A node handle refers to an open directory node, which can be a file, a directory, or a symlink. It is a number local to the file-system session.

File_system ⟩ **Node_handle**

class

```
File_system::Node_handle

Node_handle()
Node_handle(...)
valid() : bool
operator ==(...) : bool
operator !=(...) : bool
```

Accessor

valid   **bool**

Header

repos/os/include/file_system_session/file_system_session.h

File_system ⟩ Node_handle ⟩ **Node_handle**

constructor

Argument

v   **int**

File_system ⟩ Node_handle ⟩ **operator ==**

const method

Argument

other   **Node_handle const &**

Return value

bool

GENODE LABS

File_system	Node_handle	**operator !=**	
			const method

Argument

other   **Node_handle const &**

Return value

bool

File_system

File-system session interface   namespace

Types

File_handle	is a subtype of Node_handle
Dir_handle	is a subtype of Node_handle
Symlink_handle	is a subtype of Node_handle
seek_off_t	is defined as Genode::uint64_t
file_size_t	is defined as Genode::uint64_t
Mode	is an enumeration type
	Flags as supplied to file, dir, and symlink calls
Name	is defined as Genode::Rpc_in_buffer<MAX_NAME_LEN>
Path	is defined as Genode::Rpc_in_buffer<MAX_PATH_LEN>
Exception	is a subtype of Genode::Exception

Header

repos/os/include/file_system_session/file_system_session.h

A file-system client issues read or write requests by submitting packet descriptors to the file-system session's packet stream. Each packet descriptor contains all parameters of the transaction including the type of operation, the seek offset, and the length.

GENODE ⬙ LABS

File_system ‖ **Packet_descriptor**

class

```
                    Packet_descriptor
                            △
          ┌──────────────────────────────────┐
          │ File_system::Packet_descriptor   │
          ├──────────────────────────────────┤
          │                                  │
          │ Packet_descriptor(...)           │
          │ Packet_descriptor(...)           │
          │ handle() : Node_handle           │
          │ operation() : Opcode             │
          │ position() : seek_off_t          │
          │ length() : size_t                │
          │ succeeded() : bool               │
          │ succeeded(...)                   │
          │ length(...)                      │
          └──────────────────────────────────┘
```

Accessors

handle	**Node_handle**
operation	**Opcode**
position	**seek_off_t**
length	**size_t**
succeeded	**bool**

Header

repos/os/include/file_system_session/file_system_session.h

File_system ‖ Packet_descriptor ‖ **Packet_descriptor**

constructor

Arguments

buf_offset	**off_t**
	Default is 0
buf_size	**size_t**
	Default is 0

GENODE ‖ LABS

File_system ⟩ Packet_descriptor ⟩ **Packet_descriptor**

constructor

Arguments

p	**Packet_descriptor**
handle	**Node_handle**
op	**Opcode**
length	**size_t**
position	**seek_off_t**
	Seek offset in bytes
	Default is SEEK_TAIL

Details

Note, if position is set to SEEK_TAIL read operations will read length bytes from the end of the file while write operations will append length bytes at the end of the file.

File_system ⟩ Packet_descriptor ⟩ **succeeded**

method

Argument

b	**bool**

File_system ⟩ Packet_descriptor ⟩ **length**

method

Argument

length	**size_t**

GENODE LABS

File_system 〉 **Session**

class

Session

┌─────────────────────────────────┐
│   **File_system::Session**   │
├─────────────────────────────────┤
├─────────────────────────────────┤
│ service_name() : const char *   │
│ ~Session()   │
│ tx() : Tx::Source *   │
│ file(...) : File_handle   │
│ symlink(...) : Symlink_handle   │
│ dir(...) : Dir_handle   │
│ node(...) : Node_handle   │
│ close(...)   │
│ status(...) : Status   │
│ control(...)   │
│ unlink(...)   │
│ truncate(...)   │
│ move(...)   │
│ sigh(...)   │
│ sync(...)   │
└─────────────────────────────────┘

RPC interface

Header

repos/os/include/file_system_session/file_system_session.h

File_system 〉 Session 〉 **service_name**

Return value

class function

const char *

File_system 〉 Session 〉 *tx*

Request client-side packet-stream interface of tx channel

virtual method

Return value

Tx::Source *

GENODE LABS

File_system	Session	*file*

Open or create file   pure virtual method

Arguments

-	**Dir_handle**
name	**Name const &**
-	**Mode**
create	**bool**

Exceptions

Invalid_handle	Directory handle is invalid
Invalid_name	File name contains invalid characters
Lookup_failed	The name refers to a node other than a file
Node_already_exists	File cannot be created because a node with the same name already exists
No_space	Storage exhausted
Out_of_metadata	Server cannot allocate metadata
Permission_denied	

Return value

File_handle

GENODE LABS

File_system	Session	*symlink*	

Open or create symlink **pure virtual method**

Arguments

-	**Dir_handle**
name	**Name const &**
create	**bool**

Exceptions

Invalid_handle	Directory handle is invalid
Invalid_name	Symlink name contains invalid characters
Lookup_failed	The name refers to a node other than a symlink
Node_already_exists	Symlink cannot be created because a node with the same name already exists
No_space	Storage exhausted
Out_of_metadata	Server cannot allocate metadata
Permission_denied	

Return value

Symlink_handle

File_system	Session	*dir*	

Open or create directory **pure virtual method**

Arguments

path	**Path const &**
create	**bool**

Exceptions

Lookup_failed	Path lookup failed because one element of path does not exist
Name_too_long	path is too long
Node_already_exists	Directory cannot be created because a node with the same name already exists
No_space	Storage exhausted
Out_of_metadata	Server cannot allocate metadata
Permission_denied	

Return value

Dir_handle

GENODE LABS

File_system	Session	*node*	
Open existing node			pure virtual method

Argument

path   **Path const &**

Exceptions

Lookup_failed   Path lookup failed because one element of path does not exist
Out_of_metadata   Server cannot allocate metadata

Return value

Node_handle

Details

The returned node handle can be used merely as argument for status.

File_system	Session	*close*	
Close file			pure virtual method

Argument

-    **Node_handle**

File_system	Session	*status*	
Request information about an open file or directory			pure virtual method

Argument

-    **Node_handle**

Return value

Status

File_system	Session	*control*	
Set information about an open file or directory			pure virtual method

Arguments

-    **Node_handle**

-    **Control**

GENODE LABS

File_system > Session > *unlink*

Delete file or directory

pure virtual method

Arguments

| dir | **Dir_handle** |
| name | **Name const &** |

Exceptions

Invalid_handle	Directory handle is invalid
Invalid_name	name contains invalid characters
Lookup_failed	Lookup of name in dir failed
Not_empty	Argument is a non-empty directory and the backend does not support recursion
Permission_denied	

File_system > Session > *truncate*

Truncate or grow file to specified size

pure virtual method

Arguments

| - | **File_handle** |
| size | **file_size_t** |

Exceptions

Invalid_handle	Node handle is invalid
No_space	New size exceeds free space
Permission_denied	Node modification not allowed

GENODE LABS

File_system 〉 Session 〉 ***move***

Move and rename directory entry
                                                  pure virtual method

Arguments

-	**Dir_handle**
from	**Name const &**
-	**Dir_handle**
to	**Name const &**

Exceptions

Invalid_handle	A directory handle is invalid
Invalid_name	to contains invalid characters
Lookup_failed	from not found
Permission_denied	Node modification not allowed

File_system 〉 Session 〉 ***sigh***

Register handler that should be notified on node changes
                                                  pure virtual method

Arguments

| - | **Node_handle** |
| sigh | **Signal_context_capability** |

File_system 〉 Session 〉 ***sync***

Synchronize file system
                                                  virtual method

Argument

| - | **Node_handle** |

Details

This is only needed by file systems that maintain an internal cache, which needs to be
flushed on certain occasions.

GENODE LABS

File_system 〉 **Connection**

A File_system connection that upgrades its RAM quota   class

```
File_system::Connection_base
                  △
                  ┊
     File_system::Connection

   upgrade_ram()
   dir(...) : Dir_handle
   file(...) : File_handle
   symlink(...) : Symlink_handle
   node(...) : Node_handle
```

Header

repos/os/include/file_system_session/connection.h

File_system 〉 Connection 〉 **upgrade_ram**

Upgrade the session quota in response to Out_of_metadata   method

GENODE ⬙ LABS

The file-system session's `status` and `control` operations use the compound struc-
tures `Status` and `Control` as arguments. The format of the data retrieved by reading a
directory node is defined by the `Directory_entry`.

File_system 〉 **Status**

class

> **File_system::Status**
>
> directory() : bool
> symlink() : bool

Header

repos/os/include/file_system_session/file_system_session.h

File_system 〉 Status 〉 **directory**

Return value

const method

bool   True if node is a directory

File_system 〉 Status 〉 **symlink**

Return value

const method

bool   True if node is a symbolic link

File_system 〉 **Control**

class

> **File_system::Control**

Header

repos/os/include/file_system_session/file_system_session.h

GENODE LABS

File_system ⟩ **Directory_entry**

Data structure returned when reading from a directory node

class

File_system::Directory_entry

Header

repos/os/include/file_system_session/file_system_session.h

GENODE LABS

8.6.13. Loader session interface

Loader ⟩ **Session**

Loader session interface   class

Session

Loader::Session
<u>service_name()</u> : const char ∗
~Session()
alloc_rom_module(...) : Dataspace_capability
commit_rom_module(...)
ram_quota(...)
constrain_geometry(...)
parent_view(...)
view_ready_sigh(...)
fault_sigh(...)
start(...)
view_geometry(...)
view_size() : Area

········ RPC interface

Header

repos/os/include/loader_session/loader_session.h

Loader ⟩ Session ⟩ **service_name**

                                                                    class function
Return value

 const char ∗

GENODE LABS

Loader	Session	*alloc_rom_module*

Allocate dataspace to be used as ROM module by the loaded subsystem

pure virtual method

Arguments

name **Name const &**
Designated name of the rom module

size **size_t**
Size of rom module

Return value

Dataspace_capability   Dataspace_capability dataspace that contains the backing store of the rom module

Details

The content of the dataspace is made visible to the loaded subsystem not before commit_rom_module has been called. This two-step procedure enables the client to update the content of the ROM module during the lifetime of the session by subsequently calling dataspace with the same name as argument. Each time, a new dataspace is allocated but not yet presented to the loaded subsystem. When calling commit_rom_module, the most recently allocated dataspace becomes visible. The server frees intermediate dataspaces that are no longer used.

Loader	Session	*commit_rom_module*

Expose ROM module to loaded subsystem

pure virtual method

Argument

name **Name const &**

Exception

Rom_module_does_not_exist   If the rom module name wasn't allocated beforehand

GENODE LABS

Loader 〉 Session 〉 *ram_quota*

Define RAM quota assigned to the subsystem   pure virtual method

Argument

 quantum   **size_t**

Details

The quantum specified must be in the bounds of the quota attached the session. Note that
RAM resources used for ROM modules are accounted, too. If ROM modules are modified at
runtime by subsequent calls of alloc_rom_module, the resources needed for the respective
ROM modules are doubled.
If ram_quota is not called prior calling start, all available session resources will be assigned
to the subsystem.

Loader 〉 Session 〉 *constrain_geometry*

Constrain size of the nitpicker buffer used by the subsystem   pure virtual method

Argument

 size   **Area**

Details

Calling this method prior start() enables the virtualization of the nitpicker session inter-
face.

Loader 〉 Session 〉 *parent_view*

Set the parent view of the subsystem's view.   pure virtual method

Argument

 view   **View_capability**

Details

If parent_view is not called prior calling start, the subsystem's view will not have a parent
view.

Loader 〉 Session 〉 *view_ready_sigh*

Register signal handler notified at creation time of the first view   pure virtual method

Argument

 sigh   **Signal_context_capability**

GENODE LABS

Loader ⟩ Session ⟩ ***fault_sigh***

pure virtual method

Register signal handler notified when a failure occurs in the loaded subsystem.

Argument

sigh **Signal_context_capability**

Details

This signal is delivered if any child process of the subsystem produces an unresolvable page fault, exists, or triggers a CPU exception. (e. g., division by zero) For more information about the possible types of faults, please refer to the documentation of `Rm_session::fault_handler` and `Cpu_session::exception_handler`.
This method should not be called after the `start` method.

Loader ⟩ Session ⟩ ***start***

pure virtual method

Start subsystem

Arguments

binary **Name const &**

label **Name const &**
Default is `""`

Exception

Rom_module_does_not_exist If the specified binary could not obtained as rom module

Loader ⟩ Session ⟩ ***view_geometry***

pure virtual method

Set view geometry and buffer offset

Arguments

rect **Rect**

offset **Point**

Loader ⟩ Session ⟩ ***view_size***

pure virtual const method

Return value

Area View size as initialized by the loaded subsystem

GENODE LABS

Loader	**Connection**		class

Connection to Loader service

Connection `<Session>`   **Session_client**

Loader::Connection

Connection(...)

Header

repos/os/include/loader_session/connection.h

Loader	Connection	**Connection**	constructor

Arguments

env   **Env &**

ram_quota **size_t**

GENODE LABS
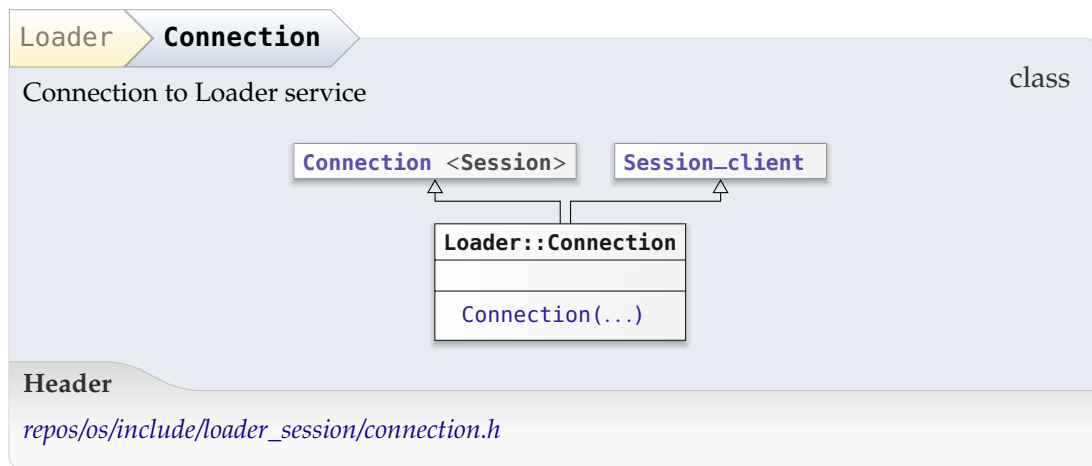
8.7. Fundamental types

8.7.1. Integer types

Genode provides common integer types in its namespace. Integer types that can be derived from built-in compiler types are defined in *base/stdint.h* and *base/fixed_stdint.h*. Whereas the former is independent from the machine type, the latter differs between 32-bit and 64-bit architectures.

Genode   namespace

Integer types

Types

   size_t   is defined as *SIZE_TYPE*

                Integer type for non-negative size values

   addr_t   is defined as `unsigned long`

                Integer type for memory addresses

   off_t   is defined as `long`

                Integer type for memory offset values

   umword_t is defined as `unsigned long`

                Integer type corresponding to a machine register

Header

repos/base/include/base/stdint.h

GENODE LABS

The fixed-width integer types for 32-bit architectures are defined as follows.

Fixed-width integer types for 32-bit architectures	root namespace

Types

genode_int8_t	is defined as `signed char`
genode_uint8_t	is defined as `unsigned char`
genode_int16_t	is defined as `signed short`
genode_uint16_t	is defined as `unsigned short`
genode_int32_t	is defined as `signed`
genode_uint32_t	is defined as `unsigned`
genode_int64_t	is defined as `signed long long`
genode_uint64_t	is defined as `unsigned long long`

Header

repos/base/include/spec/32bit/base/fixed_stdint.h

Genode

Fixed-width integer types for 32-bit architectures	namespace

Types

int8_t	is defined as `genode_int8_t`
uint8_t	is defined as `genode_uint8_t`
int16_t	is defined as `genode_int16_t`
uint16_t	is defined as `genode_uint16_t`
int32_t	is defined as `genode_int32_t`
uint32_t	is defined as `genode_uint32_t`
int64_t	is defined as `genode_int64_t`
uint64_t	is defined as `genode_uint64_t`

Header

repos/base/include/spec/32bit/base/fixed_stdint.h

GENODE LABS

The fixed-width integer types for 64-bit architectures are defined as follows.

Fixed-width integer types for 64-bit architectures	root namespace

Types

genode_int8_t   is defined as `signed char`

genode_uint8_t   is defined as `unsigned char`

genode_int16_t   is defined as `signed short`

genode_uint16_t   is defined as `unsigned short`

genode_int32_t   is defined as `signed`

genode_uint32_t   is defined as `unsigned`

genode_int64_t   is defined as `signed long long`

genode_uint64_t   is defined as `unsigned long long`

Header

repos/base/include/spec/64bit/base/fixed_stdint.h

Genode

Fixed-width integer types for 64-bit architectures	namespace

Types

int8_t   is defined as `genode_int8_t`

uint8_t   is defined as `genode_uint8_t`

int16_t   is defined as `genode_int16_t`

uint16_t   is defined as `genode_uint16_t`

int32_t   is defined as `genode_int32_t`

uint32_t   is defined as `genode_uint32_t`

int64_t   is defined as `genode_int64_t`

uint64_t   is defined as `genode_uint64_t`

Header

repos/base/include/spec/64bit/base/fixed_stdint.h

GENODE LABS

8.7.2. Exception types

Genode facilitates the use of exceptions to signal errors but it uses exception types only as textual expression of error code and for grouping errors. Normally, exceptions do not carry payload. For code consistency, exception types should inherit from the Exception base class. By convention, exceptions carry no payload.

Genode	**Exception**		class
Exception base class			
	Exception		
Header			
repos/base/include/base/exception.h			

8.7.3. C++ supplements

Genode	**Noncopyable**		class
Classes of objects not allowed to be copied, should inherit from this one.			
	Noncopyable		
Details			
This class declares a private copy-constructor and assignment-operator. It's sufficient to inherit private from this class, and let the compiler detect any copy violations.			
Header			
repos/base/include/util/noncopyable.h			

GENODE LABS

8.8. Data structures

The framework API features a small library of data structures that are solely exist to support the framework implementation. They have been designed under the following considerations:

- They should be **as simple as possible** to make them easy to evaluate for correctness. Low complexity takes precedence over performance.

- Each data structure provides a **rigid interface** that is targeted at a specific use case and does not attempt to be a power tool. For example, the `Fifo` deliberately provides no way to randomly access elements, the `List` merely provides the functionality to remember elements but lacks list-manipulation operations.

- Data structures perform no hidden anonymous memory allocations by **storing meta data intrusively**. This is precondition for allowing resources multiplexers and runtime environments to properly account their local memory allocations. Section 3.3.3 provides the rationale behind the need for full control over memory allocations.

GENODE ⟋⟋ LABS

8.8.1. List

Most book-keeping tasks in Genode rely on single-connected lists, which use the `List` template.

Genode	List		class template
Single-connected list			

```
                          List                    LT
        ────────────────────────────
        List()
        first() : LT *
        first() : LT const *
        insert(...)
        remove(...)
```

Template argument

LT **typename**
    List element type

Accessor

first **LT const ∗**

Header

repos/base/include/util/list.h

Genode	List	first	method

Return value

LT ∗   First list element

GENODE LABS
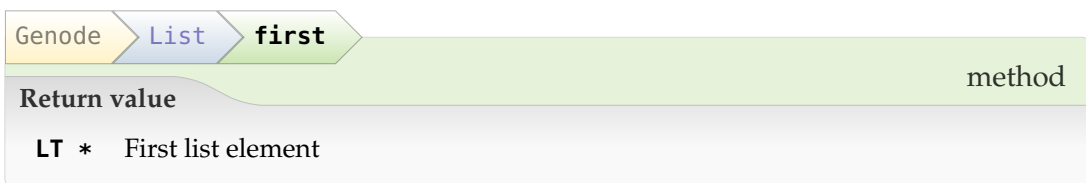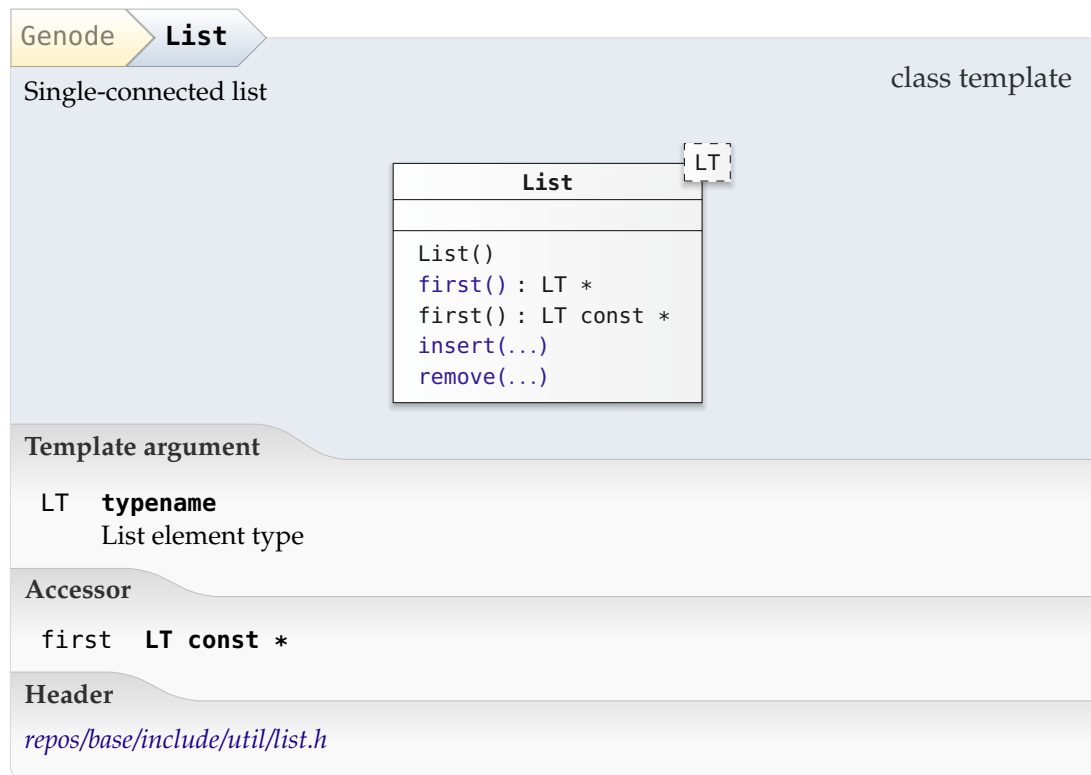
Genode ⟩ List ⟩ **insert**

Insert element after specified element into list   method

Arguments

le **LT const ***
    List element to insert

at **LT const ***
    Target position (preceding list element)
    Default is 0

Genode ⟩ List ⟩ **remove**

Remove element from list   method

Argument

le **LT const ***

GENODE ⟩ LABS

Genode **List_element**

Helper for using member variables as list elements

class template

Template argument

  T **typename**
    Type of compound object to be organized in a list

Accessor

  object **T** *

Details

This helper allow the creation of lists that use member variables to connect their elements. This way, the organized type does not need to publicly inherit `List<LT>::Element`. Furthermore objects can easily be organized in multiple lists by embedding multiple `List_element` member variables.

Header

repos/base/include/util/list.h

Genode   List_element   **List_element**

constructor

Argument

  object **T** *

GENODE LABS

8.8.2. Fifo queue

Because the `List` inserts new list elements at the list head, it cannot be used for implementing wait queues requiring first-in-first-out semantics. For such use cases, there exists a dedicated `Fifo` template.

Genode › **Fifo**

Fifo queue   class template

```
                              ┌───┐
                              │QT │
      ┌──────────────────────┴───┴─┐
      │          Fifo               │
      ├─────────────────────────────┤
      │                             │
      ├─────────────────────────────┤
      │ empty() : bool              │
      │ Fifo()                      │
      │ head() : QT *               │
      │ remove(...)                 │
      │ enqueue(...)                │
      │ dequeue() : QT *            │
      └─────────────────────────────┘
```

Template argument

QT **typename**
    Queue element type

Header

repos/base/include/util/fifo.h

Genode › Fifo › **empty**

                                                                  method
Return value

bool   True if queue is empty

Genode › Fifo › **head**

                                                              const method
Return value

QT *   First queue element

GENODE LABS
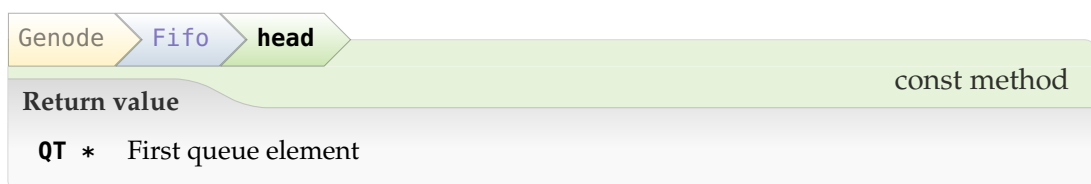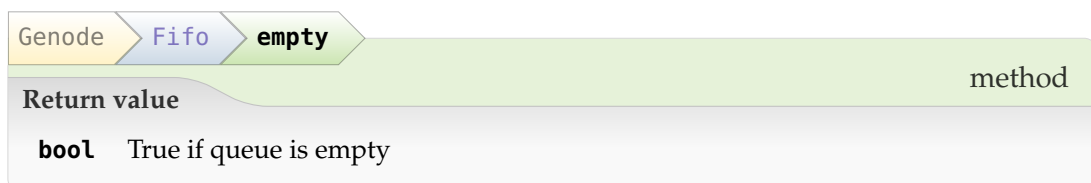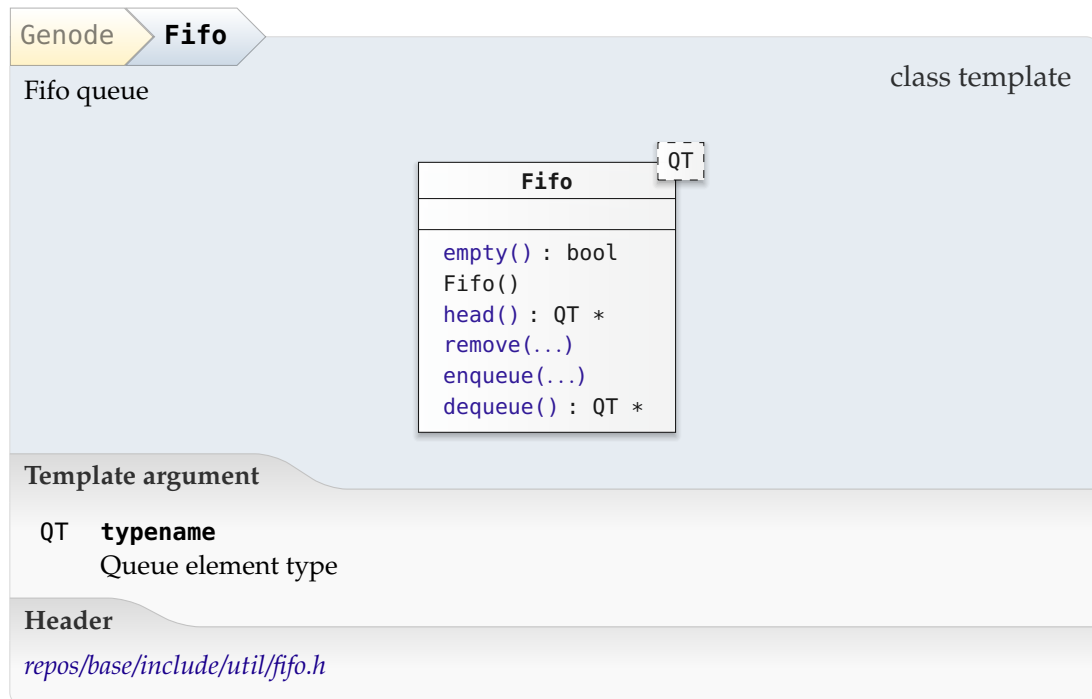
Genode 〉 Fifo 〉 **remove**

Remove element explicitly from queue

method

Argument

  qe **QT** *

Genode 〉 Fifo 〉 **enqueue**

Attach element at the end of the queue

method

Argument

  e **QT** *

Genode 〉 Fifo 〉 **dequeue**

Remove head element from queue

method

Return value

  QT *   Head element or 0 if queue is empty

GENODE LABS

Genode ❭ **Fifo_element**

Helper for using member variables as FIFO elements

class template

Template argument

T **typename**
   Type of compound object to be organized in a fifo

Details

This helper allow the creation of FIFOs that use member variables to connect their elements. This way, the organized type does not need to publicly inherit `Fifo<QT>::Element`. Furthermore objects can easily be organized in multiple FIFOs by embedding multiple `Fifo_element` member variables.

Header

repos/base/include/util/fifo.h

Genode ❭ Fifo_element ❭ **Fifo_element**

constructor

Argument

object   **T ∗**

Genode ❭ Fifo_element ❭ **object**

method

Get typed object pointer

Return value

T ∗

Details

Zero-pointer save: Returns 0 if this pointer is 0 to cover the case of accessing an empty FIFO.

GENODE LABS

8.8.3. AVL tree

For use cases where associative arrays are needed such as allocators, there is class template for creating AVL trees. The tree-balancing mechanism is implemented in the `Avl_node_base` class. The actual `Avl_node` and `Avl_tree` classes are tagged with the element type, which ensures that each AVL tree hosts only one type of elements.

Genode ⟩ **Avl_node_base**	
AVL tree	class

```
Avl_node_base

Avl_node_base()
insert(...)
remove(...)
```

Header

repos/base/include/util/avl_tree.h

Genode ⟩ Avl_node_base ⟩ **insert**	
Insert new node into subtree	method

Arguments

node	**Avl_node_base ***
policy	**Policy &**

Genode ⟩ Avl_node_base ⟩ **remove**	
Remove node from tree	method

Argument

policy	**Policy &**

GENODE LABS
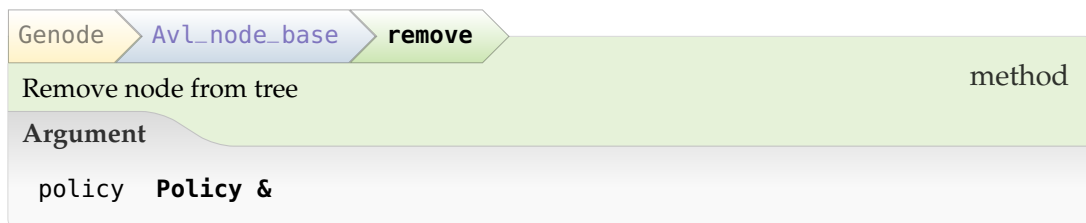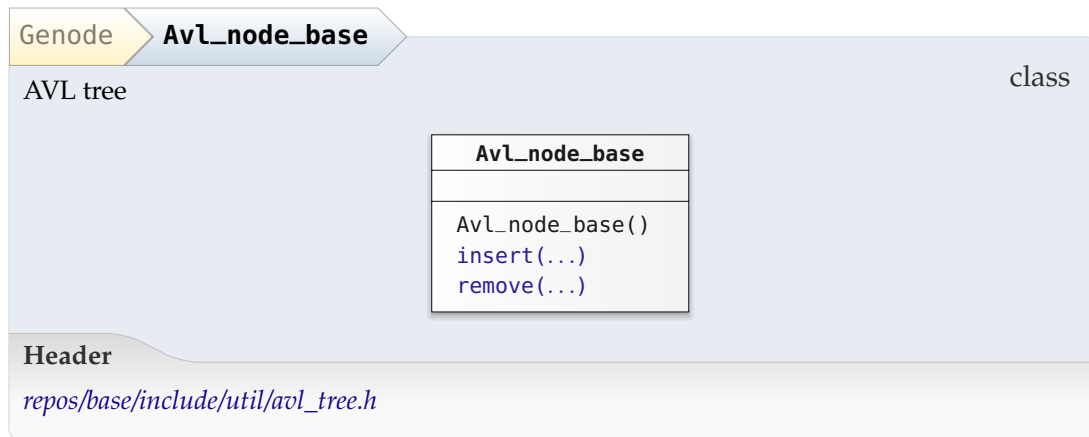
Genode **Avl_node**

AVL node   class template

Template argument

NT **typename**
     Type of the class derived from `avl_node`

Details

Each object to be stored in the AVL tree must be derived from `Avl_node`. The type of the derived class is to be specified as template argument to enable `Avl_node` to call virtual methods specific for the derived class.

The NT class must implement a method called `higher` that takes a pointer to another NT object as argument and returns a bool value. The bool value is true if the specified node is higher or equal in the tree order.

Header

repos/base/include/util/avl_tree.h

Genode Avl_node **child**

                                                                          const method

Argument

i **Side**

Return value

NT ∗ Child of specified side, or nullptr if there is no child

Details

This method can be called by the NT objects to traverse the tree.

GENODE LABS

Genode   **Avl_tree**

Root node of the AVL tree

class template

```
                              ┌─ NT ─┐
        ┌──────────────────────────┐
        │       Avl_tree           │
        ├──────────────────────────┤
        │                          │
        ├──────────────────────────┤
        │  insert(...)             │
        │  remove(...)             │
        │  first() : NT *          │
        └──────────────────────────┘
```

Template argument

NT   **typename**

Details

The real nodes are always attached at the left branch of this root node.

Header

repos/base/include/util/avl_tree.h

Genode   Avl_tree   **insert**

Insert node into AVL tree

method

Argument

node   **Avl_node**<NT> *

Genode   Avl_tree   **remove**

Remove node from AVL tree

method

Argument

node   **Avl_node**<NT> *

Genode   Avl_tree   **first**

Request first node of the tree

const method

Return value

NT *   First node, or nullptr if the tree is empty

GENODE ⬡ LABS

8.8.4. Bit array

Bit arrays are typically used for the allocation of IDs. The `Bit_array_base` class operates on a memory buffer specified to its constructor. The `Bit_array` class template addresses the common case where the ID space is dimensioned at compile time. It takes the number of bits as arguments and hosts the memory buffer for storing the bits within the object.

Genode ⟩ **Bit_array_base**	
Allocator using bitmaps	class

```
            Bit_array_base

    Bit_array_base(...)
    get(...) : bool
    set(...)
    clear(...)
```

Header

repos/base/include/util/bit_array.h

Genode ⟩ Bit_array_base ⟩ **Bit_array_base**	
Arguments	constructor

bits	**unsigned**
addr	**addr_t ***
clear	**bool**

Genode ⟩ Bit_array_base ⟩ **get**	
Arguments	const method

index	**addr_t**
width	**addr_t**

Return value

bool	True if at least one bit is set between index until index + width - 1

GENODE LABS

Genode	Bit_array_base	**set**	method

Arguments

index	**addr_t const**
width	**addr_t const**

Genode	Bit_array_base	**clear**	method

Arguments

index	**addr_t const**
width	**addr_t const**

Genode	**Bit_array**	class template

Allocator using bitmaps

Template argument

BITS	**unsigned**

Header

repos/base/include/util/bit_array.h

GENODE LABS

8.9. Object lifetime management

8.9.1. Thread-safe weak pointers

Dangling pointers represent one of the most common cause for instabilities of software written in C or C++. Such a situation happens when an object disappears while pointers to the object are still in use. One way to solve this problem is to explicitly notify the holders of those pointers about the disappearance of the object. But this would require the object to keep references to those pointer holders, which, in turn, might disappear as well. Consequently, this approach tends to become a complex solution, which is prone to deadlocks or race conditions when multiple threads are involved.

The utilities provided by *base/weak_ptr.h* implement a more elegant pattern called "weak pointers" to deal with such situations. An object that might disappear at any time is represented by the `Weak_object` class template. It keeps track of a list of so-called weak pointers pointing to the object. A weak pointer, in turn, holds privately the pointer to the object alongside a validity flag. It cannot be used to dereference the object. For accessing the actual object, a locked pointer must be created from a weak pointer. If this creation succeeds, the object is guaranteed to be locked (not destructed) until the locked pointer gets destroyed. If the object no longer exists, the locked pointer will become invalid. This condition can (and should) be detected via the `Locked_ptr::is_valid()` function prior dereferencing the pointer.

In the event a weak object gets destructed, all weak pointers that point to the object are automatically invalidated. So a subsequent conversion into a locked pointer will yield an invalid pointer, which can be detected (in contrast to a dangling pointer).

To use this mechanism, the destruction of a weak object must be deferred until no locked pointer points to the object anymore. This is done by calling the function `Weak_object::lock_for_destruction()` at the beginning of the destructor of the to-be-destructed object. When this function returns, all weak pointers to the object will have been invalidated. So it is save to destruct and free the object.

GENODE 〰 LABS

Genode ⟩ **Weak_object_base**

Type-agnostic base class of a weak object

class

Weak_object_base
~Weak_object_base() disassociate(...) lock_for_destruction()

Header

repos/base/include/base/weak_ptr.h

Genode ⟩ Weak_object_base ⟩ **disassociate**

method

Argument

ptr   **Weak_ptr_base ***

Genode ⟩ Weak_object_base ⟩ **lock_for_destruction**

Mark object as safe to be destructed

method

Details

This method must be called by the destructor of a weak object to defer the destruction until no Locked_ptr is held to the object.

GENODE ⩘ LABS

Genode	**Weak_object**	class template

Weak object

Template argument

T **typename**
Type of the derived class

Details

This class template must be inherited in order to equip an object with the weak-pointer mechanism.

Header

repos/base/include/base/weak_ptr.h

Genode	Weak_object	**weak_ptr**	method

Obtain a weak pointer referring to the weak object

Return value

Weak_ptr<**T**>

Genode	Weak_object	**weak_ptr_const**	const method

Const version of weak_ptr

Return value

Weak_ptr<**T const**> **const**

Details

This function is useful in cases where the returned weak pointer is merely used for comparison operations.

GENODE LABS

Genode **Weak_ptr_base**

Type-agnostic base class of a weak pointer

class

```
List <Weak_ptr_base>
        ↑
  Weak_ptr_base
```

Details
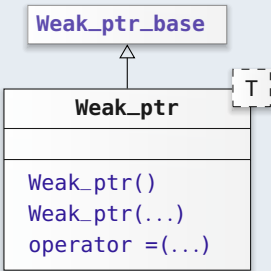
This class implements the mechanics of the the `Weak_ptr` class template. It is not used directly.

Header

repos/base/include/base/weak_ptr.h

GENODE LABS

Genode	**Weak_ptr**	

Weak pointer to a given type

class template

Weak_ptr_base

Weak_ptr	T
Weak_ptr()	
Weak_ptr(...)	
operator =(...)	

Template argument

T   **typename**

Details

A weak pointer can be obtained from a weak object (an object that inherits the `Weak_object` class template) and safely survives the lifetime of the associated weak object. If the weak object disappears, all weak pointers referring to the object are automatically invalidated. To avoid race conditions between the destruction and use of a weak object, a weak pointer cannot be de-reference directly. To access the object, a weak pointer must be turned into a locked pointer (`Locked_ptr`).

Header

repos/base/include/base/weak_ptr.h

Genode	Weak_ptr	**Weak_ptr**

Default constructor creates invalid pointer

constructor

Genode	Weak_ptr	**Weak_ptr**

Copy constructor

constructor

Argument

other   **Weak_ptr**<T> **const &**

GENODE LABS

Genode ⟩ Weak_ptr ⟩ **operator =**

method

Assignment operator

Argument

other   **Weak_ptr**<T> **const &**

Genode ⟩ **Locked_ptr_base**

class

Locked_ptr_base

Header

repos/base/include/base/weak_ptr.h

GENODE ⋙ LABS
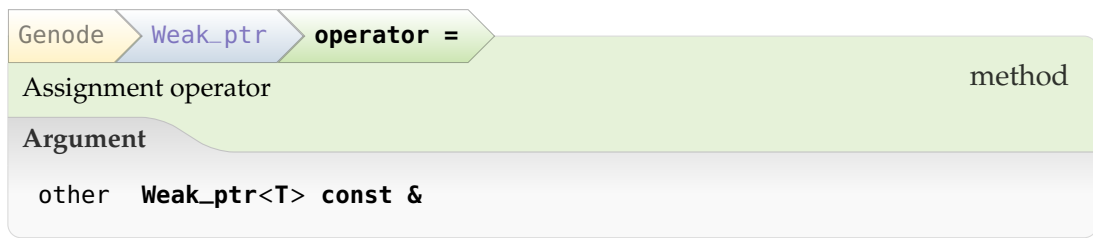
Genode **Locked_ptr**

Locked pointer
class template

```
                        Locked_ptr_base
                              △
                              ┆          ┌ ─ ┐
                        ┌─────┴──────────│ T │
                        │  Locked_ptr    └ ─ ┘
                        ├─────────────────────┤
                        │                     │
                        ├─────────────────────┤
                        │ Locked_ptr(...)     │
                        │ operator →() : T *  │
                        │ operator *() : T &  │
                        │ valid() : bool      │
                        └─────────────────────┘
```

Template argument

  T   **typename**

Details

A locked pointer is constructed from a weak pointer. After construction, its validity can
(and should) be checked by calling the `valid` method. If the locked pointer is valid, the
pointed-to object is known to be locked until the locked pointer is destroyed. During this
time, the locked pointer can safely be de-referenced.

The typical pattern of using a locked pointer is to declare it as a local variable. Once the
execution leaves the scope of the variable, the locked pointer is destructed, which unlocks
the pointed-to weak object. It effectively serves as a lock guard.

Header

repos/base/include/base/weak_ptr.h

Genode   Locked_ptr   **Locked_ptr**

constructor

Argument

  weak_ptr   **Weak_ptr<T> &**

Genode   Locked_ptr   **operator →**

method

Return value

  T *

GENODE LABS

Genode	Locked_ptr	**operator ***

method

Return value

`T &`

Genode	Locked_ptr	**valid**

const method

Return value

bool   True if the locked pointer is valid

Details

Only if valid, the locked pointer can be de-referenced. Otherwise, the attempt will result in a null-pointer access.

GENODE LABS

8.9.2. Volatile objects

The `construct_at` utility allows for the manual placement of objects without the need
to have a global placement new operation nor the need for type-specific new operators.

Genode

Utility for the manual placement of objects   namespace

Function

  • `Genode::construct_at(...) : T *`

Header

repos/base/include/util/construct_at.h

Genode > **Genode::construct_at**

Construct object of given type at a specific location   function template

Template arguments

  T   **typename**
         Object type

  ARGS   **typename...**

Arguments

  at   **void ***
         Desired object location

  args   **ARGS &&...**
         List of arguments for the object constructor

Return value

  T *   Typed object pointer

Details

We use move semantics (ARGS &&) because otherwise the compiler would create a tempo-
rary copy of all arguments that have a reference type and use a reference to this copy instead
of the original within this function.

There is a slight difference between the object that is constructed by this function and a com-
mon object of the given type. If the destructor of the given type or of any base of the given
type is virtual, the vtable of the returned object references an empty delete(void *) operator
for that destructor. However, this shouldn't be a problem as an object constructed by this
function should never get destructed implicitly or through a delete expression.

GENODE ⬚ LABS

The Genode framework promotes a programming style that largely avoids dynamic memory allocations. For the most part, higher-level objects aggregate lower-level objects as class members. This functional programming style leads to robust programs but it poses a problem for programs that are expected to adopt their behaviour at runtime. A way to selectively replace an aggregated object by a new version with updated constructor arguments is desired. The `Volatile_object` utility solves this problem by wrapping an object of the type specified as template argument. In contrast of a regular object, a `Volatile_object` can be re-constructed any number of times by calling `construct` with the constructor arguments. It is accompanied with a so-called `Lazy_volatile_object`, which remains unconstructed until `construct` is called the first time.

Genode › **Volatile_object**	
Place holder for an object to be repeatedly constructed and destructed	class template

```
                                                  ┌ ─ ─ ┐
                                                  ╎ MT  ╎
                      ┌──────────────────────────────────┐
                      │         Volatile_object          │
                      ├──────────────────────────────────┤
                      │                                  │
                      ├──────────────────────────────────┤
                      │  Volatile_object(...)            │
                      │  Volatile_object(...)            │
                      │  ~Volatile_object()              │
                      │  construct(...)                  │
                      │  destruct()                      │
                      │  constructed() : bool            │
                      │  operator →() : MT *             │
                      │  operator →() : MT const *       │
                      │  operator *() : MT &             │
                      └──────────────────────────────────┘
```

Template argument

MT   **typename**
       Type

Accessor

operator →   **MT const ***

Details

This class template acts as a smart pointer that refers to an object contained within the smart pointer itself. The contained object may be repeatedly constructed and destructed while staying in the same place. This is useful for replacing aggregated members during the lifetime of a compound object.

Header

repos/base/include/util/volatile_object.h

GENODE ⬙ LABS

Genode ❭ Volatile_object ❭ **Volatile_object**

constructor

Constructor that omits the initial construction of the object

Argument

- **Lazy ***

Genode ❭ Volatile_object ❭ **Volatile_object**

constructor template

Template argument

ARGS   **typename...**

Argument

args   **ARGS &&...**

Details

The arguments are forwarded to the constructor of the embedded object.

Genode ❭ Volatile_object ❭ **construct**

method template

Construct new object in place

Template argument

ARGS   **typename...**

Argument

args   **ARGS &&...**

Details

If the Volatile_object already hosts a constructed object, the old object will be destructed first.

Genode ❭ Volatile_object ❭ **destruct**

method

Destruct object

Genode ❭ Volatile_object ❭ **constructed**

const method

Return value

bool   True of volatile object contains a constructed object

GENODE ⩘ LABS

Genode ⟩ Volatile_object ⟩ **operator →**

Access contained object

method

Return value

`MT *`

Genode ⟩ Volatile_object ⟩ **operator ***

method

Return value

`MT &`

Genode ⟩ **Lazy_volatile_object**

Volatile object that holds no initially constructed object

class template

```
        Volatile_object <MT>
                  △
                  |                        ┌─ ─ ┐
     ┌────────────────────────────┐        | MT |
     │    Lazy_volatile_object     │        └ ─ ─┘
     ├────────────────────────────┤
     │  Lazy_volatile_object(...)  │
     └────────────────────────────┘
```

Template argument

MT **typename**

Header

repos/base/include/util/volatile_object.h

Genode ⟩ Lazy_volatile_object ⟩ **Lazy_volatile_object**

constructor template

Template argument

ARGS **typename...**

Argument

args **ARGS &&...**

GENODE LABS

8.10. Allocators

All allocators implement the generic `Deallocator` and `Allocator` interfaces. Allocators that operate on address ranges supplement the plain `Allocator` by implementing the more specific `Range_allocator` interface.

Genode 〉 **Deallocator**   class

Deallocator interface

Deallocator
free(...)
need_size_for_free() : bool

Header

repos/base/include/base/allocator.h

Genode 〉 Deallocator 〉 *free*   pure virtual method

Free block a previously allocated block

Arguments

  addr   **void ***

  size   **size_t**

Genode 〉 Deallocator 〉 *need_size_for_free*

                                           pure virtual const method

Return value

  bool   True if the size argument of `free` is required

Details

The generic `Allocator` interface requires the caller of `free` to supply a valid size argument but not all implementations make use of this argument. If this method returns false, it is safe to call `free` with an invalid size.

Allocators that rely on the size argument must not be used for constructing objects whose constructors may throw exceptions. See the documentation of `operator delete(void *, Allocator *)` below for more details.

GENODE ⬙ LABS

Genode   **Allocator**

class

Deallocator

Allocator
~Allocator()
alloc(...) : bool
alloc(...) : bool
consumed() : size_t
overhead(...) : size_t
alloc(...) : void *

Header

repos/base/include/base/allocator.h

Genode   Allocator   ***alloc***

Allocate block

pure virtual method

Arguments

size	**size_t**
	Block size to allocate
out_addr	**void **
	Resulting pointer to the new block, undefined in the error case

Return value

bool   True on success

GENODE LABS

Genode > Allocator > **alloc**

Allocate typed block   method template

Template argument

  T   **typename**

Arguments

  size   **size_t**

  out_addr   **T ****

Return value

  bool

Details

This template allocates a typed block returned as a pointer to a non-void type. By providing
this method, we prevent the compiler from warning us about "dereferencing type-punned
pointer will break strict-aliasing rules".

Genode > Allocator > *consumed*

                                                                 virtual const method
Return value

  size_t   Total amount of backing store consumed by the allocator

Genode > Allocator > *overhead*

                                                              pure virtual const method
Argument

  size   **size_t**

Return value

  size_t   Meta-data overhead per block

GENODE LABS

Genode › Allocator › **alloc**

Allocate block and signal error as an exception   method

Argument

size   **size_t**
        Block size to allocate

Exception

Out_of_memory

Return value

void ∗   Pointer to the new block

GENODE LABS

Genode ⟩ **Range_allocator**   class

Header

repos/base/include/base/allocator.h

Genode ⟩ Range_allocator ⟩ *add_range*

Add free address range to allocator   pure virtual method

Arguments

base   **addr_t**

size   **size_t**

Return value

int

Genode ⟩ Range_allocator ⟩ *remove_range*

Remove address range from allocator   pure virtual method

Arguments

base   **addr_t**

size   **size_t**

Return value

int

GENODE LABS
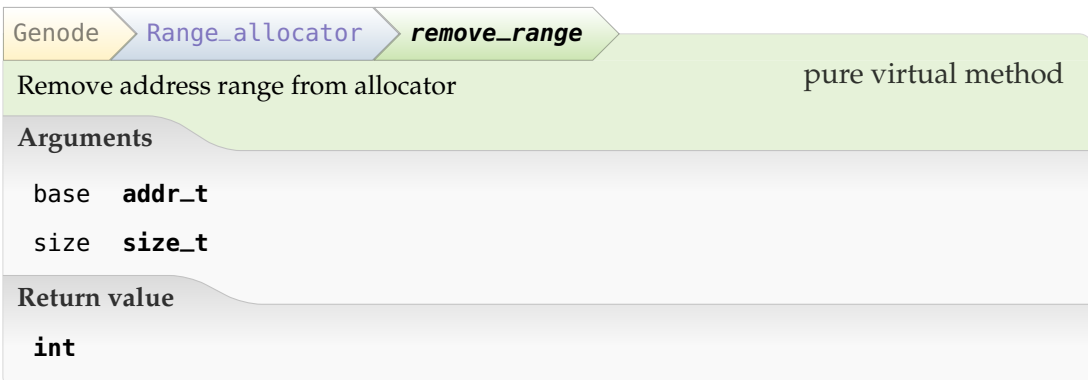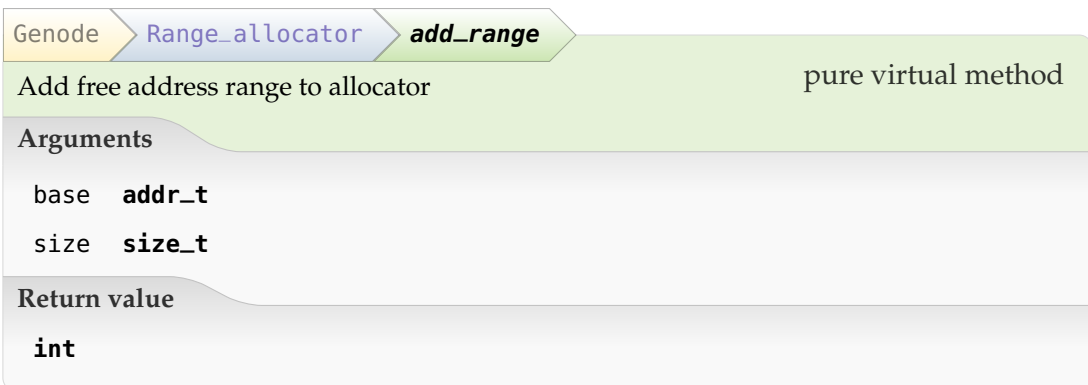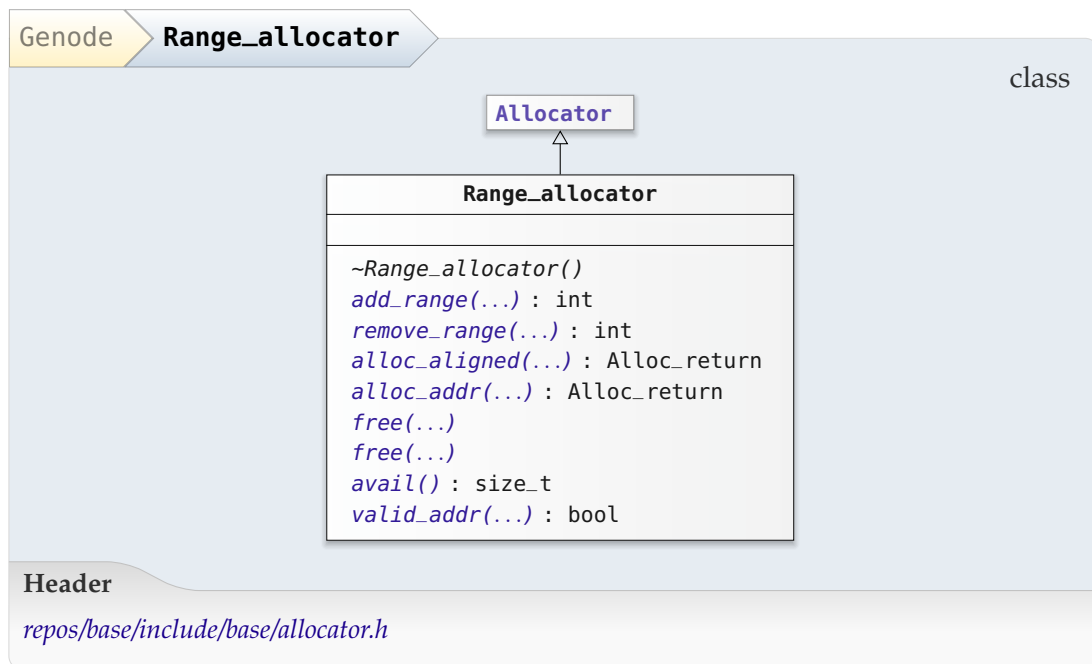
Genode	Range_allocator	***alloc_aligned***

Allocate block		pure virtual method

Arguments

size	**size_t**
	Size of new block
out_addr	**void ****
	Start address of new block, undefined in the error case
align	**int**
	Alignment of new block specified as the power of two
from	**addr_t**
	Default is 0
to	**addr_t**
	Default is ~0UL

Return value

Alloc_return

Genode	Range_allocator	***alloc_addr***

Allocate block at address		pure virtual method

Arguments

size	**size_t**
	Size of new block
addr	**addr_t**
	Desired address of block

Return value

Alloc_return	alloc_ok on success, or out_of_metadata if meta-data allocation failed, or range_conflict if specified range is occupied

GENODE LABS

Genode ⟩ Range_allocator ⟩ *free*

Free a previously allocated block   pure virtual method

Argument

  addr   **void ***

Details

NOTE: We have to declare the `Allocator::free(void *)` method here as well to make the compiler happy. Otherwise the C++ overload resolution would not find `Allocator::free(void *)`.

Genode ⟩ Range_allocator ⟩ *free*

                                                                     pure virtual method
Arguments

  addr   **void ***

  size   **size_t**

Genode ⟩ Range_allocator ⟩ *avail*

                                                               pure virtual const method
Return value

  size_t   The sum of available memory

Details

Note that the returned value is not neccessarily allocatable because the memory may be fragmented.

Genode ⟩ Range_allocator ⟩ *valid_addr*

Check if address is inside an allocated block   pure virtual const method

Argument

  addr   **addr_t**
         Address to check

Return value

  bool   True if address is inside an allocated block, false otherwise

GENODE LABS

8.10.1. Allocator guard

To protect the quantum of allocated memory from exceeding a predefined limit, an allocator guard can be used. It is typically used by server implementations that subdivide their heap into client-specific partitions dimensioned according to the session quota provided by the respective client.

Genode > **Allocator_guard**	
This class acts as guard for arbitrary allocators to limit memory exhaustion	class

```
                        Allocator
                            △
                            |
             ┌──────────────────────────────┐
             │       Allocator_guard         │
             ├──────────────────────────────┤
             ├──────────────────────────────┤
             │  Allocator_guard(...)         │
             │  upgrade(...)                 │
             │  withdraw(...) : bool         │
             │  alloc(...) : bool            │
             │  free(...)                    │
             │  consumed() : size_t          │
             │  quota() : size_t             │
             │  overhead(...) : size_t       │
             │  need_size_for_free() : bool  │
             └──────────────────────────────┘
```

Accessor

need_size_for_free   **bool**

Header

repos/base/include/base/allocator_guard.h

Genode > Allocator_guard > **Allocator_guard**	
	constructor

Arguments

allocator   **Allocator ***

amount   **size_t**

GENODE LABS

Genode ⟩ Allocator_guard ⟩ **upgrade**

method

Extend allocation limit

Argument

additional_amount   **size_t**

Genode ⟩ Allocator_guard ⟩ **withdraw**

method

Consume bytes without actually allocating them

Argument

size   **size_t**

Return value

bool

Genode ⟩ Allocator_guard ⟩ **alloc**

method

Allocate block

Arguments

size   **size_t**
           Block size to allocate

out_addr   **void ****
           Resulting pointer to the new block, undefined in the error case

Return value

bool   True on success

Genode ⟩ Allocator_guard ⟩ **free**

method

Free block a previously allocated block

Arguments

addr   **void ***

size   **size_t**

GENODE LABS

Genode	Allocator_guard	**consumed**	

const method

Return value

size_t   Amount of backing store consumed by the allocator

Genode	Allocator_guard	**quota**	

const method

Return value

size_t   Allocation limit

Genode	Allocator_guard	**overhead**	

const method

Argument

size   **size_t**

Return value

size_t   Meta-data overhead per block

GENODE LABS

8.10.2. Slab allocator

The Slab allocator is tailored for allocating small fixed-size memory blocks from a big chunk of memory. For the common use case of using a slab allocator for a certain type rather than for a known byte size, there exists a typed slab allocator as a front end of Slab.

Genode ⟩ **Slab**

Slab allocator   class

```
                                    ┌─────────────┐
                                    │  Allocator  │
                                    └─────────────┘
                                           △
                          ┌────────────────────────────────┐
                          │               Slab              │
                          ├────────────────────────────────┤
                          │                                 │
                          ├────────────────────────────────┤
                          │ Slab(...)                       │
                          │ ~Slab()                         │
                          │ insert_sb(...)                  │
                          │ any_used_elem() : void *        │
                          │ alloc(...) : bool               │
                          │ free(...)                       │
                          │ consumed() : size_t             │
                          │ overhead(...) : size_t          │
                          │ need_size_for_free() : bool     │
                          └────────────────────────────────┘
```

Accessors

consumed	**size_t**
need_size_for_free	**bool**

Header

repos/base/include/base/slab.h

GENODE ⩘ LABS
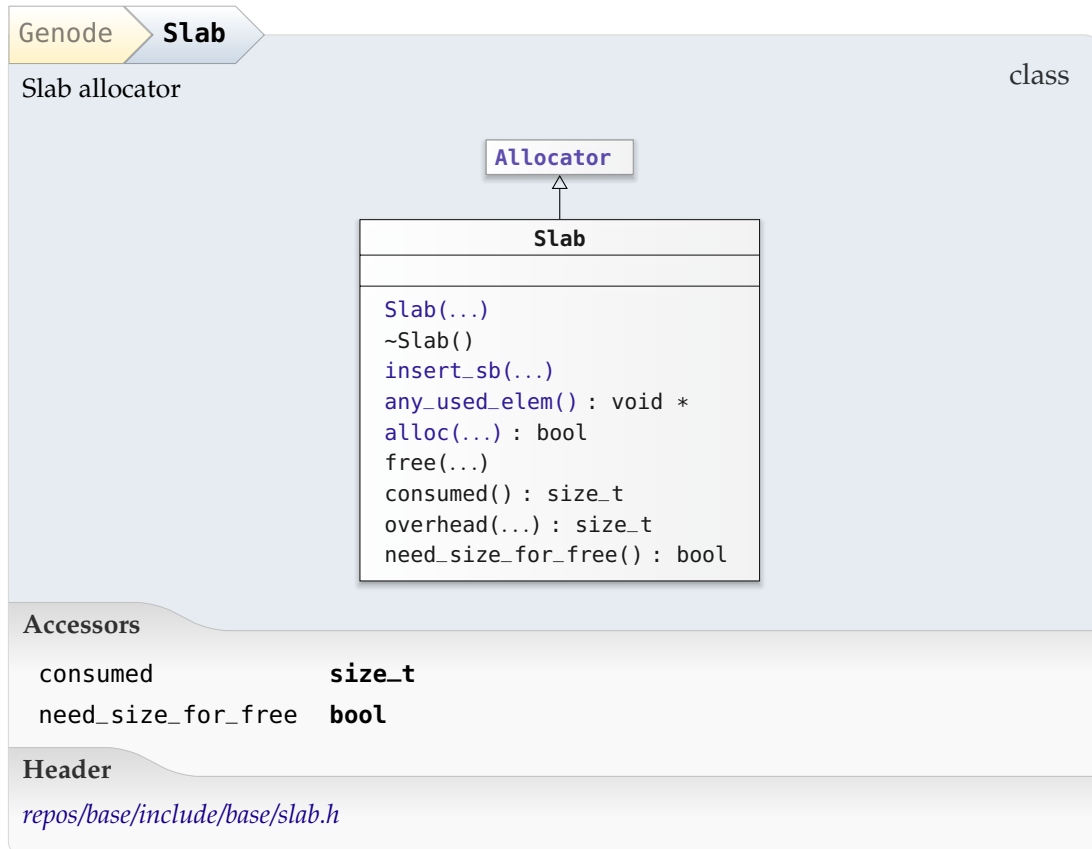
Genode › Slab › **Slab**

constructor

Arguments

slab_size	**size_t**
block_size	**size_t**
initial_sb	**void ***
backing_store	**Allocator ***
	Default is 0

Details

At construction time, there exists one initial slab block that is used for the first couple of allocations, especially for the allocation of the second slab block.

Genode › Slab › **insert_sb**

Add new slab block as backing store

method

Argument

ptr	**void ***

Details

The specified ptr has to point to a buffer with the size of one slab block.

Genode › Slab › **any_used_elem**

method

Return value

void * A used slab element, or nullptr if empty

GENODE LABS

<table>
<tr><td>Genode</td><td>Slab</td><td>**alloc**</td></tr>
</table>

Allocate slab entry
method

Arguments

size	**size_t**
addr	**void ∗∗**

Return value

bool

Details

The size parameter is ignored as only slab entries with preconfigured slab-entry size are allocated.

<table>
<tr><td>Genode</td><td>**Tslab**</td></tr>
</table>

class template

Template arguments

T	**typename**
BLOCK_SIZE	**size_t**

Header

repos/base/include/base/tslab.h

<table>
<tr><td>Genode</td><td>Tslab</td><td>**Tslab**</td></tr>
</table>

constructor

Arguments

backing_store	**Allocator ∗**
initial_sb	**void ∗** *Default is 0*

GENODE LABS

Genode › Tslab › **first_object**

method

Return value

  T *

8.10.3. AVL-tree-based best-fit allocator

In contrast to the rather limited slab allocators, `Allocator_avl` allows for arbitrary allocations from a list of address regions. It implements a best-fit allocation strategy, supports arbitrary alignments, and allocations at specified addresses.

Genode 〉 **Allocator_avl_base**

Interface of AVL-tree-based allocator   class

Range_allocator

△

Allocator_avl_base
_revert_allocations_and_ranges()
_find_by_address(...) : Block *
Allocator_avl_base(...)
~Allocator_avl_base()
any_block_addr(...) : bool
add_range(...) : int
remove_range(...) : int
alloc_aligned(...) : Alloc_return
alloc_addr(...) : Alloc_return
free(...)
avail() : size_t
valid_addr(...) : bool
alloc(...) : bool
free(...)
size_at(...) : size_t
overhead(...) : size_t
need_size_for_free() : bool

Accessors

avail	**size_t**
need_size_for_free	**bool**

Header

repos/base/include/base/allocator_avl.h

GENODE LABS

Genode ⟩ Allocator_avl_base ⟩ **_revert_allocations_and_ranges**

method

Clean up the allocator and detect dangling allocations

Details

This function is called at the destruction time of the allocator. It makes sure that the allocator instance releases all memory obtained from the meta-data allocator.

Genode ⟩ Allocator_avl_base ⟩ **_find_by_address**

const method

Find block by specified address

Arguments

addr	**addr_t**
size	**size_t** *Default is 0*
check_overlap	**bool** *Default is 0*

Return value

Block *

Genode ⟩ Allocator_avl_base ⟩ **Allocator_avl_base**

constructor

Arguments

md_alloc	**Allocator ***
md_entry_size	**size_t**

Details

This constructor can only be called from a derived class that provides an allocator for block meta-data entries. This way, we can attach custom information to block meta data.

GENODE LABS

Genode 〉 Allocator_avl_base 〉 **any_block_addr**

method

Argument

out_addr **addr_t ***
Result that contains address of block

Return value

bool True if block was found or false if there is no block available

Details

If no block was found, out_addr is set to zero.

Genode 〉 Allocator_avl_base 〉 **size_at**

const method

Argument

addr **void const ***

Return value

size_t Size of block at specified address

Genode 〉 Allocator_avl_base 〉 **overhead**

const method

Argument

- **size_t**

Return value

size_t The memory overhead per block

Details

The overhead is a rough estimation. If a block is somewhere in the middle of a free area, we could consider the meta data for the two free subareas when calculating the overhead. The sizeof(umword_t) represents the overhead of the meta-data slab allocator.

GENODE LABS

Genode **Allocator_avl_tpl**

class template

AVL-based allocator with custom meta data attached to each block.

Allocator_avl_base

△

Allocator_avl_tpl
Allocator_avl_tpl(...) ~Allocator_avl_tpl() slab_block_size() : size_t metadata(...) metadata(...) : BMDT* add_range(...) : int

BMDT,
SLAB_BLOCK_SIZE

Template arguments

BMDT **typename**
Block meta-data type

SLAB_BLOCK_SIZE **unsigned**

Header

repos/base/include/base/allocator_avl.h

Genode Allocator_avl_tpl **Allocator_avl_tpl**

constructor

Argument

metadata_chunk_alloc **Allocator ***
Pointer to allocator used to allocate meta-data blocks. if set to
0, use ourself for allocating our meta-data blocks. this works
only if the managed memory is completely accessible by the
allocator.

Genode Allocator_avl_tpl **slab_block_size**

class function

Return value

size_t Size of slab blocks used for meta data

GENODE LABS

Genode ▷ Allocator_avl_tpl ▷ **metadata**

Assign custom meta data to block at specified address

const method

Arguments	
addr	**void ***
bmd	**BMDT**

Genode ▷ Allocator_avl_tpl ▷ **metadata**

const method

Argument	
addr	**void ***

Return value

BMDT* Meta data that was attached to block at specified address

Genode ▷ Allocator_avl_tpl ▷ **add_range**

method

Arguments	
base	**addr_t**
size	**size_t**

Return value

int

GENODE LABS

8.10.4. Heap and sliced heap

Genode 〉 **Heap**

Heap that uses dataspaces as backing store

class

Allocator

```
                    Heap

Heap(...)
Heap(...)
~Heap()
quota_limit(...) : int
reassign_resources(...)
alloc(...) : bool
free(...)
consumed() : size_t
overhead(...) : size_t
need_size_for_free() : bool
```

Accessors

consumed	**size_t**
need_size_for_free	**bool**

Details

The heap class provides an allocator that uses a list of dataspaces of a RAM session as backing store. One dataspace may be used for holding multiple blocks.

Header

repos/base/include/base/heap.h

GENODE ⩘ LABS

Genode › Heap › **Heap**

constructor

Arguments

ram_session	**Ram_session** *
region_map	**Region_map** *
quota_limit	**size_t**
	Default is UNLIMITED
static_addr	**void** *
	Default is 0
static_size	**size_t**
	Default is 0

Genode › Heap › **Heap**

constructor

Arguments

ram	**Ram_session** &
rm	**Region_map** &

Genode › Heap › **quota_limit**

method

Reconfigure quota limit

Argument

new_quota_limit	**size_t**

Return value

int Negative error code if new quota limit is higher than currently used quota.

Genode › Heap › **reassign_resources**

method

Re-assign RAM and RM sessions

Arguments

ram	**Ram_session** *
rm	**Region_map** *

GENODE LABS

Genode **Sliced_heap**

Heap that allocates each block at a separate dataspace

class

Allocator

Sliced_heap
meta_data_size() : size_t
Sliced_heap(...)
~Sliced_heap()
alloc(...) : bool
free(...)
consumed() : size_t
overhead(...) : size_t
need_size_for_free() : bool

Accessors

consumed	**size_t**
need_size_for_free	**bool**

Header

repos/base/include/base/heap.h

Genode Sliced_heap **meta_data_size**

class function

Return value

size_t Size of header prepended to each allocated block in bytes

---

Genode Sliced_heap **Sliced_heap**

constructor

Arguments

ram_session	**Ram_session &**
region_map	**Region_map &**

GENODE LABS

Genode ⟩ Sliced_heap ⟩ **alloc**

method

Arguments

- **size_t**

- **void ****

Return value

bool

Genode ⟩ Sliced_heap ⟩ **free**

method

Arguments

- **void ***

- **size_t**

Genode ⟩ Sliced_heap ⟩ **overhead**

const method

Argument

size   **size_t**

Return value

size_t

GENODE LABS

8.10.5. Bit allocator

Genode	**Bit_allocator**	
		class template

```
                              BITS
            Bit_allocator

            Bit_allocator()
            alloc(...) : addr_t
            free(...)
```

Template argument

  BITS   **unsigned**

Header

repos/base/include/util/bit_allocator.h

Genode	Bit_allocator	**alloc**
		method

Argument

  num_log2   **size_t const**
              Default is 0

Return value

  addr_t

Genode	Bit_allocator	**free**
		method

Arguments

  bit_start   **addr_t const**

  num_log2   **size_t const**
               Default is 0

GENODE LABS

8.11. String processing

8.11.1. Basic string operations

There exists a small set of string-manipulation operations as global functions in the Genode namespace.

GENODE LABS

Genode

String utilities

namespace

Functions

- strlen(...) : size_t
- strcmp(...) : int
- memmove(...) : void *
- memcpy(...) : void *
- strncpy(...) : char *
- memcmp(...) : int
- memset(...) : void *
- digit(...) : int
- is_letter(...) : bool
- is_digit(...) : bool
- is_whitespace(...) : bool
- ascii_to_unsigned(...) : size_t
- ascii_to(...) : size_t
- ascii_to(...) : size_t
- ascii_to(...) : size_t
- ascii_to(...) : size_t
- ascii_to(...) : size_t
- ascii_to(...) : size_t
- ascii_to(...) : size_t
- end_of_quote(...) : bool
- unpack_string(...) : int

Header

repos/base/include/util/string.h

Genode **strlen**

global function

Argument

s **const char ***

Return value

size_t   Length of null-terminated string in bytes

GENODE LABS

Genode › **strcmp**

Compare two strings

global function

Arguments

s1   **const char ***

s2   **const char ***

len   **size_t**
      Maximum number of characters to compare, default is unlimited
      Default is ~0UL

Return value

int   0 if both strings are equal, or a positive number if s1 is higher than s2, or a negative
           number if s1 is lower than s2

Genode › **memmove**

Copy memory buffer to a potentially overlapping destination
buffer

global function

Arguments

dst   **void ***
       Destination memory block

src   **const void ***
       Source memory block

size   **size_t**
       Number of bytes to move

Return value

void *   Pointer to destination memory block

GENODE LABS

Genode **memcpy**

Copy memory buffer to a non-overlapping destination buffer   global function

Arguments

dst   **void ***
       Destination memory block

src   **const void ***
       Source memory block

size   **size_t**
       Number of bytes to copy

Return value

void *   Pointer to destination memory block

Genode **strncpy**

Copy string   global function

Arguments

dst   **char ***
       Destination buffer

src   **const char ***
       Buffer holding the null-terminated source string

size   **size_t**
       Maximum number of characters to copy

Return value

char *   Pointer to destination string

Details

Note that this function is not fully compatible to the C standard, in particular there is no zero-padding if the length of src is smaller than size. Furthermore, in contrast to the libc version, this function always produces a null-terminated string in the dst buffer if the size argument is greater than 0.

GENODE ⤳ LABS

Genode > **memcmp**

Compare memory blocks

global function

Arguments

p0   **const void ***

p1   **const void ***

size **size_t**

Return value

int 0 if both memory blocks are equal, or a negative number if p0 is less than p1, or a positive number if p0 is greater than p1

Genode > **memset**

Fill destination buffer with given value

global function

Arguments

dst   **void ***
      Destination buffer

i   **int**
      Byte value

size **size_t**
      Buffer size in bytes

Return value

void *

Genode > **digit**

Convert ASCII character to digit

global function

Arguments

c   **char**

hex   **bool**
      Consider hexadecimals
      Default is `false`

Return value

int Digit or -1 on error

GENODE LABS

Genode   **is_letter**

global function

Argument

  c   **char**

Return value

  bool   True if character is a letter

Genode   **is_digit**

global function

Arguments

  c   **char**

  hex   **bool**
         Default is false

Return value

  bool   True if character is a digit

Genode   **is_whitespace**

global function

Argument

  c   **char**

Return value

  bool   True if character is whitespace

GENODE 🖎 LABS

Genode › **ascii_to_unsigned**

Read unsigned long value from string

function template

Template argument

T **typename**

Arguments

s   **const char ***
         Source string

result **T &**
         Destination variable

base   **unsigned**
         Integer base

Return value

size_t Number of consumed characters

Details

If the base argument is 0, the integer base is detected based on the characters in front of the number. If the number is prefixed with "0x", a base of 16 is used, otherwise a base of 10.

Genode › **ascii_to**

Read boolean value from string

global function

Arguments

s   **char const ***

result **bool &**

Return value

size_t Number of consumed characters

GENODE LABS

Genode ❭ **ascii_to**

Read unsigned long value from string   global function

Arguments

  s   **const char ***

  result **unsigned long &**

Return value

  size_t Number of consumed characters

Genode ❭ **ascii_to**

Read unsigned long long value from string   global function

Arguments

  s   **const char ***

  result **unsigned long**

Return value

  size_t Number of consumed characters

Genode ❭ **ascii_to**

Read unsigned int value from string   global function

Arguments

  s   **const char ***

  result **unsigned int &**

Return value

  size_t Number of consumed characters

GENODE LABS

Genode > **ascii_to**

Read signed long value from string

global function

Arguments

s	**const char ***
result	**long &**

Return value

size_t Number of consumed characters

Genode > **ascii_to**

Read **Number_of_bytes** value from string and handle the size suffixes

global function

Arguments

s	**const char ***
result	**Number_of_bytes &**

Return value

size_t Number of consumed characters

Details

This function scales the resulting size value according to the suffixes for G (2^30), M (2^20), and K (2^10) if present.

Genode > **ascii_to**

Read double float value from string

global function

Arguments

s	**const char ***
result	**double &**

Return value

size_t Number of consumed characters

GENODE LABS

Genode **end_of_quote**

Check for end of quotation

global function

Argument

s **const char ***

Return value

bool

Details

Checks if next character is non-backslashed quotation mark.

Genode **unpack_string**

Unpack quoted string

global function

Arguments

src   **const char ***
Source string including the quotation marks ("...")

dst   **char ***
Destination buffer

dst_len **int**

Return value

int Number of characters or negative error code

GENODE LABS

To cover the common case of embedding a string buffer as a member variable in a class, there exists the `String` class template, which alleviates the need for C-style arrays in such situations.

Genode · **String**   class template

Buffer that contains a null-terminated string

```
                              ┌ ─ ─ ─ ─ ─ ┐
                              │ CAPACITY  │
          ┌──────────────────└───────────┘──┐
          │            String                │
          ├──────────────────────────────────┤
          ├──────────────────────────────────┤
          │ size() : size_t                  │
          │ String()                         │
          │ String(...)                      │
          │ length() : size_t                │
          │ capacity() : size_t              │
          │ valid() : bool                   │
          │ string() : char const *          │
          │ operator ==(...) : bool          │
          │ operator !=(...) : bool          │
          │ operator ==(...) : bool          │
          │ operator !=(...) : bool          │
          └──────────────────────────────────┘
```

Template argument

CAPACITY **size_t**
          Buffer size including the terminating zero

Accessors

valid   **bool**

string   **char const ***

Header

repos/base/include/util/string.h

Genode · String · **size**   class function

Return value

size_t

GENODE LABS

Genode　String　**String**

constructor

Arguments

str　**char const ***

len　**size_t**
　　Default is ~0UL - 1

Genode　String　**length**

const method

Return value

size_t　Length of string, including the terminating null character

Genode　String　**capacity**

class function

Return value

size_t

Genode　String　**operator ==**

const method

Argument

other　**char const ***

Return value

bool

Genode　String　**operator !=**

const method

Argument

other　**char const ***

Return value

bool

GENODE LABS

Genode String **operator ==**

const method template

Template argument

OTHER_CAPACITY **size_t**

Argument

other **String**<**OTHER_CAPACITY**> **const &**

Return value

bool

Genode String **operator !=**

const method template

Template argument

OTHER_CAPACITY **size_t**

Argument

other **String**<**OTHER_CAPACITY**> **const &**

Return value

bool

GENODE LABS
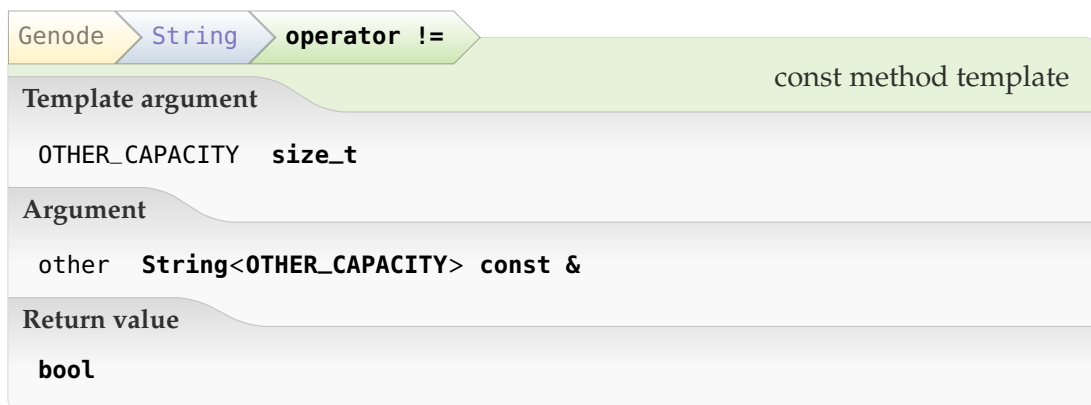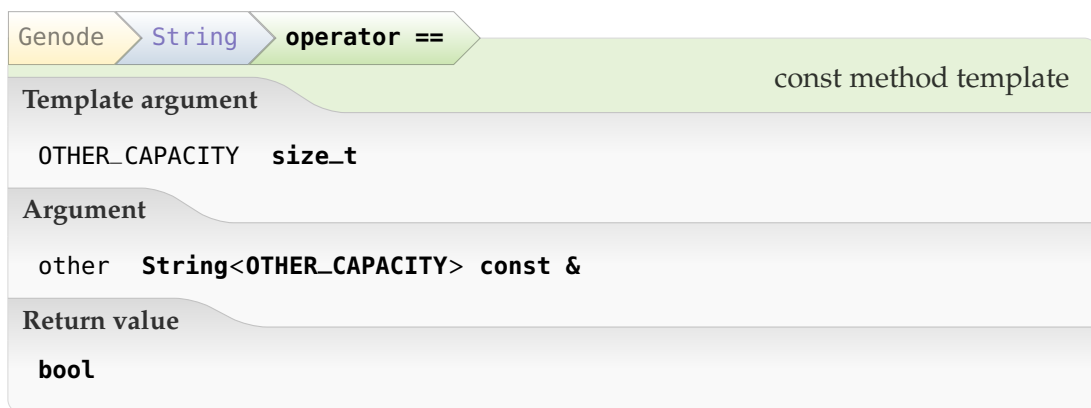
8.11.2. Tokenizing

For parsing structured text such as argument strings or XML, simple tokenizing support is provided via the Token class.

Genode	**Token**

Token   class template

```
                        ┌─────────────────────┐ ┌ ─ ─ ─ ─ ─ ─ ─ ─ ─ ┐
                        │       Token         │  SCANNER_POLICY
                        ├─────────────────────┤ └ ─ ─ ─ ─ ─ ─ ─ ─ ─ ┘
                        │                     │
                        ├─────────────────────┤
                        │ Token(...)          │
                        │ start() : char *    │
                        │ len() : size_t      │
                        │ type() : Type       │
                        │ string(...)         │
                        └─────────────────────┘
```

Template argument

SCANNER_POLICY	**typename**
	Policy that defines the way of token scanning

Accessors

len	**size_t**
type	**Type**

Details

This class is used to group characters of a string which belong to one syntactical token types number, identifier, string, whitespace or another single character.
See Scanner_policy_identifier_with_underline for an example scanner policy.

Header

repos/base/include/util/token.h

406

GENODE 𝌆 LABS

Genode > Token > **Token**

constructor

Arguments

s   **const char ***
         Start of string to construct a token from
         Default is 0

max_len   **size_t**
         Maximum token length
         Default is ~0UL

Details

The max_len argument is useful for processing character arrays that are not null-terminated.

Genode > Token > **start**

const method

Accessors

Return value

char *

Genode > Token > **string**

const method

Arguments

dst   **char ***

max_len   **size_t**

GENODE LABS

8.11.3. Diagnostic output

To enable components to produce diagnostic output like errors, warnings, and log messages, Genode offers a simple `Output` interface for sequentially writing single characters or character sequences.

Genode > **Output**

Interface for textual output   class

Output
out_char(...)
out_string(...)

Header

repos/base/include/base/output.h

Genode > Output > *out_char*

Output single character   pure virtual method

Argument

- **char**

Genode > Output > *out_string*

Output string   virtual method

Arguments

str   **char const ***

n   **size_t**
      Maximum number of characters to output
      Default is ~0UL

Details

The output stops on the first occurrence of a null character in the string or after n characters. The default implementation uses `out_char`. This method may be overridden by the backend for improving efficiency.

Functions for generating output for different types are named `print` and take an `Output &` as first argument. The second argument is a `const &` to the value to print. Overloads of the `print` function for commonly used basic types are provided. Fur-

GENODE LABS

thermore, there is a function template that is used if none of the type-specific over-loads match. This function template expects the argument to be an object with a `print` method. In contrast to a plain `print` function overload, such a method is able to incor-porate private state into the output.

Genode
                                                                   namespace

Interface for textual output

Functions

- `print(...)`
- `print(...)`
- `print(...)`
- `print(...)`
- `print(...)`
- `print(...)`
- `print(...)`
- `print(...)`
- `print(...)`
- `print(...)`

Header

repos/base/include/base/output.h

Genode > **print**
                                                      global function

Print null-terminated string

Arguments

output	**Output &**
-	**char const ***

Genode > **print**
                                                      global function

Print pointer value

Arguments

output	**Output &**
-	**void const ***

GENODE LABS

Genode ⟩ **print**

Print arbitrary pointer types

function template

Template argument

T **typename**

Arguments

output **Output &**

ptr **T ***

Details

This function template takes precedence over the one that takes a constant object reference as argument.

Genode ⟩ **print**

Print unsigned long value

global function

Arguments

output **Output &**

long **unsigned**

Genode ⟩ **print**

Print unsigned integer value

global function

Arguments

output **Output &**

value **unsigned**

Genode ⟩ **print**

Print signed long value

global function

Arguments

output **Output &**

- **long**

GENODE ⬈ LABS

Genode > **print**

Print signed integer value

global function

Arguments

output	**Output &**
value	**int**

Genode > **print**

Print signed long long value

global function

Arguments

output	**Output &**
long	**long**

Genode > **print**

Print hexadecimal number

global function

Arguments

output	**Output &**
-	**Hex const &**

Genode > **print**

Print information about object `obj`

function template

Template argument

T	**typename**

Arguments

output	**Output &**
obj	**T const &**

Details

The object type must provide a const `print(Output &)` method that produces the textual representation of the object.

In contrast to overloads of the `Genode::print` function, the `T::print` method is able to access object-internal state, which can thereby be incorporated into the textual output.

411

GENODE LABS

The component's execution environment provides an implementation of the `Output` interface that targets a LOG session. This output back end is offered to the component in the form of the `log`, `warning`, and `error` functions that accept an arbitrary number of arguments that are printed in a concatenated fashion. Each messages is implicitly finalized with a newline character.

Genode   namespace

LOG output functions

Functions

- `log(...)`
- `warning(...)`
- `error(...)`

Header

repos/base/include/base/log.h

Genode   **log**   function template

Write `args` as a regular message to the log

Template argument

  ARGS   **typename...**

Argument

  args   **ARGS &&...**

Genode   **warning**   function template

Write `args` as a warning message to the log

Template argument

  ARGS   **typename...**

Argument

  args   **ARGS &&...**

GENODE LABS

Genode › **error**

Write `args` as an error message to the log   function template

Template argument

 ARGS   **typename...**

Argument

 args   **ARGS &&...**

GENODE LABS

8.12. Multi-threading and synchronization

8.12.1. Threads

A thread is created by constructing an object of a class inherited from `Thread`. The new thread starts its execution at the `entry` member function. Thereby, each thread runs in the context of its object and can access context-specific information by accessing its member variables. This largely alleviates the need for a thread-local storage (TLS) mechanism. Threads use a statically allocated stack, which is dimensioned according to the corresponding constructor argument.

GENODE LABS

Genode **Thread**

Concurrent flow of control

class

Thread
Thread(...)
Thread(...)
~Thread()
entry()
start()
name() : Name
alloc_secondary_stack(...) : void*
free_secondary_stack(...)
cap() : Thread_capability
cancel_blocking()
native_thread() : Native_thread &
stack_top() : void *
stack_base() : void *
stack_virtual_size() : size_t
stack_area_virtual_base() : addr_t
stack_area_virtual_size() : size_t
myself() : Thread *
stack_size(...)
utcb() : Native_utcb *
join()
trace(...)
trace(...)
trace(...)

Details

A `Thread` object corresponds to a physical thread. The execution starts at the `entry()` method as soon as `start()` is called.

Header

repos/base/include/base/thread.h

GENODE LABS

Genode ⟩ Thread ⟩ **Thread**

constructor

Arguments

env	**Env &**
	Component environment
name	**Name const &**
	Thread name, used for debugging
stack_size	**size_t**
	Stack size
location	**Location**
	Cpu affinity relative to the cpu-session's affinity space
weight	**Weight**
	Scheduling weight relative to the other threads sharing the same cpu session
cpu	**Cpu_session &**

Exceptions

Stack_too_large
Stack_alloc_failed
Out_of_stack_space

Details

The env argument is needed because the thread creation procedure needs to interact with the environment for attaching the thread's stack, the trace-control dataspace, and the thread's trace buffer and policy.

Genode ⟩ Thread ⟩ **Thread**

constructor

Arguments

env	**Env &**
name	**Name const &**
stack_size	**size_t**

Details

This is a shortcut for the common case of creating a thread via the environment's CPU session, at the default affinity location, and with the default weight.

GENODE ⟋ LABS

Genode	Thread	*entry*	

Entry method of the thread

pure virtual method

Genode	Thread	*start*	

Start execution of the thread

virtual method

Details

This method is virtual to enable the customization of threads used as server activation.

Genode	Thread	**name**	

Request name of thread

const method

Return value

  Name

Genode	Thread	**alloc_secondary_stack**	

Add an additional stack to the thread

method

Arguments

  name   **char const ***

  stack_size **size_t**

Exceptions

  Stack_too_large
  Stack_alloc_failed
  Out_of_stack_space

Return value

  void*   Pointer to the new stack's top

Details

The stack for the new thread will be allocated from the RAM session of the component environment. A small portion of the stack size is internally used by the framework for storing thread-specific information such as the thread's name.

GENODE LABS

Genode 〉 Thread 〉 **free_secondary_stack**

Remove a secondary stack from the thread

method

Argument

stack_addr   **void**∗

Genode 〉 Thread 〉 **cap**

Request capability of thread

const method

Return value

Thread_capability

Genode 〉 Thread 〉 **cancel_blocking**

Cancel currently blocking operation

method

Genode 〉 Thread 〉 **native_thread**

method

Return value

Native_thread &   Kernel-specific thread meta data

Genode 〉 Thread 〉 **stack_top**

const method

Return value

void ∗   Pointer just after first stack element

Genode 〉 Thread 〉 **stack_base**

const method

Return value

void ∗   Pointer to last stack element

Genode 〉 Thread 〉 **stack_virtual_size**

class function

Return value

size_t   Virtual size reserved for each stack within the stack area

GENODE LABS

Genode ⟩ Thread ⟩ **stack_area_virtual_base**

class function

Return value

addr_t   The local base address of the stack area

Genode ⟩ Thread ⟩ **stack_area_virtual_size**

class function

Return value

size_t   Total size of the stack area

Genode ⟩ Thread ⟩ **myself**

class function

Return value

Thread *   Pointer to caller's `thread` object

Genode ⟩ Thread ⟩ **stack_size**

Ensure that the stack has a given size at the minimum

method

Argument

size   **size_t const**
          Minimum stack size

Exceptions

Stack_too_large
Stack_alloc_failed

Genode ⟩ Thread ⟩ **utcb**

method

Return value

Native_utcb *   User-level thread control block

Details

Note that it is safe to call this method on the result of the `myself` class function. It handles the special case of `myself` being 0 when called by the main thread during the component initialization phase.

GENODE ⬙ LABS

Genode > Thread > **join**

Block until the thread leaves the `entry` method

method

Details

Join must not be called more than once. Subsequent calls have undefined behaviour.

Genode > Thread > **trace**

Log null-terminated string as trace event

class function

Argument

cstring   **char const ***

Genode > Thread > **trace**

Log binary data as trace event

class function

Arguments

data   **char const ***

len   **size_t**

Genode > Thread > **trace**

Log trace event as defined in base/trace.h

class function template

Template argument

EVENT   **typename**

Argument

event   **EVENT const ***

GENODE ⋙ LABS

8.12.2. Locks and semaphores

For mutual exclusive execution of critical sections, there exists a simple lock interface providing lock and unlock semantics. The lock comes in two flavours. Cancelable locks can be unblocked by force via core's cancel-blocking mechanism. In contrast, a non-cancelable lock (`Lock`) does not reflect the cancellation of its blocking operation at the API level but transparently re-enters its blocking state after a cancellation.

Genode **Cancelable_lock** class

Basic locking primitive

```
        Cancelable_lock

  Cancelable_lock(...)
  lock()
  unlock()
```

Header

repos/base/include/base/cancelable_lock.h

Genode Cancelable_lock **Cancelable_lock** constructor

Argument

initial **State**
  Default is UNLOCKED

Genode Cancelable_lock **lock** method

Try to aquire the lock and block while the lock is not free

Exception

Genode::Blocking_canceled

Genode Cancelable_lock **unlock** method

Release lock

GENODE LABS

Genode	**Lock**		class

Locking primitives

Cancelable_lock

Lock

Lock(...)
lock()

Header

repos/base/include/base/lock.h

Genode	Lock	**Lock**	constructor

Argument

initial   **State**
          Default is UNLOCKED

Genode	Lock	**lock**	method

For the use case of using locks for protecting critical sections, the `Lock_guard` provides a convenient mechanism for the automated unlocking of a lock when leaving a variable scope.

GENODE LABS

Genode ▸ **Lock_guard**

Lock guard template

class template

```
                              ┌─ ─ ┐
                              │ LT │
        ┌─────────────────────└────┘─┐
        │       Lock_guard           │
        ├────────────────────────────┤
        │                            │
        ├────────────────────────────┤
        │  Lock_guard(...)           │
        │  ~Lock_guard()             │
        └────────────────────────────┘
```

Template argument

LT **typename**
   Lock type

Details

A lock guard is instantiated as a local variable. When a lock guard is constructed, it acquires the lock that is specified as constructor argument. When the control flow leaves the scope of the lock-guard variable via a return statement or an exception, the lock guard's destructor gets called, freeing the lock.

Header

repos/base/include/base/lock_guard.h

Genode ▸ Lock_guard ▸ **Lock_guard**

constructor

Argument

lock **LT &**

GENODE LABS

Alongside lock-based mutual exclusion of entering critical sections, organizing threads in a producer-consumer relationship is a common design pattern for thread synchronization. The `Semaphore` interface enables the implementation of this synchronization scheme.

Genode	**Semaphore**	
		class

Semaphore

```
          Semaphore
─────────────────────────
  Semaphore(...)
  ~Semaphore()
  up()
  down()
  cnt() : int
```

Header

repos/base/include/base/semaphore.h

Genode	Semaphore	**Semaphore**
		constructor

Argument

n **int**
  Initial counter value of the semphore
  Default is 0

Genode	Semaphore	**up**
		method

Increment semphore counter

Details

This method may wake up another thread that currently blocks on a `down` call at the same semaphore.

Genode	Semaphore	**down**
		method

Decrement semaphore counter, block if the counter reaches zero

GENODE LABS

Genode	Semaphore	**cnt**	method

Return value

   int   Current semaphore counter

To synchronize method calls of an object, the `Synced_interface` can be used to equip the class of the called object with thread safety.

Genode	**Synced_interface**		class template

```
                              Synced_interface        IF, LOCK

                           Synced_interface(...)
                           operator ()() : Guard
                           operator ()() : Guard
```

Template arguments

IF	**typename**
LOCK	**typename**

Accessor

operator ()	**Guard**

Header

repos/base/include/base/synced_interface.h

Genode	Synced_interface	**Synced_interface**	constructor

Arguments

lock	**LOCK &**
interface	**IF ***

Genode	Synced_interface	**operator ()**	method

Return value

Guard

GENODE LABS

8.13. Signalling

Section 3.6.2 provides the high-level description of the mechanism for the delivery of asynchronous notifications (signals). The API defines interfaces for signal transmitters and for the association of signal handlers with entrypoints. An entrypoint can be associated with many signal handlers where each handler usually corresponds to a different signal source. Each signal handler is addressable via a distinct capability. Those so-called signal-context capabilities can be delegated across component boundaries in the same way as RPC-object capabilities. If a component is in possession of a signal-context capability, it can trigger the corresponding signal handler by using a so-called signal transmitter. The signal transmitter provides fire-and-forget semantics for signal submission. Signals serve as mere notifications and cannot carry any payload.

Genode ⟩ **Signal_transmitter**

Signal transmitter   class

Signal_transmitter
Signal_transmitter(...)
context(...)
context() : Signal_context_capability
submit(...)

Details

Each signal-transmitter instance acts on behalf the context specified as constructor argument. Therefore, the resources needed for the transmitter such as the consumed memory `sizeof(Signal_transmitter)` should be accounted to the owner of the context.

Header

repos/base/include/base/signal.h

Genode ⟩ Signal_transmitter ⟩ **Signal_transmitter**

                                                                                  constructor
Argument

context **Signal_context_capability**
         Capability to signal context that is going to receive signals produced by the
         transmitter
         Default is `Signal_context_capability()`

GENODE LABS

Genode ⟩ Signal_transmitter ⟩ **context**

Set signal context   method

Argument

  context   **Signal_context_capability**

Genode ⟩ Signal_transmitter ⟩ **context**

                                                    method

Return value

  Signal_context_capability   Signal context

Genode ⟩ Signal_transmitter ⟩ **submit**

Trigger signal submission to context   method

Argument

  cnt   **unsigned**
       Number of signals to submit at once
       Default is **1**

GENODE LABS

Genode **Signal_handler**

Signal dispatcher for handling signals by an object method   class template

Signal_dispatcher_base   Signal_context_capability

△   △

T, -

Signal_handler

Signal_handler(...)
~Signal_handler()
dispatch(...)

Template arguments

T   **typename**
     Type of signal-handling class

-    **Entrypoint**

Details

This utility associates an object method with signals. It is intended to be used as a member
variable of the class that handles incoming signals of a certain type. The constructor takes a
pointer-to-member to the signal-handling method as argument.

Header

repos/base/include/base/signal.h

Genode Signal_handler **Signal_handler**

constructor

Arguments

ep   **EP &**
     Entrypoint managing this signal rpc

obj **T &**

Genode Signal_handler **dispatch**

method

Interface of Signal_dispatcher_base

Argument

num **unsigned**

GENODE LABS

A `Signal_handler` object is meant to be hosted as a member of the class that also contains a member function to be executed upon the arrival of a signal. Its constructor takes the entrypoint, the signal-handling object, and a pointer to the handling function as arguments. The following example illustrates the common pattern of using a `Signal_handler`.

```
class Main
{
  ...
  Entrypoint &_ep;
  ...
  void _handle_config();

  Signal_handler<Main> _config_handler =
    { _ep, *this, &Main::_handle_config };
  ...
};
```

In the example above, the `_config_handler` creates a signal-context capability for the `_handle_config` method. In fact, the `_config_handler` is a capability since the `Signal_handler` is derived from `Signal_context_capability`. Therefore, the `_config_handler` can be directly passed as argument to an RPC call for registering a signal handler at a server.

GENODE ⬩ LABS

8.14. Remote procedure calls

Section 3.6.1 provides the high-level description of synchronous remote procedure calls (RPC).

8.14.1. RPC mechanism

The RPC mechanism consists of the following header files:

base/rpc.h Contains the basic type definitions and utility macros to declare RPC interfaces.

base/rpc_args.h Contains definitions of non-trivial argument types used for transferring strings and binary buffers. Its use by RPC interfaces is optional.

base/rpc_server.h Contains the interfaces of the server-side RPC API. In particular, this part of the API consists of the `Rpc_object` class template.

base/rpc_client.h Contains the API support for invoking RPC functions. It is complemented by the definitions in *base/capability.h*. The most important elements of the client-side RPC API are the `Capability` class template and `Rpc_client`, which is a convenience wrapper around `Capability`.

Each RPC interface is an abstract C++ interface, supplemented by a few annotations. For example:

```
#include <session/session.h>
#include <base/rpc.h>

namespace Hello { struct Session; }

struct Hello::Session : Genode::Session
{
  static const char *service_name() { return "Hello"; }

  virtual void say_hello() = 0;
  virtual int add(int a, int b) = 0;

  GENODE_RPC(Rpc_say_hello, void, say_hello);
  GENODE_RPC(Rpc_add, int, add, int, int);
  GENODE_RPC_INTERFACE(Rpc_say_hello, Rpc_add);
};
```

The macros `GENODE_RPC` and `GENODE_RPC_INTERFACE` are defined in *base/rpc.h* and enrich the interface with type information. They are only used at compile time and

GENODE ⊿ LABS

have no effect on the run time or the size of the class. Each RPC function is represented as a type. In the example, the type meta data of the `say_hello` function is attached to the `Rpc_say_hello` type within the scope of `Session`. The macro arguments are:

```
GENODE_RPC(func_type, ret_type, func_name, arg_type ...)
```

The `func_type` argument is an arbitrary type name (except for the type name `Rpc_functions`) used to refer to the RPC function, `ret_type` is the return type or `void`, `func_name` is the name of the server-side function that implements the RPC function, and the list of `arg_type` arguments comprises the RPC function argument types. The `GENODE_RPC_INTERFACE` macro defines a type called `Rpc_functions` that contains the list of the RPC functions provided by the RPC interface.

Server side The implementation of the RPC interface inherits the `Rpc_object` class template with the interface type as argument and implements the abstract RPC interface class.

`dispatch` method of the `Rpc_object` class template, according to the type information found in the annotations of the `Session` interface.

GENODE ⩘ LABS

Genode ⟩ **Rpc_object_base**

class

Object_pool <Rpc_object_base>

Rpc_object_base
~Rpc_object_base() *dispatch(...)* : Rpc_exception_code

Header

repos/base/include/base/rpc_server.h

Genode ⟩ Rpc_object_base ⟩ *dispatch*

Interface to be implemented by a derived class

pure virtual method

Arguments

op   **Rpc_opcode**
      Opcode of invoked method

in   **Ipc_unmarshaller &**
      Incoming message with method arguments

out   **Msgbuf_base &**
      Outgoing message for storing method results

Return value

Rpc_exception_code

To make an instance of an RPC object invokable via RPC, it must be associated with an RPC entrypoint. By passing an RPC object to the `Entrypoint::manage` method, a new capability for the RPC object is created. The capability is tagged with the type of the RPC object.

Client side   At the client side, a capability can be invoked by the capability's `call` method template with the RPC functions type as template argument. The method arguments correspond to the RPC function arguments.

By convention, the `Capability::call` method is rarely used directly. Instead, each RPC interface is accompanied with a client-side implementation of the abstract interface where each RPC function is implemented by calling the `Capability::call` method. This convention is facilitated by the `Rpc_client` class template.

GENODE LABS

Genode ❭ **Rpc_client**

RPC client   class template

```
        Capability <RPC_INTERFACE>      RPC_INTERFACE
                         △                    △
                         |        |           ┆
                   ┌─────┴────────┴──┐  ┌ ─ ─ ┴ ─ ─ ─ ─ ┐
                   │  Rpc_client     │    RPC_INTERFACE
                   ├─────────────────┤  └ ─ ─ ─ ─ ─ ─ ─ ┘
                   │  Rpc_client(...)│
                   └─────────────────┘
```

Template argument

RPC_INTERFACE   **typename**

Details

This class template is the base class of the client-side implementation of the specified
RPC_INTERFACE. Usually, it inherits the pure virtual functions declared in RPC_INTERFACE
and has the built-in facility to perform RPC calls to this particular interface. Hence, the
client-side implementation of each pure virtual interface function comes down to a simple
wrapper in the line of return call<Rpc_function>(arguments...).

Header

repos/base/include/base/rpc_client.h

Genode ❭ Rpc_client ❭ **Rpc_client**

                                                              constructor
Argument

cap   **Capability**<**RPC_INTERFACE**> const &

Using this template, the client-side implementation of the example RPC interface
looks as follows.

434

GENODE ⋛ LABS

```
#include <hello_session/hello_session.h>
#include <base/rpc_client.h>
#include <base/printf.h>

namespace Hello {

  struct Session_client : Genode::Rpc_client<Session>
  {
    Session_client(Genode::Capability<Session> cap)
    : Genode::Rpc_client<Session>(cap) { }

    void say_hello()
    {
      call<Rpc_say_hello>();
    }

    int add(int a, int b)
    {
      return call<Rpc_add>(a, b);
    }
  };
}
```

For passing RPC arguments and results, the regular C++ type-conversion rules are in effect. If there is no valid type conversion, or if the number of arguments is wrong, or if the RPC type annotations are not consistent with the abstract interface, the error is detected at compile time.

8.14.2. Transferable argument types

The arguments specified to GENODE_RPC behave mostly as expected for a normal function call. But there are some notable differences:

Value types   Value types are supported for basic types and plain-old-data types (self-sufficient structs or classes). The object data is transferred as such. If the type is not self-sufficient (it contains pointers or references), the pointers and references are transferred as plain data, most likely pointing to the wrong thing in the callee's address space.

Const references   Const references behave like value types. The referenced object is transferred to the server and a reference to the server-local copy is passed to the server-side function. Note that in contrast to a normal function call that takes a reference argument, the size of the referenced object is accounted for allocating the message buffer on the client side.

GENODE ⫸ LABS

Non-const references   Non-const references are handled similar to const references. In addition the server-local copy gets transferred back to the caller so that server-side modifications of the object become visible to the client.

Capabilities   Capabilities can be transfered as values, const references, or non-const references.

Variable-length buffers   There exists special support for passing binary buffers to RPC functions using the `Rpc_in_buffer` class template provided by *base/rpc_args.h*. The maximum size of the buffer must be specified as template argument. An `Rpc_in_buffer` object does not contain a copy of the data passed to the constructor, only a pointer to the data. In contrast to a fixed-sized object containing a copy of the payload, the RPC framework does not transfer the whole object but only the actually used payload.

Pointers   Pointers and const pointers are handled similar to references. The pointed-to argument gets transferred and the server-side function is called with a pointer to the local copy.

All types specified at RPC arguments or as return value must have a default constructor. By default, all RPC arguments are input arguments, which are transferred to the server. The return type of the RPC function, if present, is an output-only value. To avoid a reference argument from acting as both input- and output argument, a const reference should be used.

8.14.3. Throwing C++ exceptions across RPC boundaries

The propagation of C++ exceptions from the server to the client is supported by a special variant of the `GENODE_RPC` macro:

```
GENODE_RPC_THROW(func_type, ret_type, func_name,
                 exc_type_list, arg_type ...)
```

This macro accepts an additional `exc_type_list` argument, which is a type list of exception types. Exception objects are not transferred as payload. The RPC mechanism propagates solely the information that the specific exception was raised. Hence, information provided with the thrown object will be lost when crossing an RPC boundary.

8.14.4. RPC interface inheritance

It is possible to extend existing RPC interfaces with additional RPC functions. Internally, such an RPC interface inheritance is realized by concatenation of the `Rpc_functions` type lists of both the base interface and the derived interface. This use case is supported by a special version of the `GENODE_RPC_INTERFACE` macro:

GENODE ⨜ LABS

```
GENODE_RPC_INTERFACE_INHERIT(base_interface,
                             rpc_func ...)
```

8.14.5. Casting capability types

For typed capabilities, the same type conversion rules apply as for pointers. In fact, a typed capability pretty much resembles a typed pointer, pointing to a remote object. Hence, assigning a specialized capability (e. g., `Capability<Input::Session>`) to a base-typed capability (e. g., `Capability<Session>`) is always valid. For the opposite case, a static cast is needed. For capabilities, this cast is supported by

```
static_cap_cast<INTERFACE>(cap)
```

In rare circumstances, mostly in platform-specific base code, a reinterpret cast for capabilities is required. It allows to convert any capability to another type:

```
reinterpret_cap_cast<INTERFACE>(cap)
```

8.14.6. Non-virtual RPC interface functions

It is possible to declare RPC functions using `GENODE_RPC`, which do not exist as virtual functions in the abstract interface class. In this case, the function name specified as third argument to `GENODE_RPC` is of course not valid for the interface class but an alternative class can be specified as second argument to the server-side `Rpc_object`. This way, a server-side implementation may specify its own class to direct the RPC function to a local (possibly non-virtual) implementation.

8.14.7. Limitations of the RPC mechanism

The maximum **number of RPC function arguments** is limited to 7. If a function requires more arguments, it may be worthwhile to consider grouping some of them in a compound struct.

GENODE ⫘ LABS

8.14.8. Root interface

Each service type is represented as an RPC object implementing the root interface. The server announces its service type by providing the service name and the capability of the service's root interface (`announce` function of the parent interface). Given the capability to the root interface, the parent is then able to create and destroy sessions.
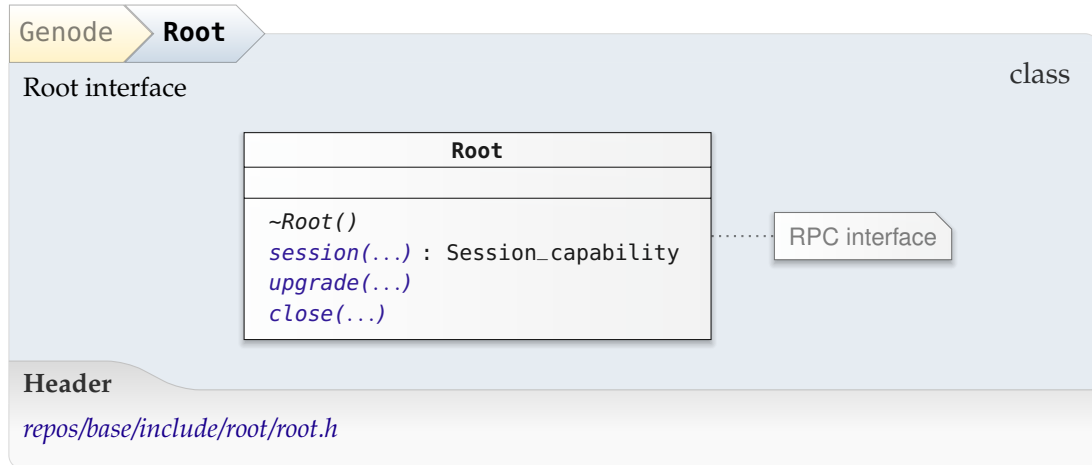
Genode	**Root**		class

Root interface

Root
~Root() *session(...)* : Session_capability *upgrade(...)* *close(...)*

RPC interface

Header

repos/base/include/root/root.h

Genode	Root	*session*

Create session   pure virtual method

Arguments

args   **Session_args const &**

affinity   **Affinity const &**

Exceptions

Unavailable
Quota_exceeded
Invalid_args

Return value

Session_capability   Capability to new session

GENODE LABS

Genode 〉 Root 〉 *upgrade*

Extend resource donation to an existing session   pure virtual method

Arguments

 session   **Session_capability**

 args   **Upgrade_args const &**

Genode 〉 Root 〉 *close*

Close session   pure virtual method

Argument

 session   **Session_capability**

GENODE LABS

Genode ⟩ **Root**

Root interface   class

Root
~Root() *session(...)* : Session_capability *upgrade(...)* *close(...)*

········ RPC interface

Header

repos/base/include/root/root.h

Genode ⟩ Root ⟩ *session*

Create session   pure virtual method

Arguments

args   **Session_args const &**

affinity   **Affinity const &**

Exceptions

Unavailable
Quota_exceeded
Invalid_args

Return value

Session_capability   Capability to new session

Genode ⟩ Root ⟩ *upgrade*

Extend resource donation to an existing session   pure virtual method

Arguments

session   **Session_capability**

args   **Upgrade_args const &**

GENODE ⚖ LABS

Genode	Root	*close*	
Close session			pure virtual method
Argument			
session	**Session_capability**		

Because defining root interfaces for services follows a recurring pattern, there exists default template classes that implement the standard behaviour of the root interface for services with multiple clients (Root_component) and services with a single client (Static_root).

441

GENODE ⋙ LABS

Genode **Root_component**

Template for implementing the root interface

class template

Rpc_object <**Typed_root**<**SESSION_TYPE**> >

Root_component
_create_session(...) : SESSION_TYPE *
_create_session(...) : SESSION_TYPE *
_upgrade_session(...)
_destroy_session(...)
md_alloc() : Allocator *
ep() : Rpc_entrypoint *
Root_component(...)
session(...) : Session_capability
upgrade(...)
close(...)

SESSION_TYPE,
POLICY

Template arguments

SESSION_TYPE	**typename**
	Session-component type to manage, derived from rpc_object
POLICY	**typename**
	Session-creation policy

Details

The POLICY template parameter allows for constraining the session creation to only one instance at a time (using the Single_session policy) or multiple instances (using the Multiple_sessions policy).

The POLICY class must provide the following two methods:

'aquire(const char *args)' is called with the session arguments at creation time of each new session. It can therefore implement a session-creation policy taking session arguments into account. If the policy denies the creation of a new session, it throws one of the exceptions defined in the Root interface.

'release' is called at the destruction time of a session. It enables the policy to keep track of and impose restrictions on the number of existing sessions.

The default policy Multiple_clients imposes no restrictions on the creation of new sessions.

Header

repos/base/include/root/component.h

GENODE ⋡ LABS

Genode 〉 Root_component 〉 *_create_session*

virtual method

Create new session (to be implemented by a derived class)

Arguments

args **const char ***

- **Affinity const &**

Exceptions

Allocator::Out_of_memory Typically caused by the meta-data allocator
Root::Invalid_args Typically caused by the session-component constructor

Return value

SESSION_TYPE *

Details

Only a derived class knows the constructor arguments of a specific session. Therefore, we cannot unify the call of its new operator and must implement the session creation at a place, where the required knowledge exist.

In the implementation of this method, the heap, provided by Root_component must be used for allocating the session object.

If the server implementation does not evaluate the session affinity, it suffices to override the overload without the affinity argument.

Genode 〉 Root_component 〉 *_create_session*

virtual method

Argument

args **const char ***

Return value

SESSION_TYPE *

GENODE LABS

Genode	Root_component	**_upgrade_session**	

virtual method

Inform session about a quota upgrade

Arguments

- **SESSION_TYPE ***

- **const char ***

Details

Once a session is created, its client can successively extend its quota donation via the `Parent::transfer_quota` operation. This will result in the invokation of `Root::upgrade` at the root interface the session was created with. The root interface, in turn, informs the session about the new resources via the `_upgrade_session` method. The default implementation is suited for sessions that use a static amount of resources accounted for at session-creation time. For such sessions, an upgrade is not useful. However, sessions that dynamically allocate resources on behalf of its client, should respond to quota upgrades by implementing this method.

Genode	Root_component	**_destroy_session**	

virtual method

Argument

session   **SESSION_TYPE ***

Genode	Root_component	**md_alloc**	

method

Return value

Allocator *   Allocator to allocate server object in `_create_session()`

Genode	Root_component	**ep**	

method

Return value

Rpc_entrypoint *   Entrypoint that serves the root component

GENODE LABS

Genode　〉　Root_component　〉　**Root_component**

constructor

Arguments

ep | **Entrypoint &**
Entry point that manages the sessions of this root interface

md_alloc | **Allocator &**
Meta-data allocator providing the backing store for session objects

GENODE LABS
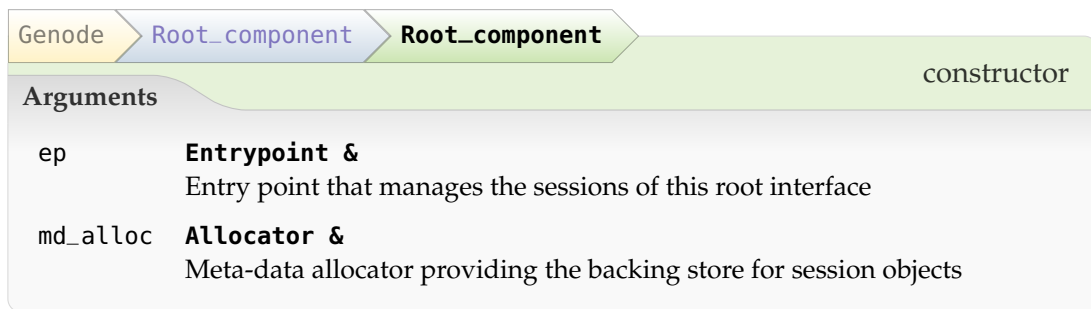
8.14.9. Server-side policy handling

The Session_label and Session_policy utilities aid the implementation of the server-side policy-selection mechanism described in Section 4.6.2.

GENODE ⋛ LABS

Genode 〉 **Session_policy**

Query server-side policy for a session request

class

Xml_node

△

Session_policy
Session_policy(...)

Header

repos/os/include/os/session_policy.h

Genode 〉 Session_policy 〉 **Session_policy**

constructor template

Template argument

N **size_t**

Arguments

label **String**<N> **const &**
      Label used as the selector of a policy

- **xml_node()**

Exception

No_policy_defined The server configuration has no policy defined for the specified
      label

Details

On construction, the Session_policy looks up the policy XML node that matches the label
provided as argument. The server-side policies are defined in one or more policy subnodes
of the server's config node. Each policy node has a label attribute. If the policy label
matches the first part of the label as delivered as session argument, the policy matches. If
multiple policies match, the one with the longest label is selected.

GENODE LABS

8.14.10. Packet stream

The Packet_stream is the building block of the asynchronous transfer of bulk data
(Section 3.6.6). The public interface consists of the two class templates Packet_stream_source,
and Packet_stream_sink. Both communication parties agree on a policy with regard
to the organization of the communication buffer by specifying the same Packet_stream_policy
as template argument.

The communication buffer consists of three parts, a submit queue, an acknowledge-
ment queue, and a bulk buffer. The submit queue contains packets generated by the
source to be processed by the sink. The acknowledgement queue contains packets that
are processed and acknowledged by the sink. The bulk buffer contains the actual pay-
load. The assignment of packets to bulk-buffer regions is performed by the source.

Genode ▸ **Packet_descriptor**

class

Default packet descriptor

```
                        Packet_descriptor
          ┌──────────────────────────────────────┐
          │                                      │
          ├──────────────────────────────────────┤
          │ Packet_descriptor(...)               │
          │ Packet_descriptor()                  │
          │ offset() : Genode::off_t             │
          │ size() : Genode::size_t              │
          │ valid() : bool                       │
          └──────────────────────────────────────┘
```

Accessors

offset   **Genode::off_t**

size   **Genode::size_t**

valid   **bool**

Details

A class used as PACKET_DESCRIPTOR arguments to the Packet_stream_policy template
must implement the interface of this class.

Header

repos/os/include/os/packet_stream.h

GENODE LABS

Genode	Packet_descriptor	**Packet_descriptor**	
			constructor

Arguments

offset   **off_t**

size   **size_t**

Genode	Packet_descriptor	**Packet_descriptor**	
			constructor

Default constructor used for instantiating arrays of packet-descriptors used as submit and ack queues.

Genode	**Packet_stream_base**	
		class

Common base of Packet_stream_source and Packet_stream_sink

```
Packet_stream_base
```

Header

repos/os/include/os/packet_stream.h

Genode	**Packet_stream_source**	
		class template

Originator of a packet stream

```
                                POLICY
Packet_stream_source
```

Template argument

POLICY   **typename**

Header

repos/os/include/os/packet_stream.h

GENODE ⫽ LABS

Genode   **Packet_stream_sink**

Receiver of a packet stream   class template

```
                                              ┌ ─ ─ ─ ─ ─ ┐
                                                POLICY
                           ┌──────────────────┐└ ─ ─ ─ ─ ─ ┘
                           │ Packet_stream_sink│
                           └──────────────────┘
```

Template argument

POLICY   **typename**

Header

repos/os/include/os/packet_stream.h

Genode   **Packet_stream_policy**

Policy used by both sides source and sink   class template

```
                                        ┌ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ┐
                                          PACKET_DESCRIPTOR,
                                          SUBMIT_QUEUE_SIZE,
                                          ACK_QUEUE_SIZE,
                        ┌───────────────────┐ CONTENT_TYPE
                        │ Packet_stream_policy│─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ┘
                        └───────────────────┘
```

Template arguments

PACKET_DESCRIPTOR   **typename**

SUBMIT_QUEUE_SIZE   **unsigned**

ACK_QUEUE_SIZE   **unsigned**

CONTENT_TYPE   **typename**

Header

repos/os/include/os/packet_stream.h

GENODE ⩘ LABS

In client-server scenarios, each client and server can play the role of either a source or a sink of packets. To ease the use of packet streams in such scenarios, the classes within the `Packet_stream_rx` and `Packet_stream_tx` namespaces provide ready-to-use building blocks to be aggregated in session interfaces.

Data transfer from server to client

Packet_stream_rx	**Channel**	
		class template

Packet_stream_rx::Channel — — — — — PACKET_STREAM_POLICY

sink() : Sink *
sigh_ready_to_ack(...)
sigh_packet_avail(...)

Template argument

PACKET_STREAM_POLICY	**typename**

Header

repos/os/include/packet_stream_rx/packet_stream_rx.h

Packet_stream_rx	Channel	*sink*
Request reception interface		virtual method

Return value

Sink *

Details

See documentation of `Packet_stream_tx::Cannel::source`.

Packet_stream_rx	Channel	*sigh_ready_to_ack*
Register signal handler for receiving `ready_to_ack` signals		pure virtual method

Argument

sigh	**Signal_context_capability**

GENODE LABS

Packet_stream_rx ⟩ Channel ⟩ **sigh_packet_avail**

pure virtual method

Register signal handler for receiving `packet_avail` signals

Argument

 sigh   **Signal_context_capability**

Packet_stream_rx ⟩ **Client**

class template

Rpc_client <CHANNEL>

△

CHANNEL

Packet_stream_rx::Client

Client(...)
sigh_ready_to_ack(...)
sigh_packet_avail(...)

Template argument

 CHANNEL   **typename**

Header

repos/os/include/packet_stream_rx/client.h

Packet_stream_rx ⟩ Client ⟩ **Client**

constructor

Argument

 channel_cap   **Capability**<**CHANNEL**>

Packet_stream_rx ⟩ Client ⟩ **sigh_ready_to_ack**

method

Argument

 sigh   **Signal_context_capability**

GENODE ⬩ LABS

Packet_stream_rx	Client	**sigh_packet_avail**	method
Argument			
sigh	**Signal_context_capability**		

453

Data transfer from client to server

Packet_stream_tx	**Channel**		

class template

> **Packet_stream_tx::Channel** ┆ PACKET_STREAM_POLICY ┆
>
> *source()* : Source *
> *sigh_ready_to_submit(...)*
> *sigh_ack_avail(...)*

Template argument

PACKET_STREAM_POLICY   **typename**

Header

repos/os/include/packet_stream_tx/packet_stream_tx.h

Packet_stream_tx	Channel	**source**

Request transmission interface

virtual method

Return value

Source *

Details

This method enables the client-side use of the Channel using the abstract Channel interface only. This is useful in cases where both source and sink of the Channel are co-located in one program. At the server side of the Channel, this method has no meaning.

Packet_stream_tx	Channel	**sigh_ready_to_submit**

Register signal handler for receiving ready_to_submit signals

pure virtual method

Argument

sigh   **Signal_context_capability**

Packet_stream_tx	Channel	**sigh_ack_avail**

Register signal handler for receiving ack_avail signals

pure virtual method

Argument

sigh   **Signal_context_capability**

GENODE LABS

Packet_stream_tx	**Client**		class template

Rpc_client `<CHANNEL>`

Packet_stream_tx::Client   `CHANNEL`

Client(...)
sigh_ready_to_submit(...)
sigh_ack_avail(...)

Template argument

CHANNEL   **typename**

Header

repos/os/include/packet_stream_tx/client.h

Packet_stream_tx	Client	**Client**	constructor

Arguments

channel_cap   **Capability**`<CHANNEL>`

buffer_alloc   **Range_allocator ***
Allocator used for managing the transmission buffer

Packet_stream_tx	Client	**sigh_ready_to_submit**	method

Argument

sigh   **Signal_context_capability**

Packet_stream_tx	Client	**sigh_ack_avail**	method

Argument

sigh   **Signal_context_capability**

GENODE LABS

8.15. XML processing

The configuration concept (Chapter 6 and Section 4.6) of the framework relies on XML syntax. Hence, there is the need to process XML-formed data. For parsing XML data, the `Xml_node` and the accompanied `Xml_attribute` utilities are provided. Those utilities operate directly on a text buffer that contains the XML data. There is no conversion step into an internal representation. This alleviates the need for allocating any meta data while extracting information from XML data. The XML parser is stateless.

Vice versa, the `Xml_generator` serves as a utility for generating XML formatted data. The scope of an XML node is represented by a lamda function. Hence, nested XML nodes can be created by nested lamda functions, which makes the structure of the XML data immediately apparent at the C++ source code. As for the `Xml_node` the `Xml_generator` does not use any internal intermediate representation of the XML data. No dynamic memory allocations are needed while generating XML-formatted output.

8.15.1. XML parsing

Genode's XML parser consists of the two classes `Xml_node` and `Xml_attribute`. Its primary use case is the provisioning of configuration information to low-level components. Consequently, it takes the following considerations into account:

Low complexity Because the parser is implicitly used in most components, it must not be complex to keep its footprint on the trusted computing base as small as possible.

Free-standing The parser must be able to operate without external dependencies such as a C runtime. Otherwise, each Genode-based system would inherit such dependencies.

No dynamic memory allocations The parser should not dynamically allocate memory to be usable in resource multiplexers and runtime environments where no anonymous memory allocations are allowed (Section 3.3.3).

Robustness The parser must be robust in the sense that it must not contain buffer overflows, infinite loops, memory corruptions, or any other defect that may make the program crash.

Other possible goals like expressive error messages, the support for more general use cases, and even the adherence to standards are deliberately subordinated. Given its low complexity, the XML parser cannot satisfy components that need advanced XML processing such as validating XML data against a DTD or schema, mutating XML nodes, or using different character encodings. In such cases, component developers may consider the use of ported 3rd-party XML parser.

GENODE ⫽ LABS

Genode ⟩ **Xml_attribute**

Representation of an XML-node attribute

class

```
            Xml_attribute
  ─────────────────────────────
  type(...)
  name() : Name
  has_type(...) : bool
  value_size() : size_t
  value_base() : char const *
  value(...)
  has_value(...) : bool
  value(...) : bool
  value(...)
  next() : Xml_attribute
```

Accessors

name	**Name**
value_base	**char const ***

Details

An attribute has the form name="value".

Header

repos/base/include/util/xml_node.h

Genode ⟩ Xml_attribute ⟩ **type**

const method

Arguments

dst	**char ***
max_len	**size_t**

Genode ⟩ Xml_attribute ⟩ **has_type**

method

Argument

type	**const char ***

Return value

bool	True if attribute has specified type

GENODE LABS

Genode ⟩ Xml_attribute ⟩ **value_size**

const method

Return value

size_t   Size of value

Genode ⟩ Xml_attribute ⟩ **value**

const method

Arguments

dst   **char ***

max_len   **size_t**

Genode ⟩ Xml_attribute ⟩ **has_value**

const method

Argument

value   **const char ***

Return value

bool   True if attribute has the specified value

Genode ⟩ Xml_attribute ⟩ **value**

const method template

Template argument

T   **typename**
     Type of value to read

Argument

out   **T ***

Return value

bool   True on success, or false if attribute is invalid or value conversion failed

GENODE ⬆ LABS

Genode › Xml_attribute › **value**

const method template

Template argument

N **size_t**

Argument

out **String**<N> *

Genode › Xml_attribute › **next**

const method

Return value

Xml_attribute   Next attribute in attribute list

GENODE LABS

Genode 〉 **Xml_node**

Representation of an XML node

class

Xml_node
Xml_node(...)
type_name(...)
type() : Type
has_type(...) : bool
value(...)
value(...) : bool
addr() : const char *
size() : size_t
content_base() : char const *
content_size() : size_t
decoded_content(...) : size_t
decoded_content() : STRING
num_sub_nodes() : size_t
next() : Xml_node
next(...) : Xml_node
last(...) : bool
sub_node(...) : Xml_node
sub_node(...) : Xml_node
for_each_sub_node(...)
for_each_sub_node(...)
attribute(...) : Xml_attribute
attribute(...) : Xml_attribute
attribute_value(...) : T
has_attribute(...) : bool
has_sub_node(...) : bool

Accessor

  type   **Type**

Header

repos/base/include/util/xml_node.h

GENODE LABS

Genode › Xml_node › **Xml_node**

constructor

Arguments

addr   **const char ***

max_len   **size_t**
          Default is ~0UL

Exception

Invalid_syntax

Details

The constructor validates if the start tag has a matching end tag of the same depth and counts the number of immediate sub nodes.

Genode › Xml_node › **type_name**

const method
Request type name of XML node as null-terminated string

Arguments

dst   **char ***

max_len   **size_t**

Genode › Xml_node › **has_type**

const method

Argument

type   **const char ***

Return value

bool   True if tag is of specified type

Genode › Xml_node › **value**

const method
Request content of XML node as null-terminated string

Arguments

dst   **char ***

max_len   **size_t**

461

GENODE LABS

Genode	Xml_node	**value**	
Read content as typed value from XML node			const method template

Template argument

T **typename**
   Type of value to read from xml node

Argument

out **T ***
     Resulting value

Return value

bool   True on success

Genode	Xml_node	**addr**	
			const method

Return value

const char *   Begin of node including the start tag

Genode	Xml_node	**size**	
			const method

Return value

size_t   Size of node including start and end tags

Genode	Xml_node	**content_base**	
			const method

Return value

char const *   Pointer to start of content

Genode	Xml_node	**content_size**	
			const method

Return value

size_t   Size of node content

GENODE LABS

Genode > Xml_node > **decoded_content**

Export decoded node content from XML node | const method

Arguments

dst   **char ***
           Destination buffer

dst_len **size_t**
           Size of destination buffer in bytes

Return value

size_t   Number of bytes written to the destination buffer

Details

This function transforms XML character entities into their respective characters.

Genode > Xml_node > **decoded_content**

Read decoded node content as Genode::String | const method template

Template argument

STRING **typename**

Return value

STRING

Genode > Xml_node > **num_sub_nodes**

| const method

Return value

size_t   The number of the xml node's immediate sub nodes

Genode > Xml_node > **next**

| const method

Exception

Nonexistent_sub_node   Sub sequent node does not exist

Return value

Xml_node   Xml node following the current one

GENODE LABS

Genode	Xml_node	**next**

const method

Argument

type　**const char ***
　　　　Type of xml node, or 0 for matching any type

Return value

Xml_node　Next xml node of specified type

Genode	Xml_node	**last**

const method

Argument

type　**const char ***
　　　　Default is 0

Return value

bool　True if node is the last of a node sequence

Genode	Xml_node	**sub_node**

const method

Argument

idx　**unsigned**
　　　Index of sub node, default is the first node
　　　Default is 0U

Exception

Nonexistent_sub_node　No such sub node exists

Return value

Xml_node　Sub node with specified index

GENODE LABS

Genode 〉 Xml_node 〉 **sub_node**

const method

Argument

type **const char ***

Exception

Nonexistent_sub_node   No such sub_node exists

Return value

Xml_node   First sub node that matches the specified type

Genode 〉 Xml_node 〉 **for_each_sub_node**

const method template

Execute functor fn for each sub node of specified type

Template argument

FN   **typename**

Arguments

type   **char const ***

fn   **FN const &**

Genode 〉 Xml_node 〉 **for_each_sub_node**

const method template

Execute functor fn for each sub node

Template argument

FN   **typename**

Argument

fn   **FN const &**

GENODE LABS

Genode | Xml_node | **attribute**

const method

Argument

idx **unsigned**
Attribute index, first attribute has index 0

Exception

Nonexistent_attribute No such attribute exists

Return value

Xml_attribute Xml attribute

Genode | Xml_node | **attribute**

const method

Argument

type **const char ***
Name of attribute type

Exception

Nonexistent_attribute No such attribute exists

Return value

Xml_attribute Xml attribute

Genode ⟩ Xml_node ⟩ **attribute_value**

Shortcut for reading an attribute value from XML node   const method template

Template argument

T   **typename**

Arguments

type   **char const ∗**
                        Attribute name

default_value   **T**
                        Value returned if no attribute with the name type is present.

Return value

T   Attribute value or specified default value

Details

Without   this   shortcut,   attribute   values   can   be   obtained   by
node.attribute(...).value(...) only. Because the attribute lookup may throw a
Nonexistent_attribute exception, code that reads optional attributes (those with default
values) has to handle the exception accordingly. Such code tends to become clumsy, in
particular when many attributes are processed in a subsequent fashion. This method
template relieves the XML node user from implementing the exception handling manually.

Genode ⟩ Xml_node ⟩ **has_attribute**

const method

Argument

type   **char const ∗**

Return value

bool   True if attribute of specified type exists

Genode ⟩ Xml_node ⟩ **has_sub_node**

const method

Argument

type   **char const ∗**

Return value

bool   True if sub node of specified type exists

GENODE ⌇ LABS

8.15.2. XML generation

Genode **Xml_generator**

Utility for generating XML

class

```
            Xml_generator

Xml_generator(...)
node(...)
node(...)
attribute(...)
attribute(...)
attribute(...)
attribute(...)
attribute(...)
attribute(...)
attribute(...)
attribute(...)
attribute(...)
append(...)
append_sanitized(...)
used() : size_t
```

Accessor

  used   **size_t**

Header

repos/base/include/util/xml_generator.h

Genode Xml_generator **Xml_generator**

constructor template

Template argument

  FUNC   **typename**

Arguments

  dst   **char ∗**

  dst_len   **size_t**

  name   **char const ∗**

  func   **FUNC const &**

GENODE LABS

Genode 〉 Xml_generator 〉 **node**

method template

Template argument

FUNC **typename**

Argument

name **char const ∗**

Genode 〉 Xml_generator 〉 **node**

method

Argument

name **char const ∗**

Genode 〉 Xml_generator 〉 **attribute**

method

Arguments

name **char const ∗**

str **char const ∗**

Genode 〉 Xml_generator 〉 **attribute**

method template

Template argument

N **size_t**

Arguments

name **char const ∗**

str **String**<N> **const &**

Genode 〉 Xml_generator 〉 **attribute**

method

Arguments

name **char const ∗**

value **bool**

GENODE LABS

Genode ⟩ Xml_generator ⟩ **attribute**

method

Arguments

| name | **char const *** |
| value | **long** |

Genode ⟩ Xml_generator ⟩ **attribute**

method

Arguments

| name | **char const *** |
| value | **long** |

Genode ⟩ Xml_generator ⟩ **attribute**

method

Arguments

| name | **char const *** |
| value | **int** |

Genode ⟩ Xml_generator ⟩ **attribute**

method

Arguments

| name | **char const *** |
| value | **unsigned long** |

Genode ⟩ Xml_generator ⟩ **attribute**

method

Arguments

| name | **char const *** |
| value | **unsigned long** |

GENODE LABS

Genode ⟩ Xml_generator ⟩ **attribute**

method

Arguments

name	**char const ***
value	**unsigned**

Genode ⟩ Xml_generator ⟩ **append**

method

Append content to XML node

Arguments

str	**char const ***
str_len	**size_t** *Default is ~0UL*

Details

This method must not be followed by calls of `attribute`.

Genode ⟩ Xml_generator ⟩ **append_sanitized**

method

Append sanitized content to XML node

Arguments

str	**char const ***
str_len	**size_t** *Default is ~0UL*

Details

This method must not be followed by calls of `attribute`.

GENODE LABS

8.16. Component management

8.16.1. Shared objects

Genode	**Shared_object**		class

Shared_object
Shared_object(...)
~Shared_object()
lookup(...) : T

Header

repos/base/include/base/shared_object.h

Genode	Shared_object	**Shared_object**	constructor

Load shared object and dependencies

Arguments

file   **char const ***
        Shared object to load
        Default is `nullptr`

flags   **unsigned**
        Lazy for lazy function binding, now for immediate binding
        Default is `LAZY`

Exception

Invalid_file

Genode	Shared_object	**~Shared_object**	destructor

Close and unload shared object

GENODE LABS

Genode ⟩ Shared_object ⟩ **lookup**

Lookup a symbol in shared object and it's dependencies   const method template

Template argument

  T **typename**
     Default is `void *`

Argument

  symbol **const char ***
      Symbol name

Exception

  Invalid_symbol

Return value

  T Symbol address

Genode ⟩ **Address_info**

                                                                   class

Address_info
Address_info(...)

Header

repos/base/include/base/shared_object.h

Genode ⟩ Address_info ⟩ **Address_info**

                                                      constructor

Argument

  addr **addr_t**

GENODE LABS

8.16.2. Child management

For components that manage a number of child components, each child is represented by an instance of the Child class. This instance contains the policy to be applied to the child (for example how session requests are routed to services) and contains the child's execution environment including the RAM session holding the child's memory quota.

Genode ⟩ **Child_policy**

Child policy interface

class

```
                        Child_policy

~Child_policy()
name() : const char *
resolve_session_request() : Service *
filter_session_args()
announce_service() : bool
filter_session_affinity(...) : Affinity
unregister_services()
exit(...)
ref_ram_session() : Ram_session *
ref_ram_cap() : Ram_session_capability
yield_response()
resource_request(...)
```
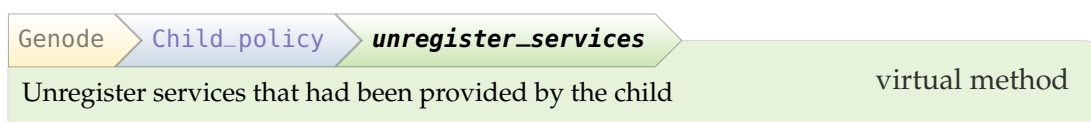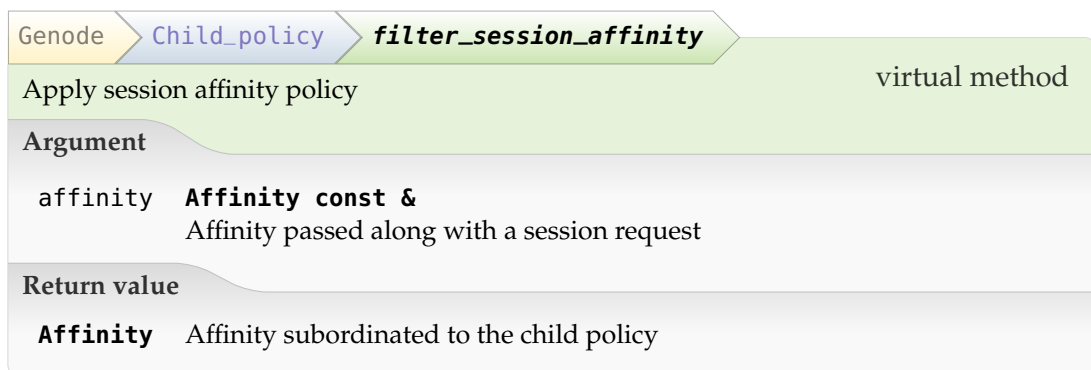
Accessor

ref_ram_cap   **Ram_session_capability**

Details

A child-policy object is an argument to a Child. It is responsible for taking policy decisions regarding the parent interface. Most importantly, it defines how session requests are resolved and how session arguments are passed to servers when creating sessions.

Header

repos/base/include/base/child.h

Genode ⟩ Child_policy ⟩ *name*

pure virtual const method

Return value

const char *   Process name of the child

GENODE ⟋ LABS

Genode › Child_policy › **resolve_session_request**

Determine service to provide a session request   virtual method

Return value

Service *   Service to be contacted for the new session, or 0 if session request could not be resolved

Genode › Child_policy › **filter_session_args**

Apply transformations to session arguments   virtual method

Genode › Child_policy › **announce_service**

Register a service provided by the child   virtual method

Return value

bool   True if announcement succeeded, or false if child is not permitted to announce service

Genode › Child_policy › **filter_session_affinity**

Apply session affinity policy   virtual method

Argument

affinity   **Affinity const &**
Affinity passed along with a session request

Return value

Affinity   Affinity subordinated to the child policy

Genode › Child_policy › **unregister_services**

Unregister services that had been provided by the child   virtual method

Genode › Child_policy › **exit**

Exit child   virtual method

Argument

exit_value   **int**

GENODE LABS

Genode ⟩ Child_policy ⟩ **ref_ram_session**

Reference RAM session   virtual method

Return value

  Ram_session *

Details

The RAM session returned by this method is used for session-quota transfers.

Genode ⟩ Child_policy ⟩ **yield_response**

Respond to the release of resources by the child   virtual method

Details

This method is called when the child confirms the release of resources in response to a yield request.

Genode ⟩ Child_policy ⟩ **resource_request**

Take action on additional resource needs by the child   virtual method

Argument

-   **Resource_args const &**

GENODE LABS

Genode › **Child**

Implementation of the parent interface that supports resource trading

class

```
                              Child

   Child(...)
   ~Child()
   heap() : Allocator *
   pd_session_cap() : Pd_session_capability
   ram_session_cap() : Ram_session_capability
   cpu_session_cap() : Cpu_session_capability
   parent_cap() : Parent_capability
   revoke_server(...)
   yield(...)
   notify_resource_avail()
   announce(...)
   session(...) : Session_capability
   upgrade(...)
   close(...)
   exit(...)
   main_thread_cap() : Thread_capability
   resource_avail_sigh(...)
   resource_request(...)
   yield_sigh(...)
   yield_request() : Resource_args
   yield_response()
```

Accessors

pd_session_cap	**Pd_session_capability**
ram_session_cap	**Ram_session_capability**
cpu_session_cap	**Cpu_session_capability**
parent_cap	**Parent_capability**
main_thread_cap	**Thread_capability**

Details

There are three possible cases of how a session can be provided to a child: The service is implemented locally, the session was obtained by asking our parent, or the session is provided by one of our children.

These types must be differentiated for the quota management when a child issues the closing of a session or transfers quota via our parent interface.

If we close a session to a local service, we transfer the session quota from our own account to the client.

If we close a parent session, we receive the session quota on our own account and must transfer this amount to the session-closing child.

If we close a session provided by a server child, we close the session at the server, transfer the session quota from the server's RAM session to our account, and subsequently transfer the same amount from our account to the client.

Header

repos/base/include/base/child.h

477

GENODE ⟋ LABS

Genode ⟩ Child ⟩ **Child**

		constructor
Arguments		

elf_ds	**Dataspace_capability**	
	Dataspace that contains the elf binary	
ldso_ds	**Dataspace_capability**	
	Dataspace that contains the dynamic linker, started if `elf_ds` is a dynamically linked executable	
pd_cap	**Pd_session_capability**	
	Capability of the new protection domain, used as argument for creating the initial thread, and handed out to the child as its environment	
pd	**Pd_session &**	
	Pd session used for assigning the parent capability of the new process	
ram_cap	**Ram_session_capability**	
	Ram session capability handed out to the child as its environment	
ram	**Ram_session &**	
	Ram session used to allocate the bss and data segments and as backing store for the local heap partition to keep child-specific meta data	
cpu_cap	**Cpu_session_capability**	
	Cpu session capability handed out to the child as its environment	
initial_thread	**Initial_thread_base &**	
	Initial thread of the new protection domain	
local_rm	**Region_map &**	
	Local address space	
remote_rm	**Region_map &**	
	Address space of new protection domain	
entrypoint	**Rpc_entrypoint &**	
policy	**Child_policy &**	
pd_service	**Service &**	
	Provider of the pd session	
	Default is _parent_service()	
ram_service	**Service &**	
	Provider of the ram session	
	Default is _parent_service()	
cpu_service	**Service &**	
	Provider of the cpu session	
	Default is _parent_service()	

Exceptions

Ram_session::Alloc_failed
Process_startup_failed

Details

Usually, the pairs of `pd` and `pd_cap`, `initial_thread` and `cpu_cap`, `ram` and `ram_cap` belong to each other. References to the session interfaces are passed as separate arguments in addition to the capabilities to allow the creator of a child to operate on locally implemented interfaces during the child initialization.

The `ram_service`, `cpu_service`, and `pd_service` arguments are needed to direct quota upgrades referring to the resources of the child environment. By default, we expect that these resources are provided by the parent.

GENODE 🗎 LABS

Genode > Child > **~Child**

virtual destructor

Details

On destruction of a child, we close all sessions of the child to other services.

Genode > Child > **heap**

method

Return value

Allocator ∗ Heap that uses the child's quota

Genode > Child > **revoke_server**

method

Discard all sessions to specified service

Argument

server **const Server ∗**

Details

When this method is called, we assume the server protection domain to be dead and all that all server quota was already transferred back to our own env()→ram_session() account. Note that the specified server object may not exist anymore. We do not de-reference the server argument in here!

Genode > Child > **yield**

method

Instruct the child to yield resources

Argument

args **Resource_args const &**

Details

By calling this method, the child will be notified about the need to release the specified amount of resources. For more details about the protocol between a child and its parent, refer to the description given in parent/parent.h.

Genode > Child > **notify_resource_avail**

const method

Notify the child about newly available resources

GENODE LABS

8.17. Utilities for user-level device drivers

To avoid common implementation bugs in device drivers, the framework facilitates the declaration of hardware register and bit layouts in the form of C++ types. By subjecting the device driver's interaction with the hardware to the to C++ type system, the consistency of register accesses with the hardware specifications can be maintained automatically. Such hardware specifications are declarative and can be cleanly separated from the program logic of the driver. The actual driver program is relieved from any intrinsics in the form of bit-masking operations.

The MMIO access utilities comes in the form of two header files located at *util/register.h* and *util/mmio.h*.

8.17.1. Register declarations

The class templates found in *util/register.h* provide a means to express register layouts using C++ types. In a way, these templates make up for C++'s missing facility to define accurate bitfields. The following example uses the `Register` class template to define a register as well as a bitfield within the register:

```
struct Vaporizer : Register<16>
{
  struct Enable : Bitfield<2,1> { };
  struct State  : Bitfield<3,3> {
    enum{ SOLID = 1, LIQUID = 2, GASSY = 3 };
  };

  static void     write (access_t value);
  static access_t read  ();
};
```

In the example, `Vaporizer` is a 16-bit register, which is expressed via the `Register` template argument. The `Register` class template allows for accessing register content at a finer granularity than the whole register width. To give a specific part of the register a name, the `Register::Bitfield` class template is used. It describes a bit region within the range of the compound register. The bit 2 corresponds to true if the device is enabled and bits 3 to 5 encode the `State`. To access the actual register, the methods `read()` and `write()` must be provided as back end, which performs the access of the whole register. Once defined, the `Vaporizer` offers a handy way to access the individual parts of the register, for example:

GENODE LABS

```
/* read the whole register content */
Vaporizer::access_t r = Vaporizer::read();

/* clear a bit field */
Vaporizer::Enable::clear(r);

/* read a bit field value */
unsigned old_state = Vaporizer::State::get(r);

/* assign new bit field value */
Vaporizer::State::set(r, Vaporizer::State::LIQUID);

/* write whole register */
Vaporizer::write(r);
```

Bitfields that span multiple registers   The register interface of hardware devices
may be designed such that bitfields are not always consecutive in a single register. For
example, values of the HDMI configuration of the Exynos-5 SoC are scattered over mul-
tiple hardware registers. The problem is best illustrated by the following example of a
hypothetical timer device. The bits of the clock count value are scattered across two
hardware registers, the lower 6 bits of the 16-bit-wide register 0x2, and two portions of
the 32-bit-wide register 0x4. A declaration of those registers would look like this:

```
struct Clock_2 : Register<0x2, 16>
{
  struct Value : Bitfield<0, 6> { };
};

struct Clock_1 : Register<0x4, 32>
{
  struct Value_2 : Bitfield<2, 13> { };
  struct Value_1 : Bitfield<18, 7> { };
};
```

Writing a clock value needs consecutive write accesses to both registers with bit shift
operations applied to the value:

```
write<Clock_1::Value_1>(clk);
write<Clock_1::Value_2>(clk >> 7);
write<Clock_2::Value>(clk >> 20);
```

The new `Bitset_2` and `Bitset_3` class templates contained in *util/register.h* allow
the user to compose a logical bit field from 2 or 3 physical bit fields. The order of

GENODE ⟋ LABS

the template arguments expresses the order of physical bits in the logical bit set. Each argument can be a register, a bit field, or another bit set. The declaration of such a composed bit set for the example above looks as follows:

```
struct Clock : Bitset_3<Clock_1::Value_1,
                        Clock_1::Value_2,
                        Clock_2::Value> { };
```

With this declaration in place, the driver code becomes as simple as:

```
write<Clock>(clk);
```

Under the hood, the framework performs all needed consecutive write operations on the registers 0x2 and 0x4.

8.17.2. Memory-mapped I/O

The utilities provided by *util/mmio.h* use the `Register` template class as a building block to provide easy-to-use access to memory-mapped I/O registers. The `Mmio` class represents a memory-mapped I/O region taking its local base address as constructor argument. The following example illustrates its use:

```
class Timer : Mmio
{
  struct Value   : Register<0x0, 32> { };
  struct Control : Register<0x4, 8> {
    struct Enable  : Bitfield<0,1> { };
    struct Irq     : Bitfield<3,1> { };
    struct Method  : Bitfield<1,2>
    {
      enum { ONCE = 1, RELOAD = 2, CYCLE = 3 };
    };
  };

  public:

    Timer(addr_t base) : Mmio(base) { }

    void enable();
    void set_timeout(Value::access_t duration);
    bool irq_raised();
};
```

GENODE LABS

The memory-mapped timer device consists of two registers: The 32-bit `Value` register and the 8-bit `Control` register. They are located at the MMIO offsets 0x0 and 0x4, respectively. Some parts of the `Control` register have specific meanings as expressed by the `Bitfield` definitions within the `Control` struct.

Using these declarations, accessing the individual bits becomes almost a verbatim description of how the device is used. For example:

```
void enable()
{
  /* access an individual bitfield */
  write<Control::Enable>(true);
}

void set_timeout(Value::access_t duration)
{
  /* write complete content of a register */
  write<Value>(duration);

  /* write all bitfields as one transaction */
  write<Control>(Control::Enable::bits(1) |
                 Control::Method::bits(Control::Method::ONCE) |
                 Control::Irq::bits(0));
}

bool irq_raised()
{
  return read<Control::Irq>();
}
```

GENODE ⧸ LABS