

RAID Storage and Network File Systems

George Porter

CSE 124

February 24, 2015

* Thanks to Dave Patterson and Hong Jiang

Andrew File System (AFS)

- Originally research project at CMU
 - Then, defunct commercial product from Transarc (IBM)
Public versions now available
 - Goal: single namespace for global files
- Example of stateful approach
 - Keeps track of clients accessing files
Uses callback-based invalidation
 - Eliminates need for timestamp checking, increase scalability
 - Complicates design

AFS Model

- Files cached on client disk through whole file transfer
 - Idea: migrate files to client machines where used
- Server provides *callback* promise
 - On file retrieval, server provides a time period for lease
 - Client can only access file while promise valid
- On writes, new file version flushed to server on close
 - Server *invalidates* cache of all clients that have callbacks
 - Last writer wins
- Fault-tolerance
 - Client failure: discard callbacks (soft state)
 - Server failure: server *must* remember all callbacks

AFS Semantics

- Read sharing
 - On file open, get current file version
 - On file close after update, create new version, invalidate appropriate client caches
- Write sharing
 - Last writer wins
 - Undefined semantics
 - OK for UNIX?
- Performance
 - Callbacks much more efficient than timestamp checking
 - AFS scales to thousands of hosts

Coda

■ Offshoot of AFS

- Observation: AFS does the work of migrating popular/necessary files to your machine on access
- Nice model for mobile clients who are often disconnected

Use file cache to make *disconnection* transparent

At home, on the road, away from network connection

■ Coda supplements AFS file cache with user preferences

- E.g., always keep this file in the cache
- Supplement with system learning user behavior

■ Consistency issues?

Coda Consistency

- How to keep cached copies on disjoint hosts consistent?
 - In mobile environment, “simultaneous” writes can be separated by hours/days/weeks
- Callbacks cannot work since no network connection is available
- Coda approach (in order):
 - Assume that write sharing the rare case
 - Attempt automatic patch
 - Fallback to manual (user) intervention
- How to efficiently reintegrate changes back to main vol.

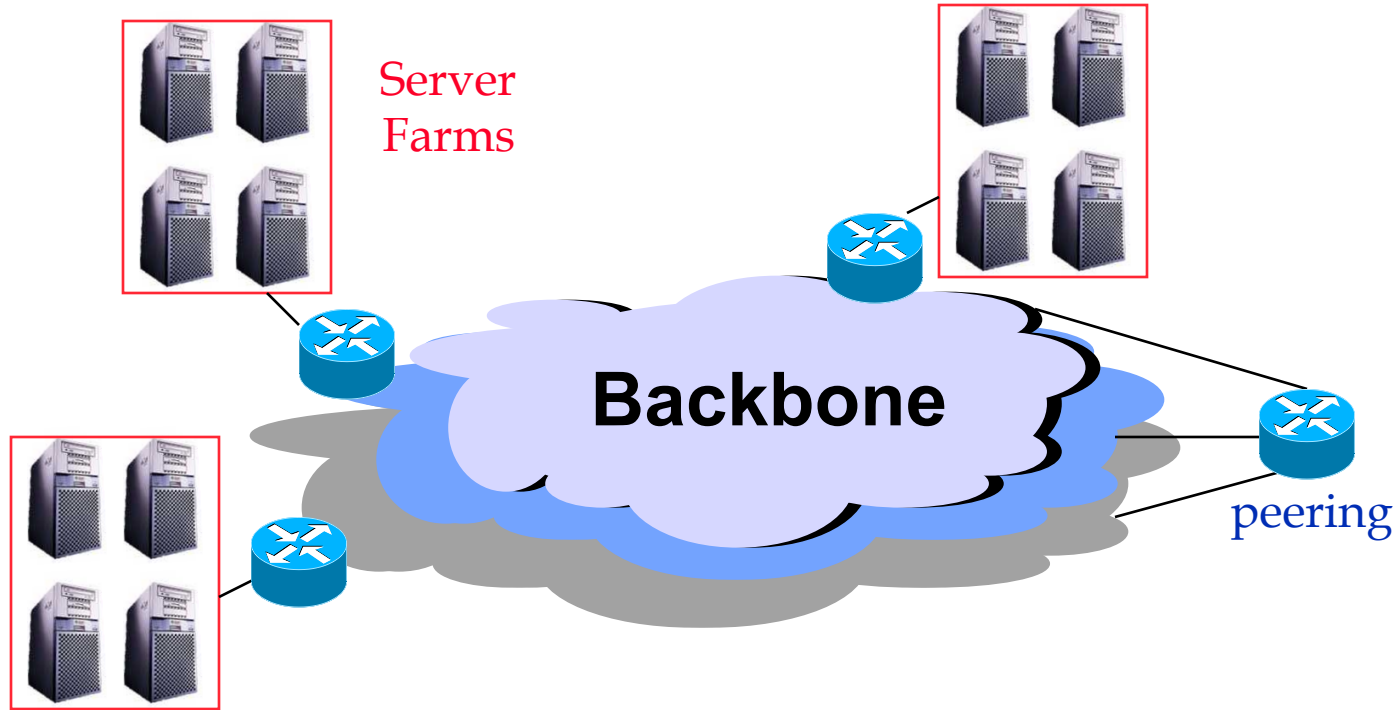
Replication, fault tolerance, and load balancing

George Porter

CSE 124

Feb 26, 2015

High Performance and Availability Through Replication?



- Improve probability that nearby replica can handle request
- Increase system complexity

The Need for Replication

- Certain mission critical Internet services must provide 100% availability and predictable (high) performance to clients located all over the world
 - With scale of the Internet, high probability that some replica/ some network link unavailable at *all* times
- Replication is the *only* way to provide such guarantees
 - Despite any increased complexities, must investigate techniques for addressing replication challenges

Replication Goals

- Replicate network service for:
 - Better performance
 - Enhanced availability
 - Fault tolerance
- How could replication lower performance, availability, and fault tolerance?

Replication Challenges

■ Transparency

- Mask from client the fact that there are multiple *physical* copies of a *logical* service or object
- Expanded role of naming in networks/dist systems

■ Consistency

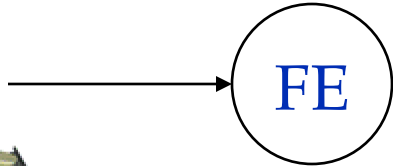
- Data updates must eventually be propagated to multiple replicas
- Guarantees about latest version of data?
- Guarantees about ordering of updates among replicas?

■ Increased complexity...

Replication Model



Client



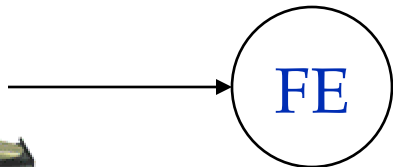
Replica



Replica

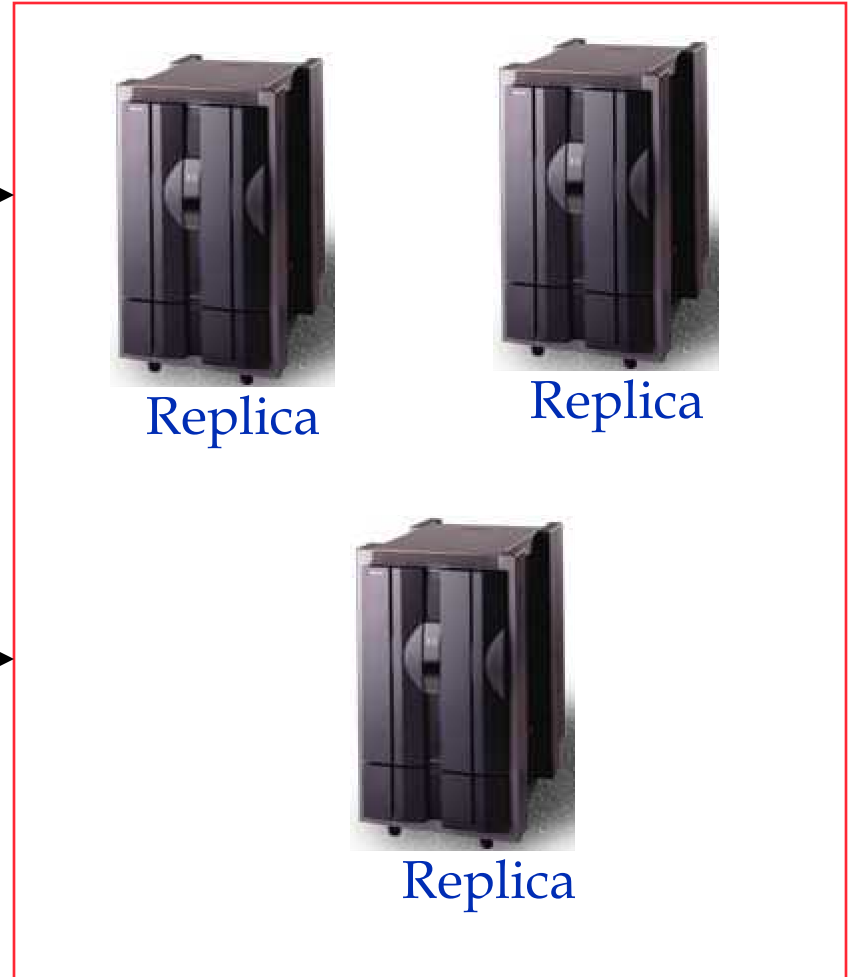


Client



Replica

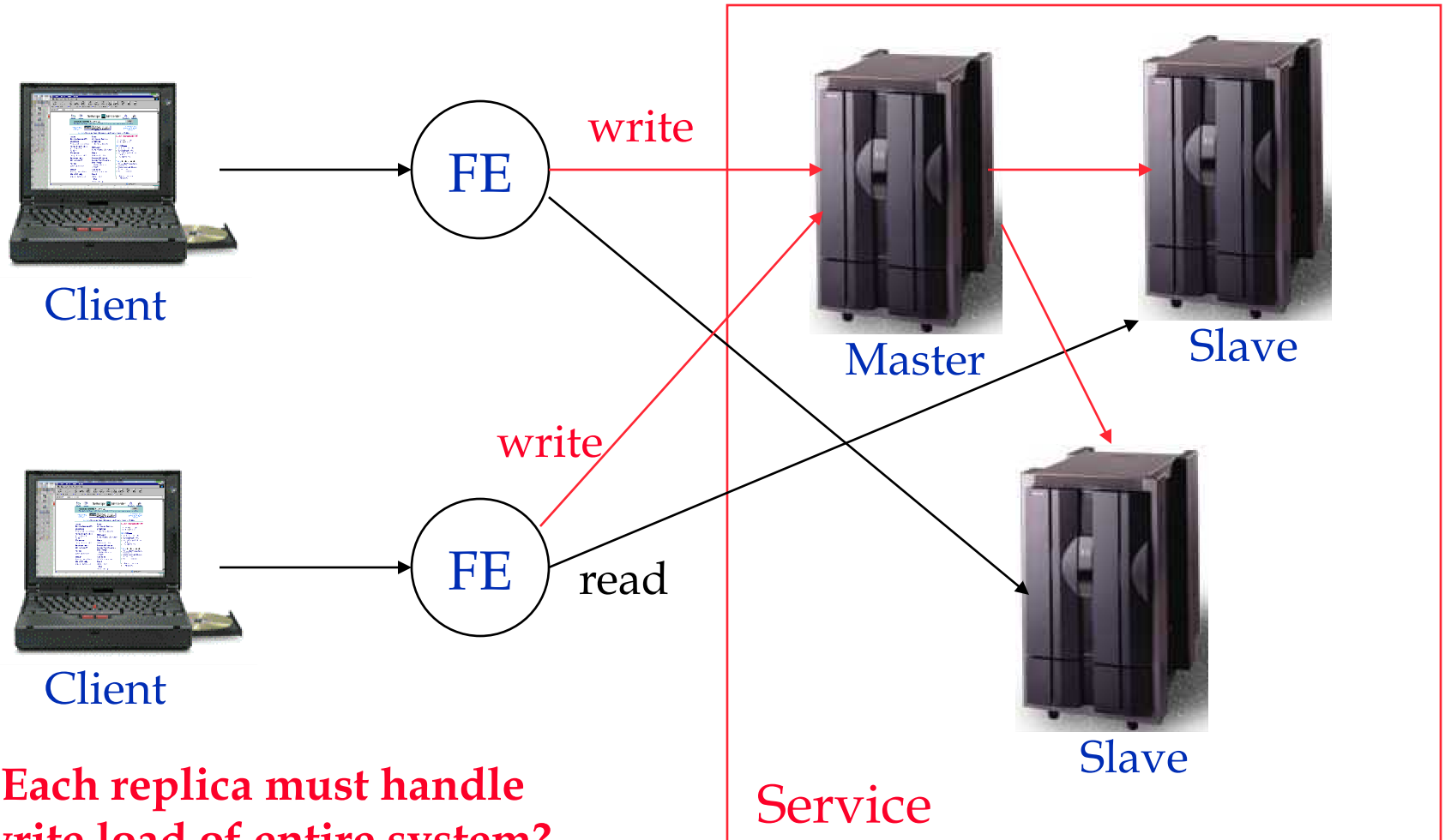
Service



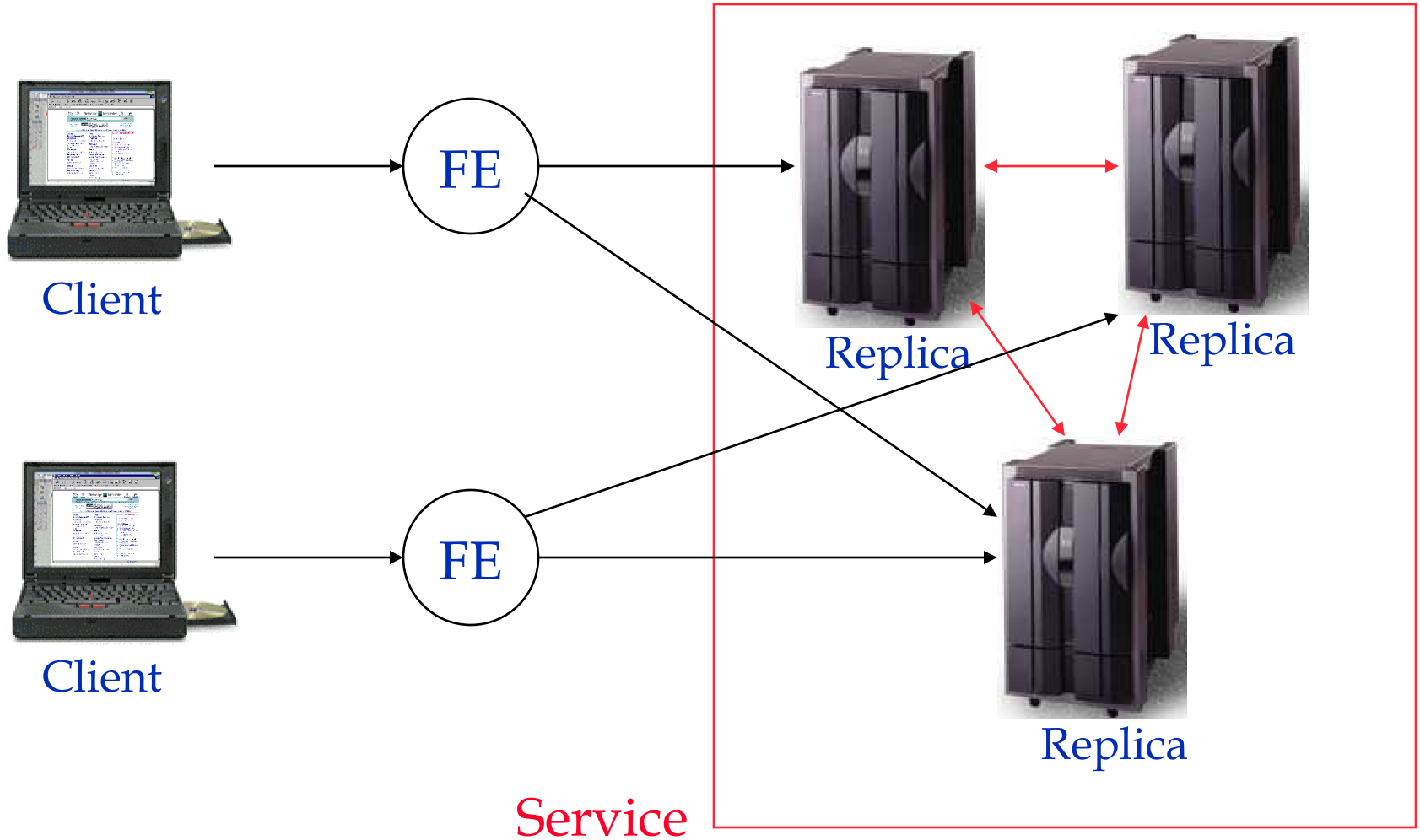
How to Handle Updates?

- Problem: all updates must be distributed to all replicas
 - Different consistency guarantees for different services
 - Synchronous vs. asynchronous update distribution
 - Read/write ratio of workload
- Primary copy
 - All updates go to a single server (master)
 - Master distributes updates to all other replicas (slaves)
- Gossip architecture
 - Updates can go to any replica
 - Each replica responsible for eventually delivering local updates to all other replicas

Primary Copy



Gossip Architecture



Update Ordering Requirements

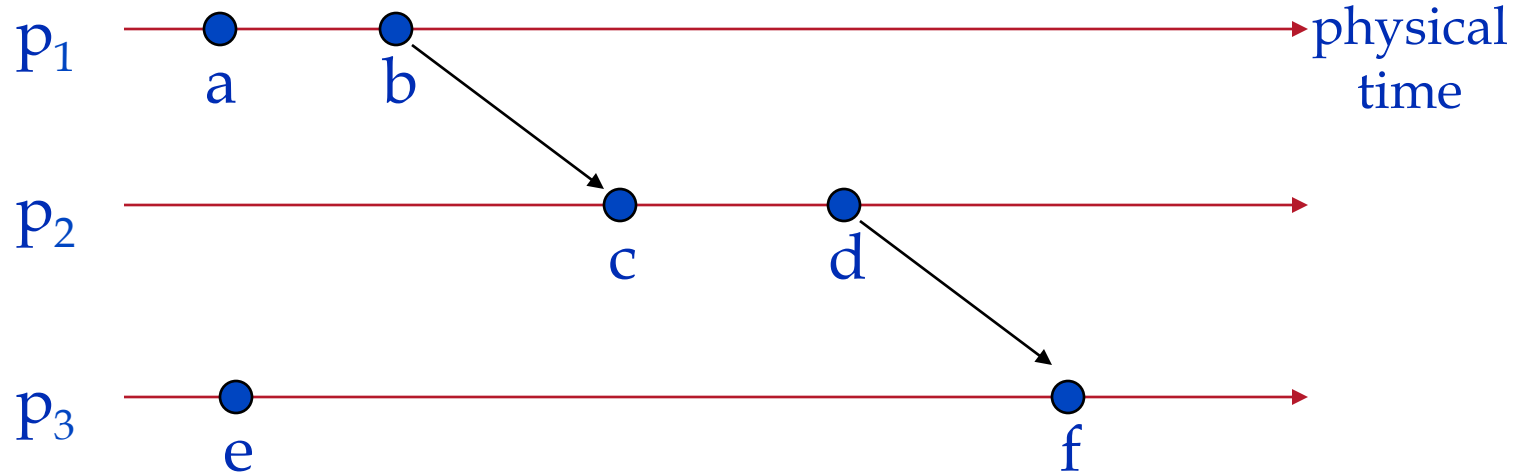
■ Total Order

- Bulletin board: all messages assigned globally unique message identifier
- For messages $r1$, $r2$, either $r1$ appears before $r2$ at all replicas or $r1$ appears after $r2$ at all replicas

■ Causal Order

- Bulletin board: message replies appear *after* original posting
- For messages $r1$, $r2$, $r1$ appears before $r2$ if $r1$ *happens before* $r2$

Happens Before



- Captures potential causal ordering (information flow)
- $a \rightarrow b$ if a takes place before b in same process
- $\text{Send}(m) \rightarrow \text{recv}(m)$
- Transitivity holds

Implementing Total Ordering

- Use *sequencer*
 - Send updates to centralized site, assign monotonically increasing identifier, distribute to all replicas
 - Single point of failure, contention
- Distributed total ordering
 - Front end sends update to all replicas
 - Each replica proposes unique id
 - Front end picks highest value
 - Transmits final value back to replicas
 - 3 messages/replica overhead

Network Partitions

- Some failure (either network or host) keeps replicas from communicating with one another
- How to proceed with read/write transactions in case where not all replicas can be contacted?

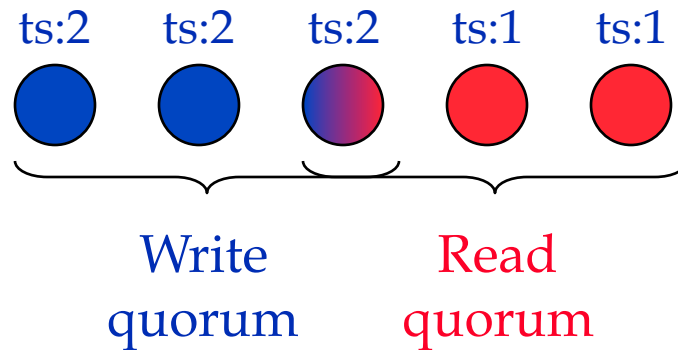
Network Partitions

- Some failure (either network or host) keeps replicas from communicating with one another
- How to proceed with read/write transactions in case where not all replicas can be contacted?
 - Optimistic versus pessimistic techniques
- Optimistic: proceed as normal, resolving conflicts later
 - Coda
- Pessimistic: ensure that replicated database can eventually be restored to consistent state without user intervention
 - Assumes that partition will eventually be repaired
 - What if the network is never fully connected?

Quorum (Voting) Consensus Techniques

- Allow updates to be completed even when only a subset of replicas are available
 - Or have the latest version of a data item
 - May need to bring some set of replicas up to date to proceed
- One technique is the weighted voting scheme
 - Assign a number of votes to each replica
 - Determine a write quorum W and a read quorum R
 $W > \text{half the total votes}$, $R + W > \text{total number of votes}$
 - Ensures that there is some overlap between read and write quorums
Read operation guaranteed access to one site with latest version

Quorum Example



- 5 replicas, read quorum: 3, write quorum: 3
 - $R + W > \#$ votes ensures overlap between any read/write quorum
 - $W > \frac{1}{2}$ votes ensures overlap between two write quorums

Quorum Consensus

- Write operations can be propagated in background to replicas not in quorum
 - Assumes eventual repair of any network partition
- Operations are slowed by the necessity of first gathering a quorum
 - Though previously, all writes had to go to all replicas

With quorum system, must only contact subset of replicas

Byzantine Failures

- Normally consider failures to be *fail-stop*
 - Meaning that a process/machine goes down without doing anything incorrect
 - In real world, corrupted memory means that failures are not always fail stop
- Consider worst-case scenario where machines/processes fail and become malicious
 - Can send arbitrary messages
 - Can lie about their identity

Consensus and Byzantine Failure

■ Byzantine Generals problem:

- N Generals encamped outside of enemy city
- Would like to come to consensus as to whether they should attack at dawn
- Use messengers to communicate

Assume messages are not lost

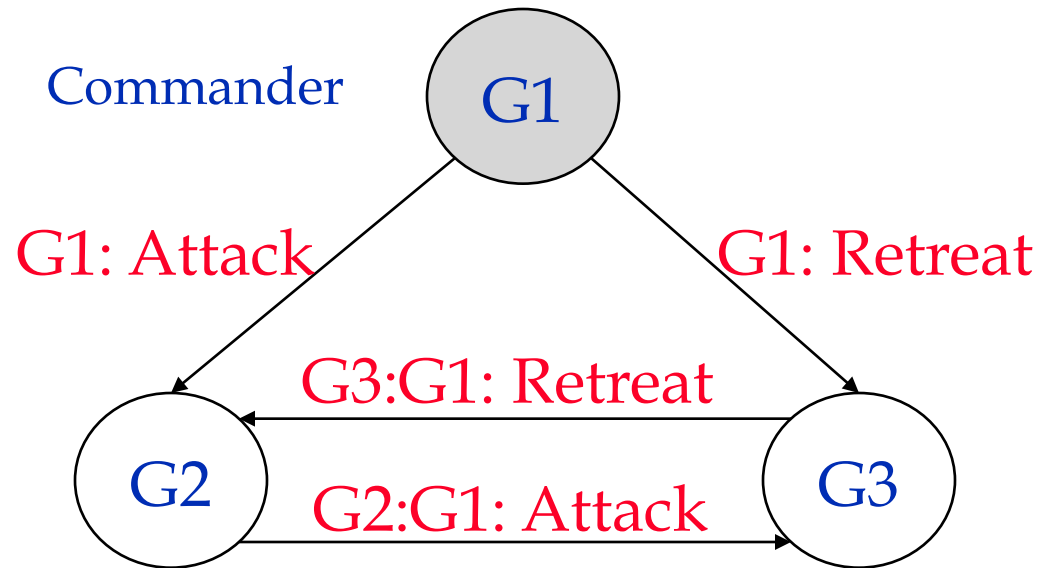
Can identify sender of each message (but no signatures)

- What happens when one or more of the generals fails (becomes malicious)?

Byzantine Consensus Goals

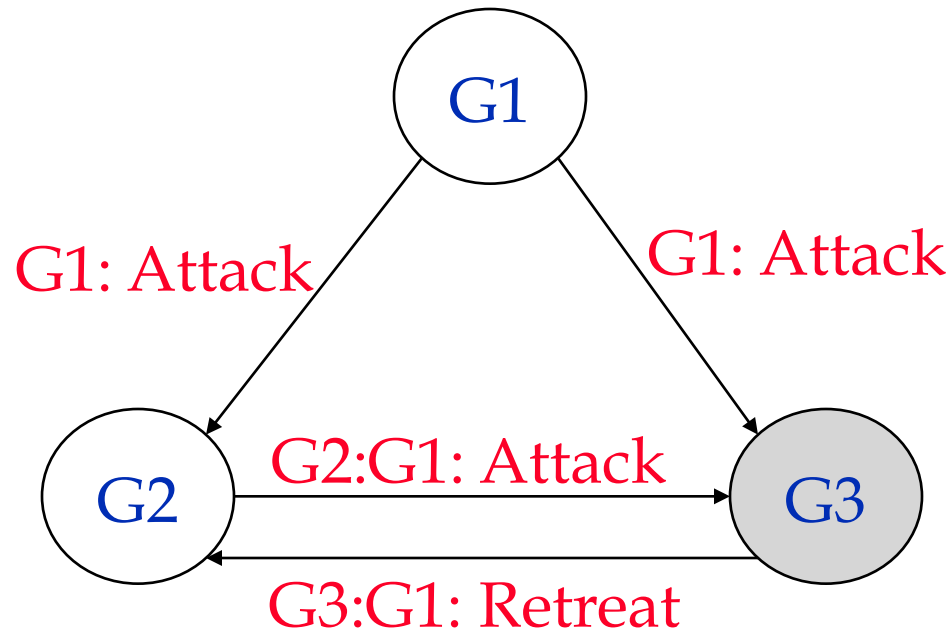
- Termination
 - Eventually, each correct process sets its decision variable
- Agreement
 - The decision value of all correct processes is the same
- Integrity
 - If the commander is correct, then all correct processes decide on the value the commander proposed

Byzantine Consensus



Neither G2 nor G3 can determine that G1 is bad;
something is wrong but attack or retreat?

Byzantine Consensus



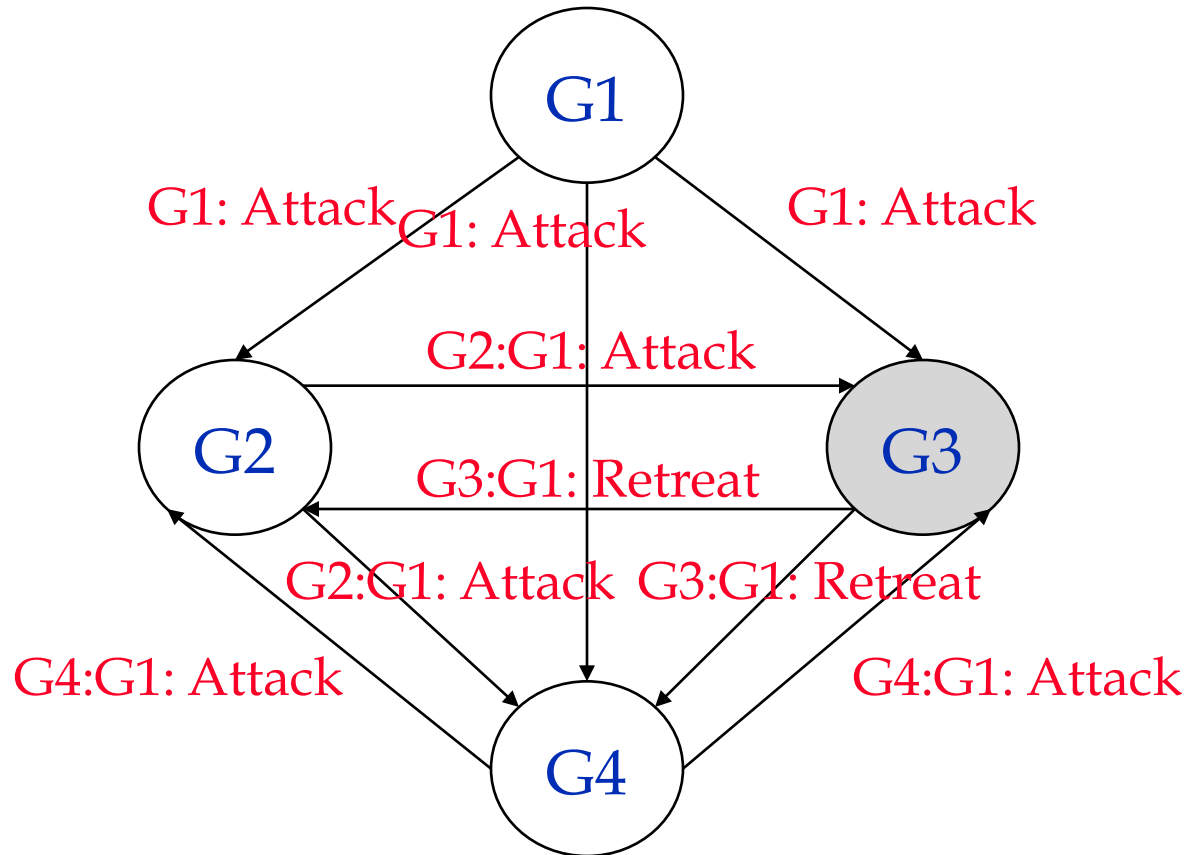
G2 realizes that someone is faulty, but who?

➔ G3 in this case

Byzantine Consensus

- In the absence of authentication:
 - $3m+1$ servers required to detect the presence of m traitors
- Consider mission critical system trying to reach consensus
 - N replicated servers performing the same calculation
 - Each server votes for the decision
 - Majority wins
 - Assume “good” (non-faulty) servers all come to the same conclusion

Byzantine Generals



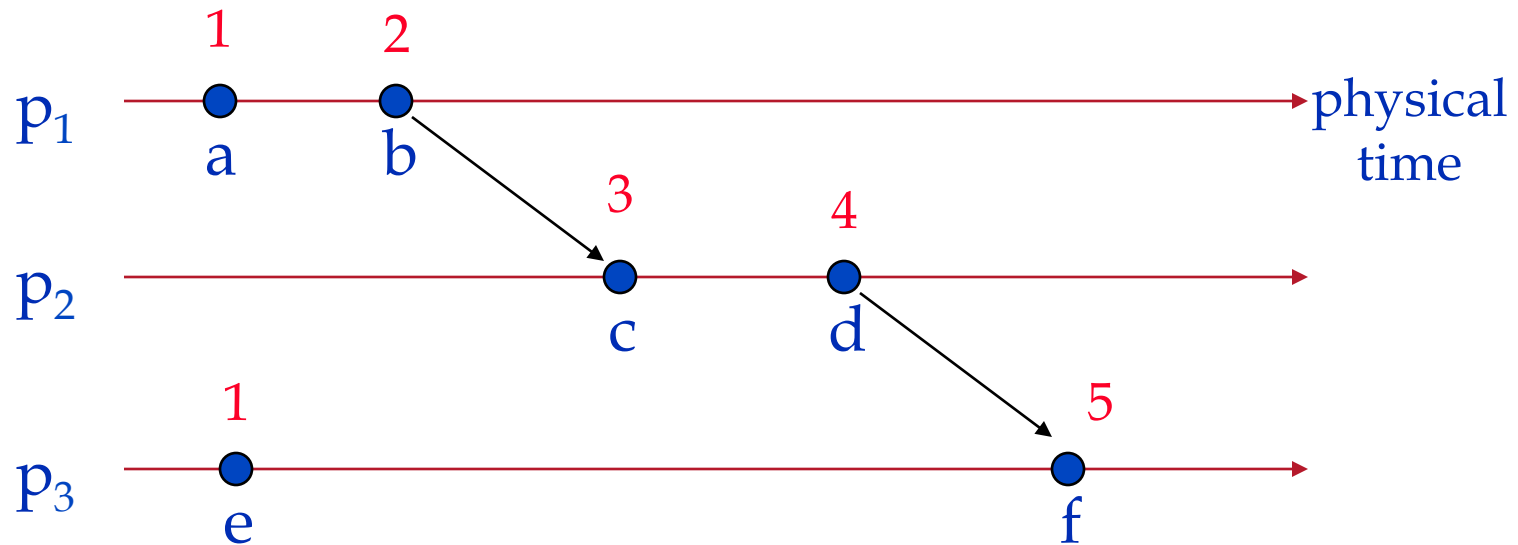
- 4 processes,
1 faulty

Eventual Consistency through Anti-Entropy

Problem Scenario

- Replicated Service
 - Perhaps across the Internet, perhaps in mobile computing environment
- Clients can apply updates to any replica (read any replica)
- Replicas may never be fully connected
 - How to achieve eventual consistency?

Logical Clocks



- Assign a monotonically increasing logical “timestamp” to all updates
- On message exchange, set local logical clock to be “one more” than remote logical clock value

Consistency through Pair-wise Message Exchange

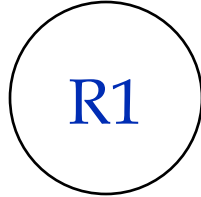
- Each replica maintains *logical time vector* of length n
 - Where n is the number of replicas
 - Entry $ltv_j[i]$ is the last known logical time for server i known by server j
- Periodically, a pair of replicas come into contact with one another → exchange messages
 - Analogous to gossip exchange
 - Logical time vector states which updates the replicas are missing relative to one another

Exchange necessary updates

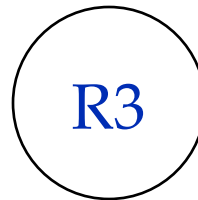
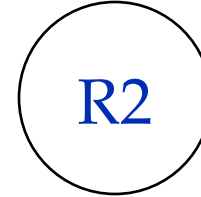
Update logical time vectors

Anti-Entropy: Simple Example

6	0	0
---	---	---



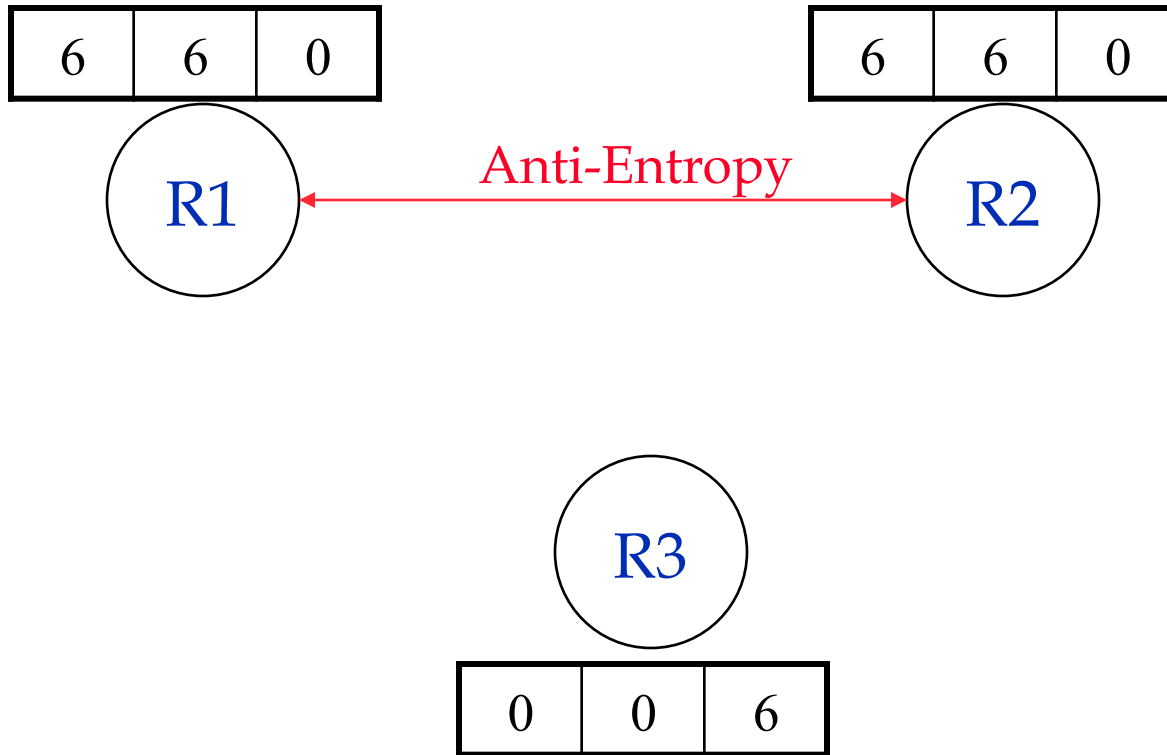
0	6	0
---	---	---



0	0	6
---	---	---

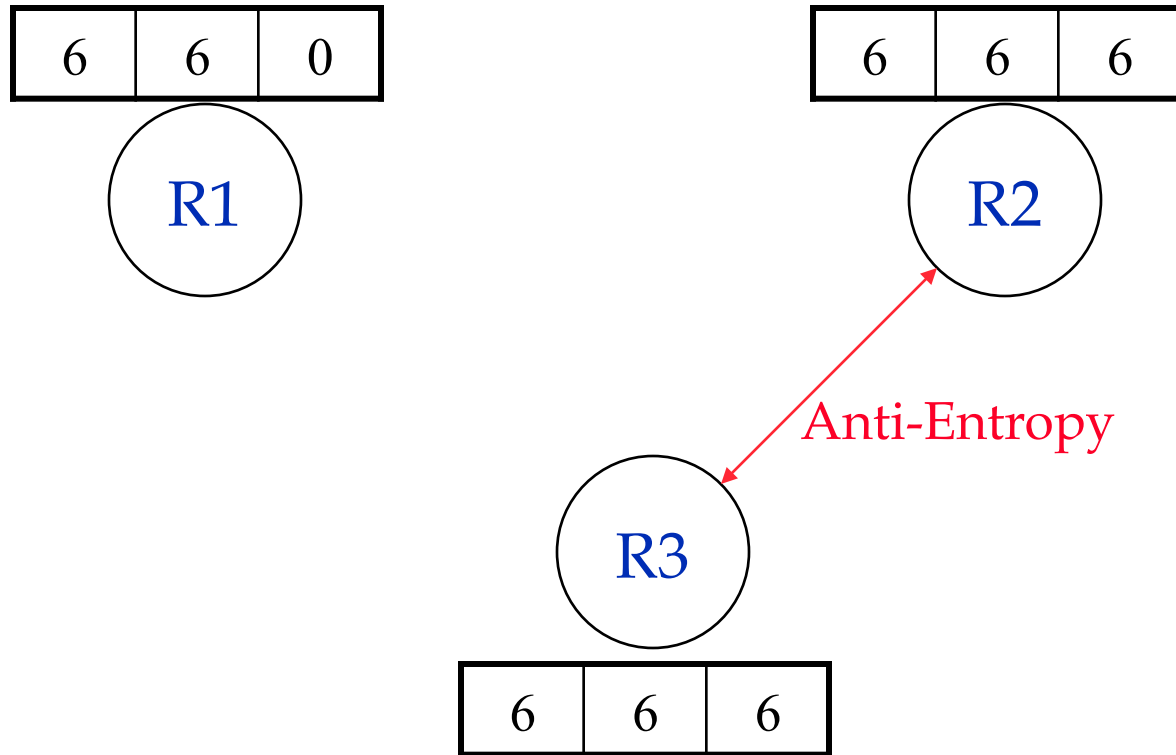
- Each of R1, R2, and R3 have seen 5 different updates (for total of 15 independent updates)

Anti-Entropy: Simple Example



- R1 and R2 perform anti-entropy
 - R1, R2 have 10 updates, R3 has 5 updates

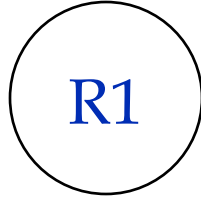
Anti-Entropy: Simple Example



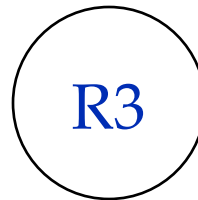
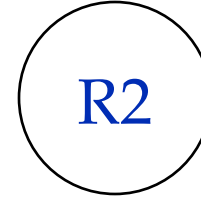
- R2 and R3 perform anti-entropy, has 15 updates
 - Its own updates at times 1-5, 10 remote updates at time 6

Anti-Entropy: Simple Example

6	6	0
---	---	---



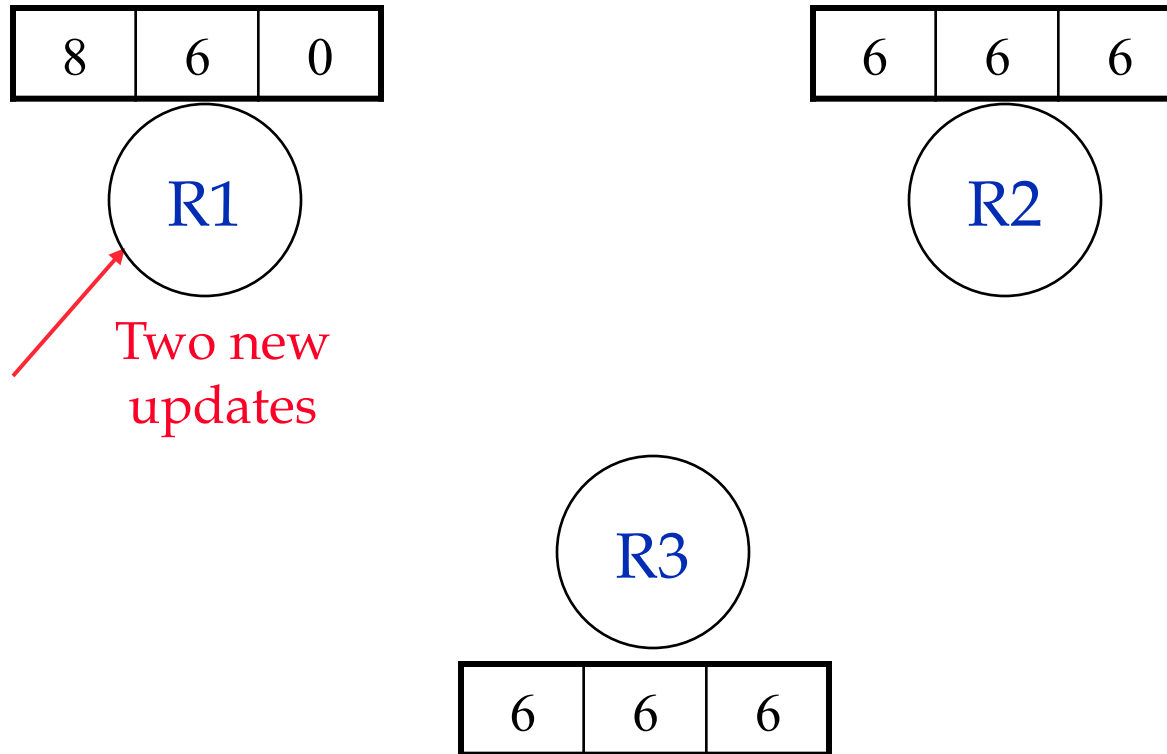
6	6	6
---	---	---



6	6	6
---	---	---

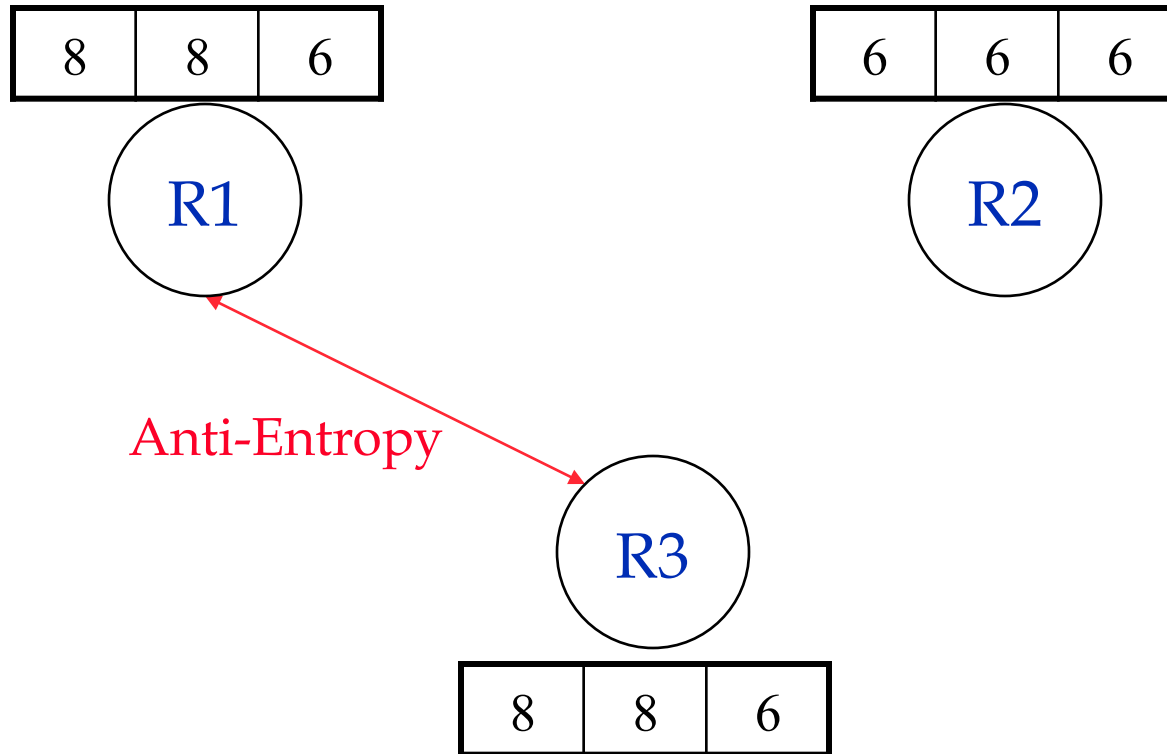
- What happens when R1 and R3 perform anti-entropy?
 - What if R1 accepts two new updates in the interim?

Anti-Entropy: Simple Example



- On Anti-Entropy between R1 and R3, R1 only propagates writes between time 6 and 8, R3 propagates all writes (after 0)

Anti-Entropy: Simple Example



- Anti-Entropy maintains causal order (happens-before) but (as described) does not impose a total order