



User-level Inter-Process Communication

This week:



Milestone:

- Sending messages between cores

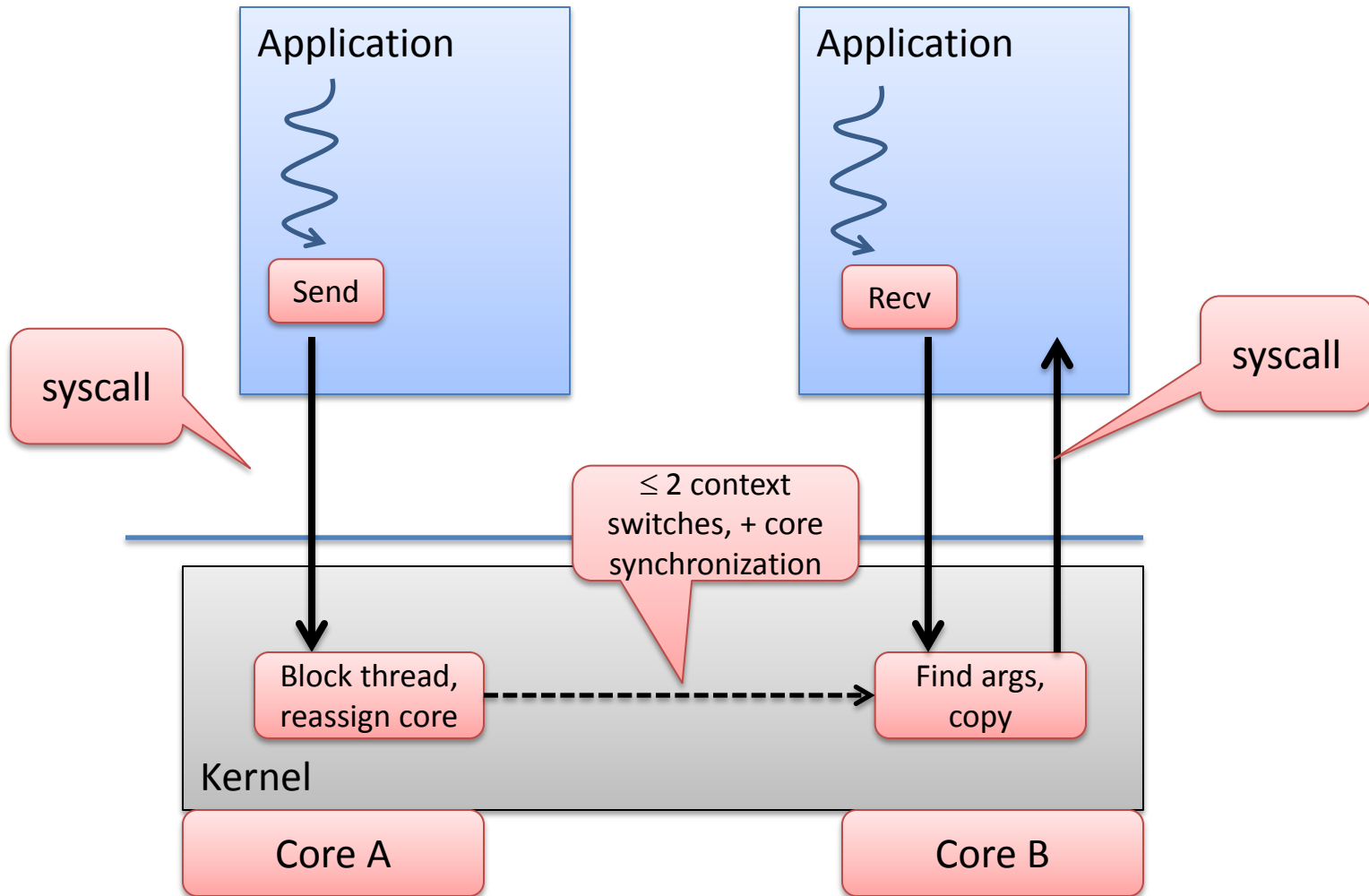
Lecture:

- User-level Remote Procedure Call
- Cache coherence protocols
- User-level messages in Barreelfish
- Dealing with the ARM memory model
- Connection setup



USER-LEVEL REMOTE PROCEDURE CALL

IPC on a multiprocessor



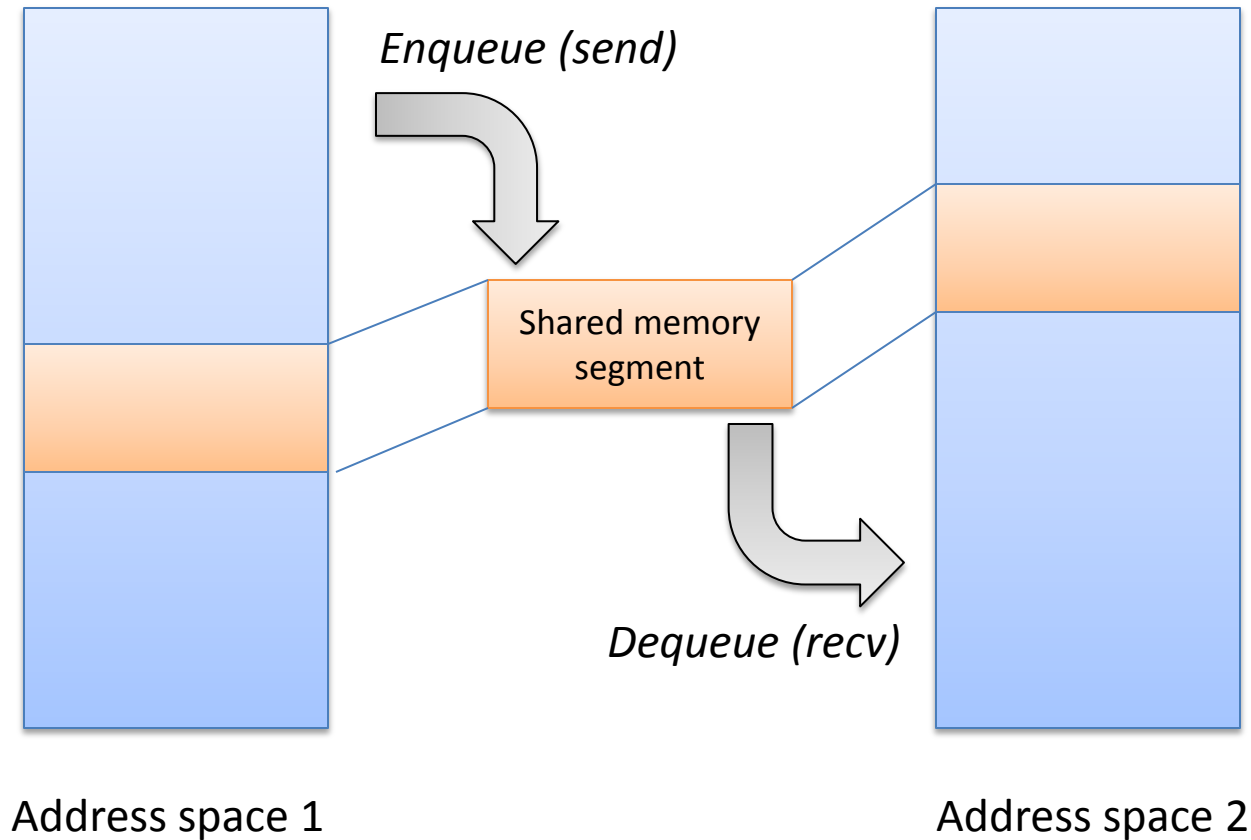
User-level RPC

(Bershad, Anderson, Lazowska, Levy, 1991)



- Arguably, URPC is to Scheduler Activations what LRPC was to kernel threads
 - Send messages between address spaces directly
no kernel involved!
 - Eliminate unnecessary processor reallocation
 - Amortize processor reallocation over several calls
 - Exploit inherent parallelism in send / receiving
- Decouple:
 - notification (user-space)
 - scheduling (kernel)
 - data transfer (also user space).

User-space producer-consumer queue



Not a new idea: c.f. hardware devices



Programmer's Guide

01/29/08

BCM57XX

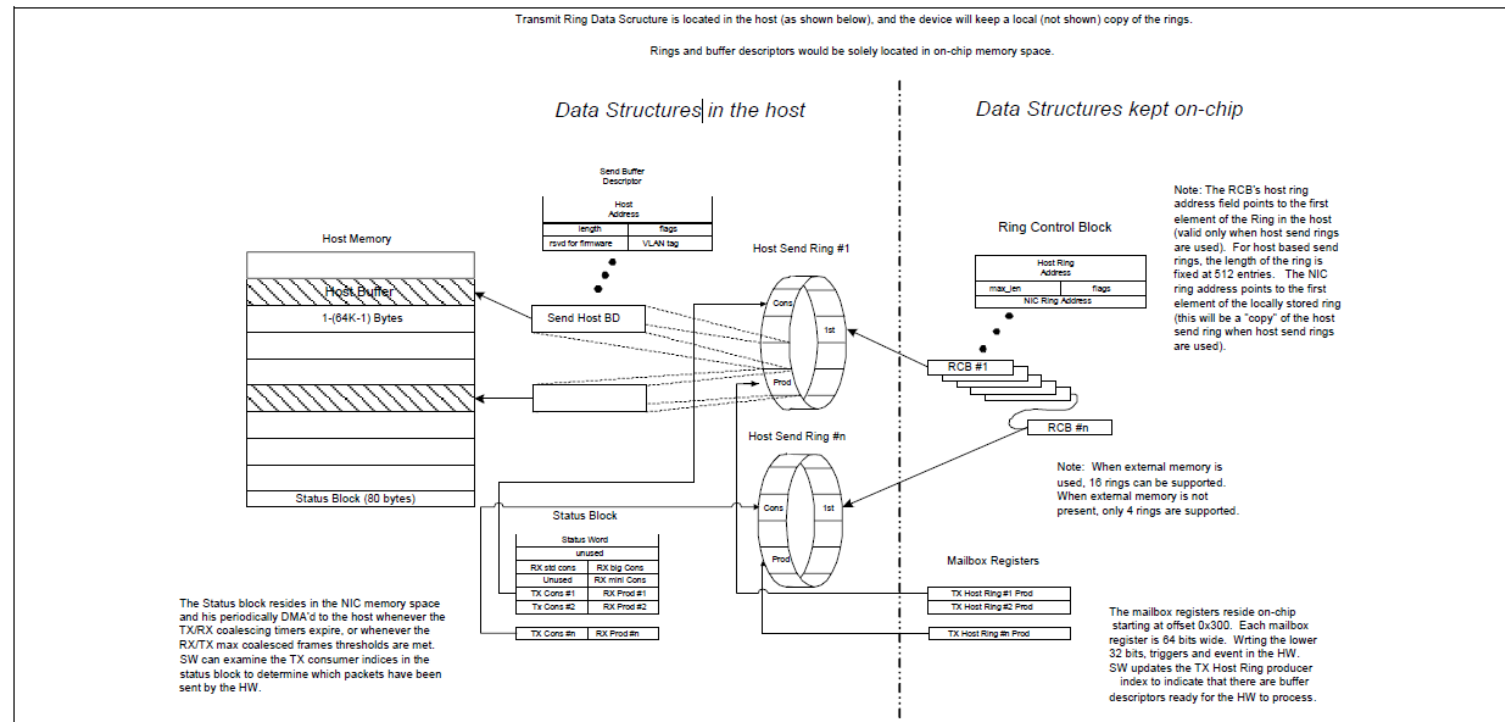
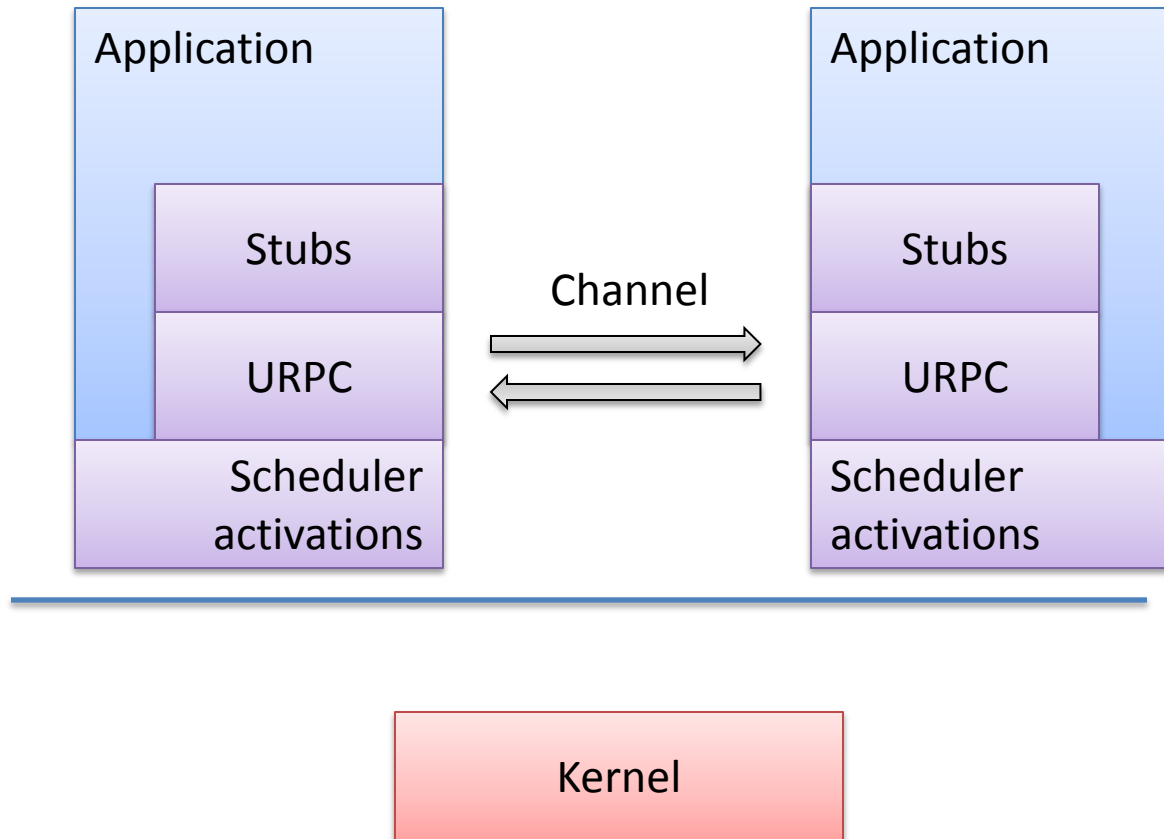
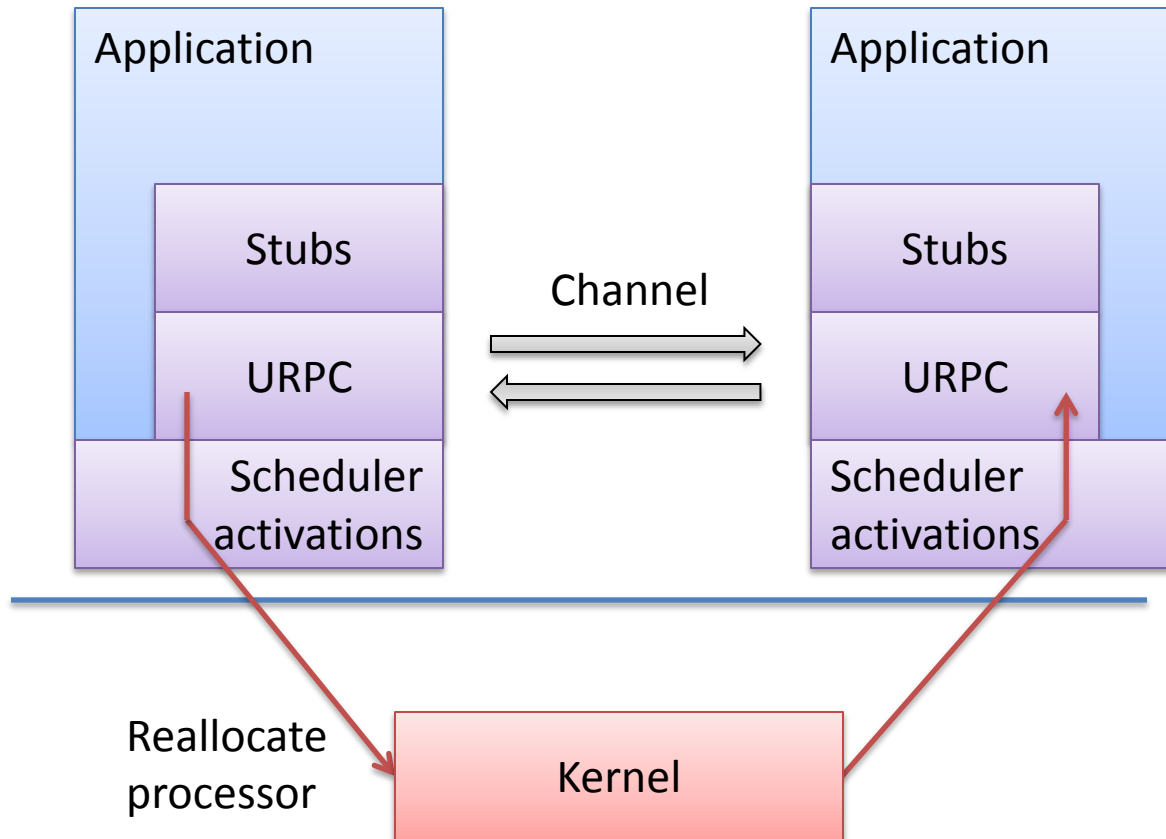


Figure 44: Transmit Ring Data Structure Architecture Diagram

URPC operation



URPC operation

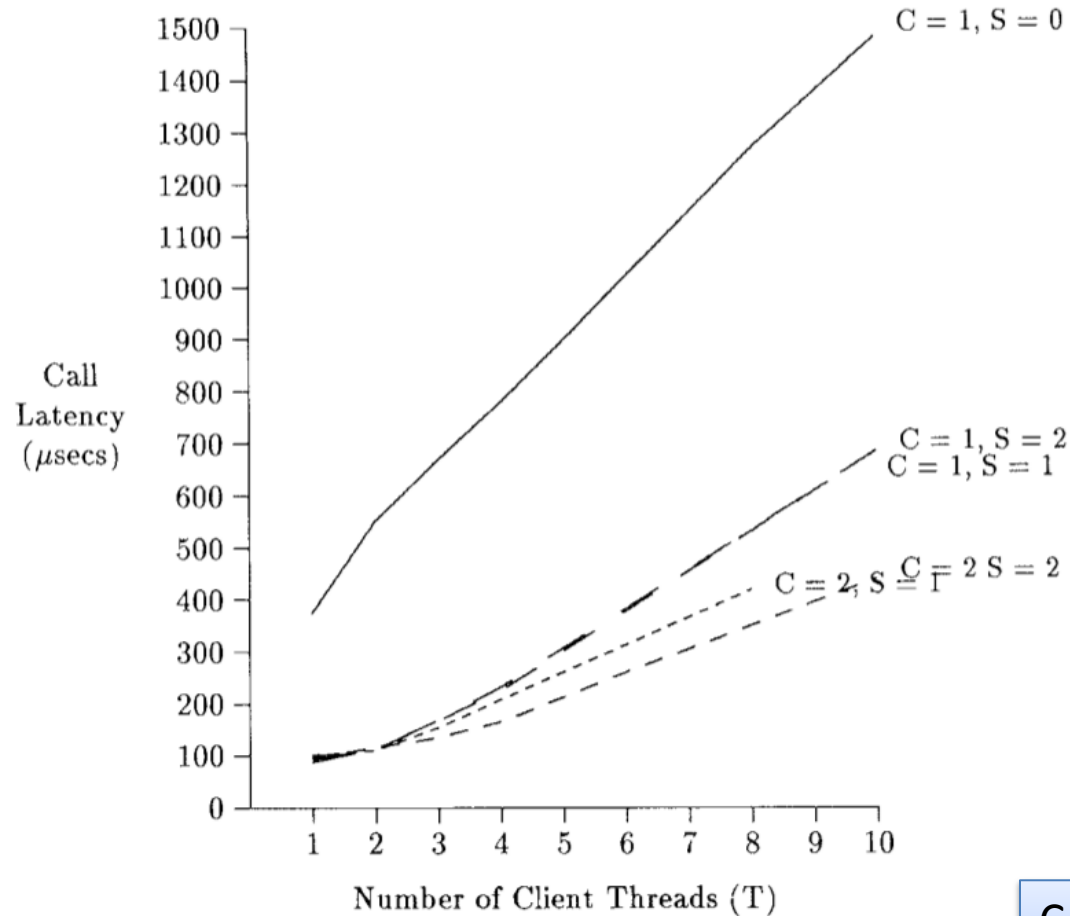


How is the kernel not involved?



- **Shared memory** channels, mapped pairwise between domains
- **Queues** with non-spinning TAS locks at each end
- Threads block on channels entirely in **user space**
- Messaging is **asynchronous** below thread abstractions
- Can switch to another thread in same address space
 - rather than block waiting for another address space
- Big win:
Multiprocessor with concurrent client and server threads

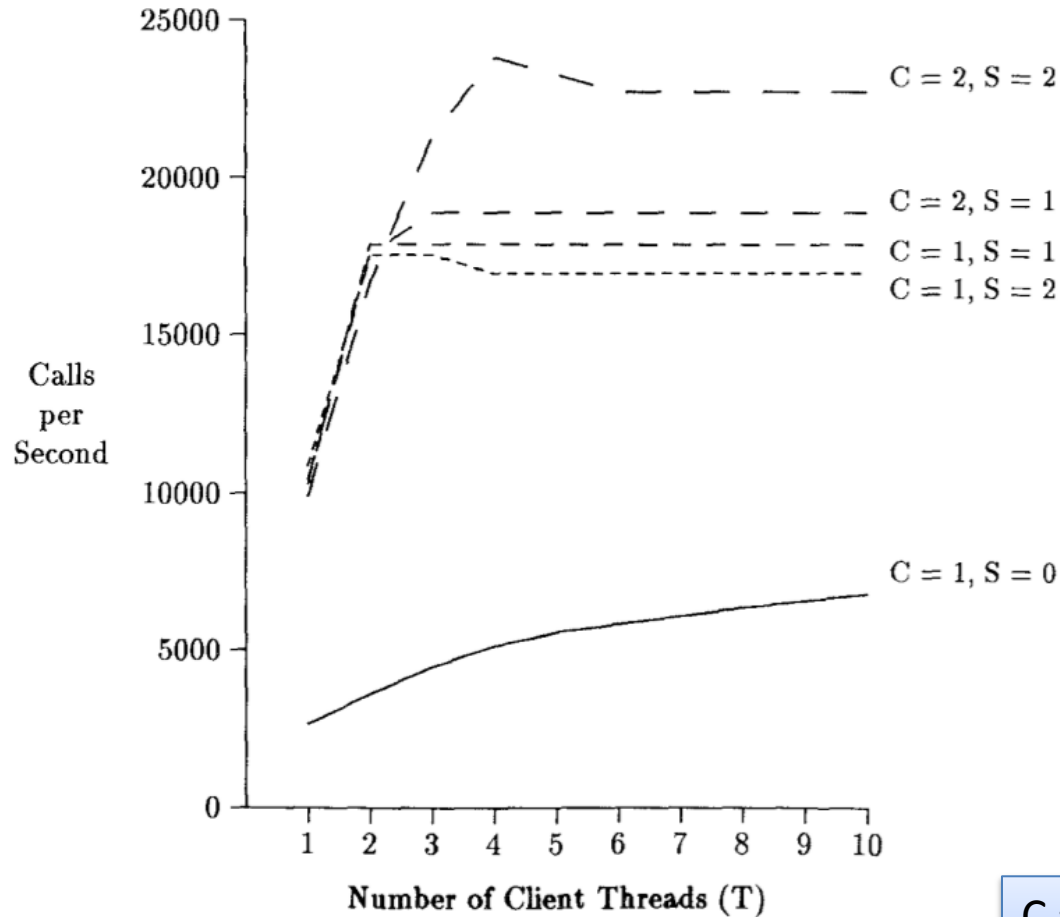
URPC latency



User-level IPC

C = #cores for clients
S = #cores for server

URPC throughput



User-level IPC

C = #cores for clients
 S = #cores for server

URPC performance



Mechanism	Operation	Performance
URPC (w/ fast threads)	Cross-AS latency	93 μ s
	Inter-processor overhead	53 μ s
	Thread fork	43 μ s
LRPC	Latency	157 μ s
	Thread fork	> 1000 μ s
Hardware	Procedure call	7 μ s
	Kernel trap	20 μ s

All on a Firefly (4-processor CVAX). Irony:

- LRPC, L4 seek performance by optimizing kernel path
- URPC gains performance by bypassing kernel entirely

Discussion



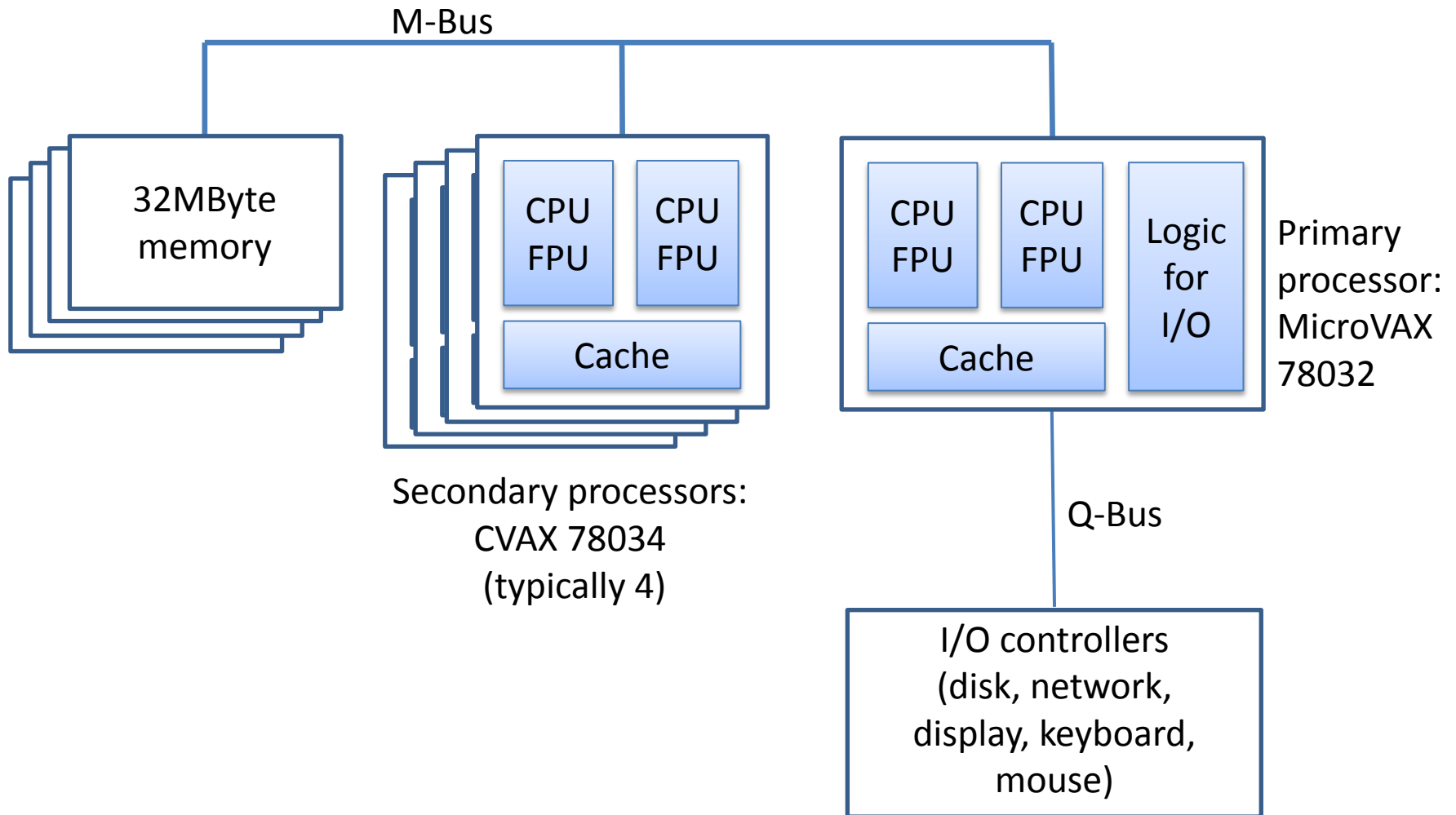
- L4, LRPC:
 - Optimize for synchronous, null RPC performance bypass scheduler
 - Hard to perform accurate resource accounting
- URPC:
 - Integrate with the scheduler
 - Decouple from event transmission
 - ⇒ slightly slower null RPC times when idle
 - ⇒ higher RPC throughput
 - ⇒ lower latency on multiprocessors

Discussion

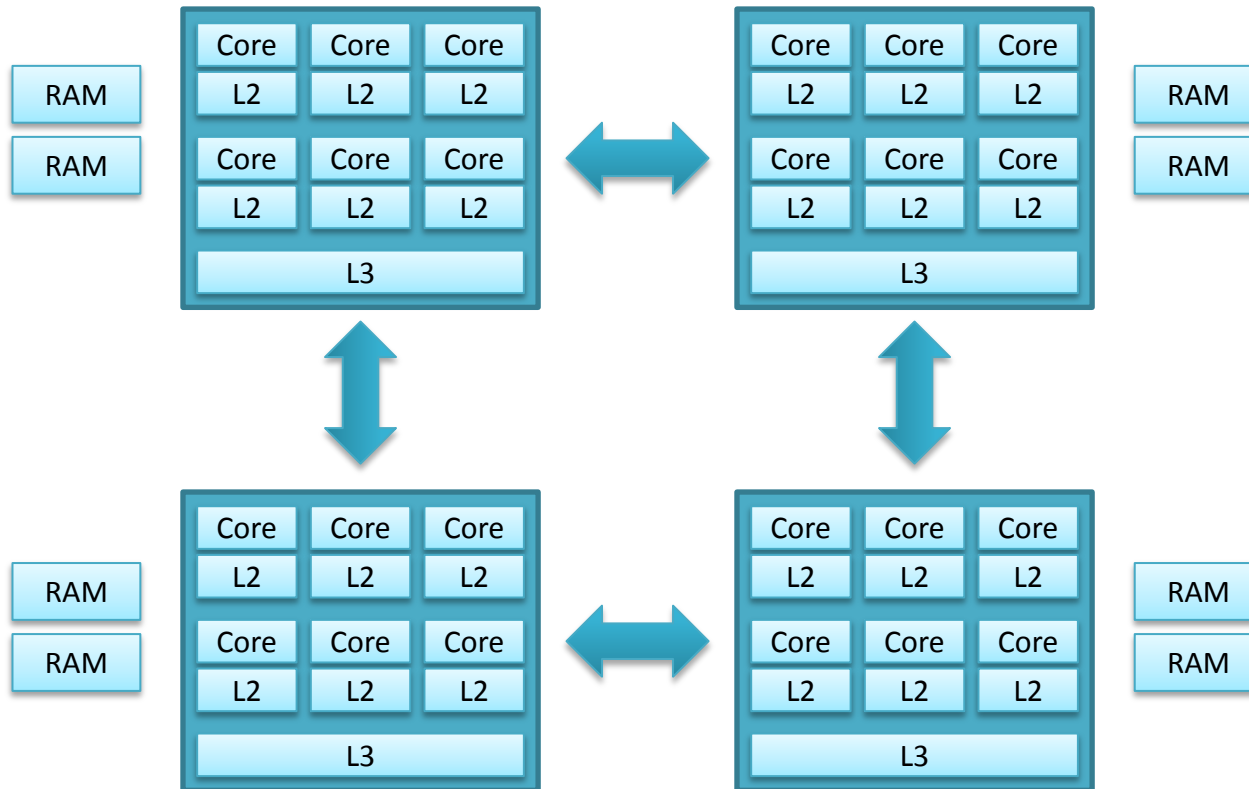


- URPC is *polled* (at least initially)
 - Don't want to block (\Rightarrow enter the kernel)
 - Works for a *fully loaded* multiprocessor
- Queue has *spinlocks* on both ends
 - Threads can be pre-empted and migrated
 - Multiple senders/receivers are possible?
- Data is *copied* into memory buffer
 - Firefly had very small caches
 - CPU/memory gap was small

Firefly c. 1991



Cache-coherent multicore c. 2011



AMD Istanbul: 6 cores, per-core L2, per-package L3

URPC today



- Machines are different:
 - NUMA, not SMP
 - Locking is expensive (remote RMW cycle)
 - Caches are large and fast, memory is slow
 - Many cores
- Implications:
 - Don't use locks
 - Spinning might be cheap
 - **Cache-cache** transfers if possible (avoid DRAM)
 - Critical to understand the coherency protocol



MESI CACHE COHERENCE

MESI protocol



- Four-state coherency protocol:
 - Modified**: This is the only copy, it's dirty
 - Exclusive**: This is the only copy, it's clean
 - Shared**: This is one of several copies, all clean
 - Invalid**
- Extra bus signal: **RdX**
 - “Read exclusive”
 - Cache can load into either “shared” or “exclusive” states
 - Other caches can see the type of read
- Also: **HIT** signal
 - Signals to a remote processor that its read hit in local cache.

MESI invariants



- Allowed combination of states for a line between any pair of caches:

	M	E	S	I
M				✓
E				✓
S			✓	✓
I	✓	✓	✓	✓

- Protocol must preserve these invariants

MSI invariants:


	M	S	I
M			✓
S		✓	✓
I	✓	✓	✓

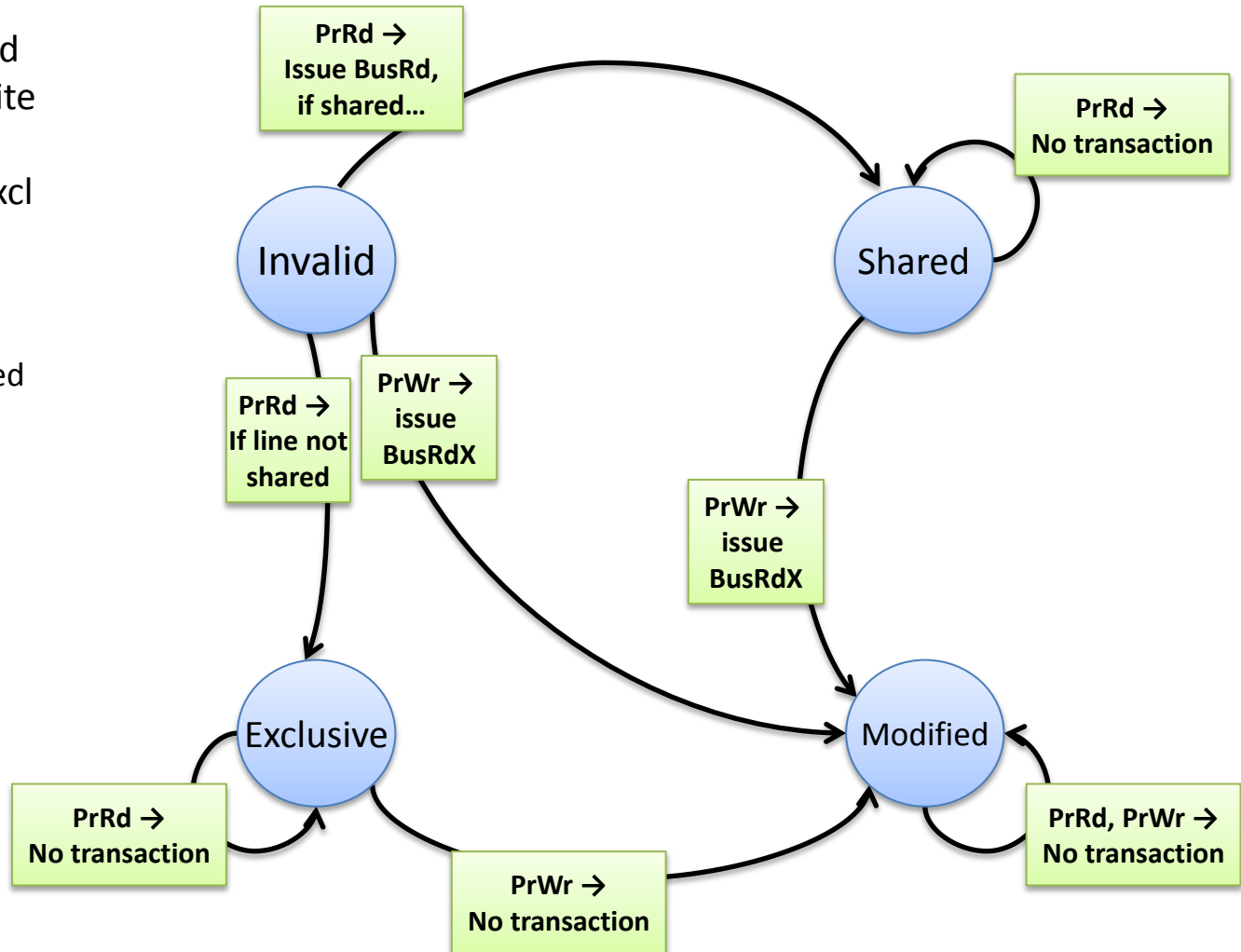
MESI state machine



Terminology:

- PrRd: processor read
- PrWr: processor write
- BusRd: bus read
- BusRdX: bus read excl
- BusWr: bus write

 Processor-initiated




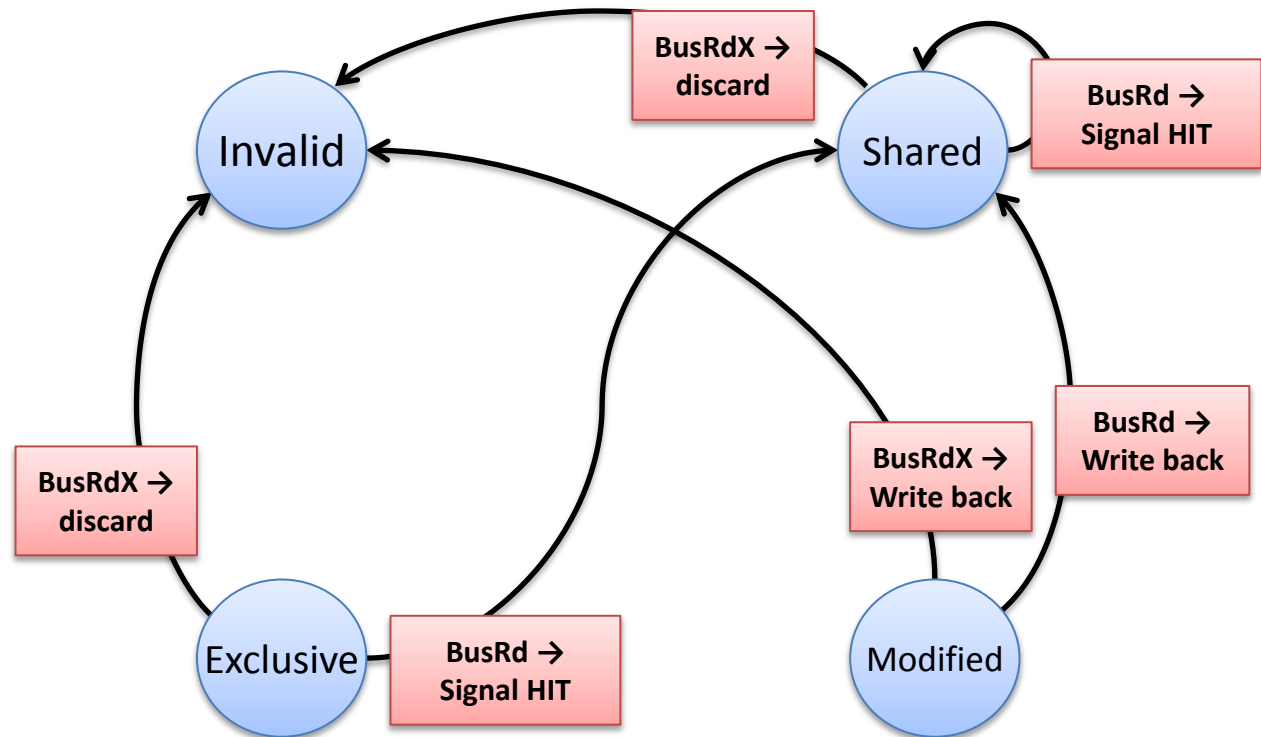
MESI state machine



Terminology:

- PrRd: processor read
- PrWr: processor write
- BusRd: bus read
- BusRdX: bus read excl
- BusWr: bus write

 Snoop-initiated

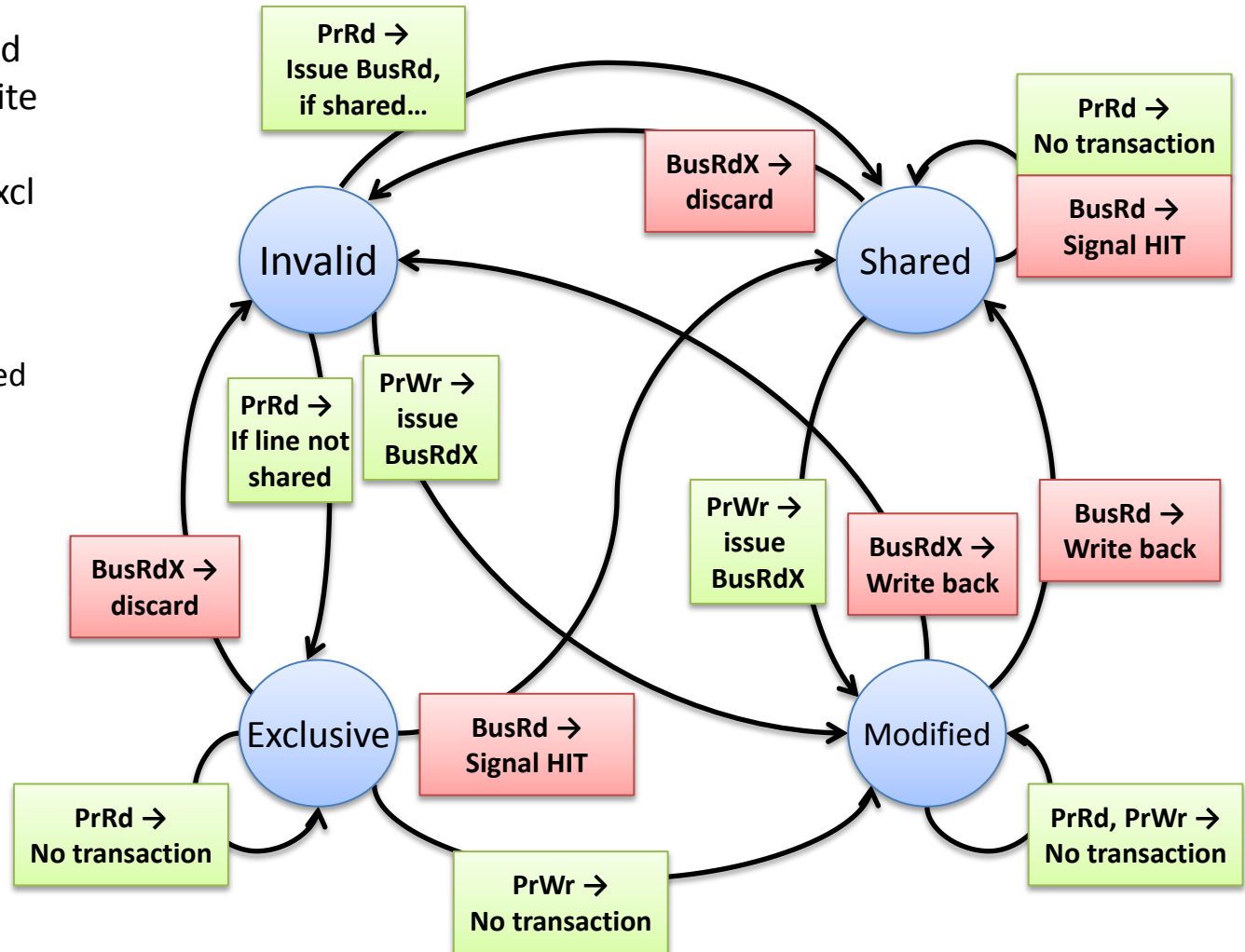
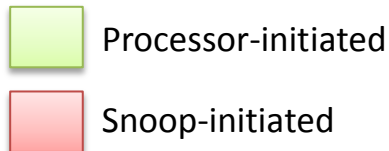


MESI state machine



Terminology:

- PrRd: processor read
- PrWr: processor write
- BusRd: bus read
- BusRdX: bus read excl
- BusWr: bus write



MESI observations



- Dirty data always written through memory
 - No cache-cache transfers
 - “Invalidation-based” protocol
- Data is always either:
 1. Dirty in one cache
 - \Rightarrow must be written back before a remote read
 2. Clean
 - \Rightarrow can be safely fetched from memory

Good if:

latency of memory \ll latency of remote cache



MOESI AND MESIF

MOESI protocol (AMD)



Add new “Owner” state: allow line to be modified, but other dirty copies to exist in other caches.

Modified:

No other cached copies exist, local copy dirty

Owner:

Multiple dirty copies exist (all consistent).

This copy has sole right to modify line.

Exclusive:

No other cached copies exist, local copy clean

Shared:

Other cached copies exist (all consistent).

One other copy might be able to write (state Owner)

Invalid:

Not in cache.

MOESI invariants



- Can quickly satisfy read request for dirty cache line without writeback to memory
 - Owner cache must respond with line.
- Read requests for clean, shared line must be served by memory
- Good if latency of remote cache < latency of main memory

	M	O	E	S	I
M					✓
O				✓	✓
E					✓
S		✓		✓	✓
I	✓	✓	✓	✓	✓

MESIF protocol (Intel)



Add new “Forward” state: designates at-most-one Shared line to serve remote requests.

Modified:

No other cached copies exist, local copy dirty

Exclusive:

No other cached copies exist, local copy clean

Shared:

Other cached copies exist, local copy clean

One other copy might be dirty (state Owner)

Invalid:

Not in cache.

Forward:

As Shared, but this is the *designated responder* for requests

MESIF invariants



- At most one copy is in Forward state.
 - Most recent cache to request line
 - Avoids incast storm if line is widely shared
- If none, request served by main memory
 - Even if shared copies exist

	M	E	S	I	F
M				✓	
E				✓	
S			✓	✓	✓
I	✓	✓	✓	✓	✓
F			✓	✓	

What does ARM do?



- It's complicated...
- ACE protocol:
 - Standardized by ARM Ltd.
 - Essentially MOESI, with some flexibility
 - Rarely implemented by SoC vendors ☹️
 - Extremely weak memory model ☹️
- Good news:
 - Assume MOESI, be **careful with your barriers**, and your code should work 😊



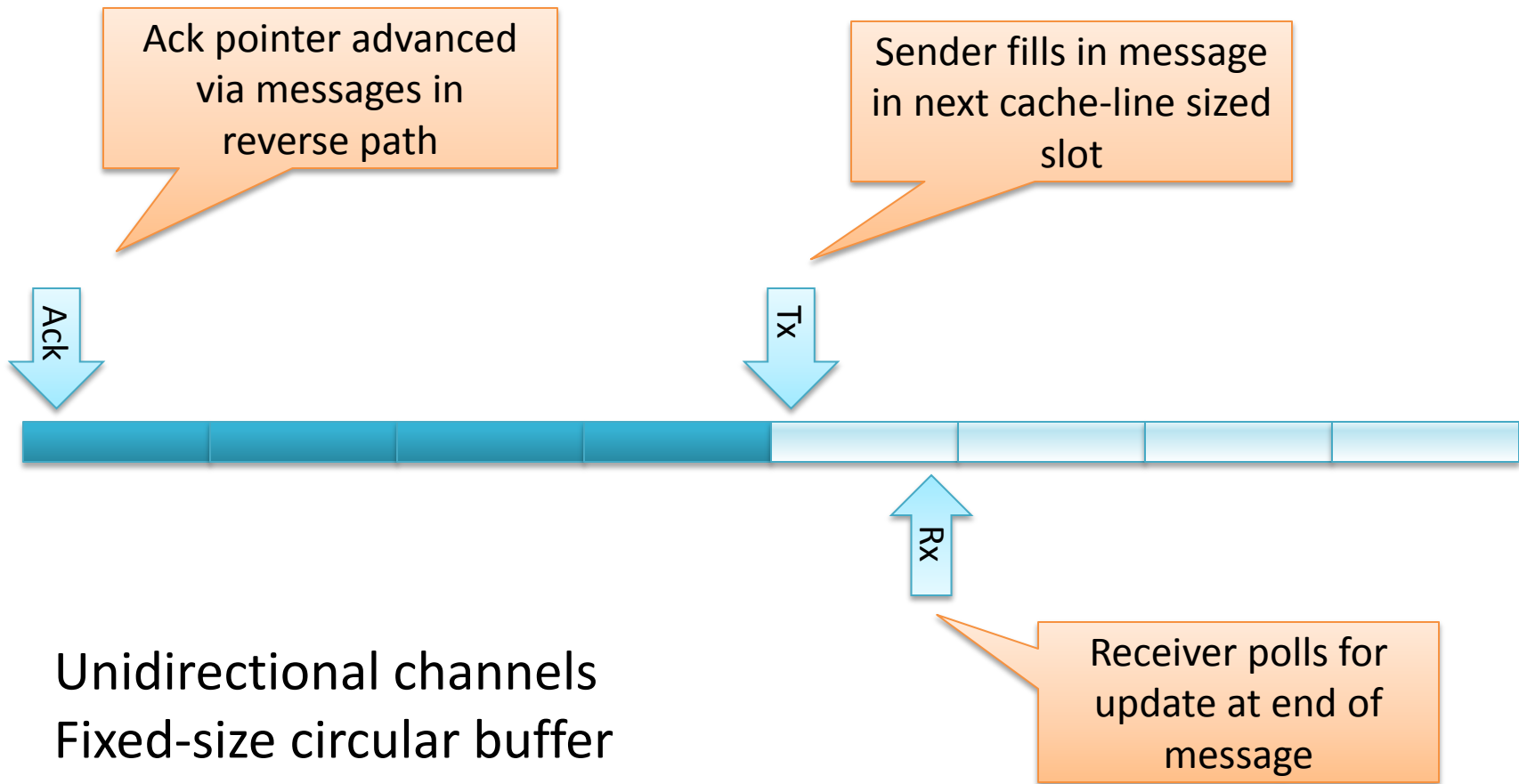
INTERPROCESS COMMUNICATION IN BARRELFISH

CC-UMP Interconnect Driver



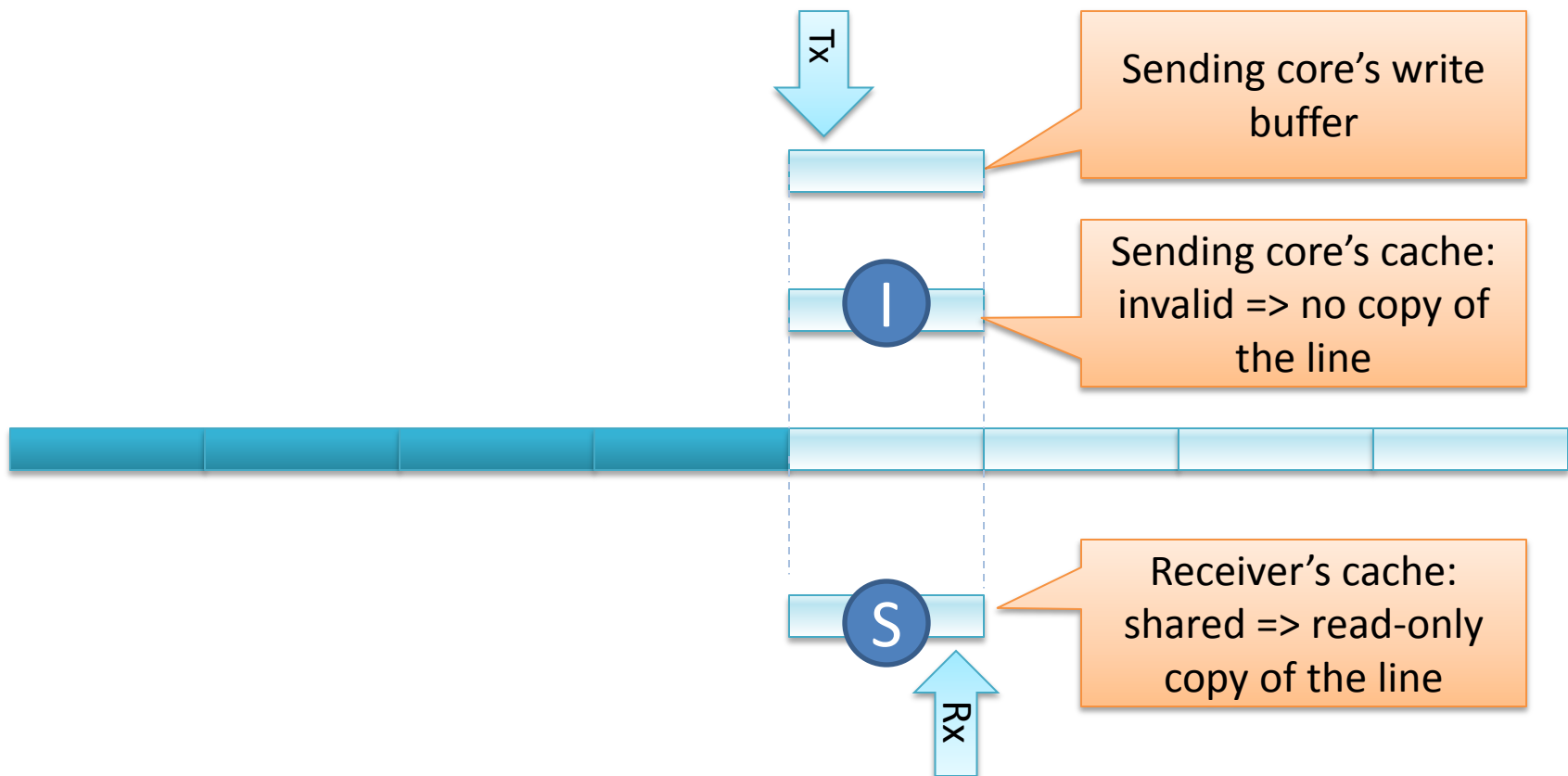
- Used for *cache-coherent shared* memory
 - inspired by URPC
- Ring buffer of *cache-line sized* messages
 - 64 bytes or 32 bytes
 - 1 word used for bookkeeping;
last one written (end of line)
- **Credit-based flow control** out of band
- **One** channel per IPC binding (not shared)

CC-UMP: cache-coherent user-space messaging



Unidirectional channels
Fixed-size circular buffer
All messages are 64-byte cache lines

CC-UMP: cache-coherent user-space messaging

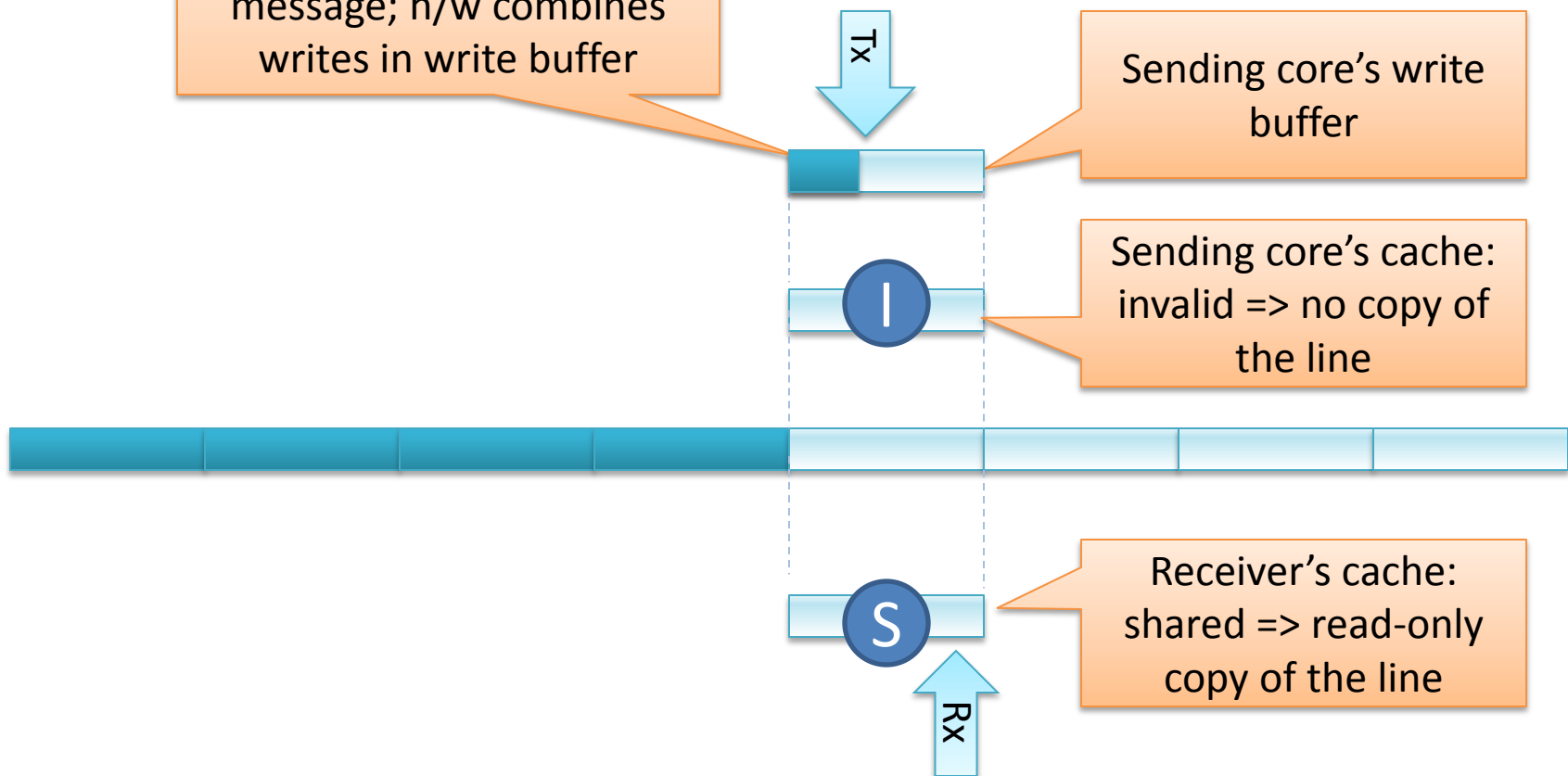


CC-UMP: cache-coherent user-space messaging



1

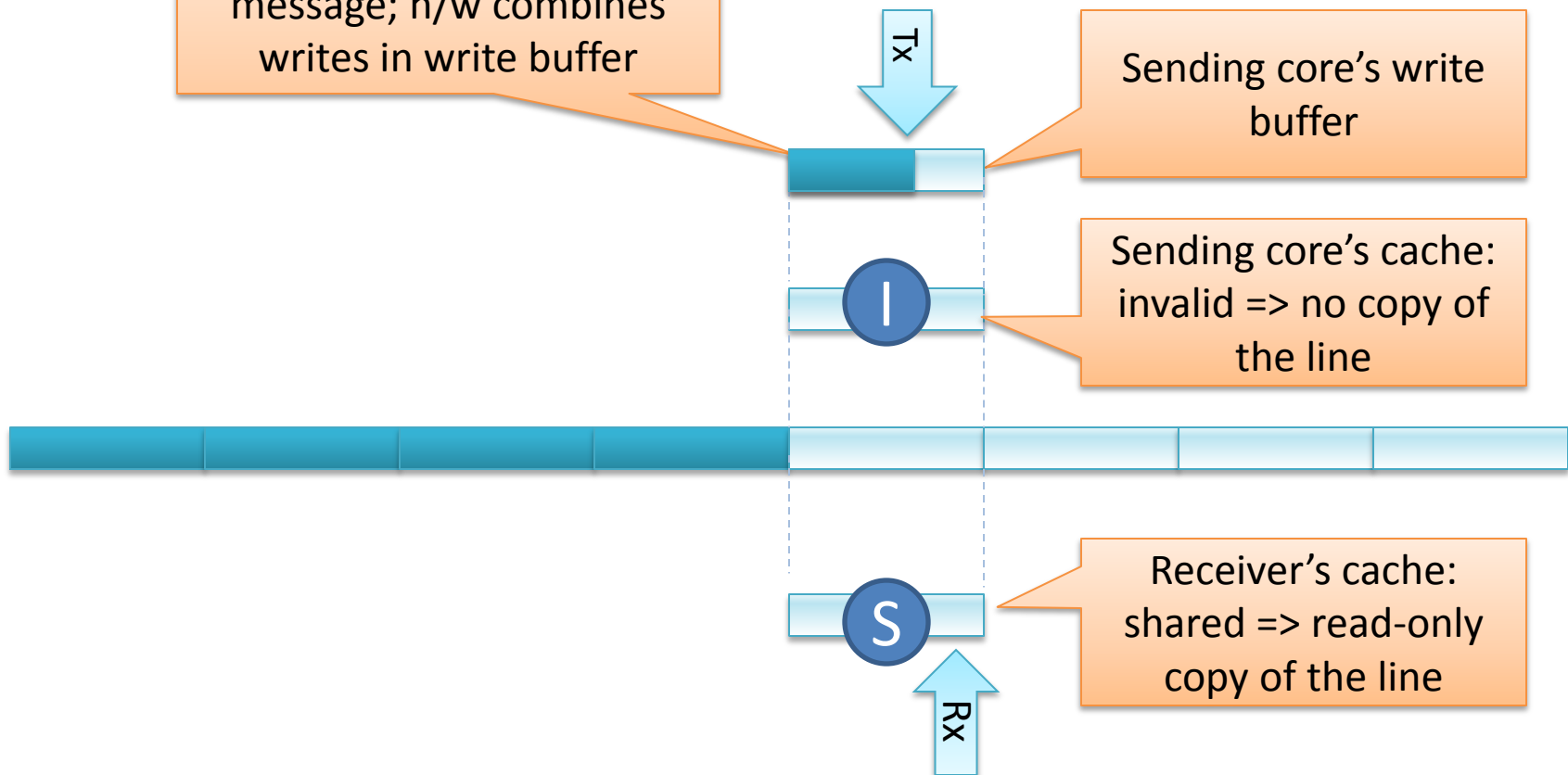
Sender starts to write message; h/w combines writes in write buffer



CC-UMP: cache-coherent user-space messaging



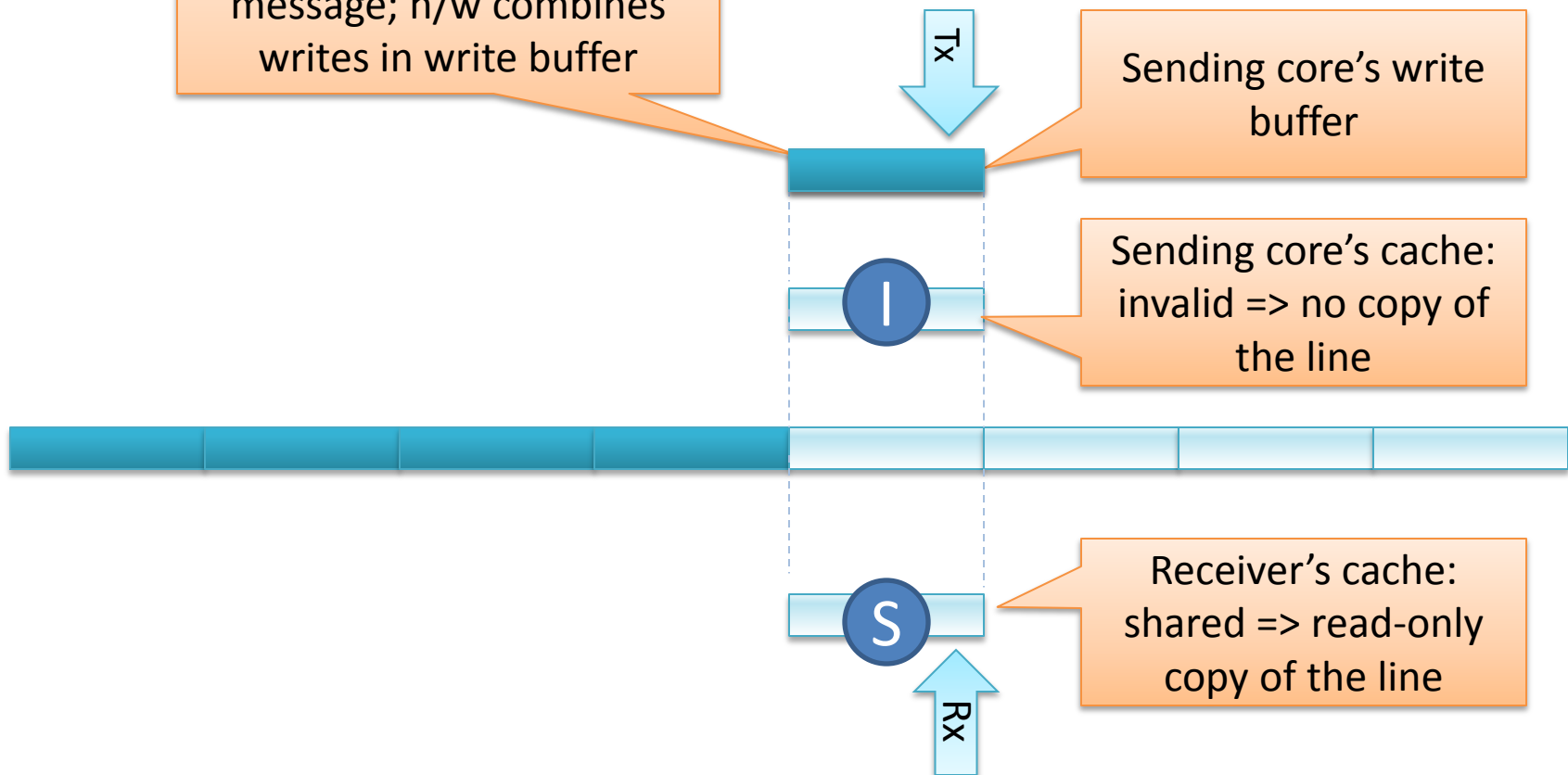
- 1 Sender starts to write message; h/w combines writes in write buffer



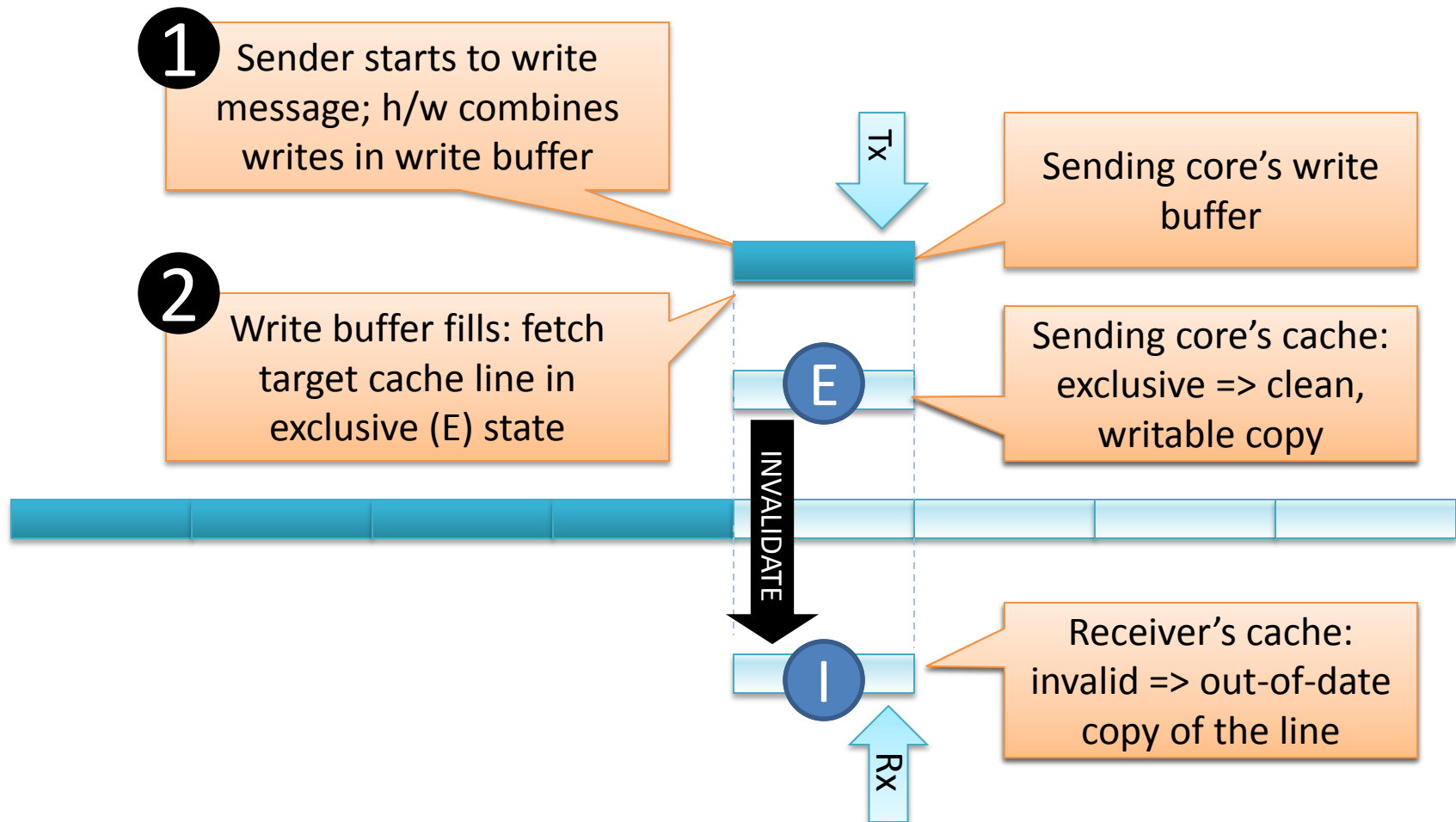
CC-UMP: cache-coherent user-space messaging



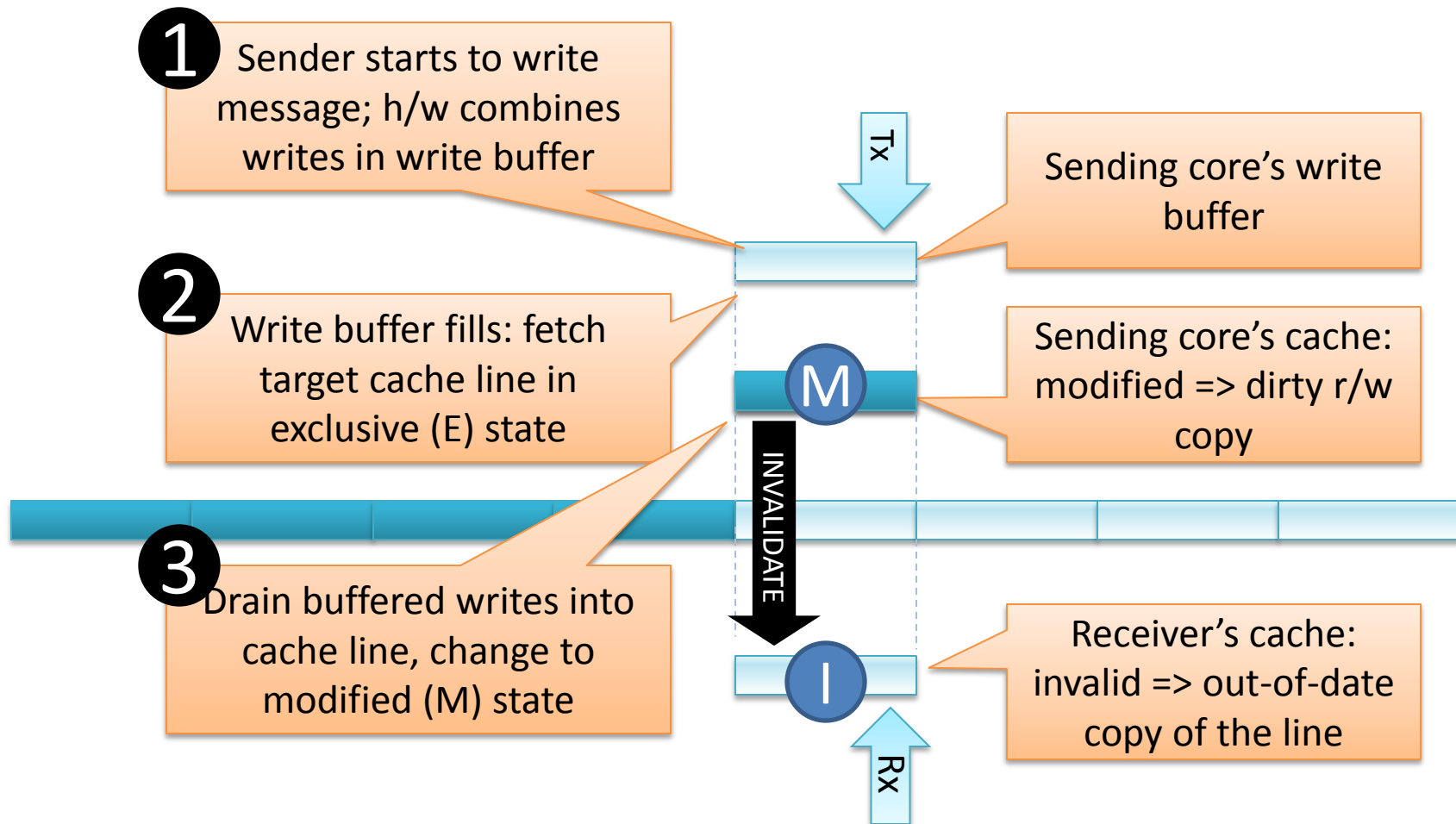
- 1 Sender starts to write message; h/w combines writes in write buffer



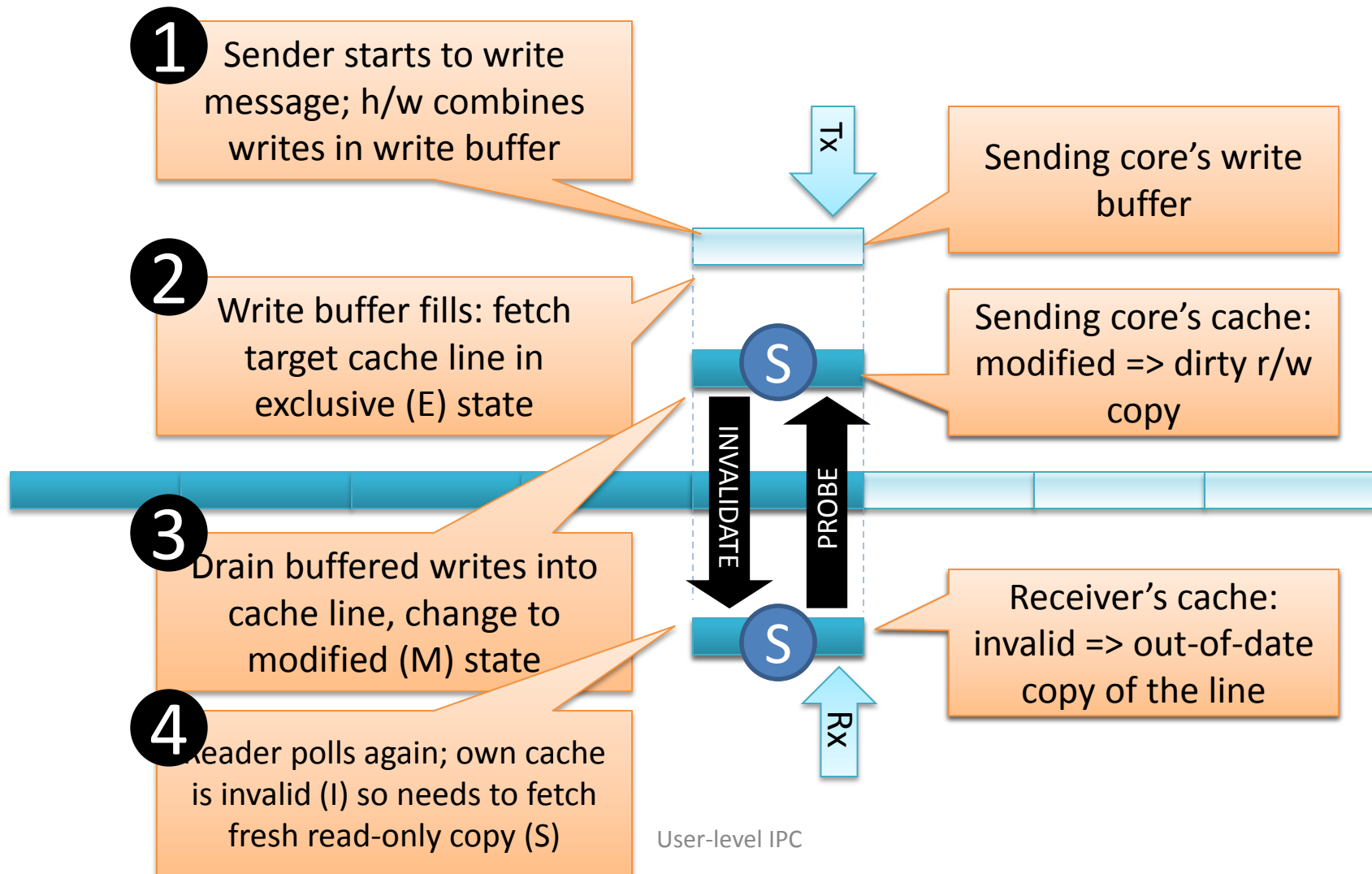
CC-UMP: cache-coherent user-space messaging



CC-UMP: cache-coherent user-space messaging



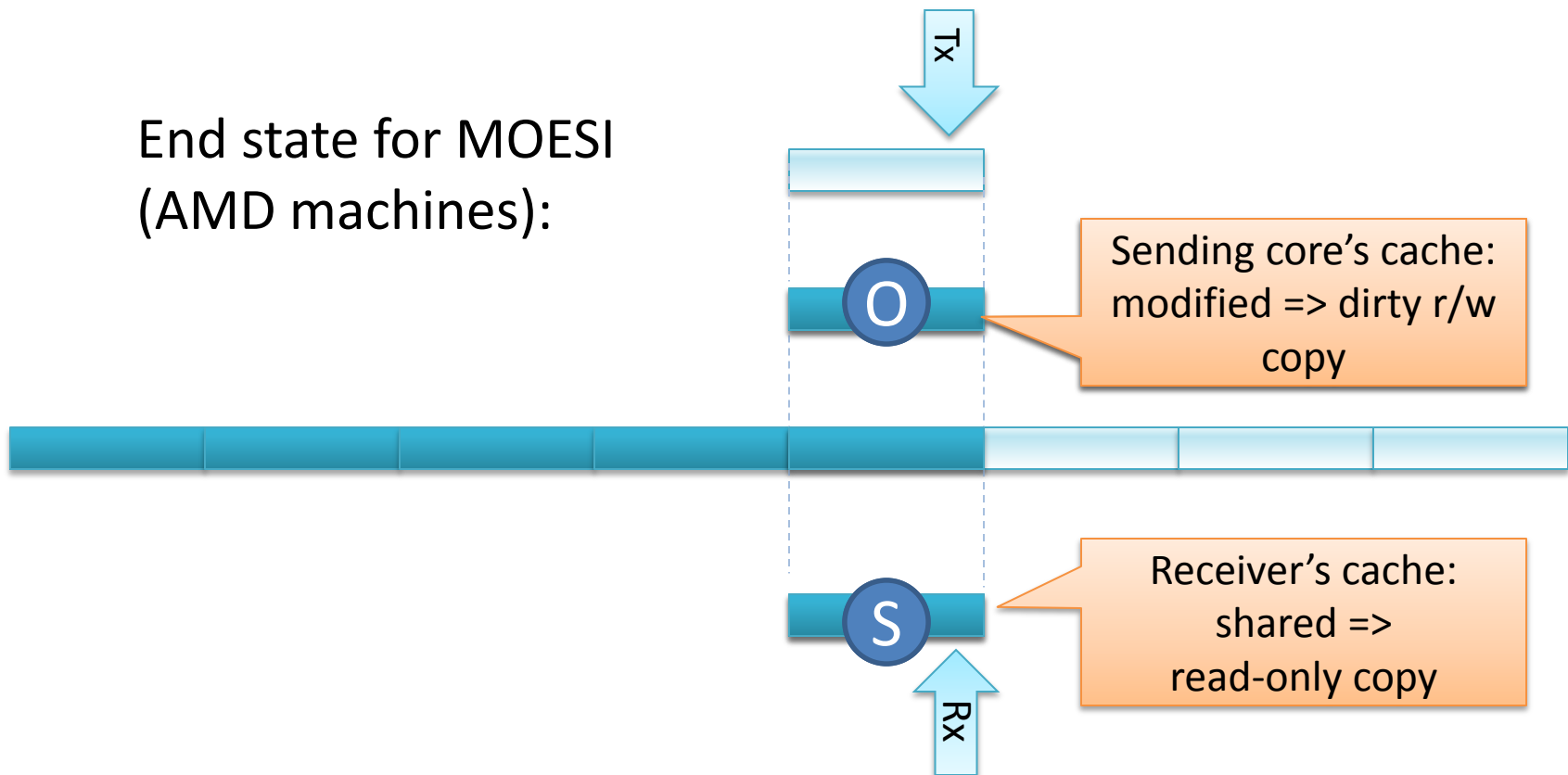
CC-UMP: cache-coherent user-space messaging



CC-UMP: cache-coherent user-space messaging



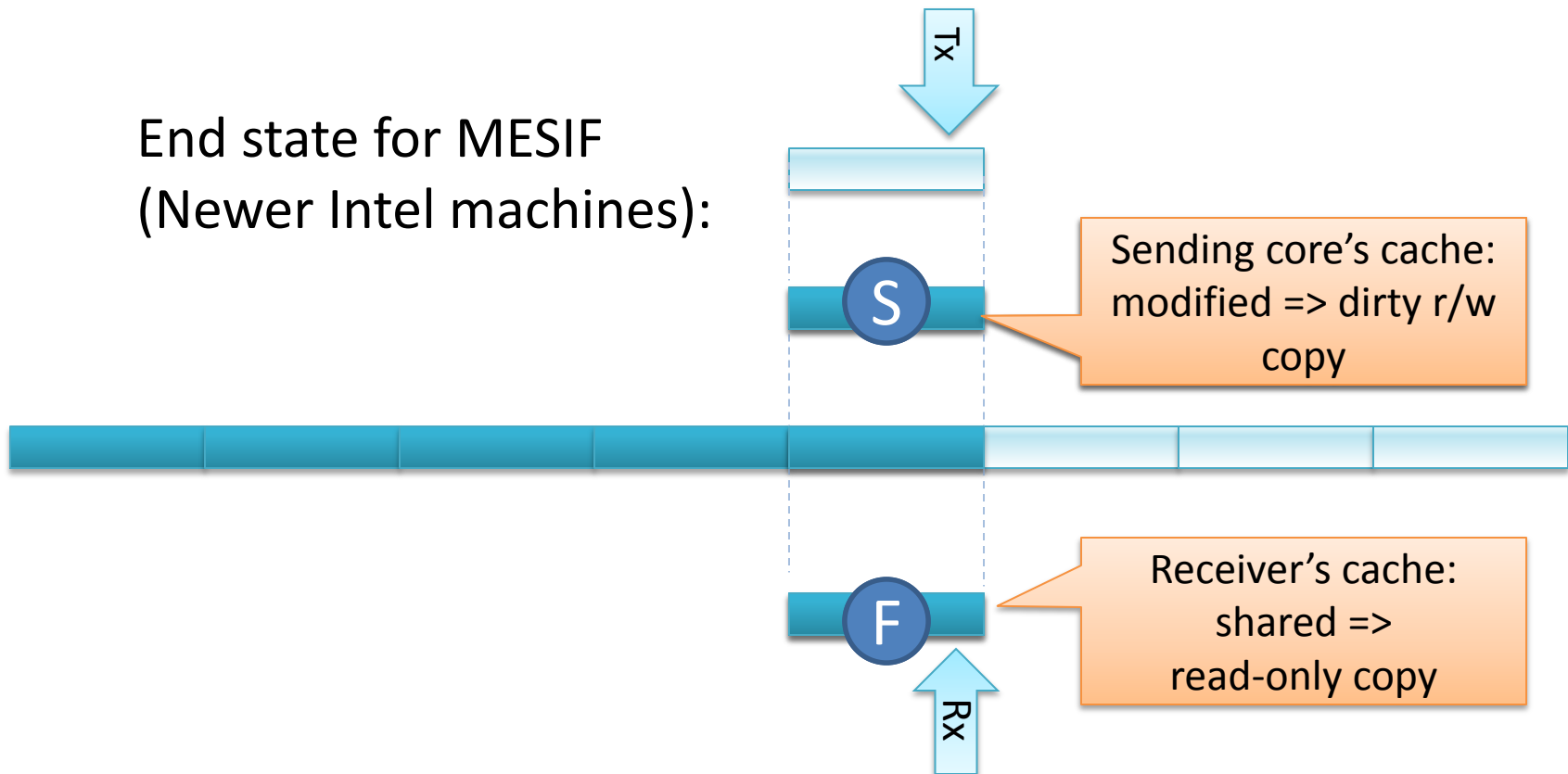
End state for MOESI
(AMD machines):



CC-UMP: cache-coherent user-space messaging



End state for MESIF
(Newer Intel machines):



UMP for AOS



- You need to extend your RPC interface to another core.
- Implement a simple UMP-style channel:
 - Don't worry about optimal performance.
 - You *do* need to worry about the caches and ARM's memory model (more in a moment).



ARM BARRIERS

Memory consistency models



If one CPU modifies memory, when do others observe it?

- **Strict/Sequential**: reads return the most recently written value
- **Processor/PRAM**: writes from one CPU are seen in order, writes by different CPUs may be reordered
- **Weak**: separate rules for synchronizing accesses (e.g. locks)
 - Synchronising accesses sequentially consistent
 - Synchronising accesses act as a barrier:
 - previous writes completed
 - future read/writes blocked

Important to know your hardware!

- x86: processor consistency
- ARM: weak consistency

ARM Barriers



3 basic types of barrier:

– dmb

- Stores before this point complete before loads and stores that come afterward.

– isb

- Further instructions aren't fetched until this completes.

– dsb

- Enforce ordering with system operations – ignore this for now.

Instruction Stream Sync



`str x0, [x1]`

`dmb`

`isb`

`b x1`

- Store opcode to `*x1`

- **Barrier**

- Branch to `*x1`

- Why `dsb` *and* `isb`?
 - `dmb` ensures store to `*x1` completes, before `dmb` itself completes.
 - `isb` ensures branch target isn't fetched until `dsb` has completed.
- Important for ELF loading!

Barriers and UMP



- This doesn't work.
- Why?

```
data= x;  
flag= 1;
```

Sender

```
while(!flag);  
y= data;
```

Receiver

Barriers and UMP



- Valid execution order:
- This actually happens on ARM.

```
4 data= x;  
1 flag= 1;
```

Sender

```
2 while(!flag);  
3 y= data;
```

Receiver

Barriers and UMP



- Does this work?

```
data= x;  
dmb;  
flag= 1;
```

Sender

```
while(!flag);  
y= data;
```

Receiver

Barriers and UMP



- Still no!
- This is valid, and observable on ARM!
- There's no data dependency between the reads.

```
2 data= x;  
3 dmb;  
4 flag= 1;
```

Sender

```
5 while(!flag);  
1 y= data;
```

Receiver

Barriers and UMP



- This (finally) works.
- ARM is wild (so is Power).
- Always think about possible reorderings.
 - Or don't write racy code!

```
data= x;  
dmb;  
flag= 1;
```

Sender

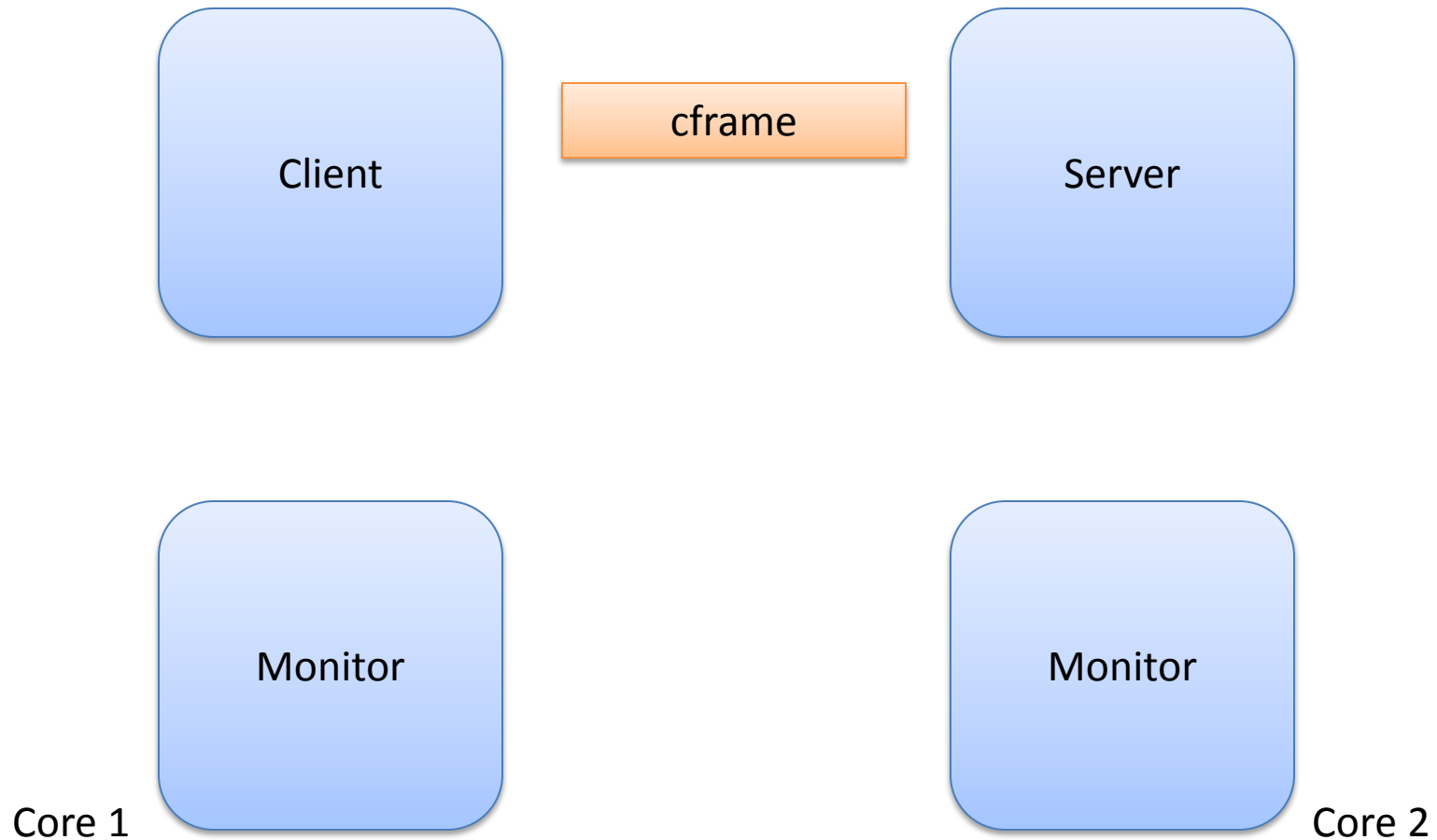
```
while(!flag);  
dmb;  
• y= data;
```

Receiver



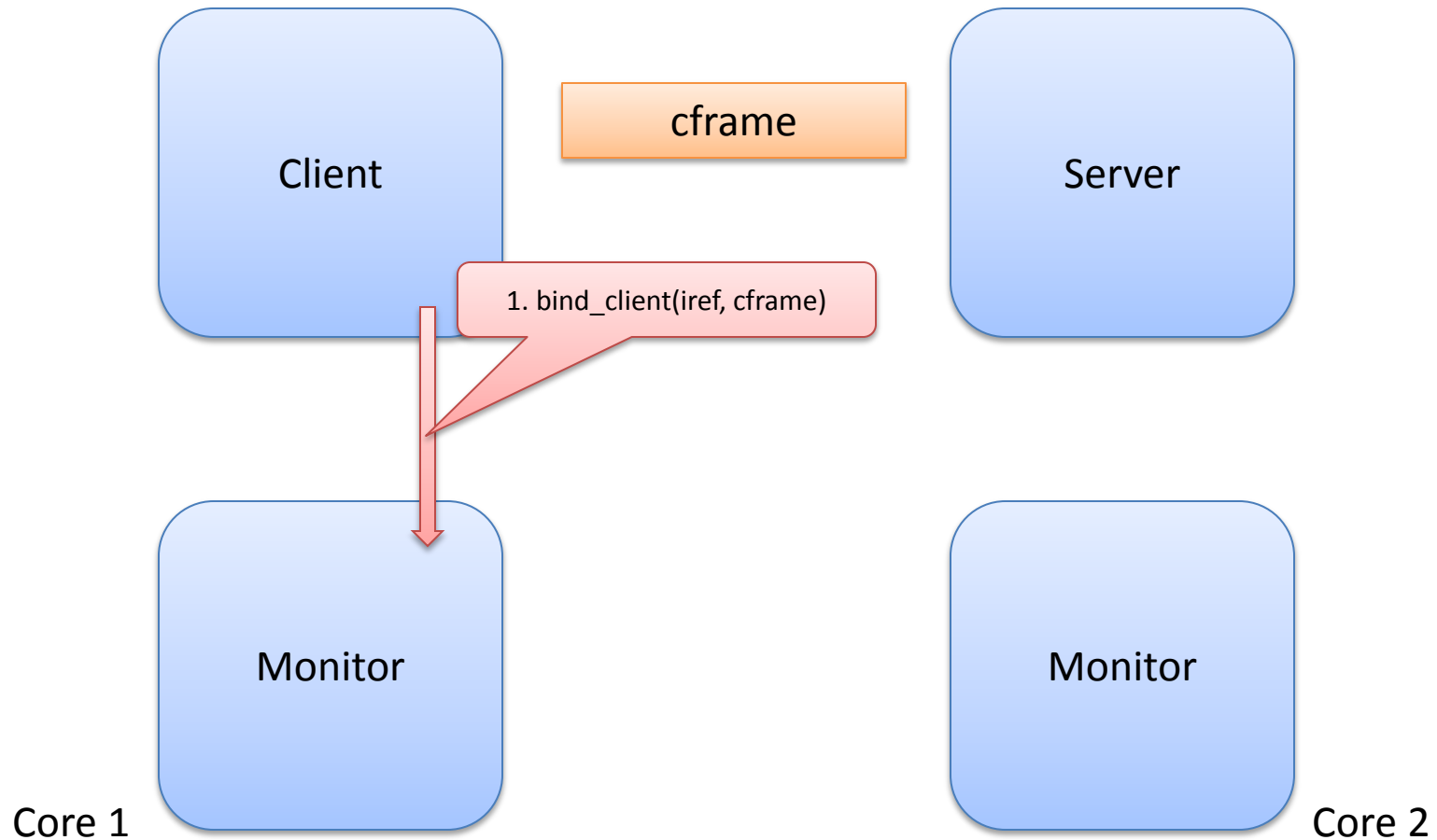
CONNECTION SETUP

Communication binding



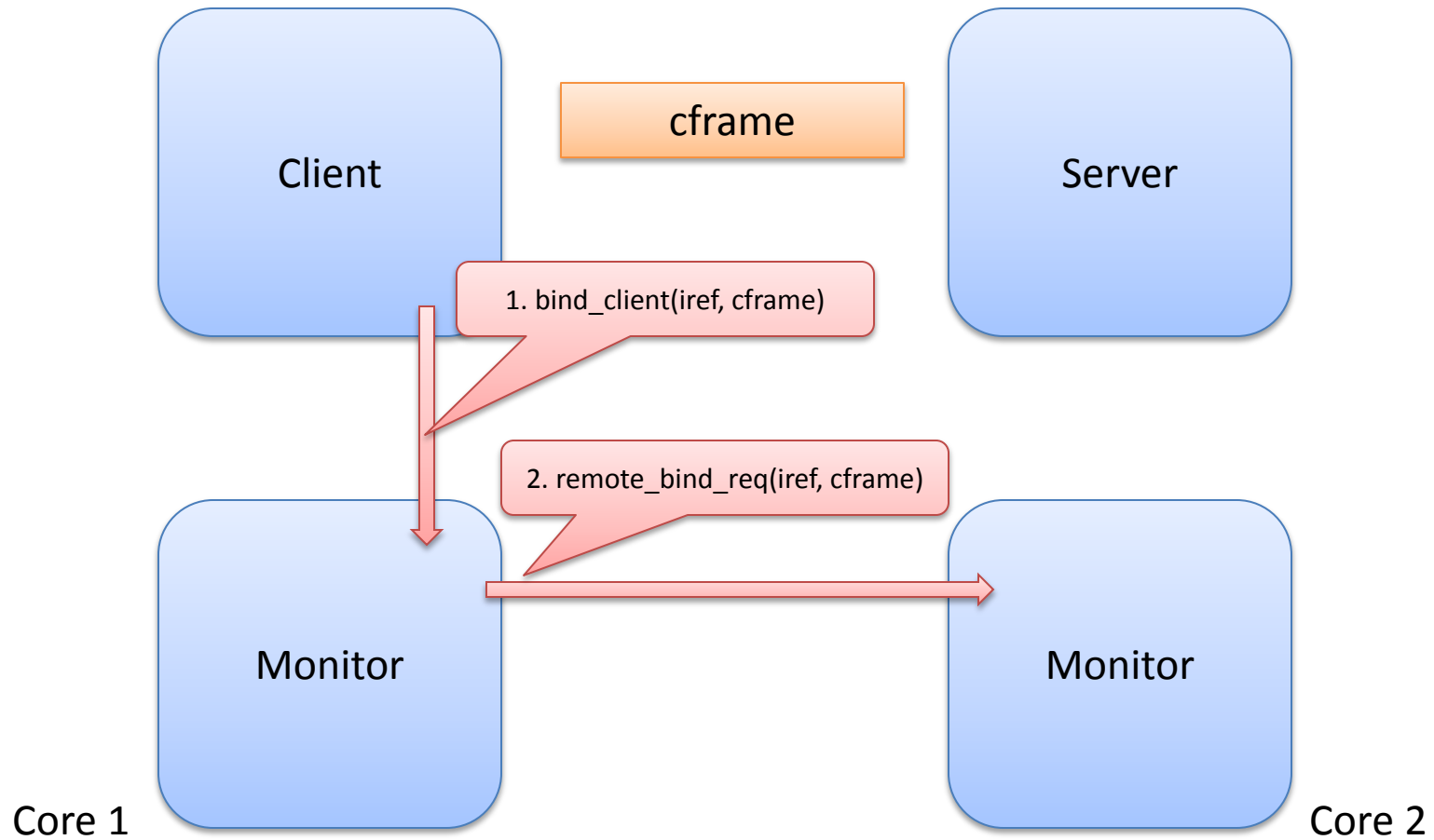
Monitors route binding requests and replies

Communication binding



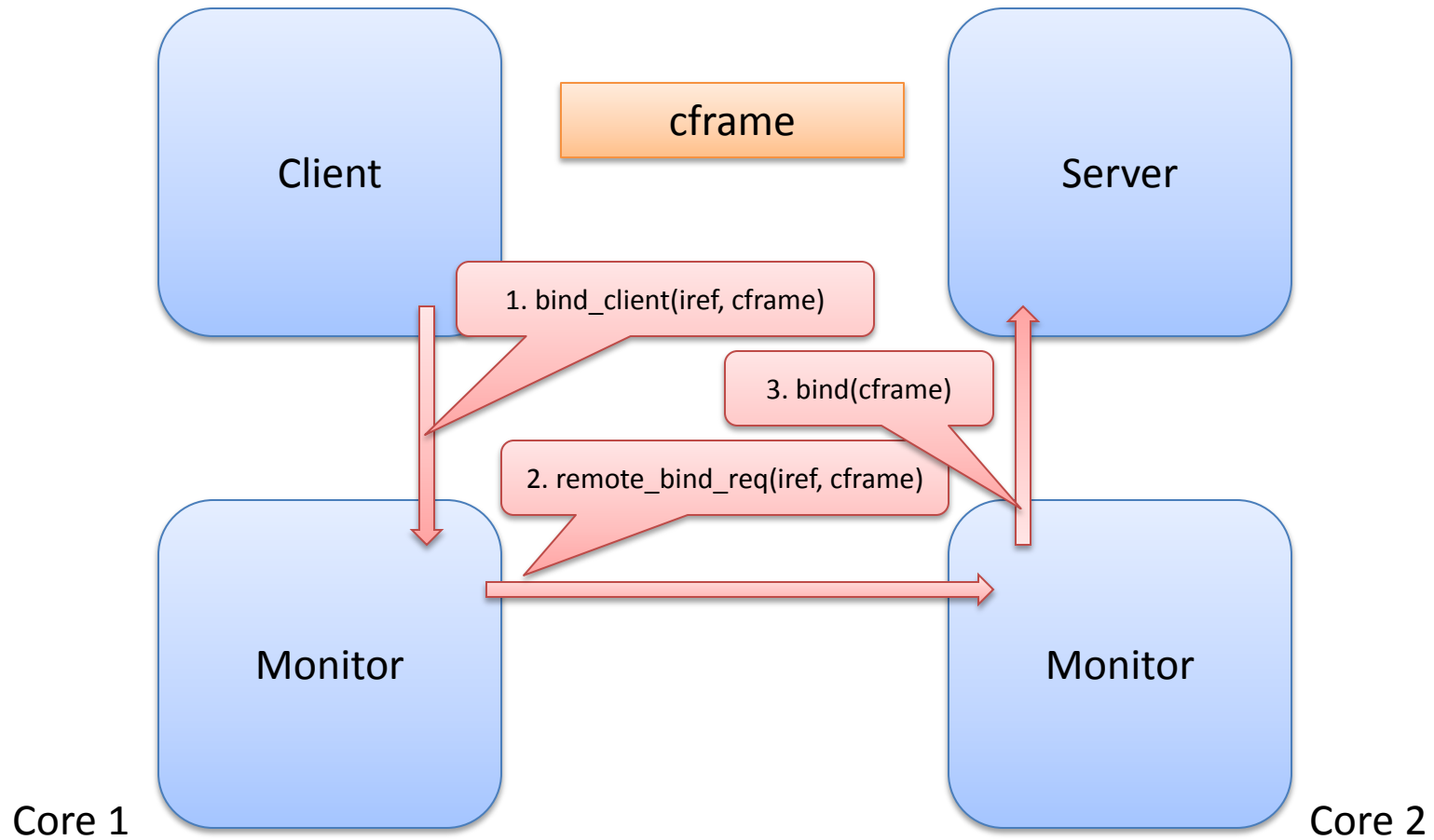
Monitors route binding requests and replies

Communication binding



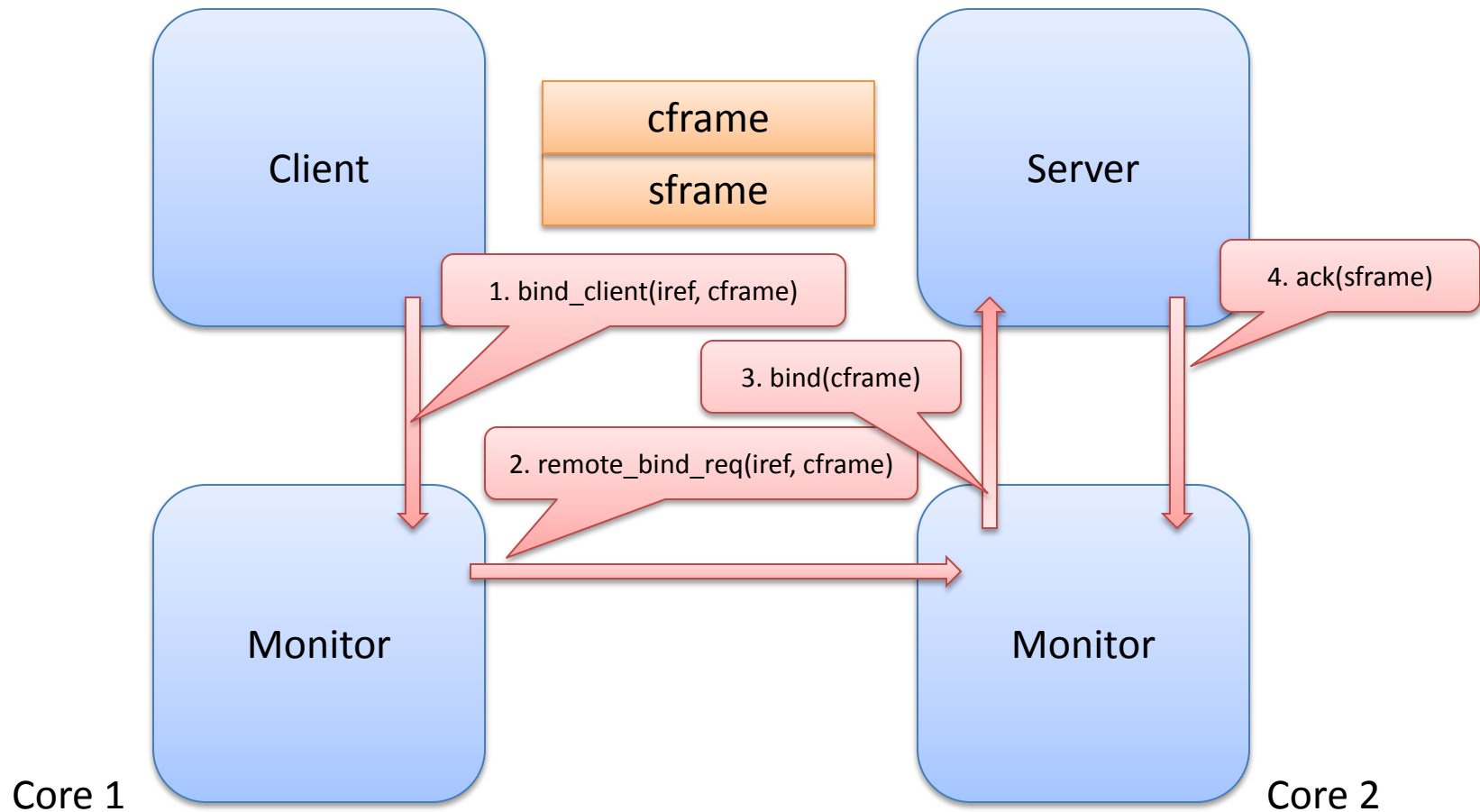
Monitors route binding requests and replies

Communication binding



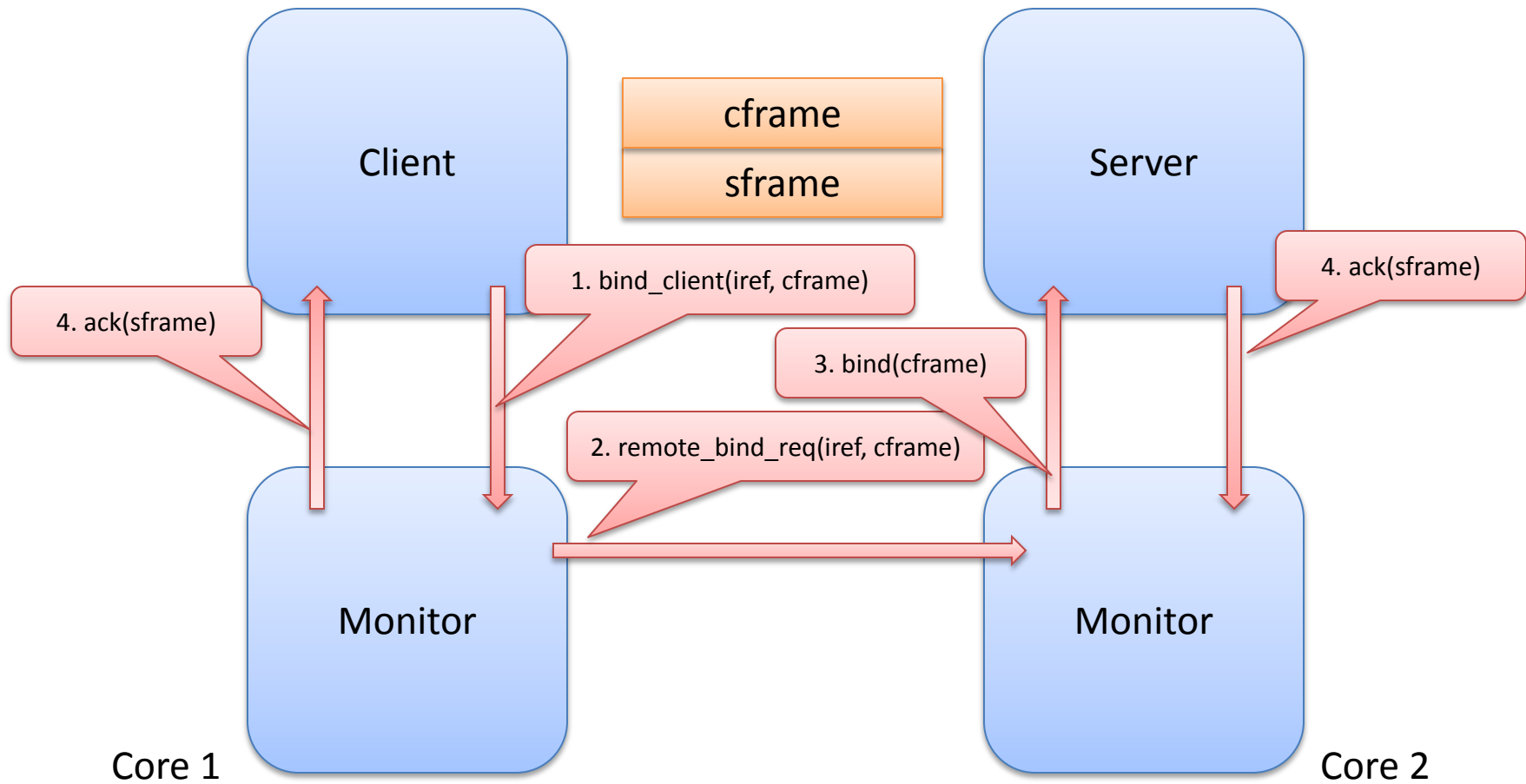
Monitors route binding requests and replies

Communication binding



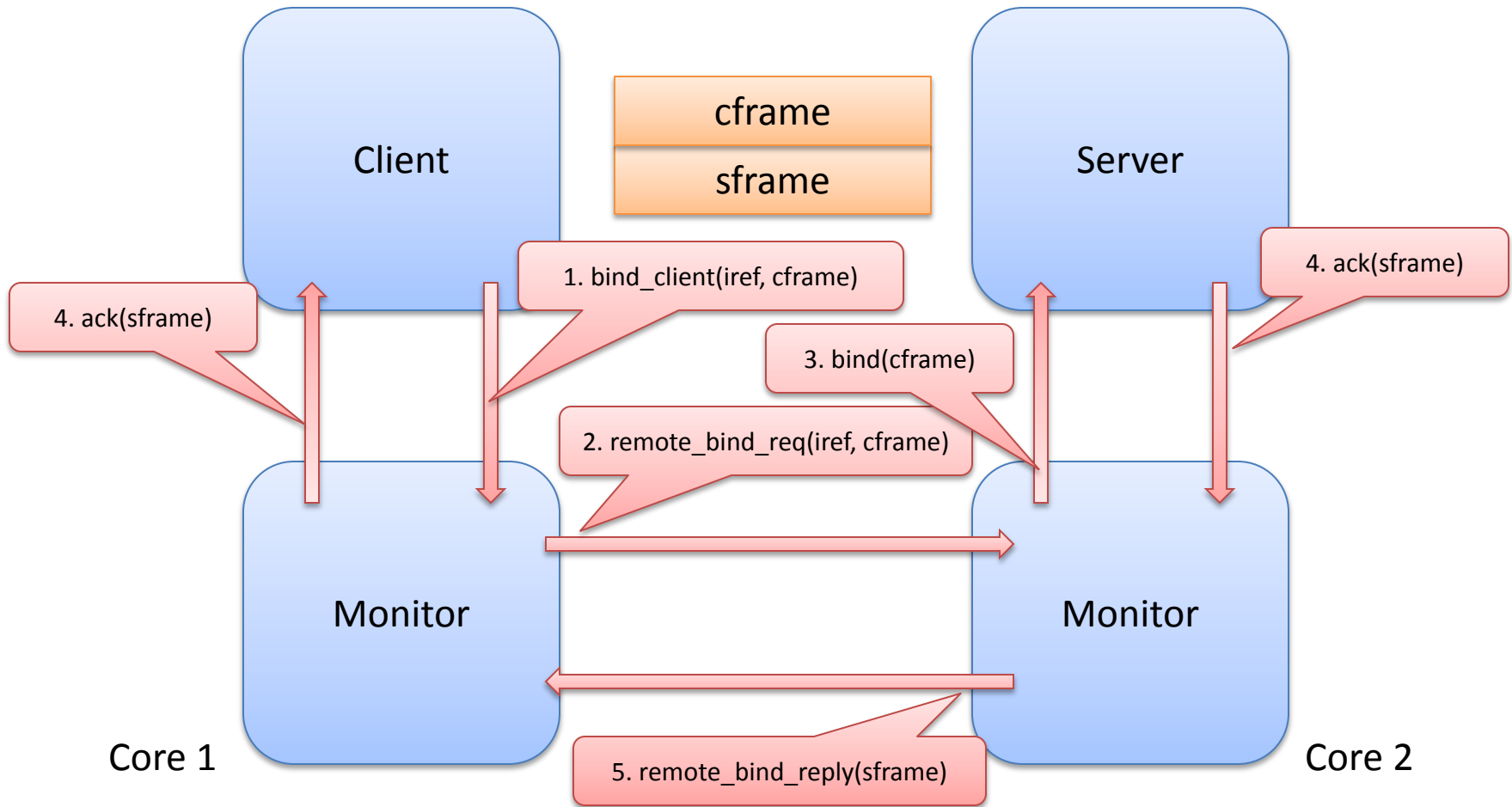
Monitors route binding requests and replies

Communication binding



Monitors route binding requests and replies

Communication binding



Monitors route binding requests and replies

Discussion



- Close to underlying interconnect messages
 - 2 x cache transactions per message
 - Main memory is not touched!
- Fixed-size units
 - 56 bytes payload
- Core-to-core not thread-thread
 - Deal with thread migration out of band
- Still polled
 - Separate *notification driver* can send an IPI



FURTHER READING

Further reading



- John Giacomoni, Tipp Moseley, and Manish Vachharajani. 2008. FastForward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP '08). ACM, New York, NY, USA, 43-52.
<http://dx.doi.org/10.1145/1345206.1345215>
- M. Schroeder and M. Burrows. 1989. Performance of Firefly RPC. In Proceedings of the twelfth ACM symposium on Operating systems principles (SOSP '89). ACM, New York, NY, USA, 83-90. <http://dx.doi.org/10.1145/74850.74859>
- Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. 1991. User-level interprocess communication for shared memory multiprocessors. ACM Trans. Comput. Syst. 9, 2 (May 1991), 175-198.
<http://dx.doi.org/10.1145/103720.114701>
- Andrew D. Birrell and Bruce Jay Nelson. 1984. Implementing remote procedure calls. ACM Trans. Comput. Syst. 2, 1 (February 1984), 39-59.
<http://dx.doi.org/10.1145/2080.357392>