Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 35

Systems Group, Department of Computer Science, ETH Zurich

Unifying Synchronization and Events in a Multicore Operating System

by

Gerd Zellweger

Supervised by

Prof. Timothy Roscoe, Adrian Schüpbach

March 19, 2012

inf | Informatik
Computer Science

Abstract

Effective coordination and synchronization between processes remains a challenge that becomes even more important with the rise of multi-core hardware.
This thesis introduces Octopus, a coordination service for the Barrelfish operating system. Octopus addresses the problem of coordination between concurrent processes or activities in Barrelfish. The Octopus design is influenced by ideas from distributed computing. We show that these ideas are transferrable to operating systems. We used a declarative, logic programming engine to implement parts of the Octopus service and evaluate the benefits and drawbacks of this approach.
In a case study, we used Octopus to write a service that is responsible for device management and bootstrapping the OS as well as a replacement for the Barrelfish name server. Our experience with Octopus has shown that it simplifies programming services that require complex coordination patterns by a great deal, while at the same time offering a flexible API for a programmer to solve a wide range of coordination and synchronization problems.

Contents

Chapter 1

Introduction

1.1 Problem Statement

In 2009, *Baumann et. al.* argued that modern hardware starts to look more and more like a distributed system [4]: Machines tend to become heterogeneous as attached devices become more intelligent and programmable, leading to a number of hardware instruction sets being used within a single machine. Hot-plug support and hardware failures lead to a dynamic system where devices, bus controllers or even cores can appear and disappear at will. Non-Uniform Memory Architectures (NUMA) are already the norm today and exhibit varying latency for accessing different caches in the machine, much like network latencies in distributed systems. Traditional operating system mostly ignore this trend because they rely on cache coherence protocols and assume uniform CPU and memory architectures. These observations led to the development of the Barrelfish multikernel architecture [3]. Barrelfish embraces ideas from distributed computing and applies them inside the operating system. The result is a more distributed system architecture, aiming to be a better fit for the hardware of the future.

1.2 Context

Coordination between concurrent processes or activities has always been a problem, with the advent of more complex, diverse, and heterogeneous hardware and more fully-featured OS designs, even the efficient bootstrapping of an OS involves complex synchronization and coordination patterns between a large number of inter-dependent tasks. In order to deal with these challenges, we introduce Octopus, a new coordination service for Barrelfish. Octopus combines ideas and insights from distributed computing and is integrated into the existing structures of Barrelfish. Further, we show how we

used Octopus to write a device manager capable of reacting to changes — due to failures in hardware or hot plugging of devices of any sort — and take appropriate measures.

1.3 Overview

In the next chapter we give some background about Barrelfish, the technology we use and introduce topics referred to throughout this thesis. Chapter 3 explains the design decisions we made and the interfaces used for Octopus. Chapter 4 goes into more details about the implementation of Octopus, and we explain how we integrated the service within Barrelfish and what the advantages and drawbacks of the underlying technology are. Chapter 5 evaluates and discusses the measured performance of the service. Chapter 6 explains how we used the service in Barrelfish to simplify the boot process and device management of the OS. Chapter 7 examines various existing systems that deal with coordination, storage and events in the area of distributed systems and operating systems. Finally, in Chapter 8, we draw conclusions about our work, discuss current limitations and give ideas for future improvements.

Chapter 2

Background

2.1 Barrelfish

Barrelfish is a research operating system being developed collaboratively by researchers from ETH Zürich and Microsoft Research. The system is intended to explore ideas about the structure of operating systems for future hardware. Barrelfish is structured as a multikernel architecture [3], in anticipation that the main challenges for operating systems will be scalability as the number of cores increases and in dealing with processor and system heterogeneity. Barrelfish tries to adapt ideas from distributed systems to meet the challenges of future multi-core hardware.

2.1.1 CPU Driver

The CPU driver is a process running in privileged mode on every core. Among other responsibilities, a CPU driver handles scheduling, memory allocation tables, low-level resource allocation, protection enforcement and authorization. It is written specifically for the CPU architecture it runs on. This allows the rest of the OS to be written in a hardware-neutral way.

2.1.2 Monitor

Aside from the CPU driver, every core runs a monitor as a user-space process. The monitor acts as a coordinator between cores: It communicates with other monitors to exchange information and is also responsible for setting up inter-dispatcher communication channels. Because it is a distinct possibility that non-coherent memory will become the norm as the number of cores increases, monitors replicate state among one another using message passing as opposed to sharing state by exploiting the coherent memory of todays hardware architectures.

2.1.3 Inter Dispatcher Communication & Flounder

In Barrelfish, a dispatcher is a unit of kernel scheduling and is responsible for the management of the kernel's threads. Communication between different dispatchers (so called Inter Dispatcher Communication or IDC) is performed over bidirectional communication channels. A channel is defined by Flounder, an interface definition language. A Flounder interface is used to describe the messages exchanged over a channel. At compile time the interface is translated into C code, which allows applications to use stubs to send messages. A server can export an interface by sending a message to the monitor. The monitor then assigns a unique interface reference (iref) to the interface. Clients can use this iref to identify the service and initiate a connection. Barrelfish has a name service called Chips. Chips is used to store and look-up irefs based on a name chosen by the server. There are a number of back-ends for message passing in Barrelfish. The one chosen at runtime depends on the underlying hardware and the location of the sender and receiver. For example, on x86 hardware there are two implementations: local message passing (LMP) using hardware registers and user message passing (UMP) using shared memory. Whereas LMP is used for IDC between two dispatchers on the same core, IDC between different cores uses UMP. Which mechanism is used is determined during connection set-up.

2.1.4 Capabilities

In order to keep track of global resources, Barrelfish uses capabilities [14] to manage ownership and type of a resource. An area in Barrelfish that uses capabilities is the memory system: In case a program needs access to RAM it can request a capability for a given region of memory. The capability is then used for all calls to the kernel to invoke operations on that memory region. In contrast to access control lists (ACL) it is possible to provide fine-grained access to a specific set of resources for individual user-space programs. User-space programs do not have direct access to capabilities but instead use capability references. This allows them to pass these references around and delegate permissions to others.

2.1.5 THC

THC [11] is a set of language constructs for composable asynchronous I/O. THC is integrated in the message passing system in Barrelfish. THC's benefit is that it avoids "stack-ripped" code: Regular Flounder bindings require to specify callbacks to handle incoming messages. Plus, sending a message can fail if the communication channel is currently busy. This usually forces

a programmer to write chains of callbacks, therefore making the code hard to read and hard to maintain.

2.2 SKB

The System Knowledge Base (SKB) runs as a service in Barrelfish. The SKB is used to store and retrieve information about the underlying hardware, gathered from various locations such as ACPI tables and the PCI configuration space. In addition, a programmer can populate the SKB with static information like topology of system boards or corrections for known errors in ACPI tables. The SKB uses the ECL^{i}PS^e CLP engine to store, retrieve and process those informations. *Schüpbach et. al.* [20] used the SKB to separate hardware configuration logic from programming device registers. They show that the former is expressed easier in ECL^{i}PS^e CLP, using a high-level, declarative language, while they still use C for the low-level interaction with device registers.

2.2.1 Constraint Logic Programming

ECL^{i}PS^e CLP [1] is a constraint logic programming software system. It is intended for general programming tasks, especially rapid prototyping. The ECL^{i}PS^e language is largely backward-compatible with Prolog. However, a number of extensions have been made, including a string data type, arithmetics with unlimited precision, double precision floating points and logical iteration. ECL^{i}PS^e comes with a rich set of libraries such as constraint solvers for problem solving and non-logical data stores. The system has a tight interface to C: A programmer can call C code from within ECL^{i}PS^e code by using external C predicates. The engine provides APIs to convert data from C to ECL^{i}PS^e types and vice-versa.

Memory Management & Garbage Collector

During the execution of a query, Prolog structures, such as numbers, strings and lists are stored on the global stack. The global stack grows as the program runs and is popped only on failure of a query. ECL^{i}PS^e provides a garbage collector to clean-up the stack during long running queries. The GC is invoked based on a watermark: Once a certain amount of new space is consumed, the garbage collector will run and remove no longer needed items from the stack. A trail stack stores information needed for backtracking. In case a rule fails, the engine needs to know where to continue with the execution. This stack is garbage collected together with the global stack.

An atom is a data type in ECL^{i}PS^e. Atoms are used to represent the names of functors. They are stored in a dictionary (similar to a hierarchical page table). New atoms are added to the directory whenever the system encounters an atom that has not been mentioned so far. The dictionary is garbage collected after a certain number of new entries have been added to get rid of atoms no longer used.

The heap is where ECL^{i}PS^e keeps all the compiled Prolog code as well as static and dynamic predicates. Heap space is also used by non-logical storage facilities — that store information across backtracking — and external C predicates. A programmer writing an external predicate can access the heap using APIs provided by the operating system and `libc` (`malloc` or the like).

Prolog

Prolog [17] is a logic programming language based on first-order logic. It has only one data type called term. Terms are any one of atoms, numbers, variables or compound terms:

Atom A name. Must start with a lowercase letter.

Number Floating point or Integer value.

Variable Placeholder for an arbitrary term, always starts with an upper case letter.

Compound term Composition of an atom with a number of arguments that are terms as well.

Prolog is a declarative language. Program logic is expressed in terms of relations. There are only three basic constructs called facts, rules and queries.

Facts express something that is always true. As an example, consider these facts expressed in natural language: "Alfred lives in New York." or "New York is a state of the US.". In Prolog we can express facts as terms:

```
% Facts written as compound terms with 2 arguments:
lives(alfred, new_york).
state(new_york, usa).
```

Rules are expressed in Horn clauses. They are used to describe relations between terms. As an example, consider the following sentence in natural language: "If X lives in Y and Y is a state of the US, X is a US resident.". Translated in Prolog, the sentence takes the following form:

```
is_us_resident(X) :- lives(X, Y), state(Y, usa).
```

The left side (`is_us_resident(X)`) is `true` in case the right side evaluates to `true`. `X` and `Y` are variables Prolog can unify to any term at runtime. The comma serves as a conjunction operator for terms.

Queries are used to initiate computations in Prolog. Computations make use of the relations we have defined beforehand:

```
?- is_us_resident(alfred).
Yes.

?- is_us_resident(mary).
No.

?- is_us_resident(R).
R = alfred.
```

Prolog uses backtracking and logical unification to find a solution for a query: In our first computation, the system reasons that Alfred lives in New York (fact) and New York is a state of the US (fact), therefore Alfred must be a US resident. For the second query Prolog will find no information about the living situation of Mary, therefore the query will fail. A variable is used in case we are interested in a solution, but do not know the names of possible residents beforehand.

2.3 Advanced Configuration and Power Interface

The Advanced Configuration and Power Interface (ACPI) specification [12] is an open standard for device configuration and power management in operating systems. It builds on previous power management BIOS code such as Advanced Power Management (APM), PNPBIOS APIs and Multiprocessor Specification (MPS) tables. ACPI aims at providing an API to ensure that power management code in operating systems can evolve independently from the underlying hardware. It is also used to find out more about the installed hardware of a system as the ACPI tables contain information about APICs and installed PCI root bridges.

2.4 Advanced Programmable Interrupt Controller

Advanced Programmable Interrupt Controller (ACPI) [6] is a device used to set up, enable, disable and route interrupts in a machine. It is designed to handle a large number of hardware interrupts in a scalable way: Each CPU has an APIC built in, referred to as local APIC. I/O APICs are used to detect interrupts of hardware devices. The operating system needs to configure the I/O APICs and tell them to which local APIC a device interrupt should be forwarded. The operating system can use local APICs to generate inter-processor interrupts (IPI) to implement efficient message passing between cores.

2.5 Peripheral Component Interconnect

Peripheral Component Interconnect (PCI) [20] commonly refers to a computer bus for attaching devices to a computer. Intel published the original PCI standard known as conventional PCI in 1993. Initially, PCI had 32-bit bus operating at 33 MHz resulting in a total bandwidth of 133 MB/s for data transfer. Increasing requirements for bandwidth, due to faster peripherals, led to a new standard created by HP, Compaq and IBM known as PCI-X. Using a 64-bit wide bus running at 133 MHz, PCI-X had a peak performance of 1 064 MB/s. PCI Express or PCIe is the latest development in PCI standards and supersedes PCI and PCI-X. It has a number of improvements including higher bandwidth (up to 16 GB/s), better error detection, and native hot-plug functionality. The operating system can find out about installed cards and their function by reading out the PCI configuration space. The OS is also responsible for programming the PCI Base Address Registers (BAR) in the configuration space in order to set up correct memory mappings for device drivers.

Chapter 3

Approach

This chapter explains the general architecture and design decisions that led to the implementation of Octopus, the Barrelfish coordination service. The service aims at providing a set of APIs, which have been around for a long time in distributed systems: A searchable key–value store for structured data that allows to build synchronization primitives and a publish–subscribe mechanism for decoupled messaging.

3.1 General Service Architecture

The Octopus coordination service is built as a client–server architecture. In Chapter 7, we give an overview of similar services in distributed computing, structured in a peer-to-peer like fashion. At the current scale Barrelfish is operating, this architecture did not seem appropriate considering the effort, but might be worth to explore at a later point in time. We used the SKB and its constraint logic programming engine ECL^{i}PS^e as a server to be able to exploit the logical unification capabilities and expressiveness of the Prolog language for our implementation.

In order to keep the dependencies with the SKB minimal, we put most of the code in libraries. Only the code that directly interfaces with the ECL^{i}PS^e CLP engine had to be added to the SKB. This should make it easier to use another engine or database in the future.

`liboctopus` Contains the APIs used by clients to interface with the Octopus service.

`liboctopus_server` Server-side message handlers & connection set-up.

`liboctopus_parser` Parser for the query language (see Section 3.2).

Please note that this section is necessarily implementation-oriented in order to provide a basis for describing the interface. For a more detailed discussion of the architecture, please refer to Section 4.1 in the next chapter.

3.2 Query Language

Currently, Barrelfish uses a straight-forward approach to interface with the SKB: Strings, written in Prolog syntax, are sent directly to the SKB, which forwards it to ECL^{i}PS^e. ECL^{i}PS^e then parses the string and executes it as a query. This approach provides a lot of flexibility for the clients but has some disadvantages:

- A client needs knowledge about the underlying Prolog code in order to write correct queries.
- Executing queries unchecked on the ECL^{i}PS^e CLP engine makes it hard to have a reliable service as it is easy to write queries that affect or corrupt the state of the stored data or the engine itself.
- Queries for the SKB have to follow the rules dictated by the ECL^{i}PS^e Prolog syntax (uppercase variable names, strings in double quotes, atoms must start with a lowercase letter etc.).

As it is desirable to have a service that is not tied too strong with ECL^{i}PS^e, we decided to come up with a designated syntax that could be used throughout all the different APIs our service should provide.

3.2.1 Record Notation

In this section we introduce the notion of a Octopus record: A set of attribute–value pairs identified by a record name. As an introduction, we give some examples of records stored in Barrelfish:

```
spawnd.1 { iref: 12 }
hw.pci.device.1 { bus: 0, device: 1, function: 0,
                  vendor: 0x8086, device_id: 0x107d,
                  class: 'C' }
```

The first line is a record created by the name service client API. It contains the interface reference for the exported service named `spawnd.1`. The second line is a record for a e1000 network card found during PCI bus enumeration. The syntax is mostly inspired by JSON (JavaScript Object Notation) [27] — a data-interchange format that is easy to read and write for humans and easy to parse and generate for machines. Table 3.1 gives a description of a record

in EBNF. We allow records to contain basic data types: strings, integers and floating points. In comparison with JSON we do not support nested records or lists. These extensions could be added in the future, if the need arises.

The current design is based on some observations we made for Barrelfish:

- Barrelfish is mostly written in C. Support for C value types and structs is needed for interoperability with Barrelfish.
- The SKB contains a lot of information read out from PCI BARs and ACPI tables. Currently the SKB stores this information by combining related register or table values using dynamic predicates. It makes sense to store some of this information as records, so clients can react to changes concerning this information (see also Section 3.3.1).
- We try to exploit the advantages of Prolog as a dynamic language by parsing the records (records are represented as strings in C) at runtime and translating them into Prolog code.

A Octopus record is the unit that gets stored in the key–value storage and the message exchanged in the publish–subscribe system.

record	::=	name ["{" {attributes} "}"]
name	::=	ident
attributes	::=	attribute | attribute "," attributes
attribute	::=	ident ":" value
value	::=	ident | string | number | float
ident	::=	lower-case-alphabetic-character {alphabetic-character | digit | "_" | "."}
string	::=	"'" {string-character} "'"
string-character	::=	any-character-except-quote | "\'"
number	::=	[−] digit {digit}
float	::=	[−] {digit} "." digit {digit}

Table 3.1: Extended Backus–Naur Form for a Octopus record.

3.2.2 Record Queries

For queries sent to the server, to store, retrieve and delete records or to store subscriptions, we extended the current record notation to support constraints and variables (see Table 3.2). Constraints and variables allow us to give a more general description of a set of records we are interested in. In addition, they are expressed easily in Prolog code with the logical unification and backtracking features of the language. As an example, consider the following record queries:

```
r'spawnd\.[0-9]+' { iref: _ }
_ { bus: 0, bus == 0, device <= 1, vendor > 100,
    class: r'C|X|T' }
```

The first query looks for a record whose name starts with "spawnd." and ends with a number. In order to do effective matchings for strings in attributes and also record names, we allow constraints to be formulated as regular expressions [29]. The underscore after iref means that our record should contain this particular attribute, but we do not care about its value. The second query corresponds to any record — the underscore keeps its meaning here — matching the specified constraints. You may have noted that the combination of `bus: 0` and `bus == 0` seems redundant when used in the same query. As we explain later when we look at the storage API in more detail, for `get` and `del` queries there is no difference between them. However, for `set` queries, the former acts as an update instruction, whereas the latter forms a constraint that the record we are updating has to satisfy for a successful update. This example also shows that it is possible to have multiple constraints with the same attribute name in a single query. In this case, we combine the constraints using a logical `AND` operator. The idea is to give the clients the ability to formulate their interest in specific records as a template. In case the need arises in the future, it is possible to extend the notation to express more complex relations (for example by using a `OR` operator).

record	::=	name ["{" {attributes} "}"]
name	::=	ident | variable | regex
attributes	::=	attribute | attribute "," attributes
attribute	::=	ident ":" value | ident constraint
value	::=	ident | string | number | float
constraint	::=	operator value | ":" regex | ":" variable
operator	::=	"<" | "<=" | ">" | ">=" | "==" | "!="
variable	::=	"_"
regex	::=	"r'" {string-character} "'"
ident	::=	lower-case-alphabetic-character {alphabetic-character | digit | "_" | "."}
string	::=	"'" {string-character} "'"
string-character	::=	any-character-except-quote | "\'"
number	::=	[−] digit {digit}
float	::=	[−] {digit} "." digit {digit}

Table 3.2: Extended Backus–Naur Form for a Octopus query.

3.3 Key–Value Storage

The idea of the key–value store is to be used for applications to share small amounts of configuration or meta-data. In addition it should be flexible enough to allow clients to build synchronization primitives such as barriers and locks on top of the provided API. These primitives can then be used to solve a wide range of problems that appear when dealing with distributed processes, including but not limited to leader election and group management, rendezvous points, configuration management and shared data storage.

3.3.1 Trigger

A trigger is an optional argument passed to the server while using the storage API (see Section 3.3.2 for the API description). A trigger will register itself in the system and notify the client asynchronously or synchronously in the future, in case a record that matches the query has been modified or added using `set`, or deleted with a `del` call. A client can install a trigger to react to changes caused by others. Trigger form the basis for writing efficient synchronization primitives (see Chapter 4.6). They work similar to watches in Zookeeper [13] but have some differences that make them more flexible:

- Triggers can be made persistent: In that case the server leaves them intact after they have fired once.
- Clients can specify exactly if a trigger should be installed in the system based on the returned error code of the query invocation.

We found these features useful as they make the life for the programmer easier and require less code: In case triggers are only one-time events, clients have to re-register a trigger every time it has fired if they are still interested in forthcoming changes. This situation is bad as resetting the trigger involves an additional call to the server. Furthermore, the result of the call must be considered as well because we might have missed a change during the time between the arrival of our initial trigger event and setting-up the next trigger. Setting the trigger based on the result of the API call also reduces the code on the client side. Lets assume a very common use case where a client wants to wait until a record exists in the system. He does an `exist` call on the record and specifies to set the trigger only if the record does not exist at the time the query is evaluated on the server. In case the `exist` call finds the record and returns successful, the client does not have to worry about incoming events for this particular trigger because the trigger was never installed.

`liboctopus` allows to create and define the behavior of a trigger by invoking a library function:

```
/**
 * \param in_case A trigger is only installed in case the
 *        resulting error value of the query invocation
 *        matches this argument
 * \param send_to Specify over which binding to
 *        send back a trigger event
 * \param mode A bitmask to set various options for a trigger
 * \param fn The handler function liboctopus will call in case
 *        it receives a trigger message.
 * \param state Optional state argument, passed along
 *        to the handler function.
 */
struct trigger oct_mktrigger(errval_t in_case,
        enum oct_binding_type send_to, oct_mode_t mode,
        trigger_handler_fn fn, void* state)
```

A client can program the trigger so that the server sends trigger messages back to the client over the event binding or the THC binding (see Section 4.1). The THC binding allows to wait for a trigger messages synchronously. This makes it easier to build blocking synchronization algorithms, as we do not need to coordinate with the event binding in that case (see Section 4.6). The `mode` argument is a bitmask that allows to specify if the trigger is checked on `set` or `del` calls or both. Another bit in `mode` is used in case we want to ignore the `in_case` argument and install the trigger in any case. Further, `mode` also encodes if the trigger is persistent. Note that a persistent trigger should always be programmed to use the event binding. This is merely to avoid running into complex situations where the THC binding receives a trigger message while waiting for a RPC response.

3.3.2 Storage API

In this section, we give an overview of the implemented interface for the storage system and explain the semantics of each function. The API we present here are message primitives from the Flounder interface, the lowest level of abstraction for clients. The `liboctopus` API builds on top of this Flounder interface and avoids a lot of the hassle when dealing with strings in C.

```
/**
 * \param query Octopus query.
 * \param t Trigger for the provided query (optional).
 * \param names Names of all records that match the query
```

```
 *        (comma separated).
 * \param tid ID of the installed trigger.
 * \param error_code Returned error of the query invocation.
 */
rpc get_names(in string query, in trigger t, out string names,
              out trigger_id tid, out errval error_code);
```

A client issues a `get_names` call in case he wants to find all the records that match the provided query. The resulting names are returned in a comma-separated string. This call is a compromise that had to be made because of the limitations with the ECL^{i}PS^e engine: We experimented with saving Prolog choice points and build something similar to iterators on top of that. However, we observed that ECL^{i}PS^e starts to fail in case a lot of queries leave choice points open to continue the execution at a later point in time. For this call, attribute–value pairs are translated into constraints using the == operator.

```
/**
 * \param query Octopus query.
 * \param t Trigger for the provided query (optional).
 * \param record First record found matching the query.
 * \param tid ID of the installed trigger.
 * \param error_code Returned error of the query invocation.
 */
rpc get(in string query, in trigger t, out string record,
        out trigger_id tid, out errval error_code);
```

The `get` call returns the first record encountered in the database that matches the provided query or returns an appropriate error code in case the system did not find a matching record. For this call, attribute–value pairs are translated into constraints using the == operator.

```
/**
 * \param query Octopus query.
 * \param mode A bitmask used to define how a record is set.
 * \param t Trigger for the provided query (optional).
 * \param get Specify if we want to receive the record we have set.
 * \param record In case get is true, contains
 *        the record that was stored in the system.
 * \param tid ID of the installed trigger.
 * \param error_code Returned error of the query invocation.
 */
rpc set(in string query, in mode m, in trigger t, in bool get,
        out string record, out trigger_id tid,
        out errval error_code);
```

A `set` call is used to store a record in the system or update an existing record. The `mode` argument is currently used to tell the server to add a sequentially increasing number at the end of the record name. This lets us create unique names for records. In `set` queries, constraints are used to select the record we want to update, whereas attribute–value pairs specify which attributes we want to add or update and to what values. The benefit is that clients can avoid the lost update problem: If multiple clients update a single record simultaneously, they can add constraints based on their current local version of the record. As an example consider the following query:

```
dataRecord { data: 12, version: 22, version == 21 }
```

In case the constraint will not match (`version` is not 21) the update for `dataRecord` (set `data` to 12 and `version` to 22) fails. In addition, we do not allow the record name to be formulated as a constraint (a regular expression or a variable): Right now, we do not have any access control or protection mechanisms for records. This means that in case a client issues a `set` call with a unspecified record name, he can never be sure how many and which records are affected by the query (even if he tries to verify it by doing a `get_names` call beforehand, it could still happen that records are added or updated in between).

```
/**
 * \param query Octopus query.
 * \param t Trigger for the provided query (optional).
 * \param tid ID of the installed trigger.
 * \param error_code Returned error of the query invocation.
 */
rpc del(in string query, in trigger t, out trigger_id tid,
        out errval error_code);
```

Deletes a record matching the query from the system. As with `set` calls, we do not allow the record name to be a constraint for the same reason. For this call, attribute–value pairs are translated into constraints using the $==$ operator.

```
/**
 * \param query Octopus query.
 * \param t Trigger for the provided query (optional).
 * \param tid ID of the installed trigger.
 * \param error_code Returned error of the query invocation.
 */
rpc exists(in string query, in trigger t, out trigger_id tid,
           out errval error_code);
```

This call is similar to the `get` call, but we only return the resulting error code and not the record itself.

```
/**
 * \param tid Trigger ID to remove
 * \param error_code Returned error of RPC invocation.
 */
rpc remove_trigger(in trigger_id tid, out errval error_code);
```

A programmer can use `remove_trigger` to remove a previously installed trigger. For non-persistent trigger it is not necessary to call this function as they are removed automatically after they have fired for the first time. Also note that the server only removes a trigger in case the same binding that initially installed the trigger requests it.

```
/**
 * \param id ID of the trigger this event corresponds to.
 * \param trigger_fn Function pointer to a handler function
 *        (supplied by the client).
 * \param mode m Bitmask informing the client of the event
 *        type.
 * \param record The record that matched with the trigger.
 * \param state State argument (supplied by the client).
 */
message trigger(trigger_id tid, handler_fn trigger, mode m,
                string record, uint64 state);
```

This is a asynchronous message, sent to a client, for a trigger event. The client receives this in case one of the triggers he added previously matched with a record during `set` or `del` calls. `liboctopus` handles these messages and uses the `trigger_fn` argument to call a handler function (supplied by the client on trigger creation). The `mode` argument informs about the event that caused the trigger to fire (i.e., was it during a `set` or `del` call?). In addition, in case a trigger has been removed, a bit in the `mode` argument is set that informs the client about it. The programmer can use this to decide if it is safe to free any state associated with the trigger. Note that for persistent triggers, an extra trigger message, with only the removed bit set in `mode`, is sent, in case we issue a `remove_trigger` request. This is necessary because triggers are usually sent over the event binding (see Section 4.1): If we remove a persistent trigger, the client might still get messages concerning the removed trigger afterwards. This happens in case the `remove_trigger` request is handled on the server before all messages for the event binding have been processed on the client.

3.4 Publish–Subscribe

The publish–subscribe API is used to decouple messaging among clients. Records are published and subscribers use record queries to subscribe to a particular set of records they are interested in. Our publish–subscribe mechanism is best characterized as a topic and content based hybrid: Regular expression constraints on record names allow us to select specific message names and constraints on values allow us to filter for messages based on the content. The API is similar to existing publish–subscribe systems featuring the three functions `publish`, `subscribe` and `unsubscribe`:

```
/**
 * \param record Message to publish.
 * \param error_code Returned error of RPC invocation.
 */
rpc publish(in string record, out errval error_code);
```

Publishes an Octopus record to subscribers.

```
/**
 * \param query Octopus query.
 * \param handler_fn Address of client handler function.
 * \param state Additional state argument.
 * \param id Subscription ID.
 * \param error_code Returned error of RPC invocation.
 */
rpc subscribe(in string query, in handler_fn handler,
              in uint64 state, out subscription_id id,
              out errval error_code);
```

Adds a subscription described through a Octopus record query. The client can pass along a handler function and state argument. They are sent back to the client with a matching, published record. For subscriptions, attribute–value pairs are translated into constraints using the == operator.

```
/**
 * \param id ID of subscription that is to be removed.
 * \param error_code Returned error of RPC invocation.
 */
rpc unsubscribe(in subscription_id id, out errval error_code);
```

Removes a subscription on the server, identified by the `id` parameter. The server only removes the subscription in case the same binding that initially installed the subscription requests it.

```
/**
 * \param id ID of the subscription this event corresponds to.
```

```
 * \param trigger_fn Function pointer to a handler function
 *        (supplied by the client on subscribe).
 * \param mode m Bitmask informing the client of the event
 *        type.
 * \param record The record that matched with the subscription.
 * \param state State argument (supplied by the client on
 *        subscribe).
 */
message subscription(subscription_id id, handler_fn handler, mode m,
                     string record, uint64 state);
```

Published messages are delivered asynchronously to clients with a matching subscription by sending this message. `liboctopus` handles incoming messages and calls the handler function with the appropriate arguments. Note that this message format is identical to the trigger message. `liboctopus` also uses the same handler function type for both. It allows a programmer to use the same event handler for publish and trigger events. A bit in the mode argument indicates that this event results from a published message. As with triggers, a unsubscribe call results in an additional subscription message with a bit set in mode that indicates the removal of the subscription.

You should note the analogy between subscribe and triggers. In terms of implementation they are indeed identical (see Section 4.5), the only difference is that triggers will be matched against records, stored using the storage API, whereas subscriptions will be matched against published records.

Chapter 4

Implementation

This chapter gives an overview of how we implemented the Octopus service in Barrelfish. We also show how we used the Octopus API, described in Chapter 3, to build distributed locks, barriers and semaphores.

4.1 Octopus Architecture

Figure 4.1 gives an overview of our architecture. As we already mentioned briefly in Section 3.1, our server is built on top of the SKB. To translate records and queries into Prolog from the language we introduced in Section 3.2, we needed a lexer and parser. We built the lexer and parser using Flex [25] and Bison [23]. The parser is used by `liboctopus_server` to generate an abstract syntax tree for record queries sent to the server, but also by the clients to read data from records received from the server. Using a designated parser also leads to a clean interface with the SKB: In case we send a request to the server, `liboctopus_server` is responsible for parsing the incoming request. The resulting abstract syntax tree (AST) is then passed on to the SKB, which translates the AST into a Prolog query and executes the query, using our storage implementation written in Prolog.

4.2 Client–Server Connection

In Figure 4.1, we see that a client has two connections to the server: a regular Flounder binding (referred to as the event binding) to receive asynchronous events (namely trigger events and published records) and a THC binding used for all communication from client to the server. A client talks to the server by using remote procedure calls (RPC). The THC and the event binding use the same Flounder interface. This design has the advantage that a

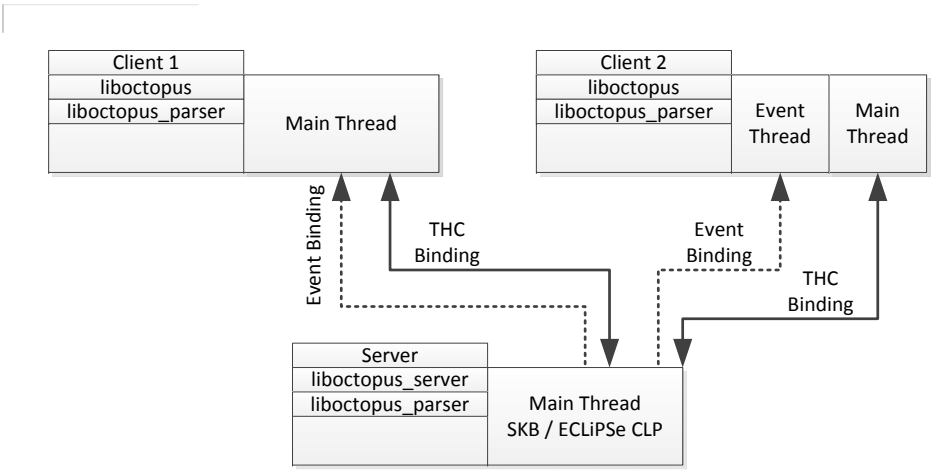

Figure 4.1: Architectural overview of the Octopus coordination service.

server can send trigger or subscription messages back over the event binding or the THC binding. A client can decide to handle the event binding either in the main thread (Client 1) or in a separate event thread (Client 2). The architecture a client should choose depends on the application.

4.3 Implementation in Prolog

One of the main challenges for this project was to come up with a database implementation for records as well as record queries (for subscriptions and triggers). We wanted to use the Barrelfish SKB for this: The SKB already serves as a centralized storage pool for information needed by Barrelfish, therefore it seemed appropriate as the storage area for records. In addition, with an implementation in Prolog it is possible for the existing Prolog code in Barrelfish to read and store records, and even generate trigger and subscription events as well. Thus, we had to integrate our service with the ECL^{i}PS^e CLP Prolog engine. Typical searchable key–value storage systems usually rely on B-trees or similar data structures for indexes that store pointer to in-memory records. This architecture allows to do fast lookups in a space efficient fashion. Prolog as a language is quite different from regular imperative languages such as C. First of all, there is no concept like references or pointers. In case we want to modify any Prolog data structure, the engine creates a copy and modifies the copy. To make things worse, the now outdated data structure must be garbage collected by ECL^{i}PS^e. The collection and copying affect the performance in a major way. For a detailed discussion of the problem see Section 5.2.2. ECL^{i}PS^e provides non-logical

data storage implementations, written as external C predicates, to avoid this limitation. In case we wanted a very efficient system, we could have just implemented everything as external C predicates. However, then there would have been no point in using Prolog in the first place. On the other hand, because Prolog is a declarative language, it allows to express the matching of constraints against records in a nice way. In addition, the built-in backtracking mechanism of the language allows us to easily find several matches of records or abort and continue with the next record in case a constraint can not be satisfied. For these reasons, our implementation is a trade-off between expressiveness and efficiency and is quite different from traditional key–value stores: We store records and record queries as Prolog terms, using the non-logical ECL^{i}PS^e storage APIs. Matching constraints in records, backtracking over stored records or finding subscriptions is implemented in Prolog. Although this approach works on its own, we added some optimizations to reduce the amount of records or subscriptions we have to consider when a query is executed. We explain the storage implementation for records, subscriptions and triggers in more details in the following sections.

4.4 Record Storage and Retrieval

Our implementation uses a non-logical hash table, provided by ECL^{i}PS^e, to store records. When the SKB receives the parsed AST for a record query, it will walk through the tree and generate Prolog terms for the record name as well as any encountered attributes and constraints. For `set` calls, attributes are transformed into an ordered set in Prolog (i.e., a sorted list). Using the record name as the key, we store the attribute set into the hash table. In case we want to retrieve a record, the corresponding query is parsed and translated into Prolog terms. Prolog then matches the resulting constraint terms against the stored records.

4.4.1 Attribute Index

Using a hash table allows queries that specify a record name to complete in constant time, independent from the amount of records we store. However, our queries also allow to search for arbitrary records, based on the values of attributes. In the beginning, the execution of these queries was very inefficient because we could not rely on hashing and had to iterate over all records until we found a match. For this reason, we decided to build an attribute index for queries. Our index for record retrieval remembers for every attribute the names of records containing the attribute. This approach can be compared to the workings of a simple boolean search engine. As an example, consider the following record from Barrelfish:

```
hw.pci.rootbridge.1 { acpi_node: 1, bus: 0, device: 0,
                      function: 0, maxbus: 255 }
```

If this record is inserted in our storage system, we have to insert the name `hw.pci.rootbridge.1` into the index for `acpi_node`, `bus`, `device`, `function` and `maxbus`. The system finds an individual attribute index by a hash table lookup. The index itself is implemented as an ordered set. We used a skip list [19] data structure to represent the set. Skip lists behave similar to binary trees, which means we can insert an entry into the index in $O(log(n))$ time, given that we have the right amount of skip pointers and they are distributed in an optimal way among the entries of the list. For retrieval, we are now able to do an index intersection based on the attributes we specify in the query. Consider this query, used in Barrelfish to find all PCI root bridges:

```
r'hw\.pci\.rootbridge\.[0-9]+' { acpi_node: 1, bus: _, device: _,
                                 function: _, maxbus: _ }
```

It expresses that we are interested only in records containing all five attributes and the regular expression in the beginning matches the record name. In order to avoid matching a lot of records with different attributes, we find the index for all five attribute names and compute an intersection of the sets. The intersection algorithm sorts the skip lists by increasing length and then continues to find the elements of the first list in all the others. Skip pointers allow us to skip many entries that are not relevant for our intersection, therefore making the intersection more efficient. The index and the intersection code is written in C. This implementation minimizes the garbage produced in Prolog as we can traverse the skip lists in-place during the intersection: Our Prolog code uses a non-deterministic external C predicate to do the intersection. The C predicate starts the intersection and returns the first record name that contains all attributes. Our Prolog implementation will then fetch the record, match the individual attributes and the record name (in our example it will verify that the record name matches the regular expression and `acpi_node` is indeed one). In case this matching fails, Prolog backtracks to the intersection predicate and our intersection algorithm will produce the next result, until no more names are found.

You may notice that there is still an extreme case where we cannot make use of our attribute index. Consider a query in the following form:

```
r'name[0-9]' {}
```

Here, we have to iterate over all currently stored records to find one whose name matches the given regular expression. It is especially bad as we have

to invoke the regex engine every time until we find a match. It is good practice to always include the attributes of interest in a query. Not only does it reduce the computing overhead on the server, it also ensures that a record contains the relevant information once it is read on the client side.
Right now, ECL^{i}PS^{e} uses the PCRE regex library [29] to match regular expressions. It might be worth it to restrict the complexity of expressions in the future and use a trie data structure to do the matching in linear time.

4.5 Triggers and Subscriptions

Our service also needed a way to store triggers or subscriptions. Triggers and subscriptions are different from records as they are expressed as a record query. Instead of just attribute–value pairs we also need to store constraints. The trigger and subscription storage works similar to the one for records: We used the non-logical hash table, provided by ECL^{i}PS^{e}, to store the queries as an ordered set of constraints. This time, the ID of the trigger or subscription serves as the key for the hash table. The problem we face here is inverse to the one of record retrieval: How do we find relevant triggers or subscriptions given a record? As an example consider the following record and two record queries:

```
hw.apic.1 { cpu_id: 0, id: 1, enabled: 1 }

_ { cpu_id: 1, enabled: 1 }
_ { enabled: 1 }
```

Lets assume the two queries are registered as subscriptions in the system and we publish the record with the three attributes `cpu_id`, `id`, and `enabled`. Both subscriptions match the provided record. We maintain an index for every attribute. These indices contain the IDs of currently registered subscriptions, if the subscription query contains the attribute in question. The index is implemented as a bitmap. A bit, corresponding to the subscription ID is set in the bitmap if the subscription contains the attribute. In the context of our example: In case we receive the record named `hw.apic.1`, the system will gather the bitmaps for all three attributes in the record and do a union of the three sets. Union is done by applying the bitwise OR operator on the bitmaps. We implemented the bitmap index using external C predicates working exactly like the index used for record retrieval.

For triggers and subscriptions, we also have the following special case where no attributes are given in a query:

```
_ {}
```

A query in this form will match with any record. We use a separate bitmap to track these subscriptions or triggers and always include it if we do a union over bitmaps.

4.6 Creating Synchronization Primitives with Octopus

One of the requirements for our service is the ability to be able to build synchronization primitives on top of it. In this section, we show our implementations, available in `liboctopus`, for locks, barriers and semaphores. An advantage of our service is that its API allows us to implement all the synchronization APIs on top of the described Flounder interface, without modifications on the server side. This should make it easier for programmers to write their own primitives, adjusted to their needs, in the future. You may notice that the provided recipes are akin to the algorithms used in Zookeeper or similar lock-services. The ideas are transferable to our service because our API uses the same ideas (especially triggers). In comparison with Zookeeper, our implementations require fewer lines of code. This reduction stems mostly from having a more dynamic trigger mechanism. Another advantage is that the THC binding allows us to wait for incoming asynchronous messages. This means we can build blocking synchronization primitives without the need for some form of thread-level synchronization because we do not need to coordinate with the event binding. In theory, we could achieve the same with a regular RPC binding, but the Flounder stub generator currently does not support this. On the other hand, it might be worth it to explore if the THC constructs can help to realise more sophisticated coordination algorithms in the future. In case we need to write asynchronous primitives, it is still possible to use the event binding to receive trigger events.

4.6.1 Locks

We used the Octopus key–value store to implement distributed locks. This type of distributed lock is intended for coarse grained locking (i.e., locks held for minutes–hours) as the overhead of the lock operation might be too large to use them for fine grained access control. Listing 1 shows the workings of the locking algorithm in pseudocode: We omitted some parts of the error handling, memory management and type declarations for simplicity. A client locking a resource will create a sequential record (i.e., a record whose name is appended with a unique, sequential increasing number), using a name previously agreed upon by all involved parties. Afterwards, the

client tries to find all existing lock records for this particular name. Because we used sequential records we can sort the retrieved record names to have a well defined order over all records waiting for the resource. The algorithm then uses this list like a queue: The position of our own lock record in the list tells us how many clients will receive the lock before us. If our record is the lowest entry, we hold the lock for the resource and can proceed. Otherwise, we issue an exist call on the record one below our position in the names array. We also pass a trigger along with this exists call. We program the trigger to be installed in case the record still exists at the time the query is executed on the server, and set it to trigger once the record is deleted. In case the record still existed, we wait for the trigger to notify us if the record gets deleted. The while loop ensures that a client, awakened by a deletion of a record not having the lowest sequence at the time, will not accidentally become the lock owner. Note that this code holds some desired properties: It is free of starvation and fair as the lock is always granted to the longest waiting client. It also does not suffer from a herd effect: In case we delete a record, only one involved party is woken up.

4.6.2 Barriers

Barriers are used to make sure the execution of code sections for multiple clients happens simultaneously. Listing 2 and Listing 3 show the implementation of a double barrier we built on top of Octopus. Every client who enters a barrier creates a sequential record. A `get_names` query is used to find out how many clients are currently waiting on the barrier. In case the threshold is not reached, the client waits for the creation of a special record, created by the last client that enters the critical section. This record is used to wake up all involved parties in the process. Leaving a barrier works the other way around: Each client deletes the record it created upon calling `oct_barrier_enter`, and waits until the extra record is deleted by the last client leaving the barrier.

4.6.3 Semaphores

Chips has integrated support for distributed semaphores, a feature used by `libposixcompat` in Barrelfish. To replace Chips (see Section 6.1), we implemented distributed semaphores on top of Octopus. Listing 4 shows the implementation in pseudocode. To increase the semaphore value by one (`oct_sem_post`), we create a sequential record. To decrease (`oct_sem_wait`), we try to delete one record. In case no record is found, or the record has been deleted between the `get` and `del` call, we wait until another record is created and try again. Note that this implementation is not optimal right now. It

suffers from a herd effect: In case one record is added using `oct_sem_post` we wake up all clients waiting for a trigger event in `oct_sem_wait`. We could avoid this by using our locking implementation to coordinate the execution of the while loop.

```
errval_t oct_lock(const char* lock_name, char** lock_record)
{
  oct_set_get(SET_SEQUENTIAL, lock_record, "%s. { lock: '%s' }",
              lock_name, lock_name);
  oct_read(*lock_record, "%s {}", &name);

  while (true) {
    oct_get_names(&names, &len, "_ { lock: '%s' }", lock_name);
    found = false;
    for (i=0; i < len; i++) {
      if (strcmp(names[i], name) == 0) {
        found = true;
        break;
      }
    }
    assert(found);

    if (i == 0) {
      // We are the lock owner
      return SYS_ERR_OK;
    }
    else {
      // Someone else holds the lock
      // Wait for our predecessor to delete
      t = oct_mktrigger(SYS_ERR_OK, OCT_ON_DEL,
      oct_BINDING_RPC, NULL, NULL);
      cl = oct_get_thc_client();
      cl->call_seq.exists(cl, names[i-1], t, &tid, &err);
      if (err_is_ok(err)) {
        cl->recv.trigger(cl, &tid, &fn, &mode,
                         &record, &state);
      }
    }
    // Our predecessor deleted his record;
    // need to re-check if we are really the lock owner now
  }
}

errval_t oct_unlock(const char* lock_record)
{
  return oct_del(lock_record);
}
```

Listing 1: Pseudocode for the locking algorithm. We show the two operations, lock and unlock, implemented in `liboctopus`.

```
errval_t oct_barrier_enter(const char* name, char** barrier_record,
                           size_t wait_for)
{
  oct_set_get(SET_SEQUENTIAL, barrier_record,
              "%s. { barrier: '%s' }", name, name);
  oct_get_names(&names, &current_records, "_ { barrier: '%s' }",
                name);

  if (current_records != wait_for) {
    t = oct_mktrigger(OCT_ERR_NO_RECORD, oct_BINDING_RPC,
                      OCT_ON_SET, NULL, NULL);
    cl = oct_get_thc_client();
    cl->call_seq.exists(cl, name, t, &tid, &err);
    if (err_is_ok(err)) {
      // Barrier already exists
    }
    if (err_no(err) == OCT_ERR_NO_RECORD) {
      // Wait until barrier record is created
      cl->recv.trigger(cl, &tid, &fn, &mode, &record, &state);
      err = SYS_ERR_OK;
    }
  }
  else {
    // We are the last to enter the barrier,
    // wake up the others
    oct_set(name);
  }

  return err;
}
```

Listing 2: Algorithm to enter a barrier in Octopus. Every client creates one record and checks how many barrier records already exist.

```
errval_t oct_barrier_leave(const char* barrier_record)
{
  oct_read(barrier_record, "%s { barrier: %s }", &rec_name,
           &barrier_name);
  oct_del(rec_name);

  err = oct_get_names(&names, &remaining_barriers, "_ { barrier: '%s' }",
                      barrier_name);
  if (err_is_ok(err)) {
    cl = oct_get_thc_client();
    cl->call_seq.exists(cl, barrier_name, t, &tid, &err);
    if (err_is_ok(err)) {
      // Wait until everyone has left the barrier
      cl->recv.trigger(cl, &tid, &fn, &mode, &record, &state);
    }
    else if (err_no(err) == OCT_ERR_NO_RECORD) {
      // already deleted
      err = SYS_ERR_OK;
    }
  }
  else if (err_no(err) == OCT_ERR_NO_RECORD) {
    // We are the last one to leave the barrier,
    // wake up all others
    err = oct_del(barrier_name);
  }

  return err;
}
```

Listing 3: Algorithm to leave a barrier. Every client will delete his record, created during `oct_barrier_enter`.

```
errval_t oct_sem_post(uint32_t id)
{
  return oct_mset(SET_SEQUENTIAL, "sem.%d. { sem: %d }", id, id);
}

errval_t oct_sem_wait(uint32_t id)
{
  t = oct_mktrigger(OCT_ERR_NO_RECORD,
                    oct_BINDING_RPC, OCT_ON_SET, NULL, NULL);
  char query[size];
  snprintf(query, sizeof(query),
           "r'sem\\.%d\\.[0-9]+' { sem: %d }", id, id);

  while (true) {
    cl = oct_get_thc_client();
    cl->call_seq.get(cl, query, t, &result, &tid, &err);
    if (err_is_ok(err)) {
      del_err = oct_del(result);

      if (err_is_ok(del_err)) {
        break; // Decreased successfully
      }
      else if (err_no(del_err) == OCT_ERR_NO_RECORD) {
        continue; // Race lost, need to start over
      }
    }
    else if (err_no(err) == OCT_ERR_NO_RECORD) {
      // No record found, wait until one is posted
      cl->recv.trigger(cl, &tid, &fn, &mode,
                       &trigger_result, &state);
    }
  }
  return err;
}
```

Listing 4: Pseudocode for the semaphore implementation. The post operation is implemented by creating a sequential record, the wait operation tries to delete a record.

Chapter 5

Performance Evaluation

This chapter describes the benchmarks we did for the Octopus service in Barrelfish and shows the achieved results. We also compare our service to Redis, which is similar to our service in terms of implementation.

5.1 Test Setup

We executed all benchmarks on a system equipped with two AMD Santa Rosa (Opteron 2200) CPUs running at 2.8 GHz on a TYAN Transport VX50 B4985 mainboard. Server and client were pinned to separate cores on the same package during our measurements.

5.2 Remarks

5.2.1 Parameter Tuning & Flags

For the following Benchmarks we changed some limits enforced by ECL^{i}PS^{e} CLP in order to support a large amount of records:

`DICT_HASH_TABLE_SIZE`**:** Increased from 8 192 to 2 097 152. Currently, record names and attribute names are stored as atoms in the system (see Section 2.2.1). This reduces the amount of garbage produced and allows us to do constant-time comparisons for strings. With a size of 8 192 entries this becomes a bottleneck in case large amounts of records are stored.

`DICT_DIRECTORY_SIZE`**:** Increased from 512 to 2 048. The amount of atoms that can be stored is limited by the size of this directory. A value of 2 048 lets us have a total of 2 097 152 atoms (in combination with the

default `DICT_ITEM_BLOCK_SIZE` of 1 024). Once this amount is reached, ECL^{i}PS^e fails to execute any queries that involve adding new atoms. The engine has a dictionary garbage collector that periodically removes atoms not needed anymore.

`HTABLE_MAX_SIZE`**:** Increased from 1 048 576 to 2 097 152. This defines the maximum size of the hash table used by the storage API to store records.

The ECL^{i}PS^e compiler has the `debug_compile` flag turned on by default. This leaves some debug instructions for tracing in the code, slowing down the execution. During our benchmarks we turned this feature off.

The garbage collector runs based on watermarks for the global and the trail stack. Collecting garbage caused a significant overhead in our measurements. We disabled the garbage collector for most of our experiments. However, for some long running experiments it was necessary to enable garbage collection otherwise ECL^{i}PS^e would fail. We discarded the outliers where the garbage collector ran, including a number of subsequent measurements that were disturbed as a result of the collection.

The Flounder stubs allow to enqueue only one message at any point in time in the messaging system. We overcame this limitation by building a queue on top of Flounder. During our measurements, we noticed that in some cases, replies from our server could not be sent directly and therefore had to be put in the queue, which caused an overhead of a few 1 000 cycles. This might seem counterintuitive at first as all communication between the client and server is done over RPC. A client blocks until he has received a response and cannot issue a new request. However, we used Flounder continuations to clean-up memory after a reply has been sent back to the client. This continuation is executed in the context of an event, so it is possible that a new request is issued from the client before the continuation is executed, therefore blocking the direct sending of the next reply. Because this is an issue of the current Flounder wait-set implementation we ignored these outliers in our experiments.

Another factor that had an impact on our measurements was the choice of the memory allocator. The current Barrelfish `malloc` and `free` implementation uses a simple free list and caused a lot of calls to `lesscore` & `morecore` for our allocation pattern. These calls are costly since we need to request memory and map it into the domains address space. We avoided this by using the dlmalloc memory allocator [24] which is more cautious in mapping and releasing memory.

5.2.2 ECL^{i}PS^e & Prolog

Initially, we tried to write most of the server functionality, including data storage, directly in Prolog. This allowed for fast prototyping but turned out to be very inefficient. Figure 5.1 shows the performance of a get-by-name call (i.e., a query that does not use a constraint for the record name) with increasing amounts of records stored. The first implementation uses the Prolog predicates `assert` and `retract` to store records as dynamic Prolog predicates. Dynamic predicates are stored in an ordered list by ECL^{i}PS^e, which means the time to retrieve a record should increase linear with the amount of records stored. However, our measurements showed that the cycles started increase significantly after 2 000 records. This approach was found to be too inefficient for our storage API.
The second implementation uses a hash table, written in Prolog, to store records based on their names as keys. As we can see, the latency for a single `get` call still increases linearly with the amount of records, even tough the algorithmic complexity of the underlying data structure suggests a constant access time. Listing 5 shows that 98 percent of the time in ECL^{i}PS^e is spent in garbage collection and `getval_body` (used to retrieve the hash table that stores all records). In order to support backtracking and logical unification, even for a simple `get` operation, Prolog needs to copy the whole hash table during `getval_body` and clean it up afterwards in `garbage_collect`. This effect becomes even worse in case we want to insert a record as we need to copy first to obtain the hash table containing all the records and copy again once we want to store the modified table. This implementation is a good example to showcase the problems when storing data in Prolog. To our knowledge, in ECL^{i}PS^e CLP it is not possible to avoid this without using external predicates written in C.

5.3 Storage API

5.3.1 Retrieve Performance

The performance of `get` operations depends on how the query sent to the server is formulated. The fastest queries are the ones that specify a record name in the query. In that case, only one record must be considered and can be found in constant time through a hash table lookup. Figure 5.2 shows, for a get-by-name call the look-up time is indeed constant for an arbitrary size of records stored in the system. The amount of records that can be stored is currently limited by the size of the directory ECL^{i}PS^e uses to store atoms. However, this limitation can be avoided by switching to strings for record names.

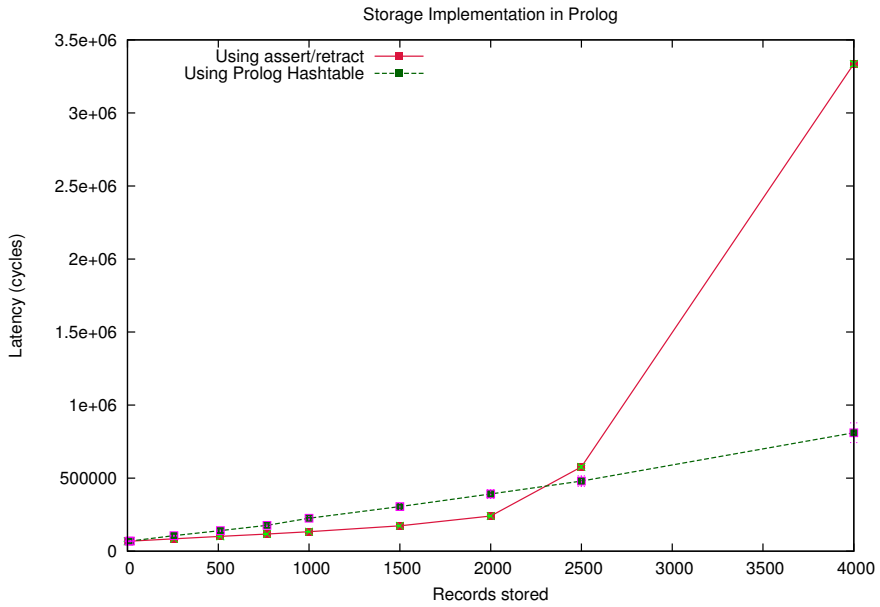

Figure 5.1: This graph shows the latency for a single `get` call while increasing the amount of records stored in the system for two different storage implementations written entirely in Prolog.

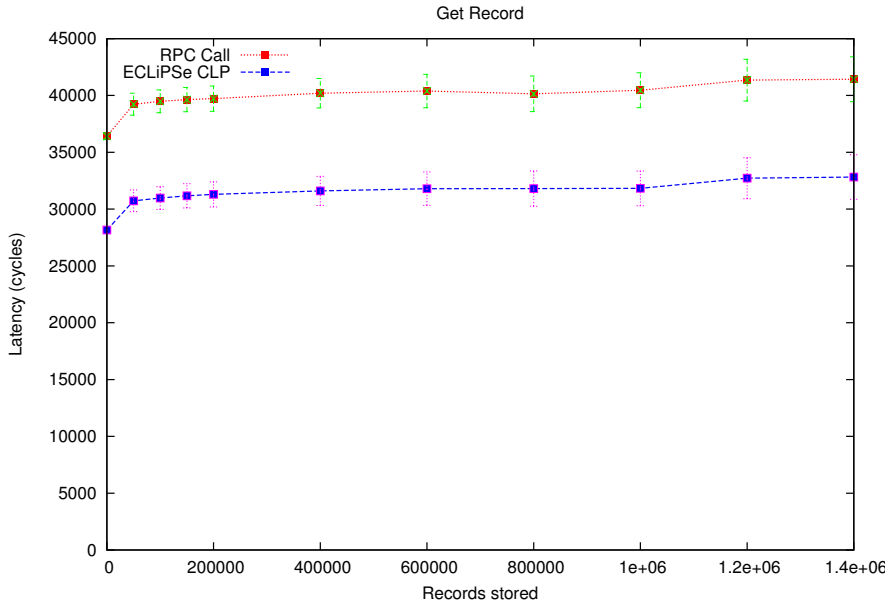

Figure 5.2: Latency to retrieve a single record with increasing amounts of records stored in the system. We show graphs for the total amount of cycles spent for the RPC call and the amount of time spent in ECL^{i}PS^{e} during the RPC call.

```
PROFILING STATISTICS
--------------------

Goal:   get_100000
Total user time:  6.28s

Predicate       Module     %Time   Time   %Cum
-------------------------------------------------------
garbage_collect   /0  sepia_kernel  54.5%   3.42s  54.5%
getval_body       /3  sepia_kernel  43.5%   2.73s  98.0%
term_hash         /4  sepia_kernel   0.6%   0.04s  98.6%
hash_find         /3  hash           0.3%   0.02s  98.9%
sort              /4  sepia_kernel   0.3%   0.02s  99.2%
get_object        /4  eclipse        0.2%   0.01s  99.4%
match_attribute   /2  eclipse        0.2%   0.01s  99.5%
hash_entry        /3  hash           0.2%   0.01s  99.7%
min1              /4  sepia_kernel   0.2%   0.01s  99.8%
member            /2  sepia_kernel   0.2%   0.01s 100.0%

Yes (6.33s cpu)
```

Listing 5: ECL^{i}PS^e profiling output for `get` operations while using a hash table, written in Prolog, for storage.

Figure 5.3 shows the cycles spent for individual parts of the system during a single get-by-name call. We can see that about 75 percent of the time is spent in ECL^{i}PS^e. The parser, memory allocation and Flounder message passing are responsible for the other 25 percent. These three areas should remain constant throughout all queries and RPC calls whereas the ECL^{i}PS^e CLP system will require even more cycles for more expensive queries. This graph shows that the best way to achieve higher performance would be to try and improve the code executed by ECL^{i}PS^e. Profiling the ECL^{i}PS^e code reveals that the most expensive function is `store_get`, used to retrieve a record from the hash table (see Listing 6). This is as expected and shows that our implementation does suffer from any other bottlenecks; for example, due to memory management.

In Figure 5.4, we see a benchmark for `get` queries where no exact record name is given and the record is identified only by a set of attributes. For this experiment we added up to 20 000 records to the system, all with a total of 5 attributes chosen at random from a set of 5, 10, 15 and 20 attribute names. We measured the time to get a single record identified by 5 attributes. We made sure that the record we are trying to retrieve has a higher lexicographical ordering than the others. This means we always traverse the whole index

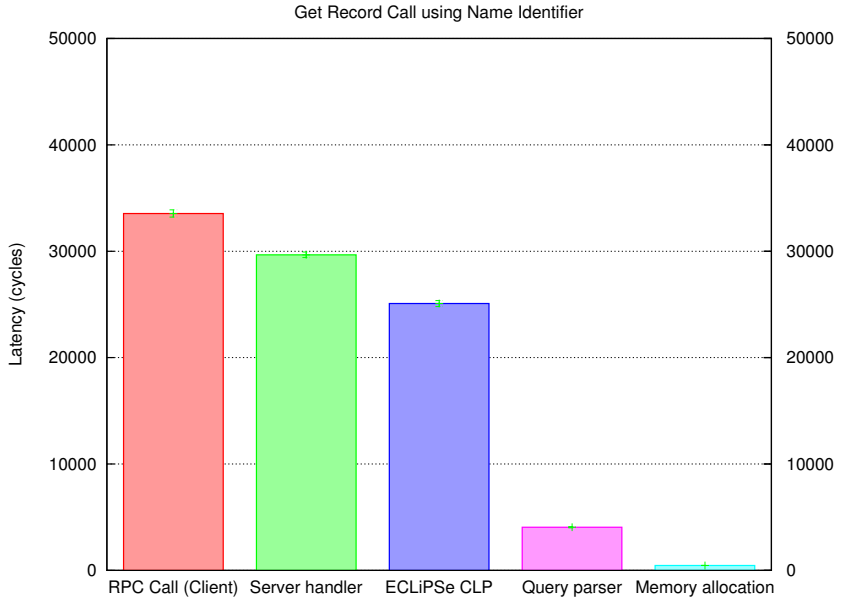

Figure 5.3: Overview of costs for a single get by name call.

to find the record, therefore we simulate a worst case scenario. The graph labelled as "5 Arguments" resembles the baseline: In that case our index does not help at all because all records contain the same 5 attributes and we have to match all records in order to find our record of interest. The graph made from records with a set of 20 attributes represents the other extreme: We are looking at $\binom{20}{5} = 15\,504$ different attribute combinations for our records. In the experiment, the index intersection only returned one result for our red graph in all cases. The linear growth we see is the increasing cost of the intersection algorithm due to growing indexes (for 20 000 records our index contains about 5 000 records per attribute). The reader should note that this is a somewhat artificial example. In reality, we usually have multiple groups of records with disjoint sets of attributes instead of a uniform distribution of records among all attributes. The index should be even more useful in that case.

5.3.2 Add Performance

Because we have to maintain our index for every record we insert, the insertion cost increases with the amount of records stored. Figure 5.5 shows a benchmark where we measured the cost of adding one record to the system while varying the amount of records already stored. We made sure that the name of the inserted record has a higher lexicographical ordering to ensure our record name is always placed at the end of the index. The graphs show,

```
PROFILING STATISTICS
--------------------

Goal:   get_1000000
Total user time:  0.48s

Predicate       Module     %Time   Time   %Cum
-------------------------------------------------------
store_get_        /4  sepia_kernel  29.8%   0.14s  29.8%
get_object        /4  eclipse        8.5%   0.04s  38.3%
match_object      /3  eclipse        8.5%   0.04s  46.8%
do__2             /1  eclipse        8.5%   0.04s  55.3%
sort              /4  sepia_kernel   8.5%   0.04s  63.8%
match_constraints /2  eclipse        6.4%   0.03s  70.2%
garbage_collect   /0  sepia_kernel   6.4%   0.03s  76.6%
append            /3  sepia_kernel   6.4%   0.03s  83.0%
make_all_constrain/3  eclipse        4.3%   0.02s  87.2%
maplist_body      /4  lists          4.3%   0.02s  91.5%
min1              /4  sepia_kernel   4.3%   0.02s  95.7%
get_by_name       /3  eclipse        2.1%   0.01s  97.9%
sort              /2  sepia_kernel   2.1%   0.01s 100.0%

Yes (0.53s cpu)
```

Listing 6: ECL^{i}PS^{e} profiling output for `get` operations.

with a sufficiently large level for the skip list (i.e., the maximum amount of forward pointers per node) the index insertion cost is negligible.

5.3.3 Comparison with Redis

The current implementation makes our system comparable to Redis [30]. Although Redis is a more mature systems with more features they still share some similarities:

1. Both use single-threaded servers (Redis supports replication by launching multiple instances)
2. Both are mainly used for in-memory data storage (Redis has some support for persistent storage)
3. All commands sent involve IPC roundtrips

For this Benchmark, we measured the throughput of a simple `get` operation with a 256 byte payload. We ran Redis (version 2.4.7) on Linux 2.6.32

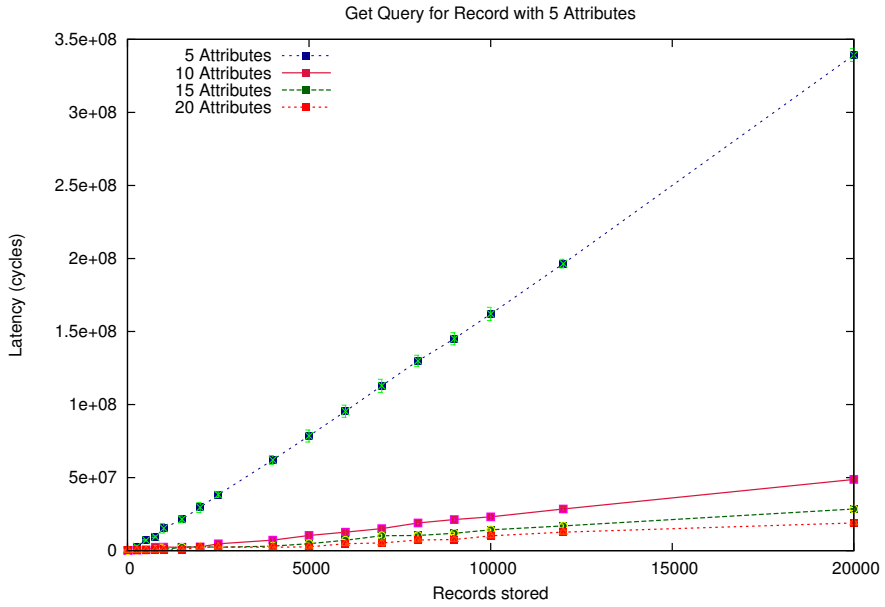

Figure 5.4: Benchmark for a `get` call of one record identified by 5 attributes with increased amounts of records stored.

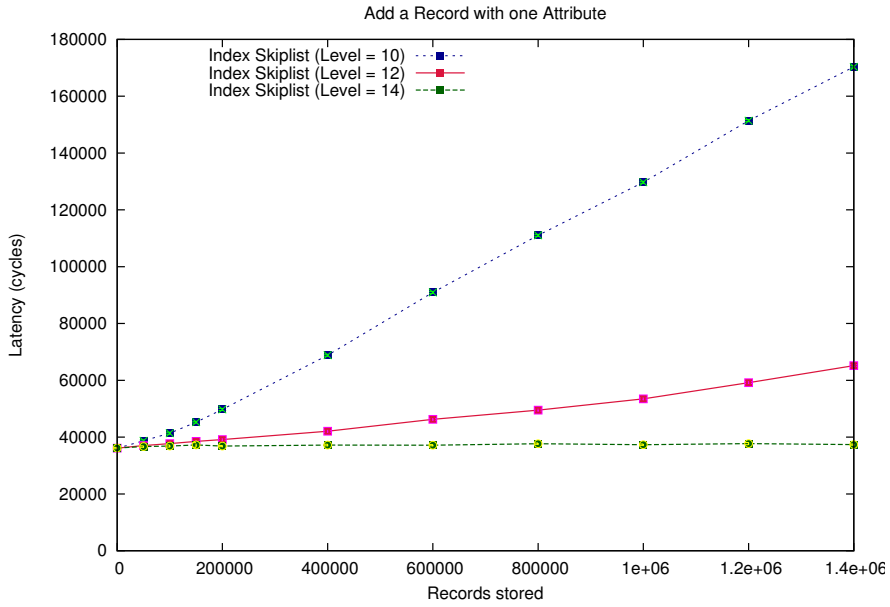

Figure 5.5: This graph shows the impact the amount of already stored records has on adding a new record while varying the maximum amount of skip pointers.

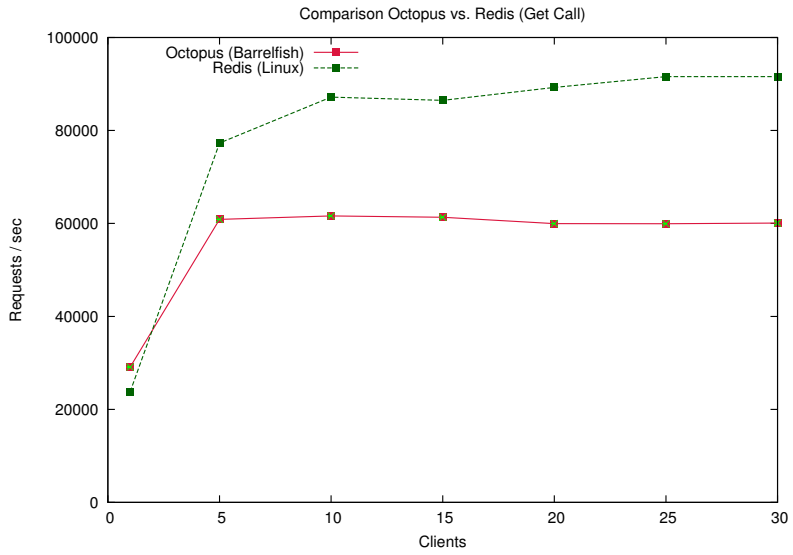

Figure 5.6: Comparison of Octopus and Redis showing the number of performed Get operations for each system with increasing amounts of clients.

pinned to one core. We used the provided `redis-benchmark` program to measure the throughput of `get` operations. We configured Redis to use a Unix domain socket for the communication between the client and server. Please note that this Benchmark does not provide a very good comparison between the two services. We are running on completely different operating systems with different IPC mechanisms and costs involved for them. In a fairer comparison we expect Redis to perform even better since the send–receive process with Unix domain sockets includes an additional system call as opposed to UMP's shared memory implementation. However, the graph (Figure 5.6) shows that on the same hardware the peak of operations for Redis is at about 90 000 whereas for Octopus it is only at 60 000. What must be considered here is that just the start-up of the ECL^{i}PS^{e} engine yields a considerable overhead which impacts the throughput of our service. We ran some tests where we executed a simple query on the engine (i.e., `true.`) and measured about 6 000 cycles for its execution. At the current scale Barrelfish is operating we expect Octopus to be efficient enough. In case the need for a faster service arises in the future, the best way to improve performance would be to avoid using an intermediate language such as Prolog.

Chapter 6

Case Study: Applying Octopus in Barrelfish

This chapter explains how we used Octopus to restructure the Barrelfish boot process. We replaced the name server implementation and wrote a device manager for Barrelfish. With the help of Octopus, the device manager is capable to find out about installed devices and start the appropriate drivers.

6.1 Replacing Chips

In order for programs to connect to one another, Barrelfish uses a name server (Chips). Clients interact with Chips, to store and retrieve interface references (irefs) based on names. In addition, Chips has an API for blocking lookups on irefs: This allows programs to wait until a specific service is running and ready to accept connections. Due to the lack of proper distributed synchronization primitives, Chips is also used for synchronization: Programs register dummy service references to signal certain events in the system. Those interested in the event can do a blocking lookup on this particular service reference to wait for the signalization of the event. Using this technique, it is even possible for a programmer to build barriers by having one client act as a coordinator.
Octopus is a perfect fit for all these tasks, so we decided to replace Chips. Service references are now stored as records. This allows us to implement all of the previous functionality Chips offered (i.e., lookup and waiting for service references to appear). With Octopus, it is also possible to extend the existing name service client API in the future; for example to store additional information along with the iref, such as the type of the interface or to get notifications in case a service record is changed or deleted. One limitation of the current implementation is that we use a regular Flounder

RPC binding to invoke calls on the Octopus server instead of the THC binding provided by `liboctopus`. Some service reference lookups are performed early in the start-up code. In that phase, memory allocation is limited to a maximum of one page per allocation. However, if we use a THC binding, we need to allocate a much larger stack. This is a limitation of the current Barrelfish memory server and is likely to be fixed in the future, rendering the extra binding unnecessary.

6.2 PCI Refactoring

Barrelfish had one service, called pci, to initialize, configure and program ACPI, PCI and I/O APICs. From an architectural standpoint it made sense to decouple pci into three different services. This allows to evolve them independently while using well defined Flounder interfaces for communication. The architecture should also be more flexible in the event of hardware changes such as the introduction of a completely new bus or interrupt controller. We explain the new architecture based on Figure 6.1: ACPI gathers information about installed CPU cores (in the form of local APICs), PCI root bridges, interrupt overrides and I/O APICs and stores this informations in the SKB. PCI does bus enumeration and is responsible for programming the PCI BARs. It uses the SKB to aid with the configuration. PCI also needs to talk to ACPI during bus enumeration (in case a new PCI root bridge is found) and to register interrupts for specific devices. PCI exports an interface that allows drivers (e1000 or serial in this example) to retrieve capabilities to access the memory regions of the device. I/O APIC is used to enable device interrupts and route them to individual local APICs. I/O APIC, PCI and ACPI need to talk to the monitor to get the capabilities for physical address or I/O regions. Of course, in reality every user-space program has a connection to the monitor, but in our diagram we restricted the connections and show only the ones relevant for the explanation. Fish is the Barrelfish shell. It communicates with ACPI to provide commands for reset and shutdown. Fish also needs to talk to the serial driver to handle input and output. Kaluga is the new device manager. It makes sure everything is started in correct order (see Section 6.3).

6.3 Kaluga Device Manager

We built the Kaluga device manager for Barrelfish based on the Octopus service. Until now, a user had to add drivers manually to the bootscript, therefore requiring a great deal of knowledge about the installed hardware. With Kaluga it is now possible to start drivers automatically in case the

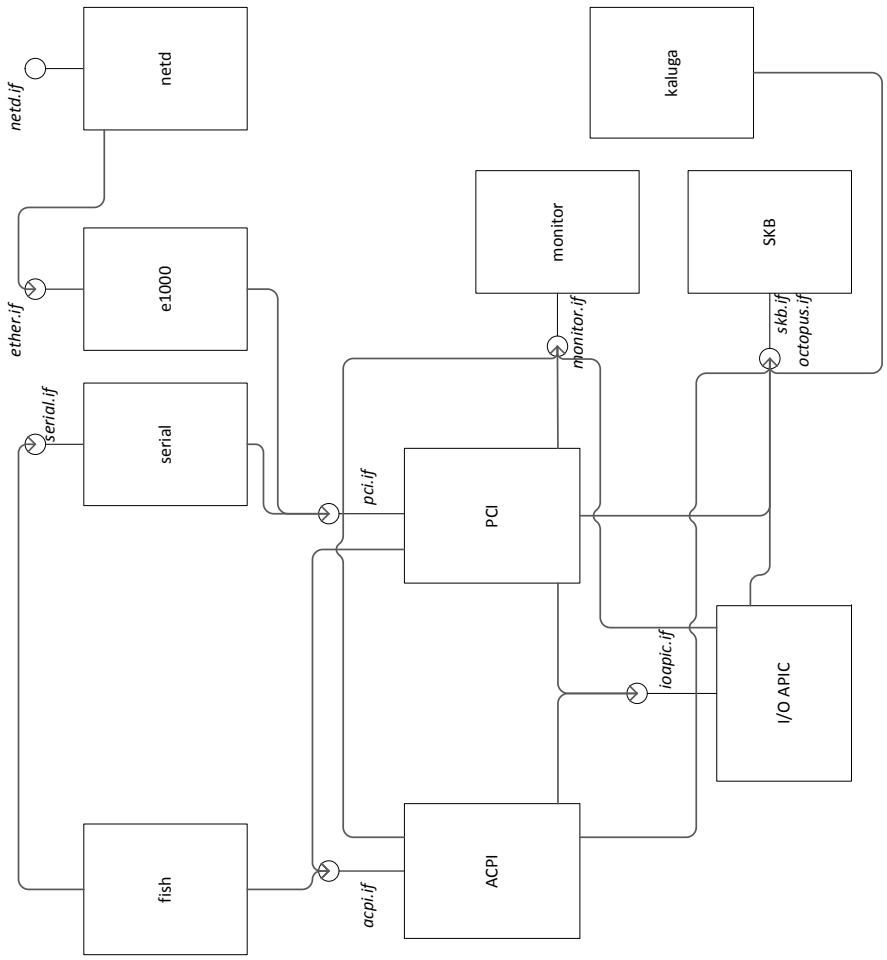

Figure 6.1: Architectural overview after the PCI refactoring. The diagram shows the different services running as user-space programs in the system, the exported interfaces for each service and the connections between services.

required hardware is found at startup or later in the event of hot plugging. Octopus allows Kaluga to be generic: The service is currently used to start PCI drivers as well as CPU drivers but can easily be extended — for example to support USB. In the following sections, we explain the current state of work and in which direction we want to go from there.

6.3.1 General Design

Deciding what drivers an operating system can start depends mostly on two factors: The available hardware and available driver binaries. The key idea for the Barrelfish architecture is to structure the OS like a distributed system. A distributed system should handle node removals, failures or attachments. In terms of device drivers this means for example the removal of a PCI card, plugging in USB devices or failure of a core, all while the system is up and running. These requirements made Octopus a good fit to use in a device manager. When Kaluga starts, its first task is to parse the `menu.lst` boot script to find out about all available driver binaries. Currently, the filesystem support in Barrelfish is still limited. We do not allow driver binaries to be added at runtime. However, if a real filesystem is used in the future, we can extend the current system to watch out for new drivers added at run time. After parsing the boot script, Kaluga starts ACPI. ACPI will then add specific Octopus records for every PCI root bridge, local APIC and I/O APIC it finds. Meanwhile, Kaluga will place various `get_names` calls to find the records, but also register triggers to receive events about forthcoming changes concering these records. As an example, we provide the following query used by Kaluga to watch for I/O APICs:

```
r'hw\\.ioapic\\.[0-9]+' { id: _, address: _, irqbase: _ }
```

In case a record is found, Kaluga will react by starting the appropriate drivers: If Kaluga finds an I/O APIC record it will start the I/O APIC driver, in case a PCI root bridge record is found it will start the PCI bus driver. For every local APIC record, Kaluga will send a core boot request to the monitor. This scheme stays the same for PCI: Once the PCI bus driver is started, it will most likely discover new devices and thereupon add records for each PCI device. Kaluga will pick up the records, either through the initial `get_names` call or through a trigger event.

6.3.2 SKB Driver Entries

Kaluga needs to figure out which driver to start for a given record. For this reason, we added a mapping database to the SKB. The following snippet is an entry made for the current AHCI driver:

```
pci_driver{
    binary: "ahcid",
    supported_cards:
    [ pci_card{ vendor: 16'8086, device: 16'2922,
                function: _, subvendor: _, subdevice: _ },
      pci_card{ vendor: 16'8086, device: 16'3a22,
                function: _, subvendor: _, subdevice: _ },
      pci_card{ vendor: 16'1002, device: 16'4390,
                function: _, subvendor: _, subdevice: _ } ],
    core_hint: 0,
    interrupt_load: 0.5,
    platforms: ['x86_64', 'x86_32']
}.
```

In case Kaluga receives a PCI device record, it will query the SKB to figure out which binary to start and on what core. Kaluga will read information such as vendor and device ID from the record and pass it on to the SKB. The SKB then tries to find a matching entry by looking at the `supported_cards` argument in the driver entries. `core_hint`, `interrupt_load` and `platforms` are currently not of much use. In the future, we plan to include these attributes to evaluate on which core a driver should run. `interrupt_load` is intended to help place drivers that are expected to produce heavy interrupt traffic (e.g., a 10 Gbit/s Ethernet driver) on a separate core whereas a mouse and keyboard driver can easily run on the same core. Right now, the query will just use the `core_hint` argument as the core recommendation for Kaluga.

6.3.3 Starting Drivers

Once Kaluga knows which driver to start there is also the question of how to start the driver. The answer can vary heavily depending on the device: A CPU driver is started by a boot-core request message to the local monitor. A network drivers always needs an instance of netd running beside the driver. At last, we have the issue with multiple devices of the same type in a system. Do we start multiple instances of a driver — as we do for CPU cores — or is one driver capable to handle all devices? We decided to allow for custom start functions for every binary found by Kaluga. The default start function, set for each binary on parsing the boot script, will just start every driver exactly once by using the spawn daemon. However, for CPU drivers or network drivers we can override this behavior: To start a CPU driver we send a message to the monitor. For network cards, we start the driver and an instance of netd. For most cases a driver programmer does not have to

worry about custom start-up functions as the default start function should suffice. Therefore, all he needs to do is to add a driver entry to the SKB.

Chapter 7

Related Work

Coordination in distributed systems has been addressed by an abundance of research. With the massive scale out in the days of the Internet, the trend continues towards distributed systems with thousands of machines located all around the world. Thus, leading to the development of highly sophisticated coordination services in the form of data-center lock managers, key–value data stores or publish–subscribe systems. This chapter gives an overview of existing research and systems related to this thesis.

7.1 Publish–Subscribe Systems

The publish–subscribe paradigm is a well-established interaction pattern and allows for flexible communication within a distributed environment, taking into consideration the decoupled nature of distributed systems. A typical publish–subscribe system knows two different types of interacting parties: A publisher, responsible for the generation of events and a subscriber, consuming the events generated by a publisher. *Eugster et al.* [22] give an overview and characterization of different publish and subscribe systems. They show that the publish–subscribe paradigm features 3 key elements:

Space decoupling: The interacting parties do not need to know each other.

Time decoupling: Interacting parties do not need to be actively participating at the same time.

Synchronization decoupling: Publishers are not blocked while producing events, and subscribers can get asynchronously notified while performing some concurrent activity.

Further, they categorize different systems into three categories:

Topic-based systems [2, 30] lets consumers subscribe to given topics (usually identified by a name) and producers generate events for given topics. Systems can support hierarchical or flat identifiers. Hierarchical addressing allows for more flexibility as topic relations can be expressed by the programmer. Topic-based systems are very similar to group communication [18].

Content-based systems [8] lets consumers use constraints to specify their subscriptions based on the content of the published messages. This allows for more expressiveness at the cost of additional complexity. Constraints (also known as filters) usually consist of logically combined comparisons. They are passed to the publish–subscribe system as regular strings, formulated in an intermediate language such as SQL or by writing executable code directly in the form of filter objects or functions.

Type-based systems make use of a better integration with object-oriented, high-level languages by directly associating events with types. Systems can exploit the static type checking at compile time to guarantee more safety.

Note that these three categories are not necessarily mutually exclusive. So called hybrid systems [16] do exist. Usually this means that the system has the characteristics of a content-based system combined with topic-based or type-based behavior.

7.2 Generative Communication

In 1985, David Gelernter proposed a new concurrent programming model (generative communication) [10] as opposed to already existing models such as shared variables, message passing and remote procedure calls (RPC). The basis for generative communication is a so called tuple space (TS). Concurrent processes that make up a distributed program usually share a common TS. Three simple functions `out()` (adds a tuple to the TS), `in()` (reads and removes a tuple from the TS, blocks if no matching tuple is found) and `read()` (reads a tuple without removing it in the TS) form the basic operations defined over a tuple space. A programmer can use these operations to build more complex interaction patterns. One advantage of generative communication is the delocalization of information. In comparison with publish–subscribe, tuple spaces share the two properties space and time decoupling. However, they do not exhibit synchronization decoupling as removing tuples from the TS is a synchronous operation. Tuple spaces have been implemented for many languages and platforms [31].

7.3 Data Center Managers

With the scale out of modern data centers, systems like Chubby [5] and ZooKeeper [13] evolved in order to provide coordination and synchronization for a large amount of machines. They store information in a hierarchical name space (i.e., like a file system) and export an API similar to the ones used for file manipulation. Zookeeper and Chubby are used as a multipurpose tool for various coordination tasks such as configuration management and storage, rendezvous, group membership, leader election, locking and barriers. Chubby is also starting to replace DNS as a name server internally at Google. Both systems use replication to achieve high availability and a form of Paxos [15] for consensus among all nodes.

7.4 Key–Value Stores

Key–value stores started to appear in order to support the massive scale of todays Internet applications. The idea is to go away from using the complex querying capabilities and costly retrieve operations of relational database management systems (RDBMS) and use key–value stores where data can be retrieved by simple primary key lookups. Most systems favor speed and simplicity over strong guarantees such as ACID in traditional data stores. Key–value store is a very general term for a wide range of systems with different purposes: Distributed key–value stores [7, 9] usually implement a form of distributed hash table (DHT). Their goal is to support massive scale and be eventually-consistent. On the other hand, systems like Redis [30] try to store all data in RAM on a single machine (although replication is supported using a master–slave pattern) with only optional support for persistence. Key–value stores allow to store data in a schema-less way. What kind of data is stored can vary: Redis for example is often referred to as a data structure server since it can store strings, hashes, lists and sets — all types that can also be stored in a programming language. Other, more document-oriented systems like MongoDB [28] allow to store semi-structured data (i.e., JSON or XML). MongoDB or other searchable key–value stores [26] are also able to perform lookups based on the values of an entry. However, they usually do not have support for complexer operations such as joins in SQL.

7.5 Singularity Driver Manifests

Singularity is an operating system written in Sing# on top of the .NET CLR. In Singularity, device driver manifests [21] are attached to the the respective driver assembly. Manifests are written in XML and contain a description of

hardware and software preconditions to run a driver. This design enables Singularity to learn the requirements for starting a driver, or possible conflicts with other drivers, without executing the driver code first. Verification can be done at compile time or run time.

Chapter 8

Conclusion

In this thesis we introduced Octopus, a coordination service for Barrelfish. Octopus is influenced by ideas in the distributed computing area. The service allows for coordination and event delivery in a multikernel operating system such as Barrelfish. It encompasses characteristics of key–value stores, publish–subscribe systems and data center lock managers. Octopus can store information and its API allows clients to build synchronization primitives. We give a reference implementation for locks, barriers and semaphores. Octopus is integrated and implemented on top of the Barrelfish operating system and uses existing mechanisms in Barrelfish such as the SKB, THC and Flounder. Our performance measurements have shown that we exhibit horizontal scaling for lookups based on keys. We investigated the benefits and drawbacks of using the SKB and basing our work on the embedded high-level CLP language in the SKB. While using a high-level language we benefited from being able to rapidly prototype different implementations and were able to rely on the expressiveness of a declarative, dynamic language. Because the main drawbacks were mostly performance related, we made some optimizations for lookups based on record values, triggers or subscriptions and show that they help to improve the performance.
As a case study, we used Octopus in Barrelfish to replace Chips and to write Kaluga, a device manager that helps with booting the OS. Kaluga uses Octopus to watch for changes in hardware and reacts appropriately to the changes. We modified some areas of the PCI and ACPI code to be compatible with Kaluga. By using Octopus, Kaluga is decoupled from the PCI and ACPI implementation and only sees the relevant information in the form of records. This allows to evolve all three services independently in the future. We think the case study proved that Octopus is well suited for coordination tasks in an operating system. It allowed us to write code that requires relatively complex interaction and synchronization among processes with little overhead, therefore speeding up the development process for Kaluga or the

Chips replacement.

8.1 Future Work

While the current state of work is already usable and a useful contribution to Barrelfish, there are still areas for further improvements. Capabilities are an important data type used in various places in Barrelfish. Right now, we have no solution to store capability references in Octopus. It would have been nice to use iterators instead of the current `get_names` call. This was not possible due to limitations in ECL^{i}PS^e. For performance, there is still a lot of room for improvements by applying ideas from traditional key–value stores. However, as long as we rely on the SKB as the underlying implementation, there will always be a tradeoff between expressiveness and performance. We currently have no access control for records. This makes it hard to have a reliable system as every process can modify or delete records. A possible way would be to use a designated record attribute to encode access restrictions or use capabilities to restrict record manipulations. To react to program failures, transient records (i.e., records that are removed automatically once their creator does no longer exist), would be useful and allow to build more reliable distributed services for Barrelfish. We were not able to implement this yet as the current process management does not allow to monitor dispatchers.

Octopus also opens a lot of possibilities for additions and new functionality in Barrelfish: The ACPI, PCI as well as USB management code can be adapted for hot plugging support. Kaluga should already be prepared to catch these events and act appropriately. Device drivers can be modified to react to changes in hardware by watching records. Octopus can work in a dynamic environment: Programs can generate records and queries at runtime. It may be worth it to explore how this can be exploited to reconfigure Barrelfish at runtime instead of having to reboot. Distributed services in Barrelfish can make use of Octopus to solve problems such as leader election or configuration management. The Prolog language would be a good fit to extend the current implementation and write algorithms that reason about the state of records — for example, to detect deadlocks by analyzing lock records at run time.

Bibliography

[1] Abderrahamane Aggoun, David Chan, Pierre Dufresne, Eamon Falvey, Hugh Grant, Warwick Harvey, Alexander Herold, Geoffrey Macartney, Micha Meier, David Miller, Shyam Mudambi, Stefano Novello, Bruno Perez, Emmanuel van Rossum, Joachim Schimpf, Kish Shen, Periklis Andreas Tsahageas, and Dominique Henry de Villeneuve. *ECLiPSe User Manual, Release 6.0*, February 2012.

[2] M. Altherr, M. Erzberger, and S. Maffeis. iBus - a software bus middleware for the Java platform. In *Proceedings of the Workshop on Reliable Middleware Systems of IEEE SRDS'99*, pages 43–53, 1999.

[3] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM.

[4] Andrew Baumann, Simon Peter, Adrian Schüpbach, Akhilesh Singhania, Timothy Roscoe, Paul Barham, and Rebecca Isaacs. Your computer is already a distributed system. why isn't your OS? In *Proceedings of the 12th conference on Hot topics in operating systems*, HotOS'09, pages 12–12, Berkeley, CA, USA, 2009. USENIX Association.

[5] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[6] Intel Corporation. Intel 82093AA I/O Advanced Programmable Interrupt Controller (I/O APIC).

[7] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, volume 41 of *SOSP '07*, pages 205–220, New York, NY, USA, 2007. ACM.

[8] Patrick Th. Eugster and Rachid Guerraoui. Content-based publish/subscribe with structural reflection. In *Proceedings of the 6th conference on USENIX Conference on Object-Oriented Technologies and Systems - Volume 6*, pages 10–10, Berkeley, CA, USA, 2001. USENIX Association.

[9] Roxana Geambasu, Amit A. Levy, Tadayoshi Kohno, Arvind Krishnamurthy, and Henry M. Levy. Comet: an active distributed key-value store. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–13, Berkeley, CA, USA, 2010. USENIX Association.

[10] David Gelernter. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, January 1985.

[11] Tim Harris, Martin Abadi, Rebecca Isaacs, and Ross McIlroy. Ac: composable asynchronous io for native languages. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 903–920, New York, NY, USA, 2011. ACM.

[12] Hewlett-Packard Corporation, Intel Corporation, Microsoft Corporation, Phoenix Technologies Ltd., Toshiba Corporation. Advanced Configuration and Power Interface Specification, 2010.

[13] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, page 11, Berkeley, CA, USA, 2010. USENIX Association.

[14] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an operating-system kernel. *Commun. ACM*, 53:107–115, June 2010.

[15] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[16] Y Liu and Beth Plale. Survey of publish subscribe event systems. *Indiana University Department of Computer Science*, (TR574):1–19, 2003.

[17] Antoni Niederlinski. *A Quick and Gentle Guide to Constraint Logic Programming via ECLiPSe*. Jacek Skalmierski Computer Studio, 2011.

[18] David Powell. Group communication. *Commun. ACM*, 39(4):50–53, April 1996.

[19] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33:668–676, June 1990.

[20] Adrian Schüpbach, Andrew Baumann, Timothy Roscoe, and Simon Peter. A declarative language approach to device configuration. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, pages 119–132, New York, NY, USA, 2011. ACM.

[21] Michael F. Spear, Tom Roeder, Orion Hodson, Galen C. Hunt, and Steven Levi. Solving the starting problem: device drivers as self-describing artifacts. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 45–57, New York, NY, USA, 2006. ACM.

[22] Patrick Th, Pascal A. Felber, Rachid Guerraoui, and Anne M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.

[23] Bison: GNU parser generator. `http://www.gnu.org/software/bison/`. [Online; accessed 03-March-2012].

[24] dlmalloc: A Memory Allocator by Doug Lea. `http://g.oswego.edu/dl/html/malloc.html`. [Online; accessed 03-March-2012].

[25] Flex: The Fast Lexical Analyzer. `http://flex.sourceforge.net/`. [Online; accessed 03-March-2012].

[26] HyperDex: A Searchable Distributed Key-Value Store. `http://hyperdex.org/`. [Online; accessed 03-March-2012].

[27] JSON: JavaScript Object Notation. `http://www.json.org/`. [Online; accessed 03-March-2012].

[28] MongoDB: A Scalable, High-performance, Open Source NoSQL Database. `http://www.mongodb.org/`. [Online; accessed 03-March-2012].

[29] PCRE: Perl Compatible Regular Expressions. http://www.pcre.org/. [Online; accessed 03-March-2012].

[30] Redis: An Open Source, Advanced Key-Value Store. http://redis.io/. [Online; accessed 03-March-2012].

[31] Tuple space: Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Tuple_space. [Online; accessed 03-March-2012].