Microkernels, Genode and Gapfruit OS

Timo Nicolai and Alice Domage

2023

Contents

1 Foreword

This is a text version of the talk "Microkernels, Genode and Gapfruit OS" given by Timo Nicolai and Alice Domage at the Zürich C++ Meetup in September 2023. It should contain all information relayed in the presentation, including figures and citations. Since the talk has not been recorded I hope this will be of interest to anyone who could not attend in person or wants to revisit certain topics. For questions or corrections you can reach Timo at timo.nicolai94@gmail.com. Happy reading.

2 About This Presentation

Hello everyone, welcome to our presentation titled "Microkernels, Genode and Gapfruit OS" In this talk we want to give you a brief introduction to the world of microkernel operating systems. First we will briefly talk about what a microkernel is, explain what their advantages are and mention some well known microkernel systems. We will then give a historical overview of the development of microkernels, focussing on what is called the *L4* familiy and introducing the most important microkernel systems in use today. We will then present common properties of microkernel systems through the lens of the *Genode* OS framework which makes it possible to build general purpose operating systems in a more or less kernel-agnostic way. Last but not least we will give an overview of our work at Gapfruit AG where we develop the Genode-based *Gapfruit* operating system.

3 About Us

But first, we will briefly introduce ourselves.

3.1 Alice Domage

Alice is an operating systems engineer at Gapfruit. She develops and maintains device drivers and components for microkernel-based operating system. Previously, she worked at Ubisoft and Veepee, and graduated from Epitech Paris.

3.2 Timo Nicolai

My name is Timo, I am an operating systems engineer at Gapfruit. I mostly work on integrating our system with Azure IoT and on realizing our public key infrastructure. Previously I have obtained a Masters degree in computer engineering at TU Dresden where I also worked at Kernkonzept, the company that develops the Fiasco.OC microkernel and its L4Re runtime environment.

4 Microkernels

4.1 What is a Microkernel?

So, let's start by briefly explaining what a microkernel is. In effect a microkernel is a very small kernel that only implements the minimal functionality needed to interact with the hardware. Everything else, including device drivers and resource management policies is then implemented by a set of userspace components.

The key mantra of most microkernel systems is that the kernel itself should, if at all possible, only implement *mechanisms* but not *policy*. That means that heuristic decisions such as how much memory to allocate to each component should not be made by the kernel. A notable exception is scheduling, which is usually implemented in kernelspace for performance reasons.

In contrast, most commonly used kernels such as Linux implement these things in the kernel itself. In the early days of operating systems these so called *monolithic* systems worked perfectly fine because most computing systems only came with a small number of peripherals. Nowadays though, device drivers alone make up the majority of the millions of lines of code in the Linux source tree.

In light of this, the microkernel approach offers several advantages:

- **Trustworthiness**: First and foremost, the size of the code that has privileged access to the hardware is kept as small as possible. We are talking about thousands instead of millions of lines of code. This means that there are almost definitely going to be fewer possible kernel-level exploits. Furthermore, exploits in device drivers cannot directly affect the kernel.

- **Reliability**: This also prevents, for example, buggy drivers from bringing down the whole system. Misbehaving userspace components can ideally simply be restarted.

- **Resource Management**: With traditional monolithic systems, reasoning about resource consumption is very difficult. Take memory for example: The kernel will usually present a user space program with the illusion of its own address space. But now what if a program allocates so much memory that the system cannot keep up? While swap memory is a thing, even that is limited and at some point the kernel will have to kill programs that consume too much memory. That by itself may sometimes be acceptable but consider the situation in which a server program allocates memory in response to client requests. Misbehaving clients could then force the kernel to kill this server by sending too many requests. Most microkernel systems solve these issues by allocating each component a fixed slice of physical memory and letting components trade memory during inter process communication. More on that later.

4.2 Examples of Microkernel Systems

I have listed some names here that some of you will be familiar with, such as:

- **MINIX** [1]: Developed by Andy Tanenbaum who later had a famous "disagreement" with Linux over kernel design.
- **GNU Hurd** [2]: The GNU Kernel that should have been what Linux is now. I think you can actually run Arch on this.
- **Zircon/Fuchsia** [3]: An OS developed by Google that more or less follows the microkernel approach. To be honest I don't really know why they are making this.
- **Redox** [4]: For the Rust enthusiasts among us, a popular open source hobbyist microkernel OS written in Rust.
- **L4 Family**: There are many more, so for simplicity we will not discuss any of these in detail but instead focus on the L4 family of microkernels which has thus far been the most relevant both academically and commercially.

4.3 L4 Microkernel History: Predecessors

Before we can delve into L4 however we must take a look at the systems that preceded it.

- **RC 4000** [5]: One of the first ever microkernel operating systems was the *RC 4000 Multiprogramming System* created by Per Brinch Hansen in 1969. This system featured a small so-called "nucleus" and a hierarchy of programs running on top of it that implemented for example resource allocation without direct kernel involvement.
- **Mach and friends** [6]: Microkernel development continued in the following decades and several new kernels emerged, for example **Mach** developed at CMU, a UNIX compatible system which GNU Hurd is based on.

These systems worked but real world use was limited due to their poor performance. Their greatest shortcoming was IPC performance: sending a one-way message between two programs could take as long as 100 microseconds which was as ridiculously slow back then as it is now. Since microkernels rely on the cooperation of many userspace components to realize basic system functionality you can imagine that slow IPC can make such a system essentially unuseable.

At this point microkernel development hit somewhat of a roadblock because it seemed like while the underlying idea was solid, acceptable performance could not be achieved.

- **L4** [7]: But along came Jochen Liedtke who realized that poor IPC performance was simply an issue of implementation. His *L4* microkernel,

based on the earlier *L3* microkernel, was implemented completely in assembly and introduced several optimizations. For example all IPC was made synchronous to avoid double buffering when copying messages between sender and receiver. Messages where also passed mainly in registers which in combination with a context switch to the receiver could avoided copying messages at all. L4's IPC code was also much more cache friendly than Mach's. All in all, this improved IPC performance by an order of magnitude which was the breakthrough needed to bring microkernels back to relevance.

4.4 The L4 Microkernel Family: Recent Developments

Spurred by L4, development of similar systems began in primarily three locations:

- **University of Karlsruhe** where Liedtke became a professor.
- **UNSW/NICTA (now called Data61)** under Gernot Heiser, probably the most prominent microkernel researcher today.
- **TU Dresden** under the lead of Professor Hermann Härtig who is now retired but was still teaching when I did my Bachelor's there.
- At Karlsruhe we saw the development of *L4Ka::Hazelnut* [8] which was written in C++, retaining most of the performance of L4. After Liedtke died in 2001, development then continued on *L4Ka::Pistachio* [9] which was developed in a more platform-agnostic way, making it easier to port to other architectures.
- At UNSW/NICTA, one major achievement has been what is now called the *OKL4* microkernel which has shipped in bilions of Qualcomm chips and iOS coprocessors. This later lead to the development of the *OKL4 Microvisor* [10]. Behind these is now the company *Open Kernel Labs*. Independently, *seL4* [11] was developed from scratch with formal verification in mind. seL4 is not only formally verified against its specification, the whole translation process from the initial high level Haskell implementation to C to assembly has been proven to be correct. Depending on how you look at it, this makes seL4 the most secure operating system in existence. It is also very fast and suitable for hard realtime applications.
- At TU Dresden, *Fiasco* [12] emerged as the first L4-type kernel that actually saw commercial use. Like Hazelnut is was written in C++ and it was also the first kernel of its type to be fully preemptible. Fiasco is still alive today, it has been renamed to *Fiasco.OC* and is being developed by the original authors at the company *Kernkonzept* based in Dresden. One of its main claims to fame is L4Linux which lets users run Linux as userspace processes directly on top of Fiasco.OC. Another more recent project to come out of Dresden is *NOVA* [13] which focuses on virtualisation. Udo

Steinberg, the creator of NOVA has recently founded *BedRock Systems* which uses the system for secure cloud computing. We also use NOVA at Gapfruit to run our system on x86.

Figure 1: L4 Microkernel Family, based on [14]

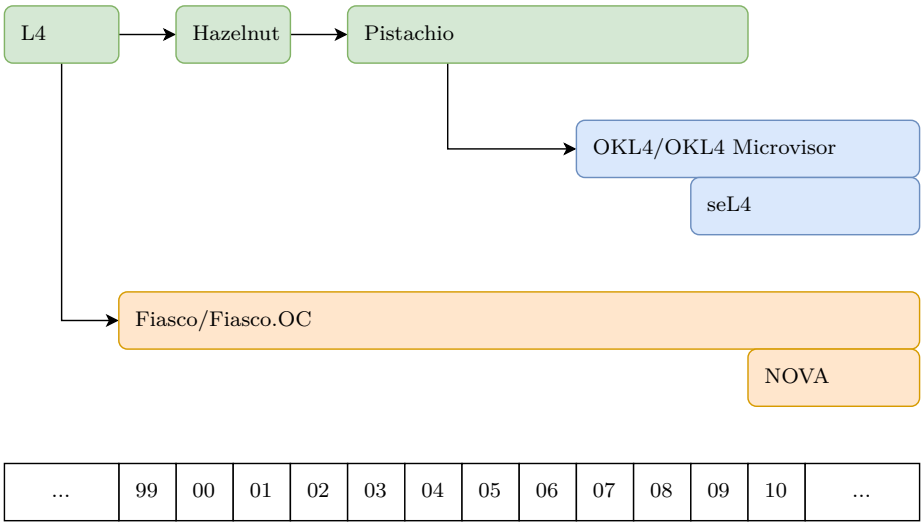


Figure 1 shows an abbreviated timeline of development of these microkernels and is adapted from a publication by Gernot Heiser. At the bottom here we have a time-axis starting in the 90s and progressing to today. The arrows indicate which kernels where directly based on the implementations of their predecessors. For example, OKL4 started as a fork of Pistachio while seL4 was developed from scratch.

4.5 The L4 Microkernel Family: The Third Generation

When talking about the history of microkernels, people tend to group them into three generations, the slow microkernels of the first generation, the L4 variants of the second generation and today's third generation microkernels which mainly focus on capability based security (more on that later), virtualisation and formal verification.

5 Genode OS Framework

5.1 What is Genode?

So, enough about how microkernels came into existence, let's take a closer look at how we can build full operating systems on top of them. As I've outlined,

microkernel operating systems are primarily made up of a number user space components. But the kernel still has to facilitate creation of these components and control how they share available hardware resources and how they communicate with each other. And since each kernel is different, the details are going to vary. So instead of focussing on any particular kernel we're going to adopt the worldview of the *Genode* [15] operating system framework.

Genode is an open source project that has been under development by the Dresden-based company *Genode Labs* for over 10 years. Genode Labs also actively develops a desktop (and now also mobile) operating system called *Sculpt* [16]. At its core, Genode is a toolbox of different components and C++ libraries that allow anyone to compose operating systems on top of a number of microkernels, including the *HW* kernel that Genode comes with and many of the L4-type kernels we have seen earlier. For development purposes it can also be run on top of Linux userspace. The Genode team tries their hardest to make this process seemlessly work on all supported kernels. Luckily the interface of most third generation microkernels is somewhat similar such that the abstractions provided by Genode are in most cases not far off from how the kernels actually operates. So by studying the architecture of Genode we can develop an understanding of microkernels function in general.

To this end we're mainly going to focus on two topics now, *capabilities* and the *component hierarchy*.

5.2 Capabilities

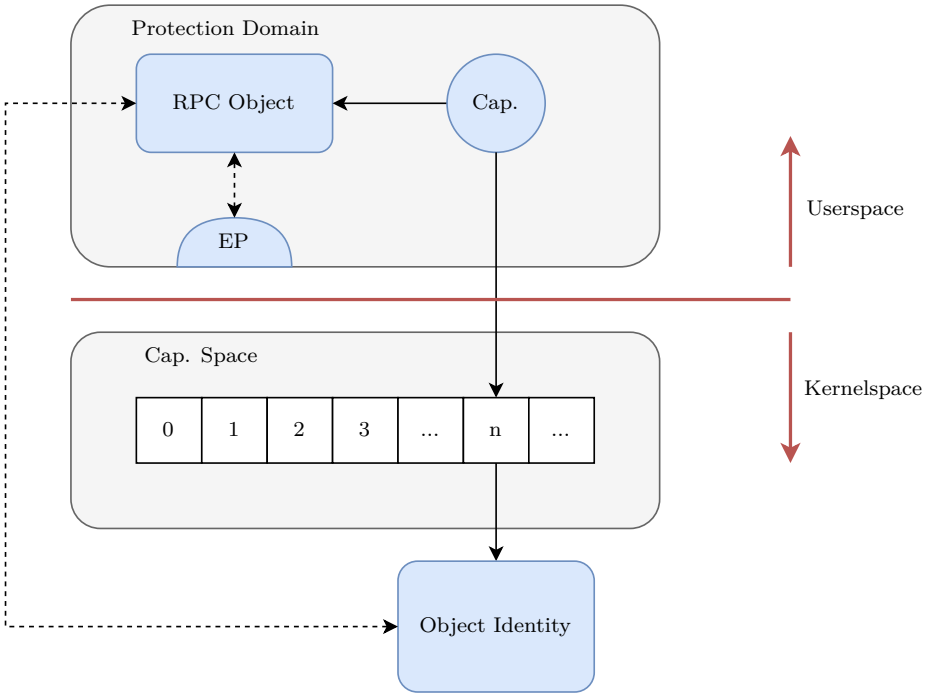
Let's start with capabilities. When you have a large number of userspace components running on a single system, the first problem you run into is of course one of communication: how do we make components aware of each and how do we let them securely communicate with each other. In other words: components need to be able to provide services to and consume services provided by other components. Third generation microkernels solve this problem using capabilities.

So far we've mainly been talking about *components*. Now we'll instead mostly talk about *protection domains*. In Genode, each component lives inside a an isolated execution environment called a protection domain that encompasses for example a number of threads and also its capabilities.

Abstractly, a capability is both a pointer and an access right to a service provided by another protection domain. Services in turn are implemented by what we call *RPC objects*. In more practical terms, a capability is a thing that a protection domain has that allows it to invoke RPCs. So to understand capabilities we have to understand how RPCs work in Genode.

5.2.1 Creation

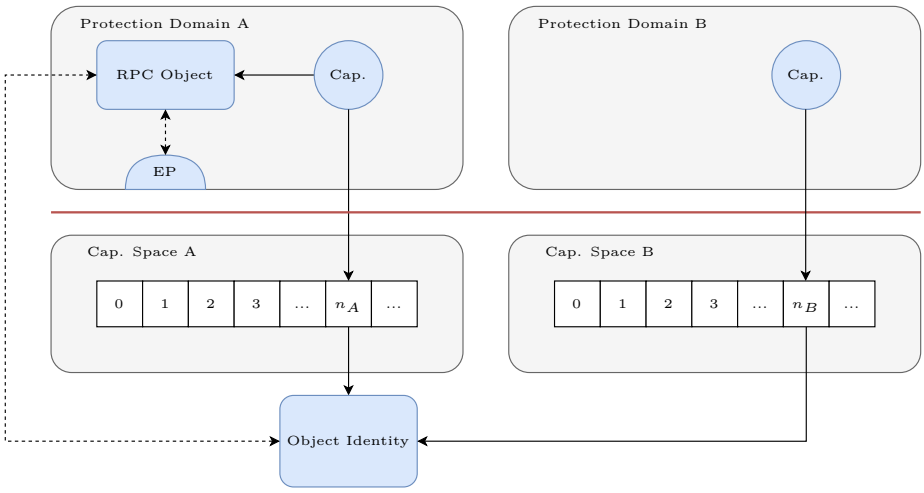
Figure 2: Capability Creation



Now, in order to make an RPC in Genode, inside some protection domain we have to create what's an aforementioned RPC object which implements a number of functions that can be called from other protection domains. In order for an RPC object to be of any use, it has to bound to a thread which we refer to as the *entrypoint* thread.

This happens in collaboration with the kernel and results in the creation of an *object identity* managed by the kernel which uniquely identifies the RPC object. Alongside the object identity, a capability is created. This capability is effectively a pointer to the object identity. Implementation-wise, a capability is most commonly simply an integer that indexes what we call the *capability space* associated with the protection domain. Each protection domain has its own capability space maintained by the kernel which serves as an indirection between the capabilities of a protection domain and the object identities controlled by the kernel. It is up to the kernel to set up and control the correct mapping between capabilities and object identities. Take a look at figure 3 for a visualization of this process.

Figure 3: Multiply Capabilities to the Same Object Identity



By itself this is not very exciting, to actually do anything useful, another process has to hold a capability that points to the same object identity such as in figure 3. Notice that the entries in the capability spaces of both protection domains can be different.

5.2.2 Invocation

Now envision a simple scenario such as in figure 3 where one protection domain **B** holds a capability to an RPC object owned by another protection domain **A**. Let's worry about how **B** has obtained this capability later.

Any thread in **B** can use the capability to invoke an RPC to the RPC object owned by **A**. In this context we usually refer to **B** as the *client* and **A** as the *server*. From the client's point of view the RPC all is synchronous and as such returns execution to the thread that made call just like a library call would. When an RPC arrives at the server, it runs in the entrypoint thread which afterwards becomes free again to handle the next incoming RPC. This way server programs always implement a form of event loop. Of course, for security reasons we can't let components community with each other willy-nilly so any RPC in the end has to be mediated by the kernel.

Here's how that works in detail:

- A thread in protection domain **B** supplies the kernel with an index into its capability space as well as opcode and arguments of the function it wants to call via the *call* operation and subsequently goes to sleep.
- The kernel checks if the corresponding entry in the capability space refers to a valid object identity.

- If so, it finds protection domain **A** which owns the corresponding RPC object and wakes up its entrypoint thread, providing the index for that same RPC object in **A**'s capability space as well as the function opcode and arguments provided by the client.
- With this info, the entrypoint thread can find and invoke the correct function in the correct RPC object.
- The return value (if on exists) of that function is then again passed to the kernel via the *reply* operation. The kernel passes passes it to the sleeping thread in protection domain **B** that is consequently woken up.

5.2.3 Delegation

So far so good, but starting from an RPC object and an associated capability in a protection domain, how do we actually create capabilities to the underlying object identity in other protection domains? This process is referred to as *delegation* and is again mediated by the kernel.

From the point of view of a protection domain, in order to send a capability to another protection domain, it is simply supplied as an argument to an RPC call. The kernel will treat such capability arguments to RPCs in a special manner: It checks if the receiving protection domain already has a capability to the underlying object identity and if not, simply create a new one by creating an appropriate entry in the receiving protection domain's capability space. That way protection domains can tell their friends: "hey, here's a capability to my RPC object, feel free to call it".

Now you might spot a chicken and egg problem here, in order to transfer a capability I need to invoke an RPC, but to do that I need a capability that someone must have probably transferred to me earlier. This cannot work if all components come into existence without holding any capabilities.

5.3 The Component Hierarchy

5.3.1 Parents and Children

The solution to this conundrum is the component hierarchy which we will talk about now. The component hierarchy is somewhat similar to the process hierarchy in UNIX-like systems in that every component except the *core* component at the root of the component hierarchy has exactly one parent component.

This parent fully creates its children by providing them a share of its own hardware resources and has complete control over its children over their whole lifetime, meaning that it can for example terminate them at will. After the creation of a child, the only capability in its protection domain is one to the parent. Due to this, children have to inherently trust their parents, but note that this is not true the other way around.

5.3.2 Services and Sessions

Now, what does this have to do with distribution of capabilities? Essentially, components can create RPC objects that implement the so called *root interface* which provides functions for setting up and tearing down what we call *sessions*. "Session" means more or less what you would expect here: Within a session, a component provides some predefined *service* via RPCs. The component can then *announce* this service to its parent, in effect saying 'hey, in case someone asks, I can do **this**".

Other children can use their parent capabilities to perform *session requests*, specifying what kind of session they want and any additional information that may be relevant. For example, a component might request a *ROM* session and provide the name of a file whose content it wants to read.

The parent can check which other child has previously announced a suitable service and initiate the transfer of a capability from the component providing the service to the one requesting it along the component hierarchy. But note that the parent is free to handle service announcements and session requests as it sees fit, it can for example just deny them, provide sessions itself, forward them to some child or its own parent and even modify the session request itself. Sticking to the previous example it could for example request a ROM session for a different file from its own parent.

Importantly, once a session it established, communication is direct and does not involve any other components along the component hierarchy such that no performance penalty is incurred.

For any such created session, there is an asymmetric trust relationship: the client needs to trust the server because the latter could in theory block on an RPC forever. On the other hand, servers do not need to (and as a rule should not) trust clients. It never needs to wait for the client anyways and should furthermore not invoke client capabilities, and retain control of any shared memory.

Another important aspect that we will not investigate in detail here due to time constraints is resource trading. To avoid the problem of client requests exhausting a server's resources, clients can donate a share of their memory to servers when initiating a session. There are several subtleties to this and its covered in detail in the Genode documentation.

Figure 4: Announcing and Requesting Sessions

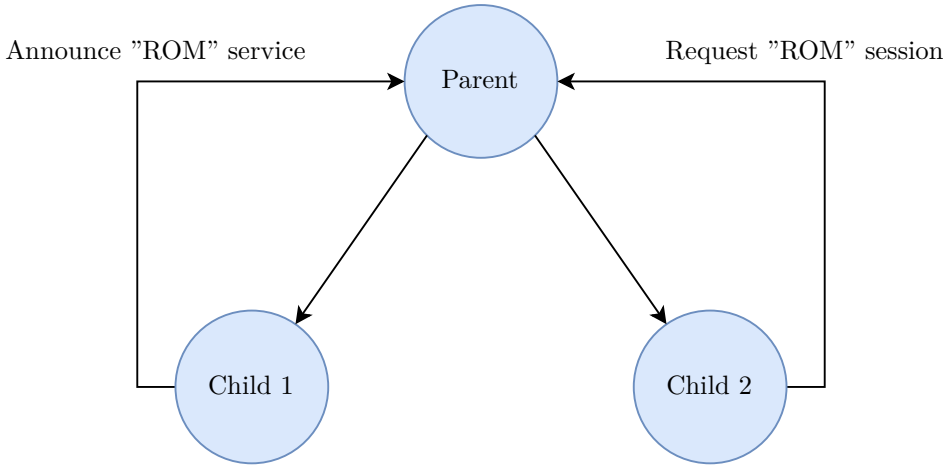
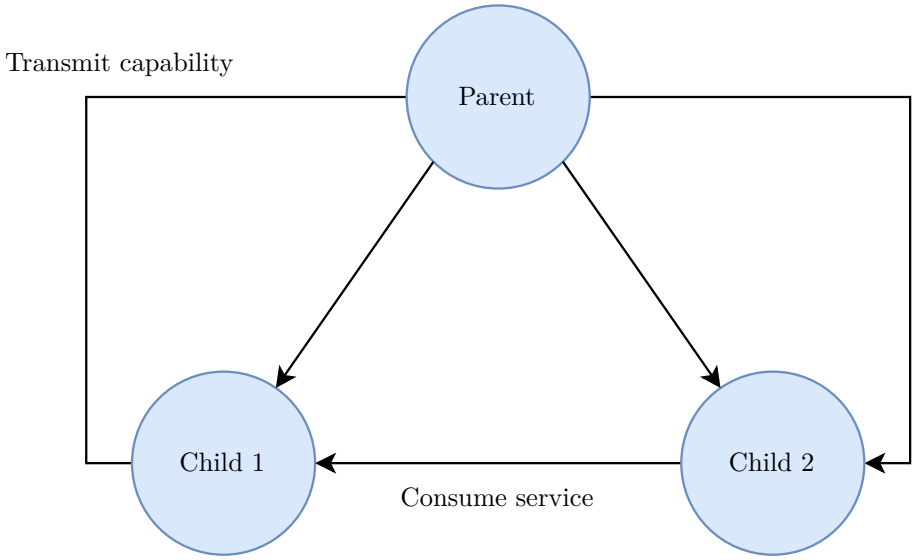


Figure 4 visualizes the process of announcing services and requesting sessions.

Figure 5: Resolving a Session Request



And figure 5 then shows one way this session request could be resolved, by passing a capability along the parent which results in a session being created

between the children.

5.4 The C++ Runtime

Okay, so much for Genode's basic architecture. Since this is the Zürich C++ meetup and not the Zürich operating system theory meetup I will also say a little bit about Genode's C++ runtime and how to write native Genode components in C++. Let's start with the runtime.

While it's generally possible, most native Genode components don't use the C++ standard library. That is mainly because it's huge and since the whole point of microkernel systems is to compose small trusted components with each other that would sort of defeat the point. What Genode does use however are exceptions and in my opinion for good reason since while you can ask your favorite C++ compiler to turn off exception support, C++ without them is basically just a more convoluted form of C. Now, this is again a bit problematic because, as you know, exception handling requires a bunch of runtime support whose exact form depends on whatever black magic your compiler generates. In any case that means that whether you use the standard library or not, some (typically also large) runtime library will be linked into your program. To get around this, Genode provides its own more minimalistic C++ runtime library.

5.5 A Simple Genode Component

Let's now take a look at the implementation of a very basic Genode component that simply outputs a message once a second five times in a row and then exits. To be exact, let's say the output of this component should look like this:

```
timer tick 1
timer tick 2
timer tick 3
timer tick 4
timer tick 5
```

Figure 6 shows the code and the first thing we notice is probably that there is no `main` function. Instead we are confronted with the `construct` function which serves as an entrypoint for the component. Arbitrary code can go in here but the most common pattern you will see in Genode programs is the one we have here: an object, here `Main`, with `static` lifetime is constructed inside 'construct' and that object then implements some sort of event loop in which it performs work and optionally handles incoming RPC requests. In contrast to a classic `main` function, when `construct` returns, the program does not terminate. Instead, the entrypoint thread we've met earlier will keep running in the background forever unless the components parent chooses to terminate it. If the lifetime of our component should be limited, we can alternatively ask the parent to free us from our misery by calling `env.parent().exit()`.

So let's dissect what is going on here:

Figure 6: A Simple Genode Component

```
/* component which outputs a message 5 times and then exits */
class Main
{
    private:

        Genode::Env &_env;

        /* timer */
        Timer::Connection _timer { _env };
        /* timer tick counter */
        Genode::size_t _timer_ticks { 0 };
        /* timer trigger callback */
        Genode::Signal_handler<Main> _timer_handler { _env.ep(), *this, &Main::_handle_timer };

        void _handle_timer()
        {
            if (++_timer_ticks > 5)
                _env.parent().exit(0);

            Genode::log("timer tick ", _timer_ticks);
        }

    public:

        explicit Main(Genode::Env &env)
        : _env { env}
        {
            /* initialize timer trigger handler */
            _timer.sigh(_timer_handler);
            /* trigger timer periodically (once every second) */
            _timer.trigger_periodic(1'000'000);
        }
};

/* component entry point */
void Component::construct(Genode::Env &env)
{
        static Main main { env };
}
```

- The `construct` receives a single reference parameter `env`. This is the initial environment of the component via which it can for example access the capability to its parent and its endpoint thread.
- We use this `env` variable to initialize an object of type 'Main' with static lifetime and then exist `construct`. From this point on the event loop takes over.
- In our `main` object, we instantiate a `_timer` object and a `_timer_handler`. The timer handler calls the function passed to it (in this case `_handle_timer`) whenever a timer event occurs. Notice that we also supply the timer handler with `_env.ep()` which is a reference to the components entrypoint thread in which `_handle_timer` will execute.
- In the constructor we initialize the timer handler and instruct the timer to trigger once every second.
- In order accurately perform an action every second, the component of course has to make use of the hardware in some way. In fact, the timer functionality is provided by some other component that implements the `Timer` session interface. By instantiating the `_timer` object, this component will implicitly send a session request to its parent who will then route it to another component, for example a timer driver, as explained earlier.
- In `_handle_timer` we simply display a message and after we have done that five times, we call `_env.parent().exit(0)`. Here we send an RPC to the parent of this component using the parent capability provided to us in the initial environment. With this RPC we ask the parent to terminate this component. The parent is of course free to ignore this request but will most likely oblige so that execution will stop at this point.

A component like this is what I would refer to as a "native" Genode component. Of course sometimes we are not only interested in writing components from scratch but want to run existing programs on Genode. To this end, Genode's build system and its 'libc' implementation provide a framework for porting existing POSIX applications to Genode with more or less minimal modifications. Additionally, Linux device drivers can be ported to Genode in a similar fashion. How this works in detail is beyond the scope of this talk and covered in the Genode documentation.

6 Gapfruit OS

6.1 Purpose

Okay, so now that we've taken a bit of a deep dive into how Genode works, let's talk a bit about a real world example operating system that is based on Genode. For this, we will take a look at what Alice and I are working on at our job.

We are developing Gapfruit OS which is a significantly more trustworthy and more reliable alternative to for example Linux-based solutions for IoT edge gateways. For those of you that don't work in the IoT space, an edge gateway is essentially some hardware device connected to the Internet that collects and processes data from a number of other IoT devices and sends that data to cloud. For example, on a factory floor you may have a few dozen to hundreds of robots that continually output data such as resource usage and sensor measurements. Instead of connecting each of these to the cloud separately, which might not even be possible because of missing computing power and a lack of security, these robots send their data to an edge gateway running Gapfruit OS which then forwards it to, for example, Microsoft Azure. This can also go the other way, for example we may want to instruct certain robots to run certain workflows from the cloud in which case these commands are routed via the edge gateway.

6.2 Challenges

Since we are thus typically controlling a physical machine that can seriously injure people or destroy expensive equipment, we have to fullfill certain requirements:

- First, with systems such as these, we typically care a lot about integrity, stability, reliability and uptime since any breaches, crashes or planned downtime may result in a very costly loss of productivity for whoever is operating the hardware connected to such an edge gateway.
- It's also important to minimize any necessary human interaction with the system since sending technicians to the factory floor to reboot devices or update software is very costly. We use the term *Zero-Touch* for this.
- Additionally, in order to enable mass deployment of edge gateways, it must be trivial to get them up and running. Once we're talking about thousands of devices in many different locations, we need to be able to just connect them to a network, boot them up and have them connect to the cloud by themselves.

6.3 Solutions

The advantages of microkernel systems that we've touched on really come into play in this scenario:

- For one, Gapfruit OS achieves very high trustworthiness because it reduces the attack surface by over 99% compared to a monolithic system.
- Thanks to Genode's component hierarchy we also have strong control over dependencies and delegation of services.
- It further increases reliability since for example device drivers can be compromised or crash without bringing down the whole system. Gapfruit OS

uses special heartbeat monitoring functionality that continually checks the liveness of components and restart them as necessary, making it possible to e.g. keep an SSH connection alive even when some part of the networking stack segfaults.

- Similarly, updating drivers becomes much simpler since due to the resilience of the system we can just kill an outdated driver at will and replace it with an updated one, without having to perform a reboot.
- With regards to mass deployment, we currently use a system that allows devices to register themselves with Azure automatically on first boot. To guarantee that only authorized devices are able to do so, we have devised a TPM-backed public key infrastructure. Explaining this in detail may be a topic for another talk. Alternatively you can tune in to our CTO Sid's presentation on Gapfruit OS at the seL4 summit next week which I will link at the end and in the text version of this talk.

Currently Gapfruit OS runs on multiple microkernels including NOVA and Genode's own microkernel with plans to support for example seL4 in the near future.

6.4 Azure Example

Figure 7: Example *Device Twin*

```
"properties": {
  "desired": {
    "dynamic": {
      "deploy": [
        {
          "start": {
            "name": "http_server",
            "pkg": "gapstage/pkg/http_server/2023-01-01"
          }
        },
        {
          "start": {
            "name": "mqtt_bridge",
            "pkg": "gapstage/pkg/mqtt_bridge/2023-02-01"
          }
        }
      ]
    }
  }
}
```

Figure 7 is a brief example of how things look like on the cloud side. This is an excerpt of what Azure calls a "device twin", a JSON document stored in the cloud via which data can be exchanged with a device running Gapfruit OS. Of particular importance is the "desired" section here. Here we declaratively specify the components that should run on the system. Whenever we update these, Gapfruit OS checks whether what is currently running on the system matches these list of components and if not, downloads any necessary dependencies from

a server, stops components that should no longer be running and starts newly added ones.

This way we can precisely control which version of which software packages runs on the device, with manual intervention or downtime. And again, this works for both user programs, drivers and everything in between.

7 References

[1] *MINIX 3*. `https://www.minix3.org/`.

[2] *GNU Hurd*. `https://www.gnu.org/software/hurd/`.

[3] *Fuchsia*. `https://fuchsia.dev/`.

[4] *Redox OS*. `https://www.redox-os.org/`.

[5] Per Brinch Hansen. *RC 4000 Software: Multiprogramming System*. Regnecentralen. 1969.

[6] Richard Rashid, Avadis Tevanian, and Michael Young. "Mach: A New Kernel Foundation for UNIX Development". In: *USENIX Summer Conference*. 1986.

[7] Jochen Liedtke. "Improving IPC by Kernel Design". In: *SIGOPS Oper. Syst. Rev.* 1993.

[8] *L4Ka::Hazelnut*. `https://www.l4ka.org/57.php`.

[9] *L4Ka::Pistachio*. `https://www.l4ka.org/65.php`.

[10] Gernot Heiser and Ben Leslie. "The OKL4 Microvisor: Convergence Point of Microkernels and Hypervisors". In: *Proceedings of the First ACM Asia-Pacific Workshop on Workshop on Systems*. 2010.

[11] Gerwin Klein et al. "SeL4: Formal Verification of an OS Kernel". In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. 2009.

[12] *The L4Re Microkernel*. `https://os.inf.tu-dresden.de/fiasco/`.

[13] *NOVA Microhypervisor*. `https://hypervisor.org/`.

[14] Gernot Heiser and Kevin Elphinstone. "L4 Microkernels: The Lessons from 20 Years of Research and Deployment". In: *ACM Trans. Comput. Syst.* (2016).

[15] *Genode Foundations*. `https://genode.org/documentation/genode-foundations/index`.

[16] *Sculpt OS*. `https://genode.org/download/sculpt`.