Unifying Synchronization and Events in a Multicore OS

Gerd Zellweger, Adrian Schüpbach, and Timothy Roscoe
Systems Group, Department of Computer Science, ETH Zurich

Abstract

In this paper, we argue that an operating system structured as a distributed system needs a coordination and a name service to make OS services work correctly. While a distributed structure allows applying algorithms from the distributed field, it also suffers from similar problems like synchronization, naming, distributed locking and coordination of service instances.

Octopus, our solution to this problem in the context of a real OS, provides an easy-to-use, high-level, uniform coordination service with events at reasonable performance. Based on this service, we describe three real use cases: device management, OS service registry and boot-up coordination.

1 Introduction

This paper reports on early work to determine the usefulness of abstracting service dependencies and synchronization from the services themselves in the context of an OS for future multicore systems.

Modern OS designs face serious challenges in the face of hardware trends. First, as core counts increase, it becomes essential for performance scalability to distribute, partition, or replicate OS state across cores. In conventional monolithic systems like Linux or Windows, this is done with per-processor data structures, whereas recent "multikernel" approaches like fos [15] and Barrelfish [2] explicitly structure the OS as a distributed system with no shared state between nodes.

APSys '12, July 23-24, 2012, Seoul, S. Korea

Second, increasing heterogeneity of cores, memory, and peripherals in a modern computer, plus growing diversity of hardware platforms, leads to large quantities of specialized code inside the OS in the form of drivers, services, resource allocators, and the like. With the advent of heterogeneous general-purpose cores, this will only intensify.

These factors lead to OS designs with a large number of components with complex interdependencies. Services must be started in an order which respects their dependencies and, preferably, minimizes startup latency. As devices (and cores) come and go, drivers must be started up and shut down. Effective power management requires knowledge about device dependency: shutting down a USB controller or PCI bus should only be done if the dependent devices are safely shut down as well.

Moreover, the OS now has complex synchronization requirements between components: hotplug events may involve careful coordination between PCI managers, ACPI subsystems, and device drivers, and schedulers and code to route interrupts may need to wait until all cores on a machine have successfully started up before proceeding.

In existing systems, resolving these dependencies and implementing synchronization patterns between OS components and models is typically hard-coded into the components themselves, and is a leading factor in OS complexity (and, therefore, reliability and correctness).

We report on early experience applying techniques devised for modern cluster-based distributed systems to this problem. We built Octopus, a coordination service for Barrelfish, inspired by facilities such as Chubby [3] and Zookeeper [8].

We describe the background in the next section, and in Section 3 we discuss the requirements of such a service, derive design principles, and present the design of Octopus. Section 4 presents our ex-

perience so far, in the form of three cases where the abstractions provided by Octopus have enabled it to replace considerably more complex code in the OS: bootstrapping, device management, and the system name service. We also briefly present performance measurements, and conclude in Section 5.

2 Background

Our ideas in this paper are part of a wider trend to apply ideas from distributed computing to multicore OS design. Traditionally, data centers have faced complex coordination problems at the level of distributed systems on clusters. Chubby [3] and Zookeeper [8] provide coordination and synchronization for large collections of machines. They organize information in a hierarchical name space and export a file system-like API. Zookeeper and Chubby are used as a multipurpose tool for various coordination tasks such as configuration management, storage, group membership, leader election, locking and mutual exclusion. Somewhat unexpectedly, Chubby also increasingly replaced DNS as a general-purpose name server internally at Google. Both systems use state-machine replication to achieve high availability, using variants of Paxos [9] for consensus among nodes.

With the increasing demand of fast access to data at massive scale, key-value stores favor simplicity in terms of data model and query complexity over strong guarantees such as the ACID properties. Distributed key-value stores implement a form of distributed hash table [4, 7], providing eventual consistency. Redis [17] is one example of a centralized RAM-based key-value store with optional master-slave replication and persistency. It aims to be lightweight and high-throughput, and stores schema-less data under keys. Redis provides a flexible set of atomic operations on single data items.

Publish-subscribe systems allow flexible interaction in distributed systems and feature three key ideas [5]. First, *space decoupling* means that interacting parties do not need to know each other. Second, *time decoupling* means that interacting parties do not need to be actively participating at the same time. Finally, *synchronization decoupling* means that publishers never block on generating data and subscribers get asynchronous data events.

In the OS context, D-Bus [6] is an interprocess communication facility for Linux and other operating systems which also supports a limited from of coordination: processes can wait for events from specific objects, and the D-Bus daemon can start processes when messages are sent to them.

Our work is implemented in Barrelfish [2]. Barrelfish is a multikernel: a distributed architecture where the OS boots individually on each core. OS services are distributed over hardware nodes and state consistency is ensured by explicit messaging. Barrelfish includes a *system knowledge base* (SKB) [13], a service based on Constraint Logic Programming which stores hardware and software knowledge declaratively. Octopus builds on the SKB and uses logical unification and reasoning as part of the coordination service. The result is a flexible key-value store with publish-subscribe patterns on persistent and transient data.

3 Coordinating an OS

Our coordination service, Octopus, is based on the interfaces provided by Chubby and Zookeeper with the goal of facilitating distributed coordination and event handling while reducing the code complexity involved in programming such functionality. However, the OS environment is somewhat different from a large cluster, and so our requirements and design principles are somewhat different.

First, at least in medium term, an OS can assume a reliable interconnect and no single CPU failures. The code complexity should be as low as possible. Therefore we settled on a lightweight, *centralized* coordination service rather a replicated system.

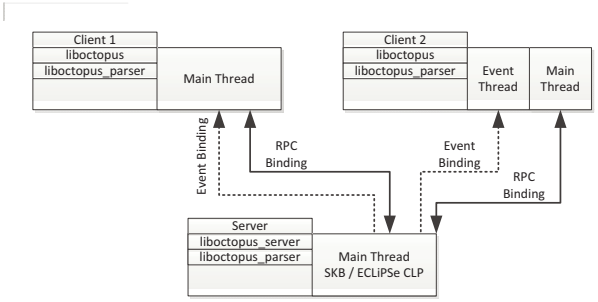
Second, the service must also be *self-contained*, to make coordination of OS boot-up possible. It should not rely on many other OS services (such as the file system or network), and should help model dependencies rather than creating new ones.

Third, information providers and consumers should be *loosely coupled*. This can be achieved by an asynchronous interface with a fast, flexible and scalable data and query model.

Fourth, the query interface should be *non-blocking*. Rather, events should asynchronously notify clients about data store changes.

Finally, a *high-level interface* should reduce the

Figure 1: Octopus: General Architecture



code complexity involved in synchronization.

While Octopus is strongly integrated with the rest of Barrelfish, we argue that the ideas it embodies are widely applicable to any OS trying to manage a complex multicore machine.

3.1 Octopus

Based on these principles, Octopus implements distributed, named synchronization primitives such as locks, barriers and semaphores above a key-value store and associated event delivery system. Octopus unifies synchronization, name service, and event handling for the OS. Using a convenient API, clients benefit from key-value store, data change events and publish-subscribe events. We distinguish between *transient* data in the publish-subscribe case which are not stored, from key-value store entries, which remain in RAM during lifetime of the OS or until they are deleted. The latter we call *persistent*, even if no disk is involved.

Octopus abstracts the key-value store behind high-level *record* entries, and a *query and update language* enables clients to add, query and modify records. Clients register for events at the record level. The two advantages of a high-level language are reduced code complexity and independence of the implementation. We describe records and the query language in more detail in section 3.2.

Octopus is built as an native extension to the SKB [13] as shown in Figure 1. Server functionality is in a library `liboctopus_server` linked with the SKB. Clients link to `liboctopus` which exports the Octopus API and communicates with the Octopus service. The `liboctopus_parser` library parses query and answer strings on both sides.

3.2 Records and Record Queries

Records are the basic data unit in Octopus. Clients add records to persistent storage and retrieve, modify or delete them. They can also register for *addition* and *deletion* events on patterns matching records of interest. Octopus also provides a publish-subscribe API for records which is similar but bypasses storage.

3.2.1 Records

Records consist of a name and an optional list of attribute-value pairs. The syntax is based on JSON (JavaScript Object Notation) [16], since it is easy to read and write for humans and machines. The following example shows a record called `hw.pci.device.1` representing a PCI network card:

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

Sequential records are a special form of records. Octopus appends a monotonically increasing number to the name defined by the client. It returns the new name to the client, allowing clients to create multiple unique ordered records, and serving as the basis for synchronization primitives.

3.2.2 Record Queries

Record queries use an extended version of the record entry syntax, allowing regular expressions for record names and attribute values and the special character '_' to match to any name or attribute value. Constraints on attribute values further specify whether records are part of the result or not. Record updates can depend on the currently stored value, as in SQL's `UPDATE` statement.

The following example matches records with any name but only those with `device <= 1`, `vendor > 100` and `class` matching the regular expression `C|X|T` belong to the result. An update sets `bus` to `5`, but only if the current value is `0`.

```
_ { bus: 5, bus == 0, device <= 1,
   vendor > 100, class: r'C|X|T'
}
```

3.3 Record Store

Whenever the Octopus service receives a *set* or *del* query from a client, it parses the query and performs the respective operation on persistent storage. Octopus stores the attribute-value pair list to the storage hash table with the record name as key. For *get* and *update* queries, it matches them to the stored records and returns the result to the client.

If the client is interested in future *add* or *del* events it creates a *trigger* along with the query and passes it to Octopus. The client specifies whether the trigger is persistent and should send an event whenever the query matches, or whether it should be automatically removed after the first event.

Octopus stores the trigger to the persistent storage hash table. Because record queries do not need to specify a fixed name, Octopus generates a trigger ID which serves as the key. Expected attribute-value pairs and constraints get stored with this ID.

The full record store API of the server is:

```
(names, err, t_id?) = get_names(q, trg?);
(record, err, t_id?) = get(q, trg?);
(err, record?, t_id?) = set(q, trg?);
(err, t_id?) = del(q, trg?);
(err, t_id?) = exists(q, trg?);
```

`get_names` returns an array of record names matching the query. `get` returns the first record to match the query. `set` inserts a new or updates an existing record. `del` deletes a record. `exists` is similar to `get`, but only returns an appropriate error code. All calls may install a trigger, in which case the server returns the trigger ID to remove it in the future. Creating triggers is done as follows:

```
(trg) = mktrigger(in_case?, send_async,
    mode, handler_fn, client_state);
(err) = rmtrigger(t_id);
```

`mktrigger` creates and configures a trigger according to flags passed by clients. It also installs the user handler function and user state. `rmtrigger` removes the trigger identified by its ID.

3.4 Publish-subscribe

Publish-subscribe in Octopus is similar, but records are not stored. Subscriptions use the same record query language and are stored like triggers. Further API calls allow clients to publish records and to subscribe to records.

3.5 Synchronization primitives

Octopus implements high-level synchronization primitives based on records. These are intended to coordinate distributed applications and are not suitable for fine-grained access control among threads. Our goal is to build synchronization primitives with few lines of code. Unifying records and change events provides a useful basis for such primitives: new clients can query existing state and existing clients receive change events. Our current functionality is based on that in Zookeeper [8]:

Locks: In an approach reminiscient of eventcounts and sequencers [12], acquiring a lock creates a sequential record using the lock name, agreed on by the clients. The client owning the record with the lowest number holds the lock. Other clients issue an *exist* call on the previous record to their own and pass a one-time trigger on its deletion. When the lock holder releases the lock (i.e. deletes the record), the next waiting client wakes up on the deletion event. As with eventcounts, no starvation occurs and the locks are fair in waiting time.

Barriers: Barriers ensure that different tasks start executing a section simultaneously. Using Octopus, we implemented a double barrier based on sequential records. Every client entering the barrier creates a sequential record and queries if the number of records is the expected number of clients entering the barrier. If so, it creates a special record indicating that all clients are ready. Otherwise, it creates a trigger waiting for this special record.

3.6 Implementation

Octopus is implemented as an extension to the ECL^{i}PS^e engine [1] which forms the basis for Barrelfish's SKB. It benefits from ECL^{i}PS^e's logical unification, backtracking, constraint evaluation and regular expression facilities, Queries are automatically matched to stored records. This reduces code complexity for applications and Octopus itself.

Octopus allows searching for records based on attribute values, and potentially all records need to be considered. An *attribute index* remembers all record names having a given attribute. Thus, Octopus quickly finds all potential matching records. The index is implemented as skip list [11], which

behaves similar to a binary tree. Finding triggers or subscriptions given a record is the opposite problem. A bitmap index indicates whether a trigger or subscription ID is relevant, given a record.

4 Early experience

We were motivated to build Octopus by three related coordination problems in the Barrelfish OS which were the cause of much complex, duplicated code and hard-to-find bugs, often timing-related.

4.1 Name service

Services in a distributed system register service references using a well-known name with a service registry [10, 14]. Clients resolve them by name, or more complex attribute-based queries. In a multikernel like Barrelfish, services also export references and applications are the clients connecting to them. Octopus allowed us to replace the previous service registry in Barrelfish with a considerably more expressive one using records of the form

```
servicename { iref: <nr>, ... }
```

where `servicename` is the well-known name and `<nr>` is the internally used reference expected by Barrelfish's connection function. Service dependencies are resolved by searching for a specific service name and waiting until it appears as a record.

4.2 Device management

We built a new Barrelfish service, dubbed Kaluga, which is the device manager responsible for starting the correct driver for each device appearing in the system. It uses driver mapping information in the SKB which bind devices to driver binaries, and registers a trigger for new device records. Device discovery by components like ACPI code, PCI drivers, and USB controllers enumerate hardware and add device records to Octopus. For each new record, Kaluga receives an event. It considers the device-driver mappings, stored in the SKB, and starts the appropriate driver.

The discovery and driver startup process is recursive. For example, the ACPI driver might find a PCIe root complex. Consequently, Kaluga starts a PCIe bus driver which enumerates devices under this root and generates new records. This triggers Kaluga to start PCIe drivers, such as a PCIe USB host controller. This driver in turn enumerates the USB bus and adds further records causing Kaluga to start USB device drivers, and so on.

4.3 System Bootstrap

Booting an OS is a complex task. Hardware has to be initialized and exported to clients, drivers and OS services have to be started. On modern hardware, other CPU cores also need to be started by the OS. Octopus has proved very useful so far in simplifying the bootstrap process in Barrelfish, and our current solution builds on both the name service and the Kaluga device manager.

In a typical scenario on x86 machines, Barrelfish startup works as follows: The BIOS starts the bootstrap core. The bootloader then starts the kernel. The ACPI flag on Intel x86 cores indicate ACPI availability. Thus, the boot code adds a device record for ACPI which causes Kaluga to start the ACPI driver. ACPI finds cores, I/O APICs and PCIe root bridges. It adds device records such that Kaluga can start the appropriate drivers.

Drivers and OS services start running and register with the name service. Depending services wait for required service references before they register with the name service. This way, the OS boot process is well coordinated. The uniform abstraction of dependencies behind Octopus records and triggers has allowed us to significantly reduce special-case code in many parts of the OS.

CPU cores beyond the first one are treated the same as regular devices. Whenever Kaluga receives a record event for a core (typically from ACPI), it starts the an appropriate kernel (or "CPU driver" in Barrelfish parlance) based on the driver database.

4.4 Performance

One goal of Octopus is building synchronization primitives with few lines of code. Table 1 shows a functionality breakdown with lines of code[1]. As the table shows, barriers, locks, semaphores and Kaluga need few lines of code. Also Octopus is implemented in relatively few lines of code, because it benefits from the high-level ECL^{i}PS^{e} language.

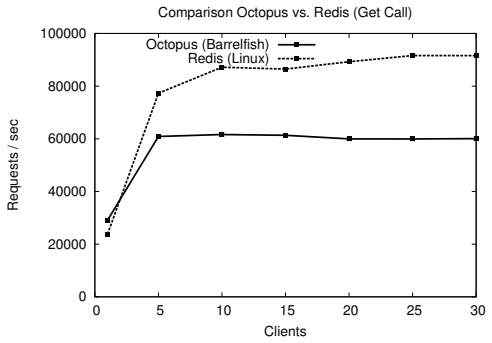
We measured the performance of Octopus on a PC with two dual-core AMD Santa Rosa CPUs run-

[1] Generated using David A. Wheeler's SLOCCount

Table 1: Lines of code

Functionality	C	Prolog	Lex	YACC
Octopus	3188	355	150	94
Barriers	102			
Locks	87			
Semaphores	106			
Kaluga	759			

Figure 2: Throughput Octopus vs. Redis



ning at 2.8 GHz. Server and client ran on different cores on the same package. We used big hash tables to reduce garbage collection and removed outliers due to context switches and other effects.

Octopus is similar in implementation to Redis [17], though simpler and less optimized. We compared the throughput of `get` calls with 256 byte payload on Redis 2.4.7 running on Linux 2.6.32 pinned to one core using the provided `redis-benchmark` program, with Octopus running on Barrelfish. Figure 2 shows the peak for Redis is at about 90000 ops/sec and for Octopus around 60000. Scalability is similar. The performance hit in Octopus is due to ECL^{i}PS^e. To compute the overhead, we run a microbenchmark issuing `get` calls to retrieve a specific record out of 1.4 million stored records. The overhead of ECL^{i}PS^e compared to the overall latency was roughly 80%.

5 Conclusion

To date, Octopus has been a net benefit both in terms of code complexity (the system is simpler with it than without it) and functionality. However, while it has solved some pressing problems for us, we are only begining to explore its implications for the rest of the OS, such as the file system and power management, to name but two areas of interest. The key insight we borrow from large-scale clusters systems is that it is beneficial to separate coordination from the rest of the system code.

References

[1] K. R. Apt and M. G. Wallace. *Constraint Logic Programming using ECL^{i}PS^e*. Cambridge University Press, 2007.

[2] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. SOSP'09, 2009.

[3] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. OSDI'06, 2006.

[4] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. SOSP '07, 2007.

[5] P. T. Eugster and R. Guerraoui. Content-based publish/-subscribe with structural reflection. COOTS'01, 2001.

[6] Freedesktop.org. D-bus. `http://dbus.freedesktop.org/`, March 2012.

[7] R. Geambasu, A. A. Levy, T. Kohno, A. Krishnamurthy, and H. M. Levy. Comet: an active distributed key-value store. OSDI'10, 2010.

[8] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: wait-free coordination for internet-scale systems. USENIX ATC'10, 2010.

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

[10] Object Management Group, Inc. *CORBA 3.1 Specification*, 2008.

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

[12] D. P. Reed and R. K. Kanodia. Synchronization with eventcounts and sequencers. *Commun. ACM*, 22(2):115–123, Feb. 1979.

[13] A. Schüpbach, S. Peter, A. Baumann, T. Roscoe, P. Barham, T. Harris, and R. Isaacs. Embracing diversity in the Barrelfish manycore operating system. MMCS'08, 2008.

[14] R. Thurlow. RPC: Remote procedure call protocol specification version 2. RFC 5531, Sun Microsystems, 2009.

[15] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *SIGOPS Operating Systems Review*, 43(2):76–85, 2009.

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

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