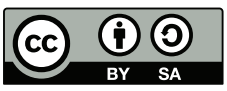
# Genode Operating System Framework

Norman Feske

March 6, 2015

## Contents



This work is licensed under the Creative Commons Attribution + ShareAlike License (CC-BY-SA). To view a copy of the license, visit http://creativecommons.org/licenses/by-sa/4.0/legalcode

# 1 Introduction

## 1.1 Universal truths

*TODO*

*TODO define term "trusted computing base"*

## 1.2 Operating-system framework

The Genode OS framework 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.

Genode is based on a recursive system structure. Each program runs in a dedicated sandbox and gets granted only those access rights and resources that are needed for 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 L4 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 use 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, Codezero, 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 was specifically developed for Genode and thereby further reduces the complexity of the trusted computing base compared to other kernels.

### Virtualization

Genode support 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.
- 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

## 1.3 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*.

## 1.4 Document structure

*TODO*

## 2 Getting started

*TODO*

- Obtaining the source code
- Source-tree structure
- Downloading 3-rd party source code
- Installing the tool chain
- Creating a build directory
- Configuring the build directory
- Building a target
- Executing a simple system scenario
  - Linux
  - Microkernel
- Tutorial for building a hello-world program

## 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 on 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 for bugs in the overly complex kernel. In other words: It is certain. Moreover, once an in-kernel function has been compromised, the defect has an unlimited reach throughout the system.

The Genode architecture was designed to give more assuring answers to the two stated questions. 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 reach of the defect is limited to the particular part 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 acquainted with each other. An operating system ultimately acts on physical hardware resources such as memory, CPUs, and peripheral devices. Section 3.4 describes how

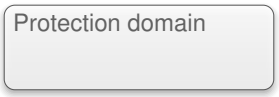
such resources are made available to components. Section 3.5 answers the question of how a new component comes to life. The variety of relationships between components and their respective interfaces call for different communication primitives. Section 3.6 introduces Genode's inter-component communication mechanisms in detail.

## 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 8.6 and 8.7 provide details for selected kernels.

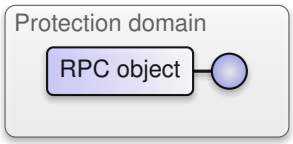
### 3.1.1 Capability spaces, object identities, and RPC objects

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 1 illustrates the relationship of an RPC object and its 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

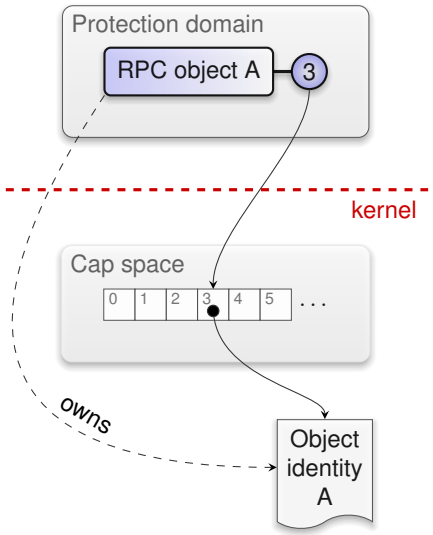



**Figure 1:** Relationship between an RPC object and its corresponding object identity.

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. Figure 2 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 2, the RPC object can 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

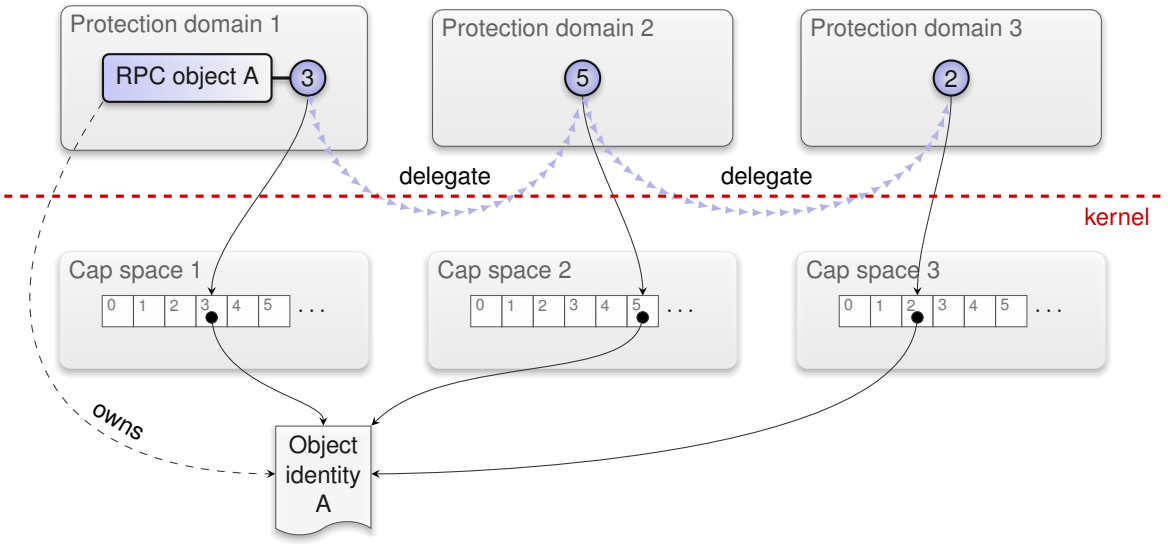



**Figure 2:** The transitive delegation of a capability from one protection domain to others.

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 puts the function arguments to 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, puts 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 *entrypoint* denoting the fact that it becomes active only when a call from a client enters the protection domain. In order to be able to act as a server, a component has to have at least one entrypoint.

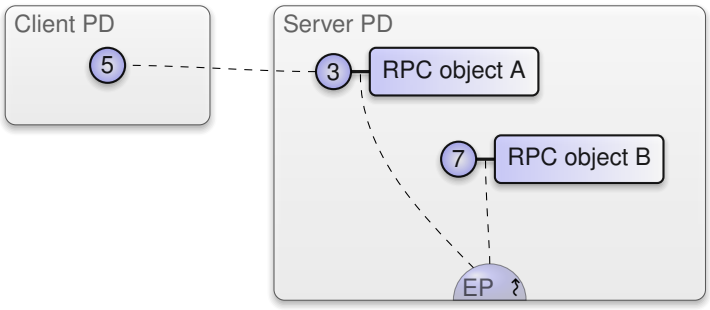



**Figure 3:** The RPC object A and B are associated with the server's entrypoint. A client has a capability for A but not for B. For brevity, the kernel-protected object identities are not depicted. Instead, the dashed line between the capabilities shows that both capabilities refer to the same object identity.

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 association between RPC objects with 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 structure called *object pool*, which allows for looking up the matching RPC object for a given capability. Figure 3 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 4 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 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 the 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

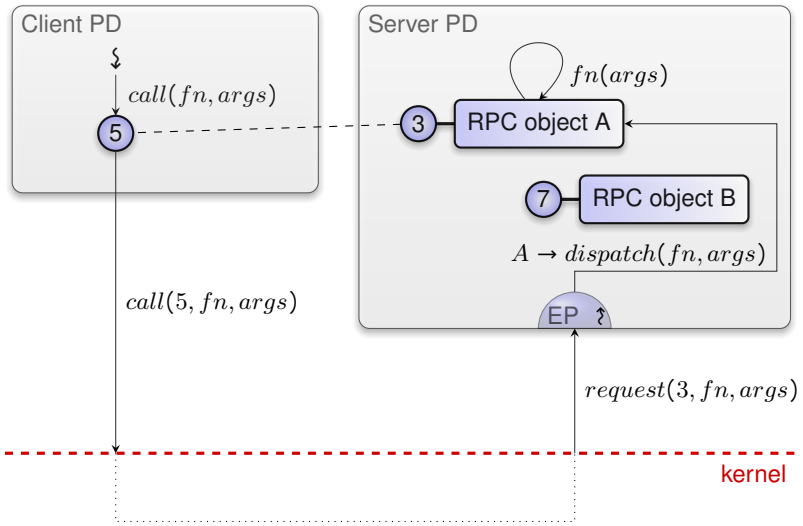



**Figure 4:** Control flow between client and server when the client calls a method of an RPC object.

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.

### 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 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 5. 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.

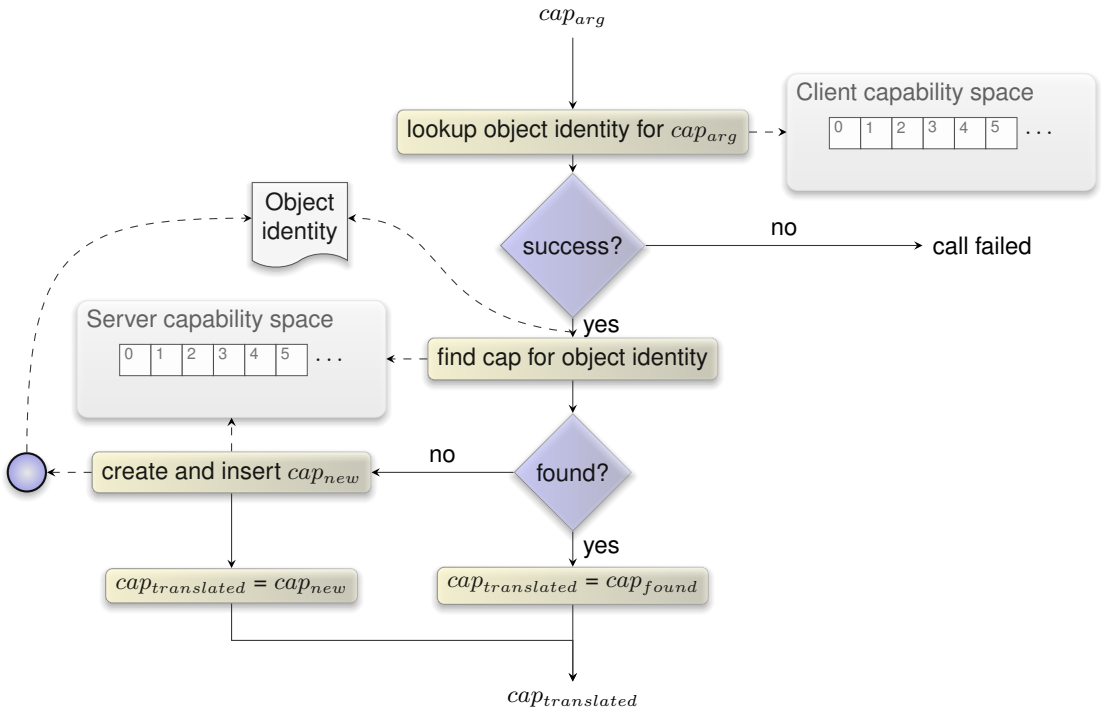



**Figure 5:** Procedure of delegating a capability specified as RPC argument from a client to a server.

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, the kernel is able to obtain the object identity referred to by the argument. Note that under no circumstances, the client can 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 slots of its capability space, the protection domain was authorized to use their referenced object identities by the means of prior delegations.

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.

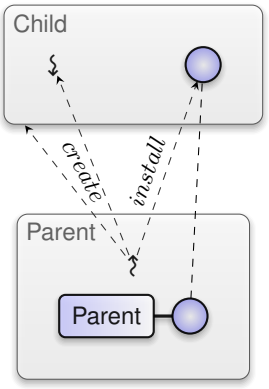



**Figure 6:** Initial relationship between a parent and a new created child.

## 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 to 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 used by the component 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.

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 anytime 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 6 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 7 shows an example scenario. The init component creates sub systems 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 creating and destroying sessions of the service. Figure 8 shows a scenario where the GUI component announces its service to the init component. The announce function

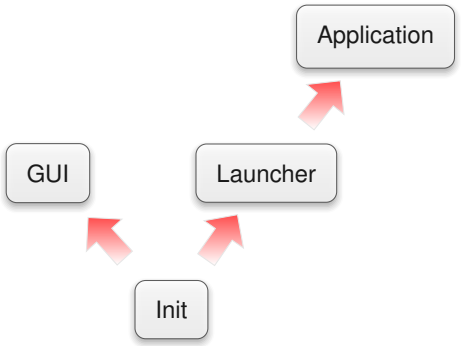



**Figure 7:** Example of a tree of components. The red arrow represents the ownership relation.

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 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 by 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 9 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 it is interested in reading user input and 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 of 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.

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

Figure 9 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

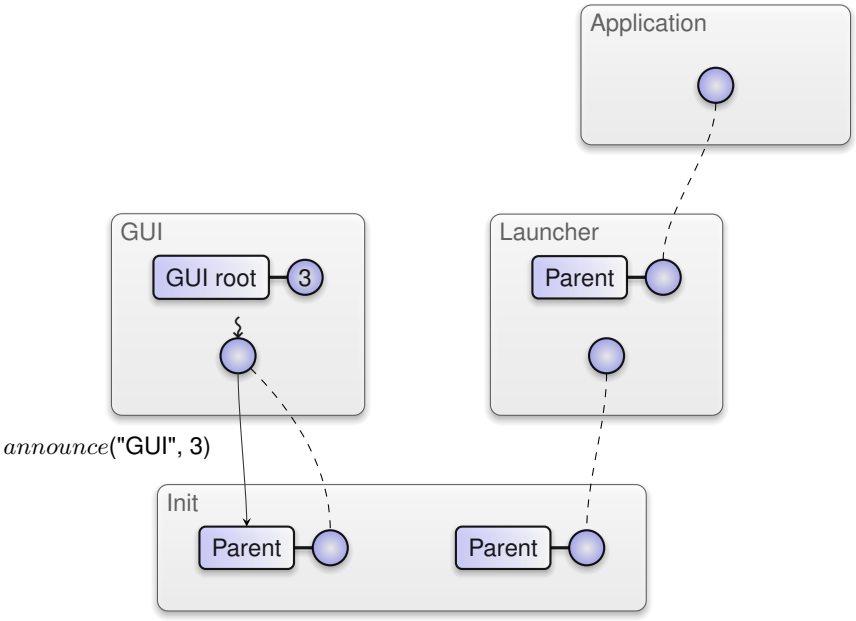



**Figure 8:** The GUI component announces its service to its parent using the parent interface.

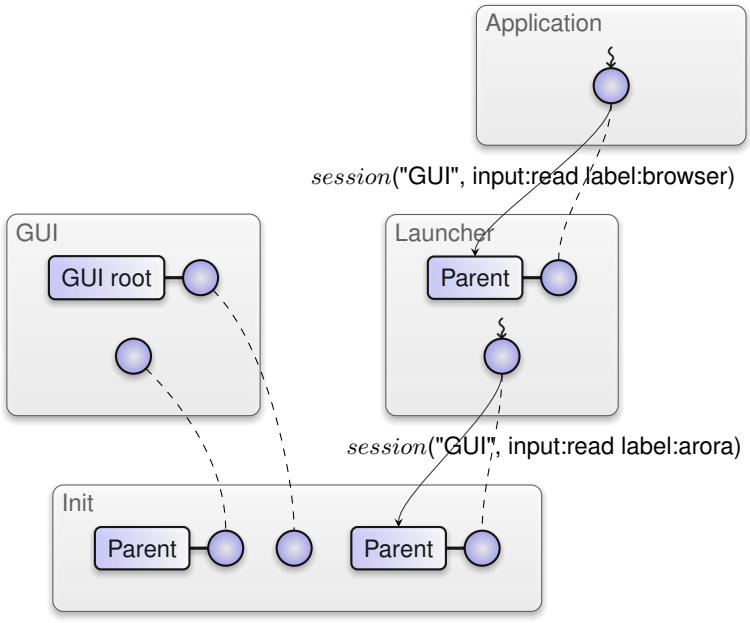



**Figure 9:** The application requests a GUI session using the parent interface.

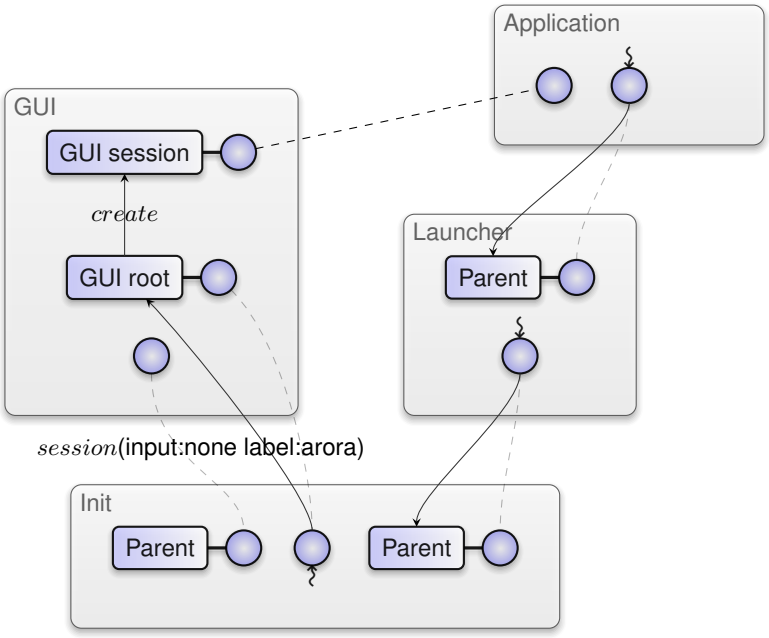



**Figure 10:** Session creation at the server.

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.

In the example, init decides to override the "input" argument. 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 respond 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 object. 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 (Section 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.

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.

If being in the *role of a server*, a component should generally not trust its clients. In 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 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 de-

scribed 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 does never need 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.

Regarding the lifetime of a session, 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 both 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. 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. Otherwise, the parent of a server might take steps to enforce its will by destructing the server altogether.

## 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 provide abstractions of those 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 an 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 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 application. 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 behaviour 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 hardly 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 behind the RAM service is to draw the analogy to 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 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,

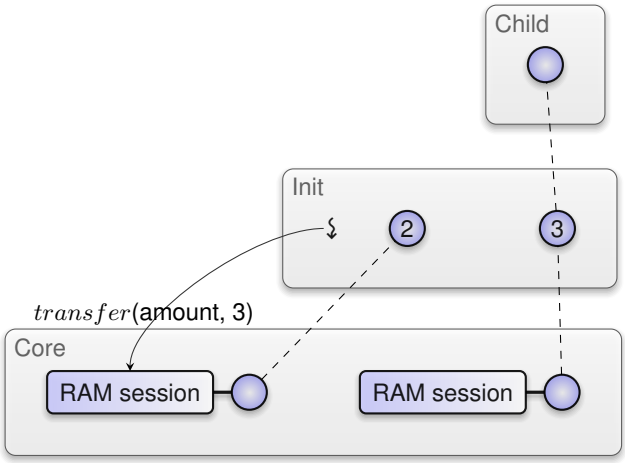



**Figure 11:** Init assigns a portion of its memory to a child. In addition to its own RAM session (2), init has created a second RAM session (3) designated for its child.

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 11 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 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

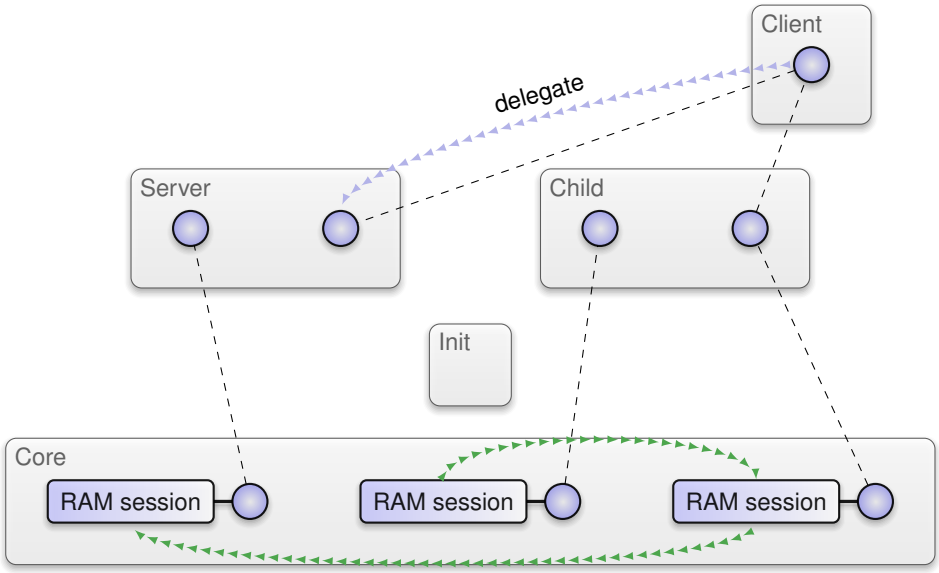



**Figure 12:** Memory-stealing attempt

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

To prevent such 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 prior known 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 distinct three different cases depending on its session-routing decision as described in Section 3.2.3.

**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 is 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.

Figure 13 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

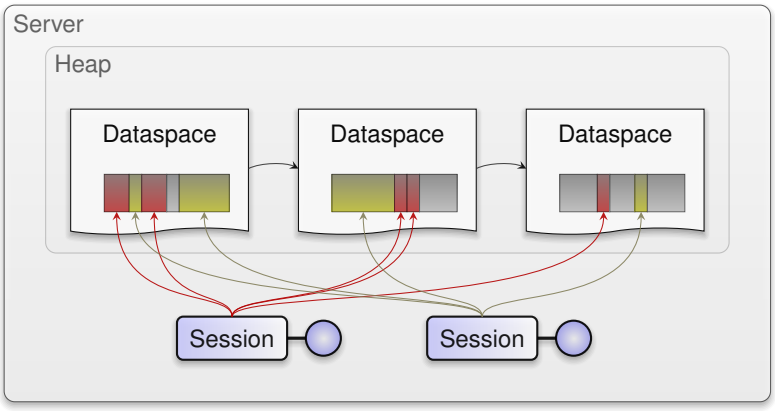



**Figure 13:** A server allocates anonymous memory on behalf of multiple clients from a single heap.

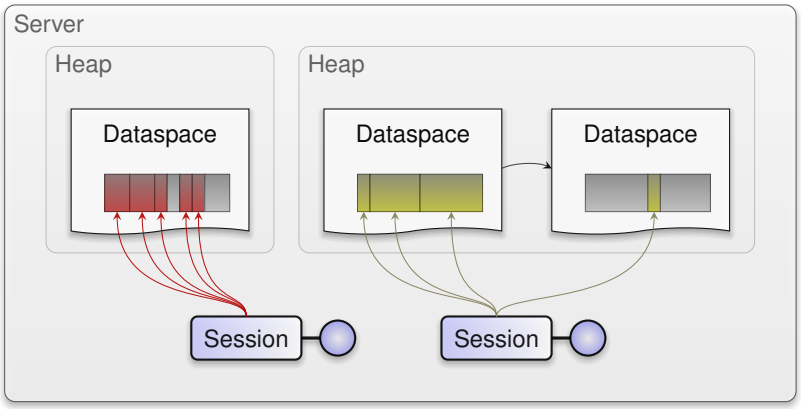



**Figure 14:** A server performs memory allocations from session-specific heap partitions.

still occupied with memory objects of another session. Therefore, the server becomes unable to comply with parent's expectation.

The solution of this problem is illustrated in Figure 14. 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 its 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 mechanism are described in Section 3.6. The child uses remote procedure calls to talk to its parent whereas the parent uses asynchronous notification to signal state changes to the child. The protocol consists of two parts, which are complementary.

**Resource requests** By issuing a resource request to it 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 had been granted or not. The child may proceed working under the low resource conditions or it may block 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 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 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.

## 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*. E.g., 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 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 (using core's RM service described in Section 3.4.5). Hence, by the means of delegating dataspace capabilities, components can establish shared memory.

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 Physical memory allocation (RAM)

A RAM session 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.

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.

### 3.4.3 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. 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.

### 3.4.4 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. At the hardware-level, the CPU isolates different protection domains via a memory-management unit. Each domain is represented by a different page directory, or an address-space ID (ASID). A PD session represents the used hardware-based protection facility.

In addition to representing the unit of memory protection, a PD comprises a capability space as introduced in Section 3.1.1. Initially, the PD's capability space is empty. However, the PD client can populate the capability space with a single capability, which is the parent capability of the component within the PD. The assignment of the parent capability is done at the creation time of the component by its parent.

A PD on its own is not useful unless it becomes associated with an address-space layout (RM session) and at least one thread of execution (CPU session). Section 3.5 explains how those sessions can be combined as basic building blocks for creating a component.

### 3.4.5 Address-space management (RM)

A region-manager (RM) session represents the layout of a virtual address space. The size of the virtual address space can be defined via session arguments at the session-creation time.

**Populating an address space** The concept behind RM sessions is a generalization of the MMU's page-table mechanism. Analogously to how a page table is populated with physical page frames, an RM session is populated with dataspaces. Under the hood, core uses the MMU's page-table mechanism as a cache for RM sessions. An RM client in possession of a dataspace capability is able to *attach* the dataspace to the RM session. Thereby the content of the dataspace becomes visible within the RM session's virtual address space. When attaching a dataspace to an RM session, 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 RM session. Furthermore, the client can specify whether the content of the dataspace should be executable or not.

The counter part of the *attach* operation is the *detach* operation, which enables the RM client to remove dataspaces from its RM session 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 RM sessions. A dataspace may also be attached multiple times to one RM session. In this case, each attach operation populates a distinct region of the virtual address space.

**Assigning threads to an address space** As for a PD session, an RM session is not useful on its own. To enable the use of the RM-session's address-space layout for a component, it must first be associated with a thread of execution. An RM client can establish this association with the RM session's *add-client* operation, which takes a thread capability (obtained from a CPU session) as argument. Once associated, core uses the address-space layout of the RM session to resolve page faults caused by the thread.

**Realizing managed dataspaces** The entirety of an RM session can be used as a dataspace. Such a *managed dataspace* is not backed by a range of physical addresses but by the range of virtual addresses of its underlying RM session. This makes RM sessions a generalization of nested page tables. An RM client can obtain a dataspace capability for a given RM session 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 thread-context area 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 creating an RM session that represents the complete thread-context area. This RM session is attached as a dataspace to the component's virtual address space. When creating a new thread with its corresponding stack, the thread's stack is not directly attached to the component's RM session but to the context area's RM session. 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-paged dataspaces. A server may hand out dataspace capabilities that are backed by RM sessions to its clients. Once the client has attached this dataspace to its address space and touches the content, the client triggers a page fault. Core responds to this page fault by blocking the client thread and delivering a notification to the RM client of the managed dataspace (the server) along with the information about the fault address within the RM session. 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.

### 3.4.6 Processing-time allocation (CPU)

A CPU session 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 associating the thread with its address space layout (RM session) as well as 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 a lock.

### 3.4.7 Object-identity allocation (CAP)

Genode regards object identities as a physical resource because each object identity is represented as a kernel object. Core's CAP service allows for the creation and destructions of new object identities. For each RPC object associated to an RPC entrypoint, the entrypoint requests the creation of a new object identity from a CAP session and associates the RPC object with the capability allocated from the CAP service.

For more information about the relationship between RPC objects and object identities, refer to Section 3.1. The interplay between RPC objects, the RPC entrpoint, and core's CAP service is described in depth in Section 3.6.1.

### 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 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 storing information, of which the physical location in memory is of no matter, 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 byes. 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 com-

ponents. 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 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 corresponds to an interrupt. The physical interrupt number is specified as session argument. Each physical interrupt number can be specified to 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 occurs, core wakes up the IRQ client and masks the interrupt at the interrupt controller until the driver has completed the IRQ handing 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 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 Asynchonous notifications (SIGNAL)

Core's SIGNAL service plays the role of a broker of 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.

### 3.4.11 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.

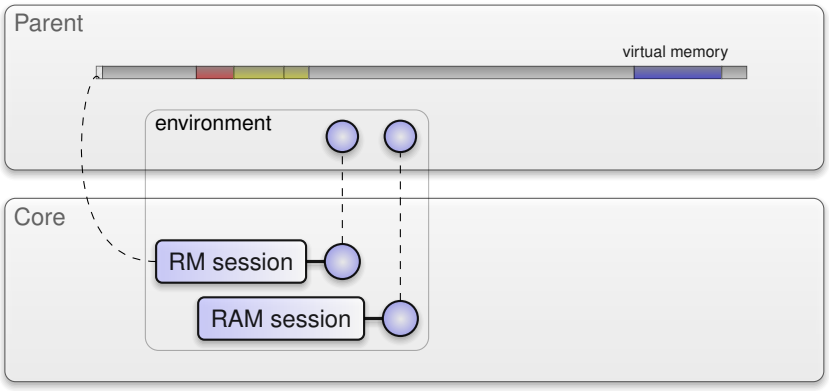



**Figure 15:** Starting point for creating a new component

## 3.5 Component creation

Each Genode component is made out of five 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

**RM** session for managing the component's address-space layout

**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 15. The parent's memory budget is represented by the parent's RAM (Section 3.4.2) session. The parent's virtual address space is represented by the parent's RM session (Section 3.4.5). Both sessions were originally created at the parent's construction time. Along with the parent's CPU session and PD session, those sessions form the parent's so-called *environment*. The parent's RM session is populated with the parent's code (shown as red), the so-called thread-context 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 3.4.3). 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 RM session. After this step, the parent is able to inspect

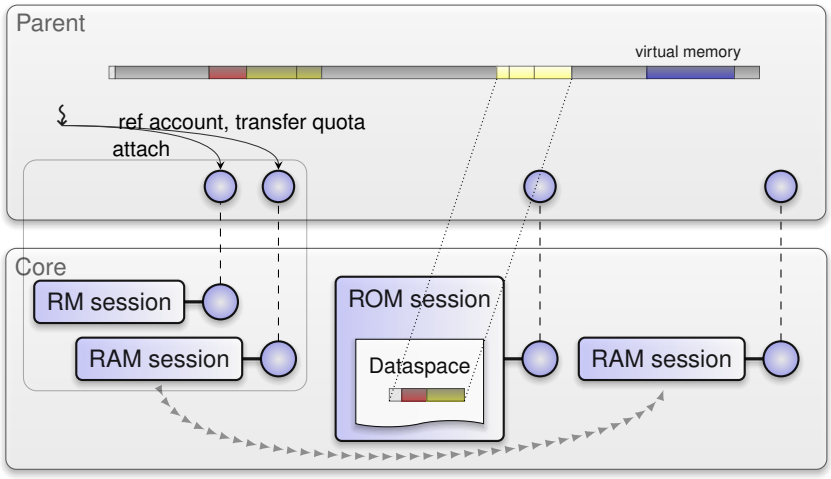



**Figure 16:** The parent creates the RAM session of the new child and obtains the child's executable

the ELF header to determine the memory required 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 16 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 session to core's RM service (Section 3.4.5). Hence, as illustrated in Figure 17, the parent has to create an RM session designated for the child. When creating the session, the parent is able to constrain the bounds of the virtual address space. By default, the first page 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 RM session is empty. It is up to the parent to populate the virtual address space with meaningful information by attaching dataspaces to the RM session. 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

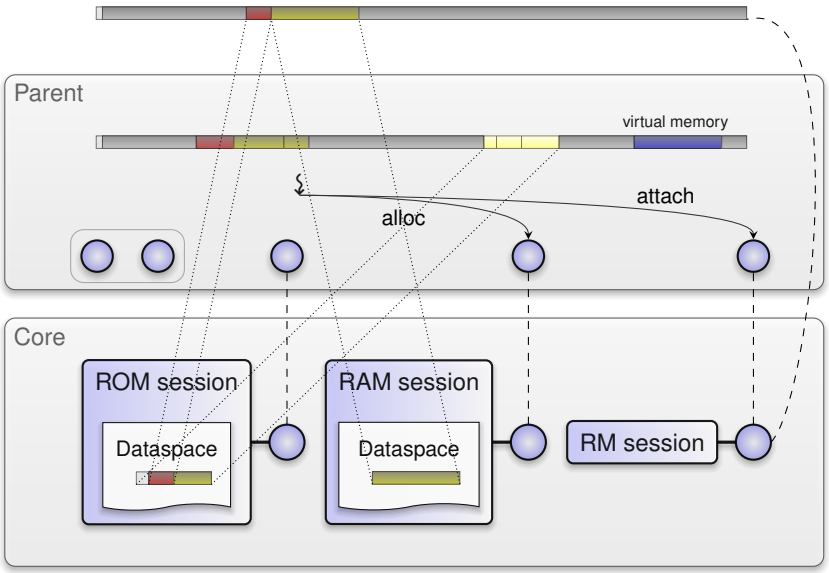



**Figure 17:** The parent creates and populates the virtual address space of the child using a new RM session (the parent's RM and RAM sessions are not depicted for brevity)

by invoking the attach operation on the child's RM-session capability. By attaching a portion of the existing ELF dataspace to the new child's RM session, 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 used kernel. 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 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 fresh 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 RM session with the RAM dataspace capability as argument.
3. Copying of the segment content from the ELF binary's dataspace to the fresh 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 RM session.
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's RM session 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 and the child's protection domain

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 RM sessions, the parent creates a CPU session (Section 3.4.6) 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 constrains. Once constructed, the CPU session can be used to create new threads by invoking the session's create-thread operation. 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. In particular, the thread needs to be associated with its address space. Otherwise, the kernel respectively core would not know how to respond to page faults triggered by the thread. To associate the thread with the virtual address space of the new child, the parent invokes the add-client operation at the child's RM session with the thread capability as argument.

The final step is the creation of the child's protection domain and the assignment of the child's initial thread to the new protection domain. A protection domain is represented by a session to core's PD service. As described in Section 3.2.1, each protection

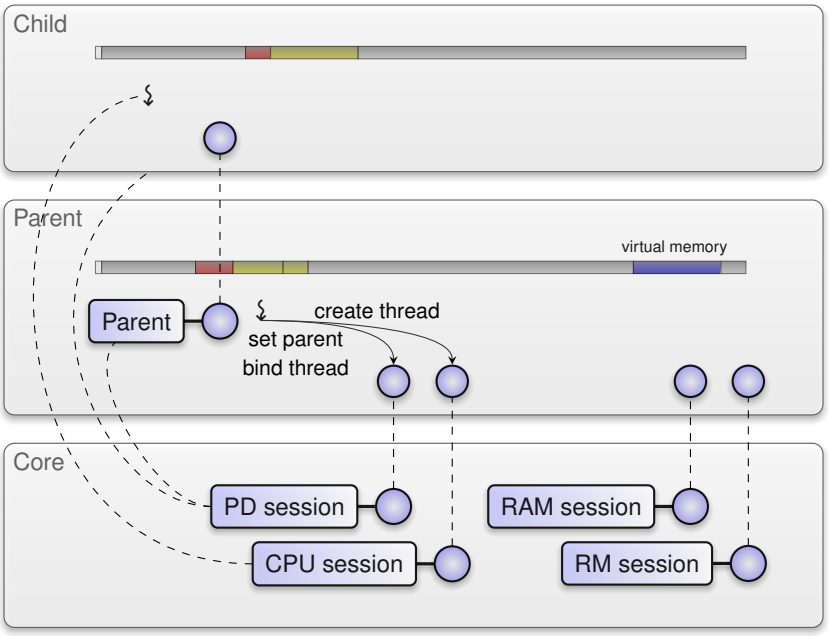



**Figure 18:** Creation of the child's protection domain and initial thread

domain has initially a single capability installed, which allows the child to communicate to its parent. Right after the creation of the PD for a new child, the parent can register a capability to a locally implemented RPC object as parent capability for the PD session. Furthermore, the parent binds the initial thread to the child's protection domain using the PD-session interface. Now that the initial thread has a defined virtual address space and a protection domain, it is the right time to kick off the execution of the thread using the start operation of the CPU session with the thread capability and the initial program counter as arguments. The latter argument corresponds to the program's entry-point address as found in the ELF header of the child's executable binary. Figure 18 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 initially loading the ELF binary, the parent has detached those dataspaces from its own RM session.

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 RM, RAM, CPU, and PD sessions from its parent. With the Genode environment in place, the startup code initializes the thread-context area, sets up the real stack for the main thread within the thread-context 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 executes global constructors before calling the program's main function. Section 8.1 describes the component-local startup procedure in detail.

## 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 the 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.

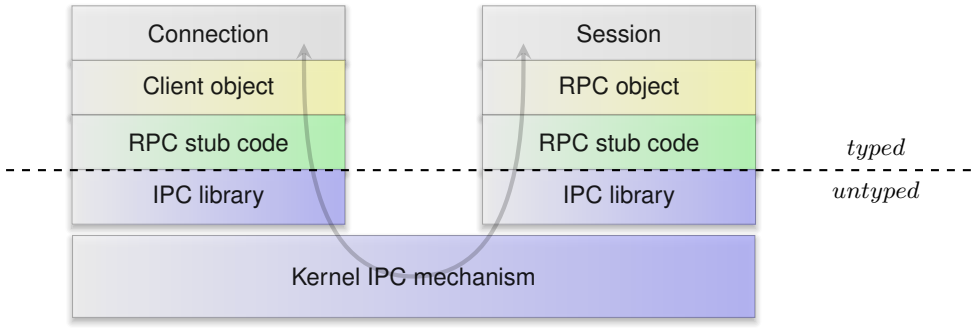



**Figure 19:** Layered architecture of the RPC mechanism

### 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 19 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 8 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 it 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.

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 used kernel.

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 *IPC client* represents the calling side of the communication channel to a given destination capability. It uses two message buffers, a send buffer for the arguments to be sent to the server and a receive buffer for the results delivered as a reply by the server. The user of an IPC client object can populate the send buffer with raw data and capabilities using the C++ insertion operator, invoke the kernel's call operation, and obtain the results by using the C++ extraction operator. The kernel's call operation blocks the execution of the IPC client until the server replied to the call.

The *IPC server* represents the callee side of the communication with potentially many IPC clients. Analogously to the IPC client, it uses two message buffers, a receive buffer for incoming requests and a send buffer for delivering the reply of the last request. Each IPC server has a corresponding untyped capability that can be used to perform calls to the server using an IPC client object. An IPC server object can be used to wait for incoming messages, read the arguments from the receive buffer using the C++ extraction operator, populate the send buffer with the reply message, and submit the reply to the client. The IPC 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 the compiler.

The client-side stub code translates C++ method calls to a sequence of operations on an IPC client object. 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 compositions. 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 19, 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 20 shows the inheritance graph of Genode's most fundamental capability types.

**Server-side class structure** Figure 21 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 the local name of the invoked object identity, the entrpoint 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.

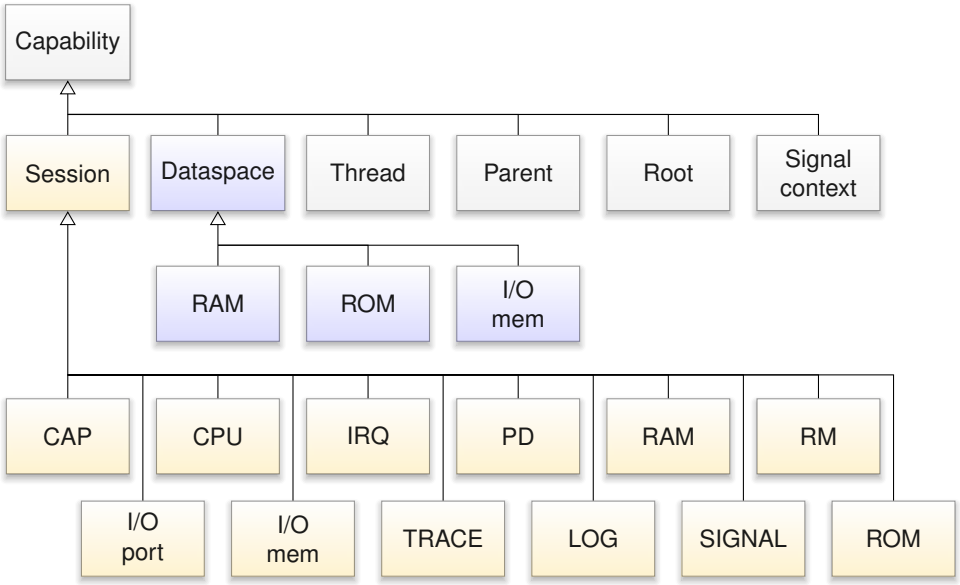



**Figure 20:** Fundamental capability types

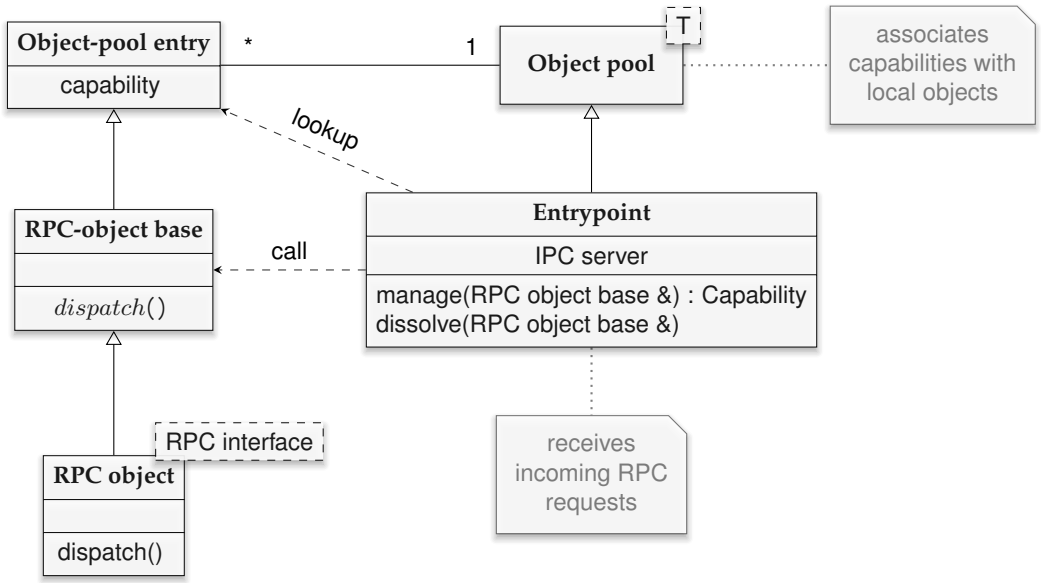



**Figure 21:** Server-side structure of the RPC mechanism

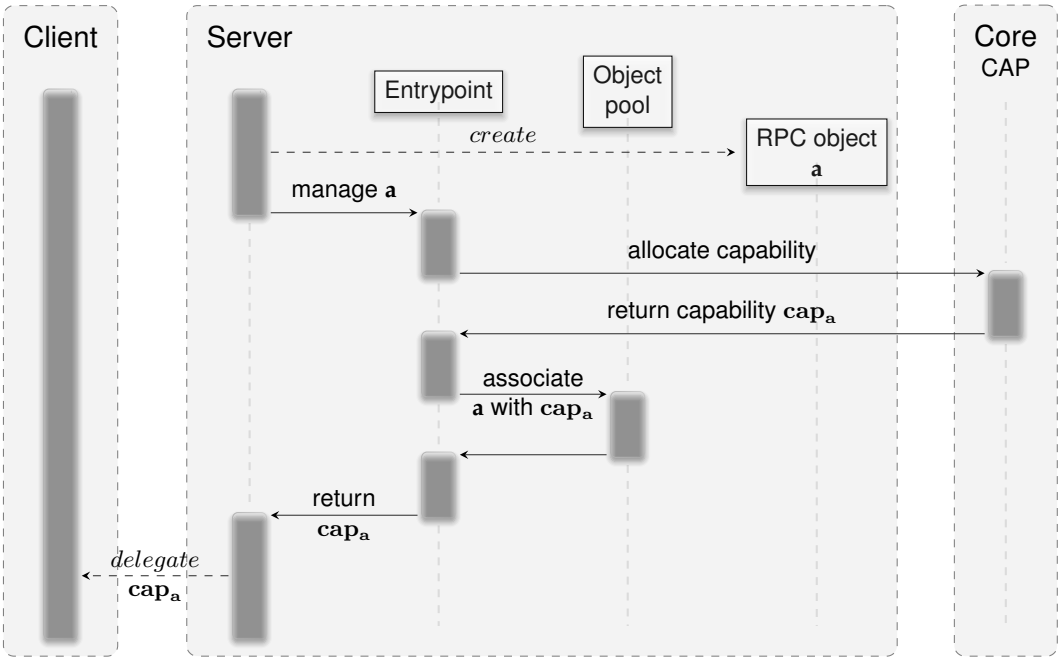



**Figure 22:** Creation of a new RPC object

**RPC-object creation** Figure 22 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 CAP service (Section 3.4.7). For allocating the new object identity, the entrypoint specifies the untyped capability of its IPC server as argument. Core's CAP service returns the new object identity in the form of a new capability that is derived from the specified capability. When invoked, the derived capability refers to the same IPC server as the original capability. But it represents a distinct object identity. The IPC server retrieves the local name of this object identity when called via the

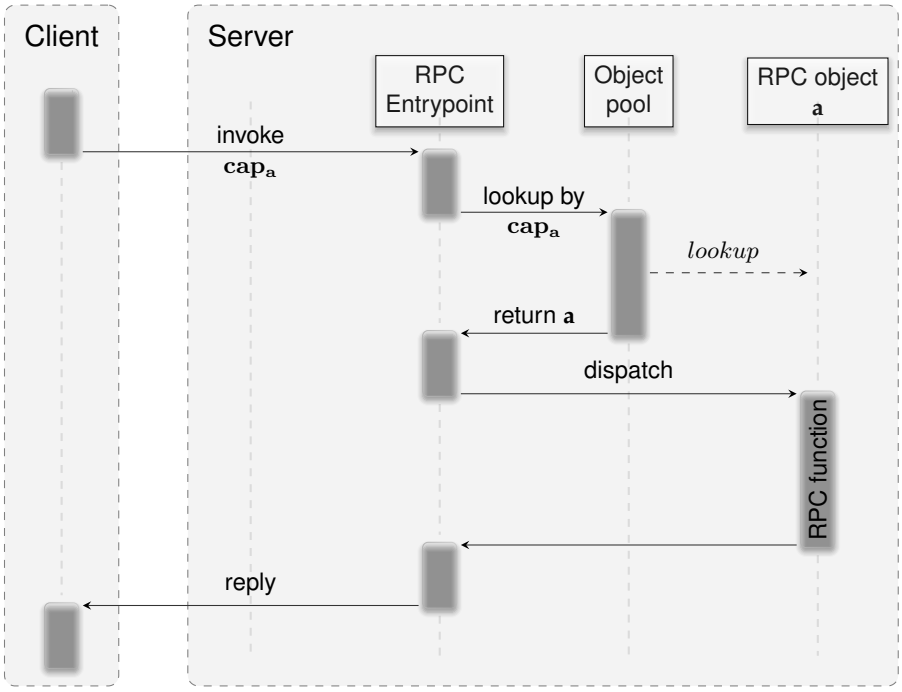



**Figure 23:** Invocation of an RPC object

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 received the authority to call the RPC object.

**RPC-object invocation** Figure 23 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 that belongs to the invoked capability and wakes up the corresponding entrypoint. On reception of the request, the entrypoint 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 could be found, the entrypoint calls the RPC object's dispatch function. This function is implemented by the server-side stub code. It converts 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 operation 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.

### 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 overheads.

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 24 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.

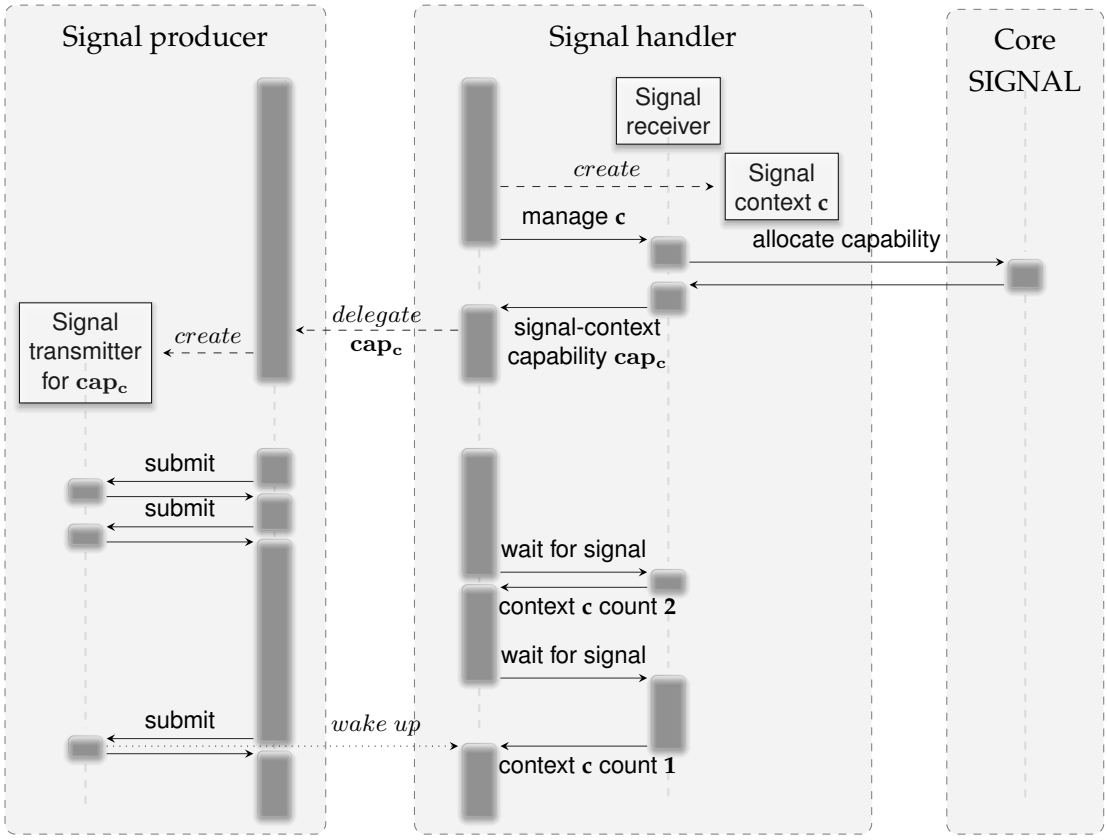



**Figure 24:** Interplay between signal producer and signal handler

**Signal-context creation and delegation** The upper part of Figure 24 depicts the steps needed by a signal handler to become able to receive asynchronous notifications.

1. Initially, the signal-handler component creates a so-called *signal receiver*. Similar to an entrypoint that can respond to RPC requests for a number of RPC objects, a signal receiver is able to receive signals for an arbitrary number of 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 the signal receiver via the manage function. Analogous to the way how RPC objects are associated with entrypoints, the manage function returns a capability for the signal context. Under the hood, the signal receiver uses core's SIGNAL service to create this kind of capability.

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 24 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* function. This function 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 signals, the signal handler is not yet ready to handle them. It is still busy with other things. However, the number of submitted signals is recorded. Once the signal handler finally calls the wait-for-signal function at the signal receiver, the call immediately returns the information about the context, to which the signals refer, and the number of signals that were submitted for this context.

3. After handling the first batch of signals, the signal handler component blocks its execution by calling the signal receiver's wait-for-signal function again. This time, no signals are immediately pending. After a while, however, the signal producer submits another signal, which eventually wakes up the signal handler with the information about the associated context.

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 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 payload. However, the number of signals is recorded, which does not require a buffer but merely a counter. Note that this counter could overflow in the presence of a producer that constantly submits signals.

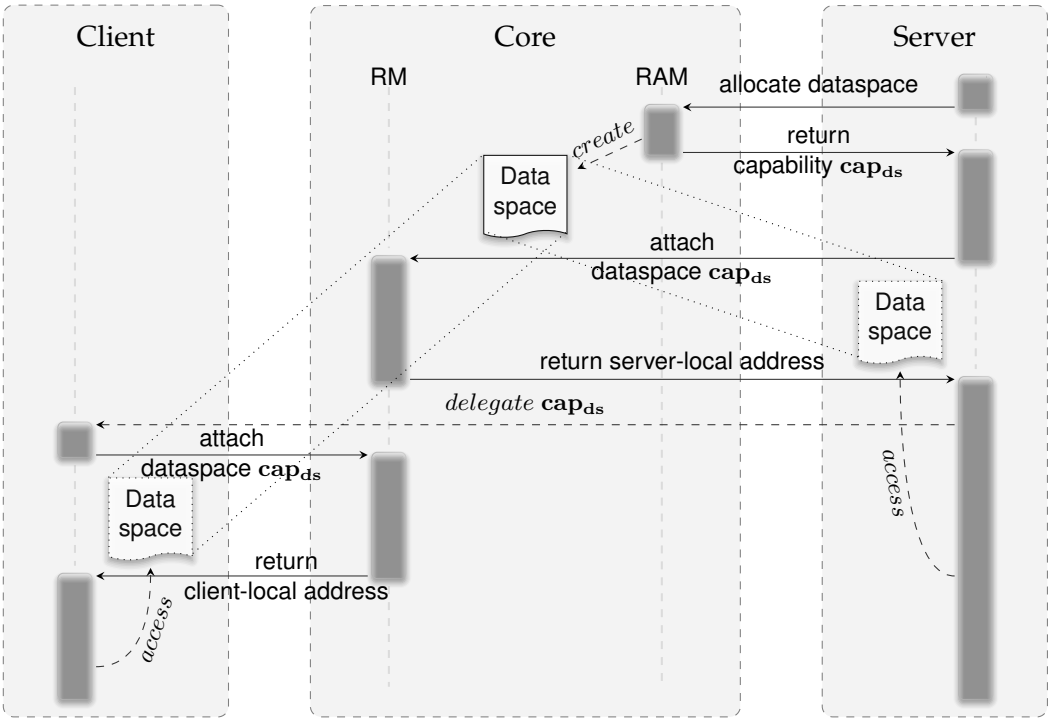



**Figure 25:** Establishing shared memory between client and server. The server interacts with core's RAM service. Both client and server interact with core's RM service.

### 3.6.3 Shared memory

By sharing memory between components, large amounts of information can be propagated across protection-domain boundaries without the active help 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 25 answers these questions with the sequence of establishing shared memory 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.

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 the first glance, this is seemingly a contradiction with 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 its RM 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 RM session, core's RM 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 savely access the dataspace content by reading or writing 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 its own RM session to make the dataspace content visible in its address space. Note that even though both client and server use core's RM 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 RM session, 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 of the client session. When the server frees the dataspace at its RAM session, core implicitly detaches the dataspace from all RM sessions. Thereby the dataspace will become inaccessible by the client.

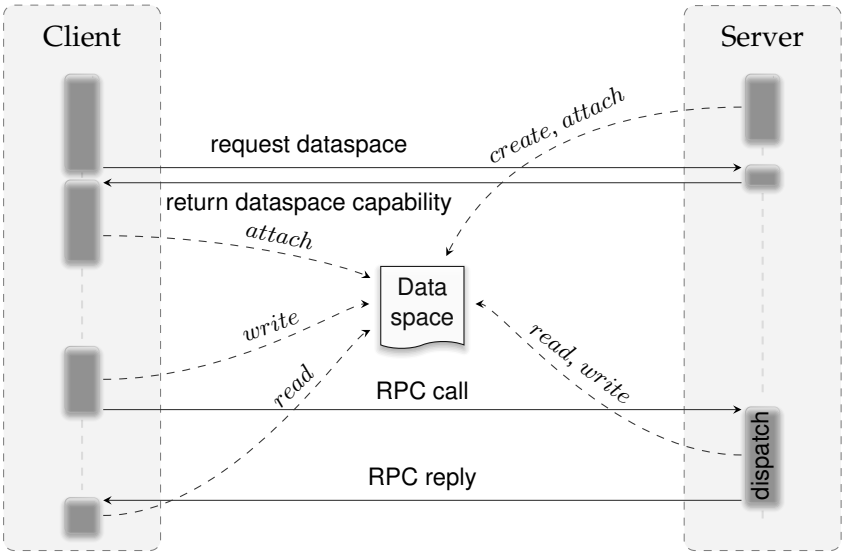



**Figure 26:** Transferring bulk data by combining synchronous RPC with shared memory

### 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 26 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 per-

formed 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 over 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 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 as 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 finished the processing of the data and wrote the results to the dataspace, it replies to the RPC. Thereby, it hands over the role as the user of the dataspace back 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.

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.

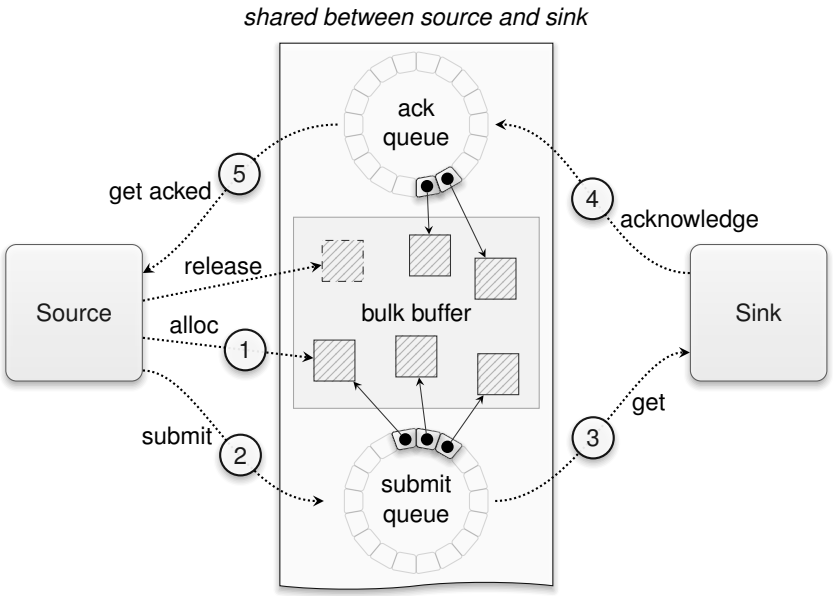



**Figure 27:** Life cycle of a data packet transmitted over the packet-stream interface

### 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 27, 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 source. At the points 3 and 4, the packet is owned by the sink. Putting a packet descriptor in the submit queue 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 an *ready-to-ack* signal. On reception of the signal, the sink wakes up and can proceed 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:

**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 operations. 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 independently 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.

# 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 designed 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 critical for performance, such a split should be clearly motivated by a benefit for security.

**Reusability** Componentization can be pursued for improved reusability while sometimes disregarding performance considerations. 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, the 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 is 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. Those 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 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.

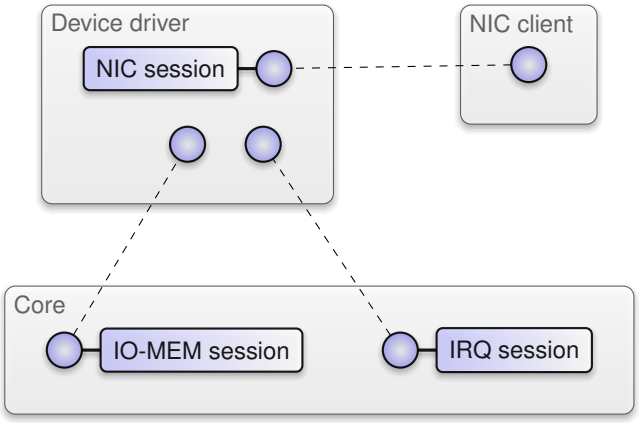



**Figure 28:** A network device driver provides a NIC service to a single client and uses core's IO-MEM and IRQ services to interact with the physical network adaptor.

## 4.1 Device drivers

A device driver translates a device interface to a Genode session interface. Figure 28 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 interrupts, and a means to transfer data from and to the device. A device-driver component 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 eventually become inconsistent. A device driver should not attempt to multiplex 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 this policy information from its configuration as provided by the driver's parent.

### 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 speak directly to the PCI configuration interface, each driver would need to carry with it the functionality to interact with PCI.

**Problem 2: of 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 tables of the Advanced Configuration and Power Interface (ACPI). 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 the 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.

**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 played 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 targeting the physical memory directly and not involving the CPU altogether, such requests are naturally not subjected by the virtual-memory mechanism implemented in the CPU in the form of a memory-management unit (MMU). Figure 29 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 reads a block from the disk, the driver 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 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

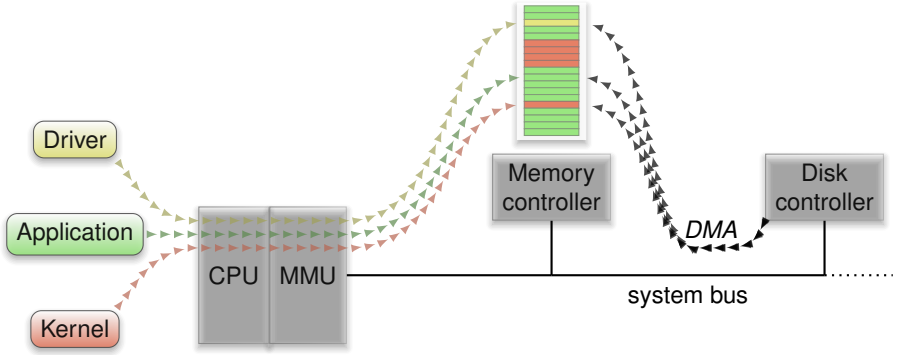



**Figure 29:** The MMU restricts the access of physical memory pages by different components according to their virtual address spaces. However, direct memory accesses issued by the disk controller are not subjected to the MMU. The disk controller can access the entirety of the physical memory present in the system.

the device, the 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 executing the device driver and providing a driver interface to the Genode world. In both cases the guest OS should not be considered as trustworthy. In contrary, it bears the risk to subvert the isolation between components. A misbehaving guest OS could issue DMA requests referring to the physical memory used by other components and 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 except for the device vendor. Hidden functionality or vulnerabilities might be present in it. By the means of DMA transactions, such firmware has unlimited access on 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 misused a DMA-capable device as a proxy to drive an attack. However, the system bus can be attacked directly with no hardware tinkering needed. There are ready-to-exploit interfaces that are fea-

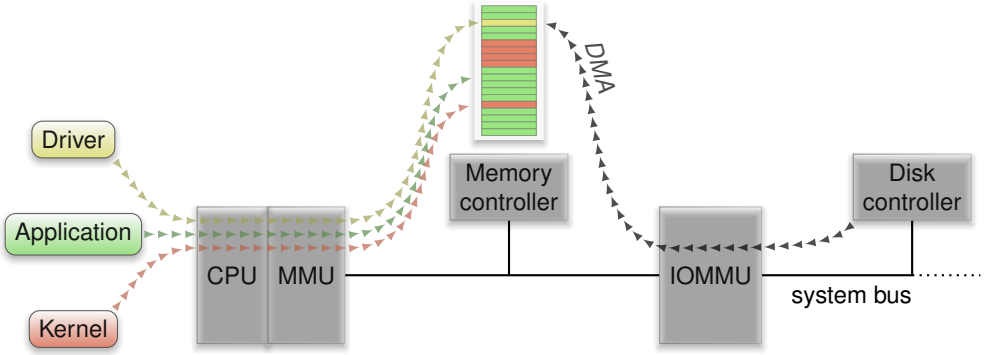



**Figure 30:** An IOMMU arbitrates and virtualizes DMA accesses issued by a device to the RAM. Only if a valid IOMMU mapping exists for a given DMA access, the memory access is performed.

tured on most PC systems. For example, most laptops come with PCMCIA / ExpressCard 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 favour 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 applies.

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 30, the IOMMU sits between the physical memory and the system bus where the devices are attached to. So each DMA request has to pass the IOMMU, which is not only able to arbitrate the access of DMA 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 8.7.7 discusses the use of IOMMUs in more depth.

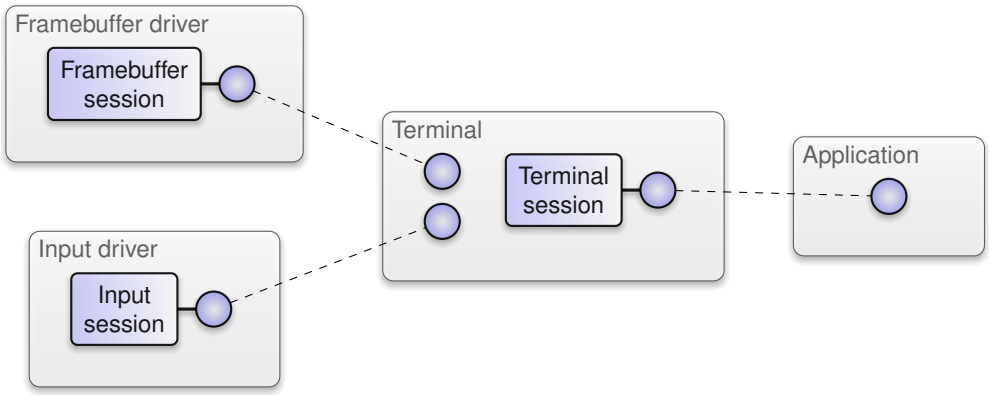



**Figure 31:** Example of a protocol stack. The terminal provides the translation between the terminal-session interface (on the right) and the driver interfaces (on the left).

## 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 31 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 and 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 has 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 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.

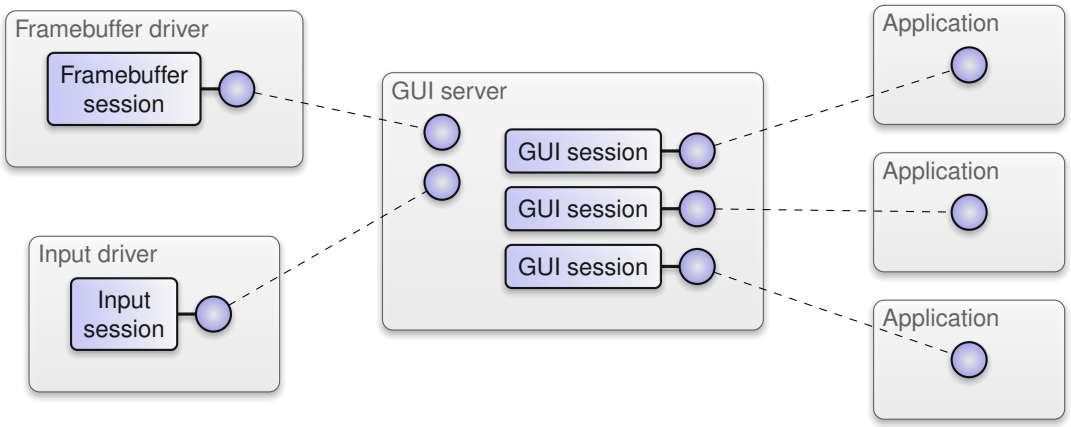



**Figure 32:** A GUI server multiplexes the physical framebuffer and input devices among multiple applications.

## 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 32, 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 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 high risk for bugs. Such bugs may, in turn, result in the unintended flow of information between clients or spoil the quality of service for all clients. Hence, resource multiplexers must be as low complex 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 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 the 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.

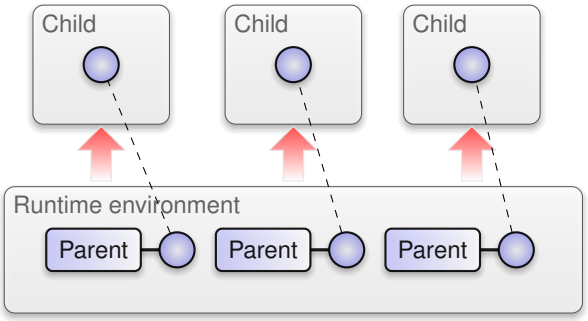



**Figure 33:** A runtime environment manages multiple child components.

## 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 33 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 be 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.

**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 including those normally provided by core such as RAM (Section 3.4.2), RM (Section 3.4.5), and CPU (Section 3.4.6). 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 wrapping 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 outside, such a multi-component application appears as leaf node of the component tree but internally, it employs a further level of componentization by executing portions of its functionality in separate child components. The primary incentive behind this approach 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.

## 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 specified name. 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 client's RM 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 contrary to the intuitive assumption that read-only data is constant, 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. It rather maintains the new version separately and informs the client by submitting a signal to the client's signal handler.
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 in 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 has fitted in 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. On reception of the dataspace request, the server destroys the original dataspace (thereby making it invisible at the client), and returns the new version of the ROM module as a different dataspace.

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

The protocol is designed such 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.

- Components obtain their configuration by requesting a ROM session for the ROM module "config" at the 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 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 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 the 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 for textual user interfaces but may also be used to transfer other serial steams 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 does never block. 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 register a signal handler that gets notified on 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 (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 this would render the server's entry point unavailable for all other clients until the TCP connection is ready. Instead, the server delivers a `connected` signal to the client. 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.
- Graphical terminal.
- Transfer of steams of data over TCP (using TCP terminal).
- Writing streams of data to a file (using 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 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:

**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 combination 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).

**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 returning the according 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.

### 4.5.6 Nitpicker GUI

The nitpicker session interface subsumes an input session and a framebuffer session as a single session (Figure 34). Furthermore it supplements the framebuffer session with the notion of views, which allows the creation of flexible multi-window user interfaces. It 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 from of session quota. Clients may request the screen mode of the physical framebuffer and are able to register a signal handler for

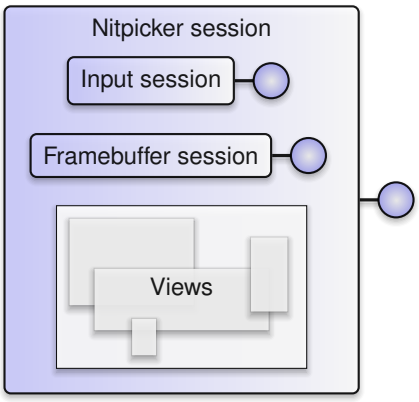



**Figure 34:** A nitpicker session aggregates a framebuffer session, an input session, and a session-local view stack.

mode changes of the physical framebuffer. This way, nitpicker clients are able to adapt themselves to changed 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 between multiple GUI applications.
- 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.

### 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 prior registered 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 needed.

**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, 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 35 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.

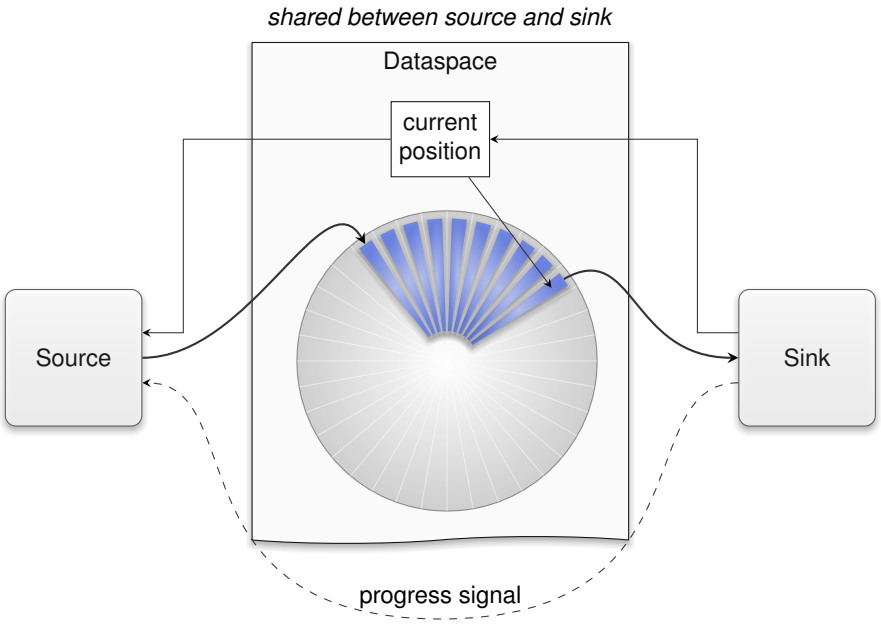



**Figure 35:** The time-driven audio-out session interface uses shared memory to transfer audio frames and propagate progress information.

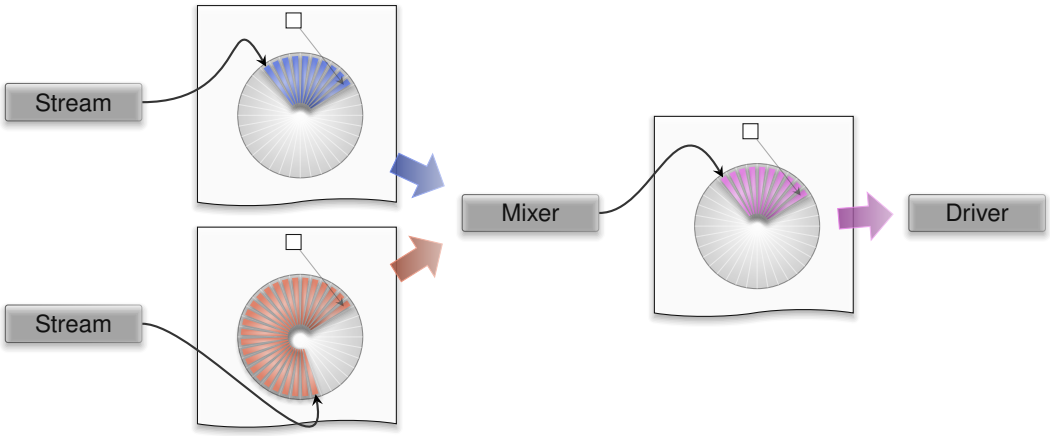



**Figure 36:** The mixer processes batches of incoming audio frames from multiple sources.

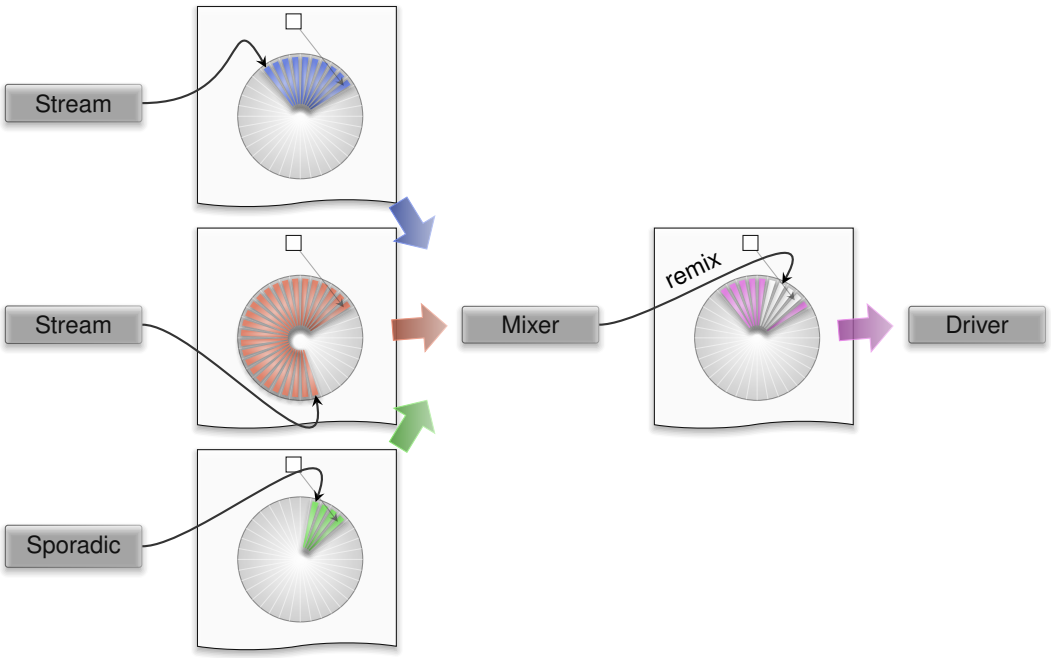



**Figure 37:** A sporadic occurring sound prompts the mixer to remix packets that were already submitted in the output queue.

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 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.

### 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).

## 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 providing 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. Usually, the server selects the policy where the session label starts with the policy's label. If multiple policies match, the one with the longest (most specific) policy label is selected. If multiple policies have the same label, the selection is undefined. This is a configuration error.

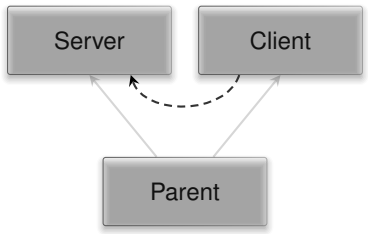
### 4.6.3 Dynamic component reconfiguration at runtime

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.

## 4.7 Component compositions

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 provides.

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 38 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 video and audio data. Instead of linking the media-codec library (libav) directly to

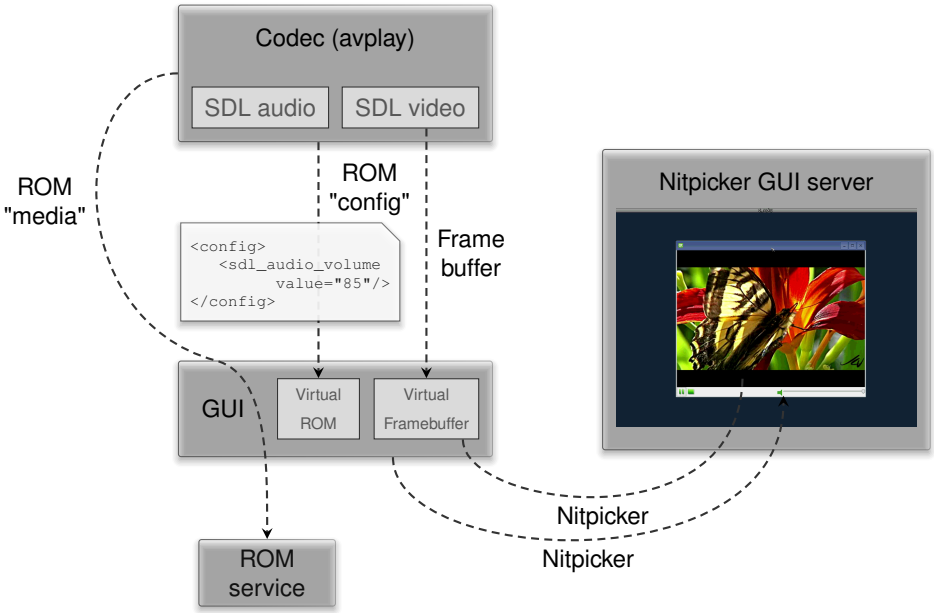



**Figure 38:** A video player executes the video and audio codecs inside a dedicated sandbox.

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 backend 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, it opens an input session (Section 4.5.4). Furthermore, it opens a ROM session for obtaining a configuration. This configuration parametrizes the audio backend 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 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 the replay, 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 backend picks up the changed configuration parameters and ajusts 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 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 go even a step further by providing the all-encompassing virtualization of all services used by the child, including core's services such as RAM, RM, 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 RM service.

The noux runtime illustrated in Figure 39 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.
- Any Unix process can allocate memory as needed. There is no explicit assignment of memory resources to Unix processes needed.

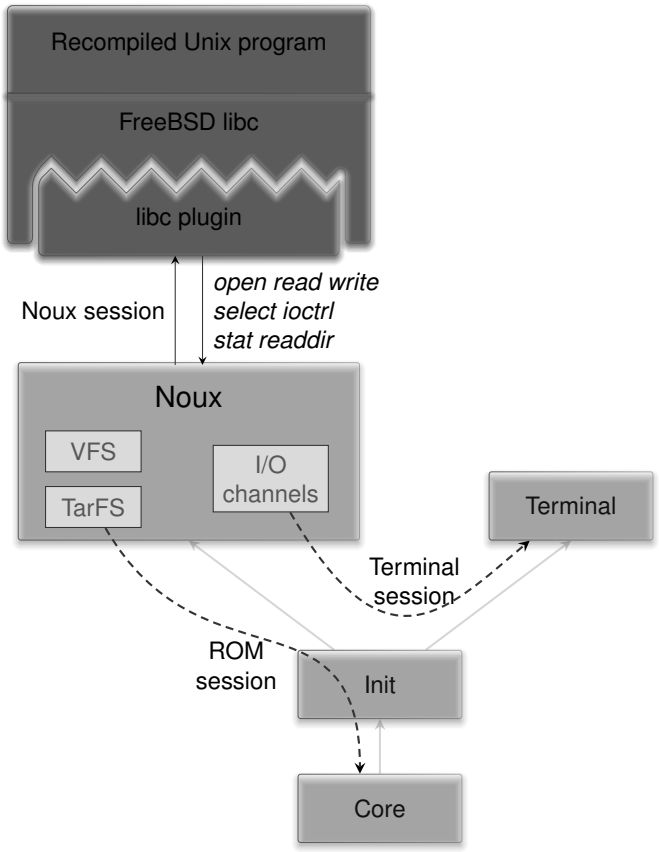



**Figure 39:** The Noux runtime provides a Unix-like interface to its children.

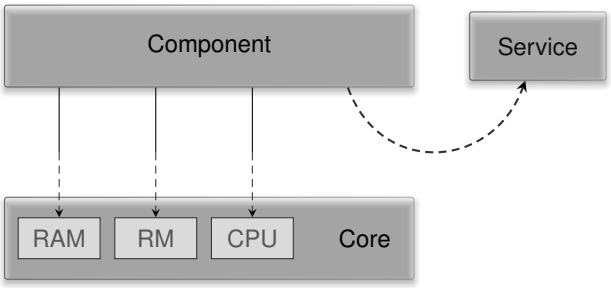



**Figure 40:** Each Genode component is created out of basic resources provided by core.

- 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 those 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 RM service, it is able to replay the address-space layout of an existing process to a new process when the process forks.

**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 a created out of low-level resources in the form of sessions provided by core. Those sessions include at least a RAM sesion, a CPU session, and an RM session as depicted in Figure 40. In addition to those low-level sessions, the component may interact with sessions provided by other components.

For debugging the 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 complexity 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 use to exclude debugging features in production builds altogether.

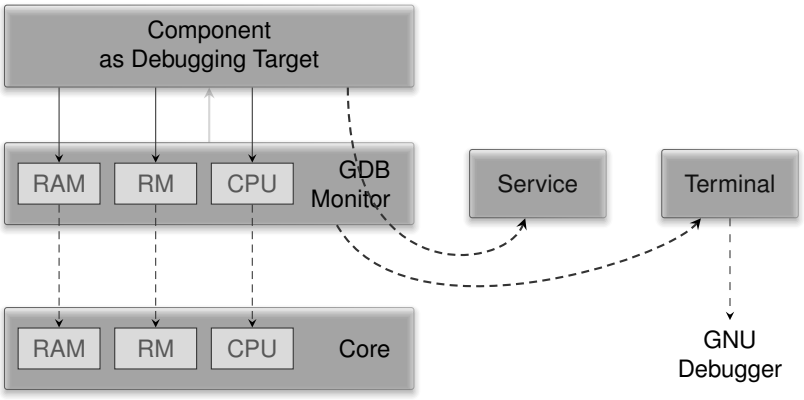



**Figure 41:** By intercepting all sessions to core's services, a debug monitor obtains insights into the internal state of its child component. The debug monitor, in turn, is controlled from a remote debugger.

In a Genode system, the component's internal state is represented in the form of core sessions. Hence, by intercepting those sessions for a child, a parent can monitor all interactions of the child with core and thereby record the child's internal state. Figure 41 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 42 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.

### 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 critical for security, complexity must be avoided. Functionality is added to such components only if it cannot be provided outside the component.

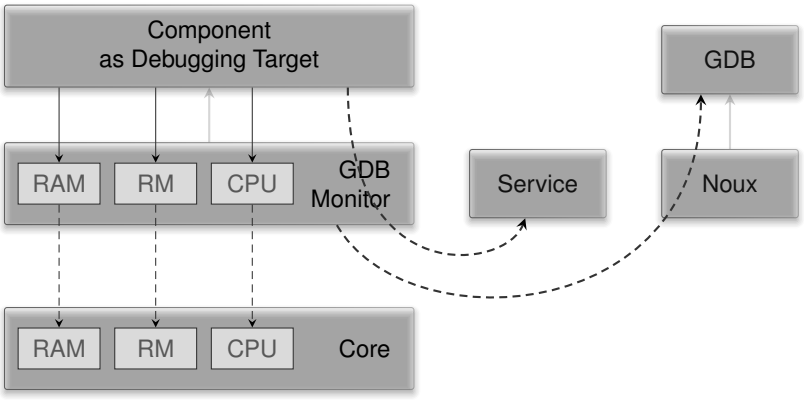



**Figure 42:** The GNU debugger is executed within a dedicated noux instance, thereby providing an on-target debugging facility.

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 RM service is used to manage the virtual address spaces of components. When a dataspace is attached to an RM session, the RM service 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 with the common wisdom that address spaces shall be randomized. Hence core's RM 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 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

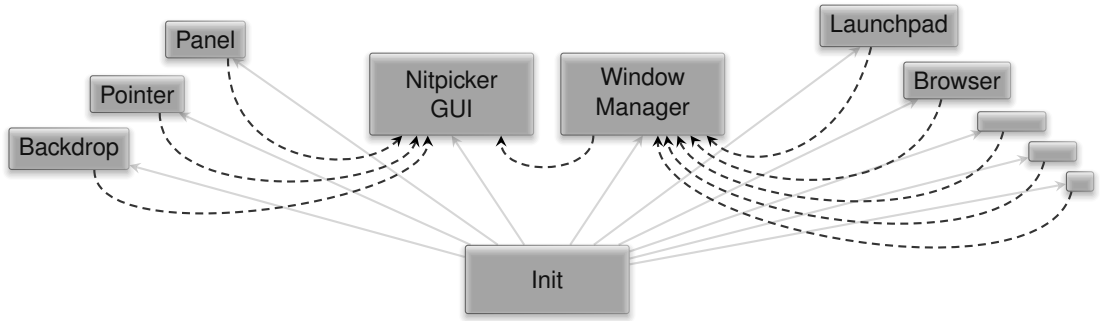



**Figure 43:** The nitpicker GUI accompanied with a window manager that interposes the nitpicker session interface for the applications on the right. The applications on the left are still able to use nitpicker directly and thereby avoid the complexity added by the window manager.

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 interposing their sessions. Figure 43 shows a window manager implemented as a separate component outside of nitpicker. Both the nitpicker GUI server and the window manager 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 RM service. Outward-facing components can benefit from the security feature by having their RM 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 becomes a security risk. A shell can be a text-command interpreter, a graphical desktop shell, a web browser that launches subsystems as plugins, or a web server that provides a remote administration interface. All those kinds of shells have in common that they contain an enormous amount of complexity that is attributed to convenience. 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 speak about using a web browser as a shell. Unfortunately, the functionality provided by these programs cannot be dismissed as it is expected by

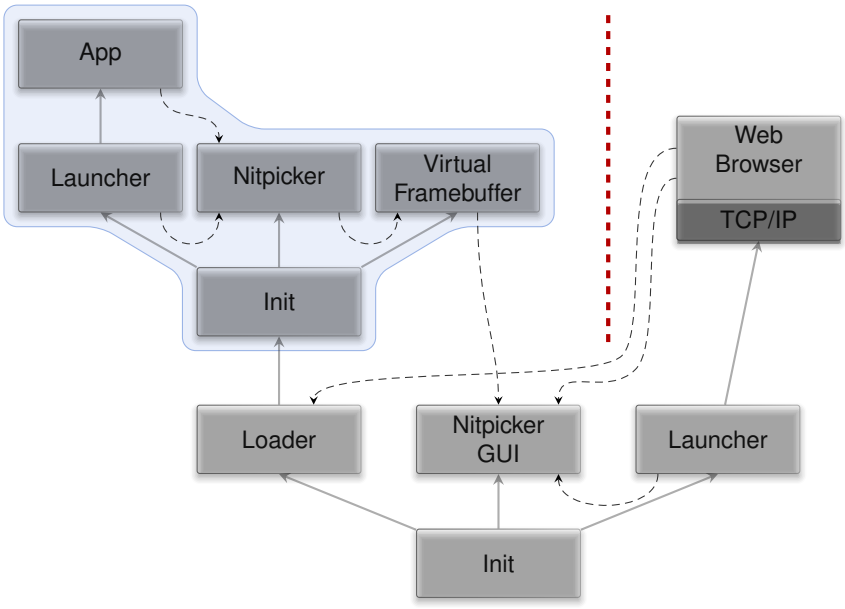



**Figure 44:** A web browser spawns a plugin by ceding the parenthood of the plugin to the trusted loader service.

the user. But the high complexity of the convenience functions fundamentally contradicts with the security-critical role of the shell as the common parent of all spawned subsystems. If the shell gets compromised, all the spawned subsystems 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 44 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 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 trans-

fers 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 by 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 likes 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).

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. So 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 an 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 could 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 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 had 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 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.

# 5 Development

*TODO*

## 5.1 Work flow

*TODO*

## 5.2 Tool chain

*TODO*

## 5.3 Build system

*TODO*

## 5.4 Ports of 3rd-party software

*TODO*

## 5.5 Run tool

*TODO*

## 5.6 Automated tests

*TODO*

## 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 would interpret 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 steered 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.

**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 its construction time.

```
<config>
  <parent-provides> ... </parent-provides>
  <default-route> ... </default-route>
  ...
  <start name="nitpicker">
    ...
  </start>
  <start name="launchpad">
    ...
    <config>
      <launcher name="L4Linux">
        <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 45:** 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 an XML format. To obtain its configuration, init requests a ROM session for the ROM module "config" from its parent, which is core. Figure 45 shows an example of such a config ROM module.

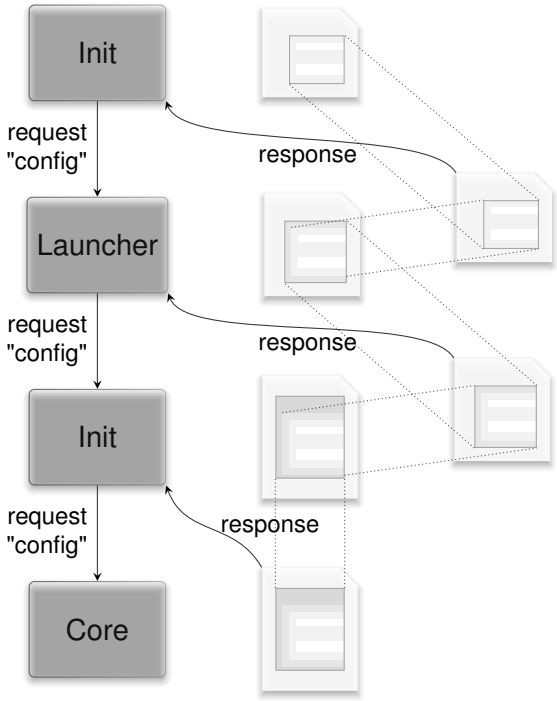



**Figure 46:** Successive interception of "config" ROM requests

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 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, which is init. 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 46 shows the successive interposing of "config" ROM requests according to the example configuration given in Figure 45. 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.

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 be-

tween client and server of a ROM session. This way, the configuration of long-living components can be dynamically changed.

## 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 further 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, the parent is up 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 requested service 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 concept is best illustrated by an example:

```
<config>
  <parent-provides>
    <service name="CAP"/>
    <service name="LOG"/>
    <service name="SIGNAL"/>
  </parent-provides>
  <start name="timer">
    <resource name="RAM" quantum="1M"/>
    <provides> <service name="Timer"/> </provides>
    <route>
      <service name="CAP">    <parent/> </service>
      <service name="SIGNAL"> <parent/> </service>
    </route>
  </start>
  <start name="test-timer">
    <resource name="RAM" quantum="1M"/>
    <route>
      <service name="Timer">  <child name="timer"/> </service>
      <service name="LOG">    <parent/>             </service>
      <service name="SIGNAL"> <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 CAP service, a LOG service, and a SIGNAL service. For each child to start, there is a `<start>` node describing resource assignments, declaring 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. To implement this service, the timer requires a CAP session. The routing table defines that CAP session requests are delegated to init's parent. The second process 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.

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="RM">     <parent/> </service>
  <service name="PD">     <parent/> </service>
  <service name="CPU">    <parent/> </service>
  <service name="SIGNAL"> <parent/> </service>
</route>
```

the following shortcut 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>
```

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.

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. The 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 sum of the quotas assigned to the children. 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:

```
<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.

```
<config>
  <parent-provides>
    <service name="CAP"/>
    <service name="LOG"/>
    <service name="ROM"/>
    <service name="RAM"/>
    <service name="CPU"/>
    <service name="RM"/>
    <service name="PD"/>
    <service name="SIGNAL"/>
  </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"/>
        <service name="SIGNAL"/>
      </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, RM, 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.

Alternatively to specifying all nested configurations in a single configuration, sub configurations can be placed in a separate ROM modules specified via the `configfile` node. For example:

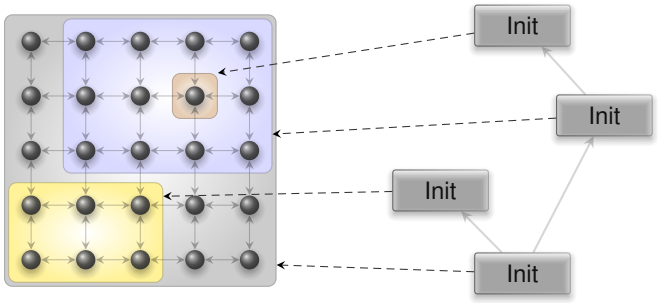



**Figure 47:** Successive virtualization of CPU affinity spaces by nested instances of init

```
<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 to apply 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 47 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 sub systems 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:

```
<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 various status information as LOG output. To enable the verbose mode, assign the value "yes" to the verbose attribute of the <config> node.

### 6.2.9 Executing children in chroot environments on Linux

On the Linux base platform, each process started by init can be assigned to a chroot environment by specifying the new root location as root attribute to the corresponding <start> node. Root environments can be nested. The root path of a nested init instance will be appended to the root path of the outer instance.

When using the chroot mechanism, core will mirror the current working directory within the chroot environment via a bind-mount operation. This step is needed to enable the execve system call to obtain the ELF binary of the new process.

In order to use the chroot mechanism when starting Genode's core as a non-root user process, the core executable must be equipped with the CAP_SYS_ADMIN and

CAP_SYS_CHROOT capabilities. CAP_SYS_ADMIN is needed for bind mounting. CAP_SYS_CHROOT is needed to perform the `chroot` syscall:

```
sudo setcap cap_sys_admin,cap_sys_chroot=ep core
```

For an example of using chroot, please refer to the *os/run/chroot.run* script.

# 7 Functional specification

*TODO*

## 7.1 Parent-child interaction

- Parent interface
- Root interface

## 7.2 Fundamental data structures

- Basic types, structured types, allocators
- Rationale behind using intrusive data structures

## 7.3 XML processing

- Parsing
- Generation

## 7.4 Process execution environment

## 7.5 Remote procedure calls

## 7.6 Signals

## 7.7 Multi-threading and synchronization

## 7.8 Process management

## 7.9 Common utilities

## 7.10 Server API

## 7.11 Support for user-level device drivers

## 7.12 Tracing

## 7.13 C runtime

# 8 Under the hood

This chapter gives insights into the inner functioning of the Genode OS framework. In particular, it explains how the concepts explained in Chapter 3 are realized on different kernels and hardware platforms.

## 8.1 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 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.

### 8.1.1 Linker scripts

Under the hood, the Genode build system uses three different linker scripts located at *repos/base/src/platform/*:

**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 8.5).

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 constructing the list of constructors and destructors differ between the 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.

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 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 8.1.2 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 `_setup_elf` function in *base/src/base/process/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.

### 8.1.2 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 *repos/base/platform/<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 8.3.1. 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 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 needed 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 *repos/base/src/platform/init_main_thread.cc* 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/platform/init_main_thread.cc*, the assembly code triggers the execution of all the steps needed for the creation 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/src/platform/init_main_thread.cc*) for setting up the real stack for the program. For placing a stack in 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 RM session from its parent. This comes down to the need for performing RPC calls. First, for requesting the RM and RAM session capabilities from the parent, and second, for invoking the session capabilities to perform the RAM allocation and RM 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 perform 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 dependencies 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 (by calling `Genode::env()`), the program requests its own RM, RAM, CPU, and PD sessions from its parent, and initializes its heap (`env()→heap()`).

With the environment constructed, the program is able to interact with its own RM and RAM sessions and can principally realize the initialization of the real stack. However, instead of merely allocating a new RAM dataspace and attaching the dataspace to the RM session, a so-called thread-context area is constructed. The thread-context area is a secondary RM session that is attached as a dataspace to the component's actual RM session (See the description of managed dataspaces in Section 3.4.5). This way, virtual-memory allocations within the thread-context area can be managed manually. I.e., the spaces between the stacks of different threads are guaranteed to remain free from any attached dataspaces. For constructing the thread-context area, a new RM session is created (*repos/base/src/base/context_area.cc*).

**Component-dependent startup code** With the Genode environment constructed and the initial stack switched to a proper stack located in the thread-context area, the component-dependent startup code of the `_main` in *repos/base/src/platform/_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.

## 8.2 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.

### 8.2.1 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 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-systems 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.

   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 construc-

tion, 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 previous step as 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 if 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.

### 8.2.2 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/base/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.

## 8.3 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. Those resources are the 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 usermode, 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.

### 8.3.1 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 as 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.

**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 has the same size as the virtual-memory range
4. The virtual memory range is 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 allocators 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 the behalf of core clients because allocations could not be reverted when closing the 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.

### 8.3.2 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 whereas others provide page tables as kernel objects. Whereas some kernels represent scheduling parameters as distinct kernel objects, others subsume scheduling 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 was 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.

### 8.3.3 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 entrypoint 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/rm_session_component.cc*), which represents the association of the pager object with its virtual address space (RM session). Given the context information about the RM session of the thread, the `pager` function looks up the region within the RM session, 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 RM session 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 RM-fault signal to the RM session where the fault occurred. This way, the RM-session client has the chance to 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 RM session. As a response to the fault, the RM-session 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 RM faults is registered for the RM session, core prints a diagnostic message and blocks the faulting thread forever.

To optimize the TLB footprint and the use of kernel memory, the RM service does 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.

## 8.4 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 to 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 in the form of the SIGNAL 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 SIGNAL service using the *alloc_context* and *free_context* RPC functions. Upon the creation of a signal context, the SIGNAL 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. In return to 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 SIGNAL 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 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 as 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 SIGNAL 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 infinitely and thereby make the server unavailable to all clients. In contrast to the untrusted signal-context capability, the SIGNAL 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 would 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 signal receiver associated with the signal and wake up the thread that called the `Signal_receiver::wait_for_signal` API function.

Both problems are solved by a dedicated thread that is spawned at the startup of the component. This signal thread blocks at core's SIGNAL service for incoming signals. The blocking operation is not directly performed on the SIGNAL session but on a decoupled RPC object called *signal source*. In contrast to the SIGNAL 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 SIGNAL 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::wait_for_signal` function at the API level.

## 8.5 Dynamic linker

The dynamic linker is a mechanism for loading ELF binaries that are dynamically-linked against shared libraries.

### 8.5.1 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 when the program is started. This invariant is enforced by the build system by stripping all base libraries from the linking step of a dynamically-linked programs. The base libraries are defined in *repos/base/mk/base-libs.mk*.

The entrypoint of a dynamically-linked program is the `main` function.

### 8.5.2 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/base/elf/*). 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 8.1.2. 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 8.1.2. 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 with the actual program. Once the linker has obtained the dataspace containing 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.

### 8.5.3 Address-space management

To load the binary and the associated shared libraries, the linker does not directly attach dataspaces to its RM session. Instead, it manages a dedicated part of the component's virtual address space manually by attaching a large managed dataspace to its RM session. 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 big-enough portions of virtual memory and populate it in multiple steps.

## 8.6 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.

### 8.6.1 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 library. The library description file is specific for each platform and located at *lib/mk/platform_<pf>/core.mk* where <*pf*> corresponds to the used hardware platform. It includes the platform-agnostic *repos/base-hw/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 header contains the name, core-local address, and size of a boot module. This meta data is used by core's initialization code in *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 while access to a boot module is needed 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.

Those 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 5.5) 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 8.1.2, 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).

### 8.6.2 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*.

### 8.6.3 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 the kernel exit, the control is passed to the user-level thread that corresponds to the head of the scheduler list.

### 8.6.4 Split kernel interface

The system-call interface of the base-hw kernel is split in 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 *include/kernel/interface.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 distinction between both parts of the kernel interface is enforced by the function `Thread::_call` in *src/core/kernel/thread.cc*.

### 8.6.5 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 SIGNAL service (Section 3.4.10). However, the creation of signal contexts is arbitrated by the SIGNAL service. This way, the kernel objects needed for the signalling mechanisms are accounted to the corresponding clients of the SIGNAL service.

### 8.6.6 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* and *bin_pd*), invoked by the PD service
- The creation, manipulation, and destruction of threads (*new_thread*, *bin_thread*, *start_thread*, *resume_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*, *bin_signal_receiver*, *new_signal_context*, and *bin_signal_context*), invoked by the SIGNAL service

### 8.6.7 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 means of CPU time or priority.

### 8.6.8 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.

### 8.6.9 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 respond to the IPI by taking the decision to schedule the receiving thread. As the IPI sender doesn't have to wait for an answer, the sending and receiving CPUs remain largely decoupled. There is no need for a complex IPI protocol between both.

**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 work, an IPI is sent to all other cores. If all cores completed the cleaning, the thread that invoked the TLB invalidation resumes its execution.

### 8.6.10 Asynchronous notifications on base-hw

The base-hw platform improves the mechanism described in Section 8.4 by introducing signal receivers and signal contexts as first-class kernel objects. Core's SIGNAL 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 by directly using the public kernel operations *await_signal*, *signal_pending*, *submit_signal*, and *ack_signal*.

### 8.6.11 Limitations of the base-hw platform

The base-hw kernel does not (yet) support the model of kernel-protected capabilities as described in Section 3.1. All kernel objects are referred to via global IDs. On this platform, a capability is represented as a tuple of a thread ID and a global object ID. Capability delegation is realized as a plain copy of those values. Since any thread can specify arbitrary global thread IDs and object IDs when performing RPC calls, capability-based security access control remains ineffective.

*Note that the lack of kernel-protected capabilities is not an inherent limitation of the design of base-hw but a temporary limitation due to the kernel's stage of its ongoing development.*

## 8.7 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 had 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

### 8.7.1 Integration of NOVA with Genode

The NOVA kernel is available via Genode's ports mechanism described in Section 5.4. 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 from 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 is only 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/*, *32bit/nova/*, and *64bit/nova/*.

### 8.7.2 Bootstrapping of a NOVA-based system

After finishing its initialization, the kernel starts the first boot module (after the kernel) 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 EC, and the hypervisor info page (HIP). Details about the HIP are provided in Section 6 of the NOVA specification.

**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 8.1.1 but the linker script located at *repos/base-nova/src/platform/roottask.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 8.1.2).

### 8.7.3 Log output on modern PC hardware

Because transmitting information over the 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 prior known. 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** The alleviate the need to adapt the kernel configuration to the used comport hardware, the bender chain boot loader can be used.

**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 been already 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 from 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 5.5) is used.

### 8.7.4 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 8.7. A brief glossary for the terminology used in the remainder of this section is given in table 1.

| 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 1:** Glossary of NOVA's terminology

**NOVA capabilities are not Genode capabilities** Both NOVA and Genode use the term "capability". However, the term does not has 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".

**PD service** A PD session corresponds to a NOVA PD.

**CPU service** NOVA distinguishes so-called global ECs from 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 executed not 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, NOVA's HIP is provided as a ROM module named "hypervisor_info_page".

**CAP service** A Genode capability corresponds to a NOVA portal. Each NOVA portal has a defined IP and an associated local EC (the Genode entrypoint). The invocation of a Genode capability is an IDC call to a portal. A Genode capability is delegated by passing its corresponding portal selector as IDC argument.

**IRQ service** NOVA represents each interrupt as a semaphore. Within core, there is one entrypoint per IRQ session. When `wait_for_irq` is called, the called IRQ entrypoint blocks on its corresponding IRQ semaphore. In the kernel, this semaphore-down operation 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, it wakes up the IRQ entrypoint, which replies to the `wait_for_irq` RPC call.

**RM service** The RM service is used for the page-fault handling as explained in Section 8.7.5. Each memory mapping installed in a component implicitly triggers the allocation of a node in the kernel's mapping database.

### 8.7.5 Page-fault handling on NOVA

On NOVA, each EC has a 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 dedicated pager EC within core. Hence, for each Genode thread, there exist two ECs. One in the PD where the thread executes and the pager EC in core.

**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 8.3.3. If the lookup for a dataspace within the faulter's RM session 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 RM session 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 RM fault is resolved by an RM client as described in the paragraph "Region is empty" of Section 8.3.3, 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 revoke the authority. When a mapping is revoked, the kernel implicitly revokes all transitive mappings that originated from the revoked node.

Because of this design, core needs to maintain a core-local memory mapping for each memory mapping established outside of core. This mapping is solely needed to revoke the memory mapping later on, for example, when a dataspace is detached from an RM session. The kernel's revoke operation takes the core-local address as argument and revokes all mappings originating from this mapping node.

### 8.7.6 Asynchronous notifications on NOVA

The NOVA base platform generally follows the mechanism described in Section 8.4. However, in contrast to traditional L4 kernels, NOVA's kernel interface does not support the out-of-order processing of RPC requests. On this base platform, the blocking of the signal thread at the signal source is realized by using a kernel semaphore shared by the SIGNAL session and the SIGNAL client. 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 performs a *down* operation on this semaphore to block. Core issues an *up* operation on the semaphore each time a signal of the SIGNAL session becomes pending. When unblocked from the semaphore's down operation, the client requests the signal's imprint and counter using a non-blocking *wait-for-signal* RPC call from the signal source.

### 8.7.7 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 got access to the extended PCI configuration space of the device, it is able to get the assignment in place.

To make NOVA's IOMMU support available to Genode components, the ACPI driver has the ability to hand out the extended PCI configuration space of a device, and a NOVA-specific extension (*assign_pci*) to the PD session interface can be used to associate a PCI device with a protection domain.

Even though these mechanisms combined principally 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 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 caring 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 ACPI 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 ACPI driver. The ACPI 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 handing out the dataspace capability to the client. If the driver requests the physical address of the dataspace, the returned address will be a valid virtual address in the associated device PD. From this design follows that a device driver must allocate DMA buffers at the ACPI server (specifying the PCI device the buffer is intended for) instead of using core's RAM service to allocate buffers anonymously. Note that the current implementation of the ACPI server assigns all PCI devices to only one device PD.

### 8.7.8 Genode-specific modifications of the NOVA kernel

NOVA is not fit to be used as 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 had been used for 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 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.

#### 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 targeted 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 platform, 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 the 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 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 thread-context 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 RM session 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 RM fault handler (outside of core) responds to the RM 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 thread context within core. Since core's thread-context 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 thread contexts 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. Con-

sequently, 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.

### 8.7.9 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`). The existence of a fixed pool implies that any component that is able to trigger allocations in the kernel is able to indirectly consume kernel resources. A misbehaving component in possession of its own PD capability selector may even forcefully trigger the exhaustion of the entire pool and thereby make the kernel unavailable. I.e., the kernel panics when running out of memory. The kernel provides no mechanism to mitigate such a resource-exhaustion-based denial-of-service attack.

On Genode, only core explicitly allocates kernel objects, which relieves the problem but does not solve it. In order to create a kernel object, a PD capability selector must be specified to the respective system call. Since PD capability selectors are never delegated to the outside of core, kernel objects cannot be directly created by arbitrary components. The creation of kernel objects is rather a side effect of the use of core's services. Thereby, core is principally in the position to restrict the use of kernel memory per client. However, such an accounting for kernel memory is not performed by the NOVA version of core.

In addition to the explicit creation of kernel objects, kernel memory is implicitly allocated when mapping nodes are inserted into the kernel's mapping database. Thereby, kernel memory is consumed as a side effect of IDC calls that carry map items. Since ECs of the same PD can perform IDC calls between one another, the allocation of mapping nodes can be artificially stressed by delegating a large number of mappings within the same PD via successive IDC calls.

Therefore, components are principally able to launch denial-of-service attacks on the kernel. In the event of an exhaustion of kernel memory, the kernel stops the system. Hence, even though the lack of proper management of kernel memory is an availability risk, it cannot be exploited as a covert storage channel.

**Bounded number of object capabilities within core** For each capability created via core's CAP service, core allocates the corresponding NOVA portal and maintains the portal's capability selector during the lifetime of the associated object identity. Each allocation of a capability via core's CAP service consumes one entry in core's capability space. Because the space is bounded, clients of the CAP service could misuse core's capability space as covert storage channel.

**Core must retain mappings to all memory used throughout the system** As mentioned in Section 8.7.5, core needs to own a mapping node before delegating the mapping to another PD as a response to a page fault. Otherwise, core could not revoke the mapping later on because the kernel expects core's mapping node as a proof for the authorization for the revocation of the mapping.

Consequently, even though core never touches memory handed out to other components, it needs to have memory mappings with full access rights installed within its virtual address space. Therefore, core on NOVA cannot benefit from a sparsely populated address space as described in Section 8.6.8 for base-hw.

**Non-executable (NX) bit on 32-bit machines**

NOVA supports the NX bit on 64-bit machines. In the x86 32-bit architecture, however, there exists no NX bit unless the x86 Physical Address Extension (PAE) is enabled. Because NOVA does not support PAE, the NX bit remains unsupported when using NOVA on 32-bit machines.