Northeastern University - Seattle

CS6650 Building Scalable Distributed Systems

Professor Ian Gorton

Building Scalable Distributed Systems

Week 4 – Scaling the Data Layer - Fundamentals

Outline

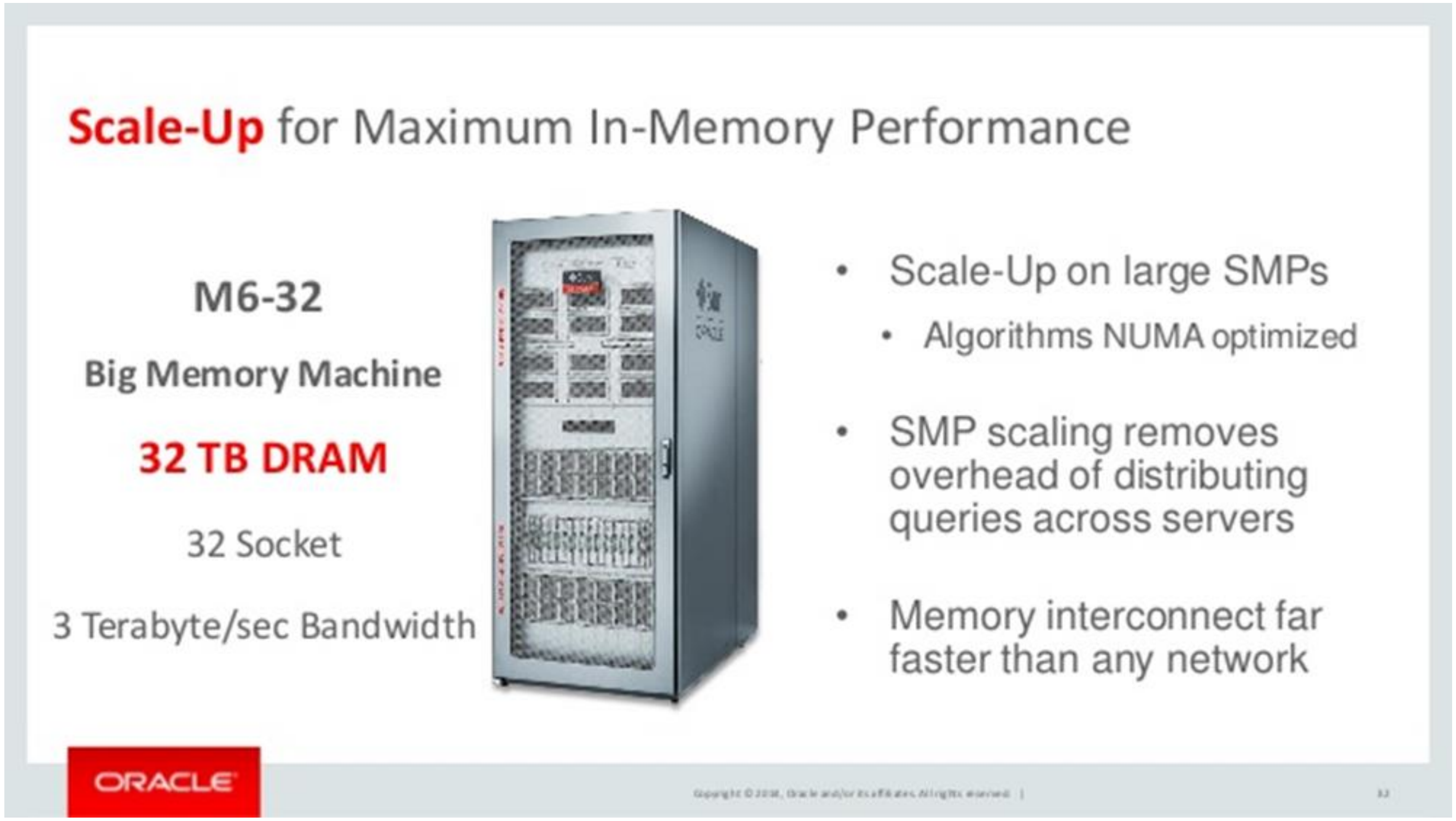
- Scaling Databases
- Partitioning and Replication
- Consistency

Scaling Databases

Relational Databases

- Established database management system technology
  - Relational model based on defined schemas and SQL query language
  - Highly optimized, stable technologies
  - Scale up easily by running database on bigger machines
    - More memory, CPUs, disks
- Many commercial/open source implementations
- De facto enterprise technologies

Scale Up – Example Hardware



Scale-Up for Maximum In-Memory Performance

M6-32

Big Memory Machine

32 TB DRAM

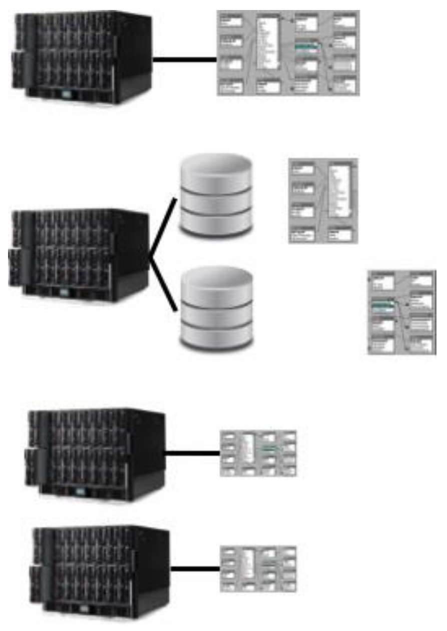
32 Socket

3 Terabyte/sec Bandwidth

- Scale-Up on large SMPs
  - Algorithms NUMA optimized
- SMP scaling removes overhead of distributing queries across servers
- Memory interconnect far faster than any network

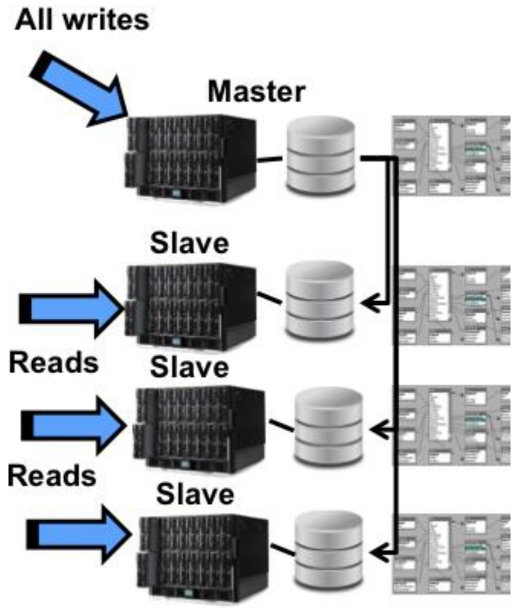
ORACLE

Scaling Relational Databases



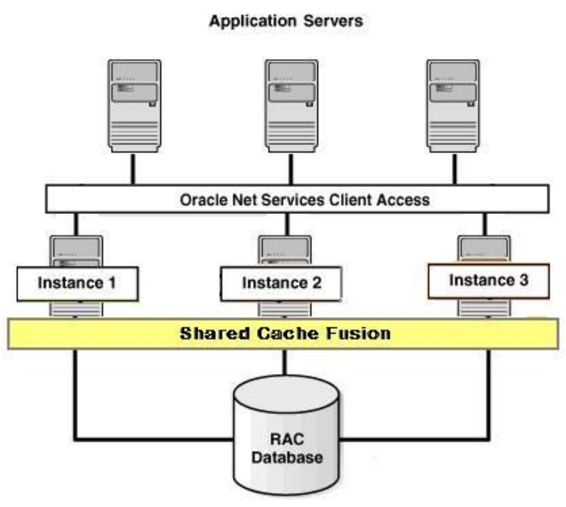
- Scale up
  - Monolithic compute resource
  - Shared disk
- Partitioned databases on disk
  - Optimizes data placement on separate disks
  - Monolithic compute resource
- Multiple database instances
  - Partition database across database engine instances
  - Functional partitioning common (e.g., customers, orders, stock)
  - More compute, more license costs

Scaling relational databases



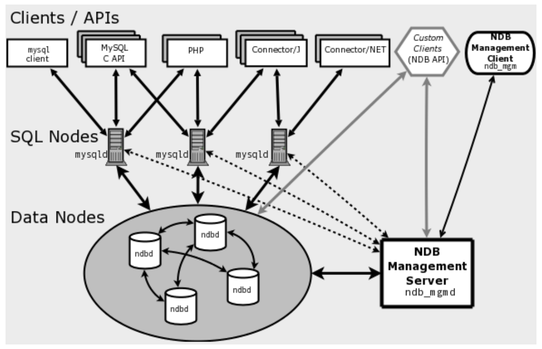
- Read/Write splitting
  - Writes to single master
  - Reads to multiple slaves
  - Reads scale
  - Writes are performance bound
  - Consistency weaker due to replication latency
- Proprietary approaches, e.g.:
  - Scale better
  - Typically require application code changes or impose SQL restrictions

Scaling Relational Databases



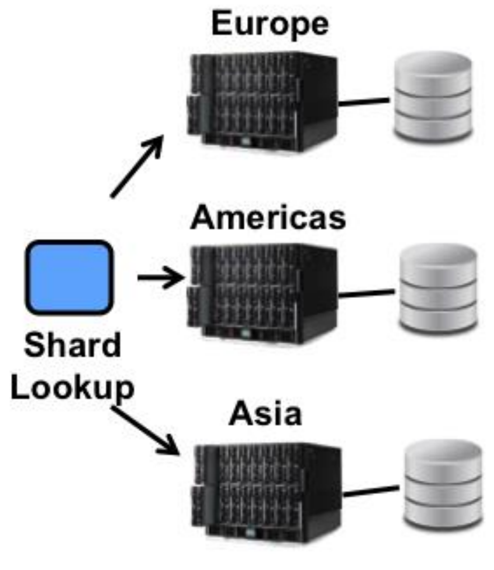
- Shared everything architecture
    - Oracle RAC
- Multiple nodes run database engine
    - Requests load balanced
- Single shared database using SAN storage
- Shared cache

Scaling Relational Databases



- MySQL cluster
- Shared nothing architecture
- distributed, multi-master with no single point of failure
- Data node manages a partition of the database
    - Partitioned by hashing on the primary key for a table

Scale out with clusters



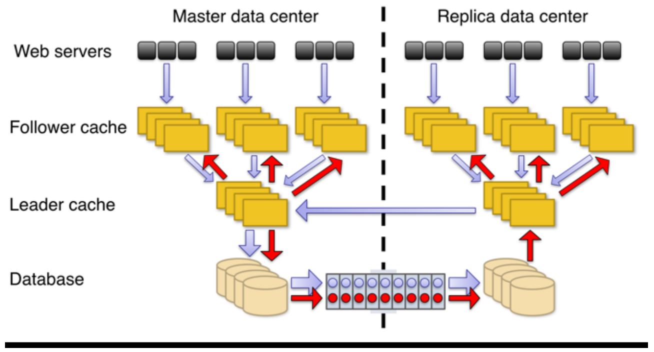
Shard - A single instance of a database, typically hosted on a commodity server, which houses a subset of the system's total dataset. Each shard within a system typically stores the same type of data, although the actual data on each shard is unique to that shard.

- Sharding
  - Horizontal data partitioning
  - Suitable for exploiting large clusters
- Many possible partitioning schemes,
  - Value based – e.g., region, customer ID
  - Hashing
- Issues:
  - Evenly distributing read load, write load, and data volume
  - Handling shard failures

Scaling Relational Databases is hard

- Relational data modeling produces general solutions that can accommodate a broad range of queries
  - Driven by structure of the data
  - Can JOIN together data from different tables at query time
- Scale up on expensive hardware is proven
- But what about scale out on commodity hardware?
  - How do we partition data horizontally?
  - How we do efficiently perform and scale distributed joins?

Facebook – Scaling MySQL

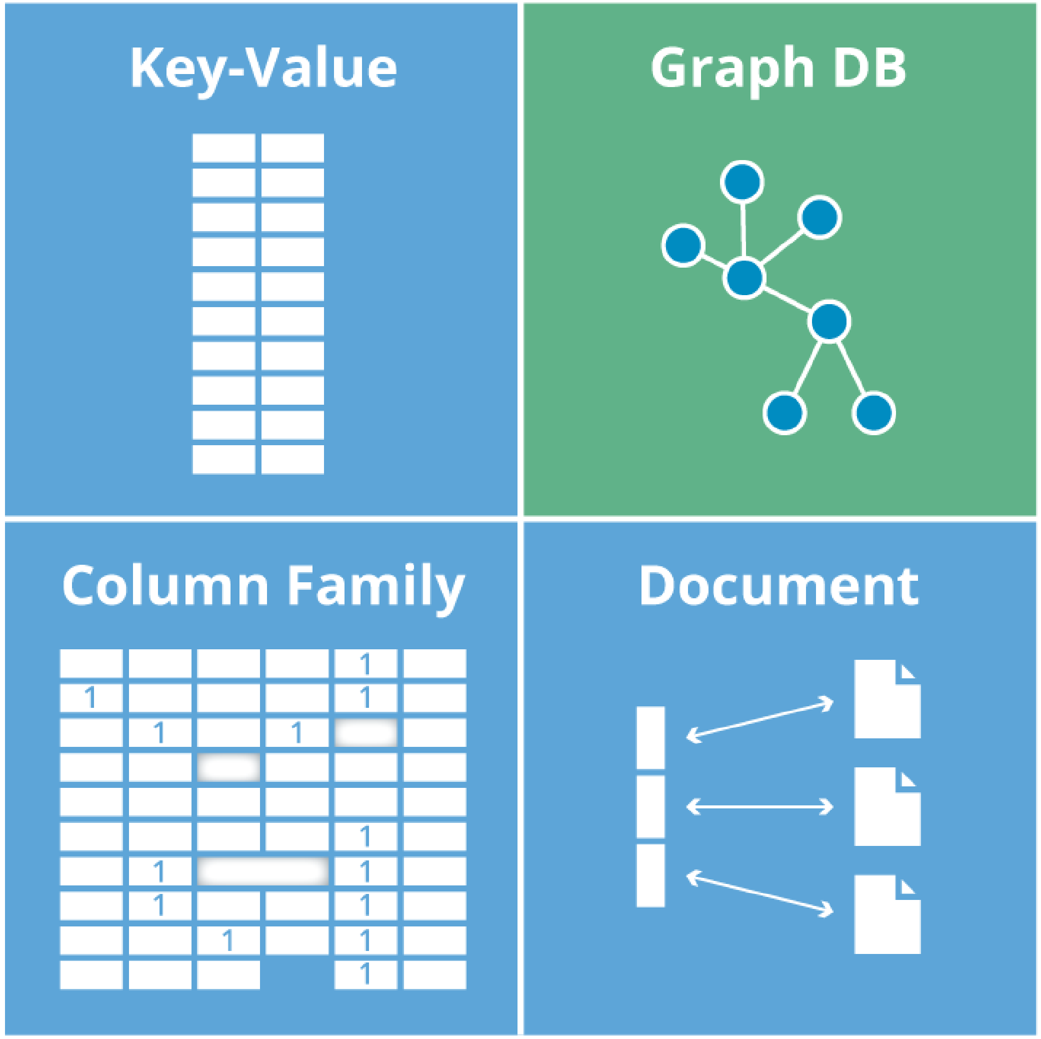


- Originally built on MySQL/memcache
- Scaling was hard
  - Extensive master-slave sharding across data centers
  - Loss of SQL language power (eg JOINs across shards)
- Built TAO on top of MySQL
  - Custom graph database layer
  - Simple API maps to fast SQL queries
  - Optimized for reads (500:1 ratio)
- Good blog post summary at https://blog.yugabyte.com/facebooks-user-db-is-it-sql-or-nosql/

Other Considerations

- RDBMS originally conceived to store ‘business data’
    - Correct and timely
    - Consistent (ACID – we’ll be back on this one!)
    - Recoverable
- As Internet drove growth for sites, new, non business critical, data types emerged
    - Log files
    - Tweets
    - Images
    - Social graphs
    - etc

New databases emerge



NoSQL – Horizontally-scalable database technology

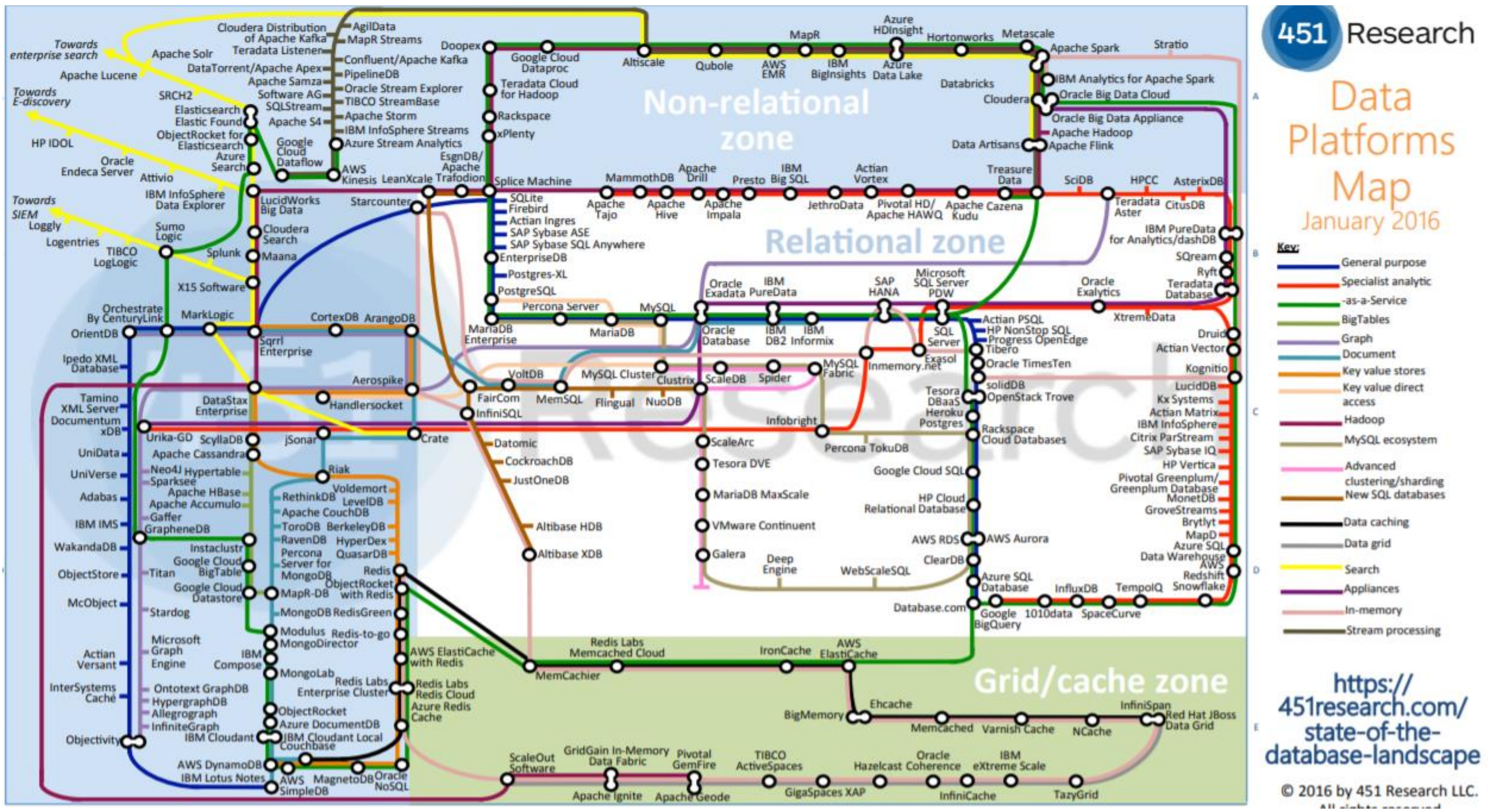
- Designed to scale horizontally and provide high performance and scalability on commodity hardware
- Large variety of:
  - Data models
  - Query languages
  - Scalability mechanisms
  - Consistency models
- NoSQL data modeling driven by the needs of the specific applications that utilize the database
  - Driven by the application use cases and data access patterns
  - Denormalized for performance

Not Only SQL

Distributed Databases

- Move to distribution of data driven by:
  - Scale – BIG data!
  - Ecomonics – many commodity low cost machines are cheap for building clusters and data centers
  - Performance – more resources can be thrown at a problem
  - Availability – Internet-scale apps needs very high availability

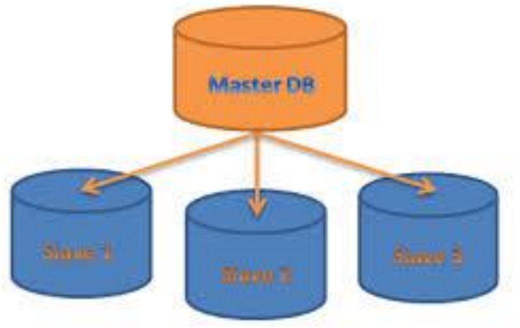
Database Landscape

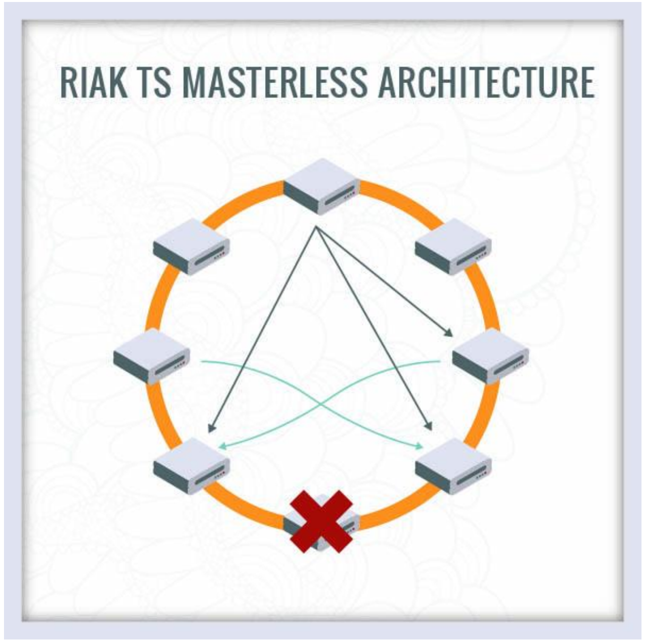


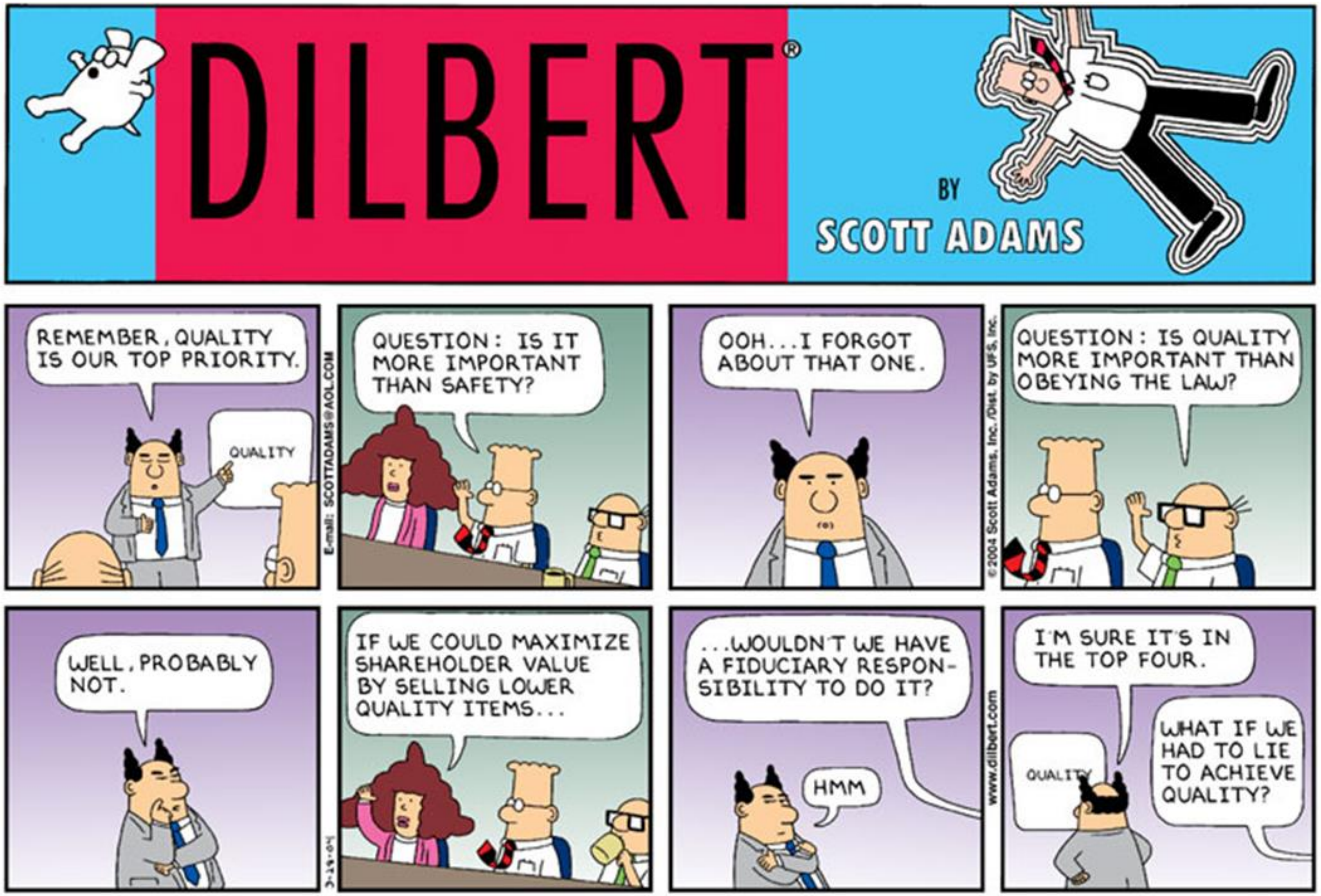
Partitioning and Replication

Two basic distributed database architectures

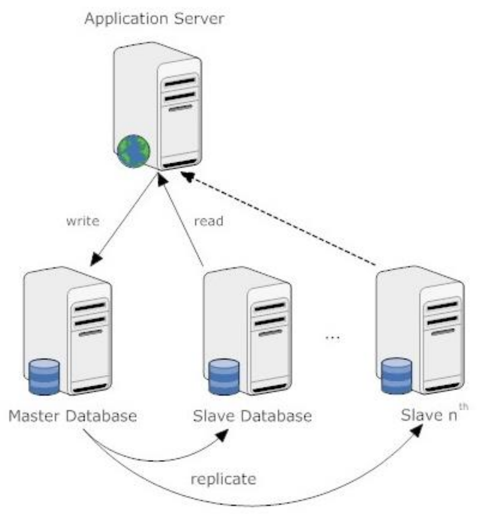
- Leader-based (Master-Slave)
- Leaderless (Peer-to-Peer)
  - Aka masterless
  - Shared nothing
- Strengths and weaknesses for each …
  - Yep. Design trade-offs ….





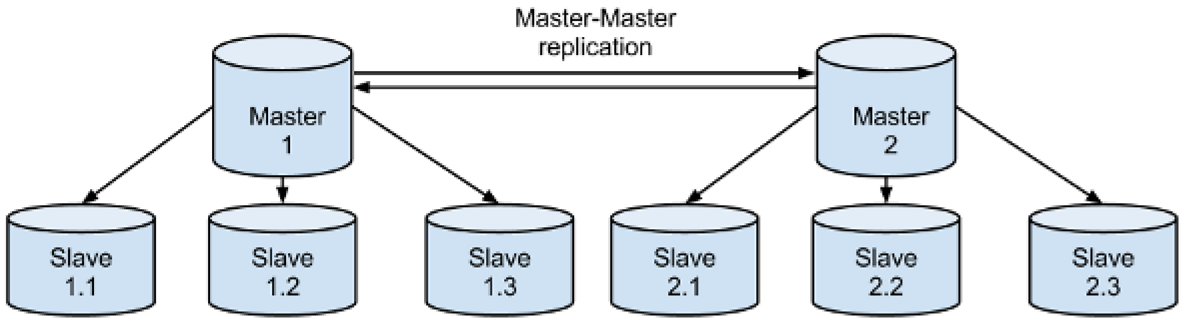

DILBERT®
BY SCOTT ADAMS
REMEMBER, QUALITY IS OUR TOP PRIORITY.
QUALITY
QUESTION: IS IT MORE IMPORTANT THAN SAFETY?
OOH... I FORGOT ABOUT THAT ONE.
QUESTION: IS QUALITY MORE IMPORTANT THAN OBEYING THE LAW?
WELL, PROBABLY NOT.
IF WE COULD MAXIMIZE SHAREHOLDER VALUE BY SELLING LOWER QUALITY ITEMS...
...WOULDN'T WE HAVE A FIDUCIARY RESPONSIBILITY TO DO IT?
HMM
I'M SURE IT'S IN THE TOP FOUR.
WHAT IF WE HAD TO LIE TO ACHIEVE QUALITY?
QUALITY
E-mail: SCOTTADAMS@AOL.COM
© 2004 Scott Adams, Inc./Dist. by UFS, Inc.
www.dilbert.com
3-28-04

Leader-based Architecture

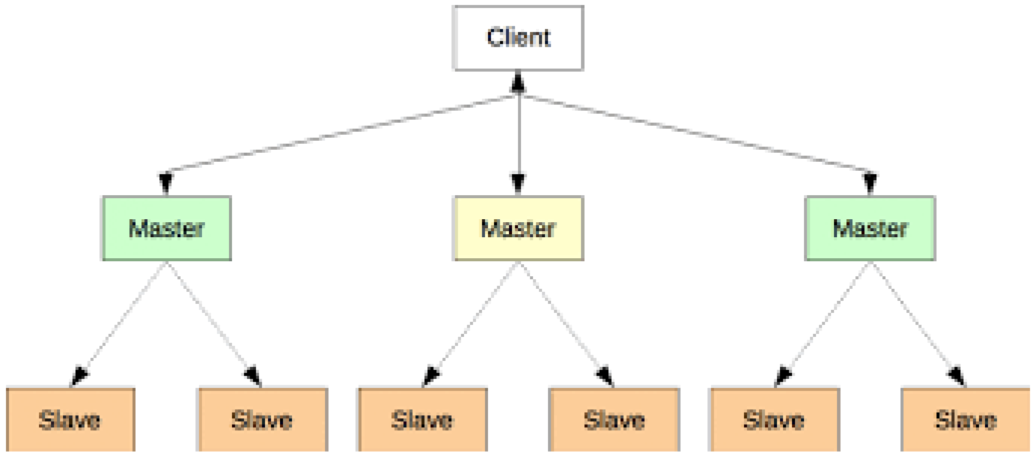


- One leader, many followers
  - Write to leader
  - Read from followers
- Strengths:
  - Single copy of data (ie the leader) is source of truth
  - Off loads reads to followers to enhance performance/scalability
- Weaknesses
  - Window of inconsistency
  - Bottleneck at leader for writes
  - Availability?

Multi-Leader-Follower Architecture



Replicated MLS



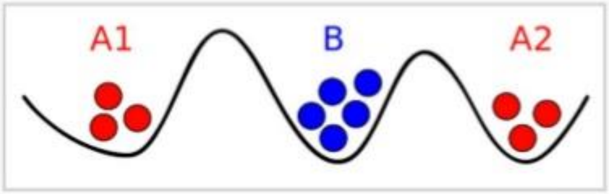
Partitioned MLS

Leader Failure Handling

- What if the leader node fails?
  - Machine crashes
  - Network fails (possibly transient)
- Sounds easy – let’s elect a new leader
- Assume 4 followers
  - Elect one as leader
  - three remains followers

Achieving Consensus in Distributed Systems

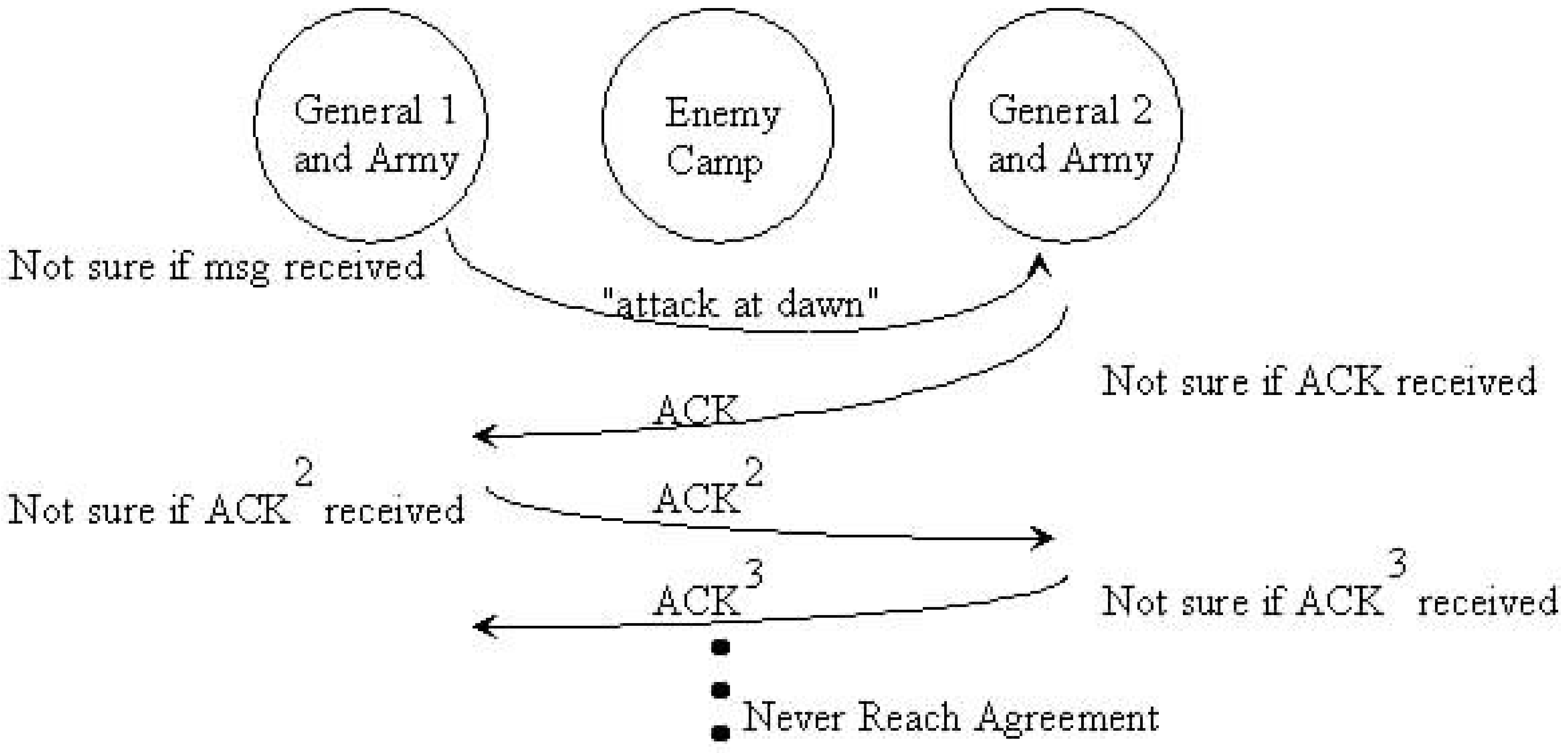
Case:Two Generals' Problem



Positions of the armies. Armies A1 and A2 need to communicate but their messengers may be captured by army B.

https://www.youtube.com/watch?v=X7jzXlt6CgE

The Two Generals Problem



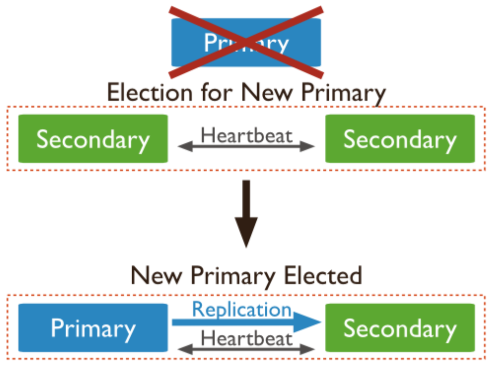
The Two Generals Problem

No 100% message delivery guarantee

It can be proven that this is impossible to solve

Ways to make the chance of failure to achieve consensus very small?

New Leader Election



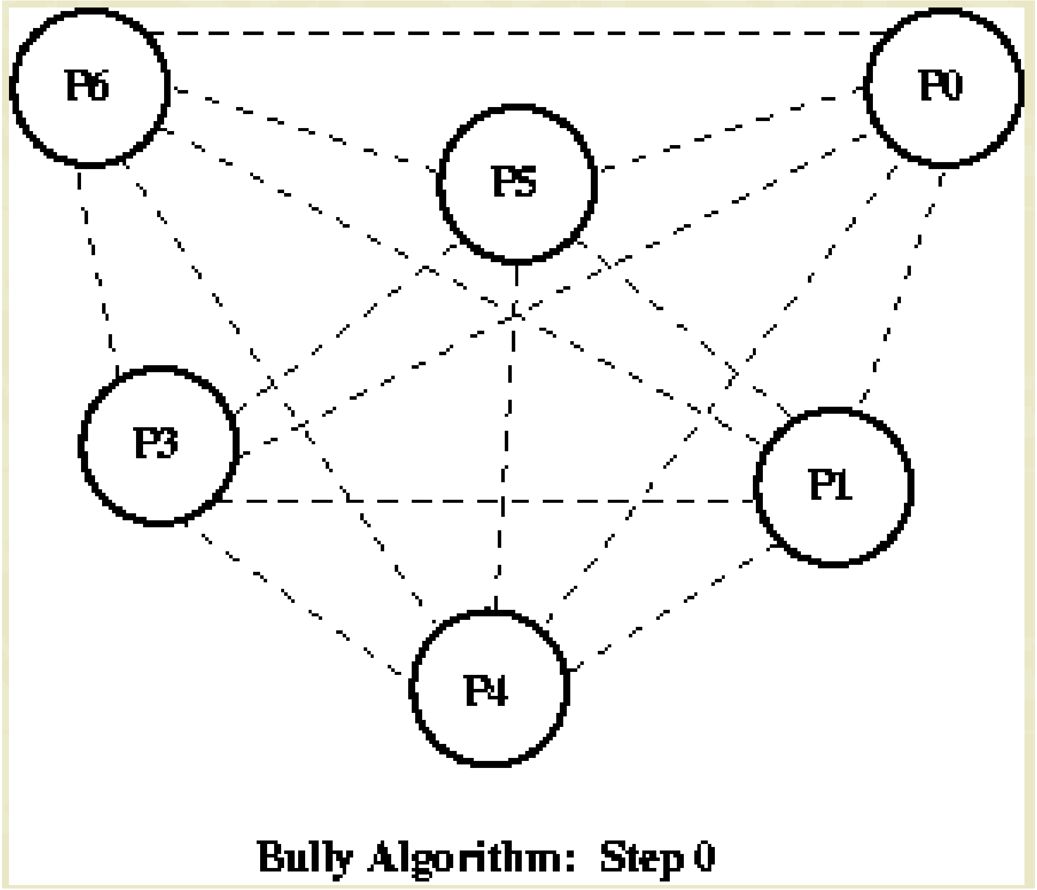
- Primary/Leader fails
  - How detected?
  - Potentially many followers (eg 7)
- Election must satisfy **safety** and **liveness** properties :
  - only one single node can enter the elected state and it will become the leader of the distributed system.
  - every node will eventually enter an elected state or a non-elected state.
- Many problems to address, eg:
  - What if followers are disconnected from each other?
    - Network partition, split brain problem
  - What if leader was simply unavailable and reconnects during the election process?

New Leader Election – The Bully Algorithm

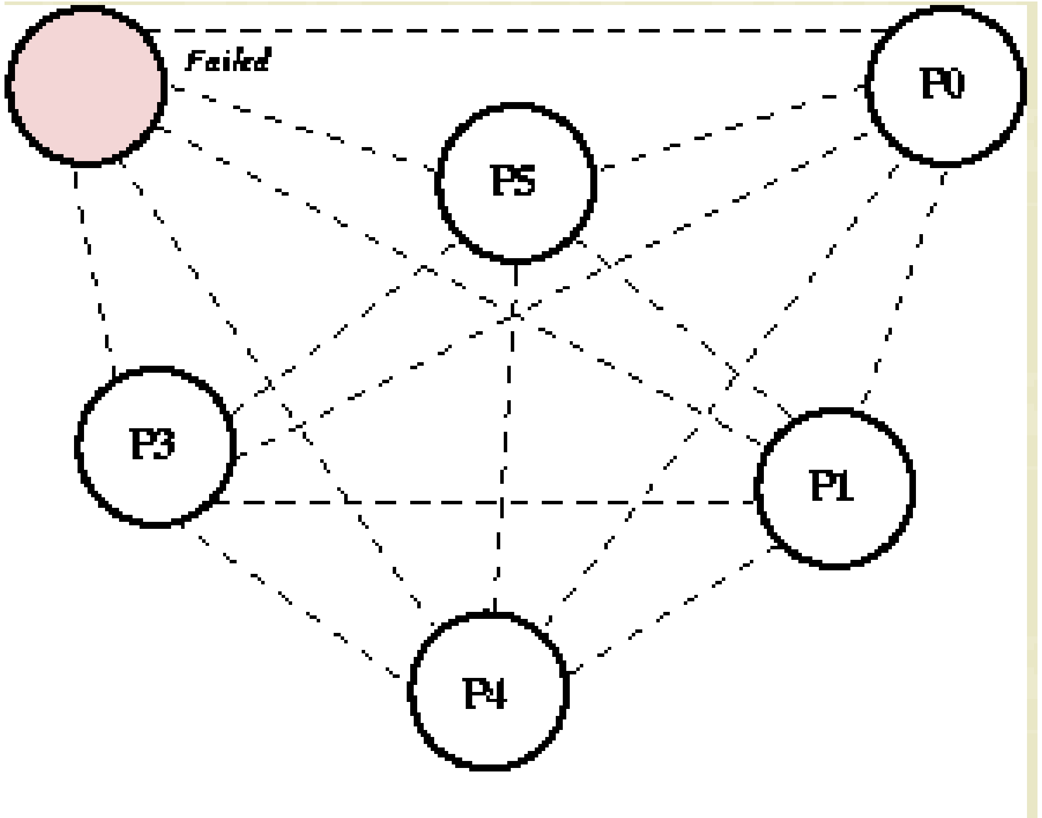
- Bully algorithm proposed by Garcia-Molina.
- basic assumptions:
  - The system use timeouts to detect process failure (coordinator)
  - each process has a unique number in the system
  - every process knows the process number of all other processes and which processes have the higher number
  - Processes do not know which processes are currently up and which processes are currently down.
  - Once an election is held a process with the highest process number is elected as a coordinator which is agreed by other processes [4].

H. Garcia-Molina, "Elections in distributed computing system," IEEE Transaction Computer, vol.C-31, pp.48- 59, Jan.1982.

Bully Algorithm

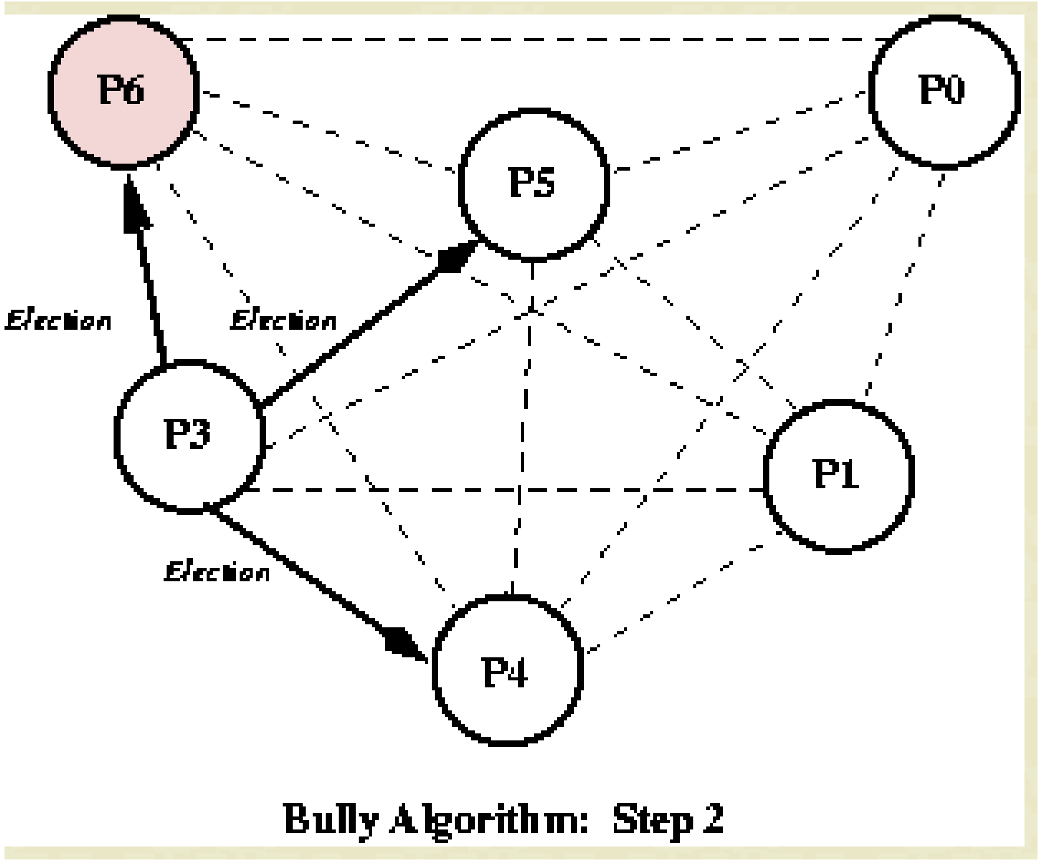


Bully Algorithm



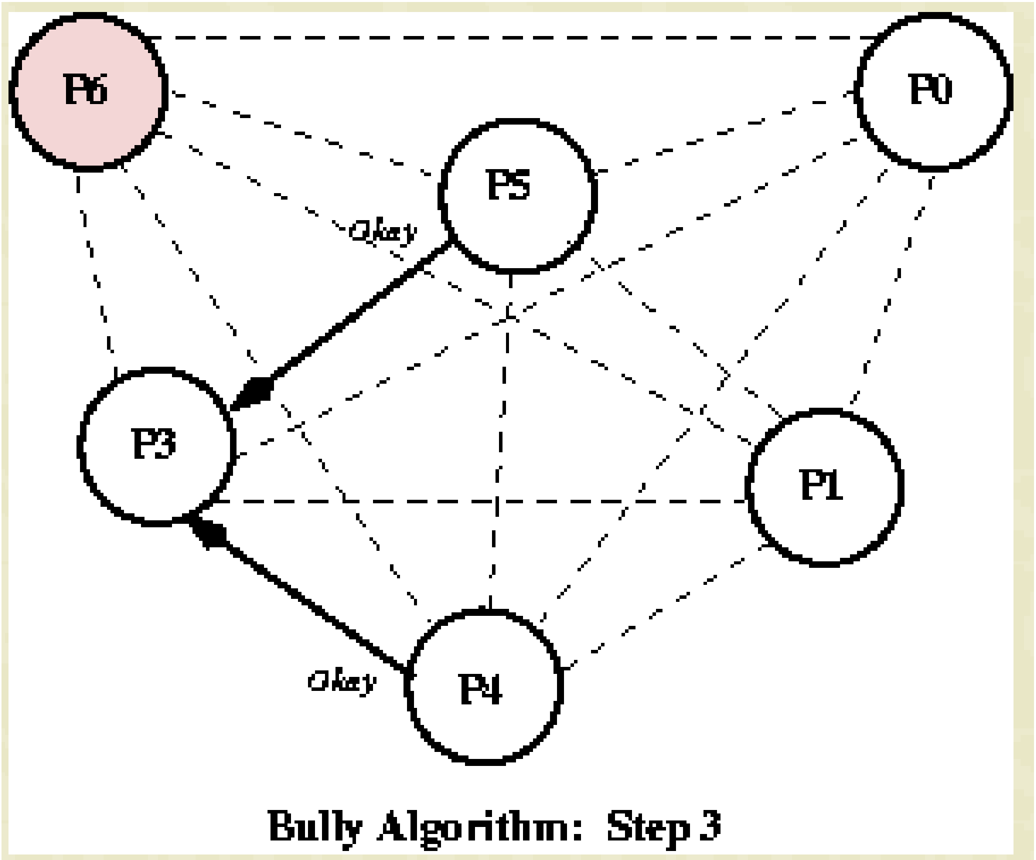
Bully Algorithm: Step 1

Bully Algorithm



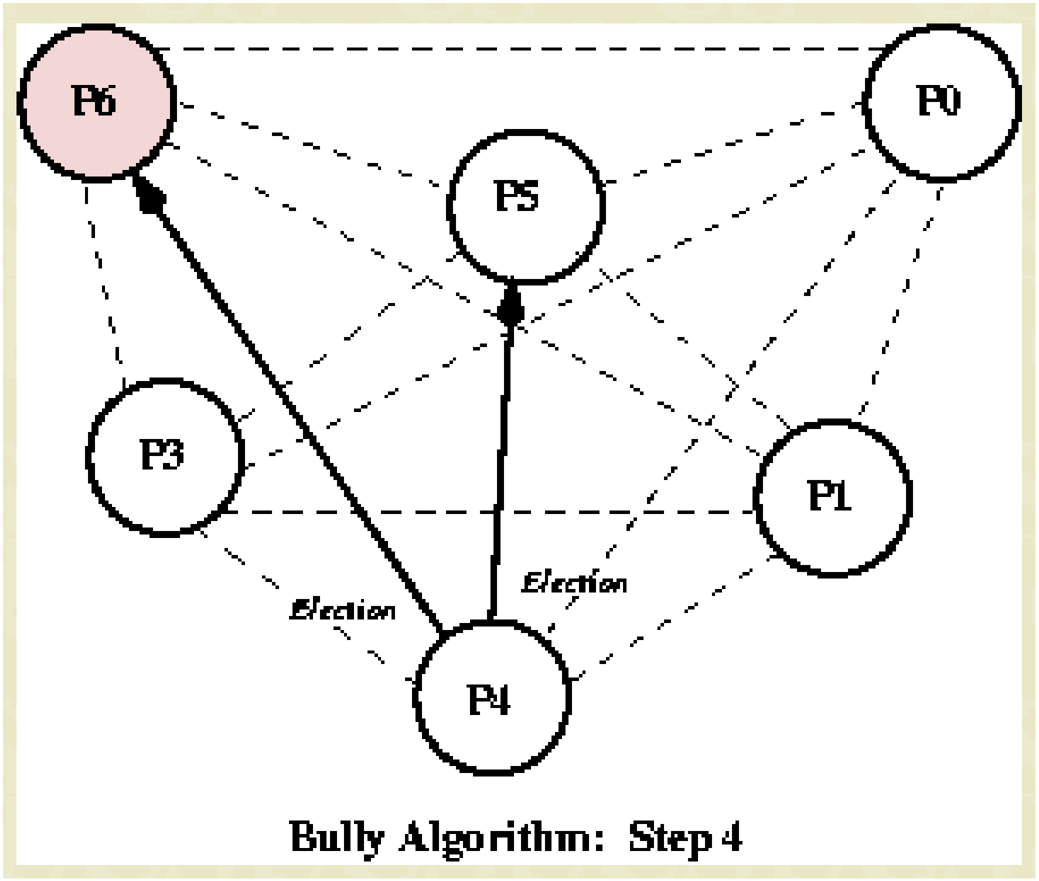
Bully Algorithm: Step 2

Bully Algorithm

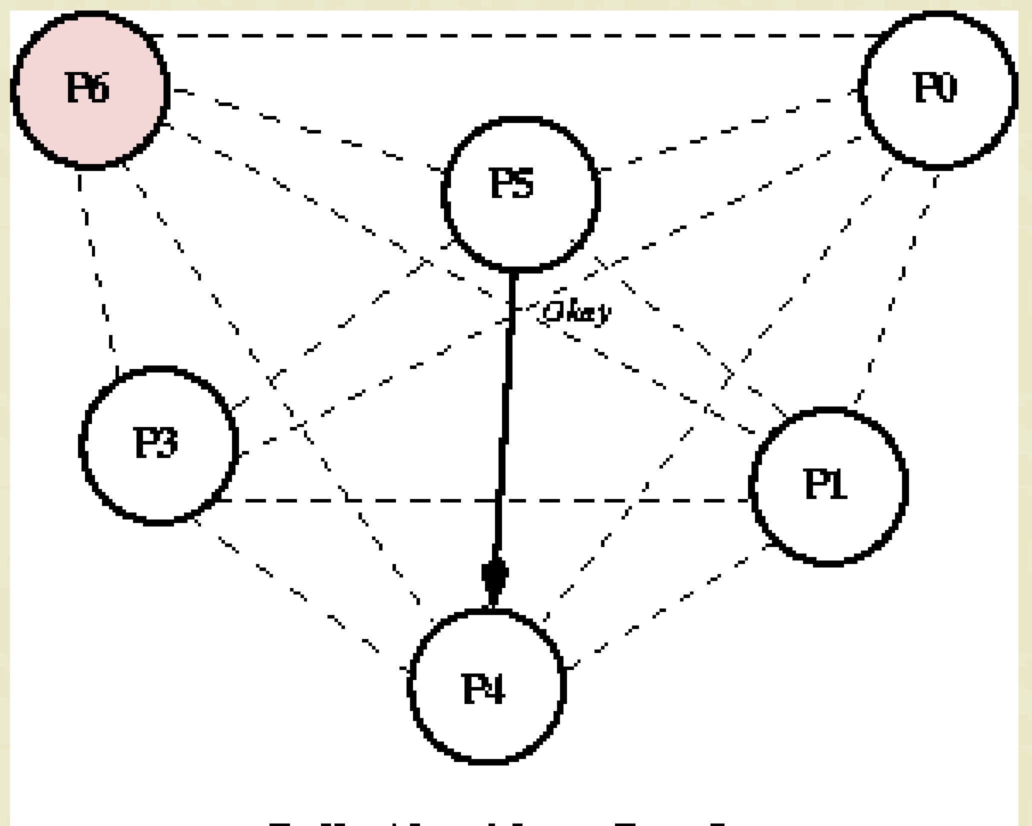


Bully Algorithm: Step 3

Bully Algorithm

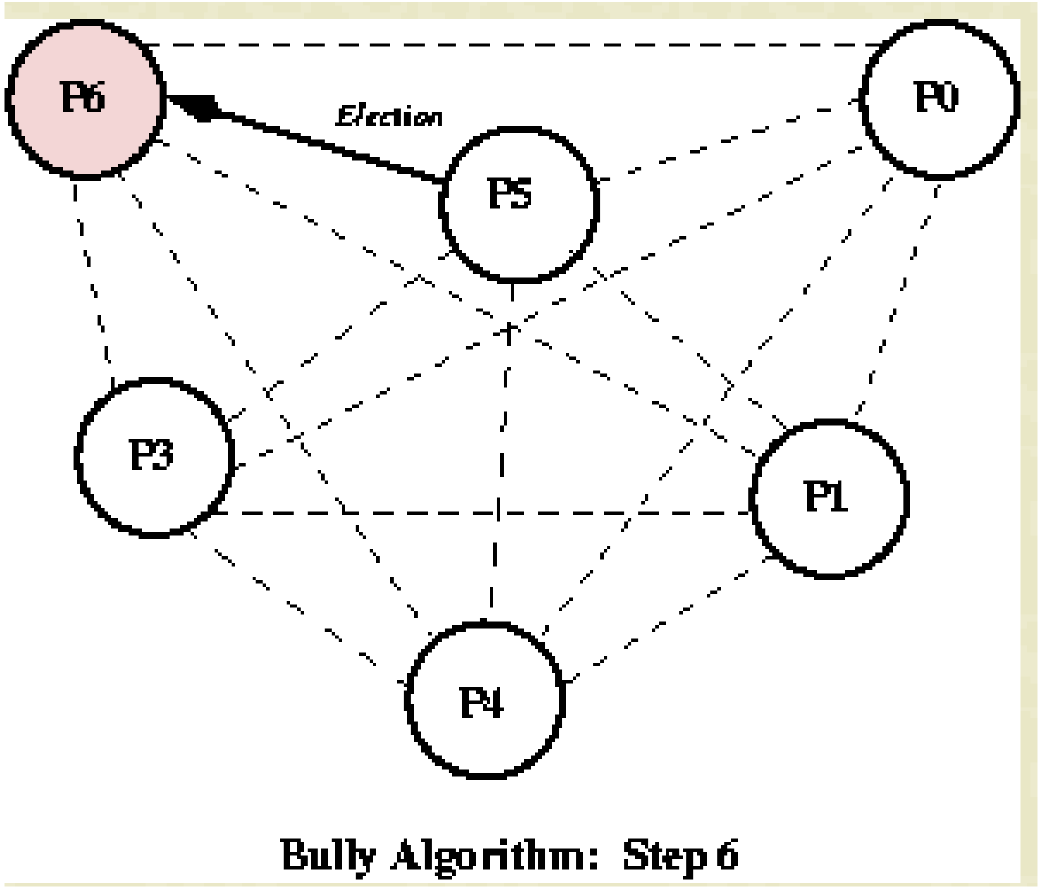


Bully Algorithm



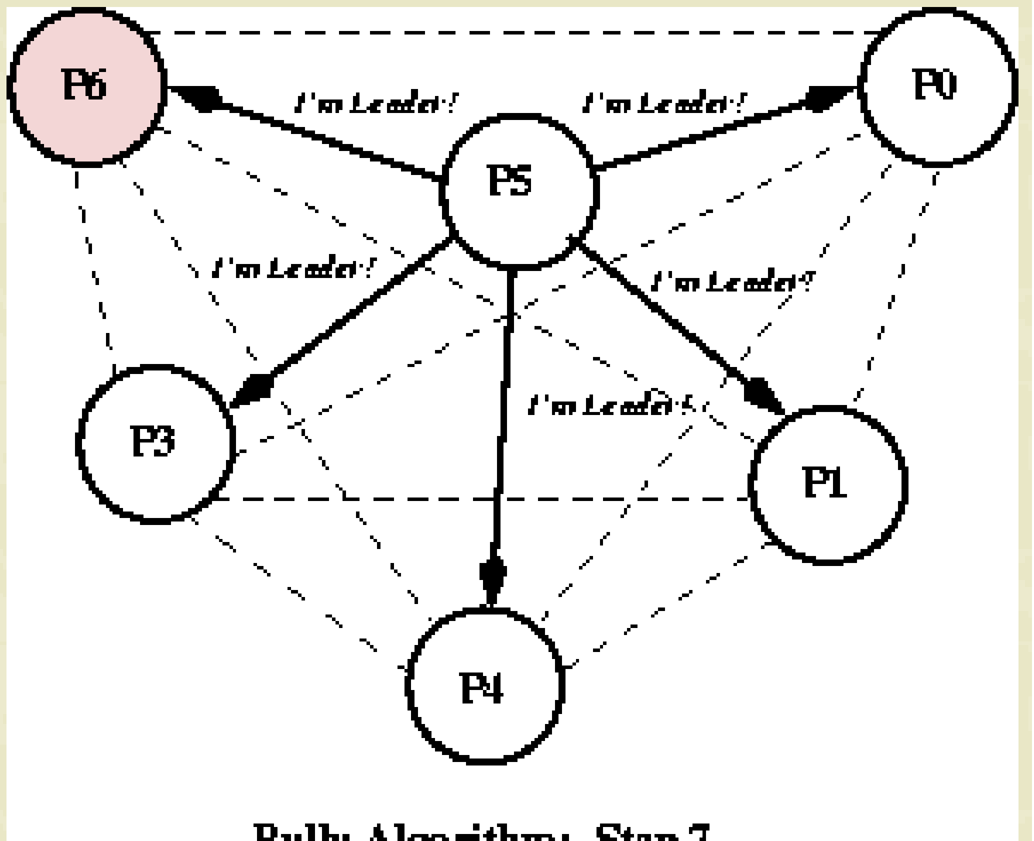
Bully Algorithm: Step 5

Bully Algorithm



Bully Algorithm: Step 6

Bully Algorithm



Bully Algorithm: Step 7

Bully Algorithm - analysis

- Complexity?
- Time to make a decision?
  - What if highest number process is flaky?
- Tolerant to network partition?
  - Nodes partitioned and each elects its own leader

Before we go on – A note on Byzantine Faults

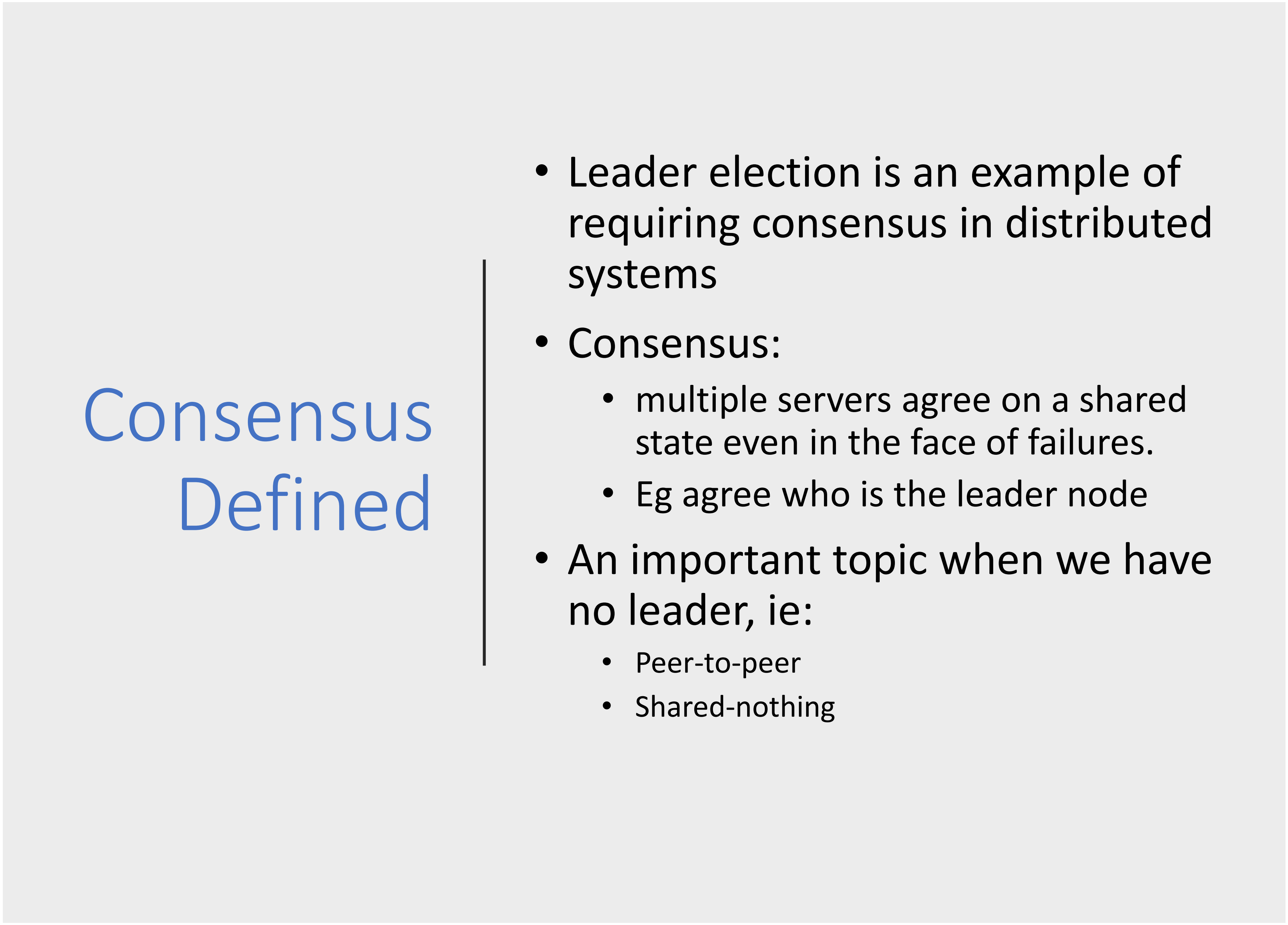
- A version of the N generals problem in which generals can lie,
- Analogous:
  - System compromised by intruders
- We assume systems don't lie in this course
- Some types of systems must be ***Byzantine Fault Tolerant*** though:
  - Flight control systems: radiation can corrupt bits in memory
  - Multi-organizational systems (eg Bitcoin)

Lab Exercise

Let’s Experiment with data models in MySQL

Consistency

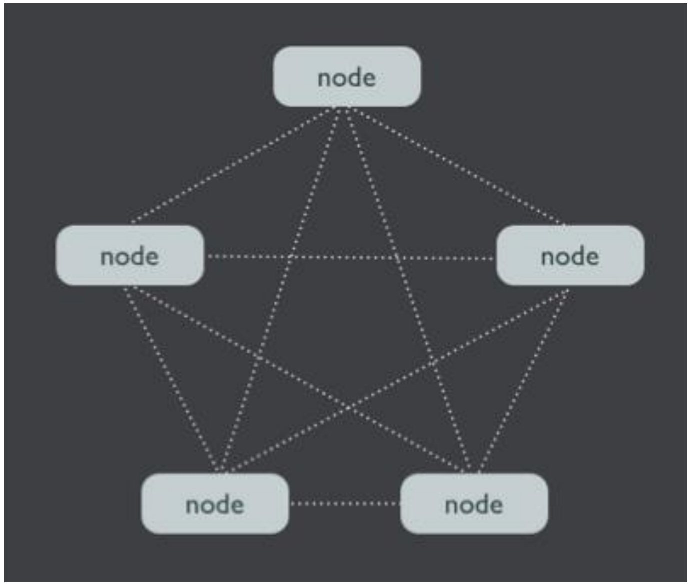
a majority of Core Servers in a cluster (N/2+1)



Consensus Defined

- Leader election is an example of requiring consensus in distributed systems
- Consensus:
  - multiple servers agree on a shared state even in the face of failures.
  - Eg agree who is the leader node
- An important topic when we have no leader, ie:
  - Peer-to-peer
  - Shared-nothing

Peer-based Replication



- "leaderless"
  - No leader copy
  - All copies equal
- Read to and write from any copy
- Updates propagate to replicas
  - Synchronously
  - Asynchronously
- Advantages?
- Disadvantages?

Algorithms

(more on this later)

- Leader election approaches:
  - https://en.wikipedia.org/wiki/Leader_election #Universal_leader_election_techniques
- General consensus
  - Paxos (complicated!!)
    - https://www.microsoft.com/en-us/research/publication/paxos-made-simple/
  - Raft
    - https://raft.github.io/
    - In Search of an Understandable Consensus Algorithm
    - http://thesecretlivesofdata.com/raft/
- Another approach:
  - https://www.mongodb.com/presentations/replication-election-and-consensus-algorithm-refinements-for-mongodb-3-2

Peer-based Replication

Advantage – can write to any node

Improved write throughput

How do we keep all copies the same if multiple clients write to same object on different replicas?

This brings us to replica consistency …

Consistency

- Two extremes:
- Strong consistency:
  - All replicas must exhibit same value to clients at all times
  - Important in e.g finance/banking
- Eventual Consistency
  - Some replicas may be stale (hopefully not for long)
  - Clients may see inconsistent values when reading the same value
  - Fine in e.g. Twitter feeds

Eventual Consistency

Promotes availability over consistency

Optimistic

Consistency achieved with some latency (eventually)

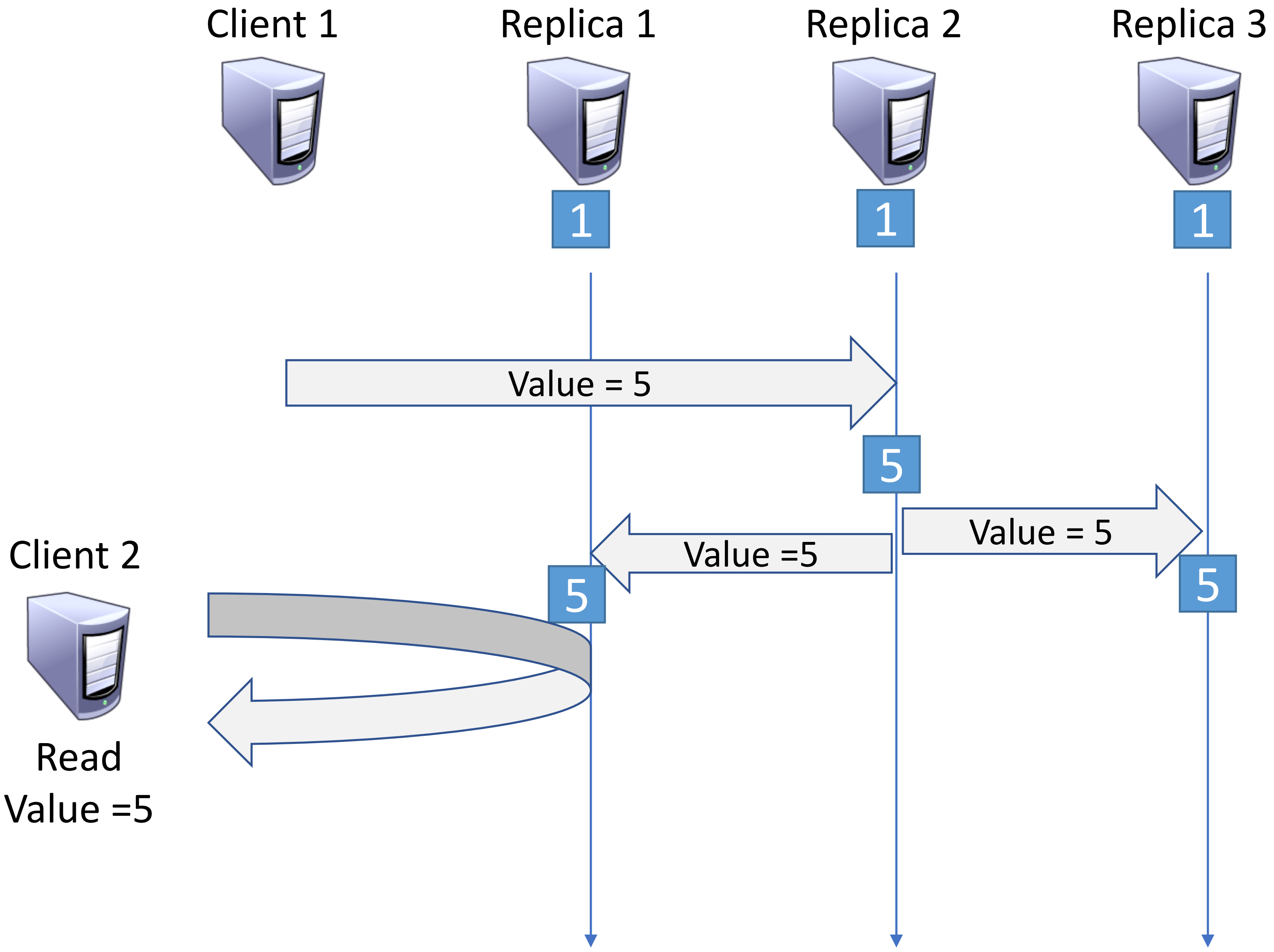
Availability achieved through partitioning and replication

Requires successful write to one or more replicas

Consistency achieved through background mechanisms

Latency for achieving consistency affected by number of replicas

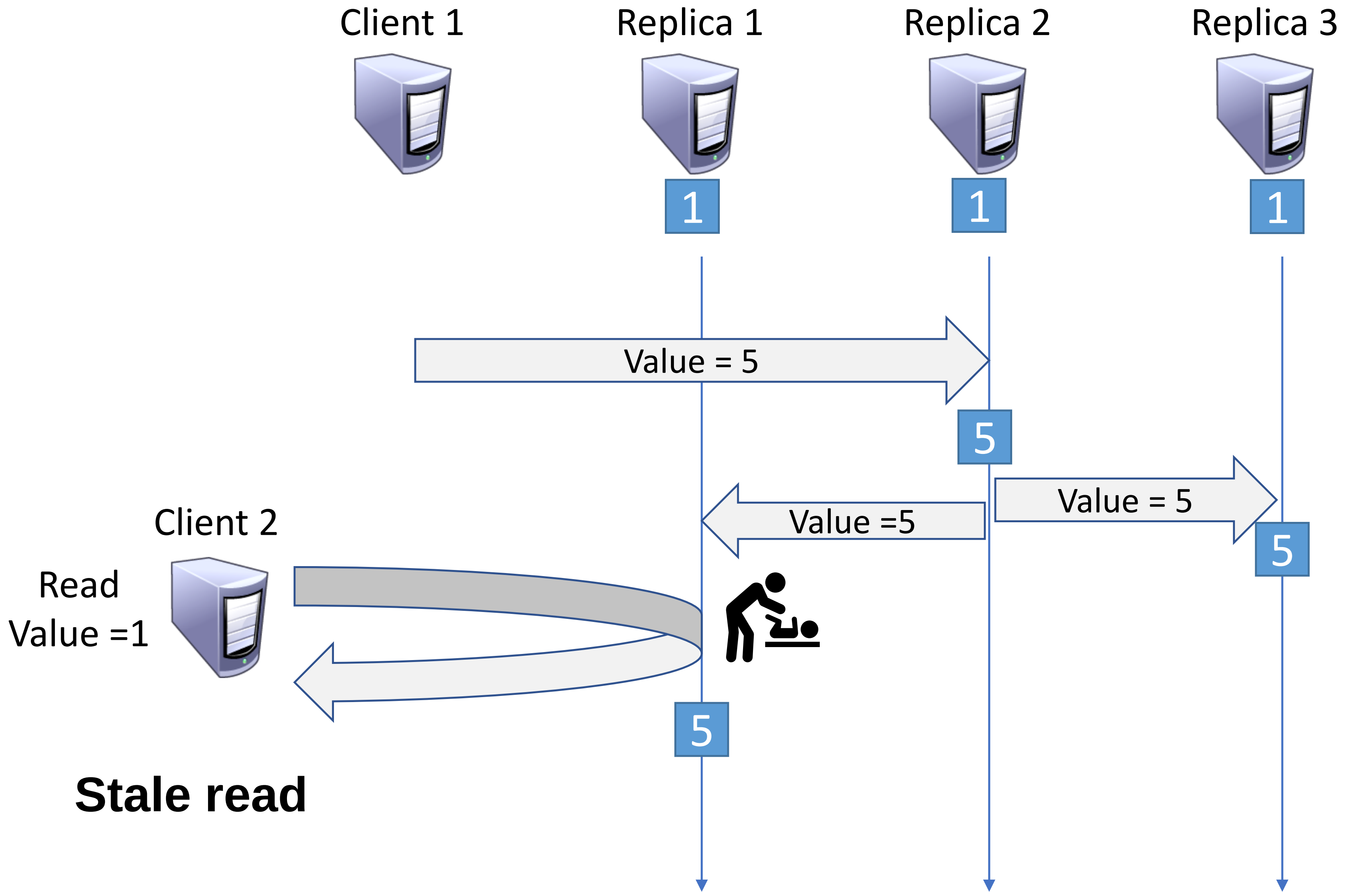
Eventual Consistency



Eventual Consistency

- Client writes are sent to (typically) one coordinating replica
- Coordinating replica:
  - Performs update and informs client
  - Sends update to other replicas
  - Receives acknowledgement of success (hopefully!)
  - If a replica is dead, things get tricky
    - Eg hinted handoff (more later)
- If N replicas:
  - Client may choose to write to >1 replicas
  - Decrease inconsistency window 🏆
  - Longer write latencies ☠

Eventual Consistency

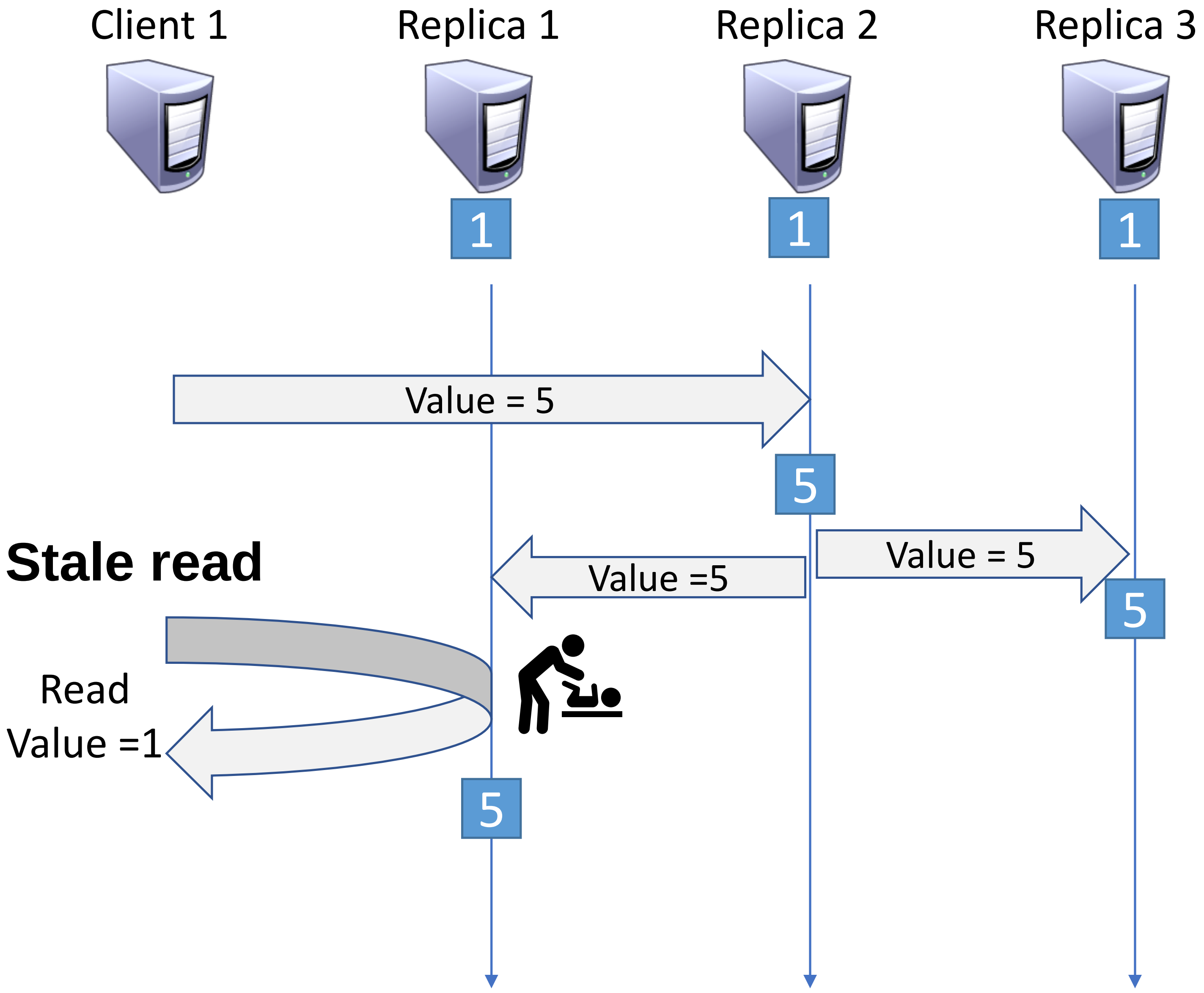


Stale read

Read your own writes

- if a process performs a write *w*, then that same process performs a subsequent read *r*, then *r* must observe *w*'s effects.
- Example:
  - User changes default credit card on a site
    - i.e. write new credit card details
  - User executes a transaction with default credit card
    - i.e. read credit card details
  - Expect to use new card (not old one!)

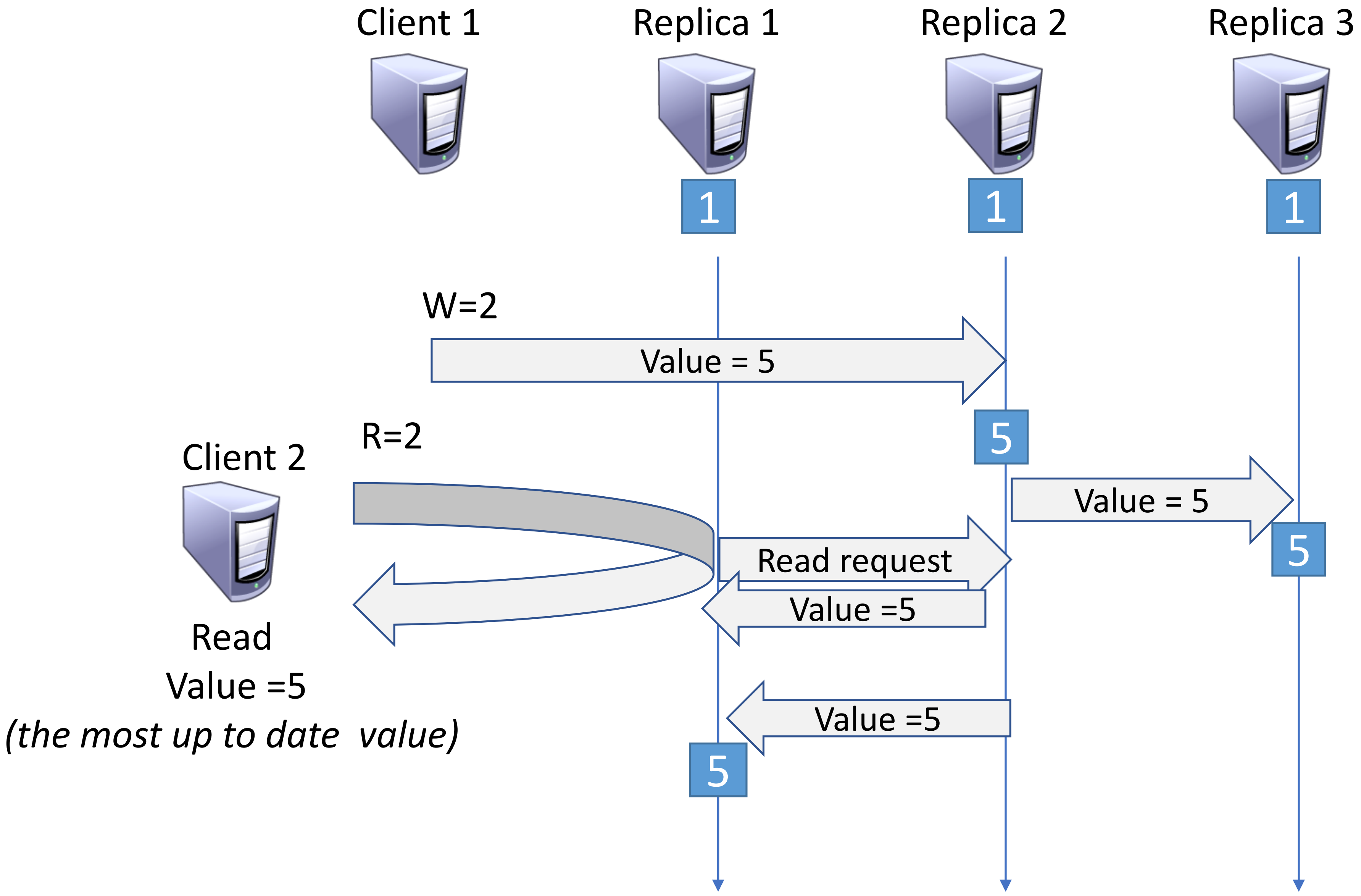
Eventual Consistency



Quorum Reads and Writes

- Quorum = majority
- In a leaderless system, let's define:
    - N is number of replicas
    - W = number of replicas that need to be updated on every write
    - R = number of replicas that need to be read to get latest value
- Assume data is versioned somehow (version number, timestamp) so latest value can be discerned

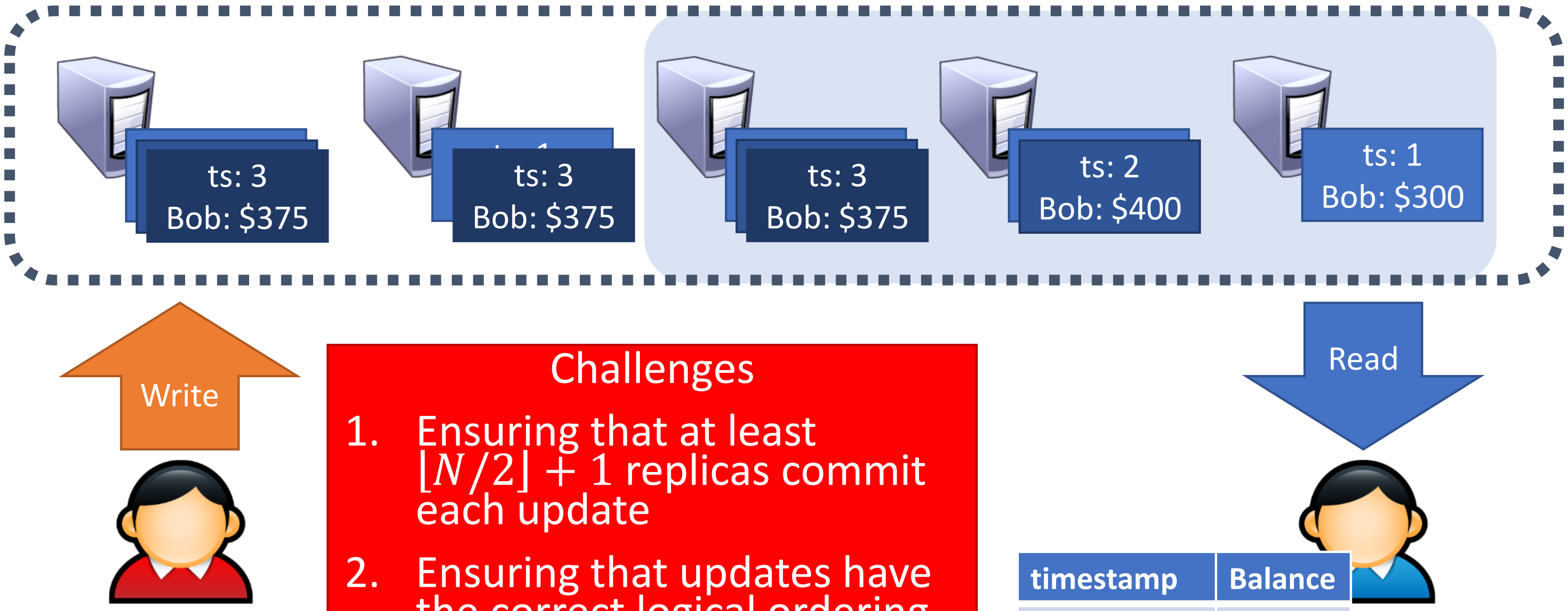
Eventual Consistency



Quorums

- If both W and R obey (N/2)+1, this is known as **quorum**
- Essentially a majority of replicas must accept a write or agree on a read
- If N=5, quorum is W=3 and R=3
    - Or W+R>N
- Read your own writes?
    - Overlap in set of nodes written to and read from
    - If everything works fine, yep!

High-Level Quorum Example



Challenges

1. Ensuring that at least $\lfloor N/2 \rfloor + 1$ replicas commit each update
2. Ensuring that updates have the correct logical ordering

timestamp	Balance
1	$300
2	$400
3	$375

Advantages of Quorums

- **Availability**: quorum systems are more resilient in the face of failures
- **Efficiency**: can significantly reduce communication complexity
  - Do not require all servers in order to perform an operation
  - Requires a subset of them for each operation

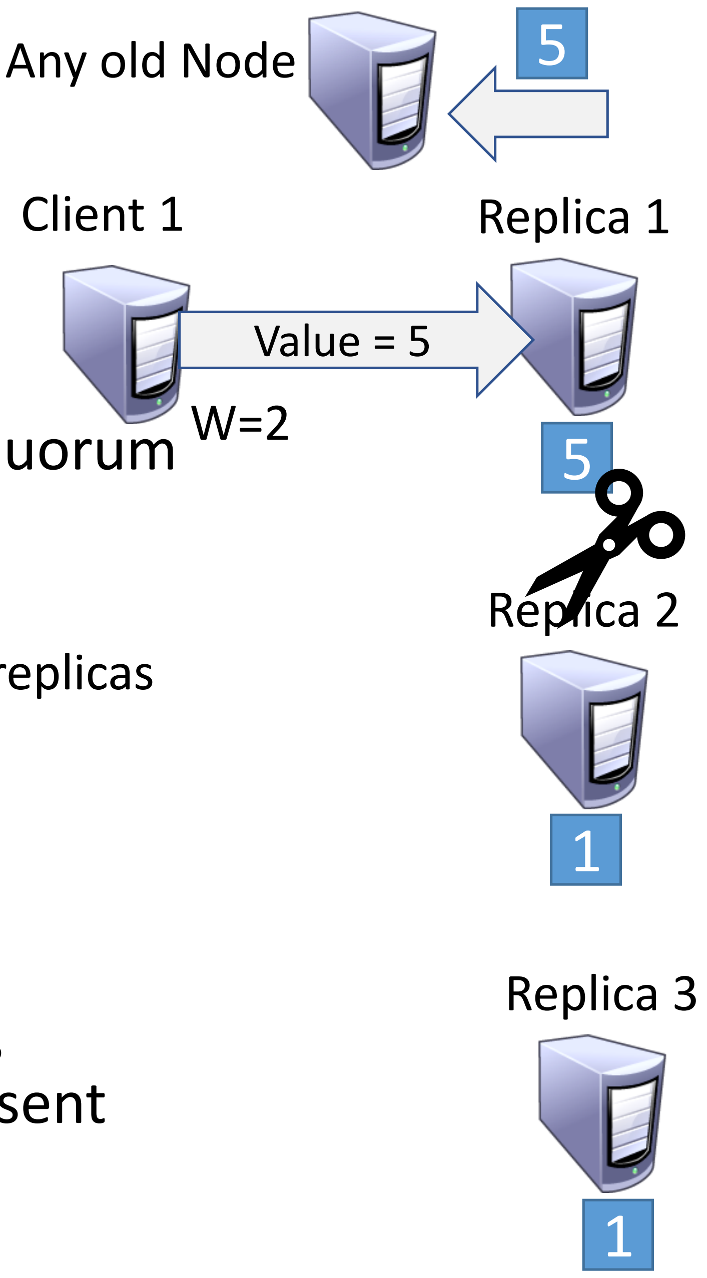
Quorum ‘Quirks’

- A write in progress when read issued on same key
  - New value not persisted on all replicas
  - What is returned?
- Write succeeds on some replicas but fails on others (e.g. partition, power fails) such that W replicas not updated
  - Writes are not rolled back on successful replicas
  - Error returned to client – retry?
  - If no retry, reads may return stale data and object will be inconsistent until next write
- ‘Sloppy’ quorums … next topic!!

Quorum Consistency

- We can adjust values to achieve required consistency levels:
  - If N=5, R=5, W=1, what is effect?
  - If N=5, R=1, W=5, what is effect?
  - If N=5, R=1, W=5, and one replica is down, what is effect?
  - If N=5, R=2, W=1, what is effect?

Hinted Handoffs

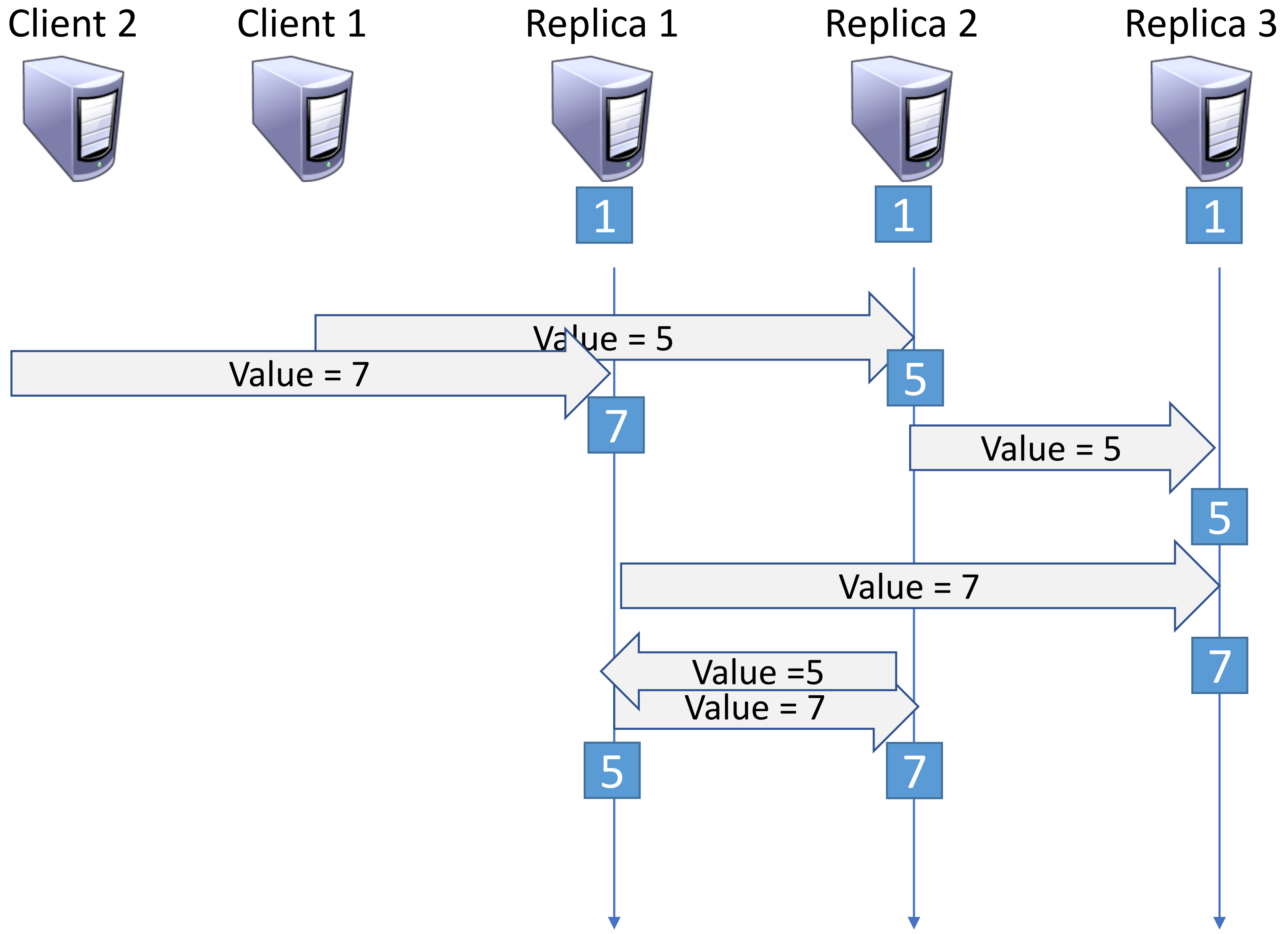


- Assume network partition:
- Client writes/reads cannot achieve quorum
- If availability is key requirement
    - Accept write even with W=2
    - Write to a reachable node and update replicas when they become available
    - Known as a Sloppy Quorum
        - As opposed to a strict quorum
        - Assured durability on N nodes
        - Temporary home for value
- Hinted handoff occurs when replicas become available and recent values sent from temporary home

Handling Conflicts

- Writes to the same key can be sent to different replicas 'at the same time'
- Two writes to same key can overlap as replicas are updated
    - Variable network latencies
    - Node failures
    - Network partition
    - Slow nodes
- Essentially writes do not know about each other as they occur on different set of nodes

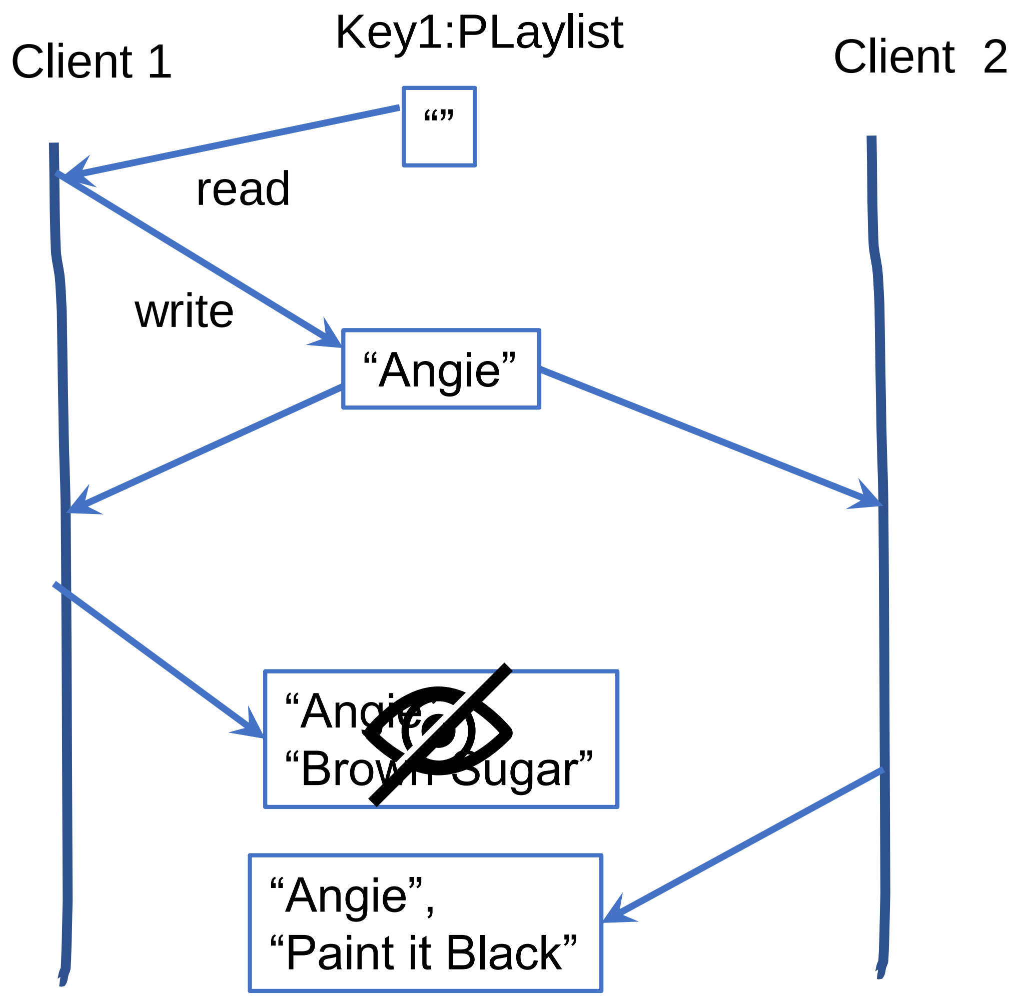
Conflicts


Client 2
Client 1
Replica 1
Replica 2
Replica 3
1
1
1
Value = 5
Value = 7
5
7
Value = 5
5
Value = 7
7
Value =5
Value = 7
5
7

Simple Solution – Last Writer Wins

- Use timestamps to determine most up to date value
- If we see conflicting writes, simply persist the most recent value at all replicas
- Ensures replicas remain consistent
- Issue: which clock generates timestamp?
  - Client's?
  - Coordinator's?
  - Replicas?

Last Writer Wins



Lose Some Writes!!

- Effectively a race condition
- Writes can be overwritten
  - Ie not durable
  - But reported as successful to the client!
    - Written to W replicas
- LWW leads to data loss
- Safe only when a key is written once and afterwards is regarded as immutable
  - Changes written to new rows and merged programmatically

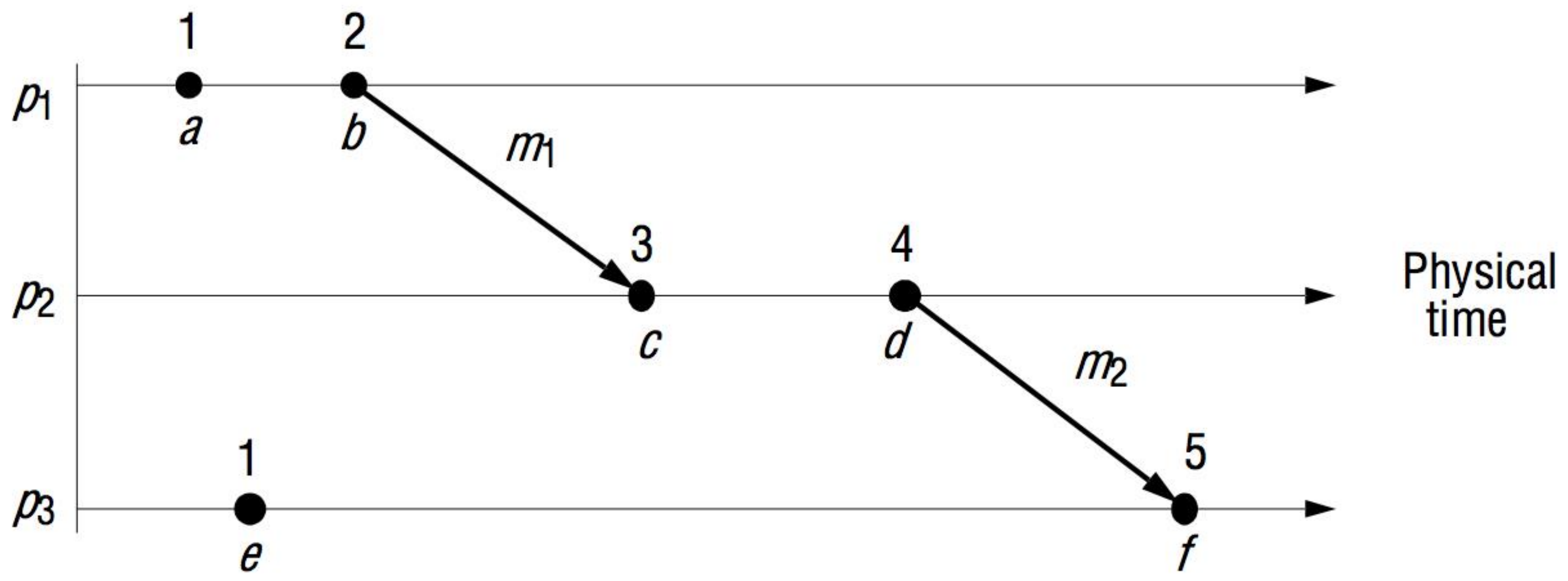
Conflicts – a Better Solution - Causality

- Track causality
- Example:
  - Two users login to systems at 8.00am on local machines
  - Clock drift means we don’t know which logged on first?
  - If user1 logs in and sends message to user2, and message is received before user2 logs in, we know:
    - User1 login ‘happened before’ User2 login
  - If message received after users2 logs in
    - We still don’t know the order
    - Events are not causally related
    - i.e concurrent

Lamport Clocks

- Algorithm to provide a partial ordering of events in a distributed system
- Every process maintains a local logical clock
    - Effectively a counter, initialized to zero
    - When a process sends a message or executes an internal step, it sets clock $\leftarrow$ clock + 1
    - assigns the resulting value as the clock value of the event.
- If it sends a message:
    - it piggybacks the resulting clock value on the message.
- When a process receives a message,
    - it sets its clock $\leftarrow$ max(clock, message timestamp)+1
    - the resulting clock value is taken as the time of receipt of the message.

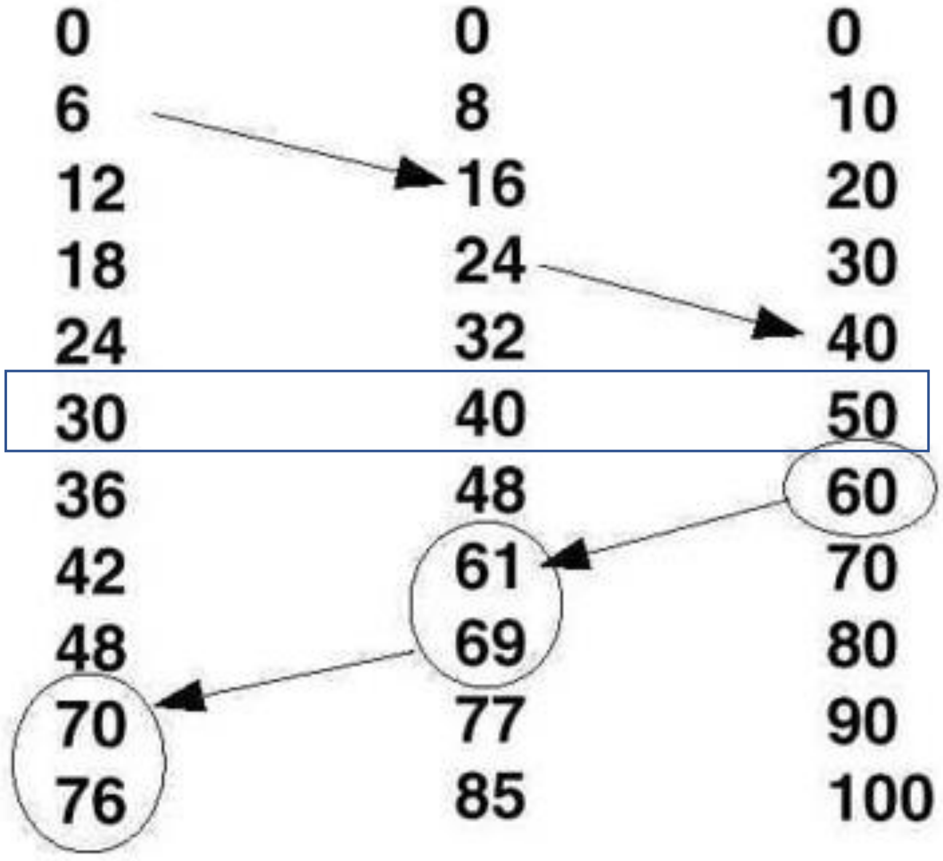
Lamport Clocks



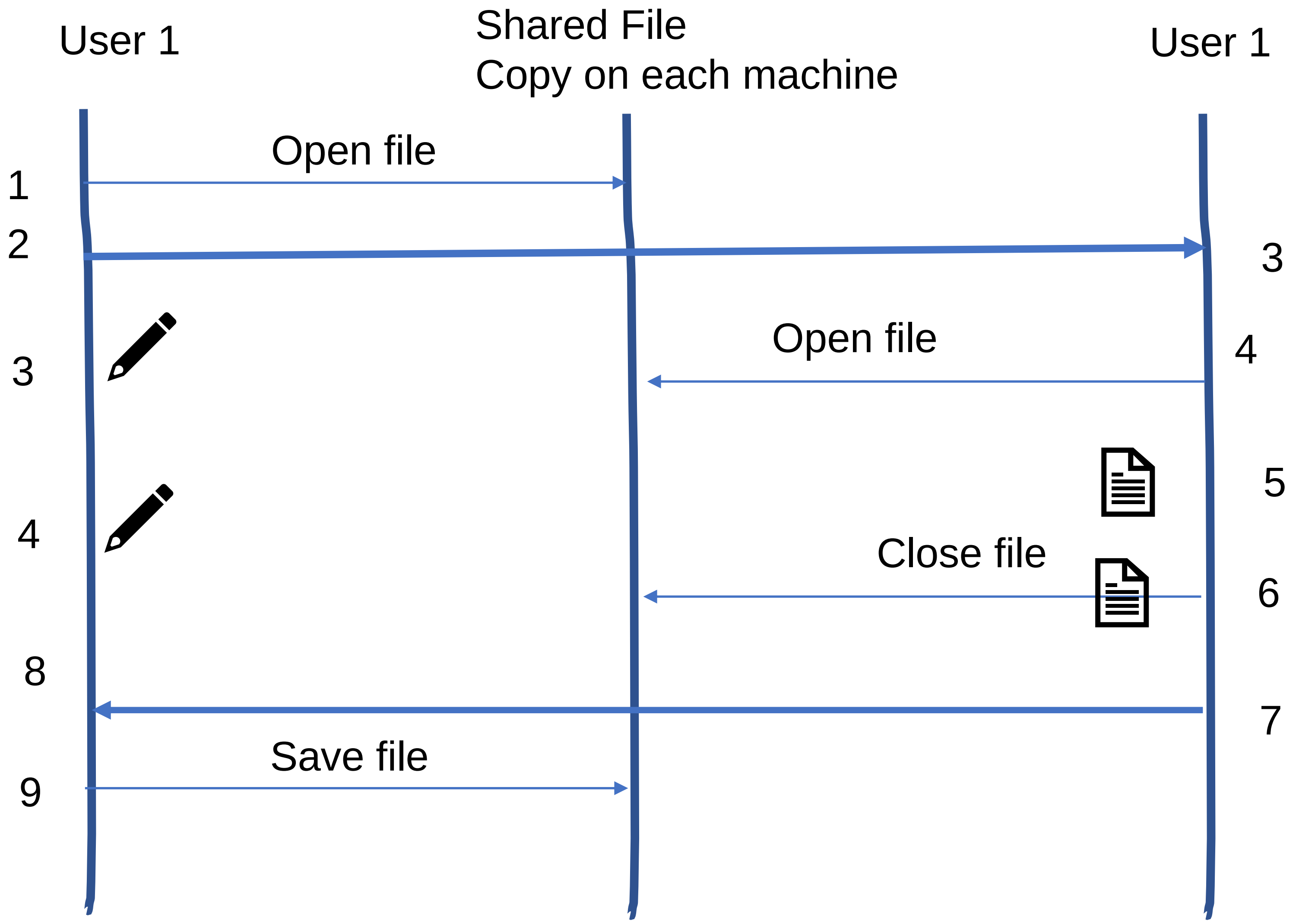
Lamport Clocks

- Identify a **happened-before** ordering that is described numerically
  - if C(a)<C(b), C(a) happened-before C(b).
  - Partial causal ordering
- Only meaningful in terms of messages flowing between a group of processes
  - If *a* and *b* are two events in the same process, and *a* comes before *b*, then *a* -> *b.*
  - If *a* denotes the sending of a message and *b* the receipt of that message, then *a* -> *b.*
  - If *a* -> *b* and *b* -> *c*, then *a* -> *c*.

Lamport Clocks – Partial Ordering



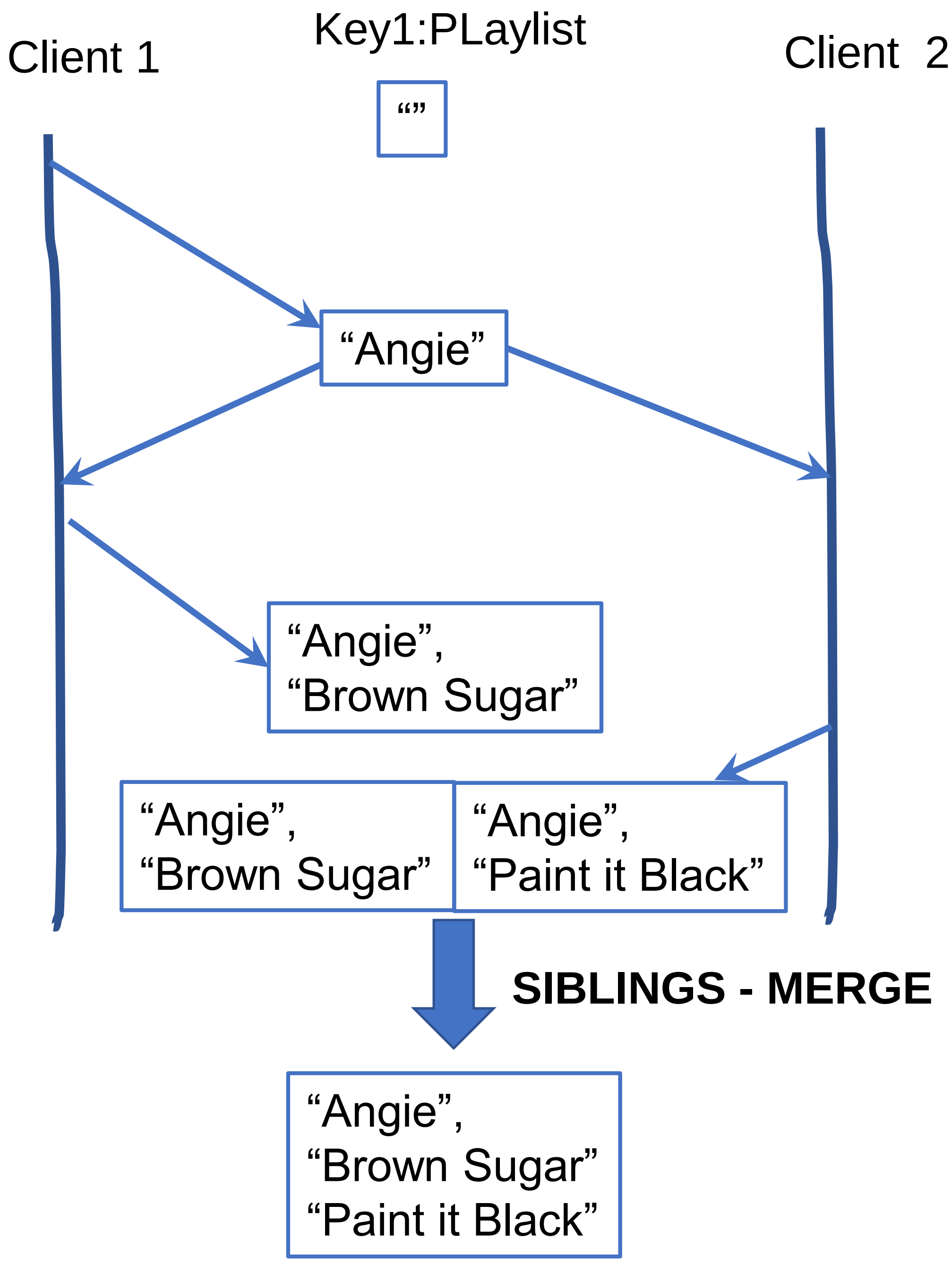
Lamport Clocks - Example



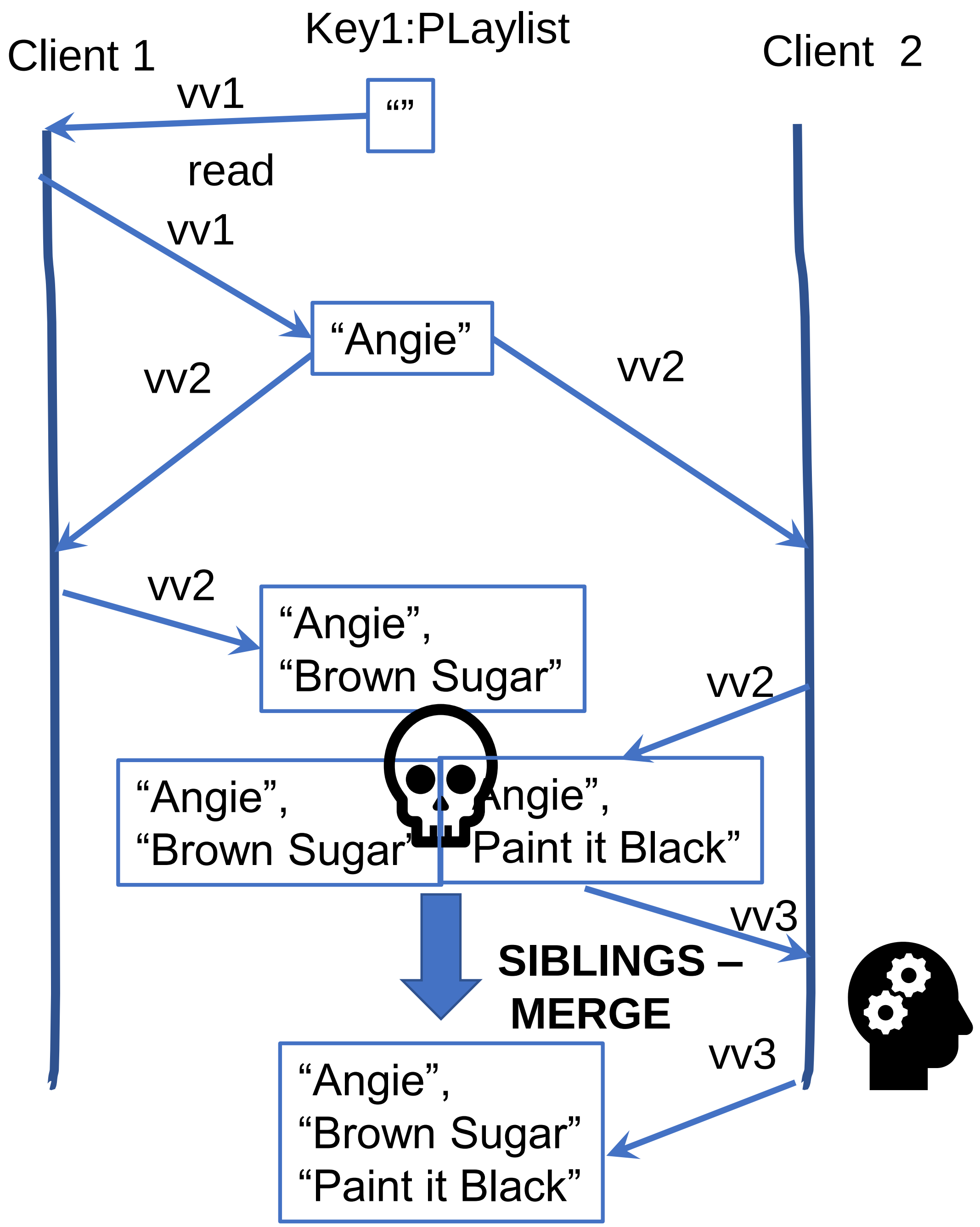
Conflicts

- How do we detect the siblings for a key?
  - Versioning
    - Vector Clocks
    - Version Vectors
- Who does the merge?
  - The client – database presents siblings and client decides what to do
    - Maybe with help of a person!!
  - Conflict-free Replicated Data Type (CRDTs)
    - Sets, lists, maps, counters, ordered lists, etc
    - Ongoing research …

How do we handle conflicts?



Basic Algorithm



Conflict Resolution

- Server creates a version for each key/replica
- Clients must read key and get associated version before writing
- Client write sends new value along with version that was read
- If server version of key same as version number in write, server updates value and creates new version
- If server version not same as version number in write, server creates siblings due to a concurrent write (also if no version sent with write)
- When client read is returned siblings, it must merge values and write to create a new version number

Strong Consistency

(next week!)

Summary

- Relational databases scale best vertically and suited to business data
- Different applications can tolerate weaker guarantees and need massive scalability
- NoSQL databases based on leader-follower and leaderless architectures
- Eventual consistency is achievable in leaderless systems using quorums (as long as everything behaves!)
- Conflict resolution possible in leaderless systems using versioning