Caches, Coherence, and Consistency (and Consensus)

Dan Ports, CSEP 552

Caching

- Simple idea: keep a duplicate copy of data somewhere faster
- Challenge: how do we keep the cached copy consistent with the master?
- What does it even mean to do that?
  - ideally, user/app couldn't tell the cache was even there
- Today will be about answering those questions

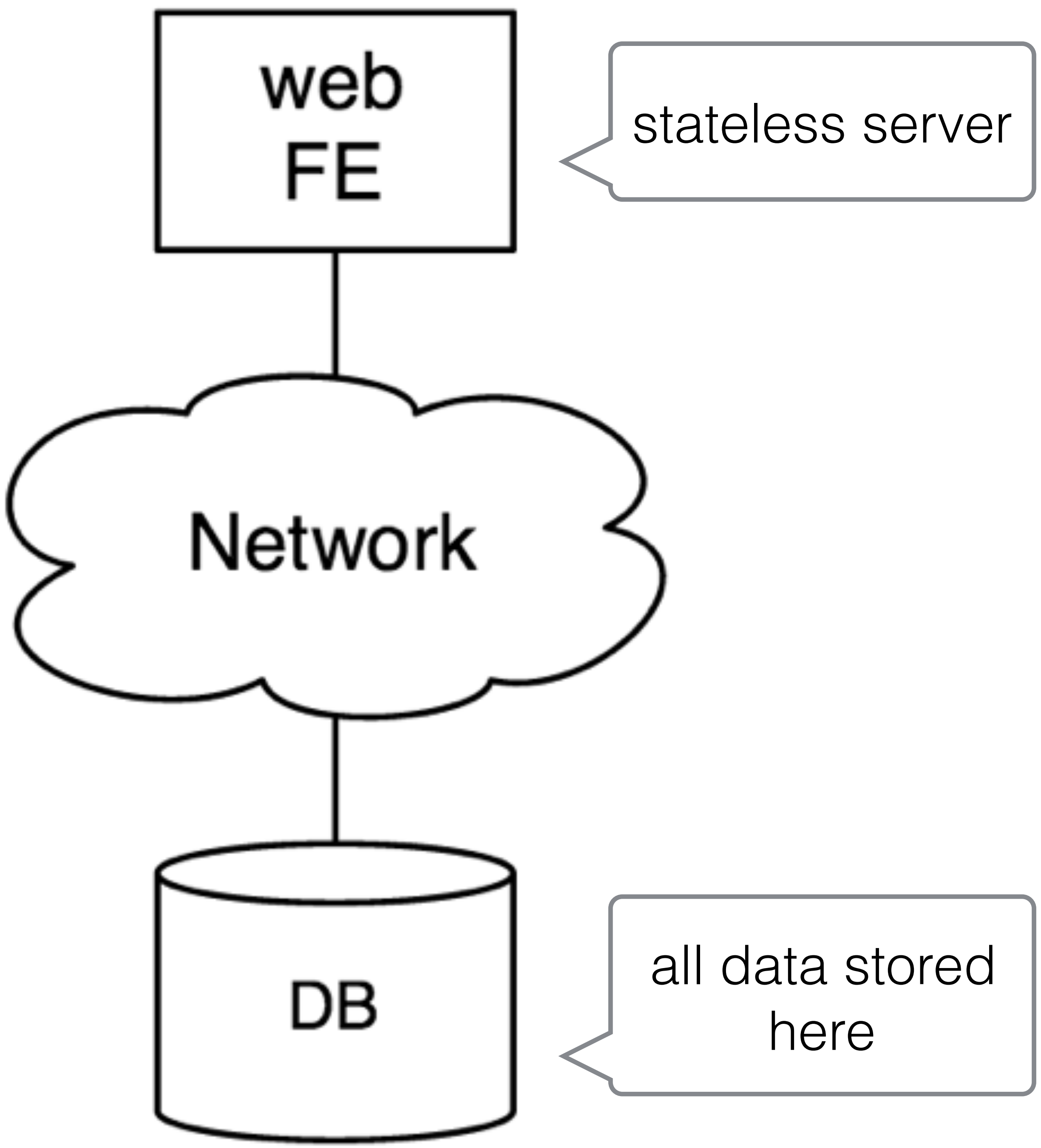
Why do we want caching?

- Reduce load on a bottleneck service
(exploit locality)

- Better latency
(cache is more conveniently located & hopefully faster)

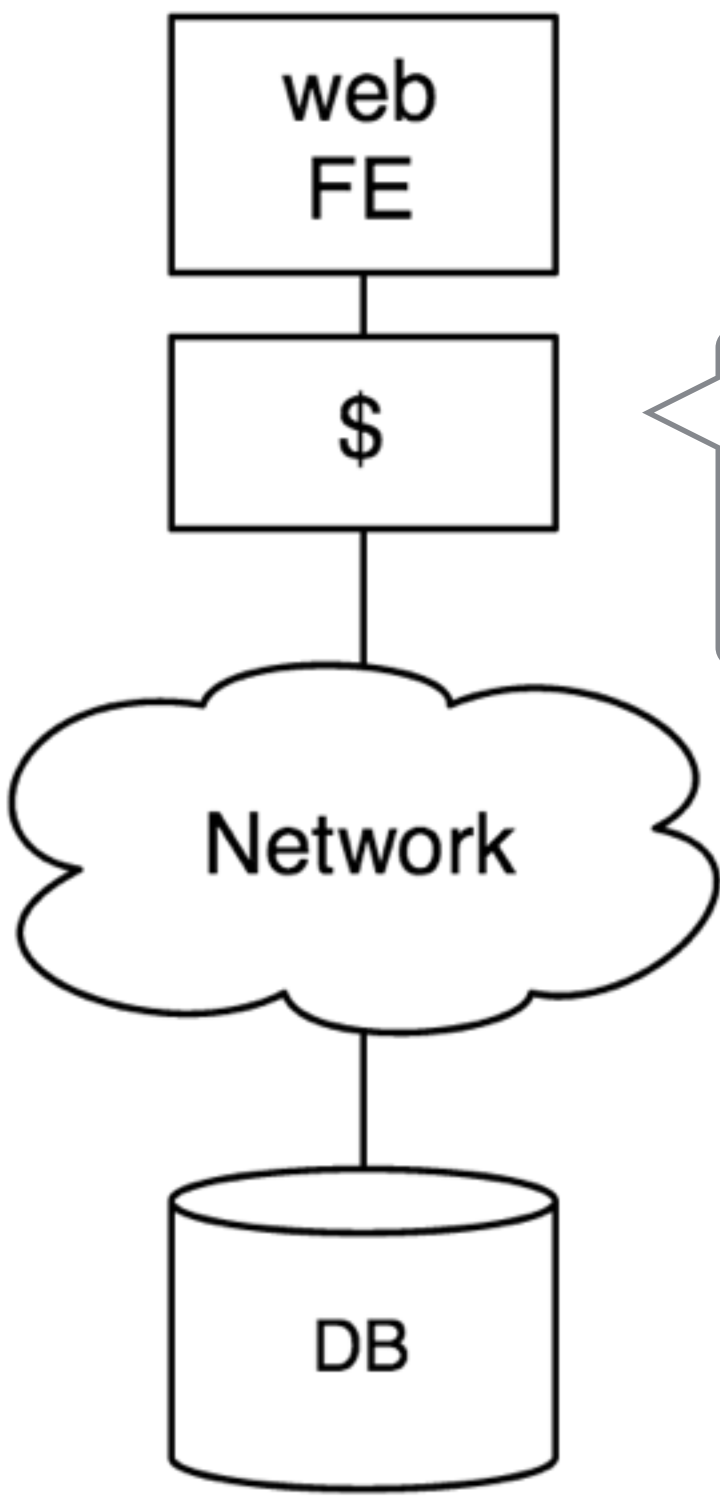
- High-level view:
caching: move data to where we want to use it
vs RPC: move computation to where the data is

Web Service Architecture



Adding a Cache

Idea: store recent DB results
in the cache so we can reuse them



cache on FE
machine
(in RAM)

Cache details

- What do we do with writes?
  - update the cache first, then update the database
  - synchronously (write-through): safe but slow
  - asynchronously (write-back): fast but not crash-safe
- What do we do if the cache runs out of space?
  - throw data away (e.g., least-recently-used)

Cache semantics

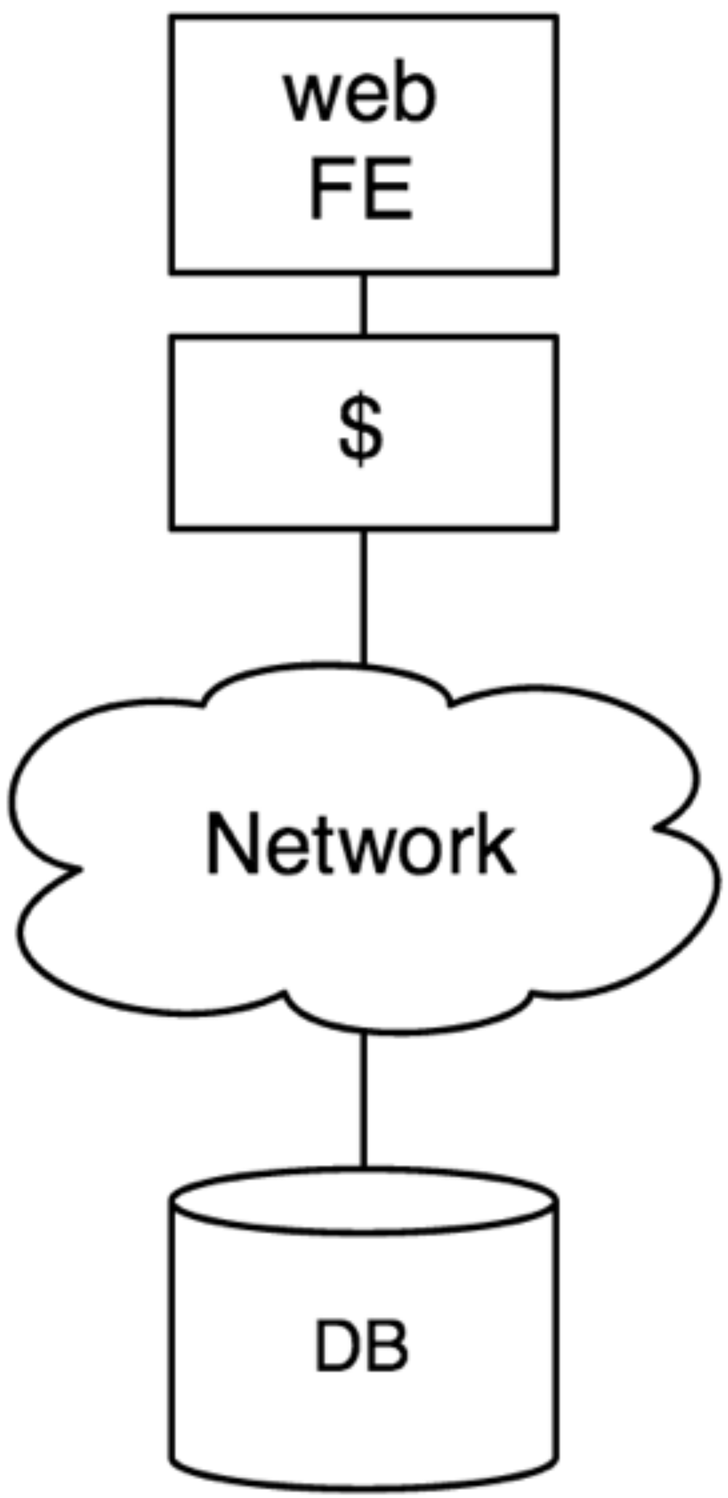
- Does this cache behave the way we'd like it to?
- i.e., can an application tell that the cache is there?

Terminology

- **Coherence**: the value returned by a read operation is always the value most recently written to that object

- Unfortunately the terminology is inconsistent
  - Coherence: properties about the behavior of multiple reads/writes to **same** object
  - Consistency: properties about behavior of multiple reads/writes to **different** object

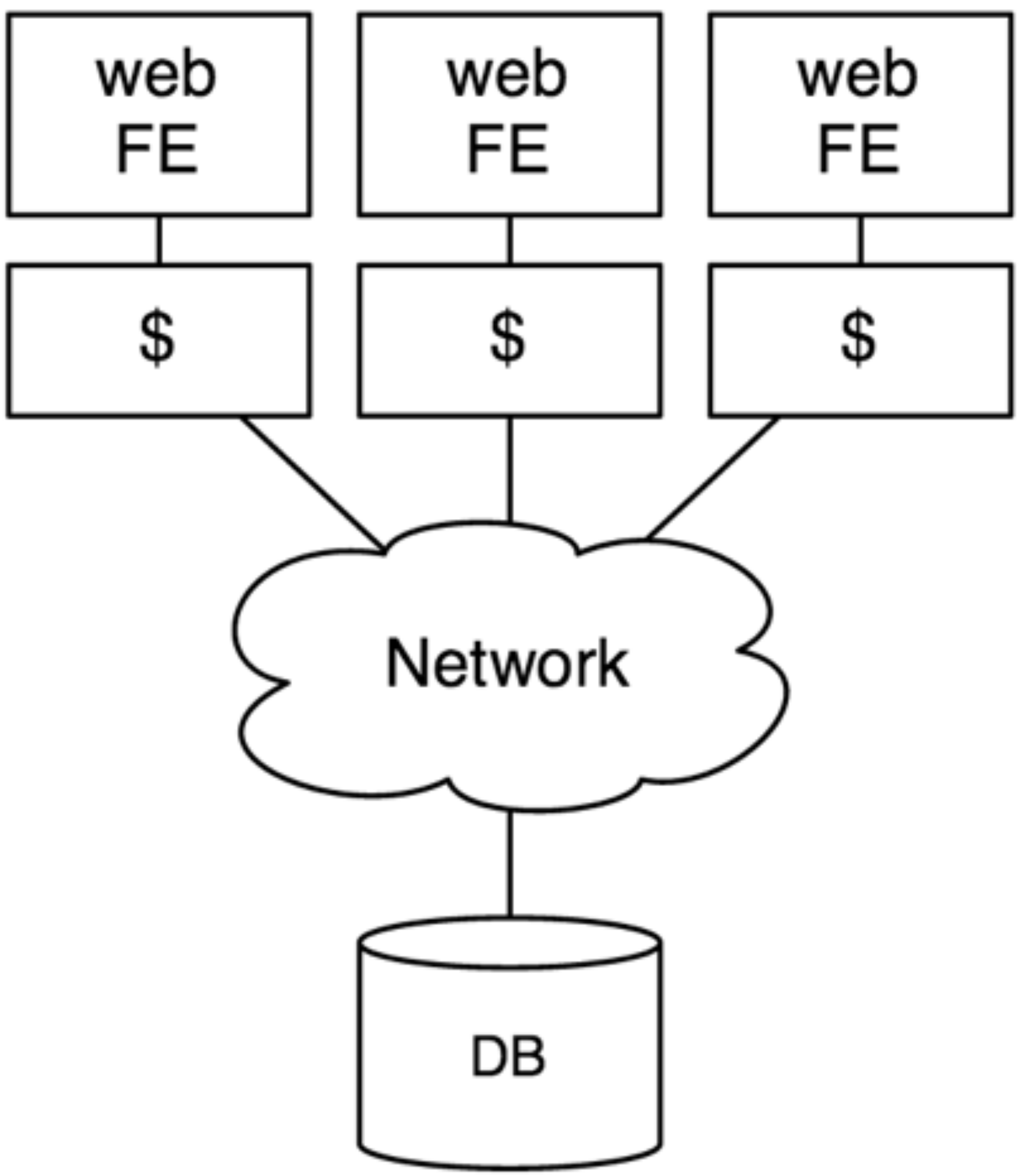
Cache coherence



Is this cache coherent?

Yes!
All writes go to cache first &
all reads check there first
=> always see latest write

Scaling up



Multiple front-end servers each with its own cache

Suppose we use the same protocol as before:

- update local cache
  - then update DB synchronously

Is the cache coherent now?

What are other systems that uses caches?

- Just about everything…
  - web browsers
  - NFS
  - DNS
  - processors!
    (lots of terminology comes from here)

How could we fix this?

Idea: invalidations

- Protocol: on a write, update the DB *and* send invalidations to other caches
- Which order should we do these in?
- Does that provide coherence?

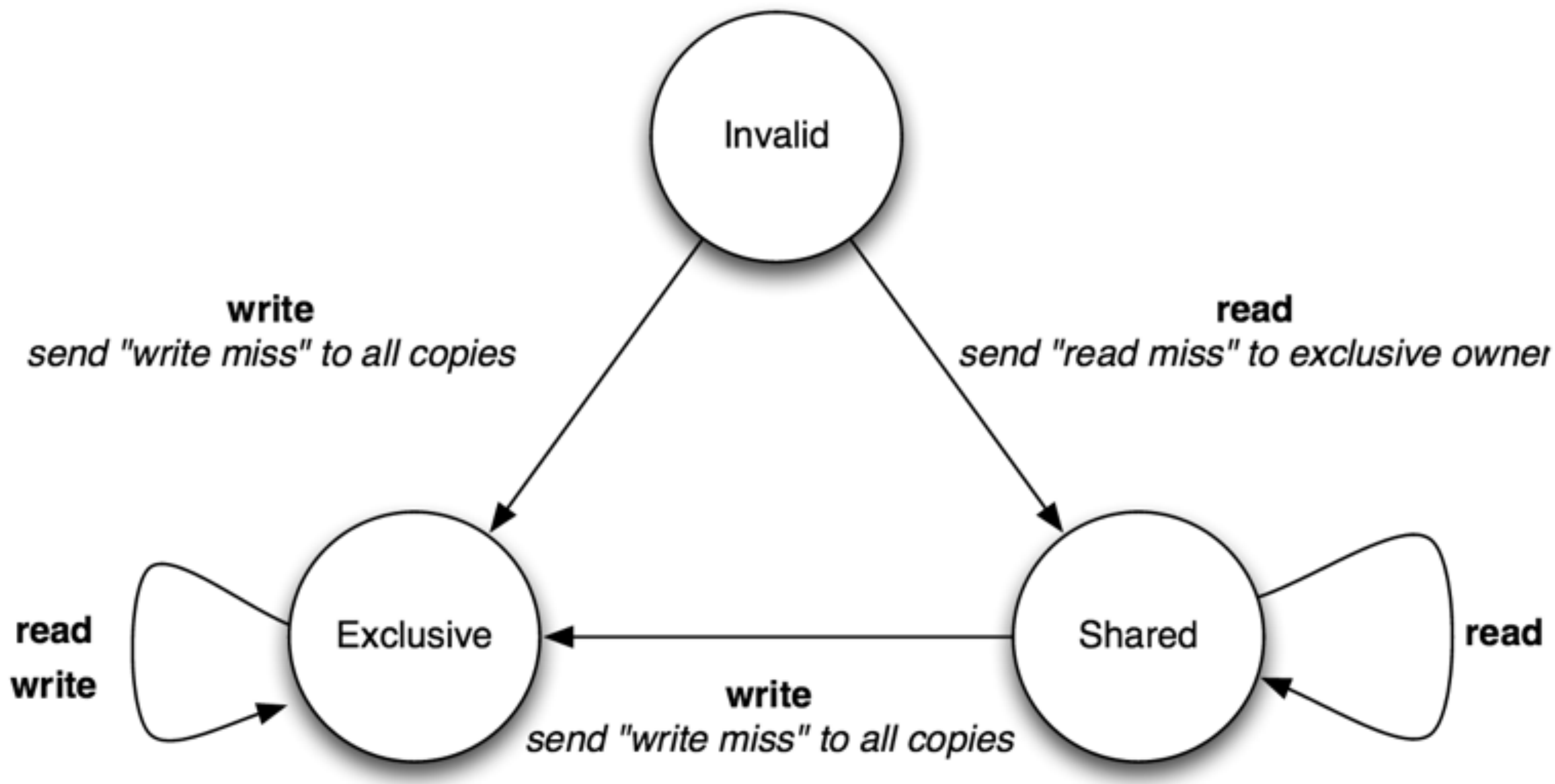
Idea: add locking

- When A writes X:
  - A notifies all caches and DB not to allow access to X, waits for acknowledgments
  - A updates DB, updates caches, waits for acks
  - A releases the lock
- Does this provide coherence?
- Is this efficient?

Better idea: exclusive ownership

- Basic idea: at most one cache is allowed to have a dirty (modified) copy at any time
- Each entry on each cache is in one of three states:
    - invalid (no cached data)
    - shared (read/only)
    - exclusive (read/write)
- X has exclusive access => all other caches invalid

Better idea: exclusive ownership



State transitions

- How does one cache transition to exclusive state?
  - send write-miss RPC to everyone else,
    wait for responses
  - upon receiving write-miss:
    if holding shared, go to invalid
    if holding exclusive, write back and go to invalid
- Does this protocol work?
  - need to be careful about two caches concurrently
    trying to get exclusive state (locking)

Performance

- Single node can now repeatedly write object w/o coordination
- Contention: concurrent reads/writes to same object
  - cached item bounces back and forth between caches
- Need to keep track of which caches have shared/exclusive copies (distributed state)
- **Performance costs are fundamental to providing coherence!**

What if we wanted something cheaper?

- Maybe OK to see an old value as long as it's not more than 15 seconds out of date?
- Maybe OK to see an old value, as long as it's not before our last update?
- Maybe OK to see an old value if the last update was logically concurrent?
- Infinite possibilities for defining weak consistency/ coherence models!

Coherence in NFS

- Design choice: don't want server to keep track of which clients have cached data
- Client periodically checks if cached copy is up to date
- Only real guarantees:
  dirty cache blocks flushed on close(),
  open() invalidates any old cached blocks
  ("close-to-open consistency")

Coherence vs Consistency

- Coherence: properties about the behavior of multiple reads/writes to **same** object
- Consistency: properties about behavior of multiple reads/writes to **different** object

- When weakening our semantics, consistency properties start to matter a lot…

Consistency Example

```
node0:
  v0 = f0();
  done0 = true;

node1:
  while(done0 == false)
    ;
  v1 = f1(v0);
  done1 = true;

node2:
  while(done1 == false)
    ;
  v2 = f2(v0, v1);
```

intent:
node2 executes f2
w/ results from
node0 and node1

node2 waits for node1,
so should wait for
node0 too

Is this guaranteed?

Memory Model

- Behavior of this code depends on memory model
  - linearizable: behaves like a single system
  - serializable / sequentially consistent:
    behaves like a single system *to programs running on it*
  - eventually consistent: if no more updates, all nodes
    eventually have the same state. Before that… ?
  - weakly consistent:
    doesn't behave like a single system

Linearizability

- Strongest model
- A memory system is linearizable if:
  every processor sees updates in the same order that they actually happened in real time
  - i.e., every read sees the result of the most recent write that finished before the read started

Is this linearizable?

```
P1:             W(x)1
P2: R(x)0                 R(x)1
```

Is this linearizable?

```
P1: W(x)1
P2:                R(x)2  R(x)2
P3:       W(x)2
```

Is this linearizable?

```
P1: W(x)1
P2:                 R(x)1  R(x)1
P3:         W(x)2
```

Linearizability is restrictive

- Need to make sure that caches are invalidated before operation completes
- Even though this might not have been necessary
- P2 needed to see effects of P3's update, even though no explicit communication between them (even if logically concurrent!)
- Why is this restriction useful?

Serializability (Sequential Consistency)

- Appears as though all operations from all processors were executed in a sequential order; reads see result of previous write in that order
- Operations by each individual processor appear in that sequence in program order (i.e., in the order executed on that processor)
- Slightly less strong than linearizability: no real time constraint

Is this serializable?

```
P1:           W(x)1
P2: R(x)0                R(x)1
```

Is this serializable?

```
P1: W(x)1
P2:                R(x)1  R(x)1
P3:        W(x)2
```

```
Yes - valid order:
W(x)1 R(x)1 R(x)1 W(x)2
```

Implementing sequential consistency

- Requirement 1: *Program order requirement*
  - each process must ensure that its previous memory op is complete before starting the next in program order
  - cache systems: write must invalidate all cached copies
- Requirement 2: *Write atomicity*
  - Writes to the same location must be serialized, i.e., become visible to all processors in same order
  - value of write can't be returned by any read
    until write completes

Causal consistency

- A read returns a causally consistent version of the data
- if A receives message M from B, reads will return all updates that B made before sending M
  - i.e., will see all writes that happens-before your read

Causal vs sequential consistency

- Is causal consistency weaker than sequential consistency?
  - Yes - don’t need to decide an order for causally unrelated writes!
- Why is this useful?
  - can build a system that doesn’t coordinate on causally unrelated writes — fast!
  - if two nodes are unable to communicate with each other, can still ensure causal consistency but not sequential

Is this causally consistent?

```
P1: W(x)1 R(y)0
P2:                    R(y)2  R(x)0
P3:               W(y)2
```

Is this causally consistent?

```
P1: W(x)1
P2:                    R(y)2  R(x)0
P3:       R(x)1 W(y)2
```

Weaker consistency levels

- Weak consistency: anything goes
- Eventual consistency: if all writes stop, system eventually converges to a consistent state where read(x) will always return same value
  - until then… anything goes
- Eventual consistency is popular:
  NoSQL databases (Redis, Cassandra, etc). Why?

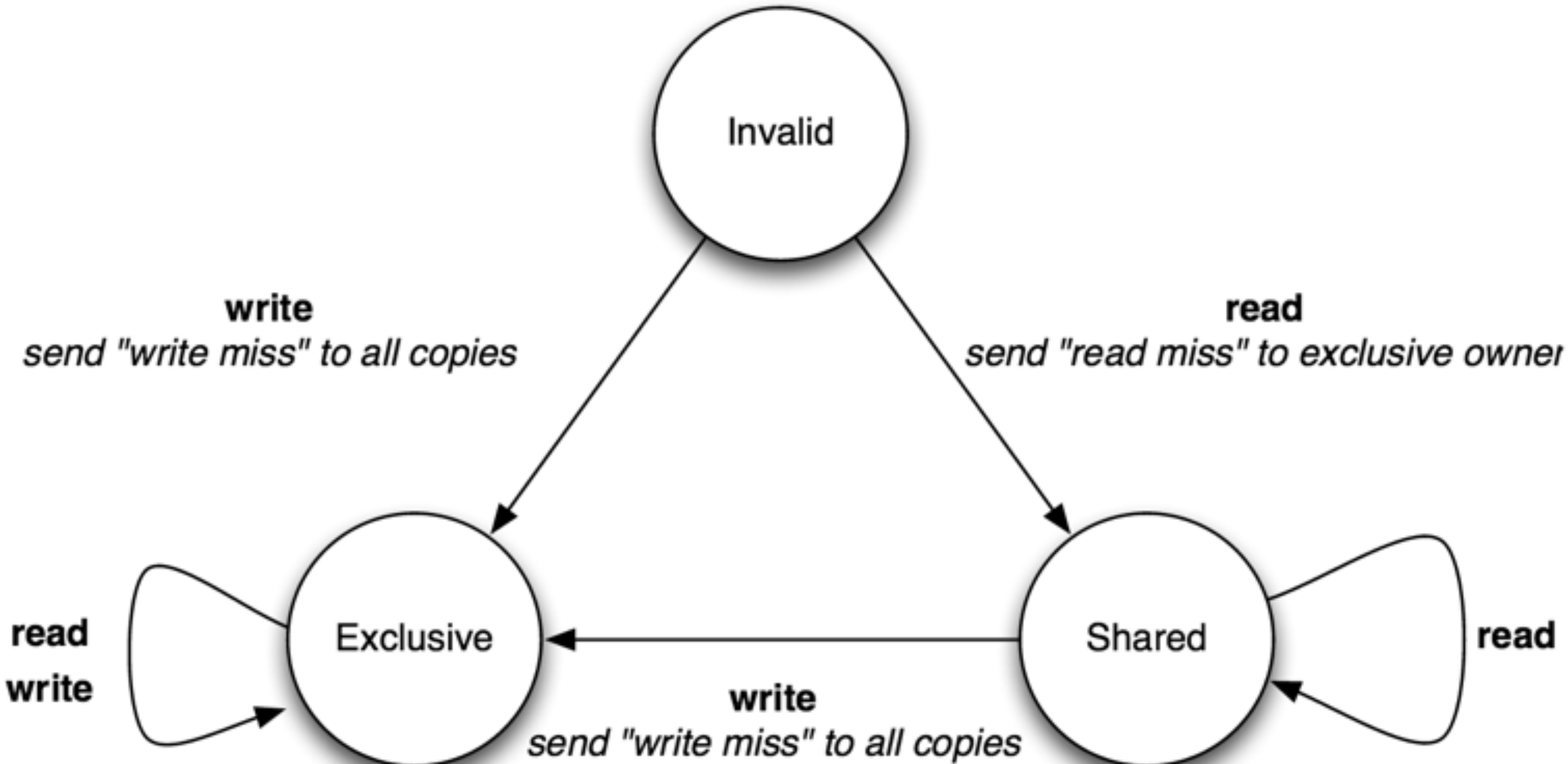
Ivy DSM

- Goal: distributed shared memory
  - a runtime environment where many machines share memory
  - make a distributed system look like a giant multiprocessor machine
- Why would we want this?

Ivy approach

- Use hardware virtual memory / protection to make DSM transparent to application
- Recall virtual memory:
  - OS installs mappings:
    virtual address -> {physical addr, permissions}
    (permissions = read/write, read-only, none)
  - App violates permissions => trap to OS
- Here, exploit this to fetch pages remotely
  & run cache coherence protocol

Ivy protocol



Granularity of coherence

- In hardware shared memory:
  usually one cache line (~64 bytes)
- What does Ivy use?
- Why the difference?
- What are the tradeoffs involved?

Ivy semantics

- What memory model does Ivy provide?
- Coherence of individual memory locations?
- What about consistency?
  Is it sequentially consistent?

Implementing sequential consistency

- Requirement 1: *Program order requirement*
  - each process must ensure that its previous memory op is complete before starting the next in program order
  - cache systems: write must invalidate all cached copies
- Requirement 2: *Write atomicity*
  - Writes to the same location must be serialized, i.e., become visible to all processors in same order
  - value of write can't be returned by any read
    until write completes

Design options

Table I. Spectrum of Solutions to the Memory Coherence Problem

Page synchronization method	Page ownership strategy			
		Dynamic		
			Distributed manager	
	Fixed	Centralized manager	Fixed	Dynamic
Invalidation	Not allowed	Okay	Good	Good
Write-broadcast	Very expensive	Very expensive	Very expensive	Very expensive

Performance

- What performance gain would we hope for?
  N nodes => N * single node throughput
- Why wouldn't we achieve this?

Performance

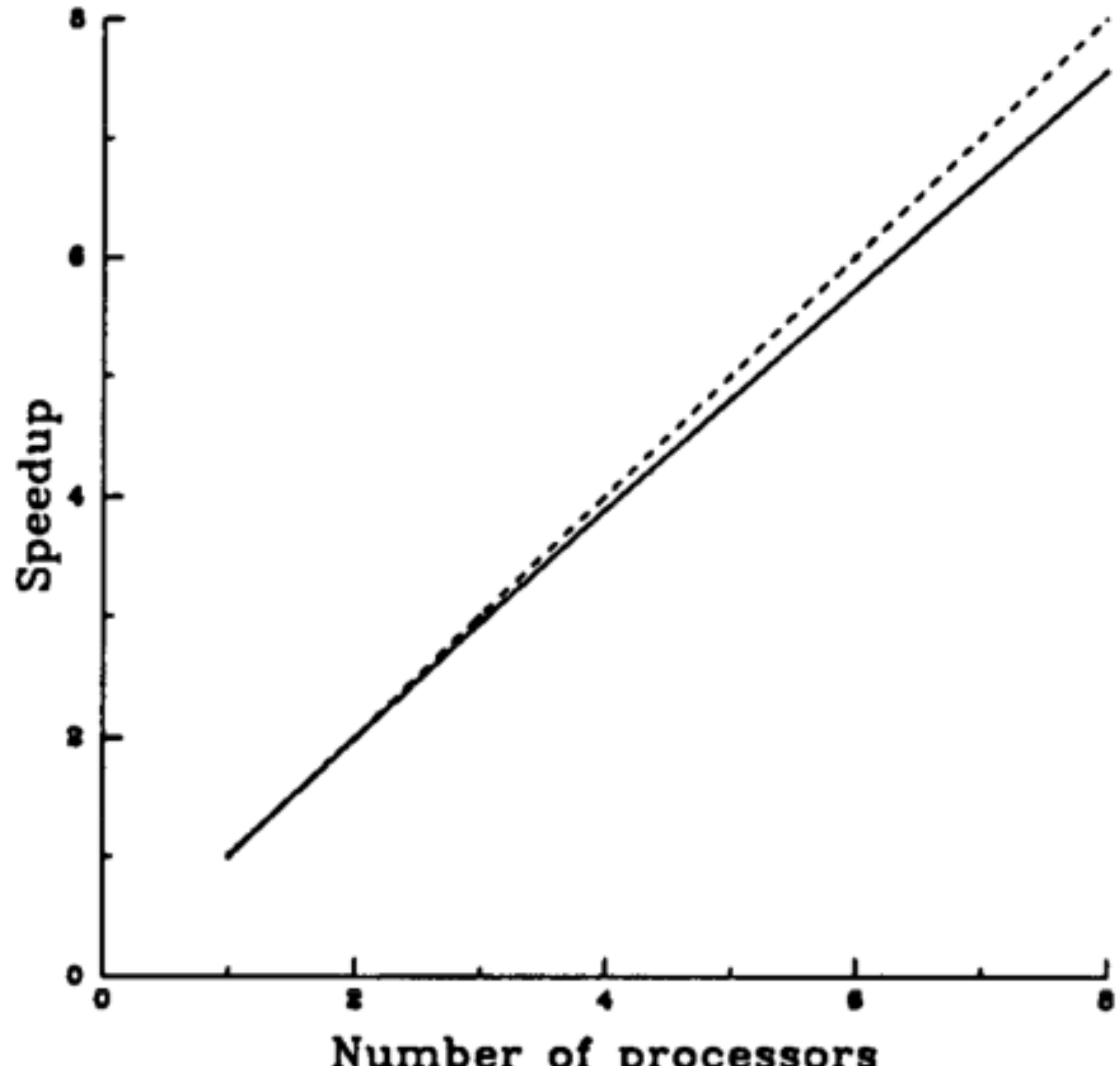


Fig. 10. Speedup of the matrix multiplication program.

Performance

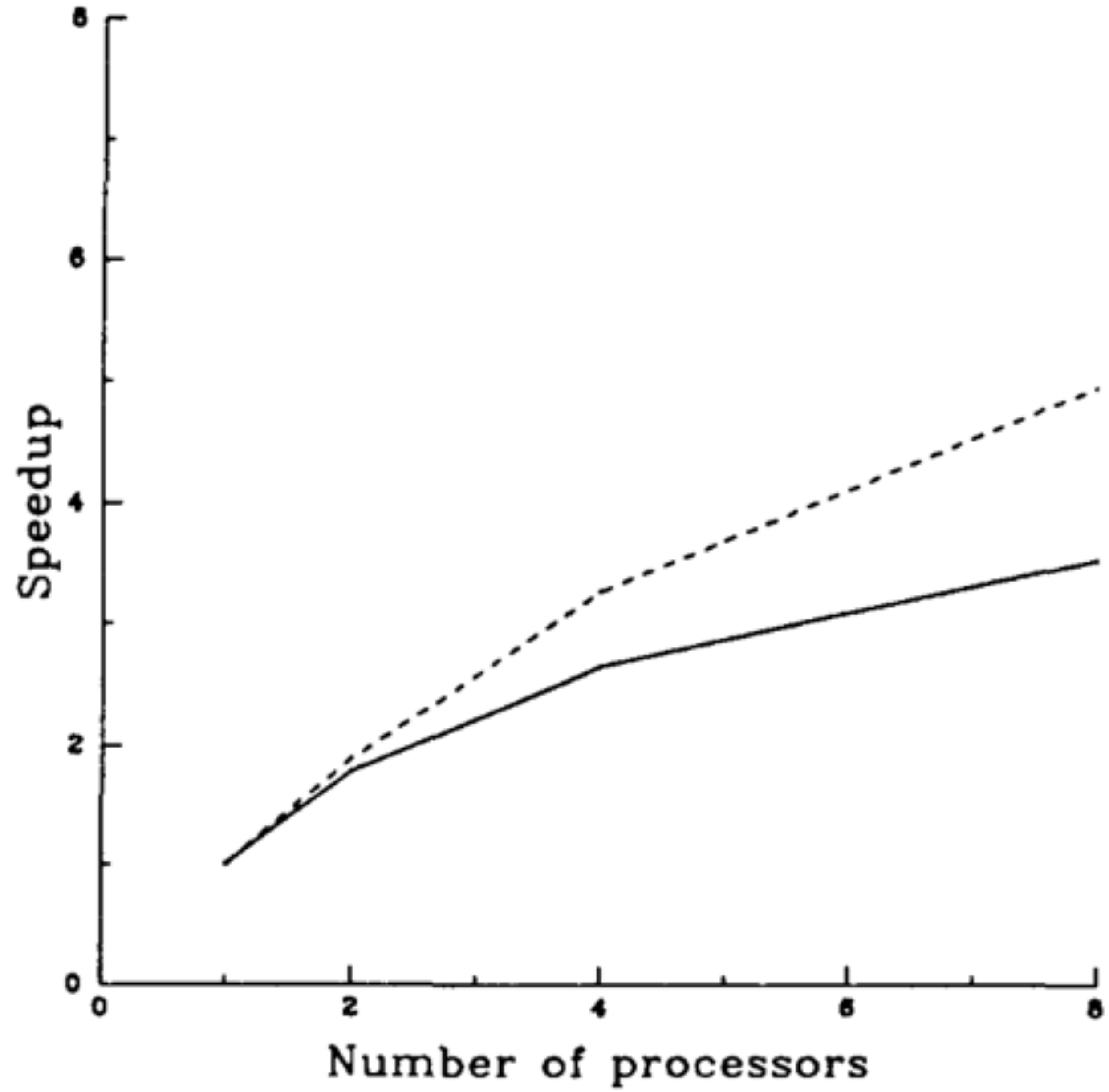


Fig. 8. Speedup of the merge-split sort.

Discussion

- Should we use DSM instead of message passing?
- Does DSM scale?
- Would it make sense to provide weaker consistency in DSM?

Intro to Consensus

- Fundamental problem in distributed systems:
  get a group of nodes to agree on a value
  even though some of them might fail

- Lots of problems ultimately boil down to consensus

- Lab 3 uses consensus for a reliable replicated state machine

- Next week: consensus algorithms -
  Paxos & Viewstamped Replication

Consensus Problem

- Multiple processes, each starting with an input
- Processes run a consensus protocol,
  then output a chosen value once it's complete
- **Safety** requirement:
  - **consistency**: all non-faulty processes output the same value
  - **validity**: that value was proposed by some node
    (i.e., can't just choose 0!)
- **Termination**:
  eventually all non-faulty processes output a value

System model

- **Assumptions about the world:**
- Asynchronous network
  - messages can be delayed indefinitely
  - but messages that are repeatedly sent will eventually be received
- Some processes can crash
  - just stop executing the protocol

FLP Result

- **No deterministic consensus protocol guarantees both safety and termination in an asynchronous network where one process can crash!**

Warning:
handwaving imminent!

FLP Intuition

- Suppose process A sends a message to process B but hasn't gotten a reply back (e.g., after retrying)
- Problem: is B crashed, or is the network just slow?
- Should A wait for B before deciding?
  - if yes: maybe B is crashed, so it'll wait forever!
  - if no: maybe B is just slow, and will decide something else

A bit more formal

- Consider executions of a distributed system: the sequence in which the network delivers messages to their recipients

- **Bivalent state**: a state where the network could affect which value the processes choose

FLP proof sketch

- All fault-tolerant algorithms have bivalent starting conditions
- For any bivalent state, there's some sequence of message deliveries that leads to another bivalent state
    - Intuition: suppose there's some message *m* that causes the system to go from bivalent to 0-valent. What if we delay it?
    - Tricky part: in fact, we could delay it until delivering *m* keeps the system bivalent
    - Can repeat indefinitely, causing algorithm to take forever

So what?

- We still need consensus algorithms!
- But they must somehow avoid the FLP limitation
    - always safe but don't always terminate
    - randomized; terminates w/ high probability
    - bound on message delivery time
    - assume loosely synchronized clocks
    - …
- Next week: Paxos
  not guaranteed to terminate in all cases

Why stick to an asynchronous model?

- In practice, we *could* come up with a decent bound on network latency & use this as a timeout
- But it would be have to be pretty high
- Resulting algorithm would have that timeout hardcoded
- Asynchronous algorithms are self-tuning