# ADVANCED PARALLEL COMPUTING 2017
# LECTURE 07 - SCALABLE COHERENCE

Holger Fröning
holger.froening@ziti.uni-heidelberg.de
Institute of Computer Engineering
Ruprecht-Karls University of Heidelberg

*Some material by Falsafi, Hardavellas, Nowatzyk of EPFL, Northwestern, CMU*

# SCALABLE CACHE COHERENCE

Snooping: totally ordered interconnection network

- Requests: Get-S, Get-M, Put-M

Part 1: bus bandwidth

- Replace non-scalable, frequency-limited shared medium with a scalable point-to-point interconnection network
- Mesh, tori, BMIN, etc…

Part 2: snooping bandwidth

- Observation: most snoops result in no action
- I.e., cache line is not shared
- Replace non-scalable broadcast protocol with more scalable directory protocol
- Broadcasts => multicasts

# BASICS OF DIRECTORY COHERENCE PROTOCOLS

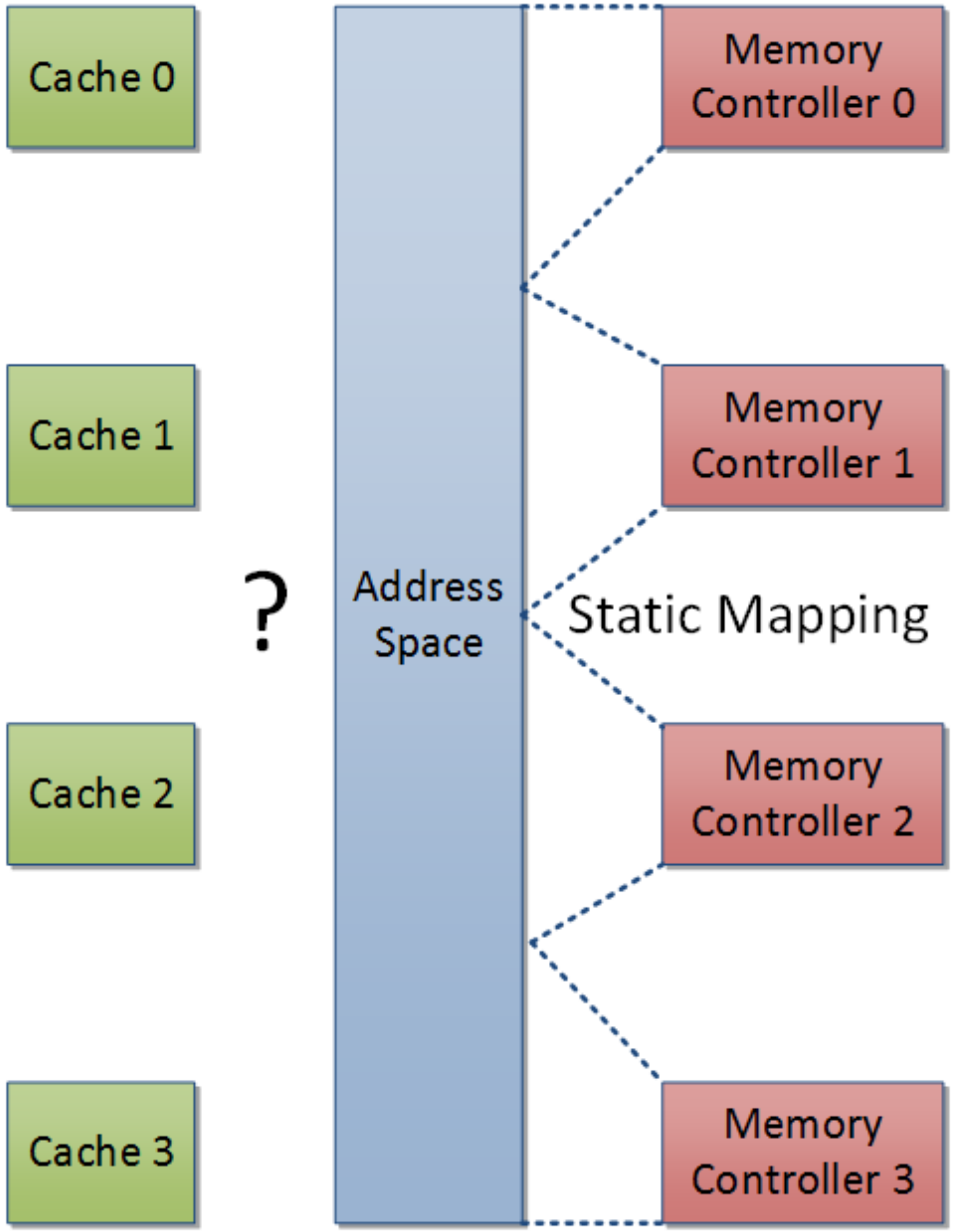
# DIRECTORY COHERENCE PROTOCOLS

Observation: physical address space is statically partitioned for a multi-processor system

- System with multiple memory controllers
- Easy to determine which memory module "home" holds a given line

Observation: caching is highly dynamic, frequent copying & migration of cache lines

- Difficult to determine which processors have a given line in their caches
- Main reason for bus-based protocols: inherent broadcast probes all caches
- Simple and fast, but non-scalable



# DIRECTORY COHERENCE PROTOCOLS

Directories: non-broadcast coherence protocol

- Extend memory to track caching information
- I.e., references to caches that include copies

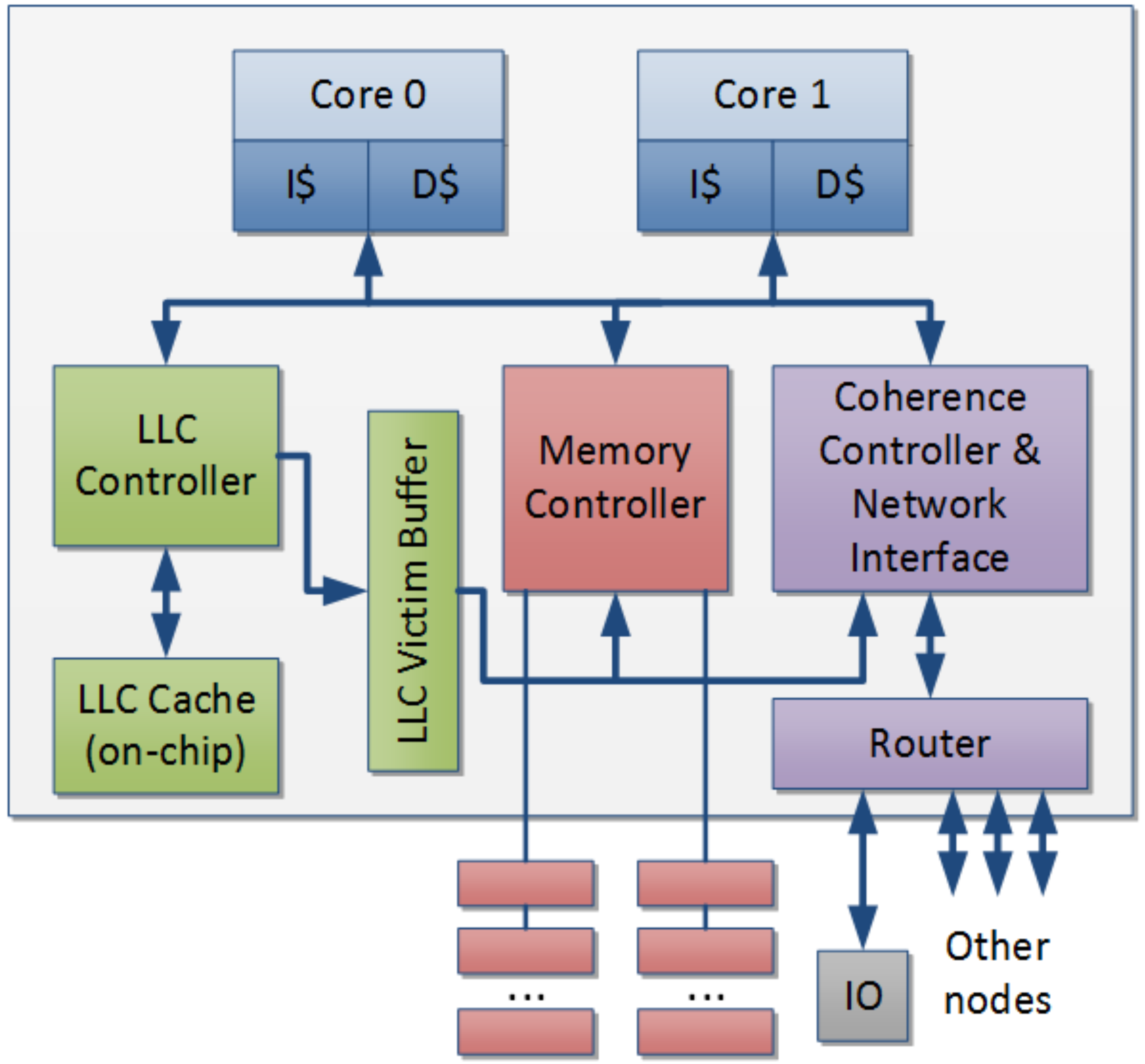
For each physical cache line (that is homed here), track:

- Owner: processor that has a dirty copy (modified state)
- Sharers: processors that have clean copies (shared state)

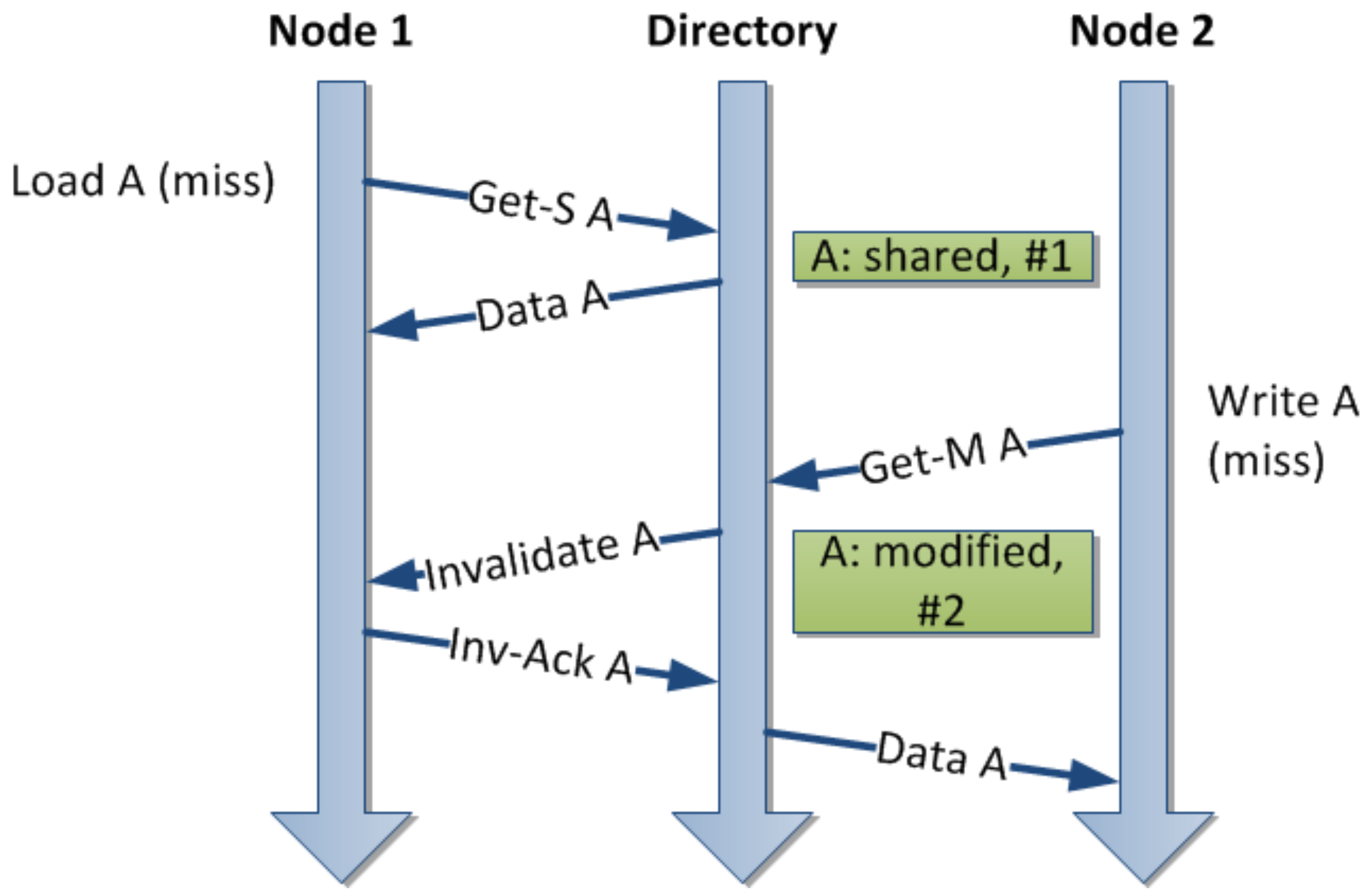
Each requesting processor sends coherence events to home directory

- Home directory forwards events to relevant caches only

=> Multi-cast instead of broad-cast



# BASIC OPERATION: READ/WRITE



Centralized Directory

Distributed Directory

# CENTRALIZED DIRECTORY

Single directory contains a copy of the cache tags of all nodes

Advantages

- Send Invalidate/Update only to the nodes that have copies
- Central serialization point: easy to get strong consistency
- Works like a bus

Disadvantages

- Not scalable, contention point
- Directory size/organization changes with number of nodes

# DISTRIBUTED DIRECTORY

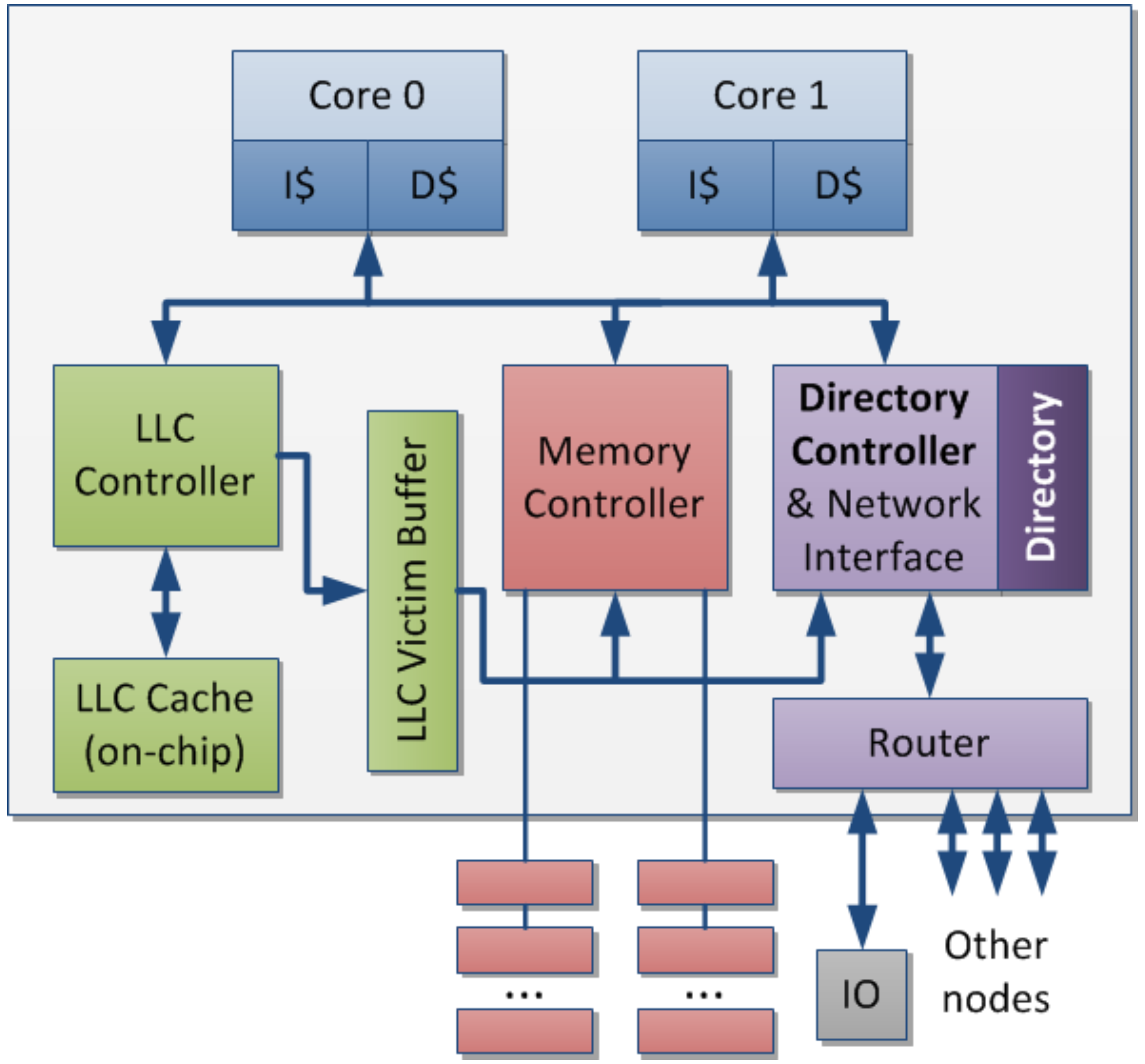
Distribute directory among memory modules

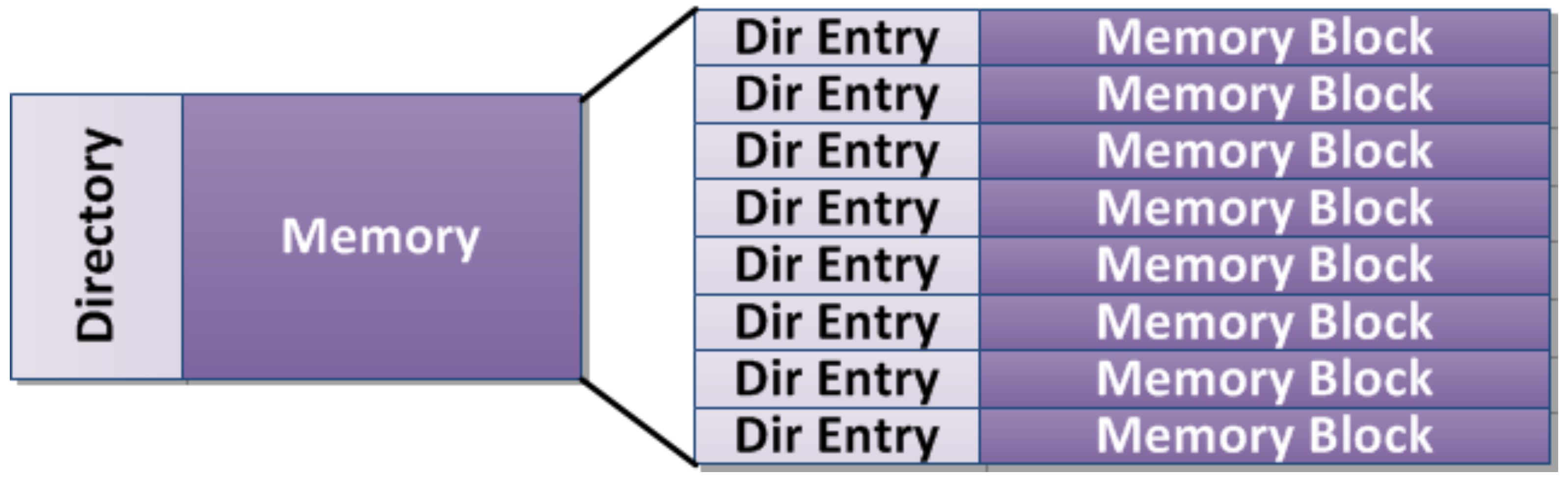
- Memory block = coherence block (= cache line)
- Home node: node with directory entry
- Usually also the main memory storage for cache line
- Scalable: directory grows with memory capacity
- Common trick: steal bits from ECC for directory state

Problems

- Directory can no longer serialize accesses across all addresses

=> Memory consistency becomes responsibility of processor (interface)





| | |
|---|---|
| Dir Entry | Memory Block |
| Dir Entry | Memory Block |
| Dir Entry | Memory Block |
| Dir Entry | Memory Block |
| Dir Entry | Memory Block |
| Dir Entry | Memory Block |
| Dir Entry | Memory Block |
| Dir Entry | Memory Block |

# CONTENTS OF A DIRECTORY

Directory State

- Invalid, Exclusive, Shared, …
- Few stable states
- Usually 2-3 bits are enough
- Number of outstanding invalidations, …
- Transient states

Pointer to exclusive owner

Sharer list

- List of caches that may have a copy
- May include local node
- Not necessarily precise, but conservative

=> Often 10s of transient states, often with references to node IDs, etc.

- Transient state changes frequently, need fast RMW ops

Design options

- Keep in directory: scalable (high concurrency), but slow
- Keep in separate memory
- Keep in directory, use cache to accelerate access
- Keep in protocol controller
- Transaction State Register File – like MSHRs

One directory entry per memory block (e.g., 64 bytes)
=> Huge directories
=> Entry size matters

# CONTENTS OF A DIRECTORY: EXCLUSIVE OWNER

Only one reference => simple node ID (N bits), $2^N$ nodes

Can share storage with sharers list ==> only 1 bit

- Disjoint use

May point to a group of nodes that internally maintain coherence (hierarchy, e.g., via snooping)

- Coarse directories (think of multi-core)

May treat local node differently

# CONTENTS OF A DIRECTORY: SHARER LIST

Key to scalability – number of sharers can be huge

Must efficiently represent node subsets

Observation: most blocks only cached by 1-2 nodes

- However, a few blocks are heavily shared: synchronization variables

Bit vectors: one bit per processor/node/cache

- Storage requirements grow with system size -> not scalable

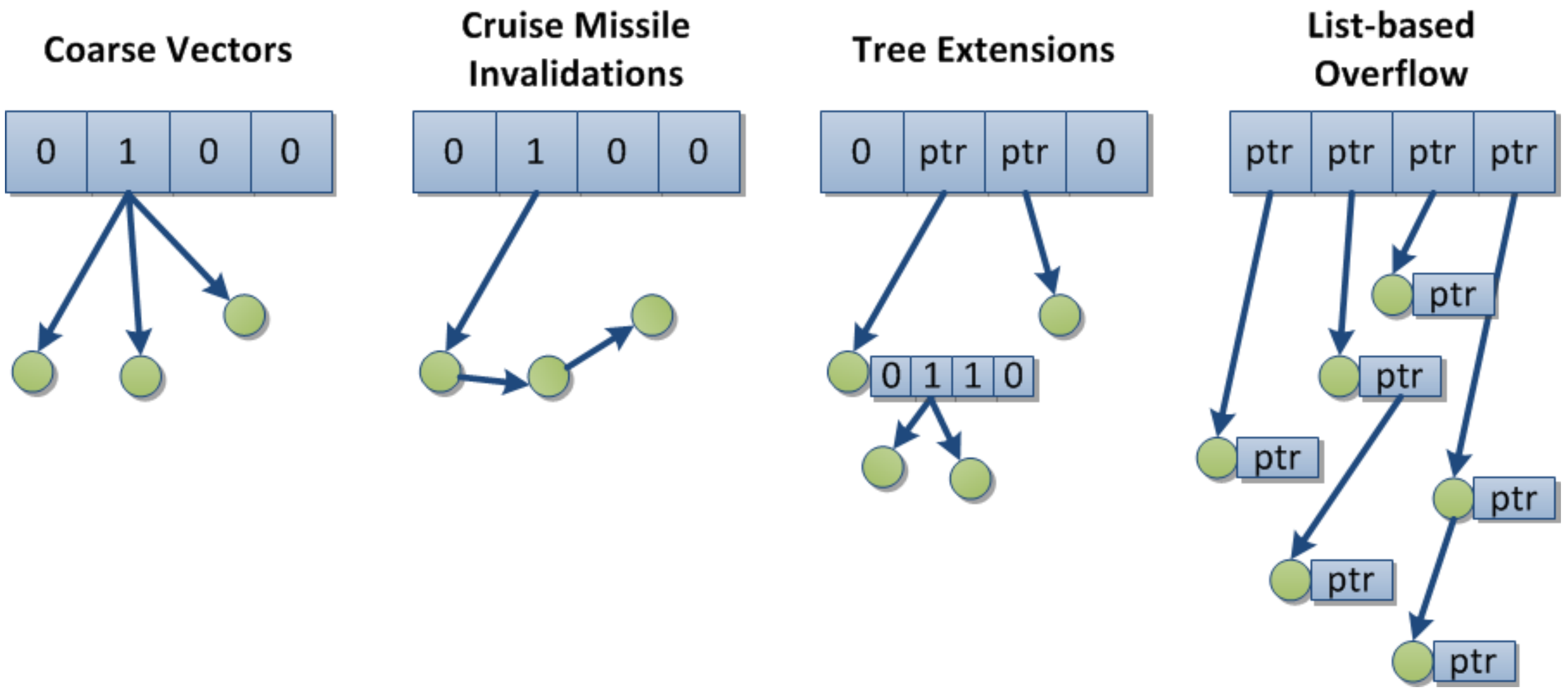
Limited pointer list: fixed number of pointers to node IDs (e.g., 4)

- If more than n sharers: (a) recycle one pointer (invalidation), (b) revert to broadcast, (c) handle in software

Linked lists: each node stores pointer to previous and next sharer

- Double linked list: Scalable Coherent Interconnect (SCI)
- Single linked list: S3.mp
- Scalable, but poor performance: long invalidation latency, replacement difficult (in particular for single linked list)

# CONTENTS OF A DIRECTORY: SHARER LIST OPTIMIZATIONS



Example: How to send around an invalidate request

Open question is the number of responses to expect (serialization)

# ADVANCED: NOTIFICATION OF CLEAN EVICTION

Should directory learn when clean blocks are evicted? (Put-S)

- Avoid broadcast, free pointer in limited pointer schemes (scarce resource) (yes)
- Avoids unnecessary invalidate messages later (yes)
- Read-only data never invalidated (extra evict messages) (no)
- Notification traffic is unnecessary (no)
- Avoid some protocol races (yes)

Race example without Put-S

- Cache silently evicts a block in S
- Cache sends Get-S request to re-obtain that evicted block in S
- Before receiving the response, it receives an invalidation
- Part of first or second period?
- Be safe: always invalidate block when it arrives (more efficient solutions complicate the protocol)

# ALTERNATIVE: SPARSE DIRECTORIES

Most of memory is invalid, why waste directory space?

- No need to reserve a directory entry for each memory block

Instead, use a directory cache

- Only a cached memory block gets an entry
- Any address without an entry is invalid
- If full, need to evict and invalidate a victim entry
- Generally needs to be highly associative

Cache invalidation patterns

- Assumption: on a write to a shared location, number of caches to invalidate is typically small
- Otherwise: broadcasts would be better than directories
- Experience tends to validate this

=> Sparse directory could maintain only a pointer to one node that is caching a block. Caching block contains pointer to next one.

# COMMON SHARING PATTERNS

- Code and read-only objects
  - No problem
- Migratory objects
  - Even as number of caches grow, typically only 1-2 invalidation
- Mostly-read objects
  - Invalidations are expensive but infrequent => ok
- Frequently read/written objects (task queues)
  - Frequent invalidations, hence sharer list is typically short
- Synchronization objects
  - Low-contention locks: few invalidations
  - High-contention locks: might need special support (e.g., MCS)
- Badly-behaved objects

# DIRECTORY EXAMPLE

# DIRECTORY EXAMPLE

L: Local Node

Node initiating the transaction

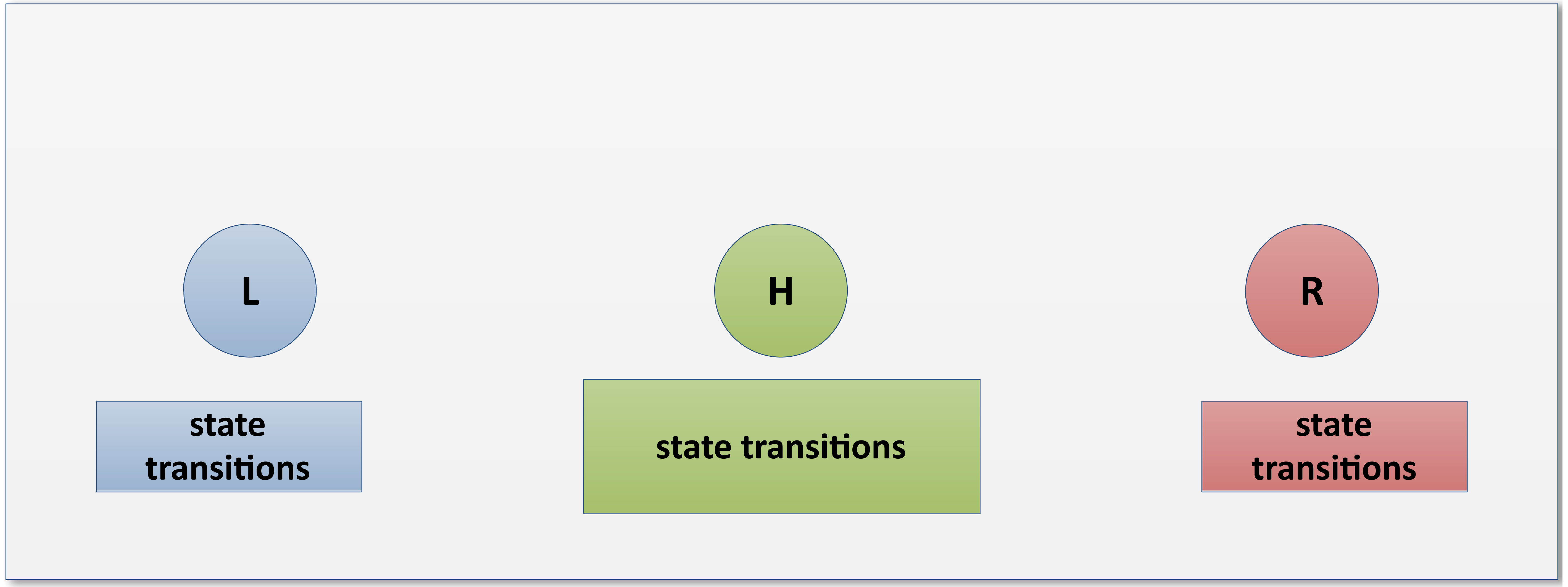
H: Home Node

Node hosting the memory block

R: Remote Node

Any other node participating in the transaction

Assume only MSI states



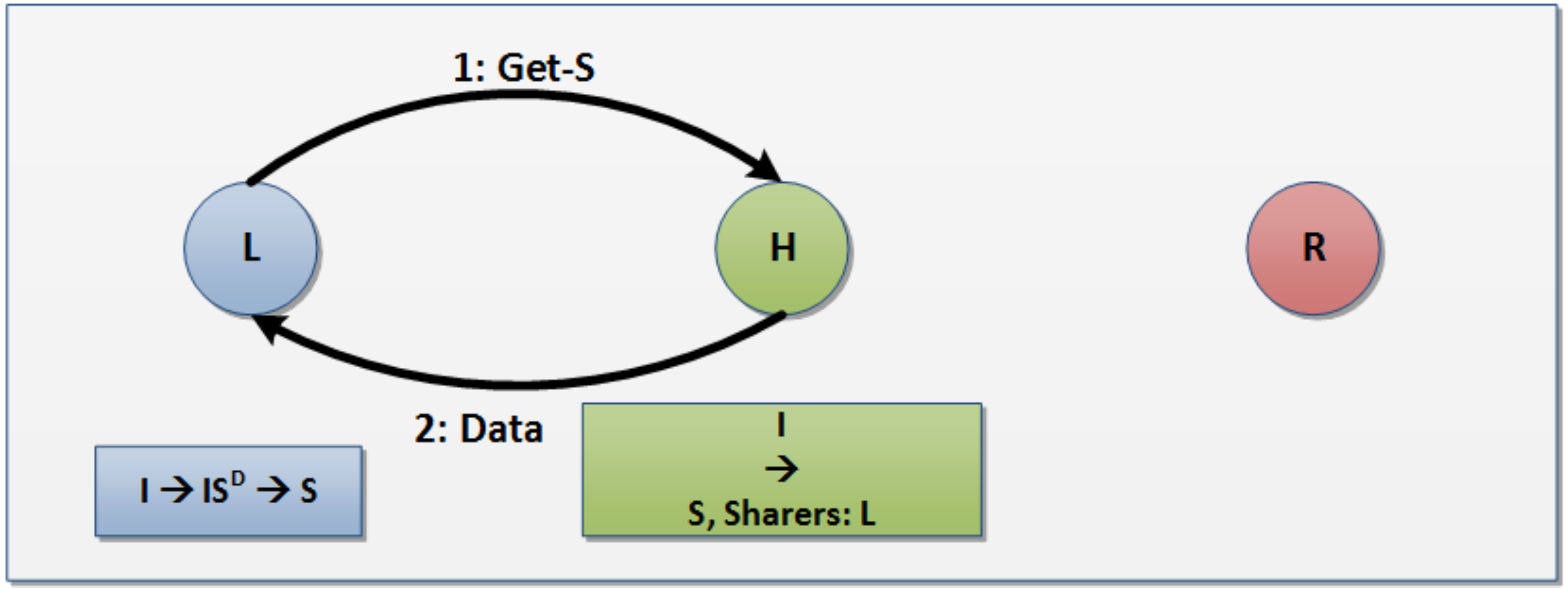
# DIRECTORY EXAMPLE: READ TRANSACTION

L: cache miss on a load instruction

Transient states (like $IS^D$)

- Superscript A: waiting for acknowledgement
- Superscript D: waiting for data

Same if there are other sharers:
H: {I,S} -> S

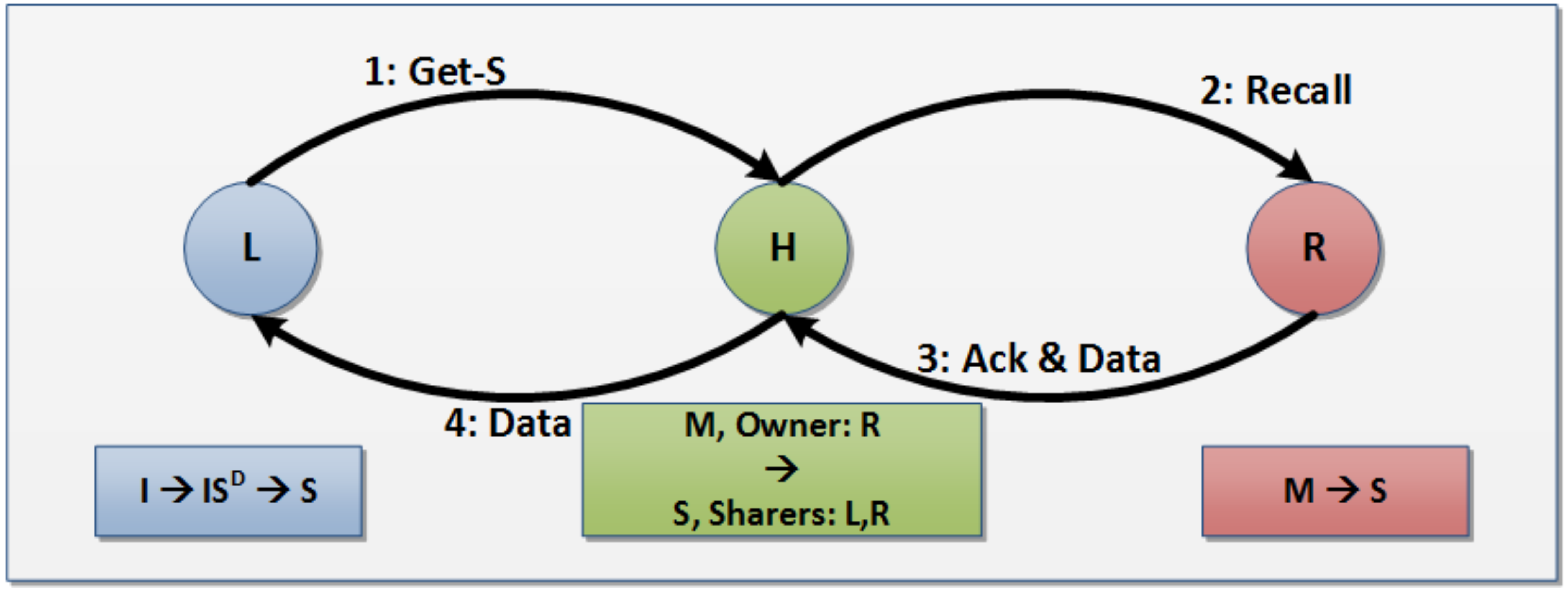


# DIRECTORY EXAMPLE: 4-HOP READ TRANSACTION

L: cache miss on a load instruction

Block was previously in modified state at R

Average latency per hop?



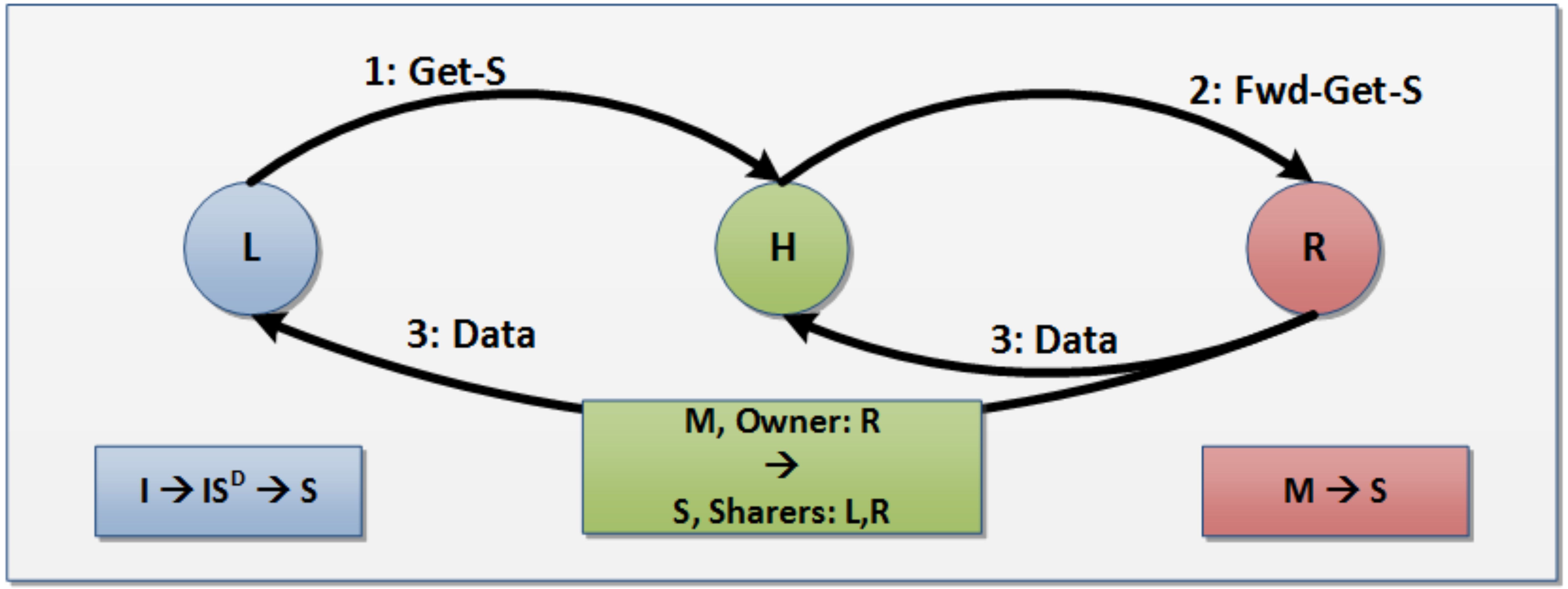
# DIRECTORY EXAMPLE: 3-HOP READ TRANSACTION

L: cache miss on a load instruction

Block was previously in modified state at R

Optimization: send response directly to L

Why also send data to H?



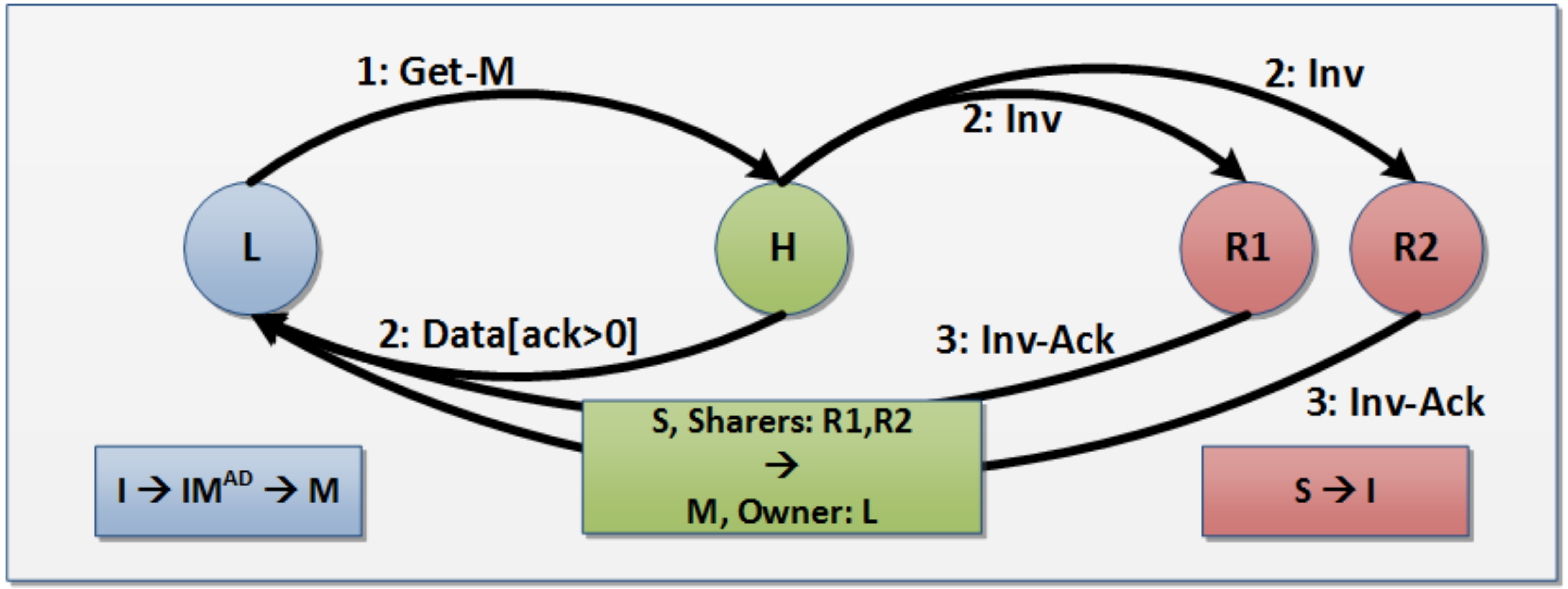
# DIRECTORY EXAMPLE: 3-HOP WRITE TRANSACTION

L: cache miss on a store instruction

Block was previously in shared state at R1, R2

Similar applies to L: S -> M

Note on ack>0



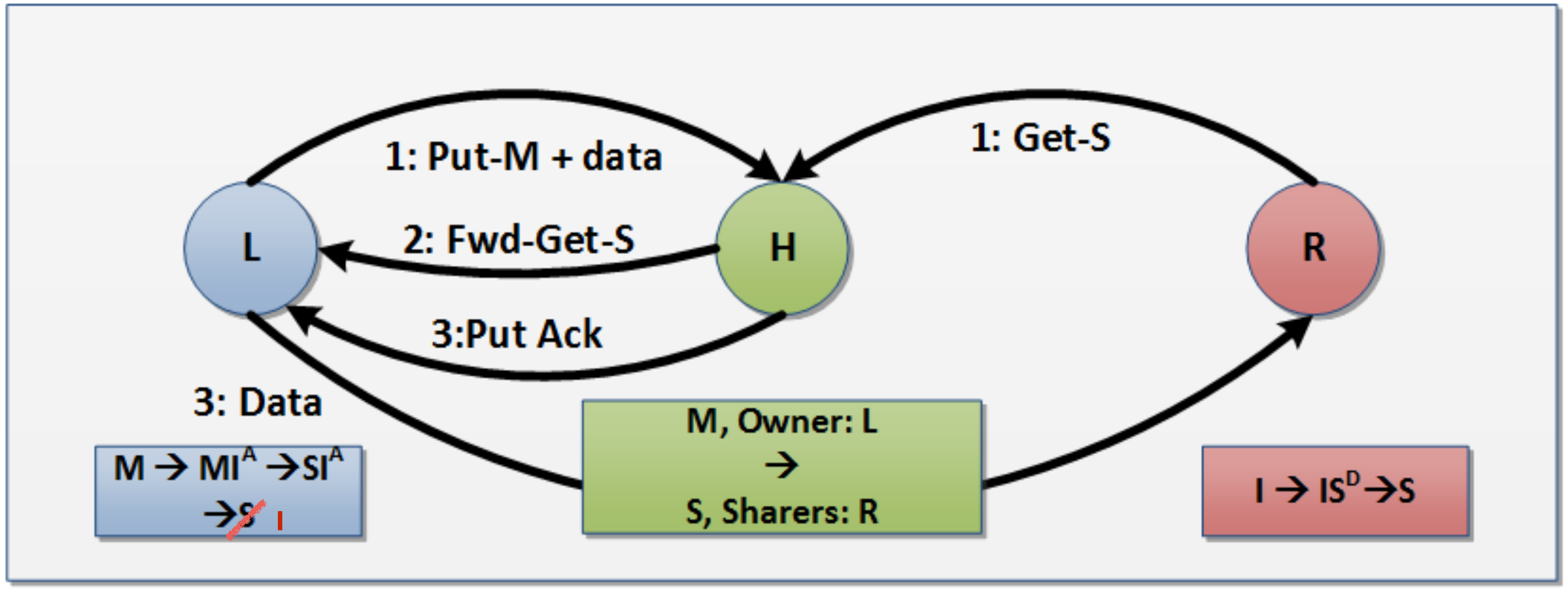
# DIRECTORY EXAMPLE: WRITEBACK-READ RACE

L: has dirty copy, wants to write back to H

R concurrently sends a Get-S to H

> H rather forwards the Get-S request than serving it, as caches are much faster than a memory access

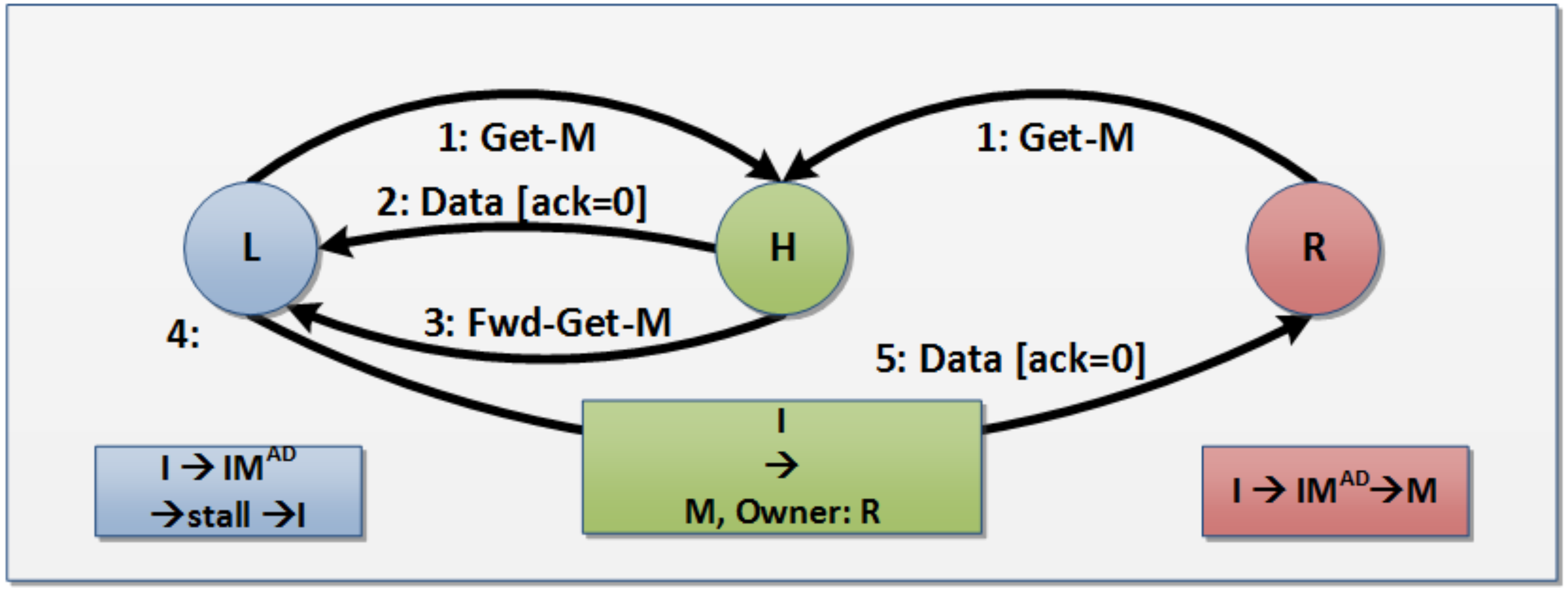
For R sending a write to H: Fwd-Get-M & L: M -> $MI^A$ -> $II^A$ -> I



# DIRECTORY EXAMPLE: STORE-STORE RACE

Line is invalid, both L and R race to obtain write permission

- L stalls for a single store, then forwards ownership



# DESIGN PRINCIPLES

# DESIGN PRINCIPLES

Think of sending and receiving messages as separate events

At each step, consider what new requests can occur

- E.g.: Can a new writeback overtake an older one?

Two messages traversing same direction implies a race

- Need to consider both delivery orders
    - Usually results in a branch in the coherence FSMs
- Need to make sure messages can’t stick around lost
    - Every request needs an ack
    - Extra states to clean up messages
- Often, only one node knows how a race resolves
    - Might need to send messages to tell others what to do

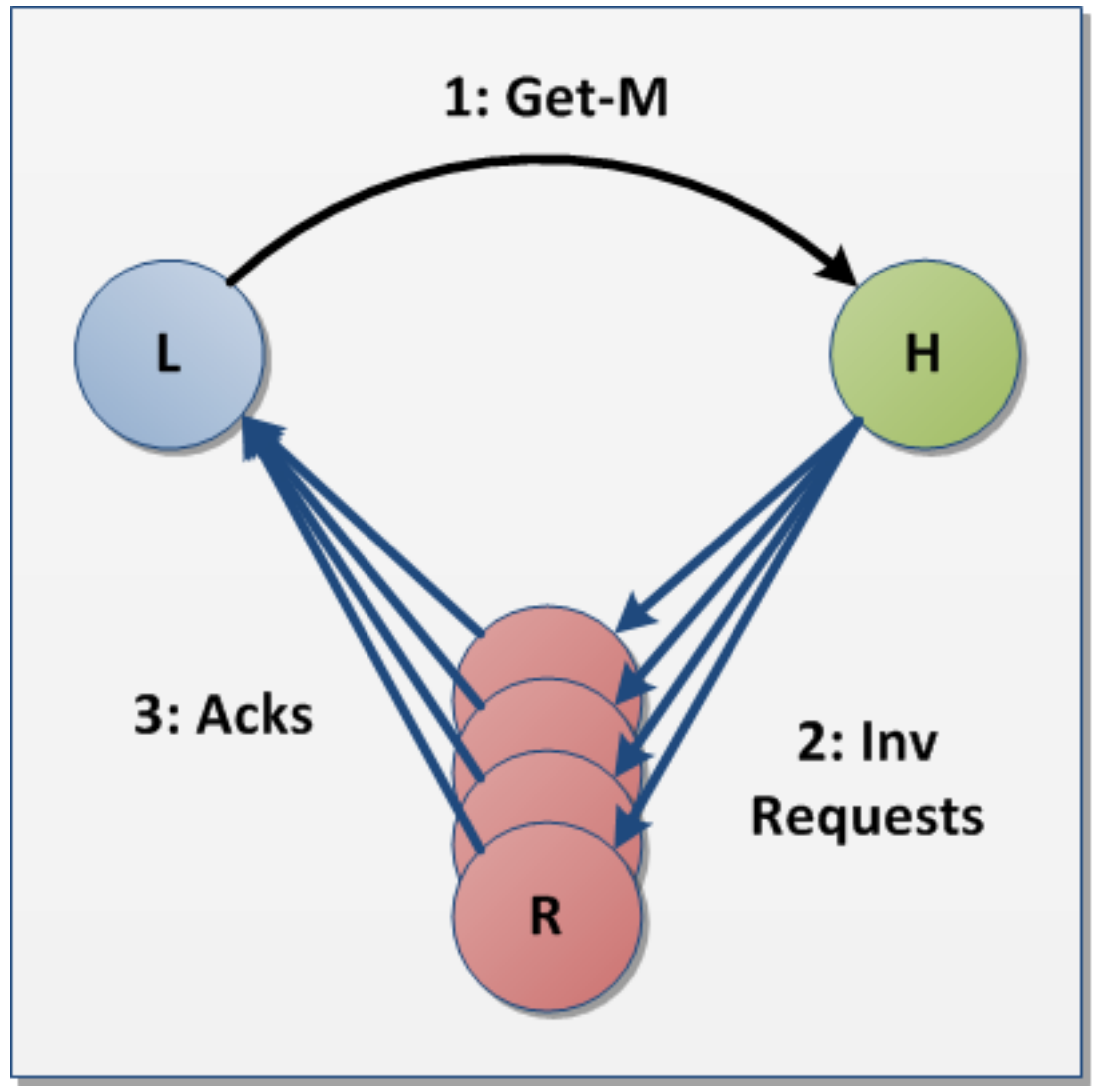
# CACHE COHERENCE PROTOCOL ISSUES (1)

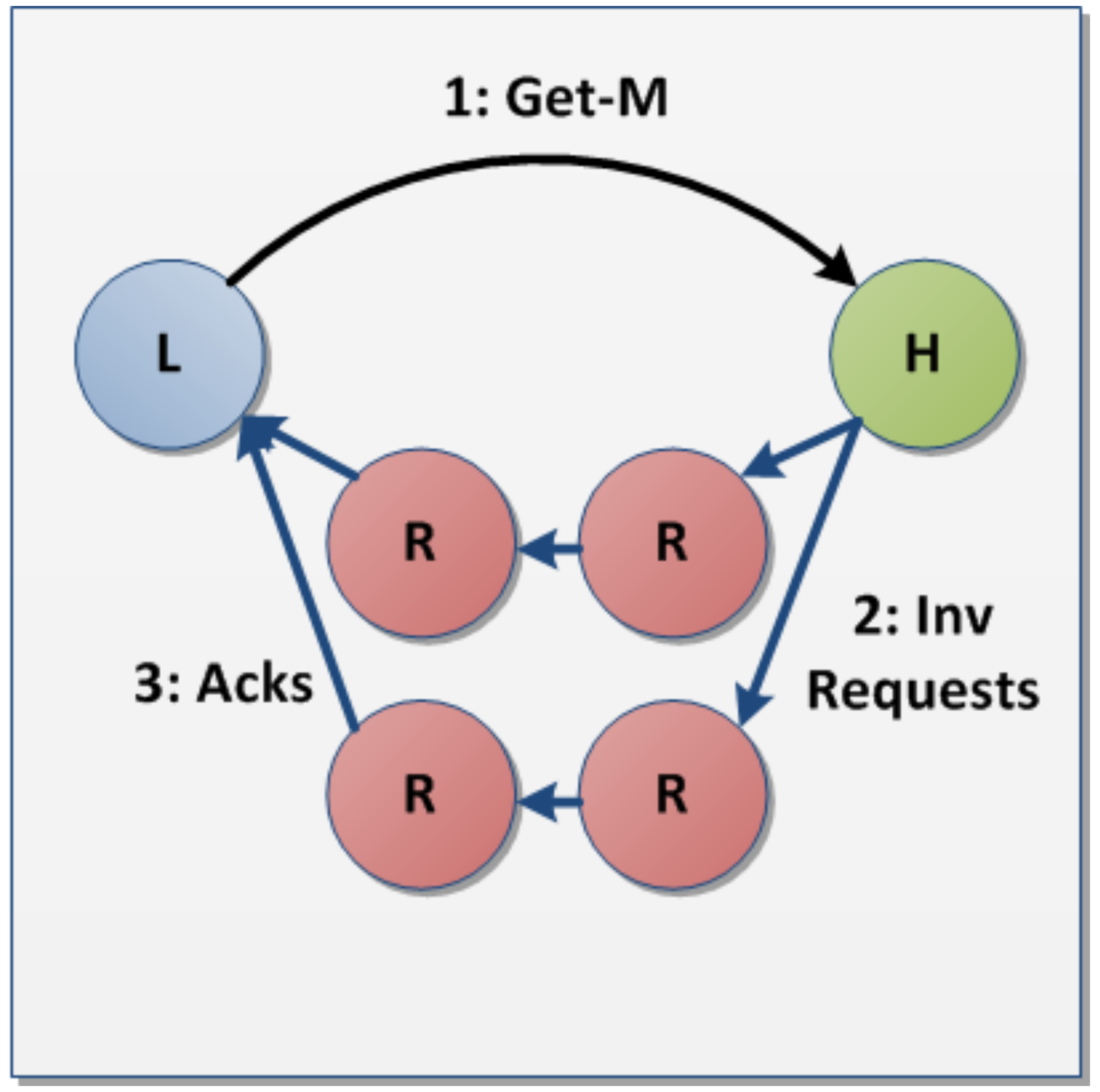
- NACKs are used to re-issue requests
  - E.g., when racing
- Does the protocol use negative acknowledgements (retries)?
  - NACKs are a bad idea for CC protocols (livelock, starvation, fairness)
  - Usually a consequence of protocol interaction

# CACHE COHERENCE PROTOCOL ISSUES (2)

Is the number of active messages (sent but unprocessed) for one transaction bound?

- Scalability issue – how much queue space is needed
- MSHR – Miss-Status Handling Registers
- Example: coarse-vector vs. cruise-missile invalidation





# CACHE COHERENCE PROTOCOL ISSUES (3)

Does the protocol require clean eviction notifications (Put-S)?

- (Silent evict) -> Get-M -> Inv ?

How/when is the directory accessed during transactions?

- Directory in main memory: latency issues

How to deal with transient states?

- Keep it in the directory: unlimited concurrency
- Keep it in a pending transaction buffer (e.g., transaction state register file): faster, but limited pending transactions
- Occupancy free: upon receiving an unsolicited request, can directory determine final state solely from current state?

# CACHE COHERENCE PROTOCOL ISSUES (4)

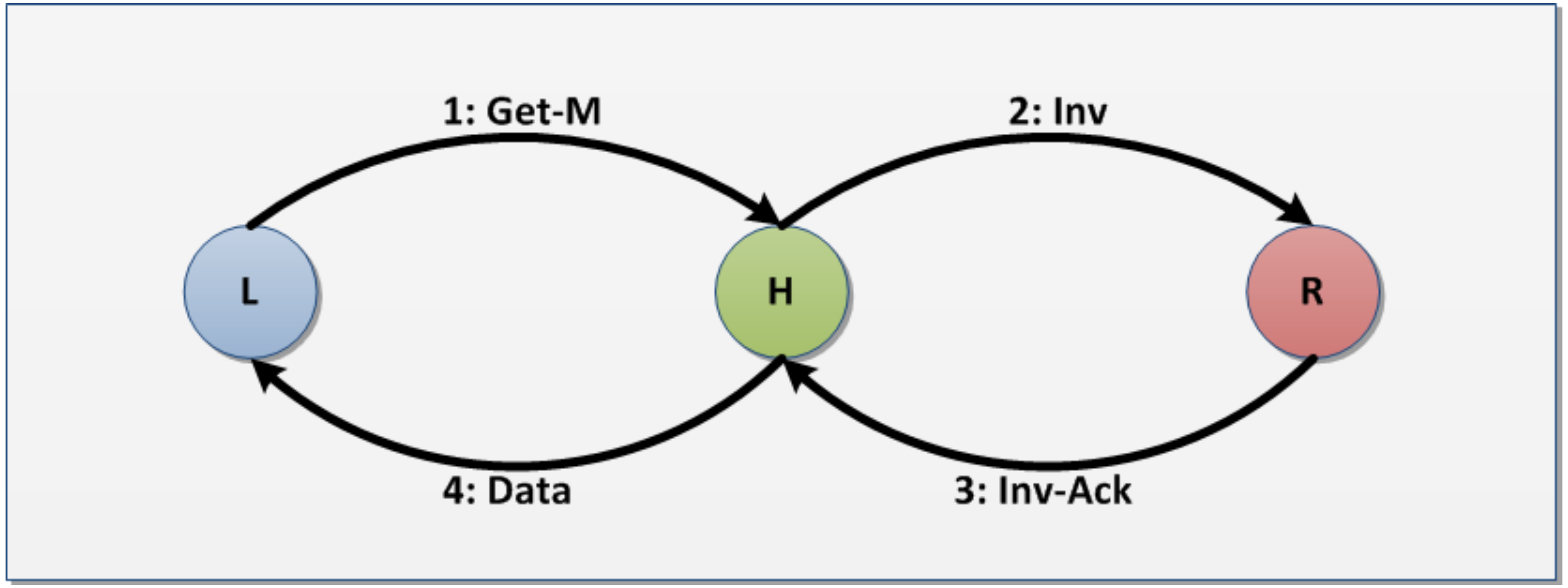
How many lanes are needed to avoid deadlocks?

- At least two to avoid protocol deadlocks

How to assign lanes to message types?

- Secondary (forced) requests may not be blocked by new requests
- Replies (completing a pending transaction) may not be blocked by new requests

More may be needed by IO, complex forwarding



# CACHE COHERENCE PROTOCOL ISSUES (5)

All messages should be acknowledged

- Requests elicit replies

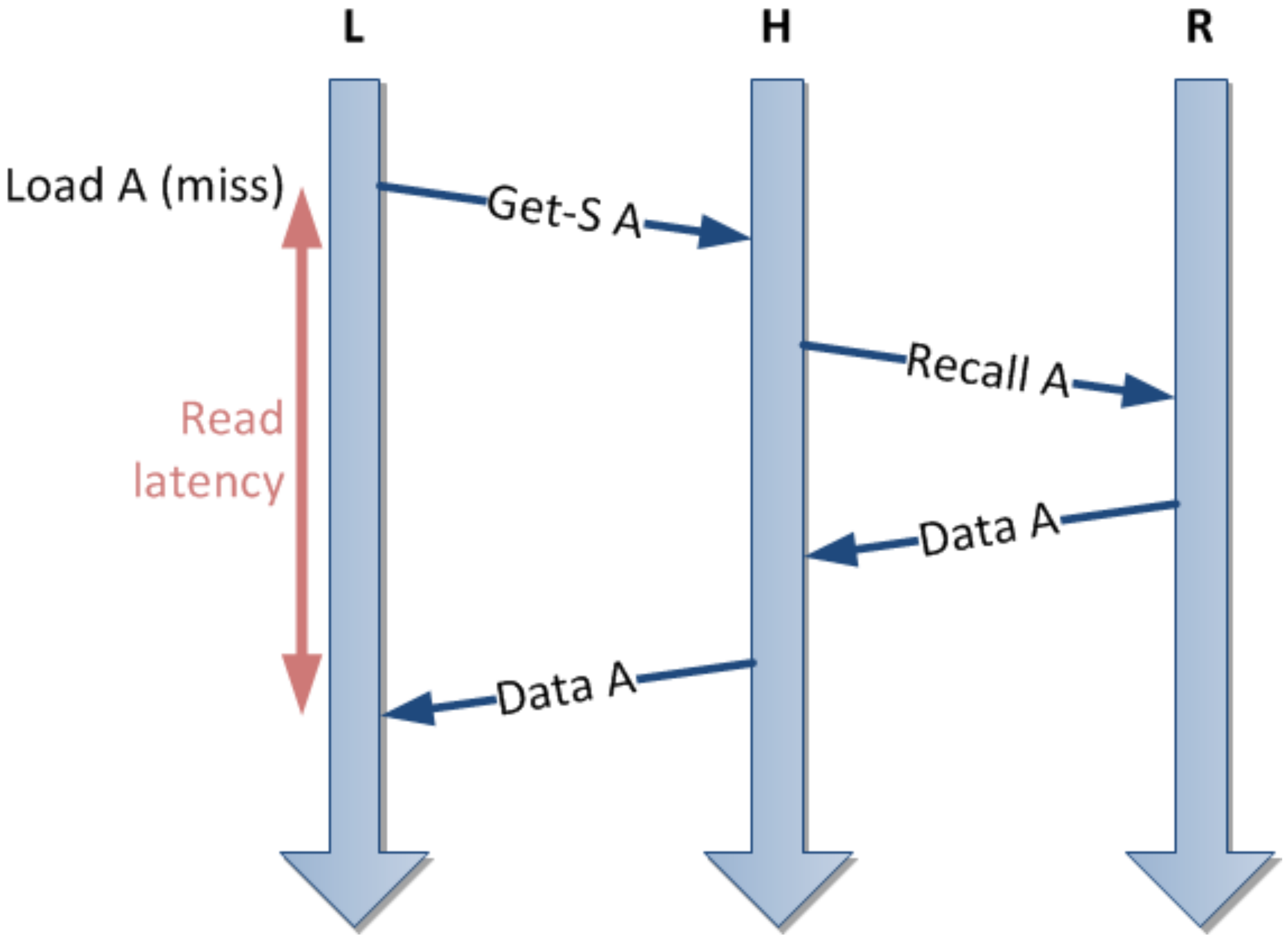
Maximum number of potential concurrent messages for one transaction should be small and constant

- I.e., independent of number of nodes in the system

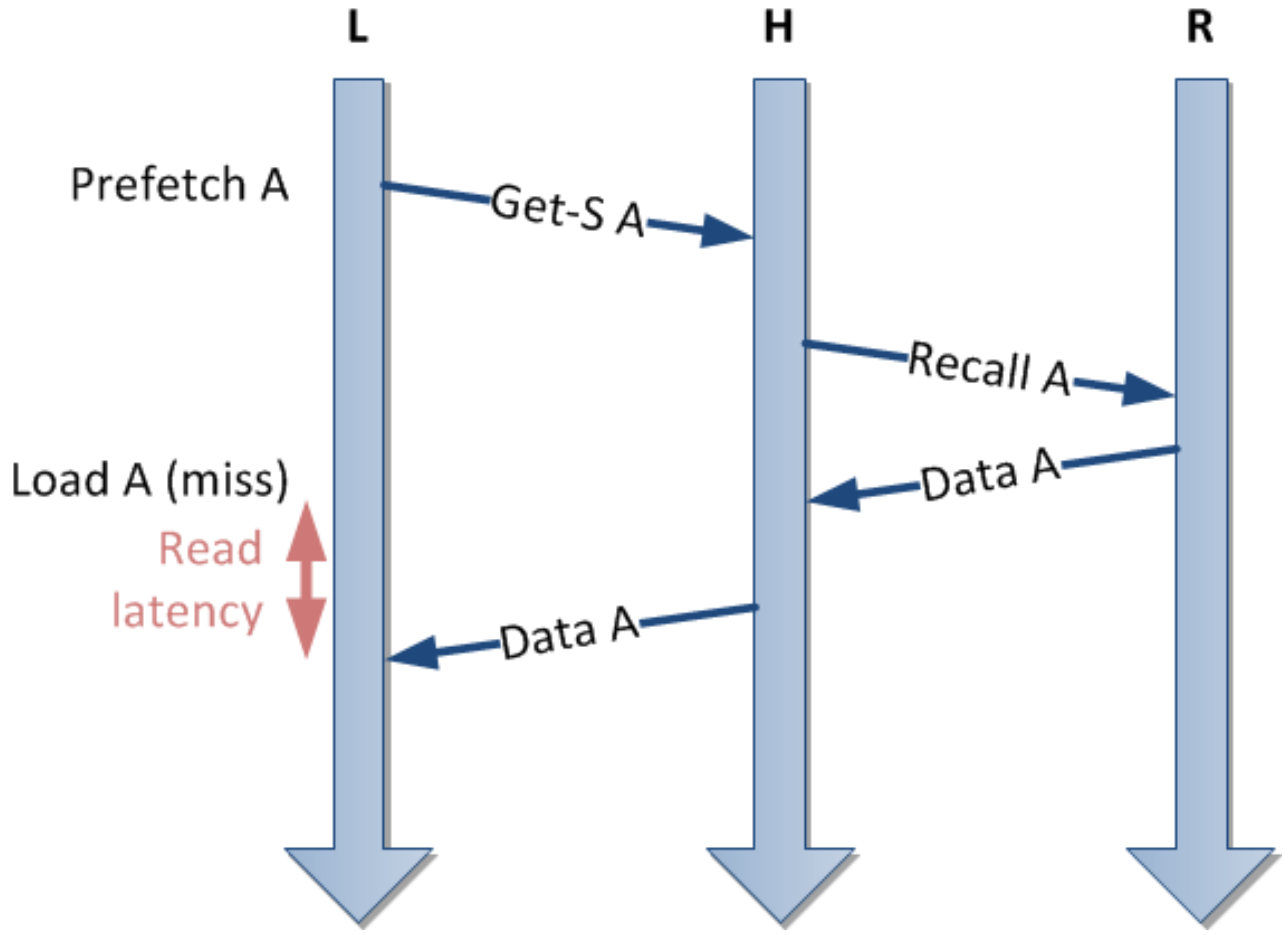
Minimize traffic, minimal interaction, low talk/silence ratio

Use context information to avoid NACKs

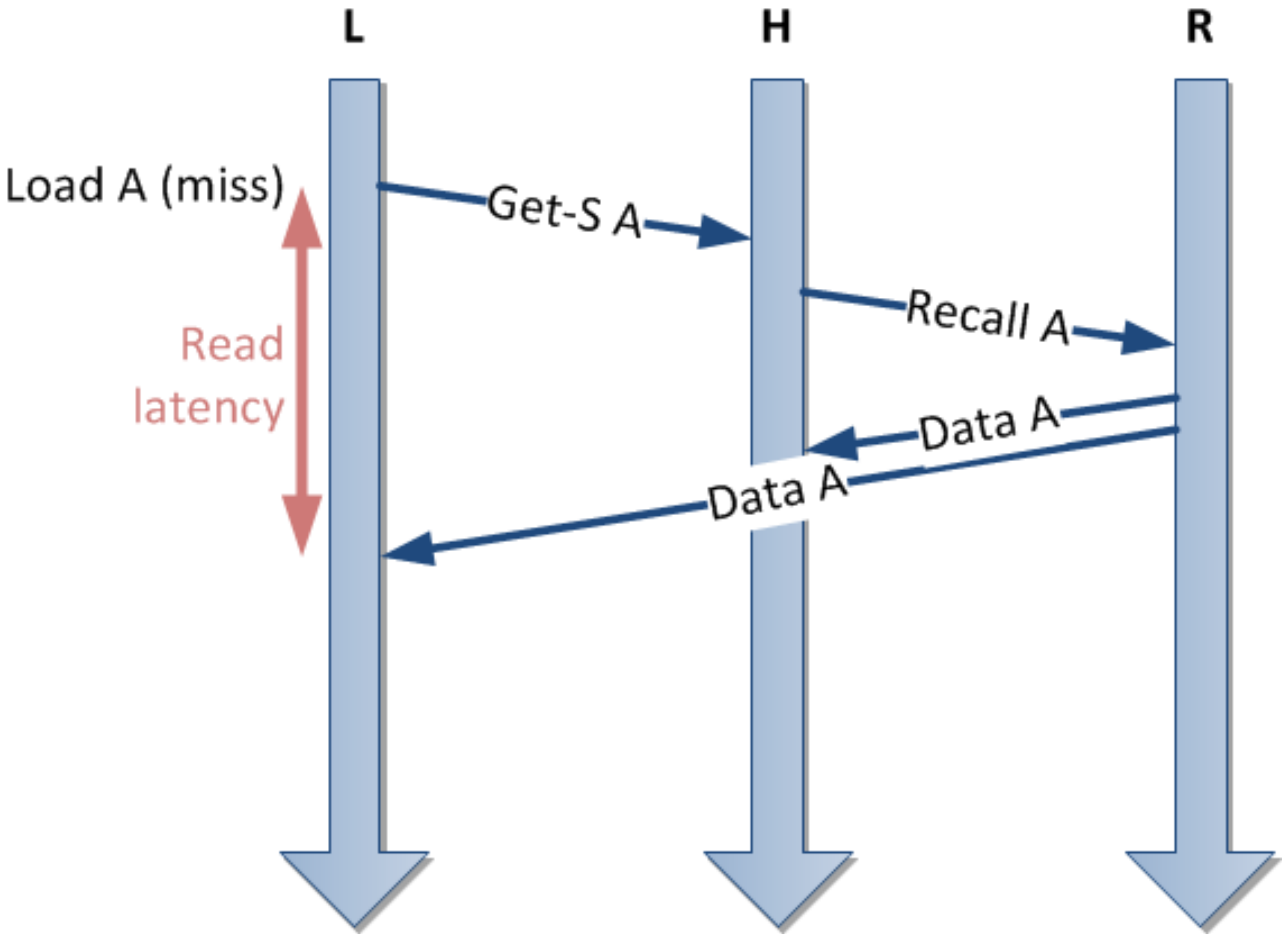
# OPTIMIZING COHERENCE PROTOCOLS



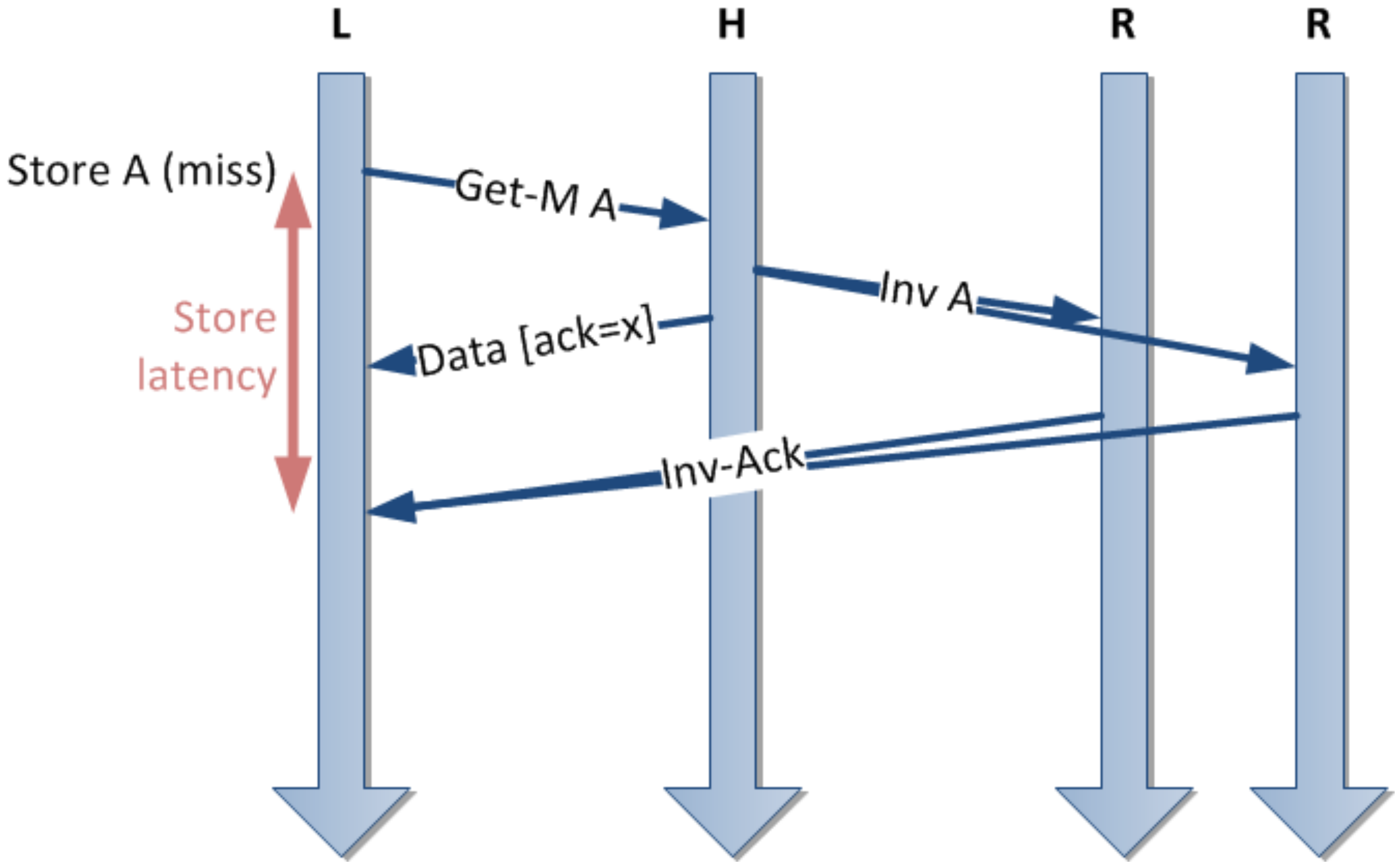
# OPTIMIZING COHERENCE PROTOCOLS: PREFETCHING



# OPTIMIZING COHERENCE PROTOCOLS: 3-HOP READS



# OPTIMIZING COHERENCE PROTOCOLS: 3-HOP WRITES



# OPTIMIZING COHERENCE PROTOCOLS: MIGRATORY SHARING

Each read/write pair results in read miss & upgrade miss

Coherence FSM can detect this pattern

- Detect via back-to-back read-upgrade sequences
- Transition to “migratory M” state
- Upon a read, invalidate current copy, pass in “migratory E” state

| Node 1 | Node 2 | Node 3 |
|---|---|---|
| Read X | | |
| Write X | | |
| | | Read X |
| | | Write X |
| | Read X | |
| | Write X | |
| | | |
| | | |

# OPTIMIZING COHERENCE PROTOCOLS: PRODUCER/CONSUMER

Upon read miss, downgrade instead of invalidate

- Detect because there are 2+ readers between writes

More sophisticated optimizations

- Keep track of prior readers
- Forward data to all readers upon downgrade

| Node 1 | Node 2 | Node 3 |
|---|---|---|
| Read X | | |
| Write X | | |
| | | Read X |
| | Read X | |
| Read X | | |
| Write X | | |
| | | Read X |
| | Read X | |

# OPTIMIZING COHERENCE PROTOCOLS: SHORTCOMINGS

Optimizations have to be included in coherence FSM

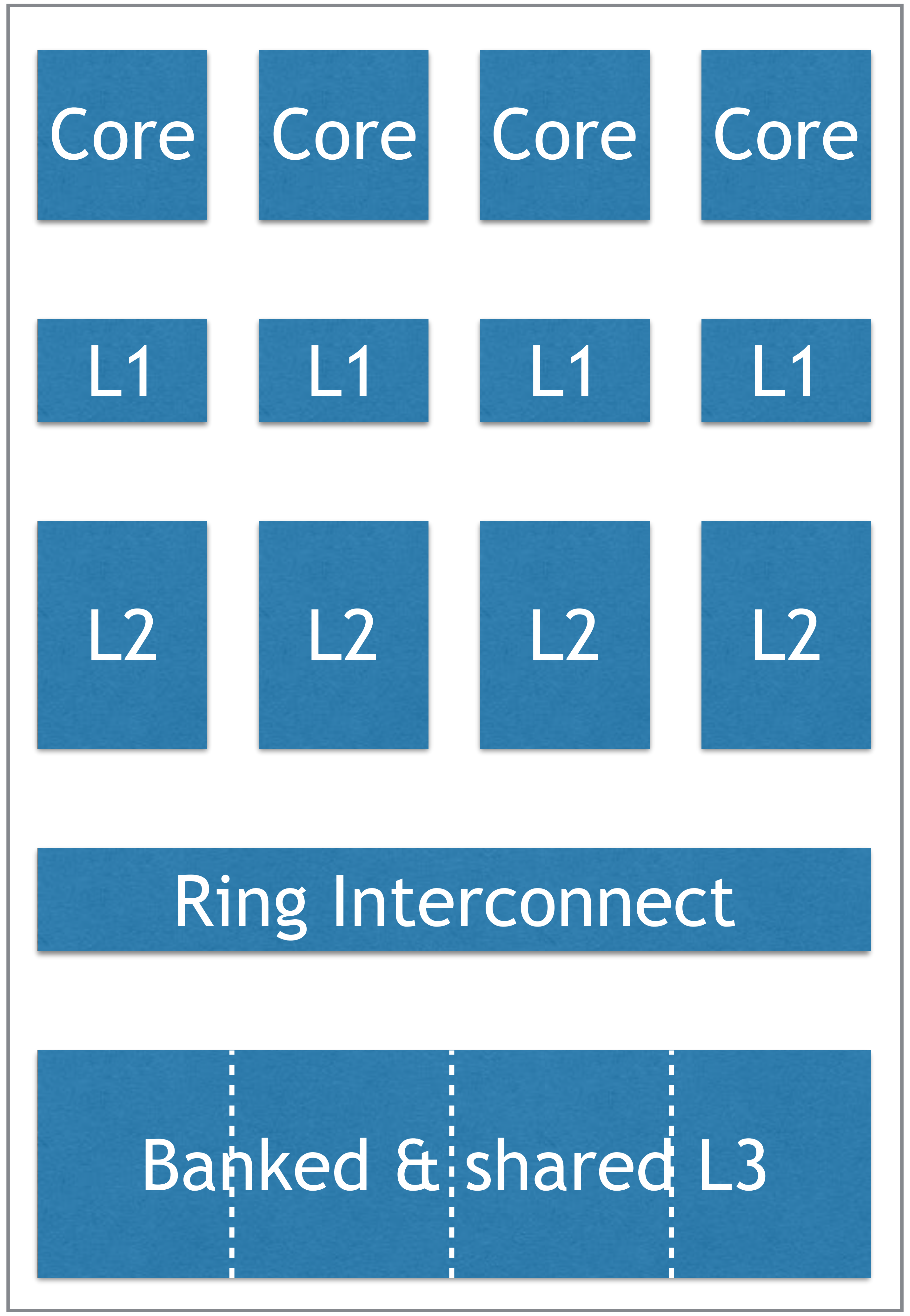
- Complex! Adds transitions, states, races...
- Hard to verify even for basic protocols
- Can target only simple sharing patterns
- Can learn only one pattern per address at a time

Each optimization contributes to state explosion

# SUMMARY

- No free lunch: scalability comes at a certain price
- Latency of paramount importance
  - Large-scale NUMA systems already suffer a lot from latency issues
- Various optimizations possible, but often negative impact on latency due to complexity
- Number of sharers is typically small => limited multicast & limited number of directory entries
- Main challenges
  - Reducing the overhead of directory storage
  - Outstanding messages and latency
- However/thus, many machines today employ directories
  - Examples: AMD’s probe filter (HT), Intel’s snoop filter (QPI), SGI Altix, NUMASCALE (successor of SCI), ...

# EXAMPLE: INTEL CORE I7 PROCESSOR



Centralized directory for all blocks in L3

Inclusion property is important

Directory maintains list of L2 caches containing block

=> Multicast of coherence messages to L2s that contain block

Instead of broadcast