

Unifying Synchronization and Events in a Multicore OS

Gerd Zellweger **Adrian Schüpbach** Timothy Roscoe
Systems Group, ETH Zurich



This talk

- ▶ How to coordinate a distributed systems like OS?

This talk

- ▶ How to coordinate a distributed systems like OS?
- ▶ Multicore hardware leads to new OS designs
 - ▶ “Multikernel” OSs like fos and Barrelfish
 - ▶ Structure OS as a distributed system
 - ▶ No shared state between nodes

This talk

- ▶ How to coordinate a distributed systems like OS?
- ▶ Multicore hardware leads to new OS designs
 - ▶ “Multikernel” OSs like fos and Barrelfish
 - ▶ Structure OS as a distributed system
 - ▶ No shared state between nodes
- ▶ Nice for scalability and handling of heterogeneity
- ▶ However facing similar problems as in classical distributed systems within OS

This talk

- ▶ How to coordinate a distributed systems like OS?
- ▶ Multicore hardware leads to new OS designs
 - ▶ “Multikernel” OSs like fos and Barrelfish
 - ▶ Structure OS as a distributed system
 - ▶ No shared state between nodes
- ▶ Nice for scalability and handling of heterogeneity
- ▶ However facing similar problems as in classical distributed systems within OS
- ▶ Services and applications on nodes execute independently
 - ▶ Still perform a common task
 - ▶ Need to be coordinated

Motivation: Waiting for dependencies

- ▶ Waiting for dependencies
- ▶ “Solution”
 - ▶ Store known fact “instance_done”
 - ▶ Depending process queries for this fact
- ▶ Typical pseudo code in Barrelfish:

```
while (not("instance_done")) {  
    query("instance_done");  
    thread_yield();  
}
```

- ▶ Used to wait for termination of dependent processes
- ▶ Coordinate system bootup
- ▶ Used to resolve driver dependencies
 - ▶ NIC driver must wait for PCIe driver to initialize PCIe bus

What do we need?

- ▶ Need ability to store information
- ▶ Need notification about new information, rather than while loop

What do other people do?

- ▶ Apache Zookeeper
- ▶ Google Chubby
- ▶ Redis

Within same machine

- ▶ Different environment
 - ▶ Reliable interconnect
 - ▶ No single CPU failures within machine
- ▶ Decided to go for simple centralized service

Our approach in Barrelfish: Octopus

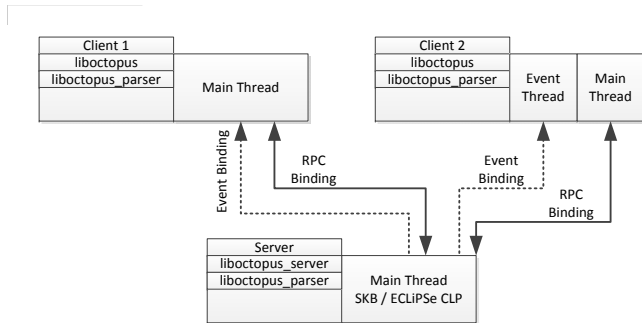
- ▶ Searchable key–value store with event triggers
- ▶ Publish–subscribe system
- ▶ Synchronization primitives



Requirements

1. OS bootup coordination
→ self-contained
2. Information providers and consumers often independent
→ loosely coupled asynchronous interface
3. Query interface should not block client
→ non-blocking event mechanism
4. Reduce code complexity for programmer
→ high-level interface

Octopus architecture



- ▶ Client–server architecture
- ▶ Event binding for asynchronous events
- ▶ RPC binding for bidirectional, synchronous communication

Records and record queries

▶ Records

- ▶ Unit of storage for the key–value store
- ▶ Unit of transfer for the publish–subscribe system

```
memserver { iref: 20 }
```

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

▶ Record queries

- ▶ Adds compare operators, regular expressions and variables
- ▶ Used for key–value store and publish–subscribe

```
hw.pci.bridge { bus: 0, acpi_node >= 0, function: _ }
```

```
r'hw\.pci\.device\.[0-9]+' { bus == 0, class: r'C|X|A' }
```

Sequential records and triggers

- ▶ Sequential record
 - ▶ Record with automatically appended increasing number
 - ▶ Record name passed by application

Sequential records and triggers

- ▶ Sequential record
 - ▶ Record with automatically appended increasing number
 - ▶ Record name passed by application
- ▶ Triggers
 - ▶ Triggers notify about future changes of the key–value store
 - ▶ Installable for *add* and *delete record* or both
 - ▶ One-time triggers and continuous triggers

Implementation

- ▶ Octopus is based on ECLⁱPS^e CLP
 - ▶ Prolog + constraint extensions
- ▶ Records and queries translated to ECLⁱPS^e CLP terms
 - ▶ Flex and Bison
- ▶ ECLⁱPS^e's backtracking takes care of searching record entries
- ▶ Result sent back to client

Use cases

- ▶ Distributed synchronization primitives
 - ▶ Locks
 - ▶ Barriers
 - ▶ Semaphores
- ▶ Kaluga device manager based on Octopus triggers

Use case: locks

- ▶ Clients agree on a lock name

Use case: locks

- ▶ Clients agree on a lock name
- ▶ Acquire lock
 - ▶ Every client adds a sequential record
 - ▶ If the client has the lowest sequence number, it holds the lock
 - ▶ Otherwise it installs a delete trigger on the record with the previous sequence number

Use case: locks

- ▶ Clients agree on a lock name
- ▶ Acquire lock
 - ▶ Every client adds a sequential record
 - ▶ If the client has the lowest sequence number, it holds the lock
 - ▶ Otherwise it installs a delete trigger on the record with the previous sequence number
- ▶ Release lock
 - ▶ Delete own sequential record
 - ▶ This triggers the next waiting client

Use case: Kaluga device manager

- ▶ Goal: Start specific driver for a newly found device
- ▶ Discovery code and device driver do not know each other

Use case: Kaluga device manager

- ▶ Goal: Start specific driver for a newly found device
- ▶ Discovery code and device driver do not know each other
- ▶ Approach
 - ▶ Kaluga installs trigger for all new hardware records
 - ▶ Discovery code adds records for each device found
 - ▶ Kaluga gets notified for each new device
 - ▶ Considers HW ID → driver binary mapping
 - ▶ Starts driver

Use case: Kaluga device manager

- ▶ Goal: Start specific driver for a newly found device
- ▶ Discovery code and device driver do not know each other
- ▶ Approach
 - ▶ Kaluga installs trigger for all new hardware records
 - ▶ Discovery code adds records for each device found
 - ▶ Kaluga gets notified for each new device
 - ▶ Considers HW ID → driver binary mapping
 - ▶ Starts driver
- ▶ Example
 - ▶ PCIe bus driver finds USB host controller
 - ▶ Kaluga starts USB host controller driver
 - ▶ USB host controller driver finds specific USB devices
 - ▶ Kaluga gets notified and starts USB drivers

Evaluation

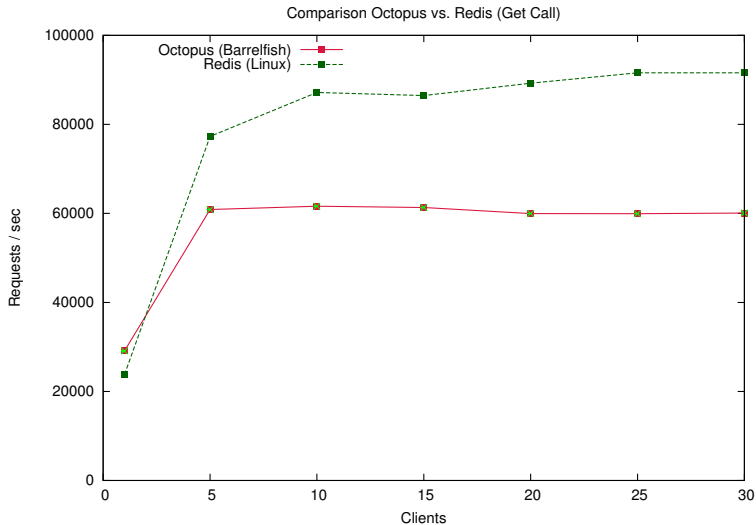
- ▶ We care about
 - ▶ Octopus should be a clean coordination framework
 - ▶ Small code complexity for programmers
 - ▶ Reasonable performance

Evaluation

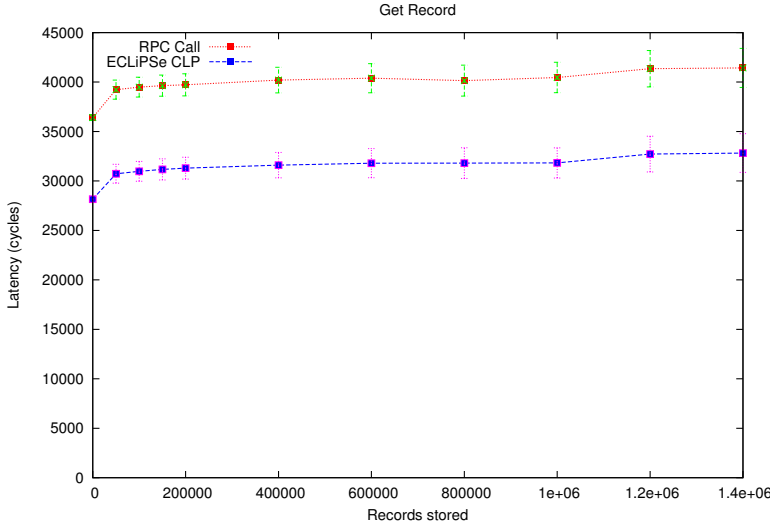
Table: Lines of code

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

Evaluation

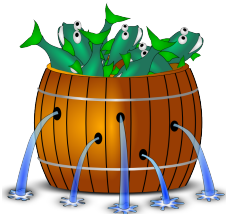


Evaluation



Conclusion

- ▶ OSs with distributed structure face distributed systems problems
- ▶ Different environment than cluster
- ▶ Octopus provides easy-to-use lightweight coordination framework
- ▶ Early experiences
 - ▶ Distributed synchronization primitives
 - ▶ Kaluga device manager



Download:
<http://www.barrelfish.org>