Operating Systems Meet Fault Tolerance

Microkernel-Based Operating Systems

Maksym Planeta Björn Döbel

24.01.2017

"If there's more than one possible outcome of a job or task, and one of those outcome will result in disaster or an undesirable consequence, then somebody will do it that way." (Edward Murphy jr.)

Outline

- Murphy and the OS: Is it really that bad?
- Fault-Tolerant Operating Systems
  - Minix3
  - CuriOS
  - L4ReAnimator
- Dealing with Hardware Errors
  - Transparent replication as an OS service

Why Things go Wrong

- Programming in C:

  This pointer is certainly never going to be NULL!

Why Things go Wrong

- Programming in C:

  This pointer is certainly never going to be NULL!

- Layering vs. responsibility:

  Of course, someone in the higher layers will already have checked this return value.

Why Things go Wrong

- Programming in C:

  This pointer is certainly never going to be NULL!

- Layering vs. responsibility:

  Of course, someone in the higher layers will already have checked this return value.

- Concurrency:

  This struct is shared between an IRQ handler and a kernel thread. But they will never execute in parallel.

Why Things go Wrong

- Programming in C:

  This pointer is certainly never going to be NULL!

- Layering vs. responsibility:

  Of course, someone in the higher layers will already have checked this return value.

- Concurrency:

  This struct is shared between an IRQ handler and a kernel thread. But they will never execute in parallel.

- Hardware interaction:

  But the device spec said, this was not allowed to happen!

Why Things go Wrong

- Programming in C:

  This pointer is certainly never going to be NULL!

- Layering vs. responsibility:

  Of course, someone in the higher layers will already have checked this return value.

- Concurrency:

  This struct is shared between an IRQ handler and a kernel thread. But they will never execute in parallel.

- Hardware interaction:

  But the device spec said, this was not allowed to happen!

- Hypocrisy:

  I'm a cool OS hacker. I won't make mistakes, so I don't need to test my code!

A Classic Study

- A. Chou et al.: *An empirical study of operating system errors*, SOSP 2001
- Automated software error detection (today: `https://www.coverity.com`)
- Target: Linux (1.0 - 2.4)
  - Where are the errors?
  - How are they distributed?
  - How long do they survive?
  - Do bugs cluster in certain locations?

Revalidation of Chou's Results

- N. Palix et al.: *Faults in Linux: Ten years later*, ASPLOS 2011
- 10 years of work on tools to decrease error counts - has it worked?
- Repeated Chou's analysis until Linux 2.6.34

Linux: Lines of Code

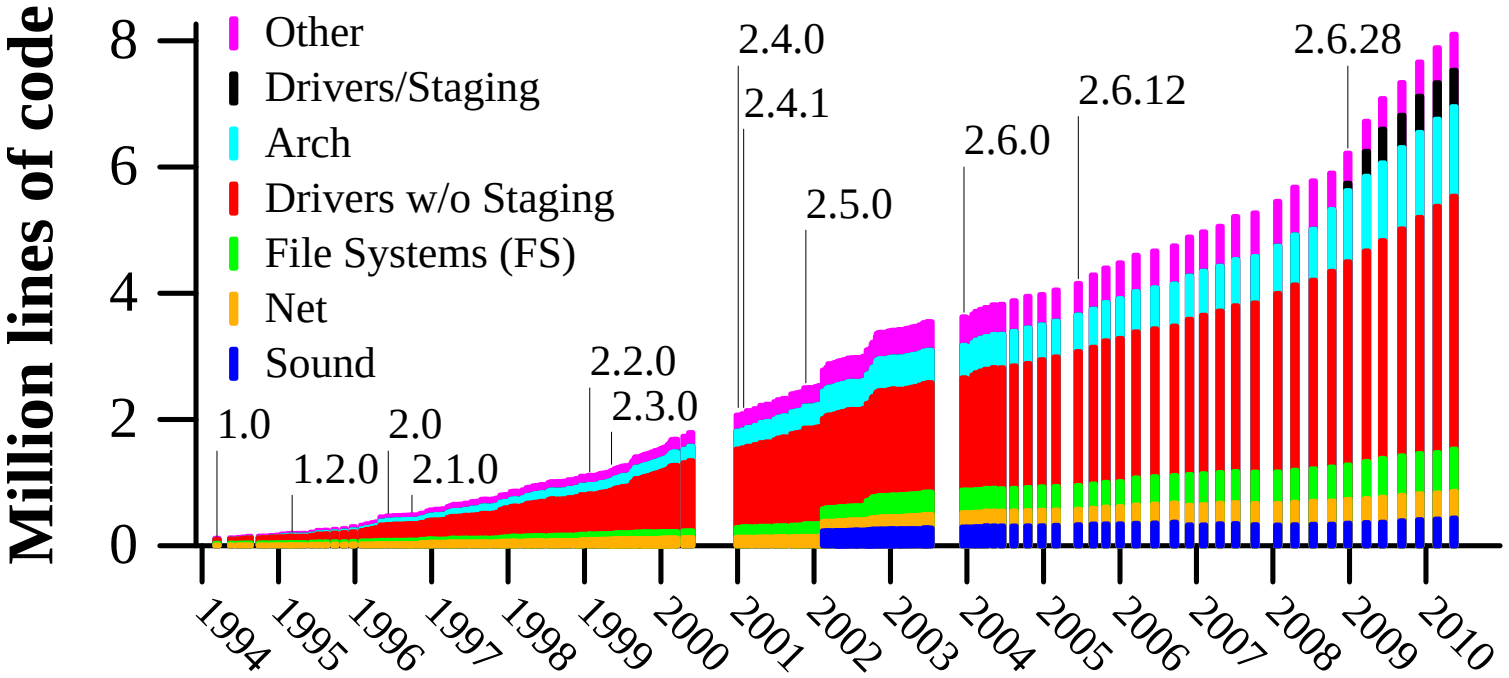


Figure: Linux directory sizes (in MLOC) [13]

Faults per Subdirectory (2001)

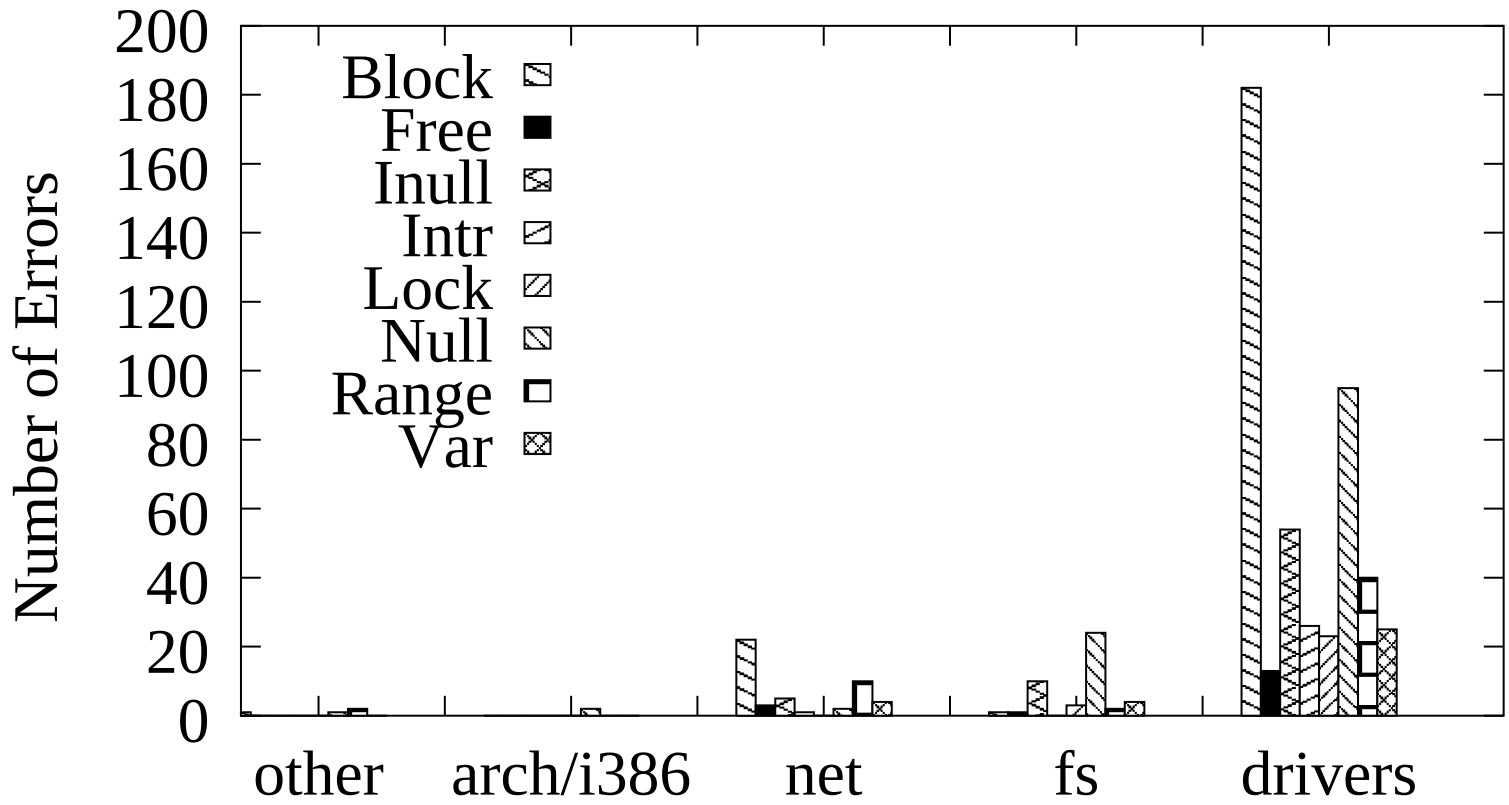


Figure: Number of errors per directory in Linux [3]

Fault Rate per Subdirectory (2001)

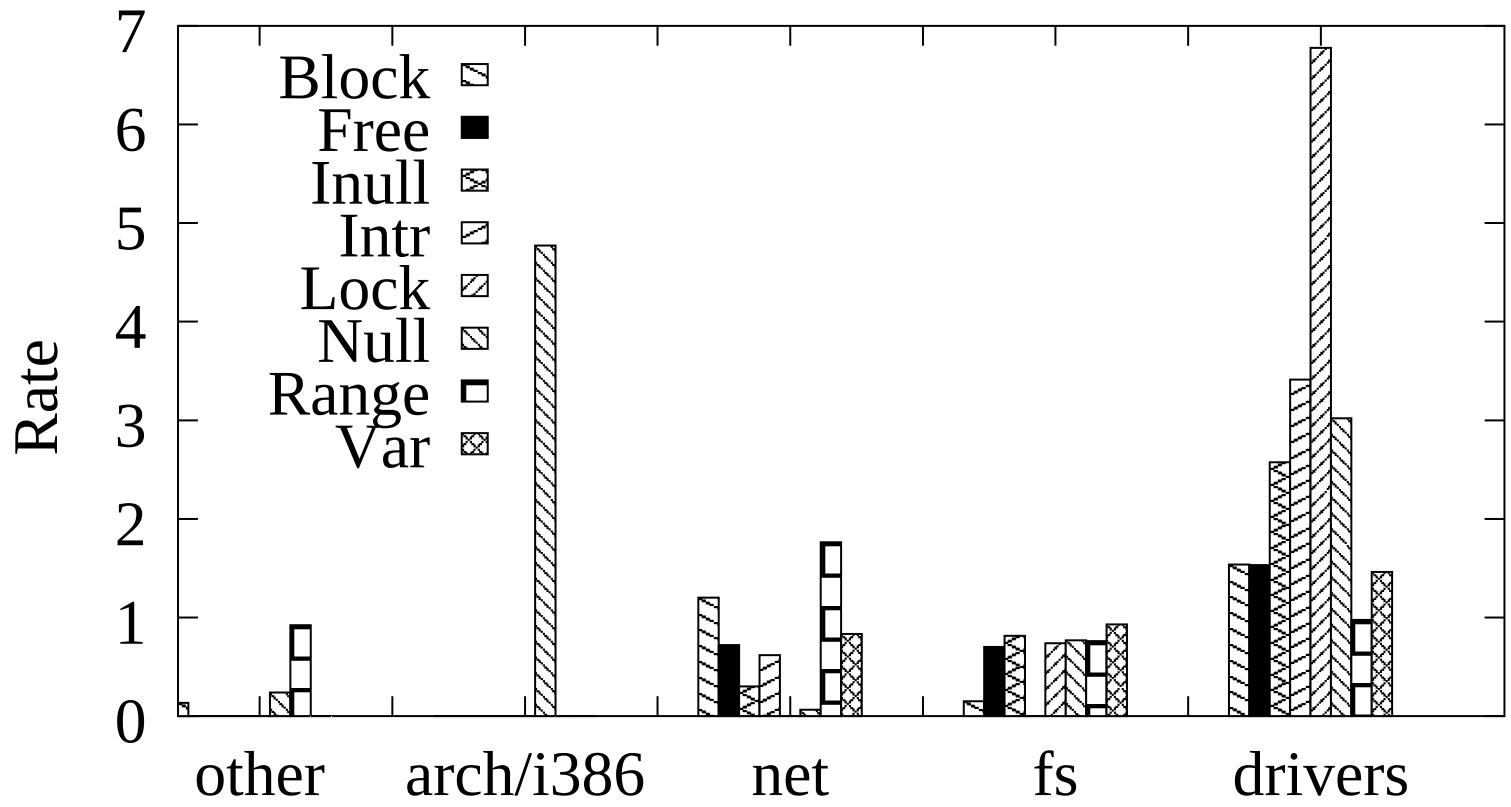


Figure: Rate of errors compared to other directories [3]

Fault Rate per Subdirectory (2011)

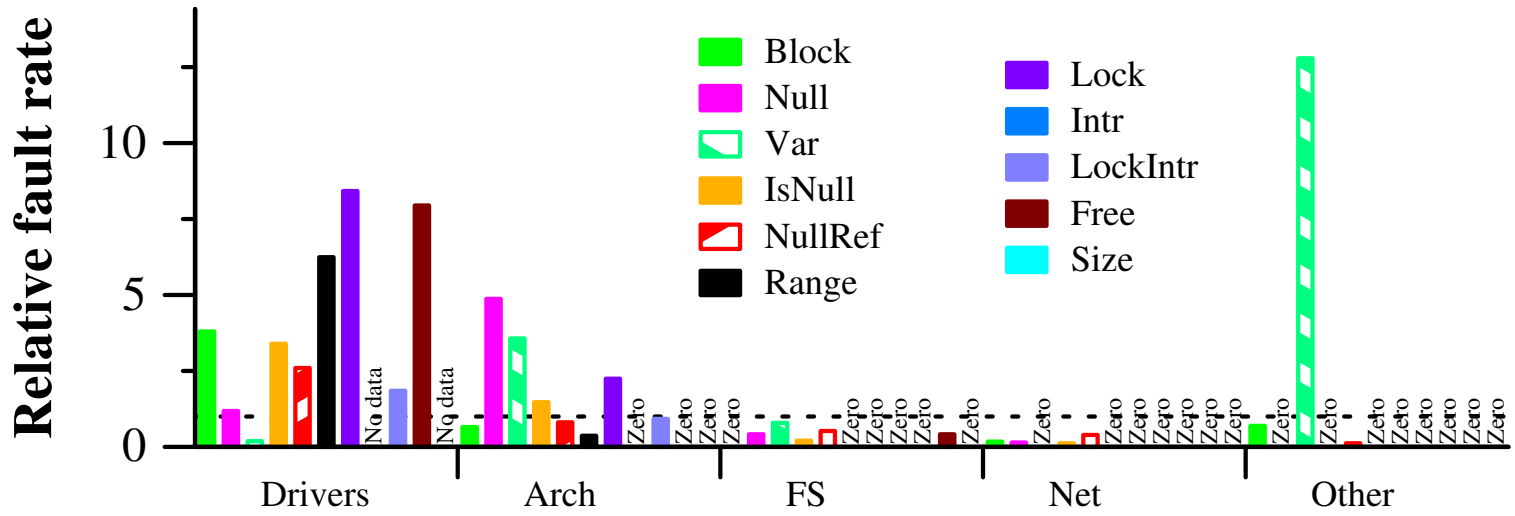


Figure: Linux directory sizes (in MLOC) [13]

Number Bug Evolution (2011)

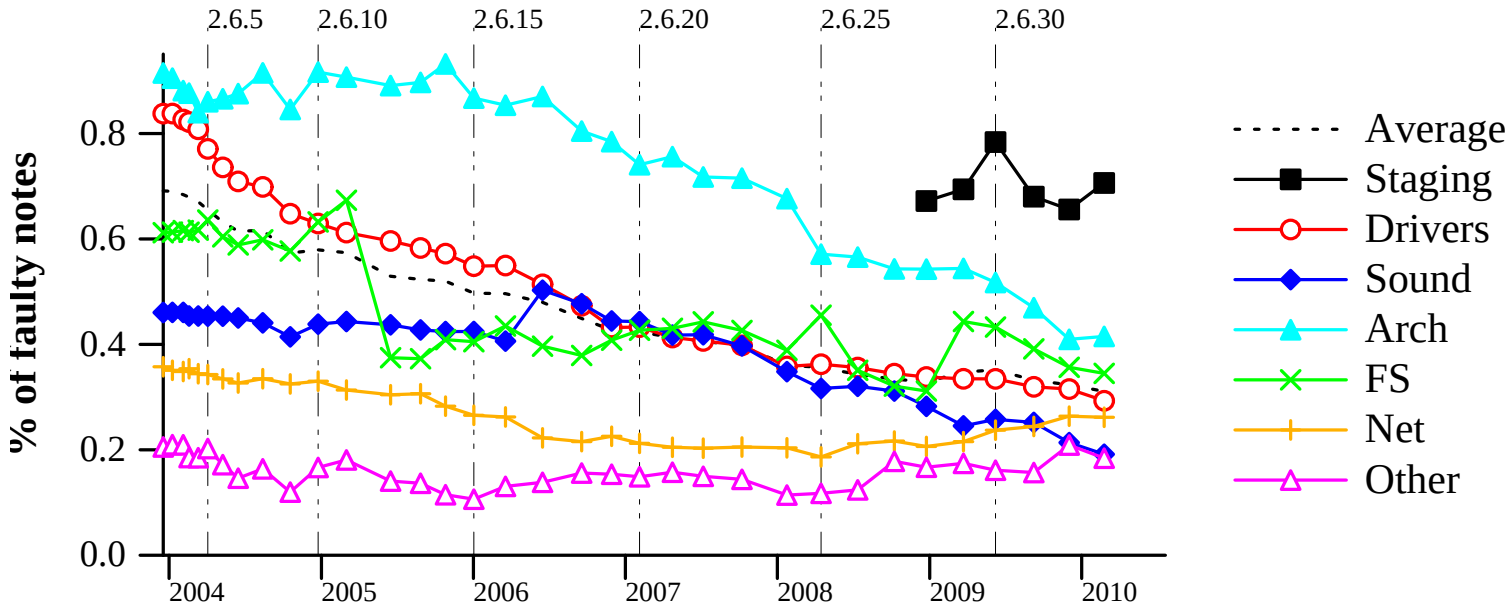
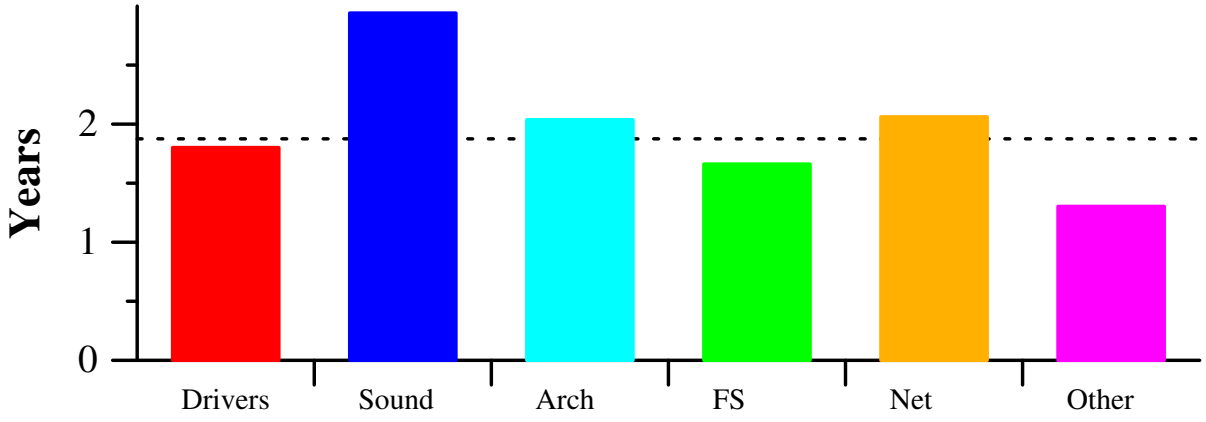
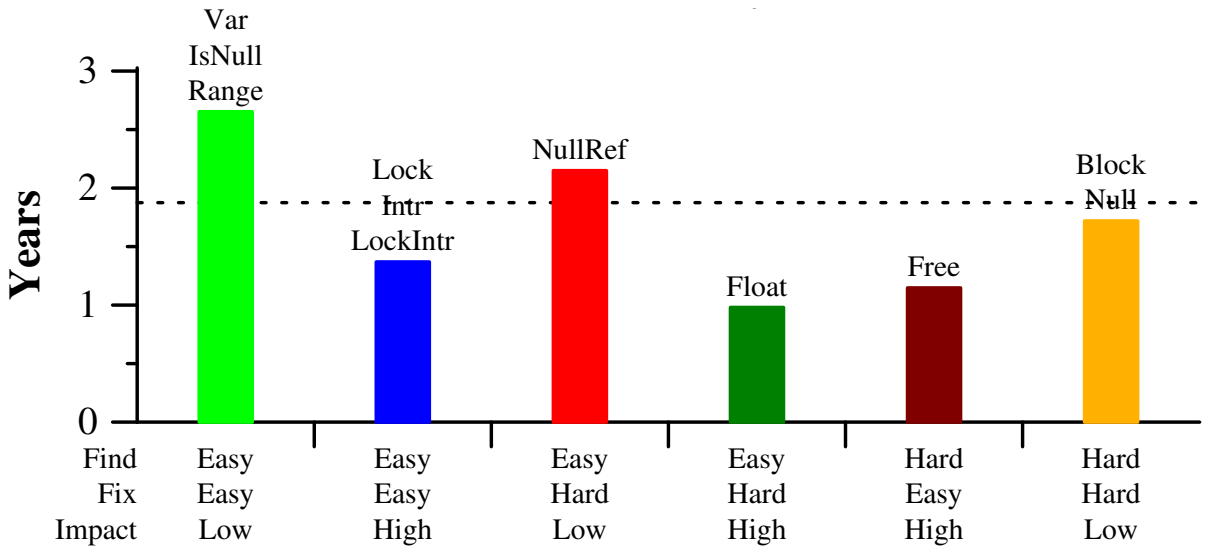


Figure: Linux directory sizes (in MLOC) [13]

Bug Lifetimes (2011)



(a) Per directory



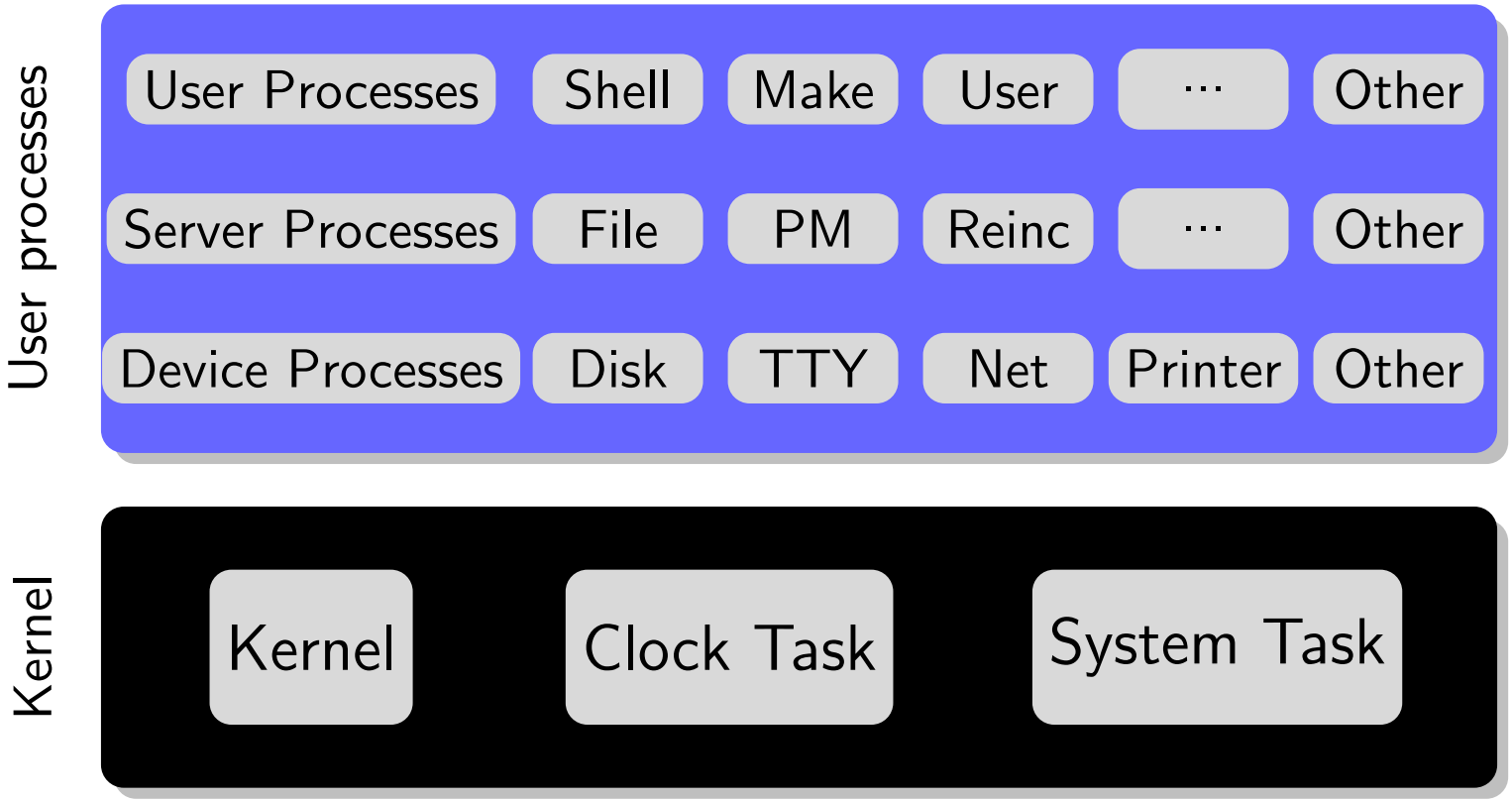
(b) Per finding and fixing difficulty, and impact likelihood

Figure: Average fault lifespans [13]

Break

- Faults are an issue.
- Hardware-related stuff is worst.
- Now what can the OS do about it?

Minix3 – A Fault-tolerant OS



Minix3: Fault Tolerance[1]

- Address Space Isolation
  - Applications only access private memory
  - Faults do not spread to other components
- User-level OS services
  - Principle of Least Privilege
  - Fine-grain control over resource access
    - e.g., DMA only for specific drivers
- Small components
  - Easy to replace (micro-reboot)

[1] Jorrit N Herder et al. "Fault isolation for device drivers". In: *DSN*. 2009, pp. 33–42.

Minix3: Fault Detection

- Fault model: transient errors caused by software bugs
- Fix: Component restart
- *Reincarnation server* monitors components
  - Program termination (crash)
  - CPU exception (div by 0)
  - Heartbeat messages
- Users may also indicate that something is wrong

Repair

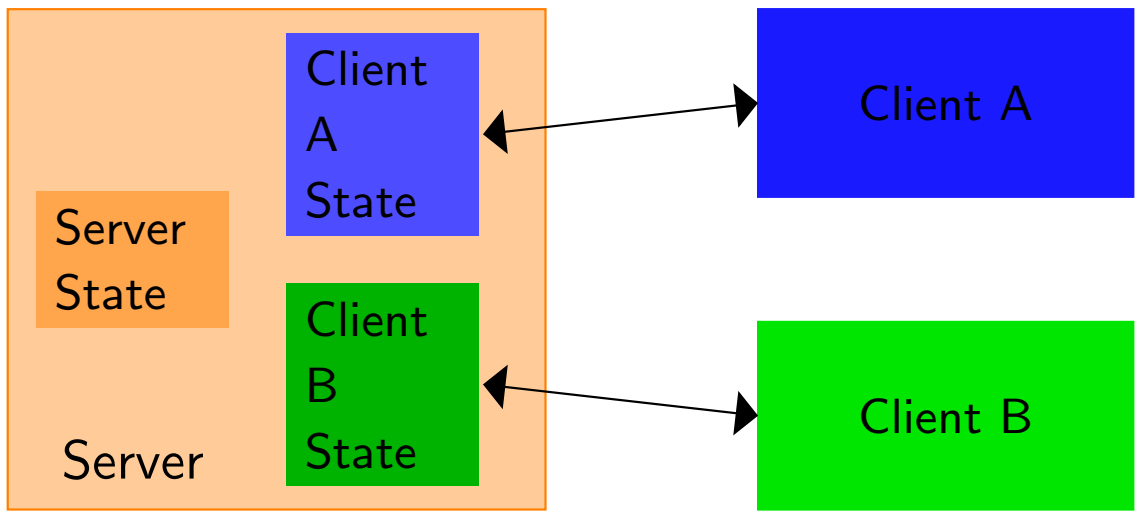
- Restarting a component is insufficient:
  - Applications may *depend* on restarted component
  - After restart, *component state* is lost
- Minix3: explicit mechanisms
  - Reincarnation server signals applications about restart
  - Applications store state at data store server
  - In any case: program interaction needed
    - Restarted app: store/recover state
    - User apps: recover server connection

Break

- Minix3 fault tolerance:
  - Architectural Isolation
  - Explicit monitoring and notifications
- Other approaches:
  - CuriOS: smart session state handling
  - L4ReAnimator: semi-transparent restart in a capability-based system

CuriOS: Servers and Sessions[2]

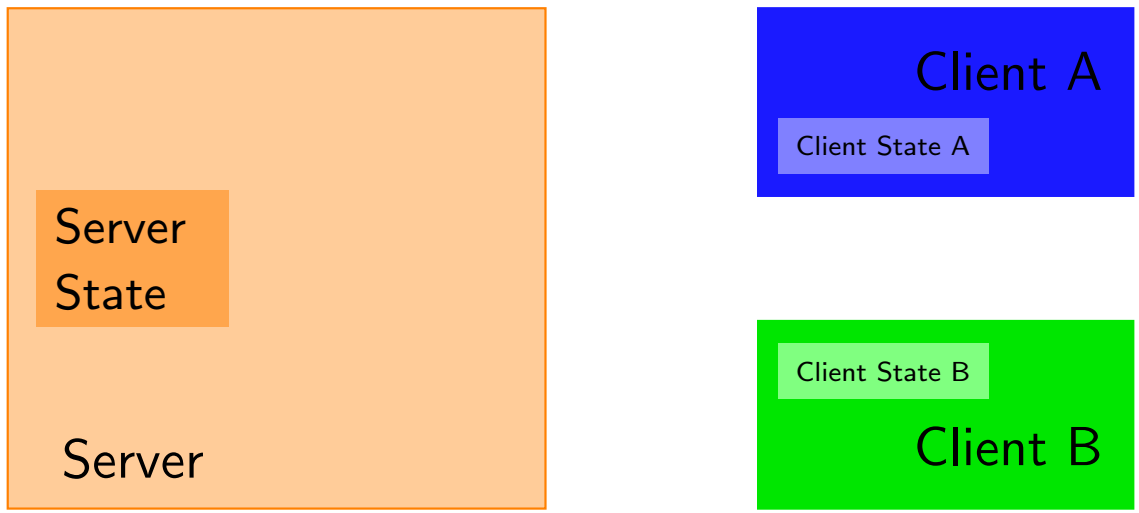
- State recovery is tricky
  - Minix3: Data Store for application data
  - But: applications interact
    - Servers store *session-specific* state
    - Server restart requires potential rollback for every participant



[2] Francis M David et al. "CuriOS: Improving Reliability through Operating System Structure." In: *OSDI*. 2008, pp. 59–72.

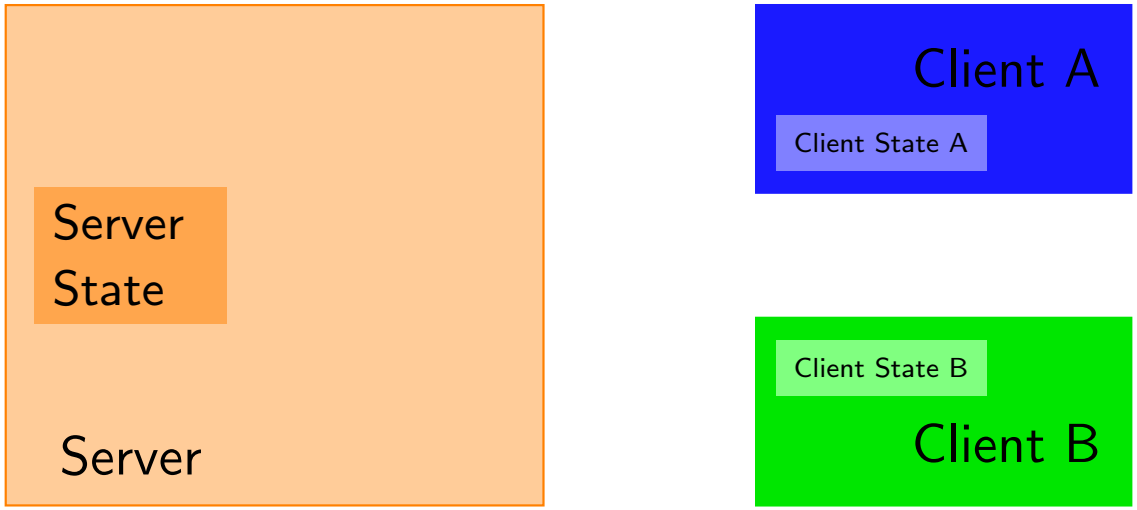
CuriOS: Server State Regions

- CuriOS kernel (CuiK) manages dedicated session memory: *Server State Regions*
- SSRs are managed by the kernel and attached to a client-server connection



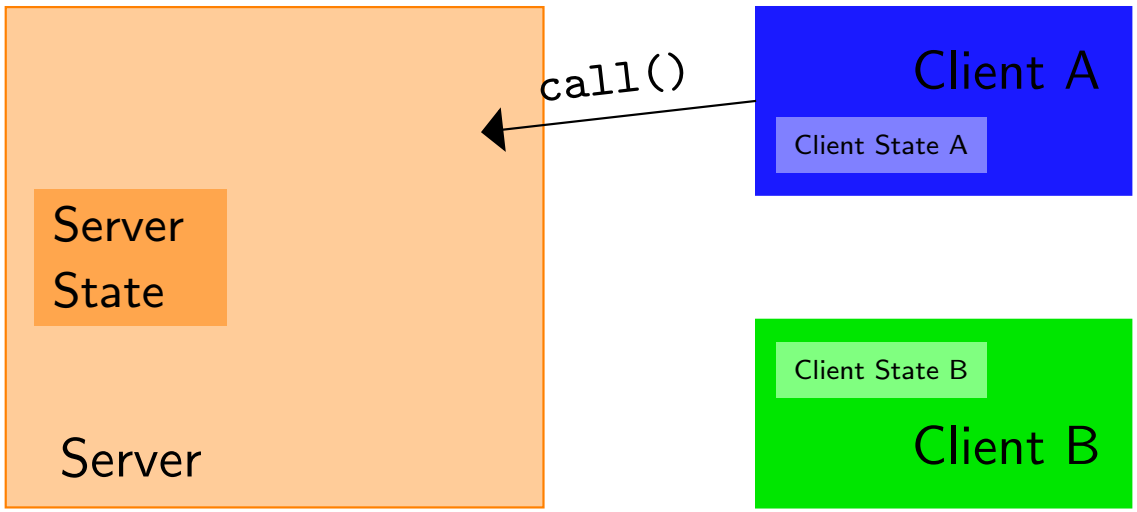
CuriOS: Protecting Sessions

- SSR gets mapped only when a client actually invokes the server
- Solves another problem: failure while handling A's request will never corrupt B's session state



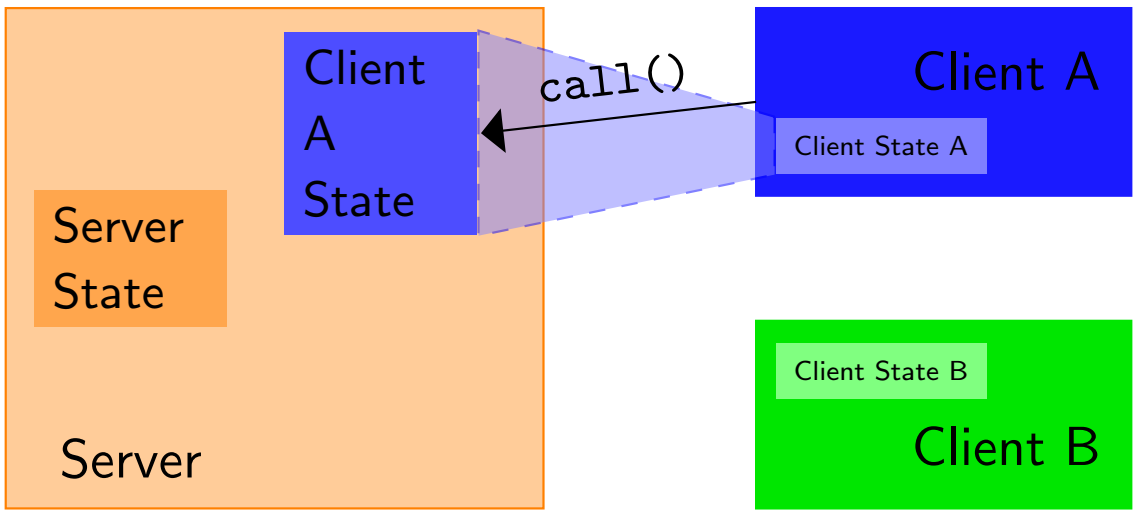
CuriOS: Protecting Sessions

- SSR gets mapped only when a client actually invokes the server
- Solves another problem: failure while handling A's request will never corrupt B's session state



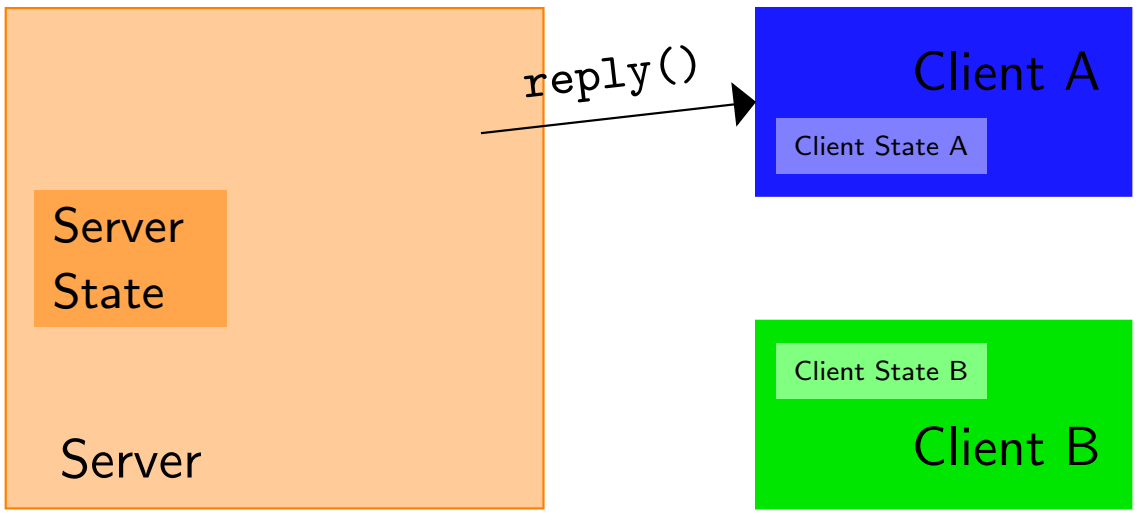
CuriOS: Protecting Sessions

- SSR gets mapped only when a client actually invokes the server
- Solves another problem: failure while handling A's request will never corrupt B's session state



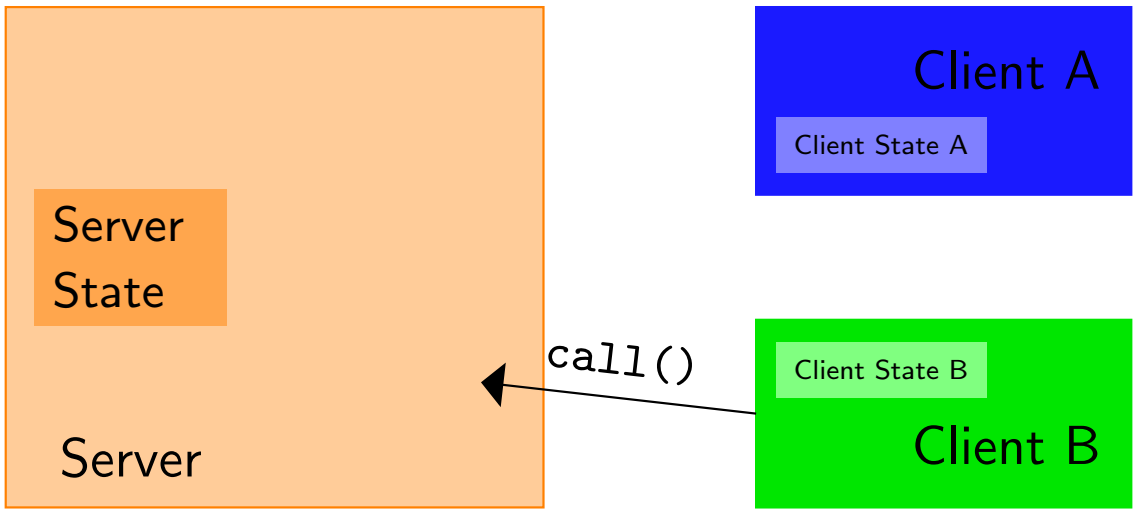
CuriOS: Protecting Sessions

- SSR gets mapped only when a client actually invokes the server
- Solves another problem: failure while handling A's request will never corrupt B's session state



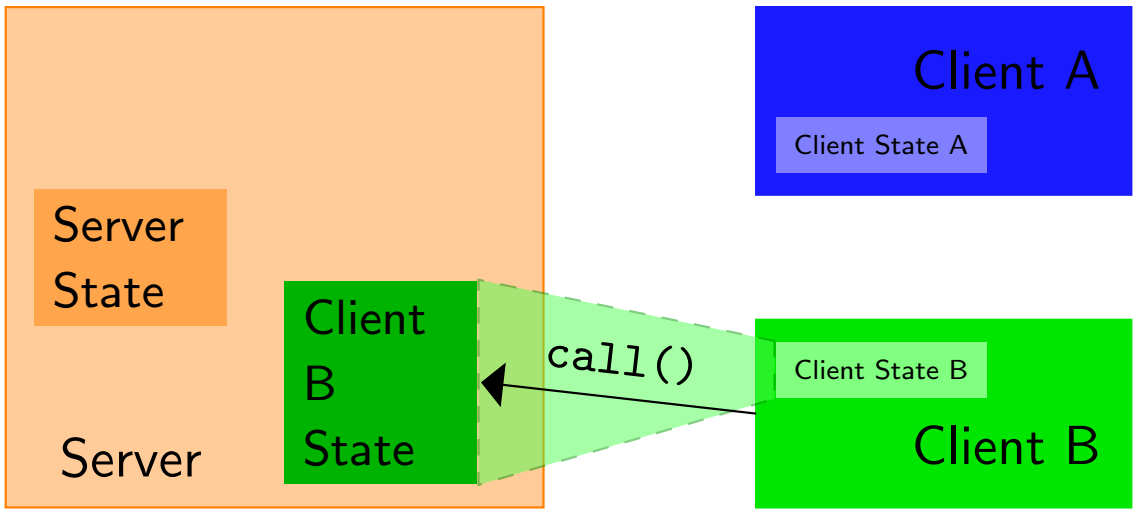
CuriOS: Protecting Sessions

- SSR gets mapped only when a client actually invokes the server
- Solves another problem: failure while handling A's request will never corrupt B's session state



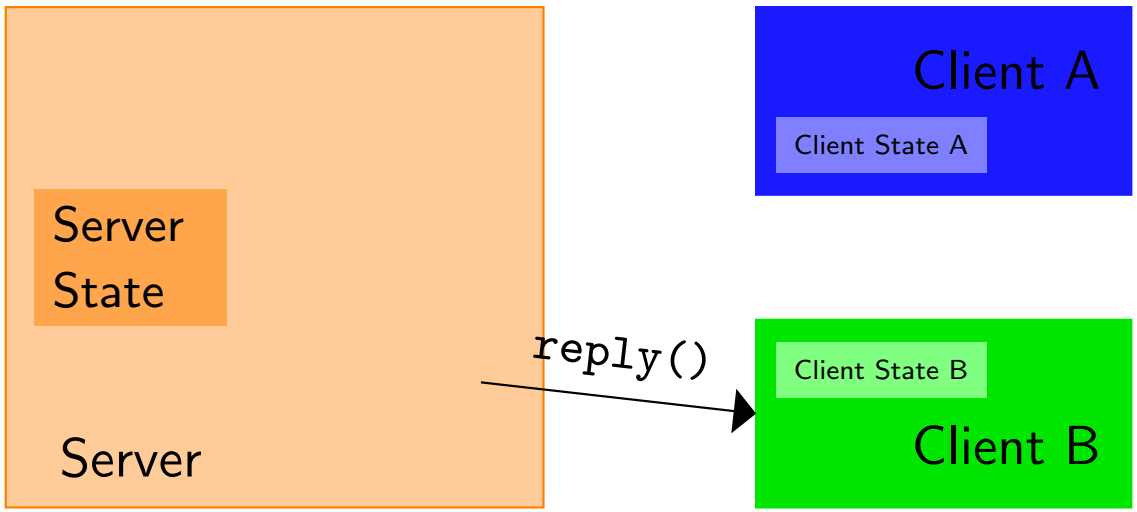
CuriOS: Protecting Sessions

- SSR gets mapped only when a client actually invokes the server
- Solves another problem: failure while handling A's request will never corrupt B's session state



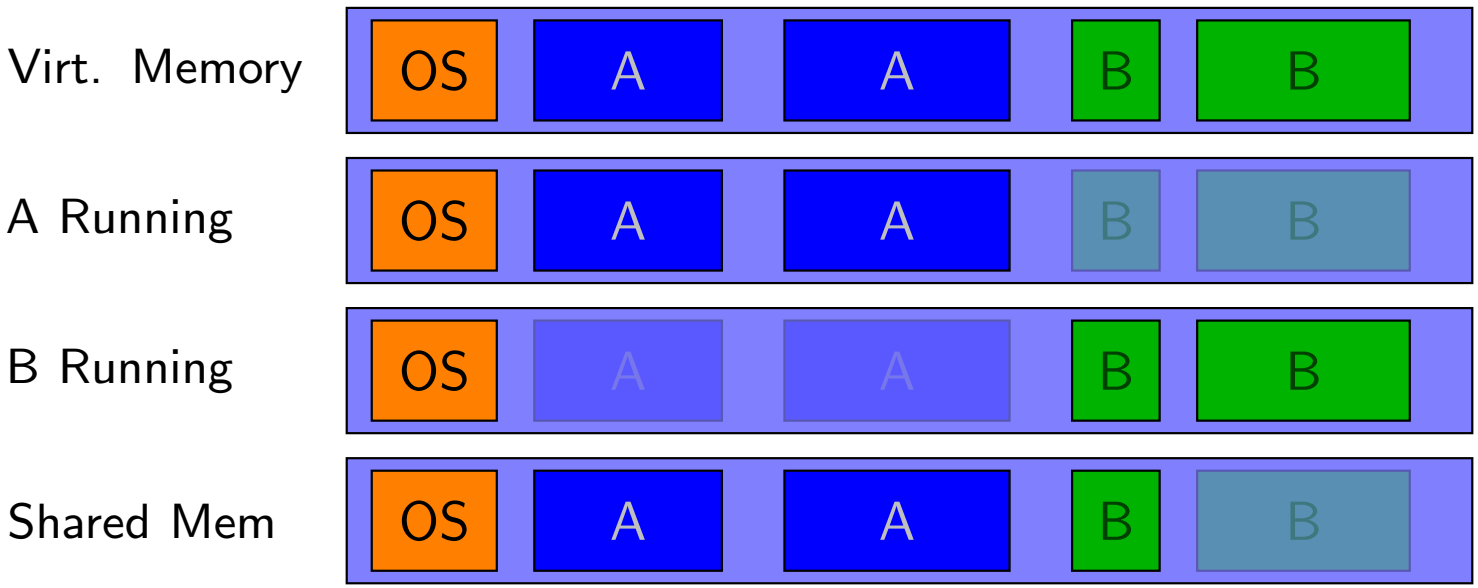
CuriOS: Protecting Sessions

- SSR gets mapped only when a client actually invokes the server
- Solves another problem: failure while handling A's request will never corrupt B's session state



CuriOS: Transparent Restart

- CuriOS is a *Single-Address-Space OS*:
  - Every application runs on the same page table (with modified access rights)



Transparent Restart

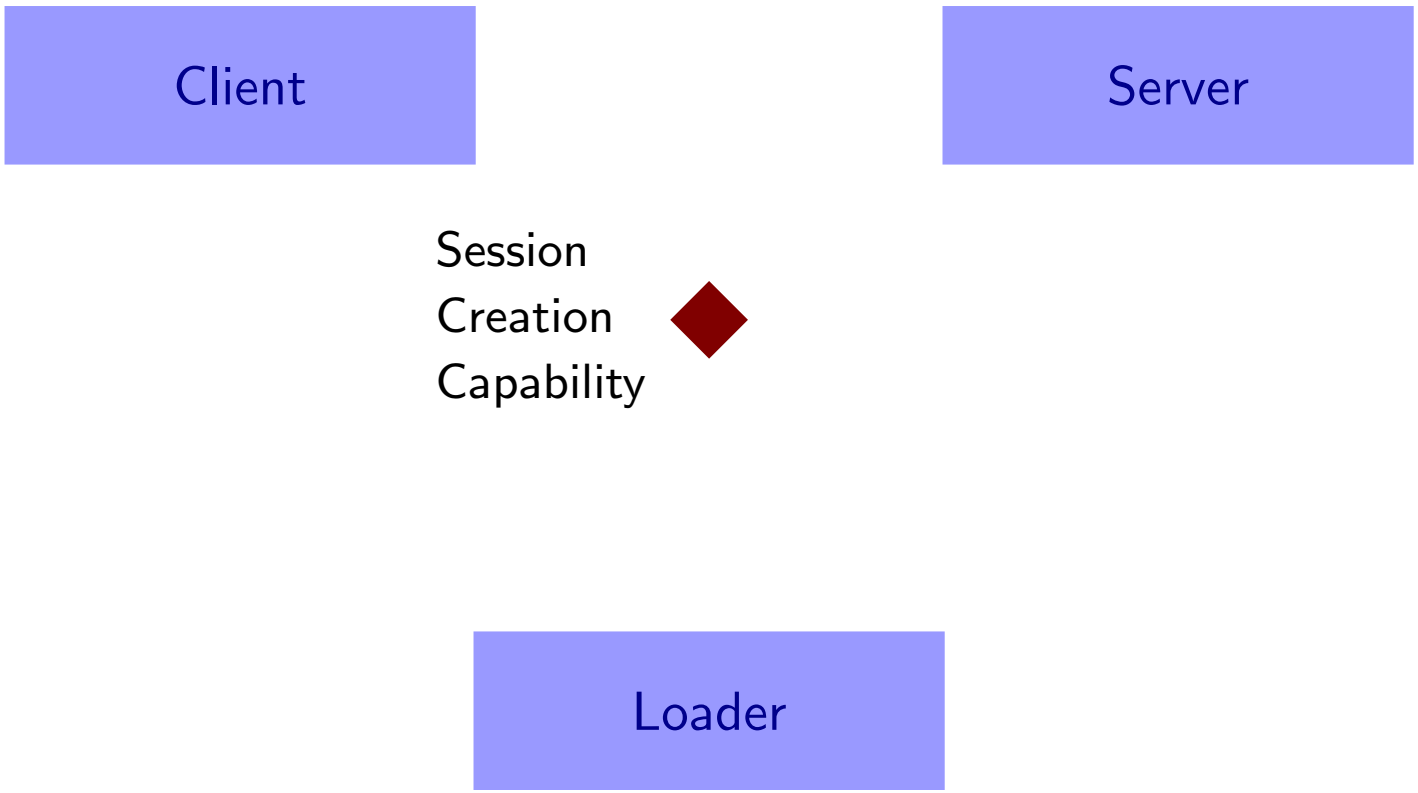
- Single Address Space
  - Each object has unique address
  - Identical in all programs
  - Server := C++ object
- Restart
  - Replace old C++ object with new one
  - Reuse previous memory location
  - References in other applications remain valid
  - OS blocks access during restart

L4ReAnimator: Restart on L4Re[3]

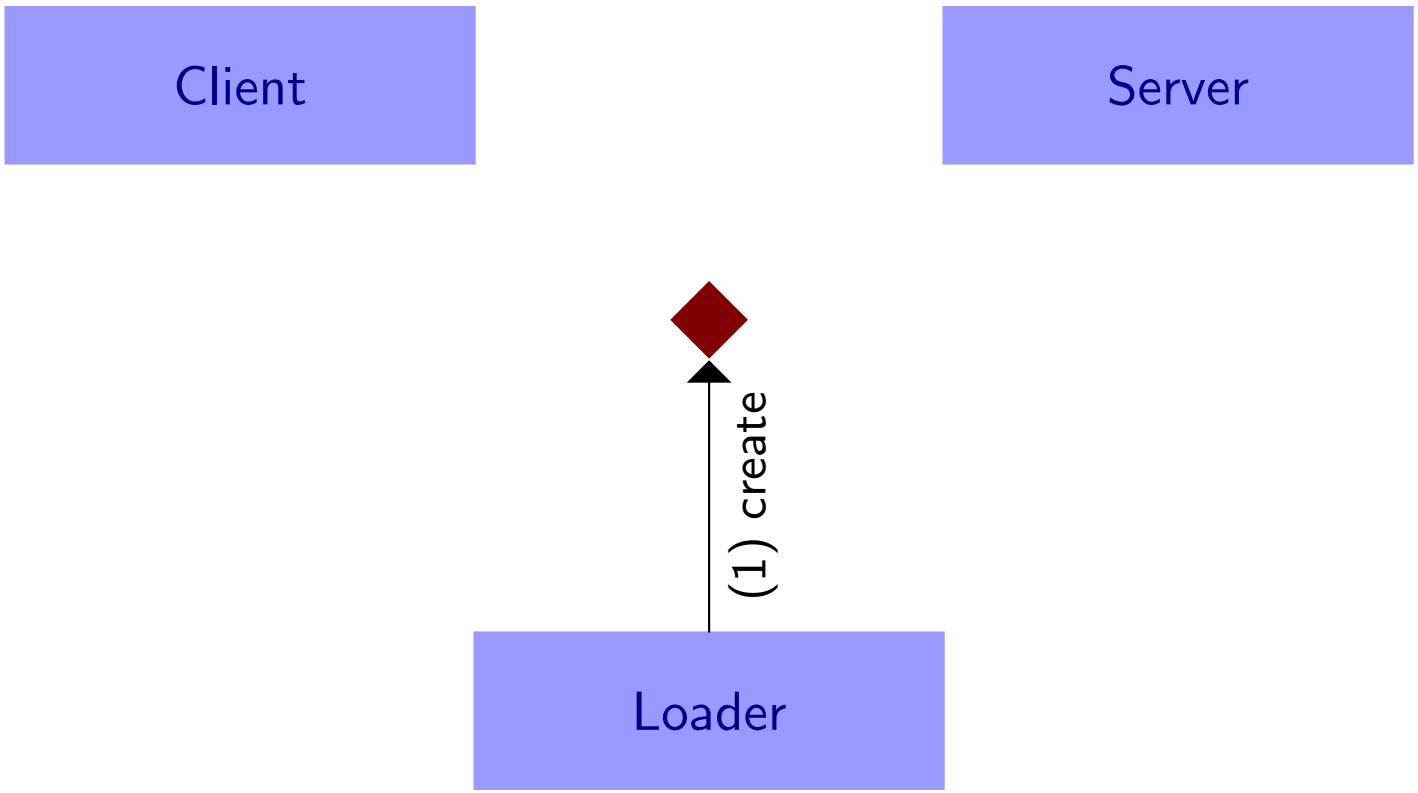
- L4Re Applications
  - Loader component: ned
  - Detects application termination: parent signal
  - Restart: re-execute Lua init script (or parts of it)
  - Problem after restart: capabilities
    - No single component knows everyone owning a capability to an object
    - Minix3 signals won't work

[3] Dirk Vogt, Björn Döbel, and Adam Lackorzynski. "Stay strong, stay safe: Enhancing reliability of a secure operating system". In: *Workshop on Isolation and Integration for Dependable Systems*. 2010, pp. 1–10.

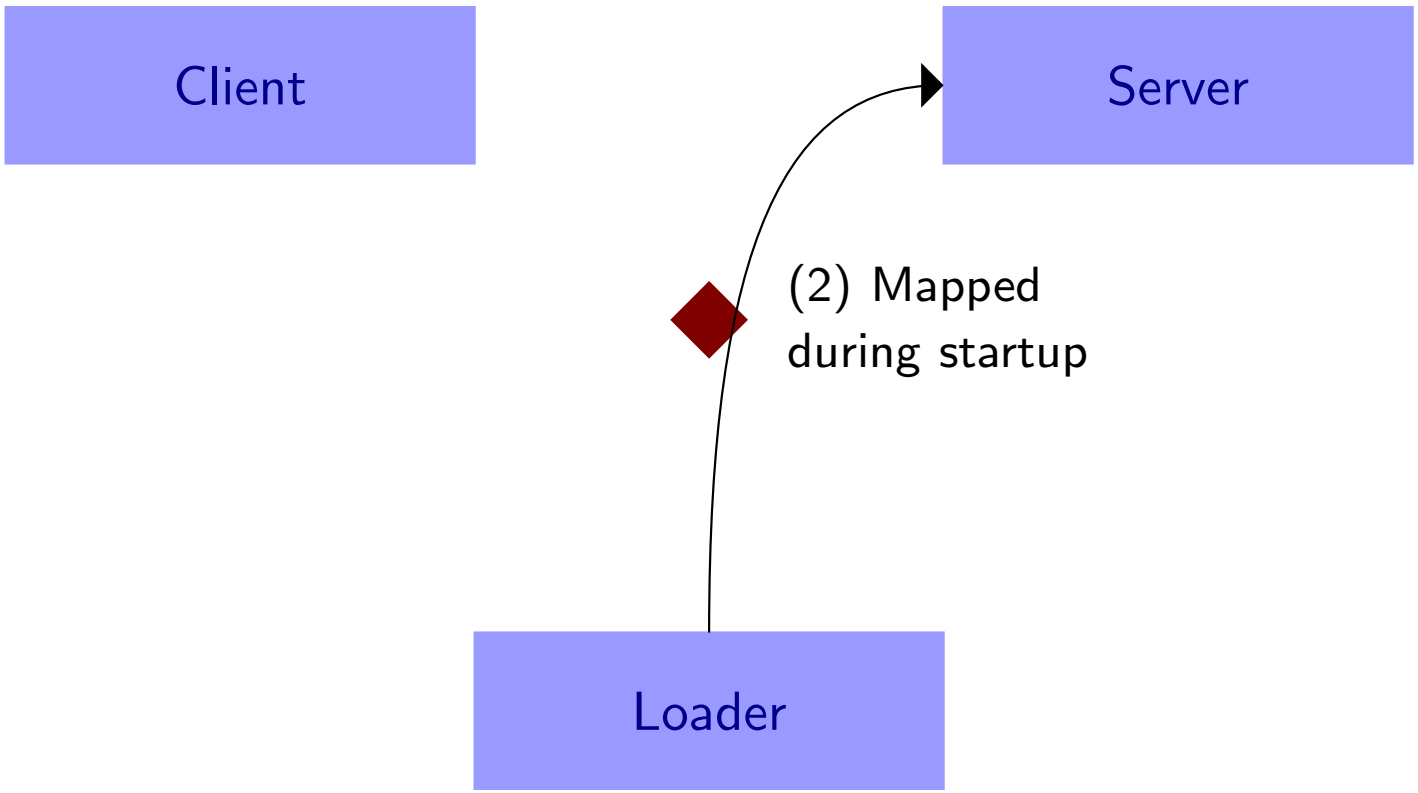
L4Re: Session Creation



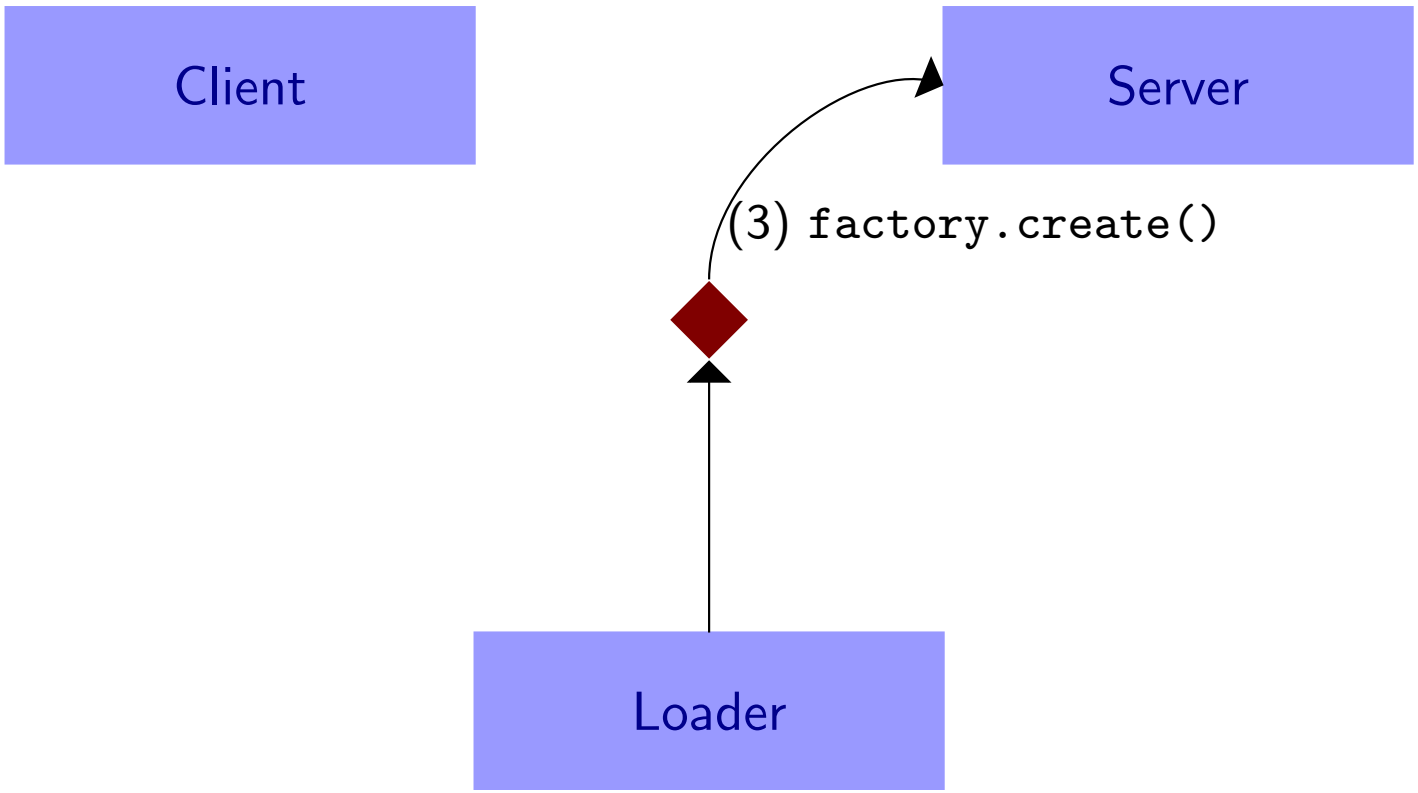
L4Re: Session Creation



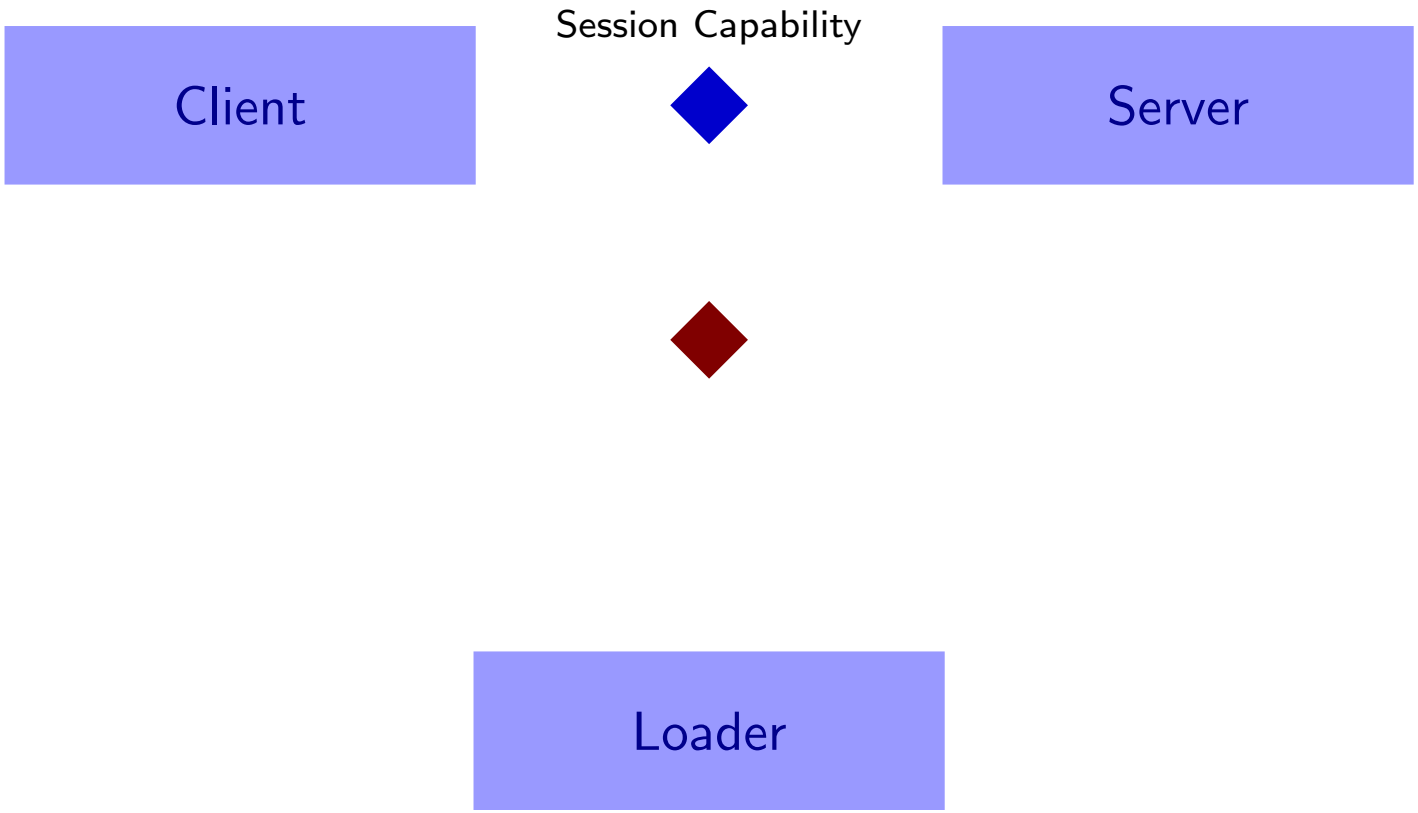
L4Re: Session Creation



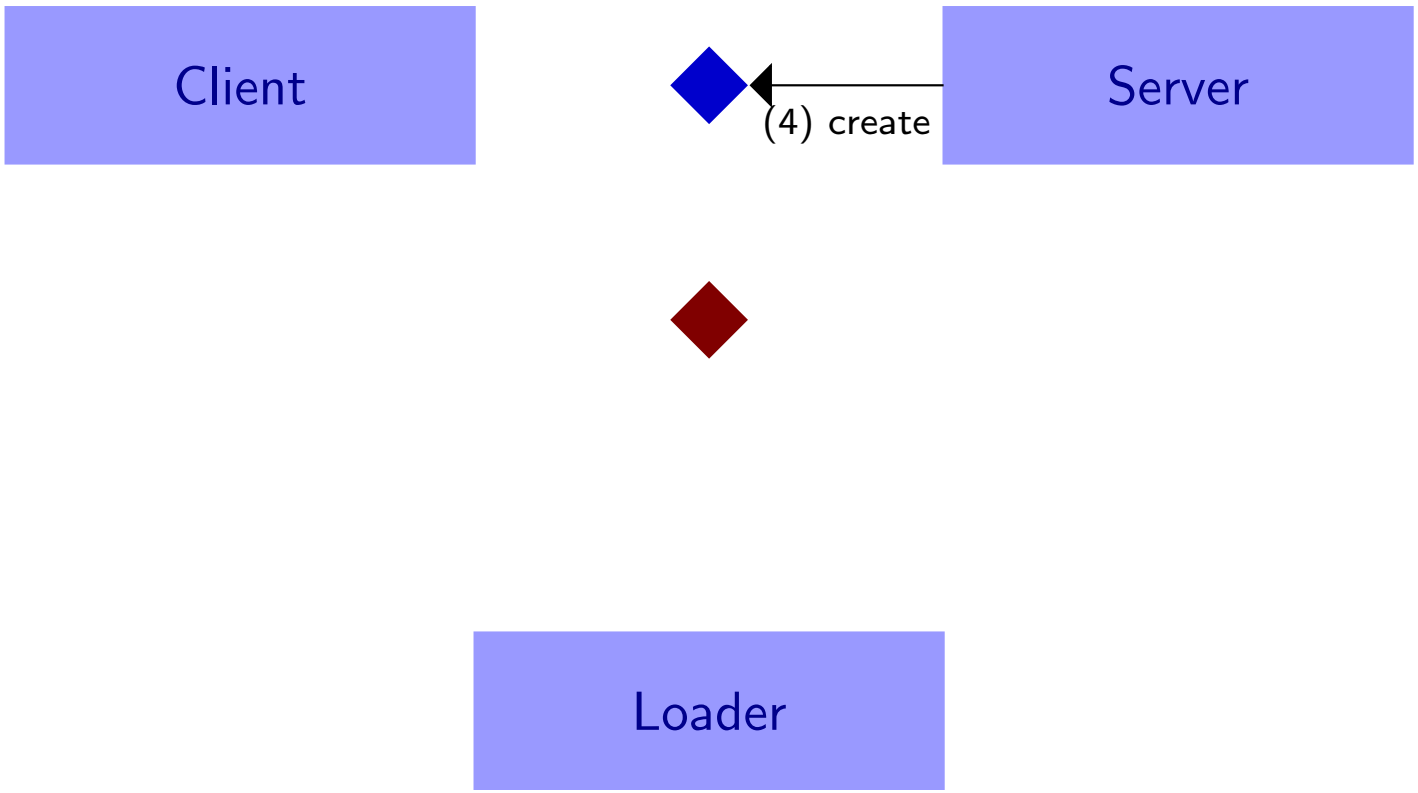
L4Re: Session Creation



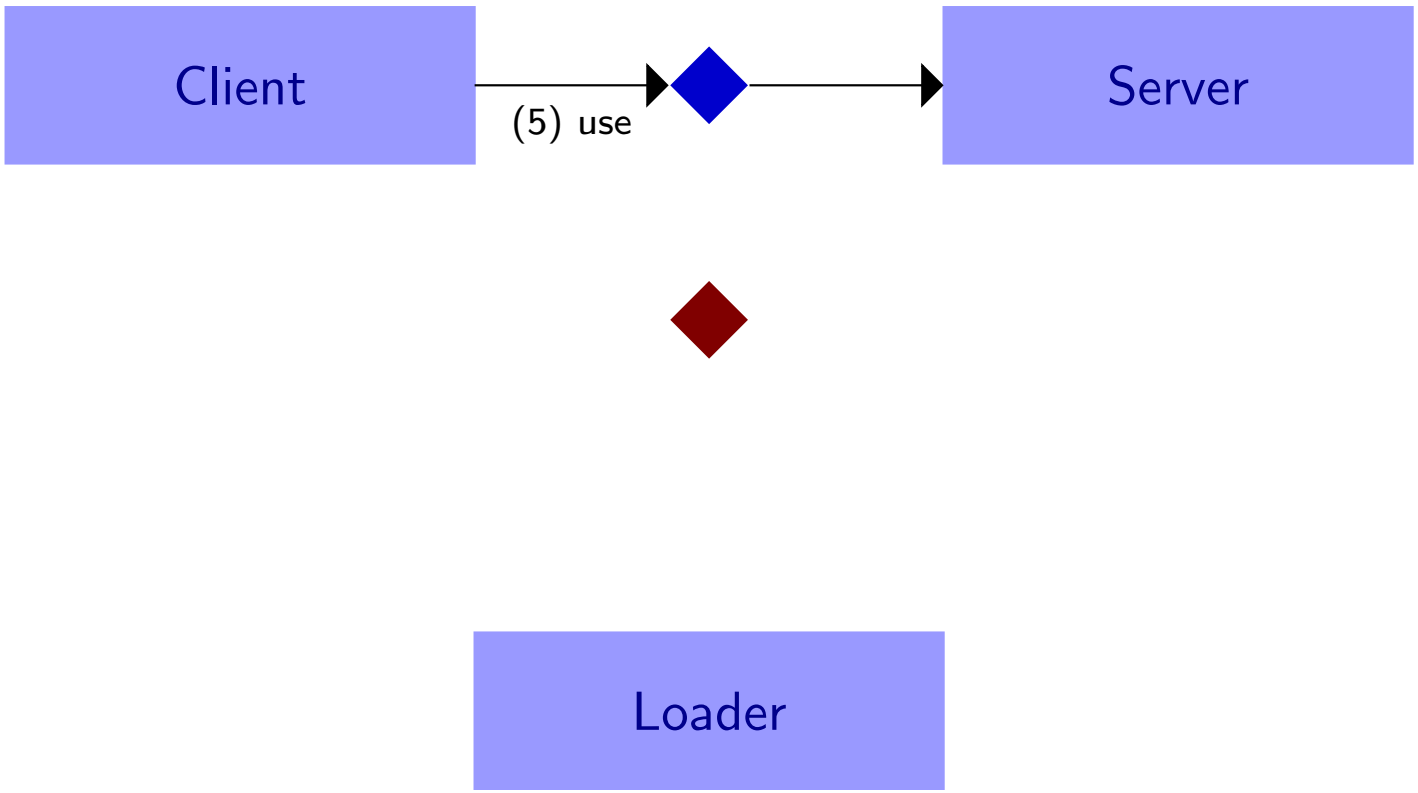
L4Re: Session Creation



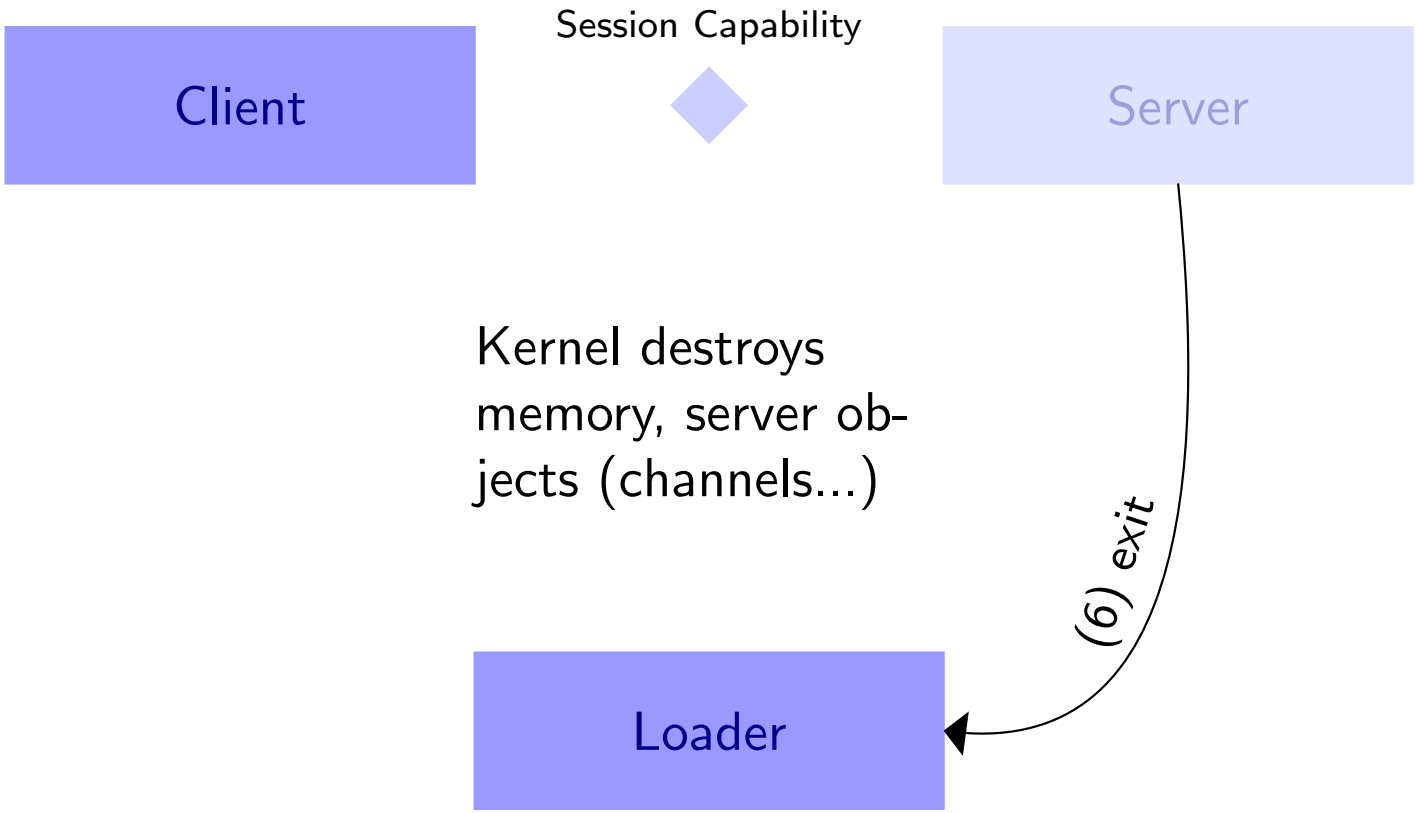
L4Re: Session Creation



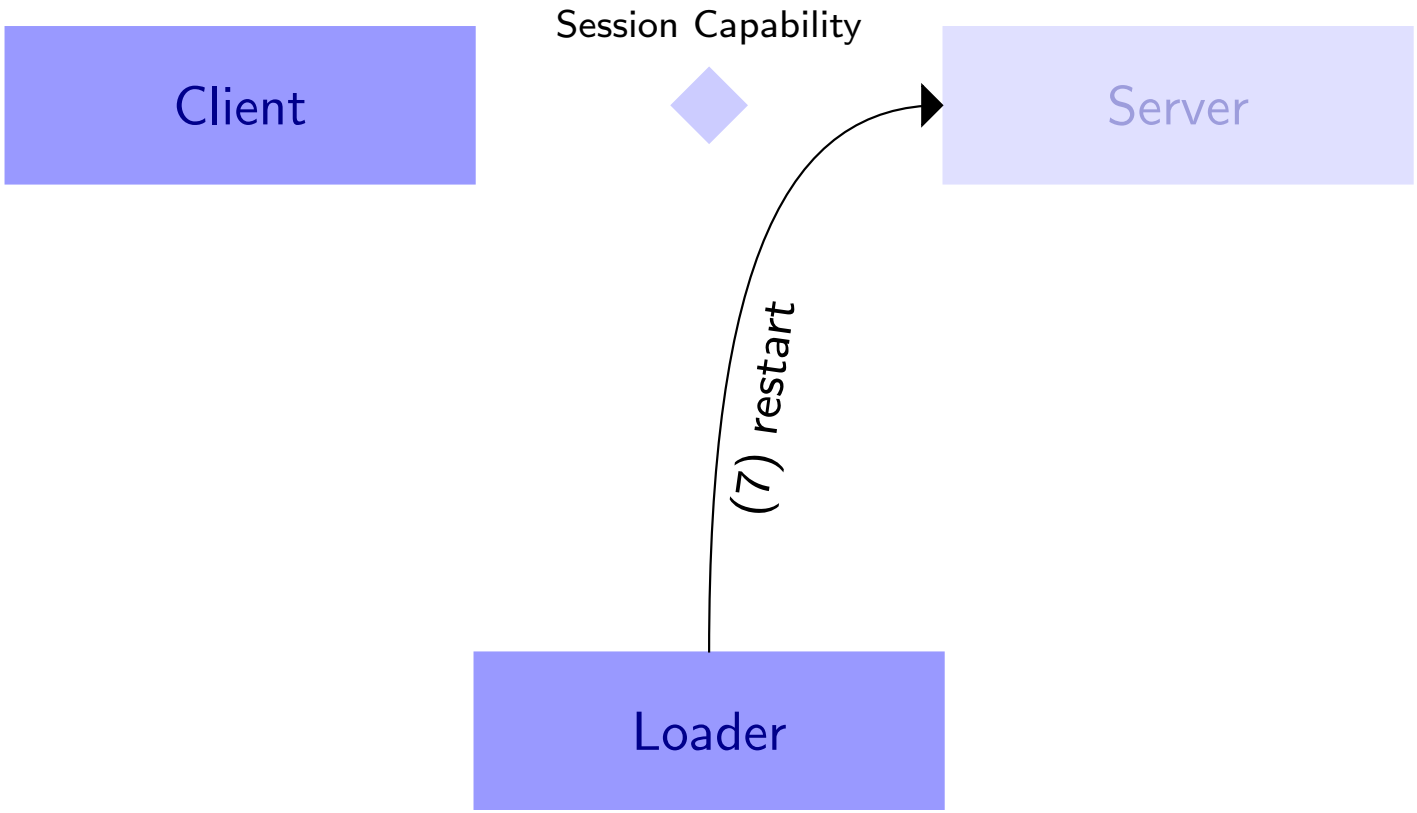
L4Re: Session Creation



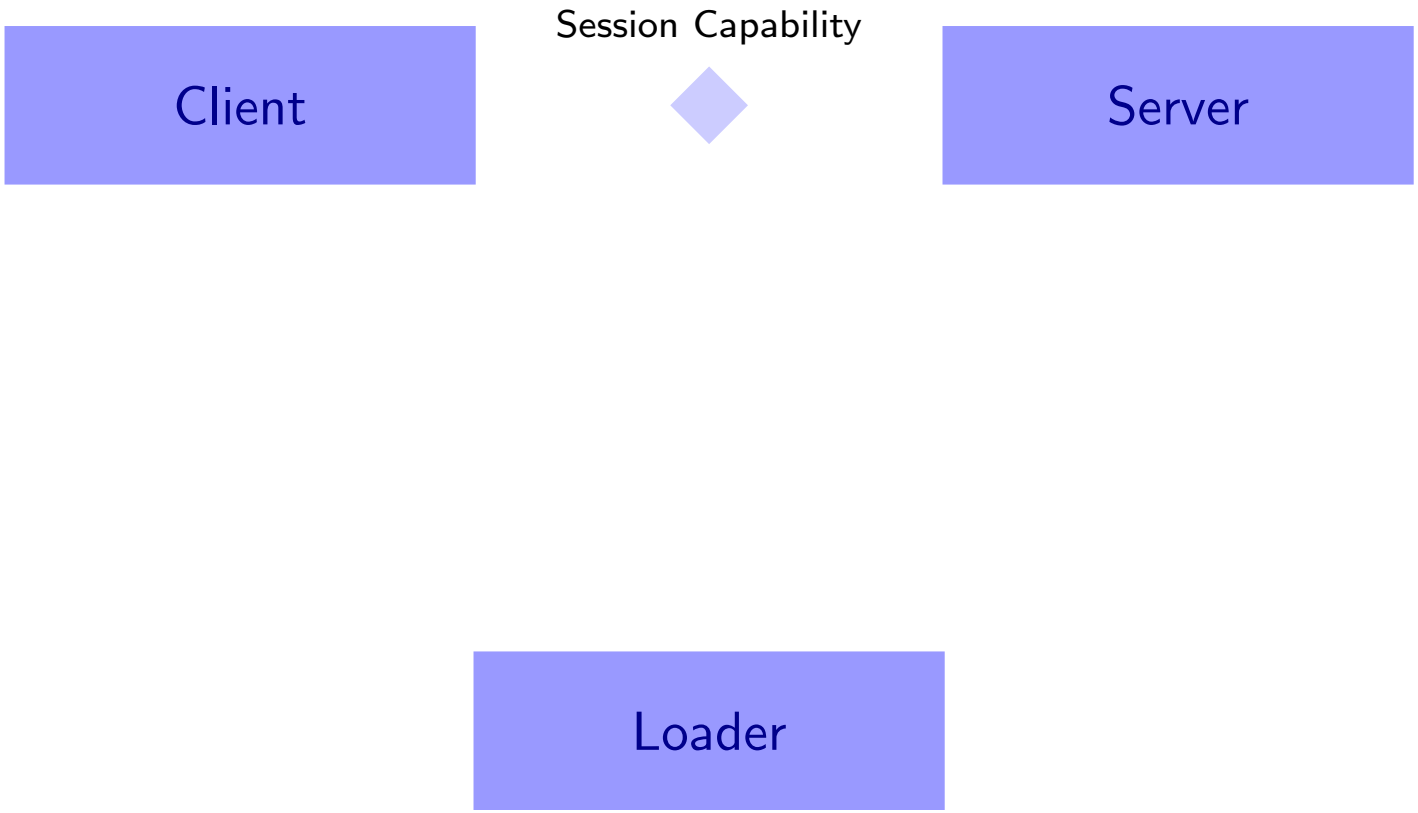
L4Re: Server Crash



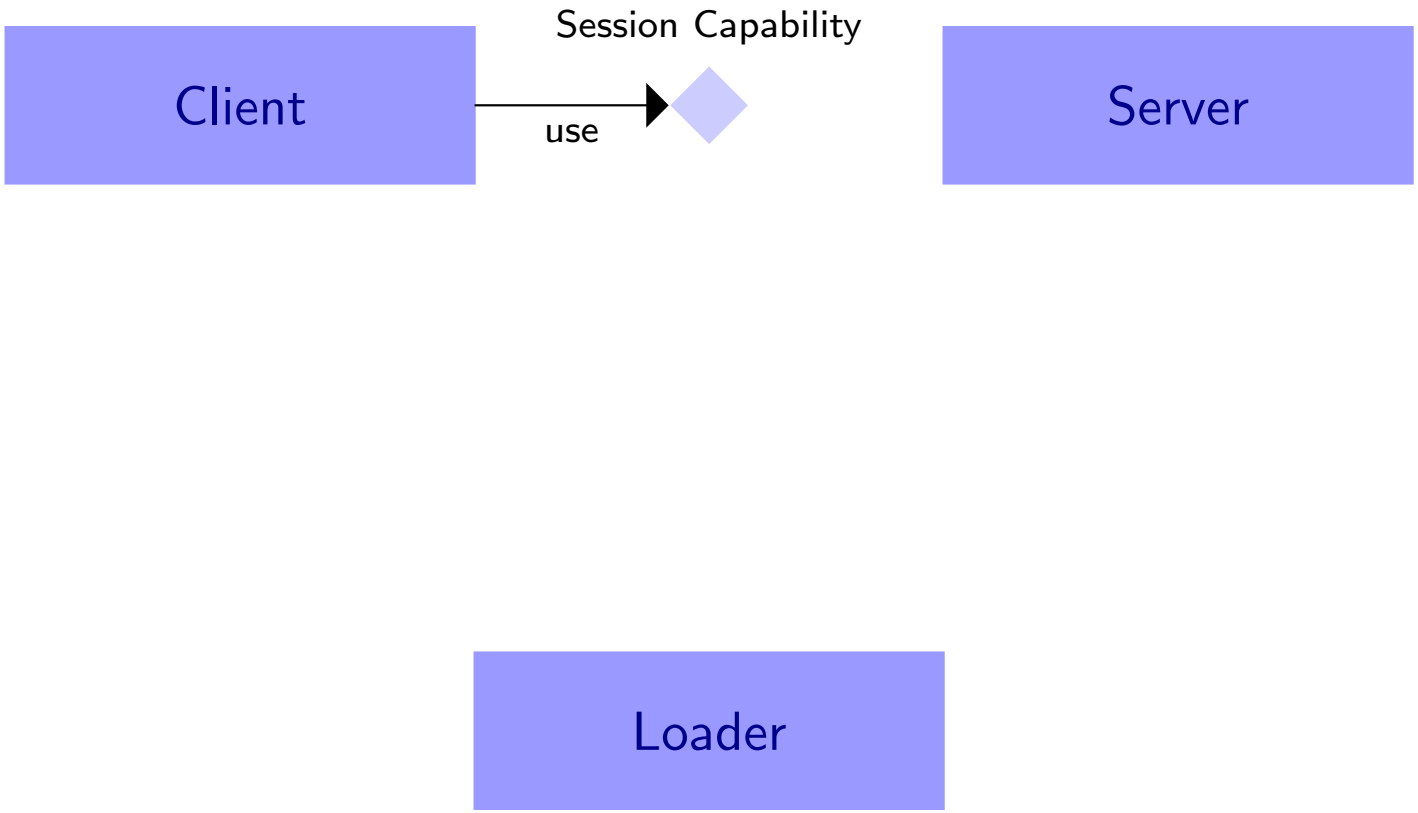
L4Re: Server Crash



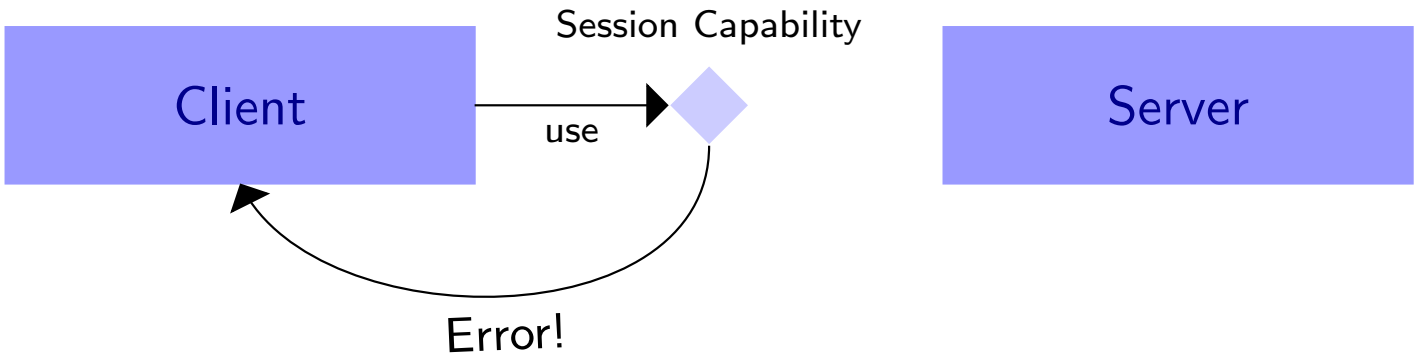
L4Re: Restarted Server



L4Re: Restarted Server



L4Re: Restarted Server



L4ReAnimator

- Only the application itself can detect that a capability vanished
- Kernel raises *Capability fault*
- Application needs to re-obtain the capability: execute *capability fault handler*
- Capfault handler: application-specific
  - Create new communication channel
  - Restore session state
- Programming model:
  - Capfault handler provided by server implementor
  - Handling transparent for application developer
  - *Semi-transparency*

L4ReAnimator: Cleanup

- Some channels have resources attached (e.g., frame buffer for graphical console)
- Resource may come from a different resource (e.g., frame buffer from memory manager)
- Resources remain intact (stale) upon crash
- Client ends up using old version of the resource
- Requires additional app-specific knowledge
- *Unmap handler*

Summary

- L4ReAnimator
  - Capfault: Clients detect server restarts lazily
  - Capfault Handler: application-specific knowledge on how to regain access to the server
  - Unmap handler: clean up old resources after restart

seL4: Formal verification of an OS kernel[4]

- seL4: `https://sel4.systems/`
- Formally verify that system adheres to specification
- Microkernel design allows to separate components easier
- Hence verification process is easier

[4] Gerwin Klein et al. "seL4: Formal verification of an OS kernel". In: *SOSP*. 2009, pp. 207–220.

Verification of a microkernel

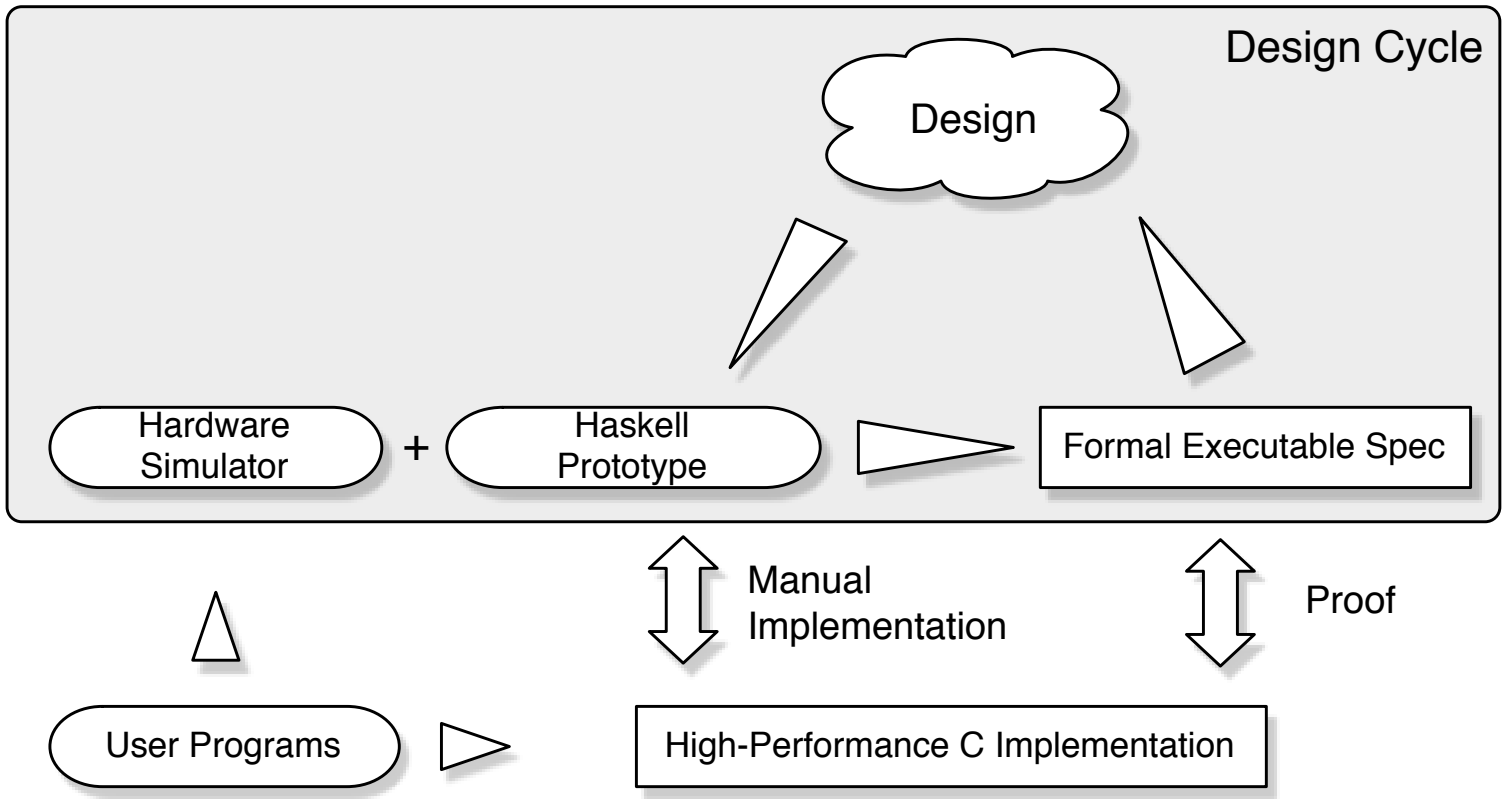


Figure: The seL4 design process [11]

Refinement of verification

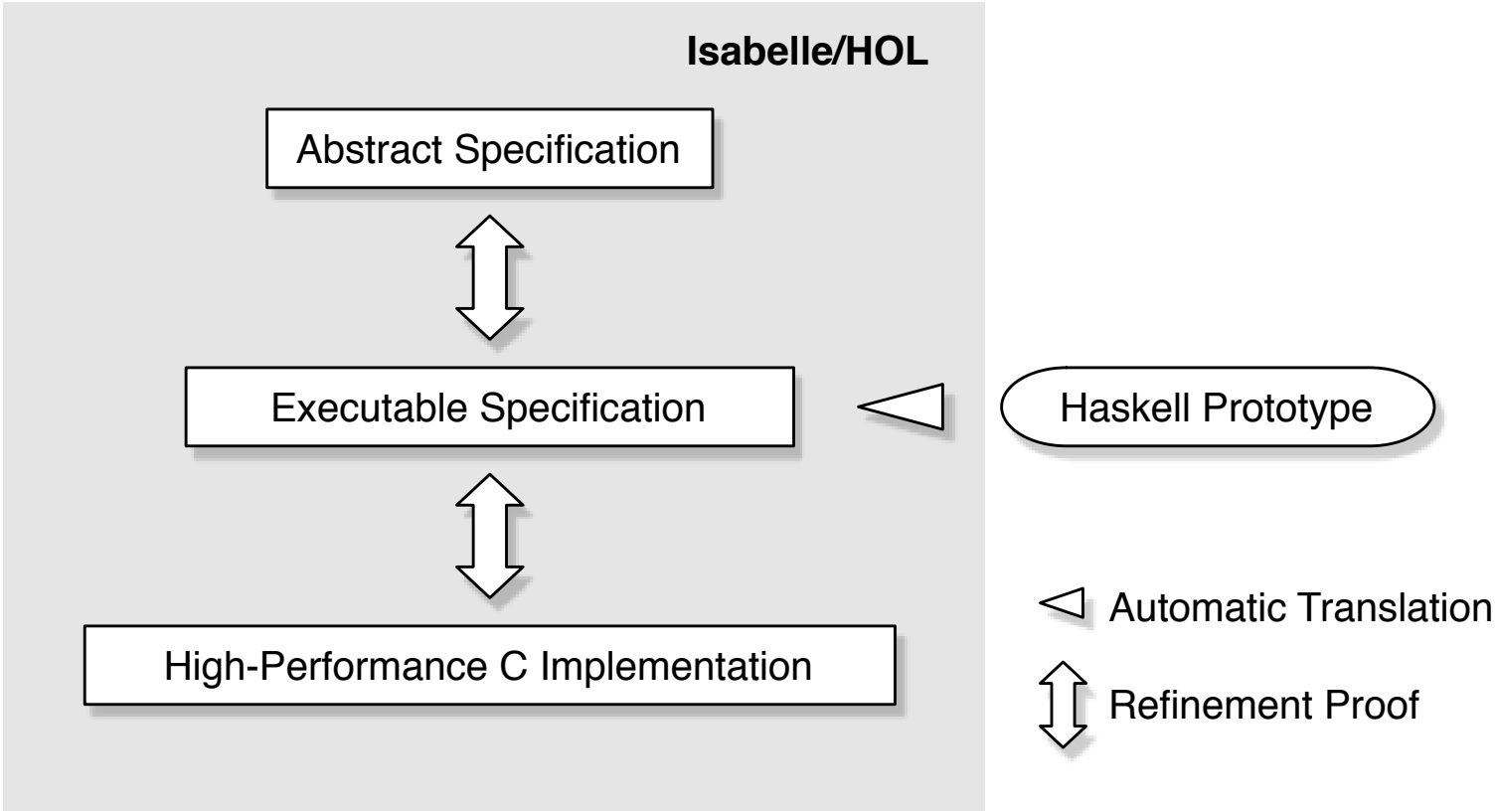


Figure: Refinement layers in the verification of seL4 [11]

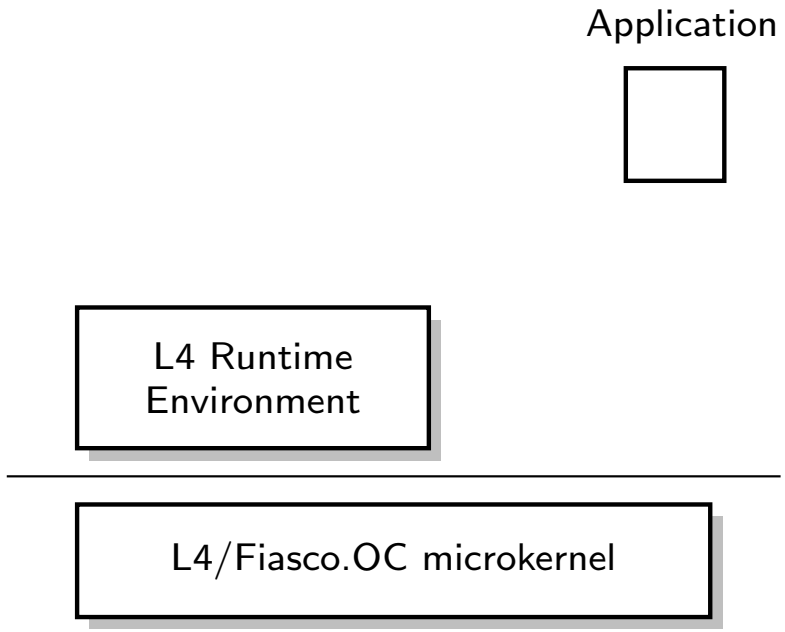
Break

- seL4
  - Assumes correctness of compiler, assembly code, and hardware
  - DMA over IOMMU
  - Architectures: arm, x86
  - Virtualization
  - Future: Verification on multicores
- All these frameworks only deal with software errors.
- What about hardware faults?

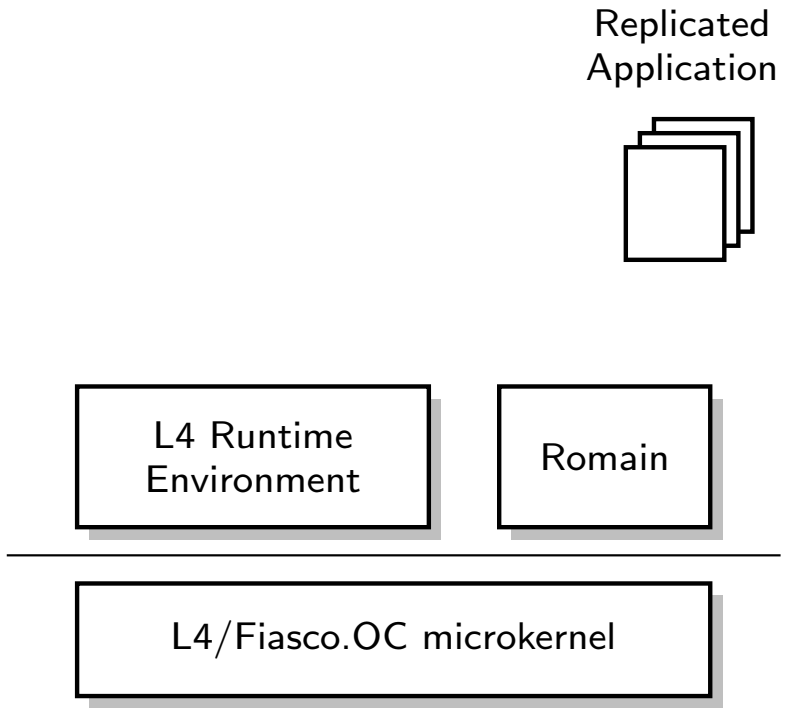
Transient Hardware Faults

- Radiation-induced soft errors
  - Mainly an issue in avionics+space?
- DRAM errors in large data centers
  - Google study: >2% failing DRAM DIMMs per year [14]
  - ECC insufficient [10]
- Decreasing transistor sizes $\rightarrow$ higher rate of errors in CPU functional units [5]

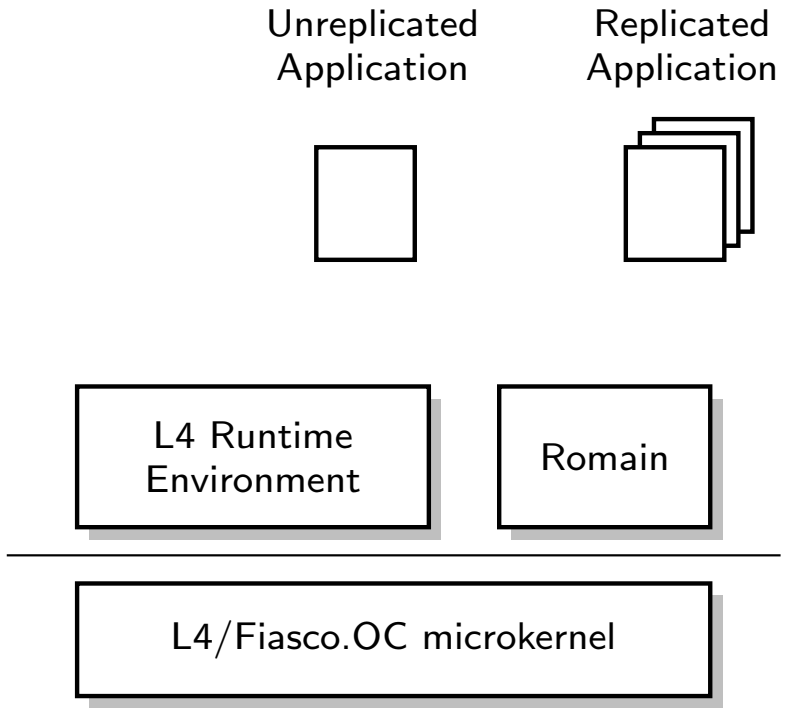
Transparent Replication as OS Service [7, 6]



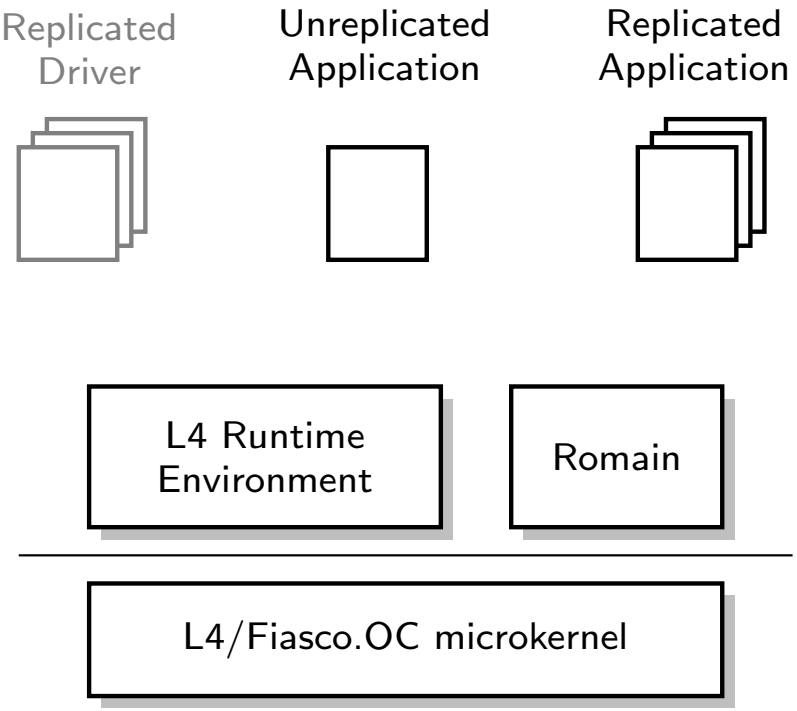
Transparent Replication as OS Service [7, 6]



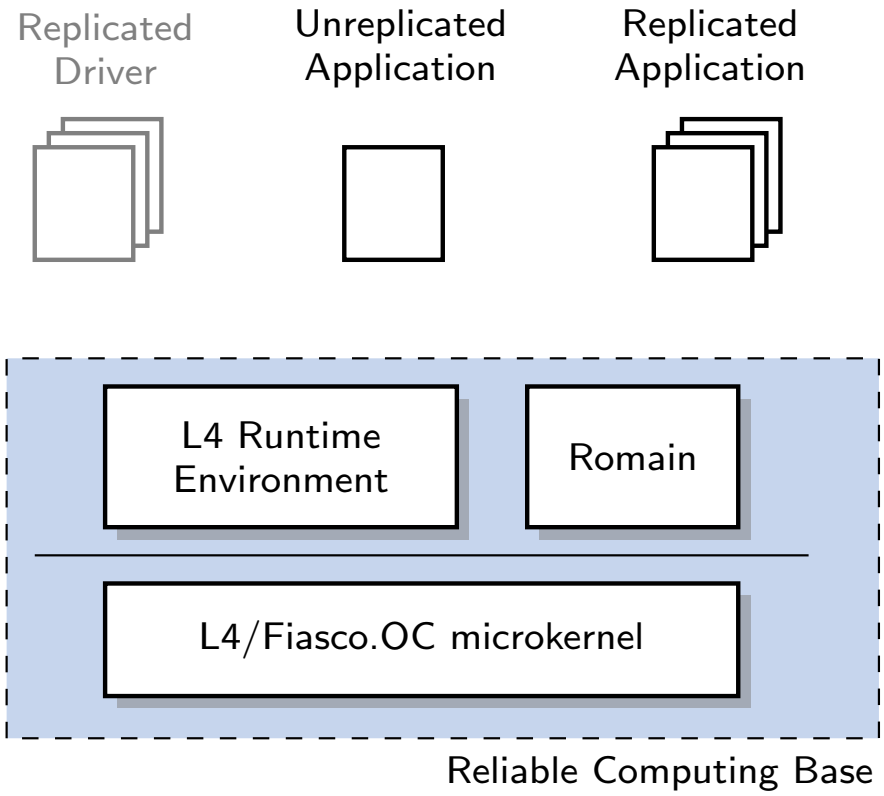
Transparent Replication as OS Service [7, 6]



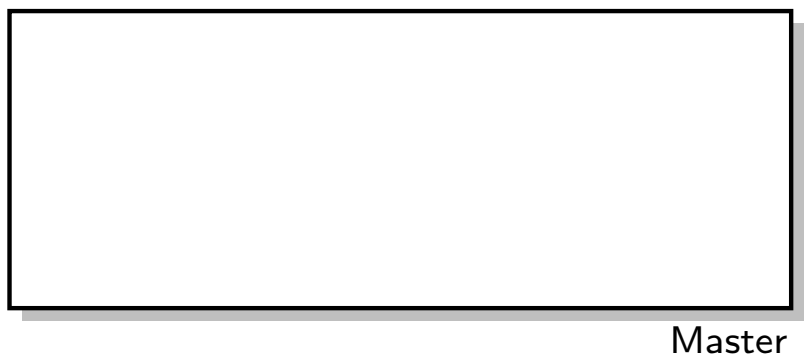
Transparent Replication as OS Service [7, 6]



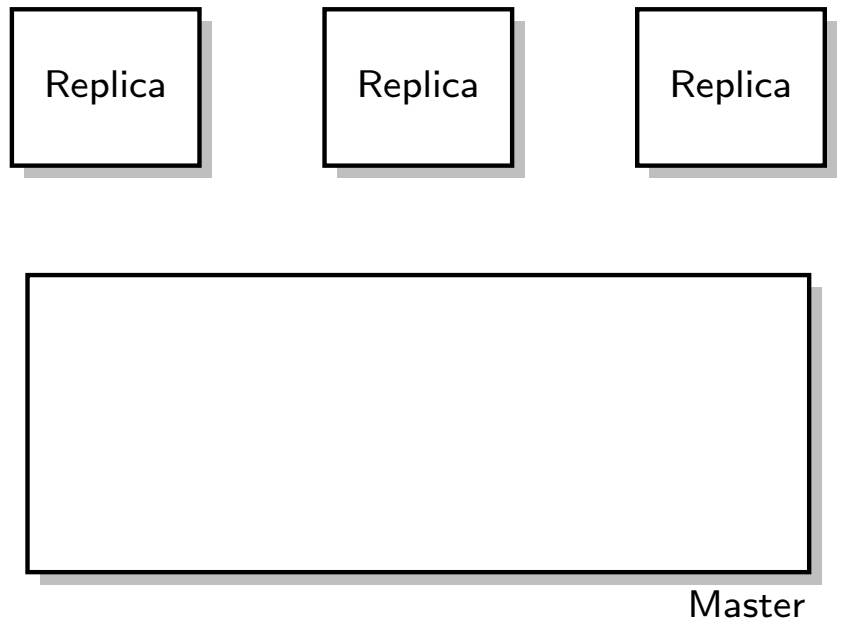
Transparent Replication as OS Service [7, 6]



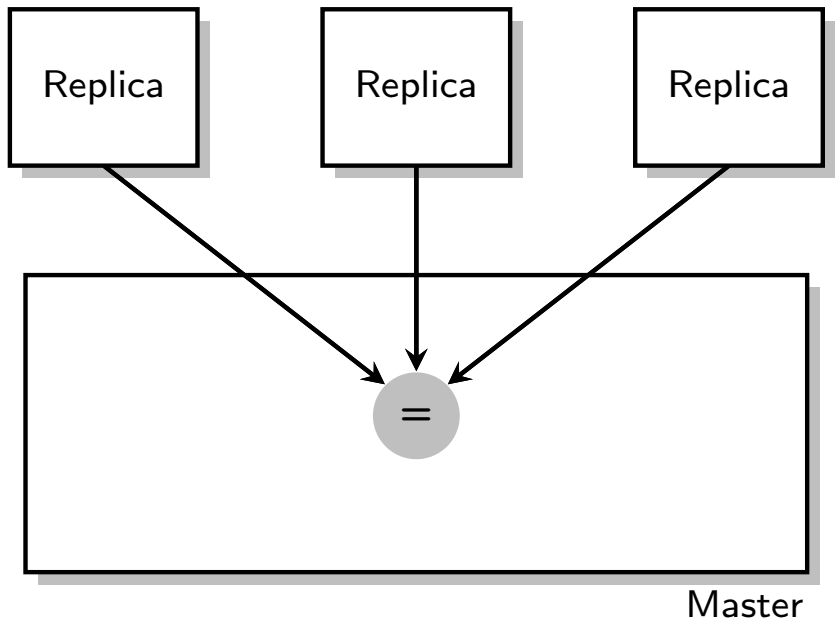
Romain: Structure



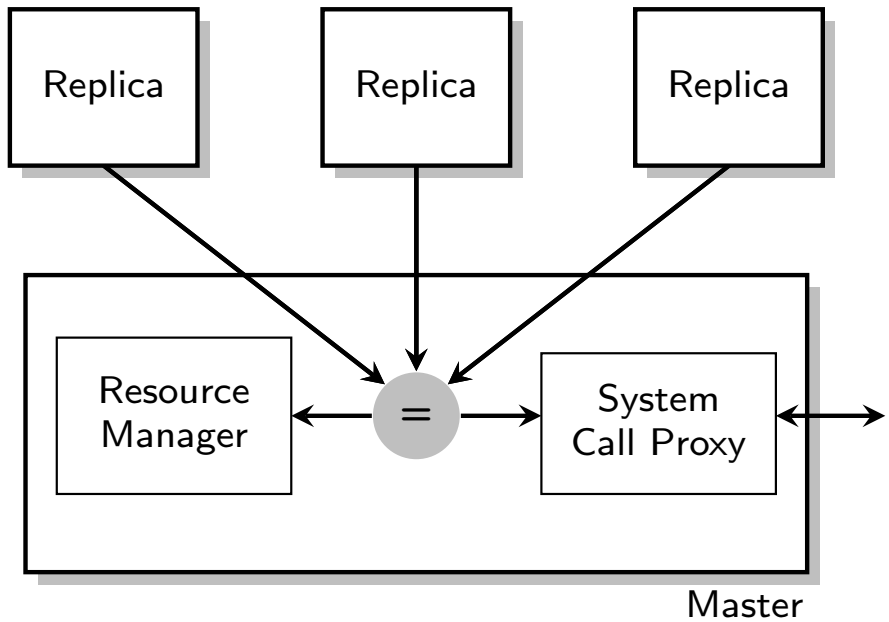
Romain: Structure



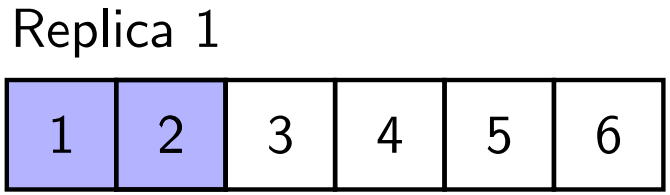
Romain: Structure



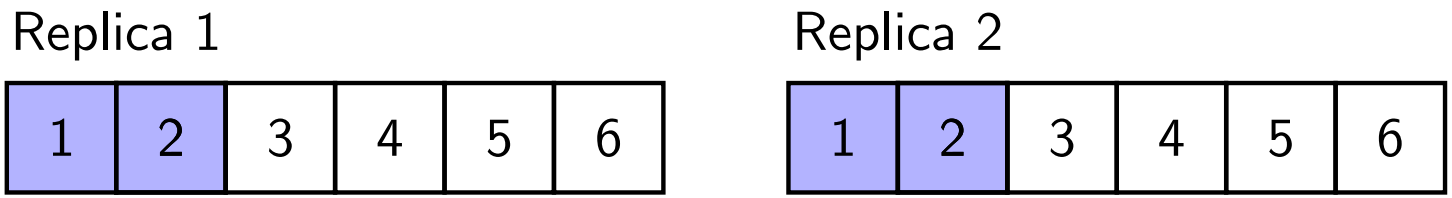
Romain: Structure



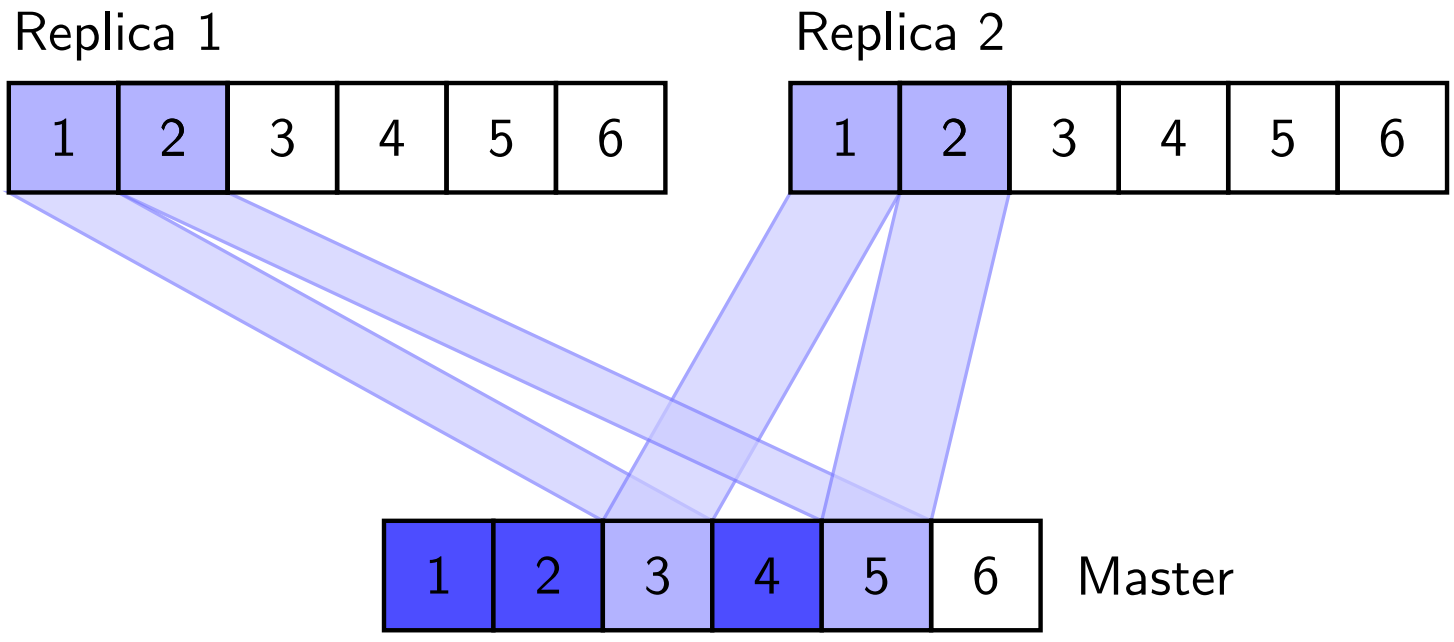
Resource Management: Capabilities



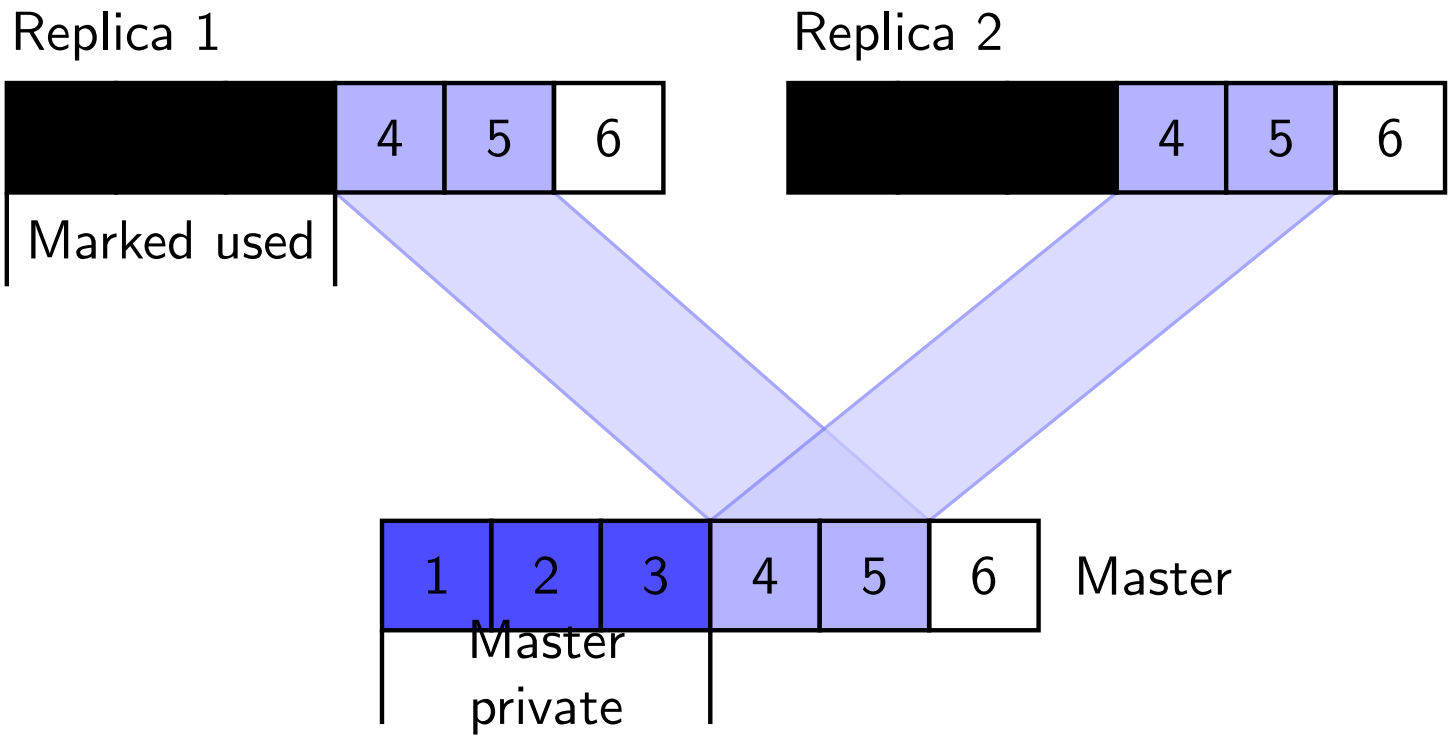
Resource Management: Capabilities



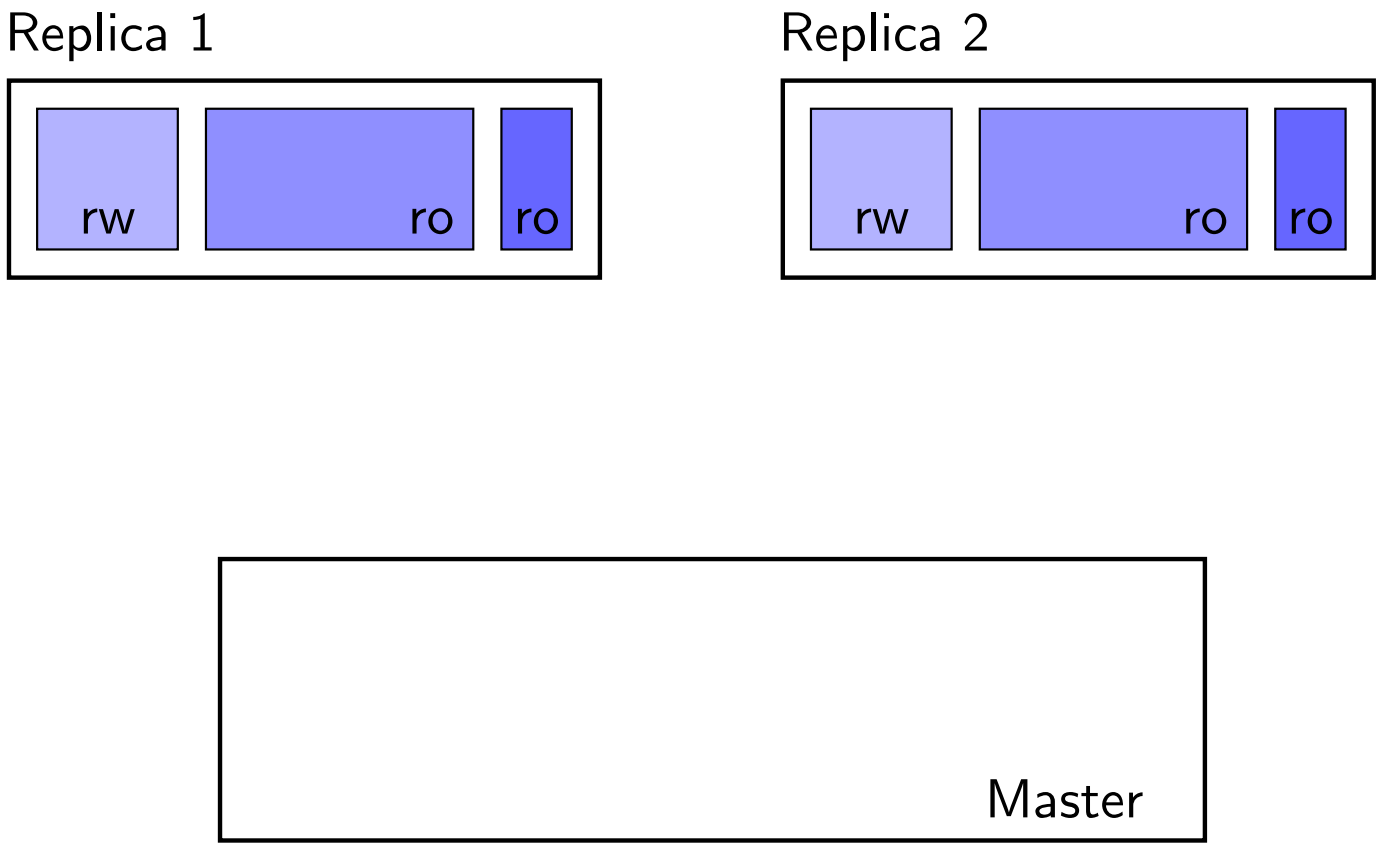
Resource Management: Capabilities



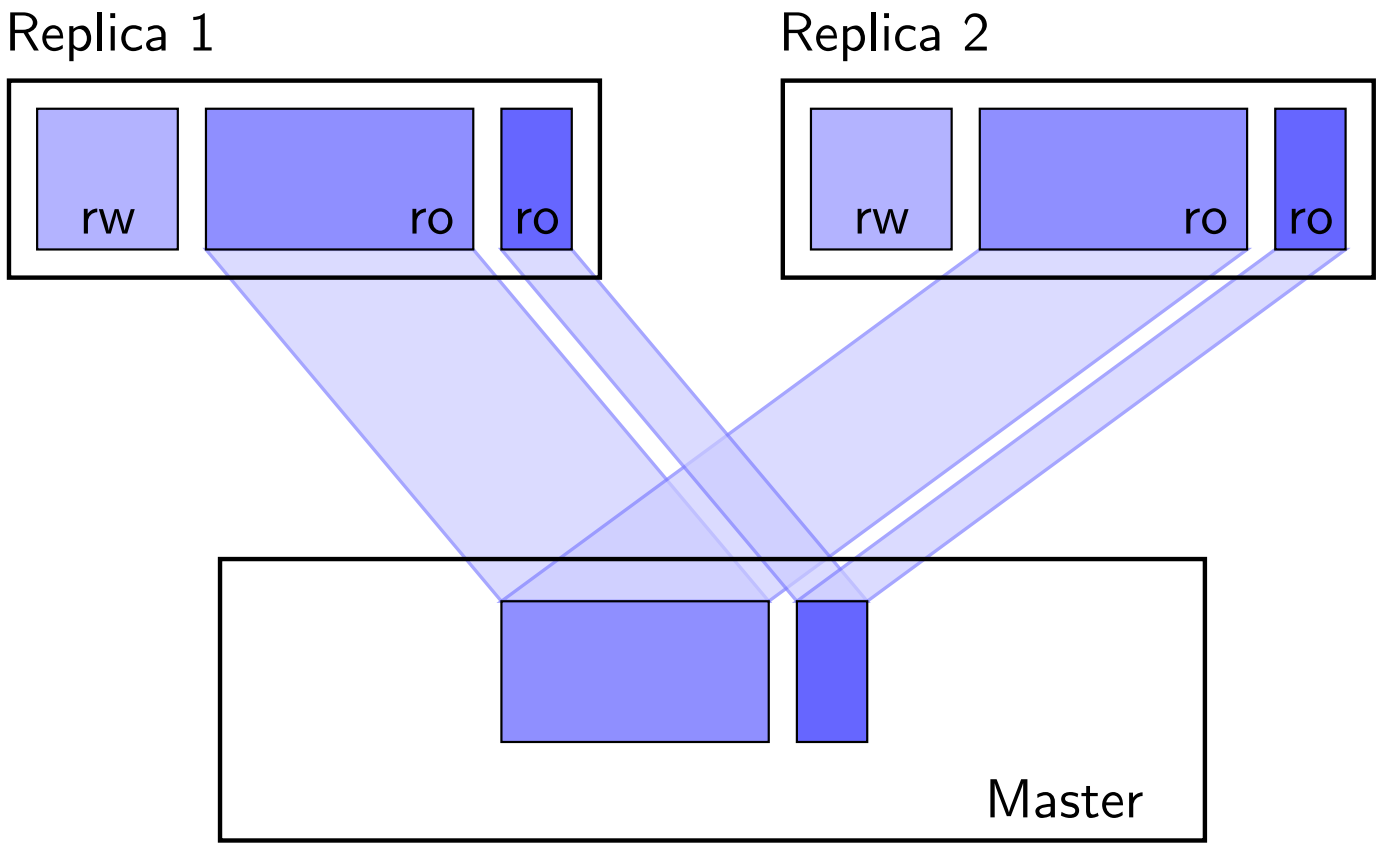
Partitioned Capability Tables



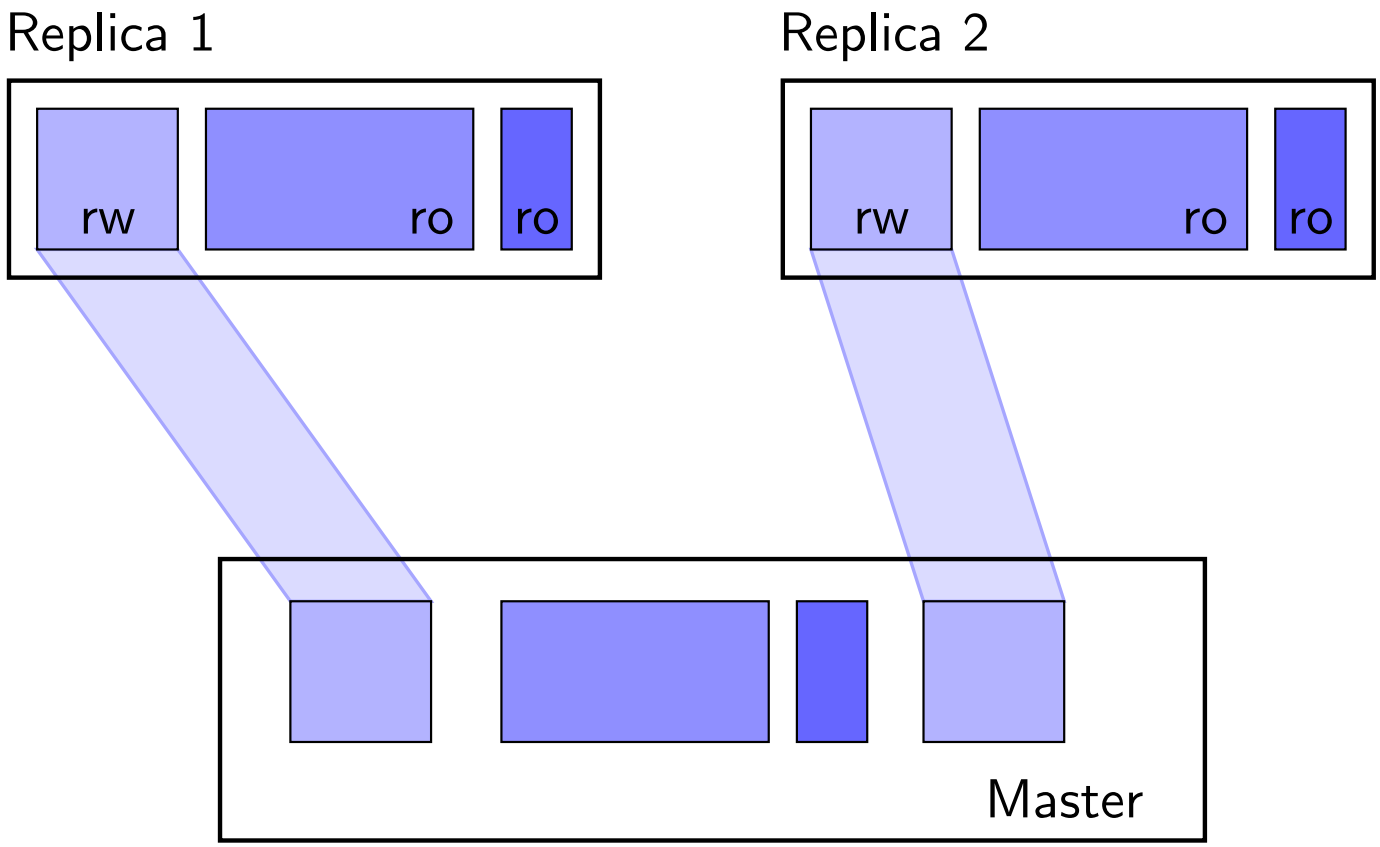
Replica Memory Management



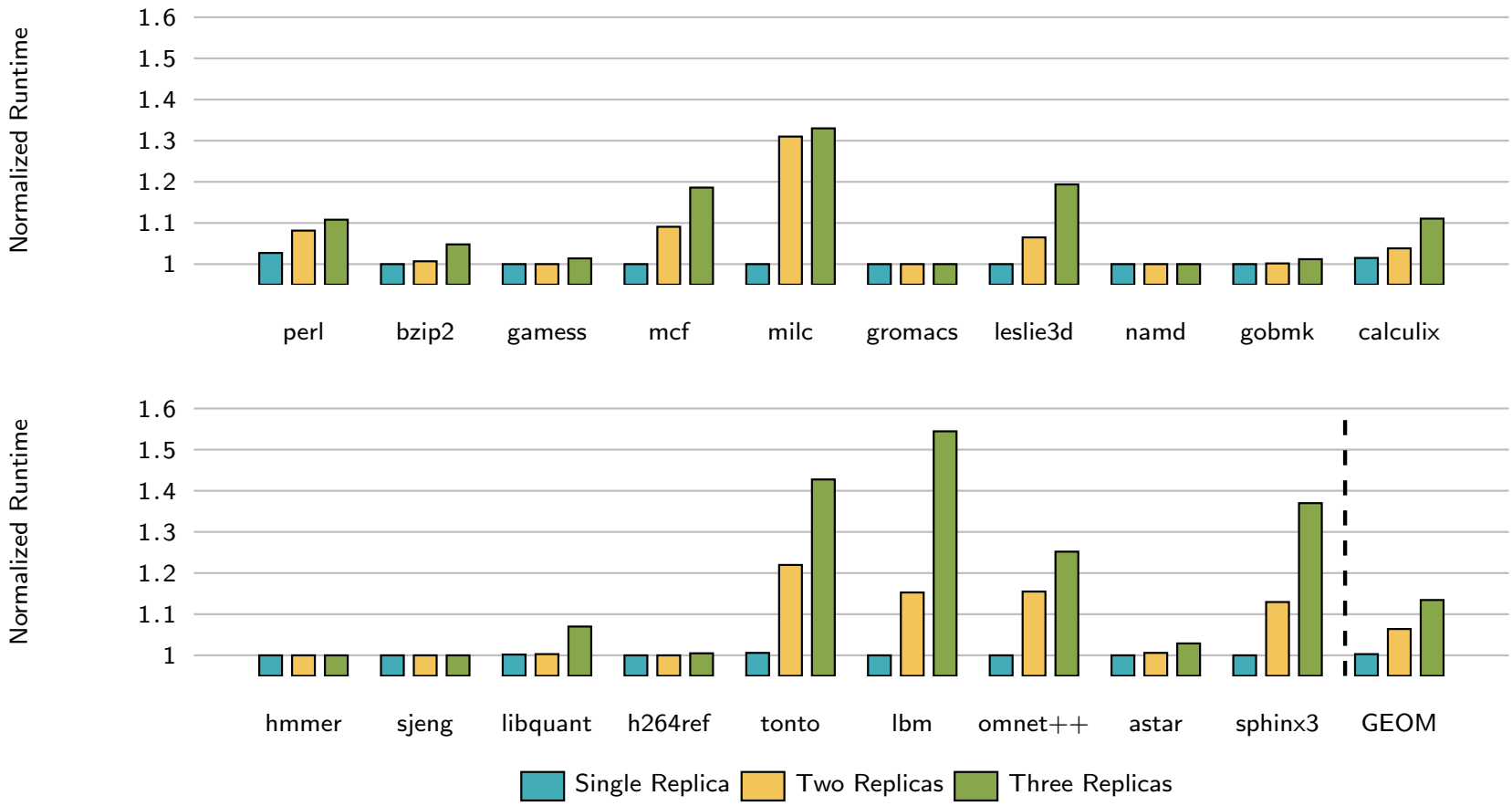
Replica Memory Management



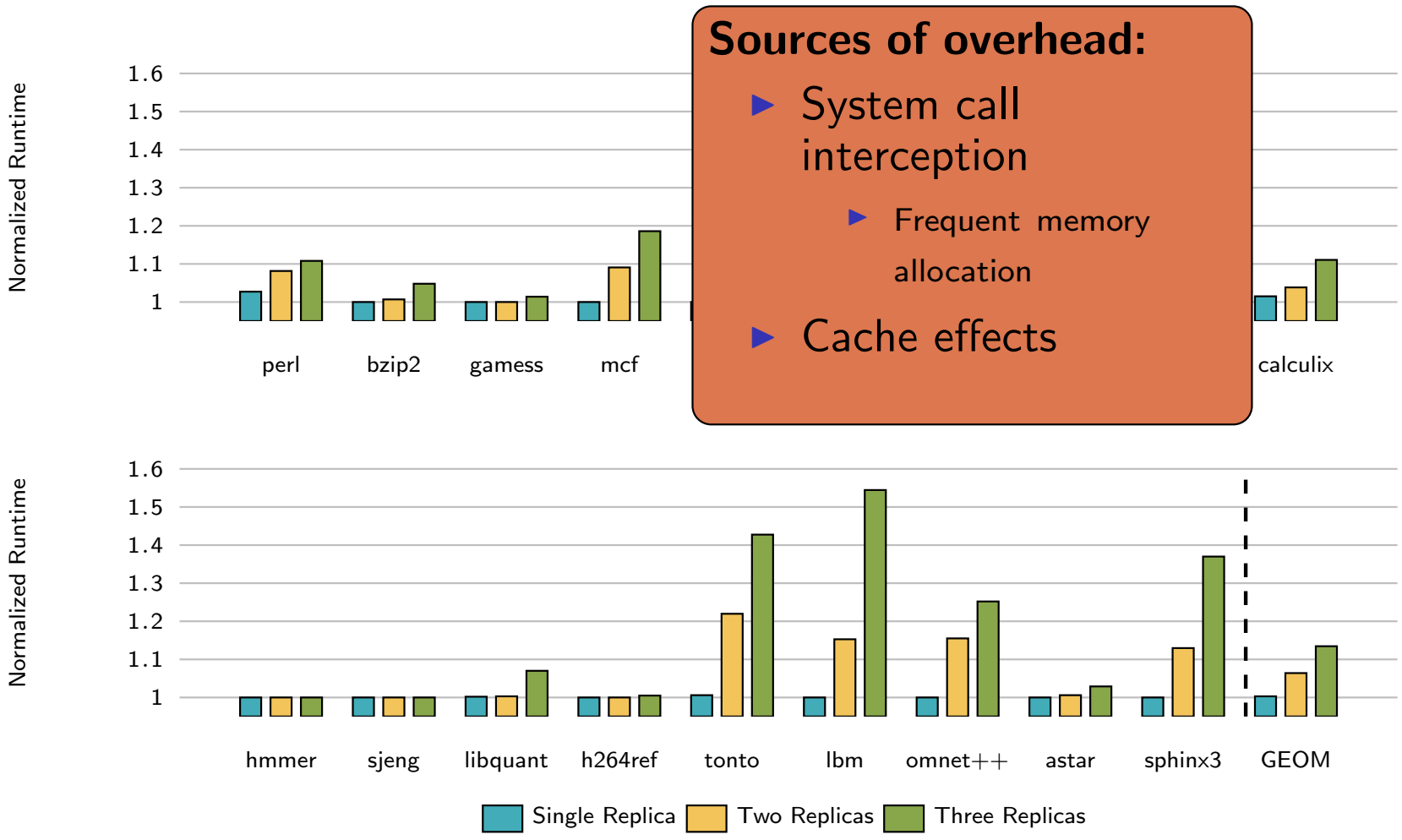
Replica Memory Management



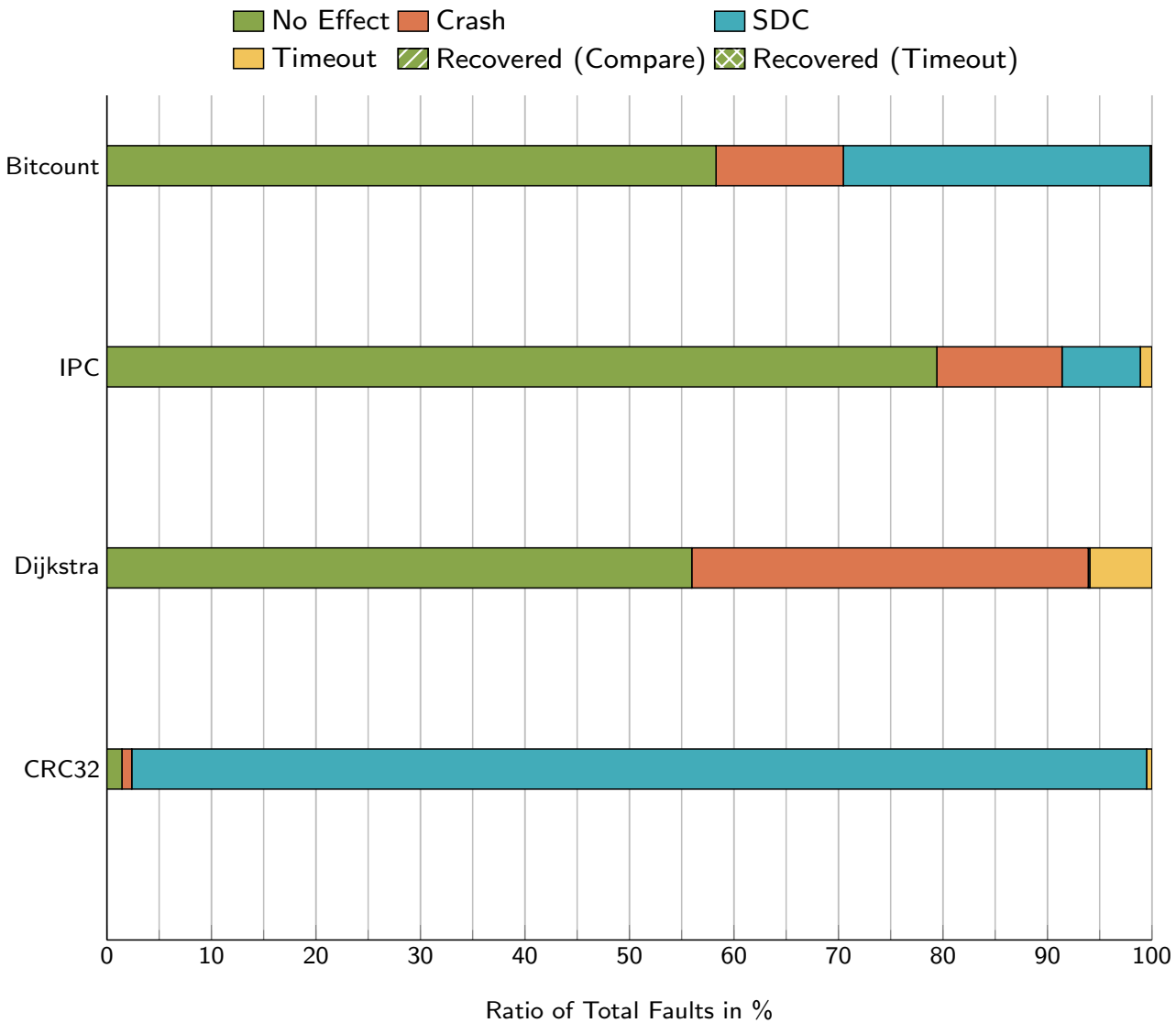
Replicating SPEC CPU 2006 [8]



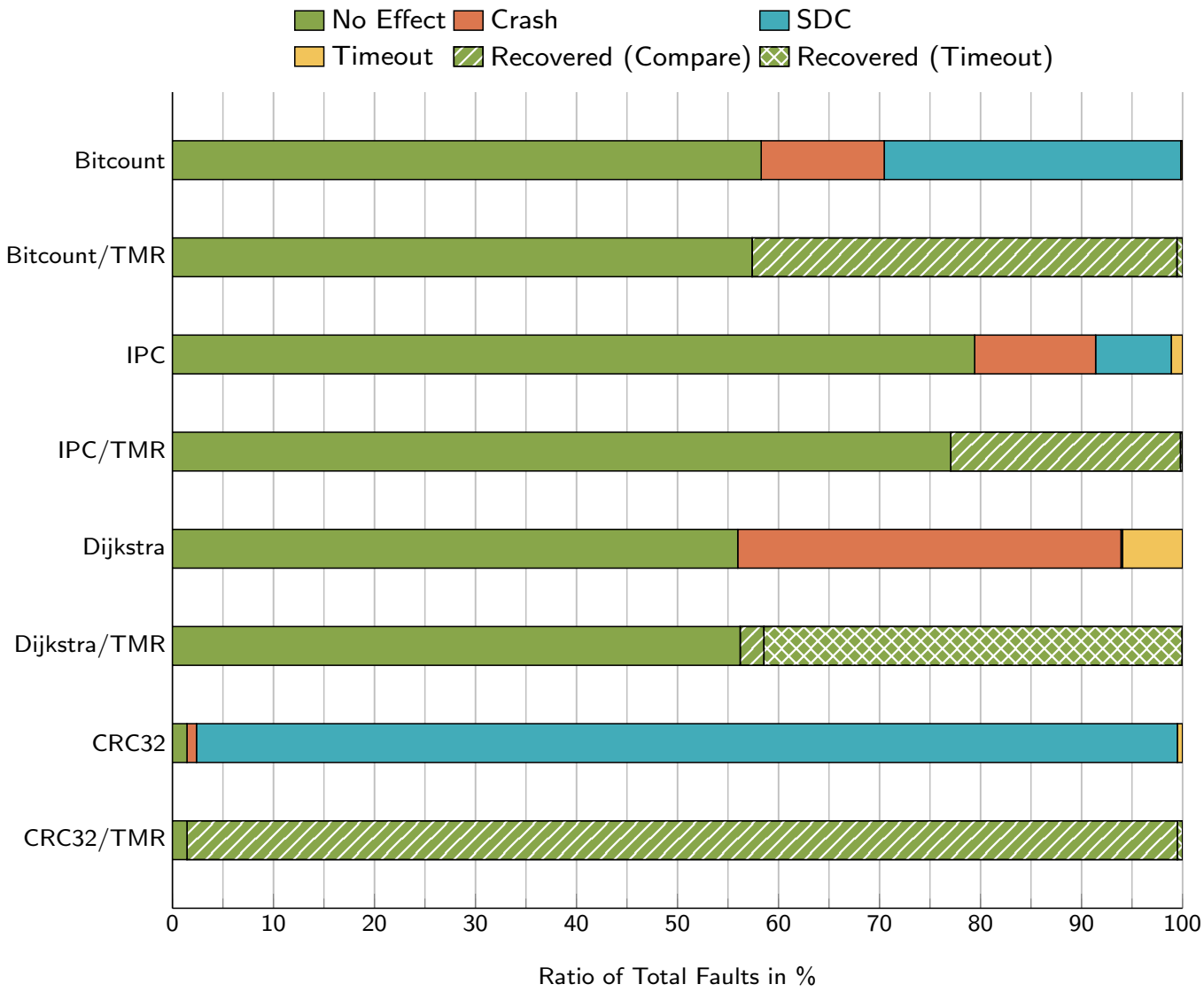
Replicating SPEC CPU 2006 [8]



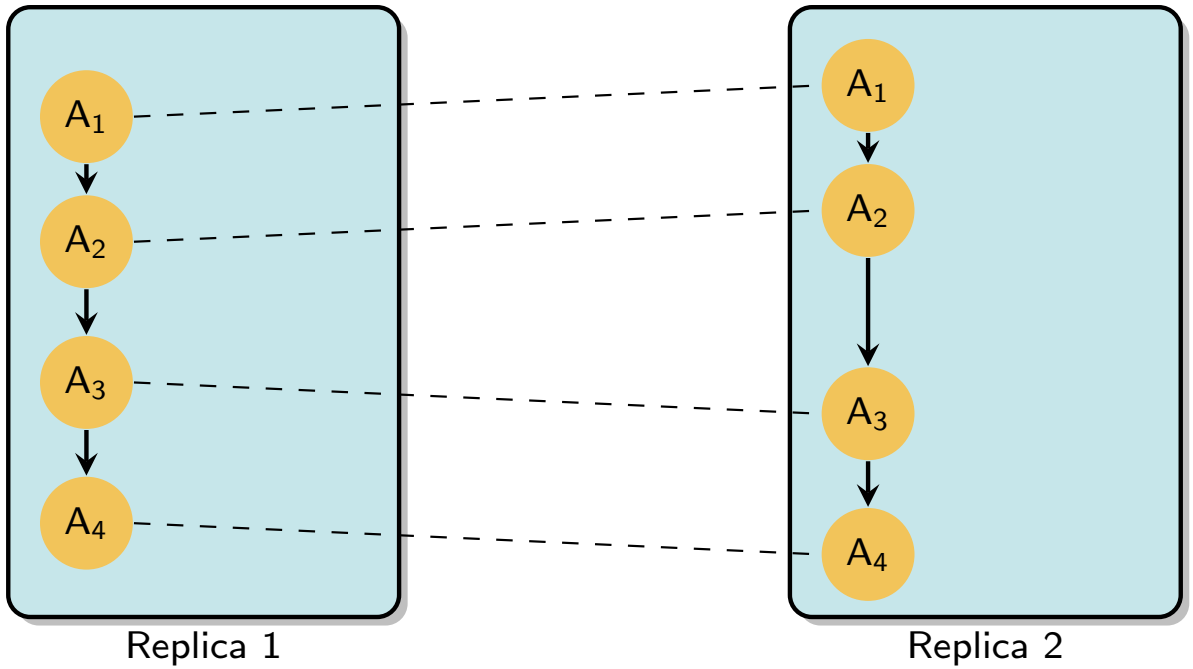
Error Coverage [8]



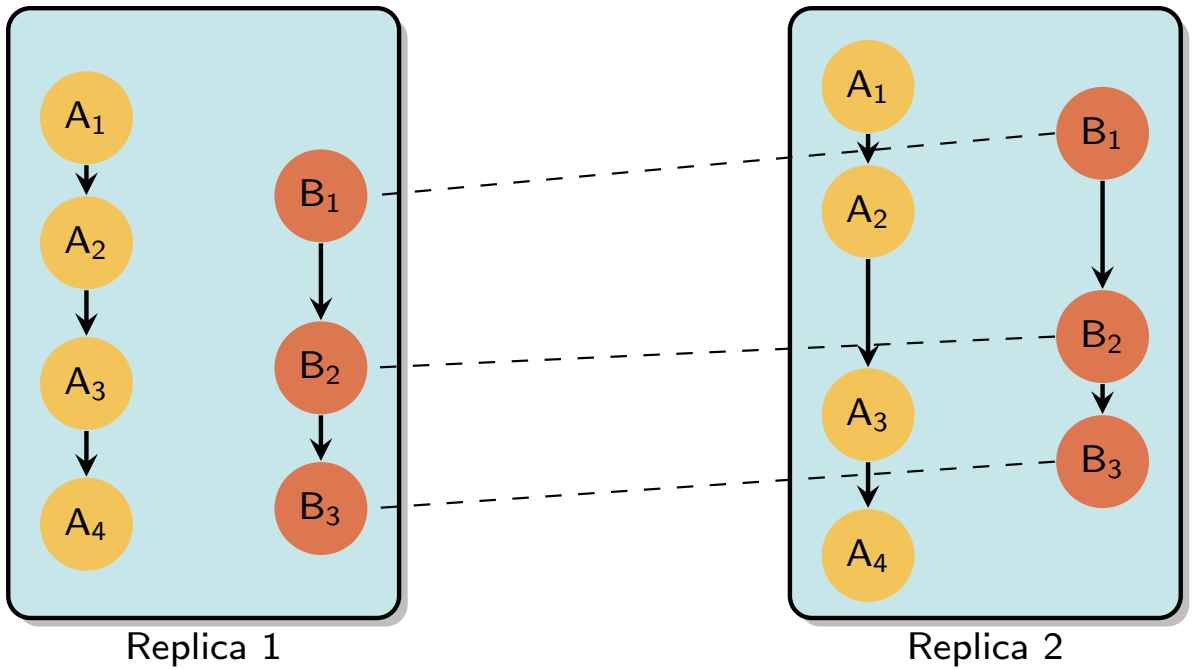
Error Coverage [8]



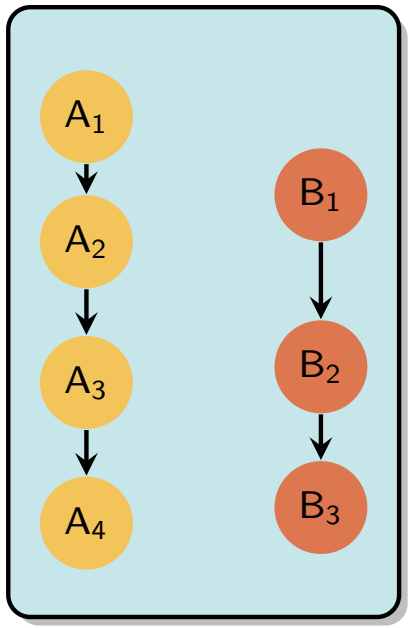
How About Multithreading?



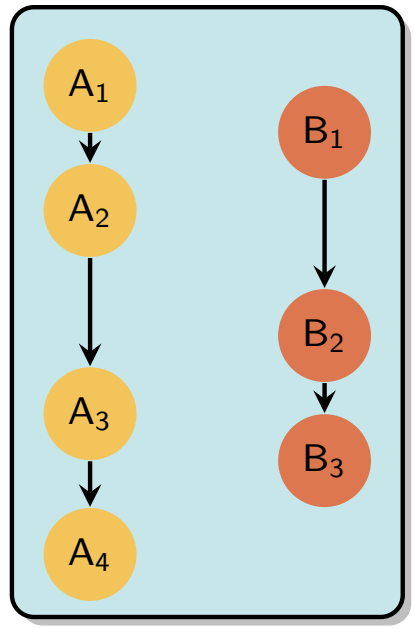
How About Multithreading?



How About Multithreading?

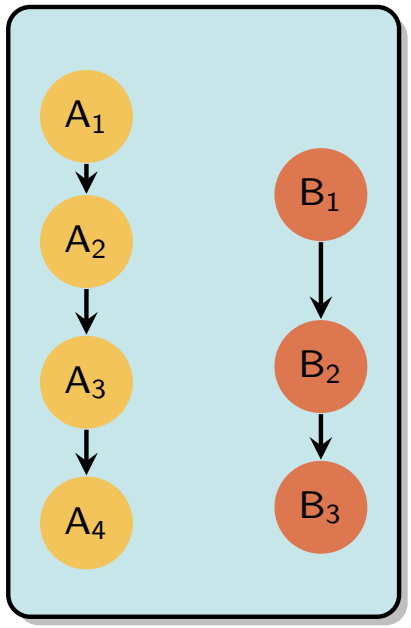


Replica 1

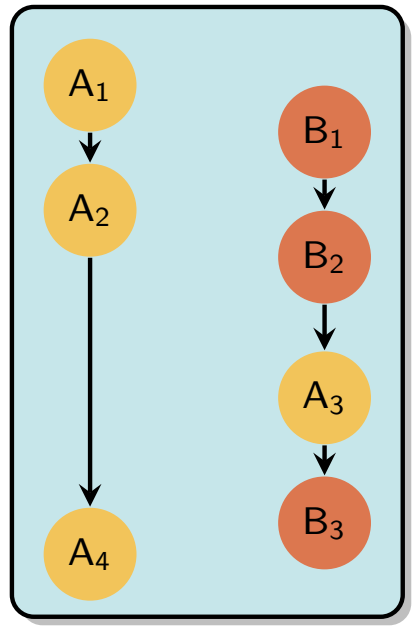


Replica 2

Problem: Nondeterminism



Replica 1



Replica 2

Solution: Deterministic Multithreading

- Related work: debugging multithreaded programs
- Compiler solutions [2]:
  No support for binary-only software

Solution: Deterministic Multithreading

- Related work: debugging multithreaded programs
- Compiler solutions [2]:
  No support for binary-only software
- Workspace-Consistent Memory [1]:
  Requires per-replica and per-thread memory copies

Solution: Deterministic Multithreading

- Related work: debugging multithreaded programs
- Compiler solutions [2]:
  No support for binary-only software
- Workspace-Consistent Memory [1]:
  Requires per-replica and per-thread memory copies
- Lock-Based Determinism
  - Reuse ideas from Kendo [12]

Solution: Deterministic Multithreading

- Related work: debugging multithreaded programs
- Compiler solutions [2]:
  No support for binary-only software
- Workspace-Consistent Memory [1]:
  Requires per-replica and per-thread memory copies
- Lock-Based Determinism
  - Reuse ideas from Kendo [12]
  - **Only for lock-based software!**

Enforced Determinism

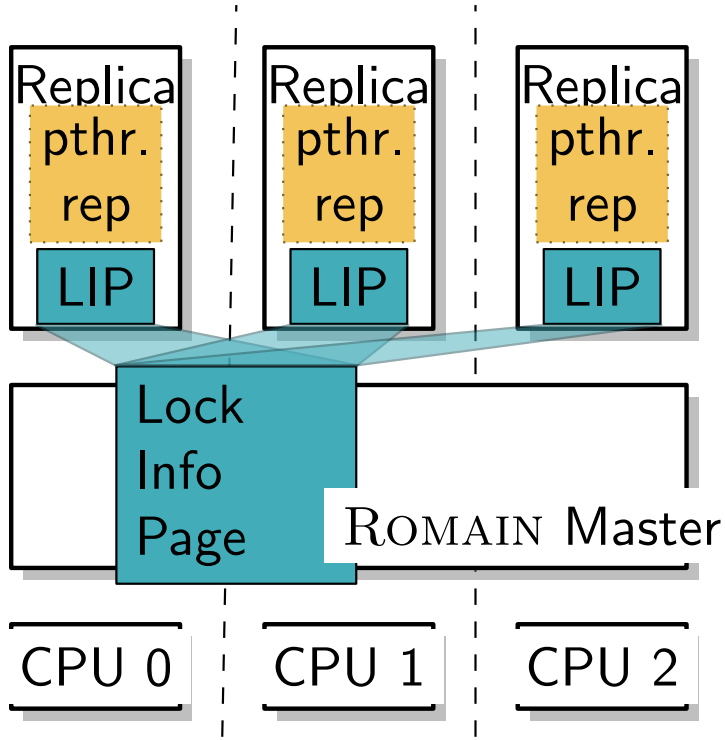
- Adapt `libpthread`: place `INT3` into four functions
  - `pthread_mutex_lock`
  - `pthread_mutex_unlock`
  - `__pthread_lock`
  - `__pthread_unlock`
- Lock operations reflected to Romain master
- Master enforces lock ordering

Enforced Determinism

- Adapt `libpthread`: place `INT3` into four functions
  - `pthread_mutex_lock`
  - `pthread_mutex_unlock`
  - `__pthread_lock`
  - `__pthread_unlock`
- Lock operations reflected to Romain master
- Master enforces lock ordering
- 300x overhead for worst-case microbenchmark in TMR!

Cooperative Determinism

- Replication-aware `libpthread`
- Replicas agree on acquisition order w/o master invocation
- Trade-off: `libpthread` becomes single point of failure

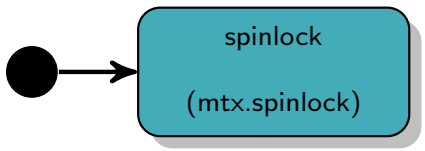


Cooperation: Lock Acquisition

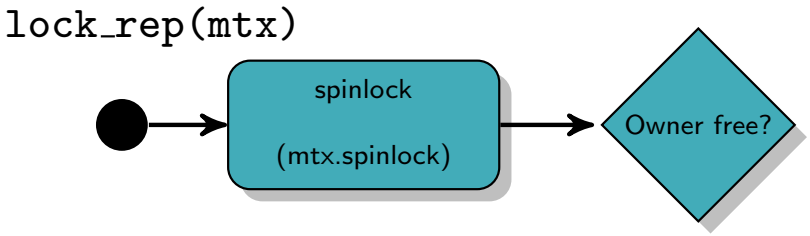
`lock_rep(mtx)`

Cooperation: Lock Acquisition

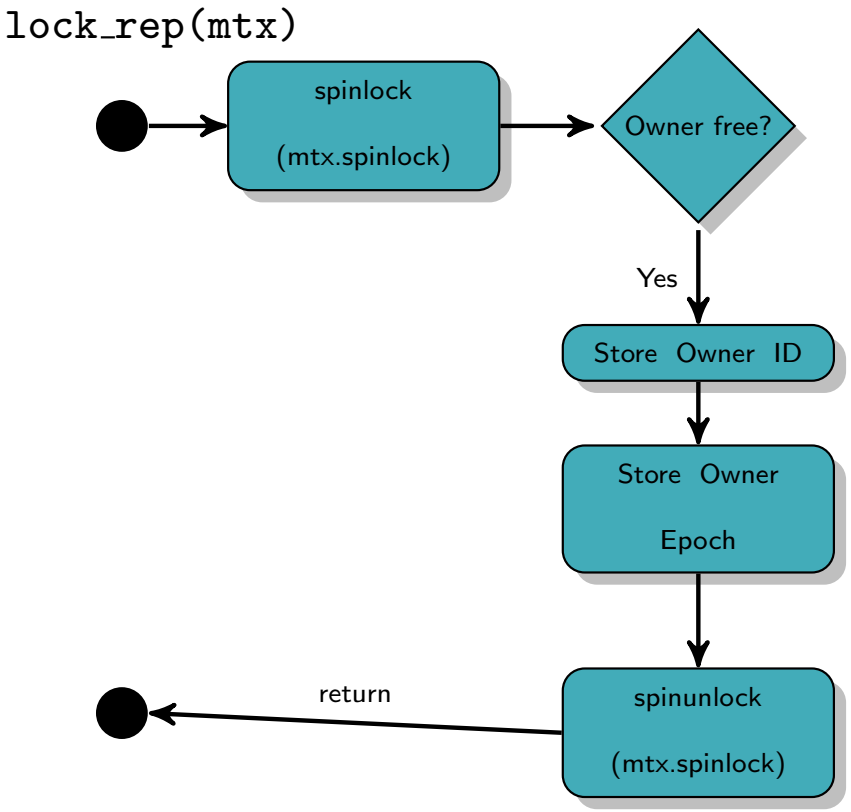
`lock_rep(mtx)`



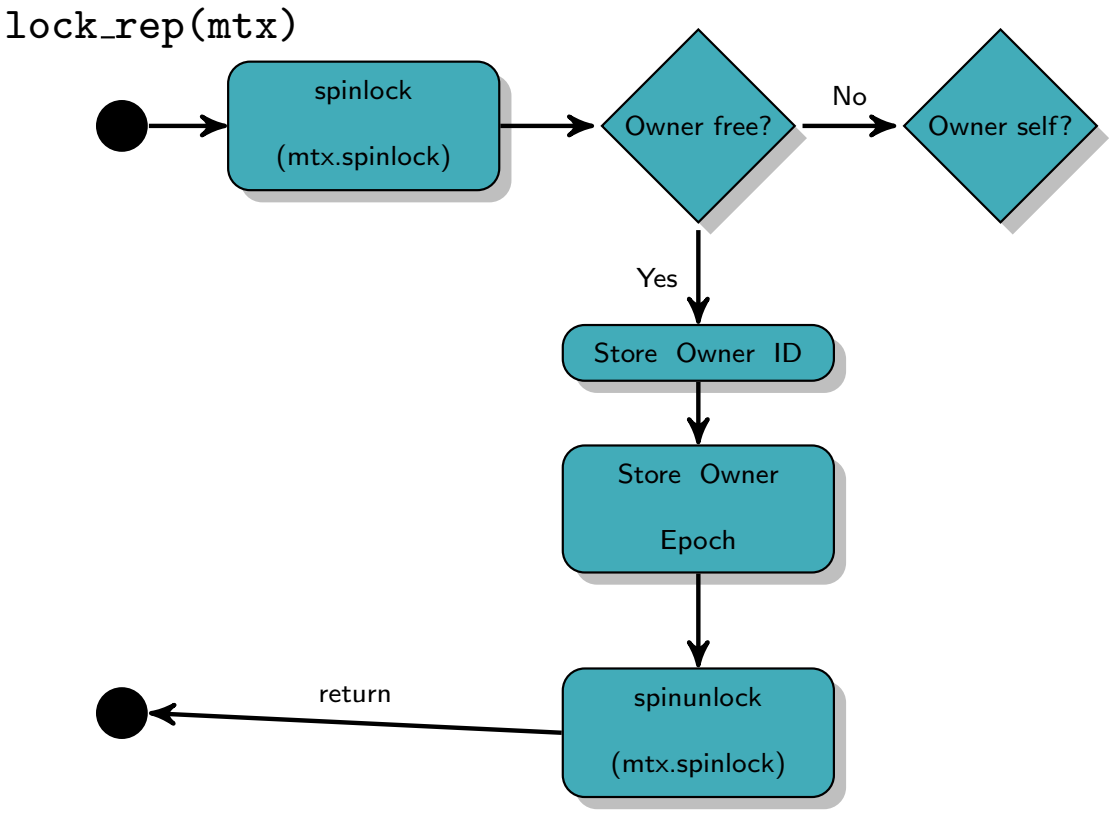
Cooperation: Lock Acquisition



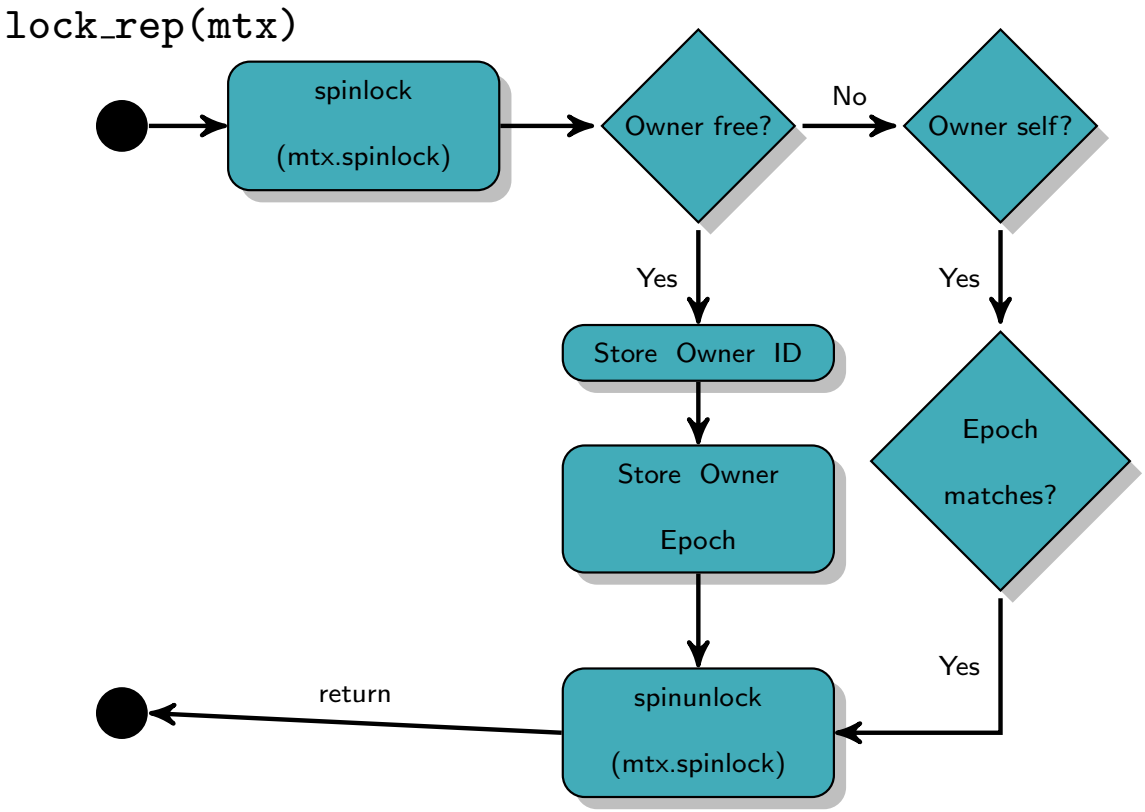
Cooperation: Lock Acquisition



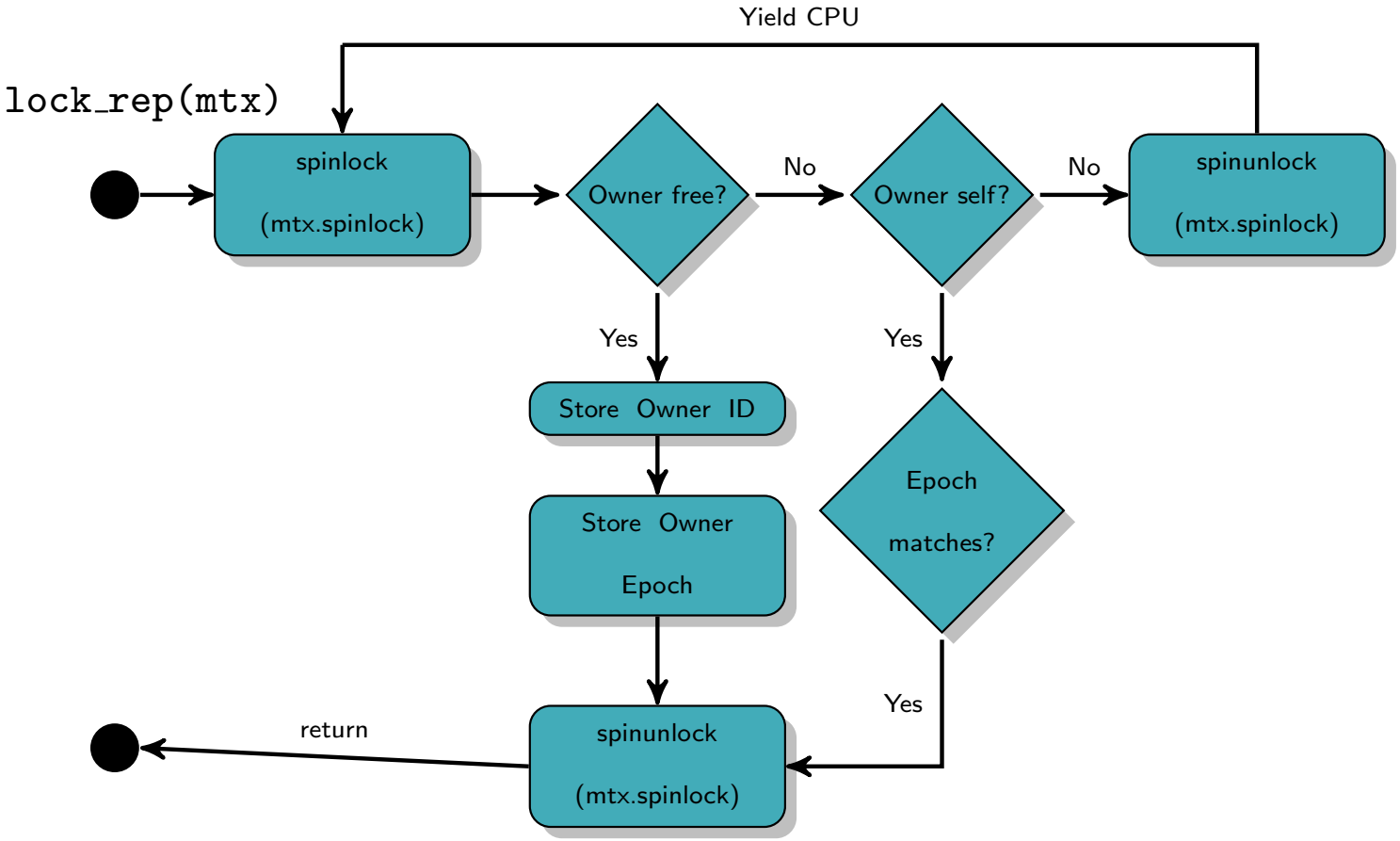
Cooperation: Lock Acquisition



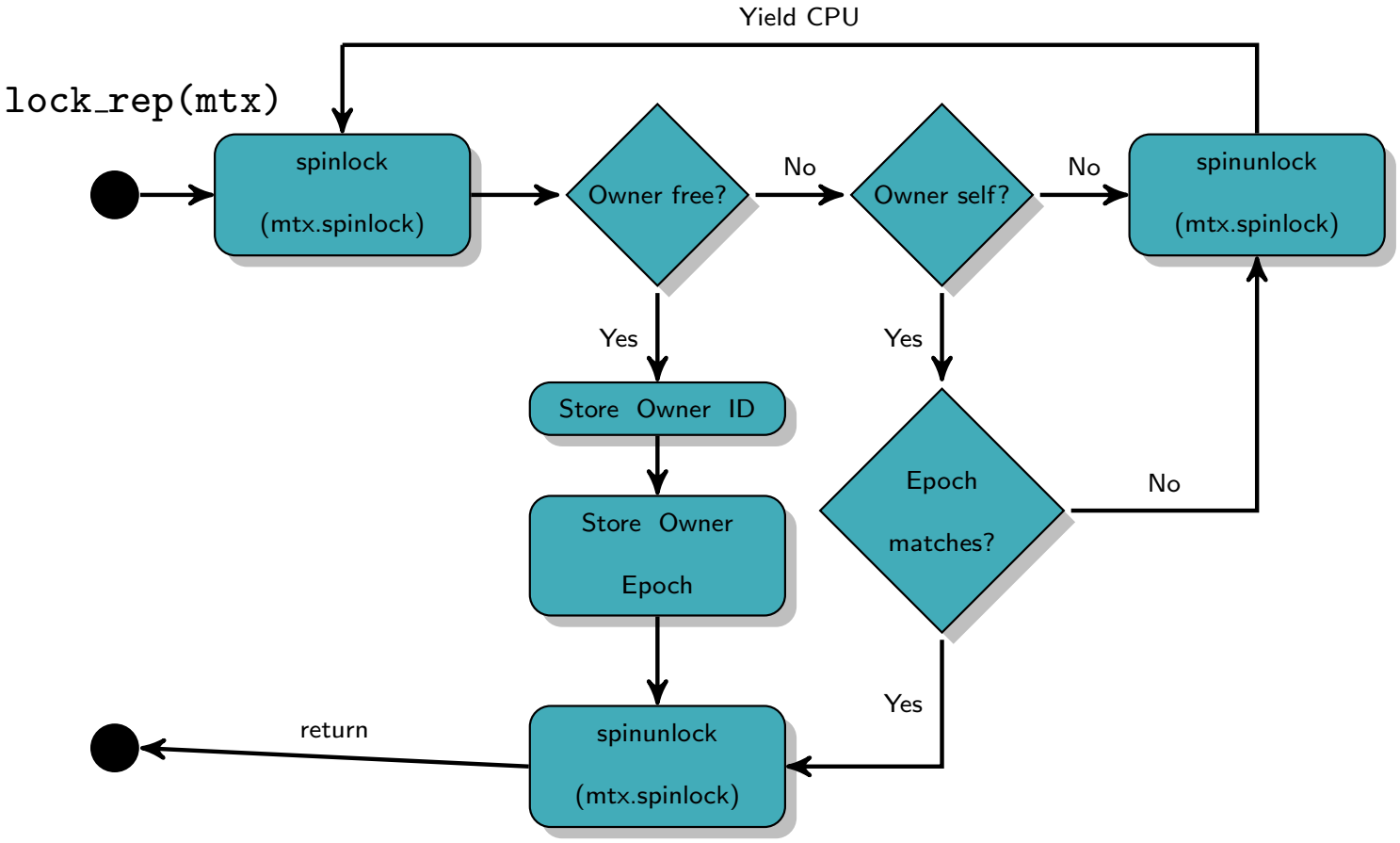
Cooperation: Lock Acquisition



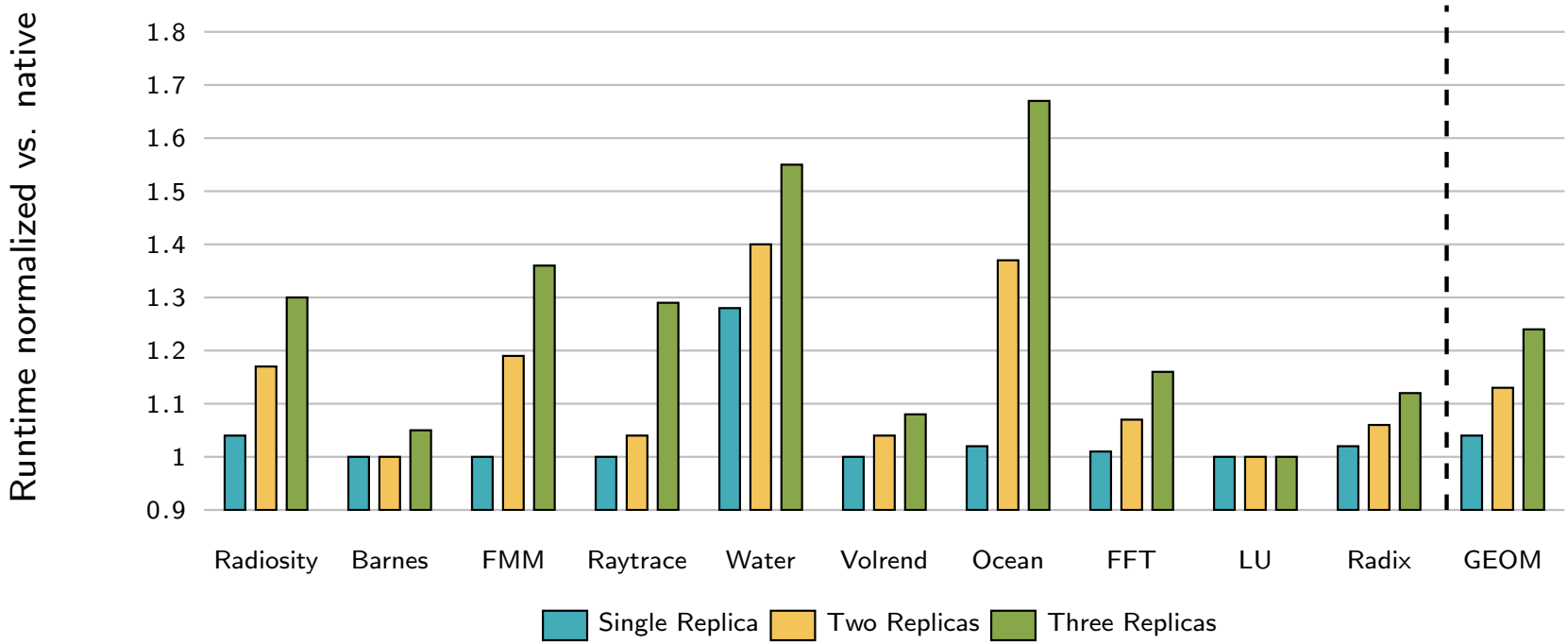
Cooperation: Lock Acquisition



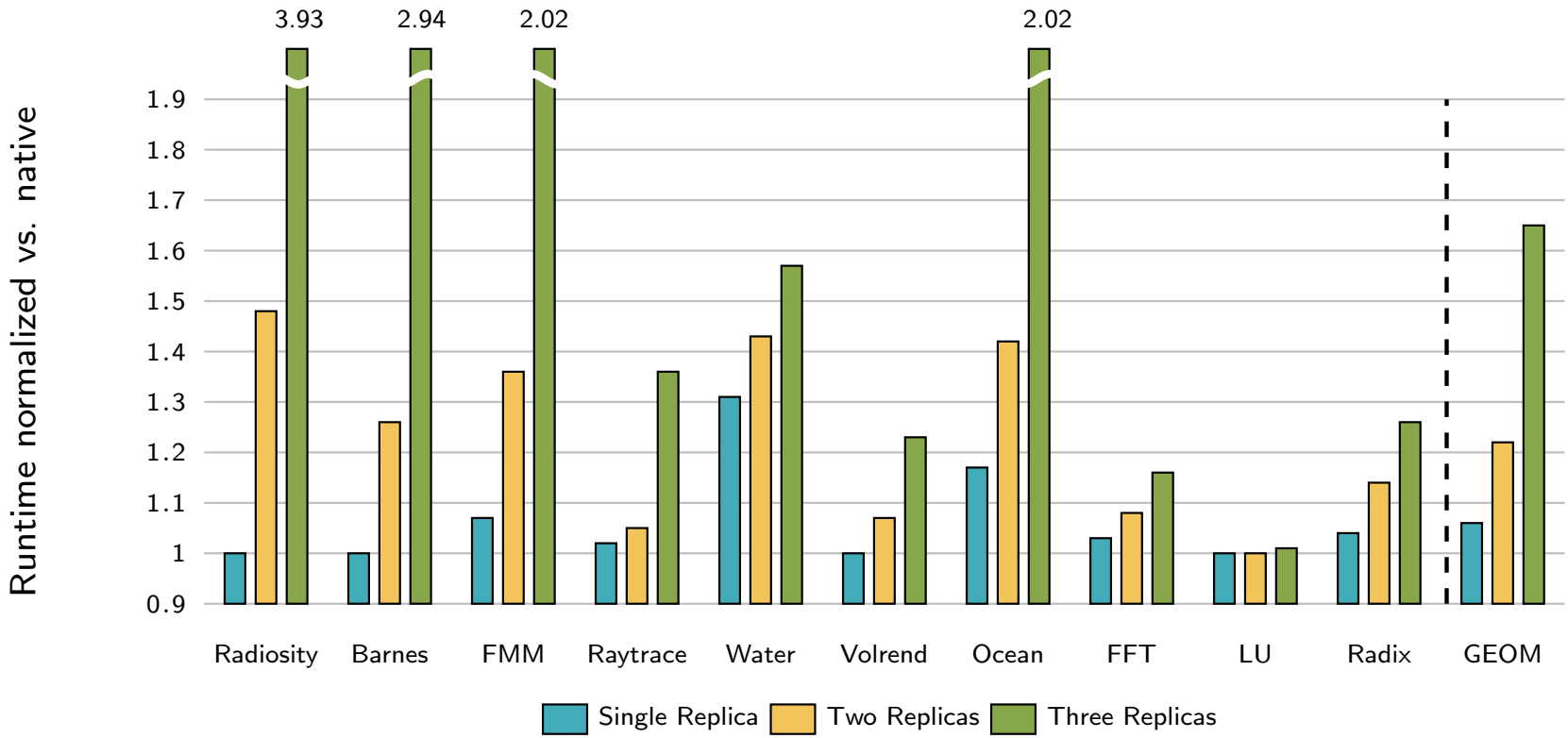
Cooperation: Lock Acquisition



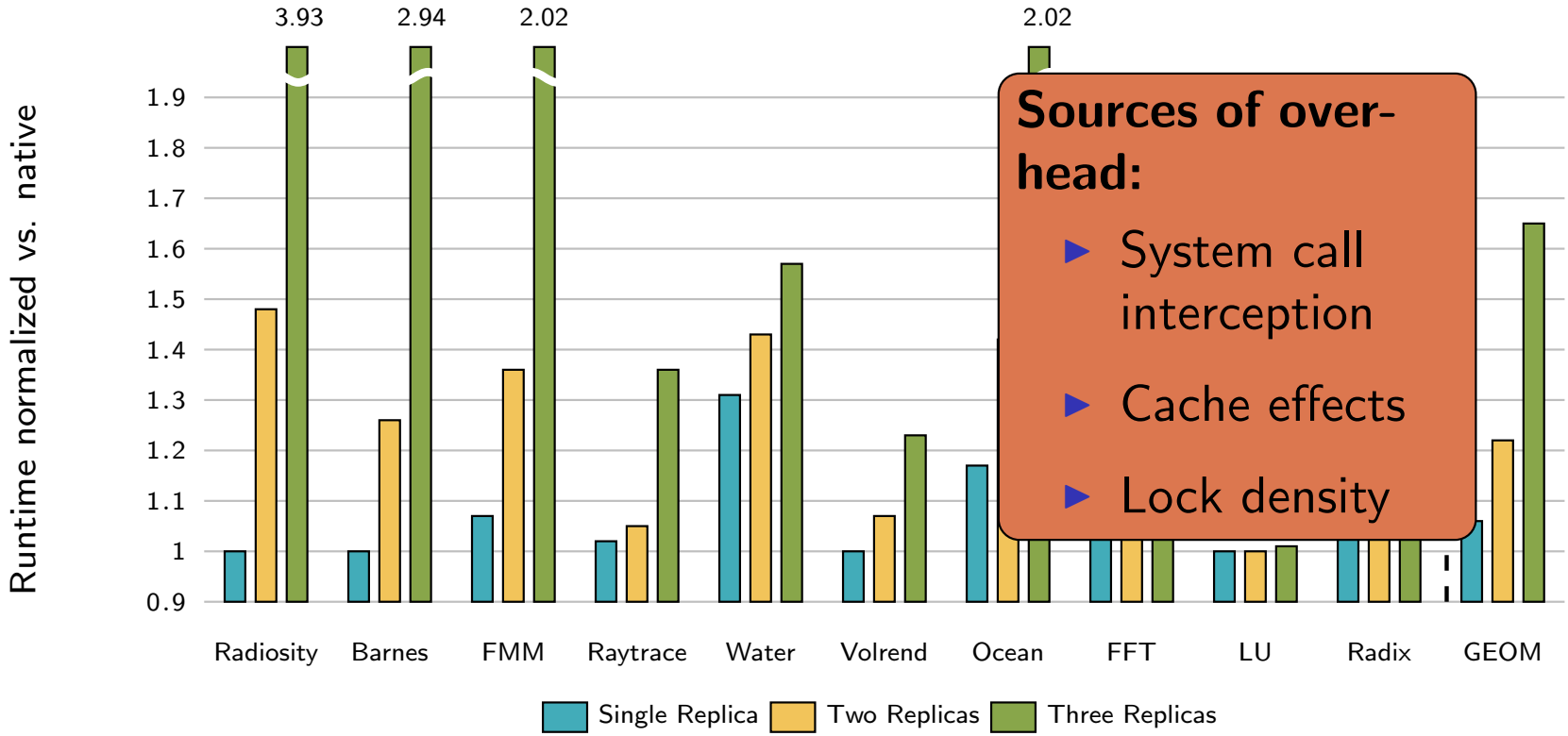
Overhead: SPLASH2, 2 workers [8]



Overhead: SPLASH2, 4 workers



Overhead: SPLASH2, 4 workers



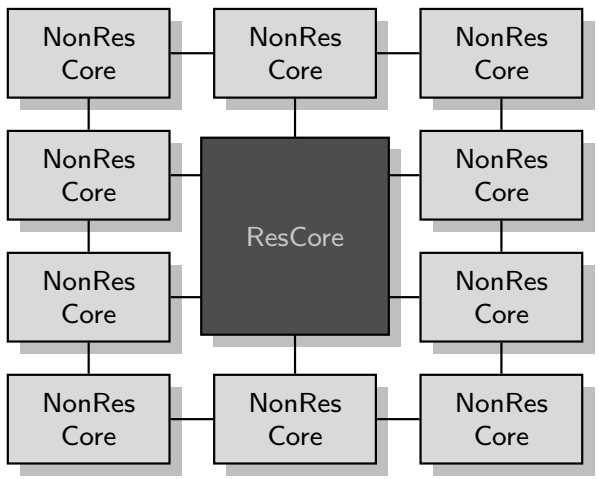
Sources of overhead:

- System call interception
- Cache effects
- Lock density

Hardening the RCB

- **We need:** Dedicated mechanisms to protect the RCB (HW or SW)
- **We have:** Full control over software
- Use FT-encoding compiler?
  - Has not been done for kernel code yet
- RAD-hardened hardware?
  - Too expensive

Why not split cores into resilient and non-resilient ones?



Summary

- OS-level techniques to tolerate SW and HW faults
- Address-space isolation
- Microreboots
- Various ways of handling session state
- Replication against hardware errors

Further Reading

- **Minix3**: Jorrit Herder, Ben Gras,, Philip Homburg, Andrew S. Tanenbaum: *Fault Isolation for Device Drivers*, DSN 2009
- **CuriOS**: Francis M. David, Ellick M. Chan, Jeffrey C. Carlyle and Roy H. Campbell *CuriOS: Improving Reliability through Operating System Structure*, OSDI 2008
- **L4ReAnimator**: Dirk Vogt, Björn Döbel, Adam Lackorzynski: *Stay strong, stay safe: Enhancing Reliability of a Secure Operating System*, IIDS 2010
- **seL4**: Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick and others *Formal verification of an OS kernel*, SOSP 2009
- **Romain**:
  - Björn Döbel, Hermann Härtig, Michael Engel: *Operating System Support for Redundant Multithreading*, EMSOFT 2012
  - Björn Döbel, Hermann Härtig: *Can We Put Concurrency Back Into Redundant Multithreading?*, EMSOFT 2014

Bibliography I


Amittai Aviram et al. "Efficient system-enforced deterministic parallelism". In: *OSDI*. 2012, pp. 111–119.


Tom Bergan et al. "CoreDet: a compiler and runtime system for deterministic multithreaded execution". In: *ACM SIGARCH Computer Architecture News*. 2010, pp. 53–64.


Andy Chou et al. "An empirical study of operating systems errors". In: *SOSP*. 2001, pp. 73–88.


Francis M David et al. "CuriOS: Improving Reliability through Operating System Structure." In: *OSDI*. 2008, pp. 59–72.


Anand Dixit and Alan Wood. "The impact of new technology on soft error rates". In: *International Reliability Physics Symposium (IRPS)*. 2011, 5B–4.


Björn Döbel and Hermann Härtig. "Can we put concurrency back into redundant multithreading?" In: *EMSOFT*. 2014, pp. 1–10.

Bibliography II


Björn Döbel, Hermann Härtig, and Michael Engel. "Operating system support for redundant multithreading". In: *EMSOFT*. 2012, pp. 83–92.


Björn Döbel. "Operating System Support for Redundant Multithreading". Dissertation. TU Dresden, 2014.


Jorrit N Herder et al. "Fault isolation for device drivers". In: *DSN*. 2009, pp. 33–42.


Andy A Hwang, Ioan A Stefanovici, and Bianca Schroeder. "Cosmic rays don't strike twice". In: *ASPLOS*. 2012, pp. 111–122.


Gerwin Klein et al. "seL4: Formal verification of an OS kernel". In: *SOSP*. 2009, pp. 207–220.


Marek Olszewski, Jason Ansel, and Saman Amarasinghe. "Kendo: efficient deterministic multithreading in software". In: *ASPLOS*. ACM, 2009, pp. 97–108.


Nicolas Palix et al. "Faults in Linux: Ten years later". In: *ASPLOS*. 2011, pp. 305–318.

Bibliography III

Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. "DRAM errors in the wild: a large-scale field study". In: *SIGMETRICS/Performance*. 2009, pp. 193–204.

Dirk Vogt, Björn Döbel, and Adam Lackorzynski. "Stay strong, stay safe: Enhancing reliability of a secure operating system". In: *Workshop on Isolation and Integration for Dependable Systems*. 2010, pp. 1–10.