Fault-Tolerant Distributed Systems

Rui Carlos Oliveira
rco@di.uminho.pt

Informatics Department
Minho University

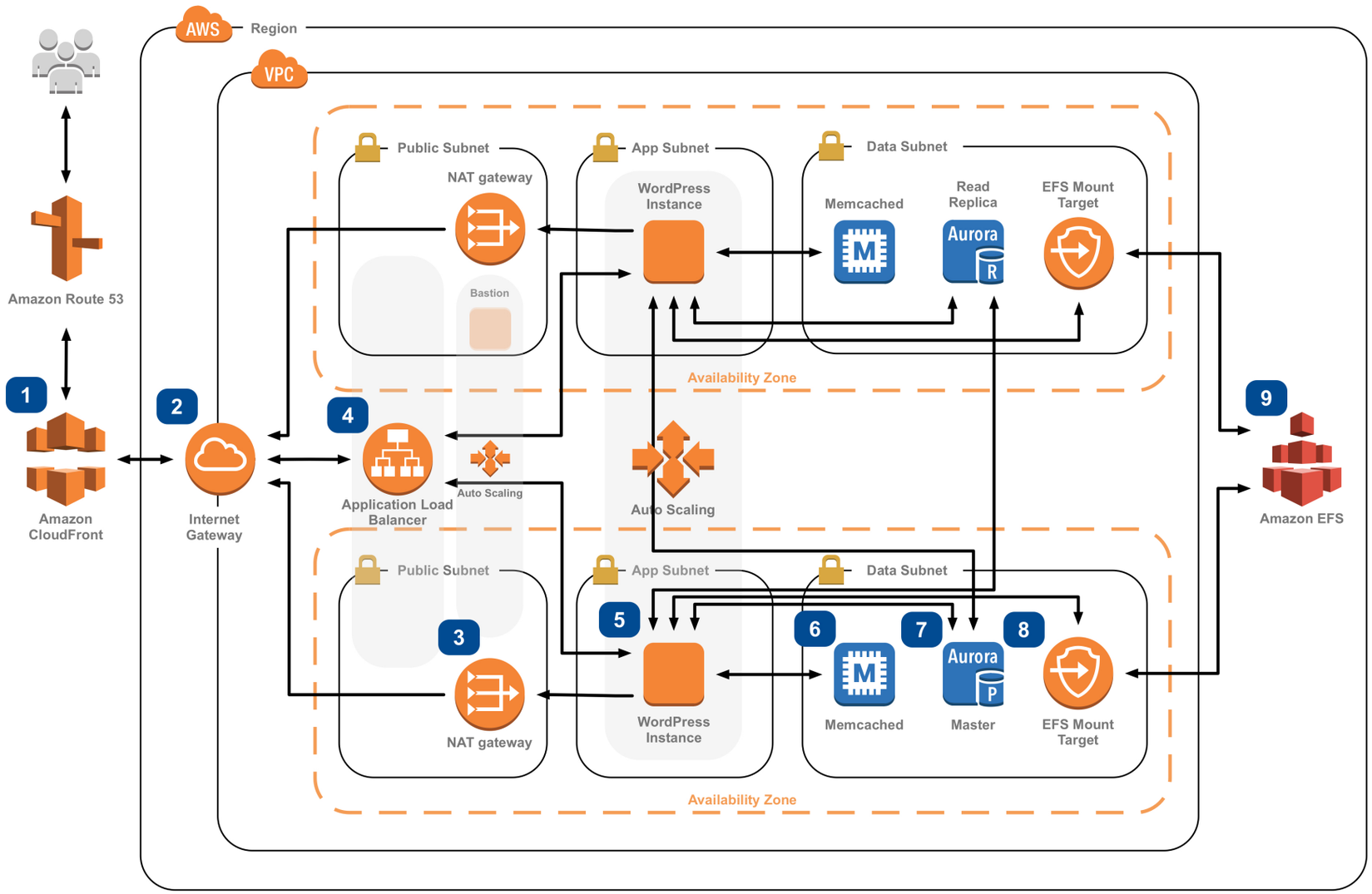
2021/22

Syllabus

- **Introduction to fault-tolerant distributed systems**
- Models of distributed systems and related faults
- Data replication
- Distributed consensus
- State machine replication
- Database replication

Introduction to fault-tolerant distributed systems

- Software systems are ever more complex, bigger, and full of intricate dependencies



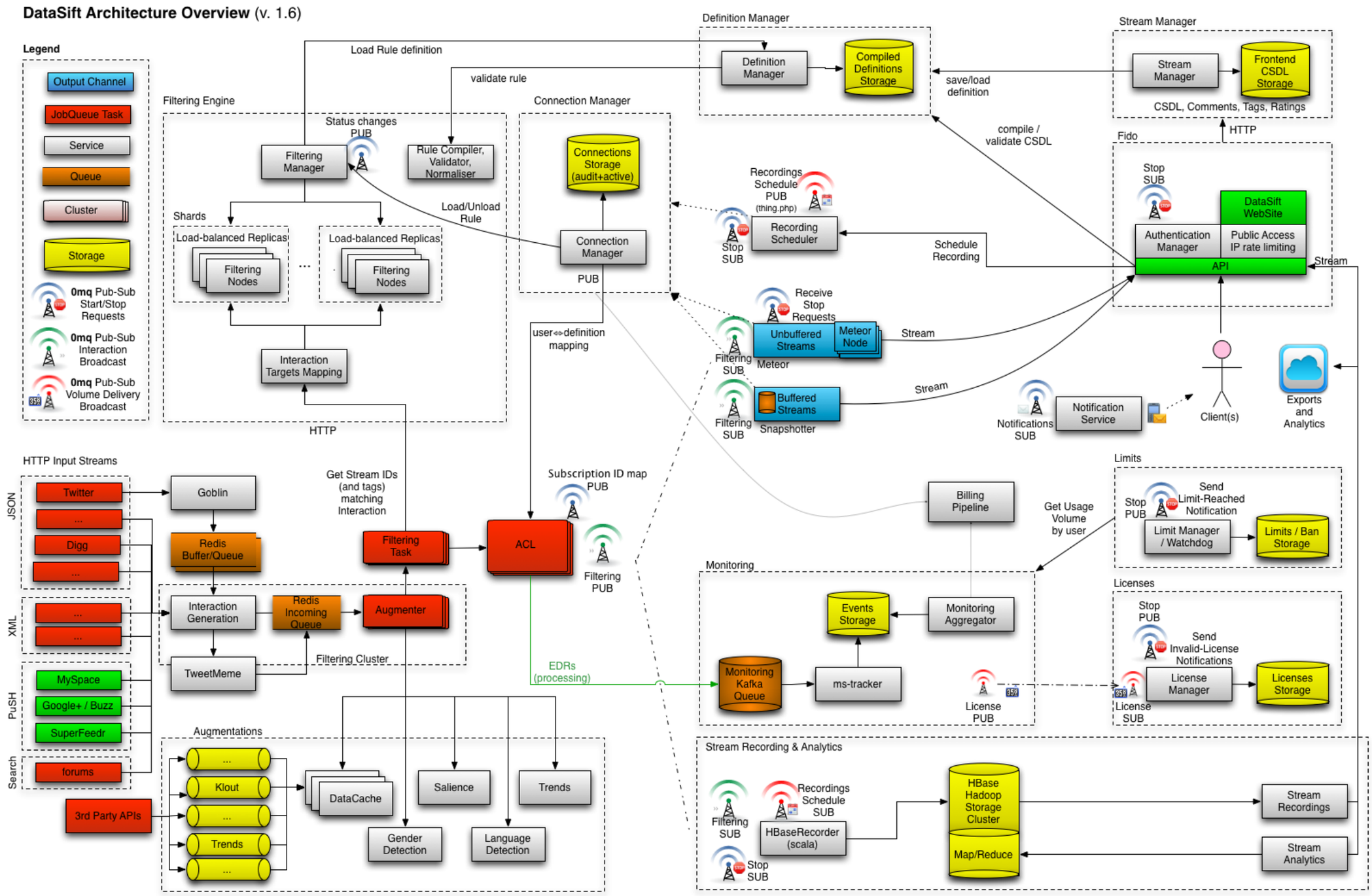
1. Static and dynamic content is delivered by **Amazon CloudFront**.
2. An **Internet gateway** allows communication between instances in your VPC and the Internet.
3. **NAT gateways** in each public subnet enable Amazon EC2 instances in private subnets (App & Data) to access the Internet.
4. Use an **Application Load Balancer** to distribute web traffic across an Auto Scaling Group of Amazon EC2 instances in multiple AZs.
5. Run your WordPress site using an **Auto Scaling group** of **Amazon EC2 instances**. Install the latest versions of WordPress, Apache web server, PHP 7, and OPcache and build an Amazon Machine Image that will be used by the Auto Scaling group launch configuration to launch new instances in the Auto Scaling group.
6. If database access patterns are read-heavy, consider using a WordPress plugin that takes advantage of a caching layer like **Amazon ElastiCache** (**Memcached**) in front of the database layer to cache frequently accessed data.
7. Simplify your database administration by running your database layer in **Amazon RDS** using either Aurora or MySQL.
8. Amazon EC2 instances access shared WordPress data in an Amazon EFS file system using **Mount Targets** in each AZ in your VPC.
9. Use **Amazon EFS**, a simple, highly available, and scalable network file system so WordPress instances have access to your shared, unstructured WordPress data, like php files, config, themes, plugins, etc.

amazon web services

AWS Reference Architectures

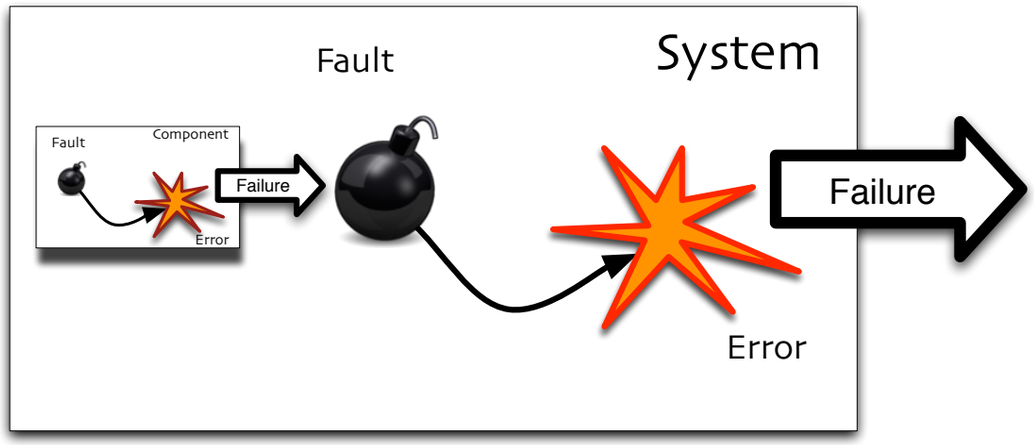
Introduction to fault-tolerant distributed systems

- Software systems are ever more complex, bigger, and full of intricate dependencies



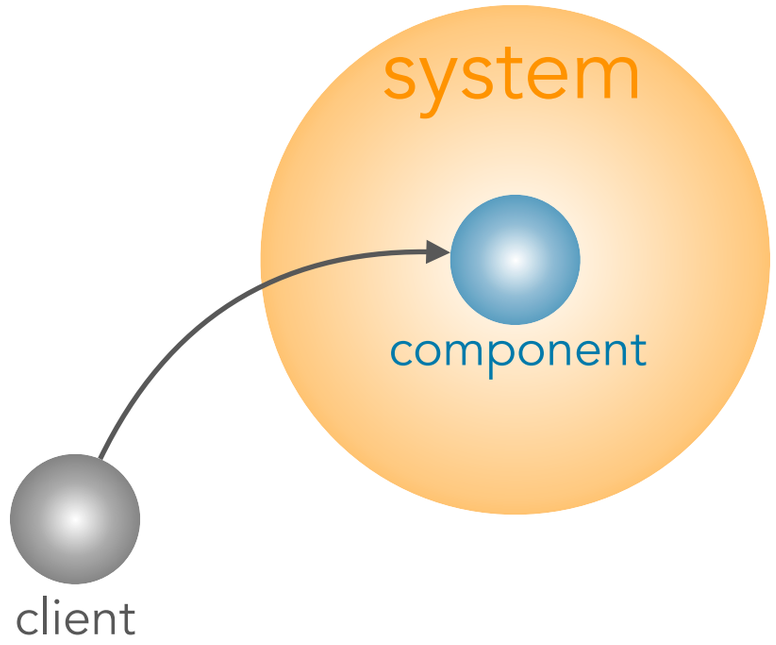
Introduction to fault-tolerant distributed systems

- Goal of the course
  - Learn how to make a software system fault-tolerant
- Impact of faults
  - Performance
  - Unavailability
  - Misbehaviour
- The challenge
  - Make *fault-tolerant X* service as available as possible (availability)
    - 99% to 99.999% of uptime (3.65 days to 5.2 minutes of downtime)
  - Any behaviour of *fault-tolerant X*, needs to be a behaviour of X (safety)

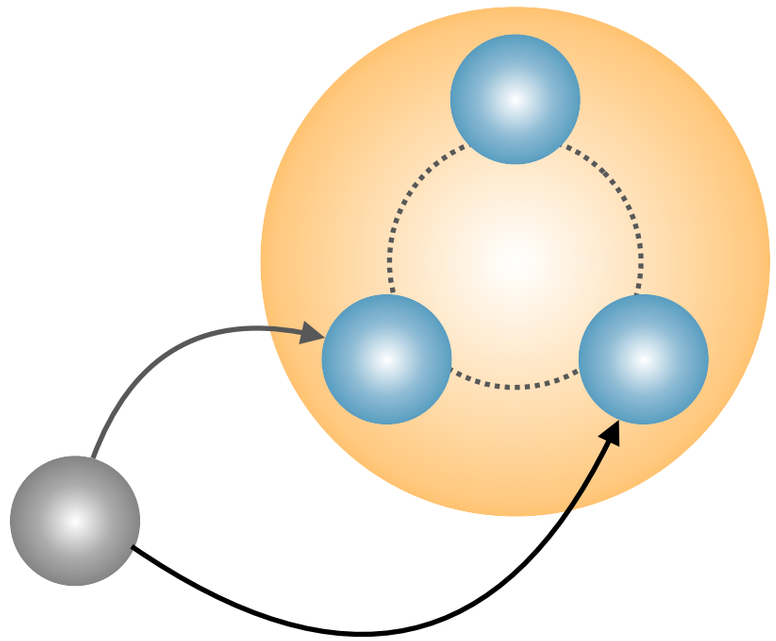


Introduction to fault-tolerant distributed systems

- Fault tolerance requires replication
  - As any system component is prone to fail
  - Any vital component needs to be replicated
    - Remove any *Single Point Of Failure (SPOF)*

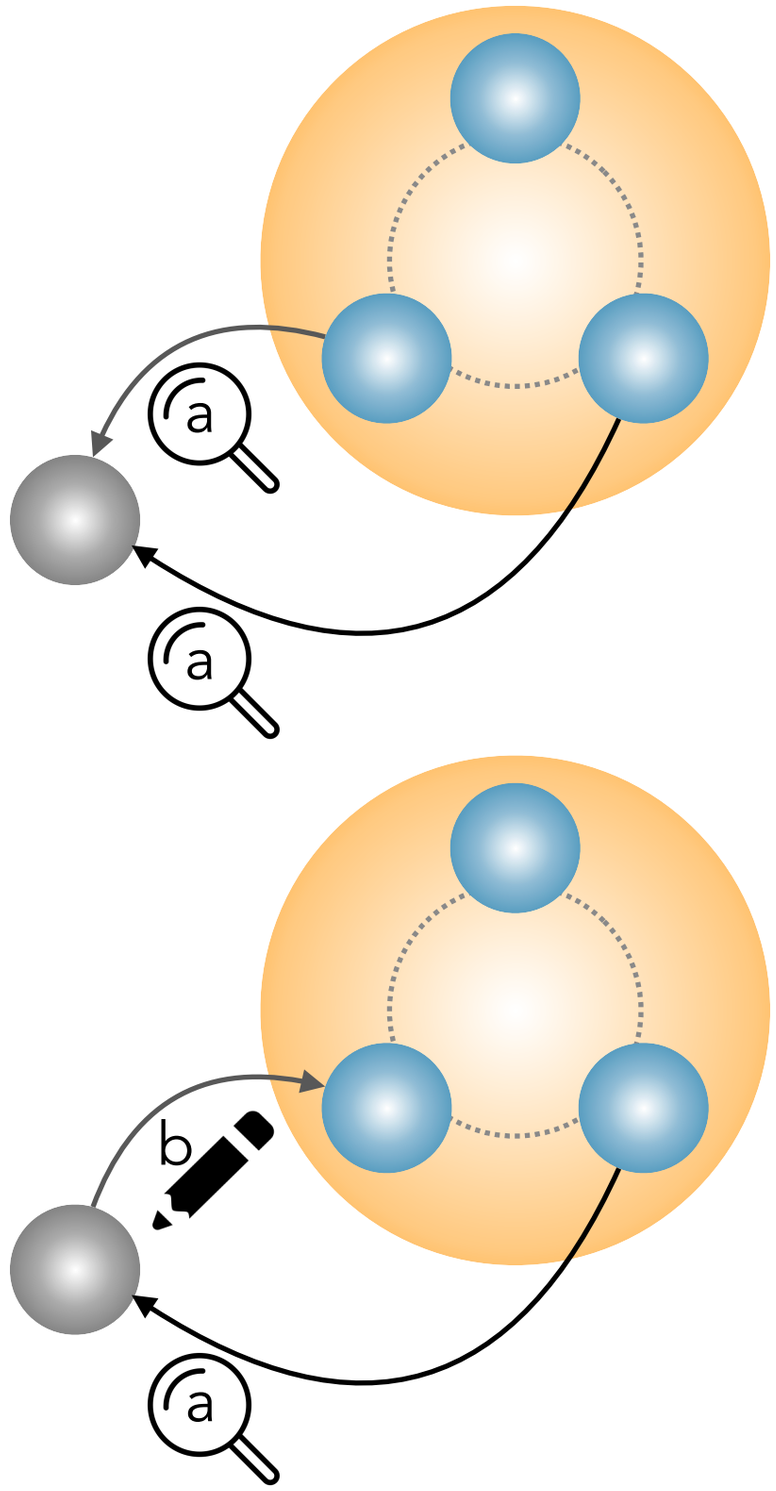


- Distributed systems suit replication well
  - Modularity
  - Incremental redundancy
  - Graceful degradation
  - Heterogeneity
  - Encapsulation



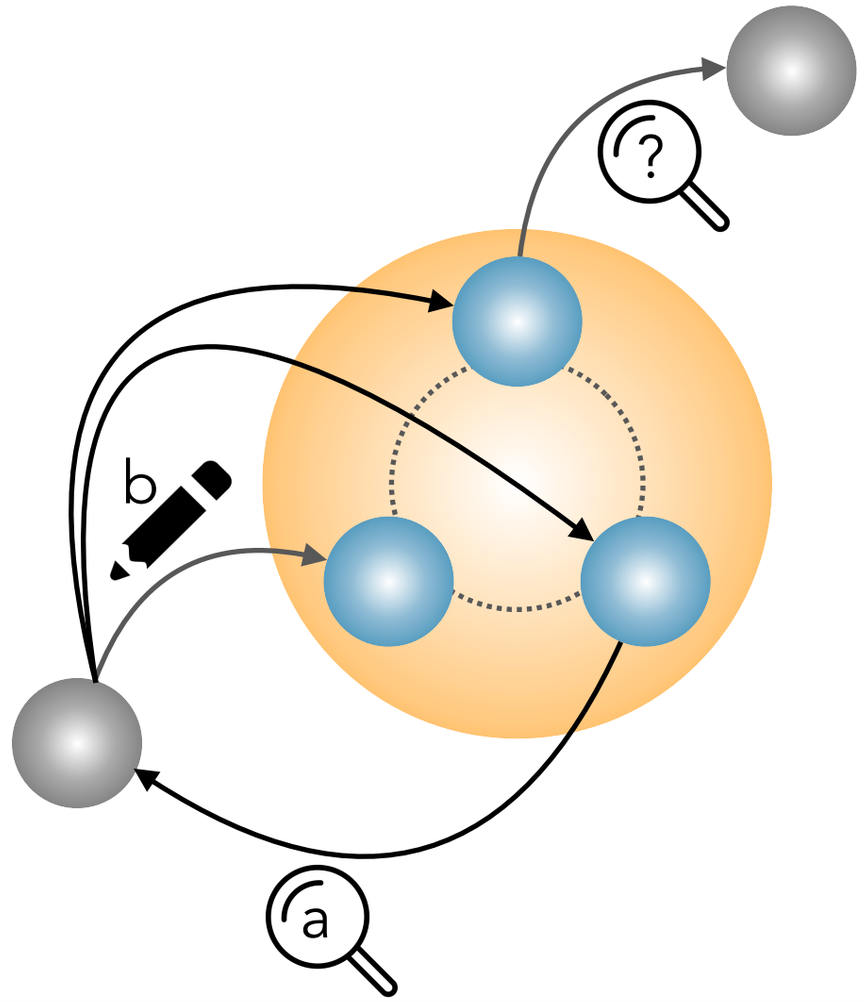
Introduction to fault-tolerant distributed systems

- Any replication approach depends on
  - The component's state mutability



Introduction to fault-tolerant distributed systems

- Any replication approach depends on
  - The component's state mutability
  - The concurrency model



Introduction to fault-tolerant distributed systems

- Any replication approach depends on
  - The component's state mutability
  - The concurrency model
  - The type and number of faults (fault model)
  - The workload bias (usually between reads and writes)

Introduction to fault-tolerant distributed systems

Reading material

- A. Avizienis, J-C Laprie, B. Randell, and C. Landwehr
  "Basic Concepts and Taxonomy of Dependable and Secure Computing",
  IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING, VOL. 1, NO. 1, JANUARY-MARCH 2004

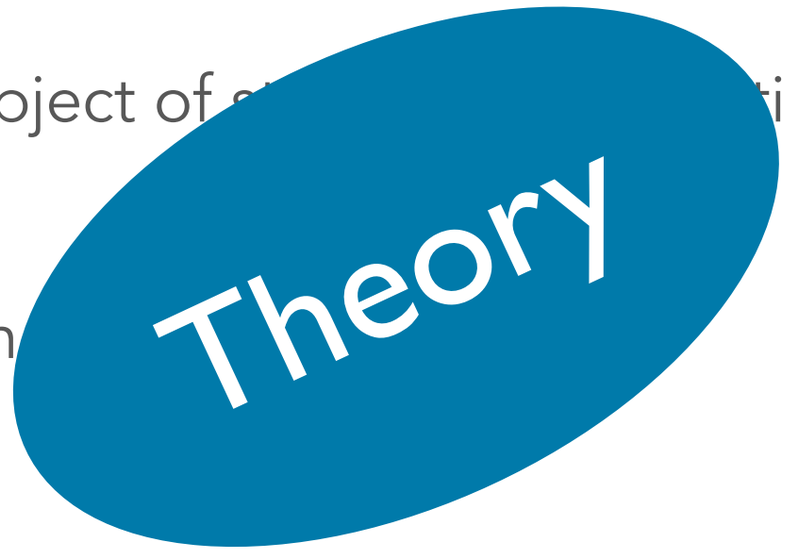
Syllabus

- Introduction to fault-tolerant distributed systems
- **Models of distributed systems and related faults**
- Data replication
- Distributed consensus
- State machine replication
- Database replication

Models of distributed systems and related faults

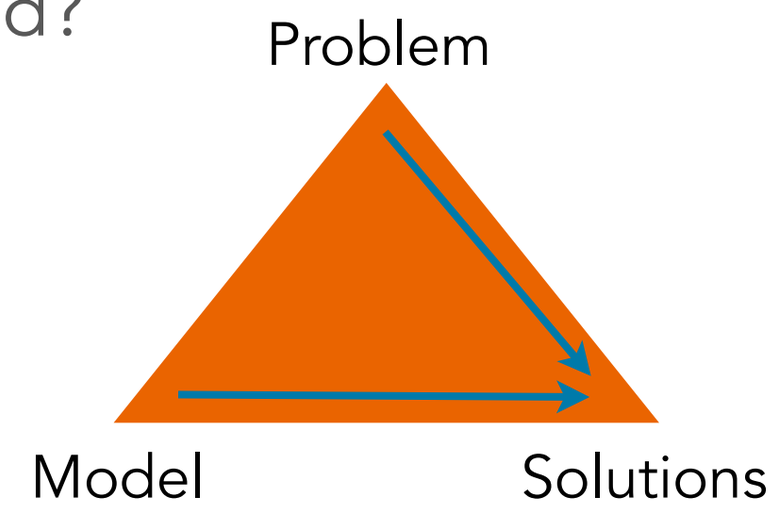
- Experimental observation
  - Build things and observe how they behave in va
  - Experience enables us to build things for that have been studied

- Modelling and analysis
  - Formulate a model by simplifying the object of s ting a set of rules to define its behaviour
  - Analyze the model and infer consequen



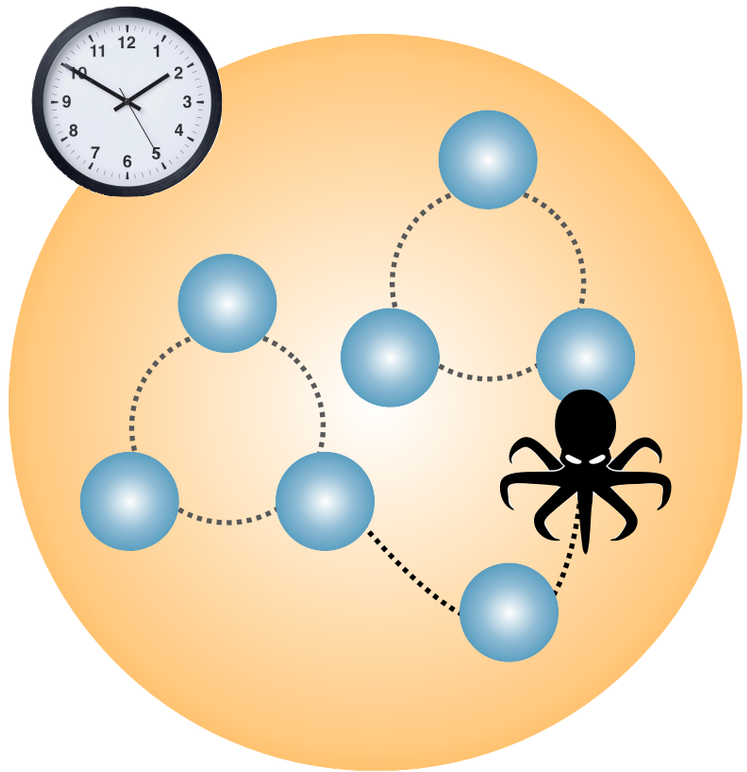
Models of distributed systems and related faults

- Modelling a system consists of identifying and precisely characterising its entities and the behavioural rules that are relevant for the problems in hand
- A model needs to be tractable in that it does not contain uninteresting details that would impair the desired analysis and, at the same time, it needs to be accurate so that it captures all attributes affecting the phenomena of interest
- We seek answers to two fundamental questions:
  - Feasibility: What kind of problems can be solved?
  - Cost: What is the relative cost of solutions?



Models of distributed systems and related faults

- A minimal model for a distributed system consists of a finite set of autonomous sequential processes connected through a finite set of point-to-point communication channels
- We picture the existence of an adversary with a limited power to impair the correct behaviour of processes or channels
- For the problems we will be considering we model the system as:
  - A finite set of sequential processes
  - A finite set of communication point-to-point channels
  - An adversary
  - Rules governing the behaviour of the above



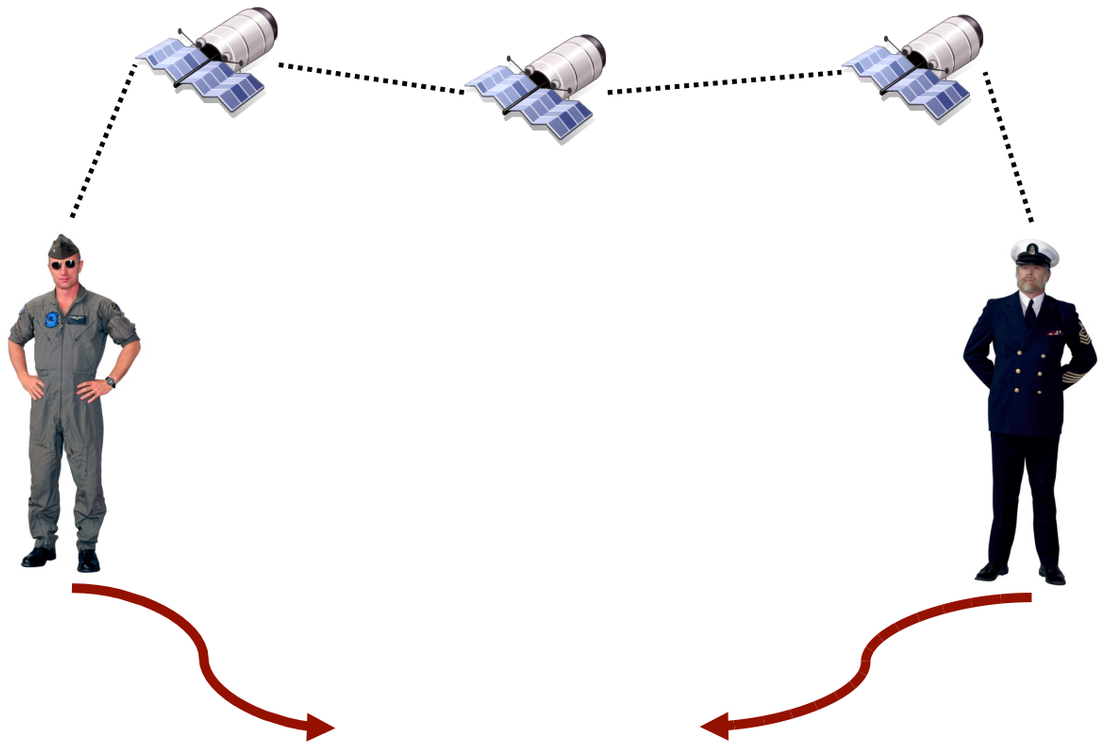
Models - What kind of problems can be solved?

- System model:
  - Two processes, A and B, communicate by sending and receiving messages
  - Neither process can fail. However, the channel can experience transient failures, resulting in the loss of a subset of the messages that have been sent

- A **Coordination problem**:
  - Devise a protocol where either of two actions α and β are possible, but (i) both processes take the same action and (ii) neither takes both actions

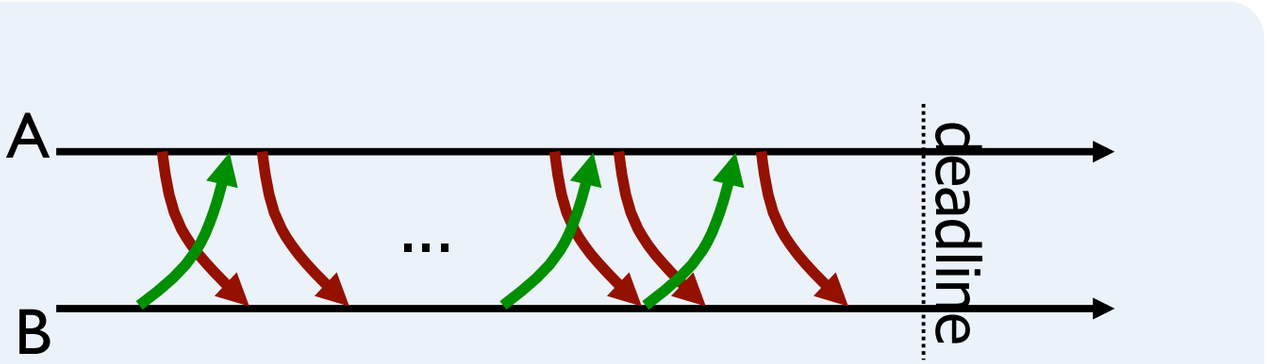
Problem: Agree on a simultaneous attack

Both attack: Win!
Only one attacks: Dead
No one attacks: Safe but…



Models - What kind of problems can be solved?

- Any solution will consist in a message exchange between A and B
- Without channel failures the solution would be straightforward
- Interesting case: Messages received is a subset of messages sent
- Consider an hypothetical solution:



- Does the last message matter?
- And the second last?
- What if the “solution” was the one with the minimum message exchange?

Models - Synchrony and Asynchrony

- Of major importance when modelling a distributed system are the assumptions we make regarding the synchrony or asynchrony of processes and communication channels

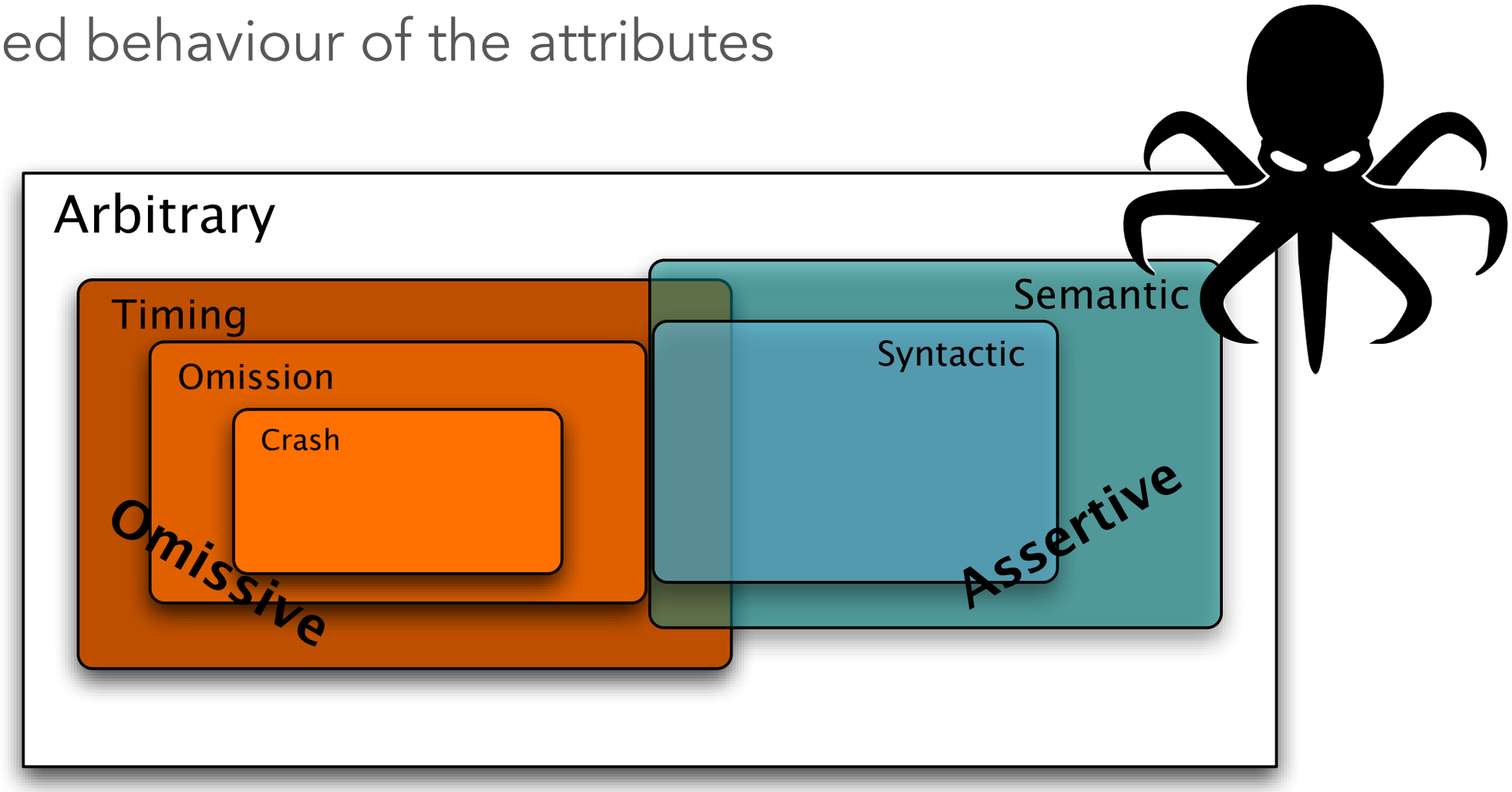
- Modelling an attribute as asynchronous means that we make no assumptions, absolute or relative, with respect to the time it takes to perform an action. An asynchronous model leads to universal solutions with respect time
- An attribute is synchronous if we assume it take at most t time units to perform an action
- Models can be semi-synchronous in that the attributes are synchronous
  - But *t* is unknown
  - But only eventually

Models - What is the relative cost of solutions?

- System model A:
  - A set of processes P_1,..., P_n. Each process P_i has a unique integer identifier id_i
  - There is no adversary
  - All processes start executing at the same time but **no synchrony assumptions**
- An Election problem:
  - Devise a protocol to elect a unique leader eventually known by all processes
- System model S:
  - Processes take at most t_p per step and messages take at most t_m to be delivered
- Homework:
  - Devise a solution in S that is not a solution in A. How do they compare?

Models of distributed systems and related faults

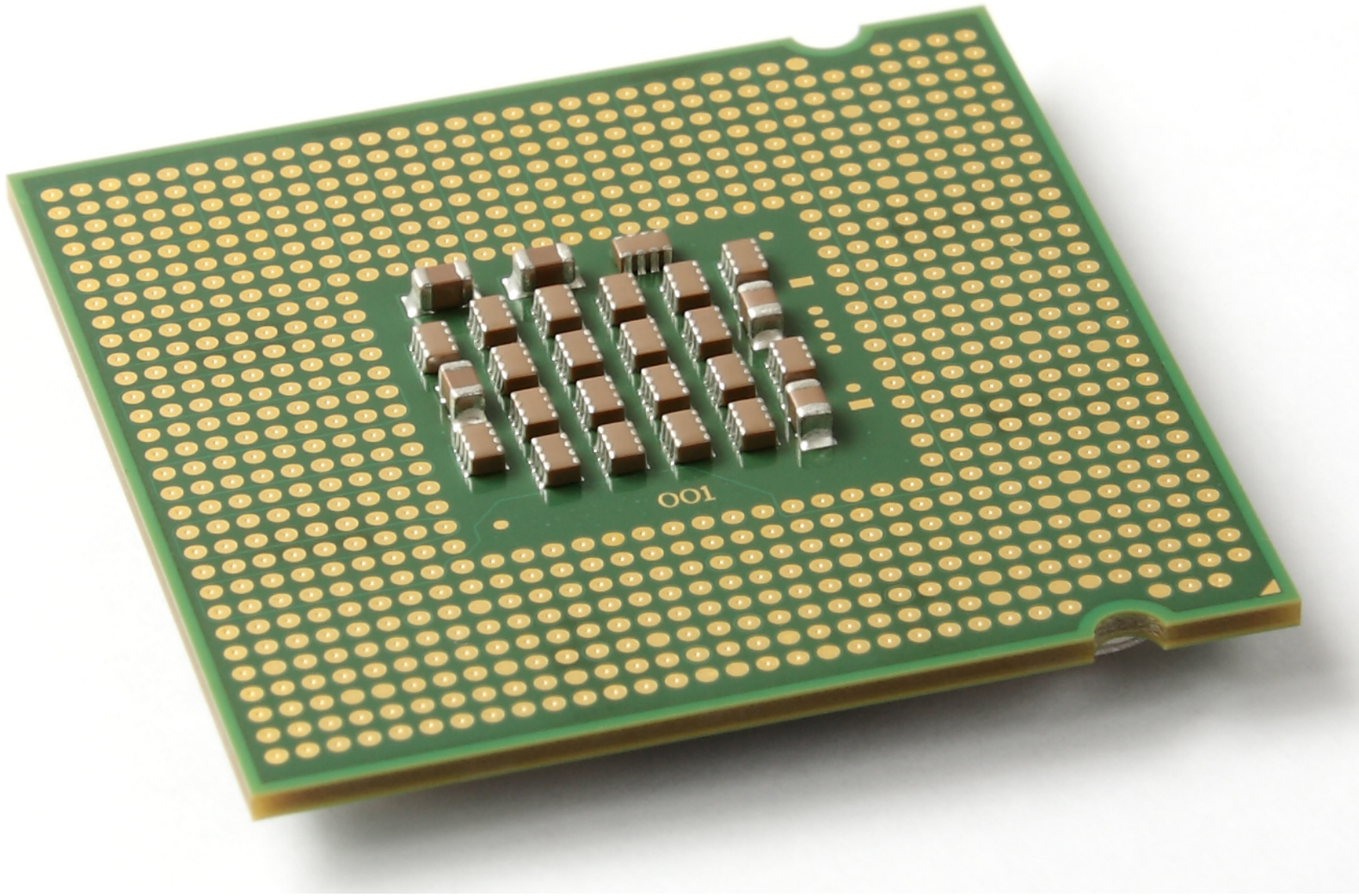
- The characterisation of the adversary can be done through the identification of the type, number and frequency of the deviations to the specified behaviour of the attributes



- Omissive: fail-stop, crash-stop, crash-recovery, crash-link, receive, send and general omissions
- Assertive: Syntactic/Semantic, Byzantine

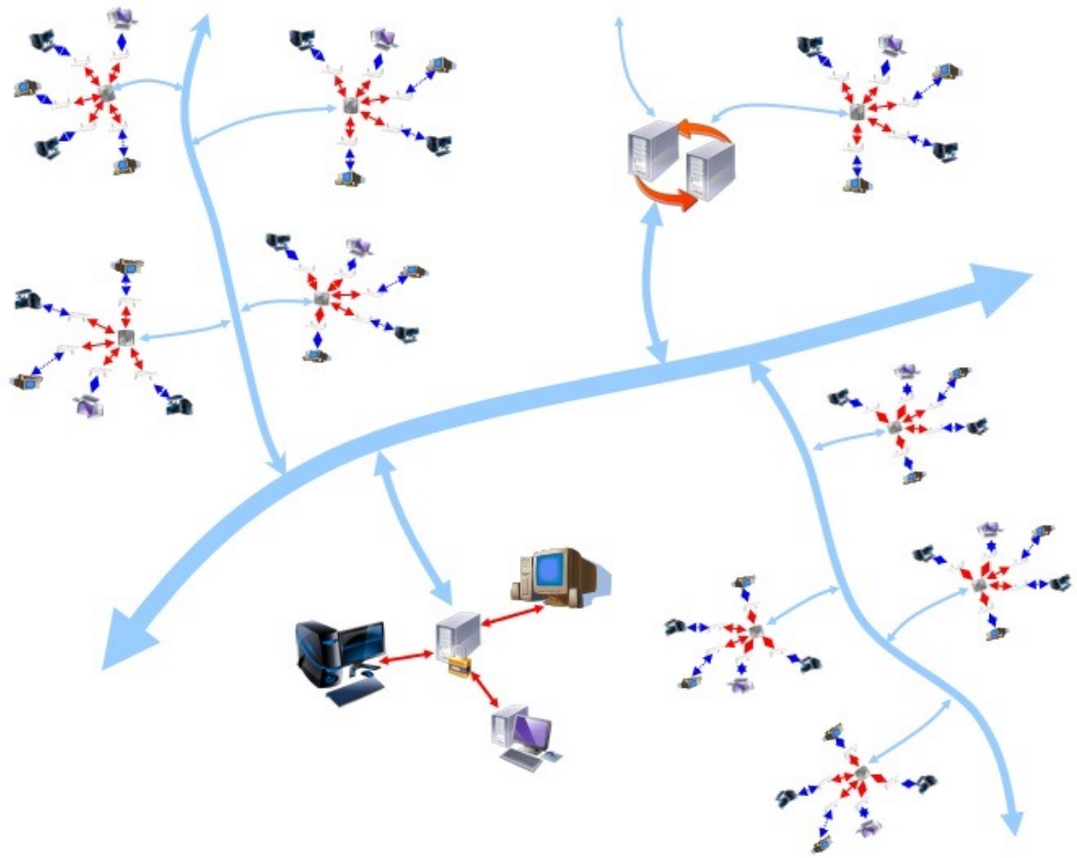
Models of distributed systems - Processing

- Set of processes
  - Unknown
  - Known size
  - Known ids
- Process faults
  - Crash
  - Crash and recovery
  - Misbehave arbitrarily



Models of distributed systems - Communication

- Topology:
  - Fully connected (clique)
- Connected graph:
  - Static or Dynamic
  - Partitions
    - Transient or Permanent
- Reliability:
  - From all messages lost to no messages lost
  - Can messages be created?



Models of distributed systems - Time

- Synchronous:
  - Known upper bounds on message delays, processing delays, and clock differences
- Partially synchronous:
  - Unknown fixed bounds
  - Known eventual bounds
- Asynchronous:
  - No assumptions on bounds and clocks

Models - What kind of problems can be solved?

Static known participants

Synchronous
Reliable static

Synchronous
Reliable dynamic

Synchronous
Reliable clique

Synchronous
Unreliable clique

Synchronous
Bounded unreliable
Clique

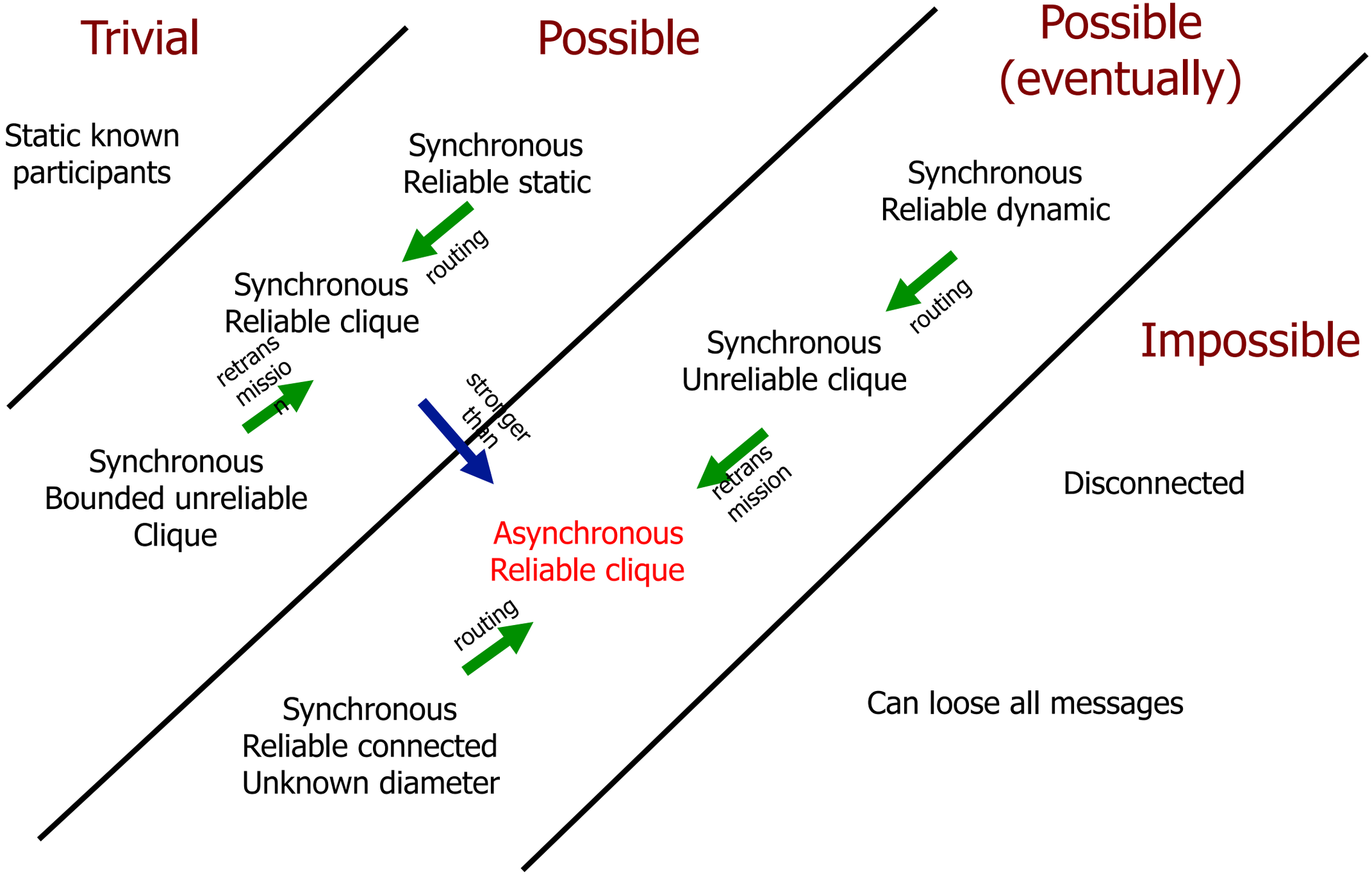
Asynchronous
Reliable clique

Disconnected

Synchronous
Reliable connected
Unknown diameter

Can loose all messages

Feasibility: Leader Election



Models of distributed systems and related faults

Reading material

- Fred. B. Schneider

  “What good are models and what models are good?”

  DISTRIBUTED SYSTEMS (2ND ED), SAPE MULLENDER (ED.), ADDISON-WESLEY 1993

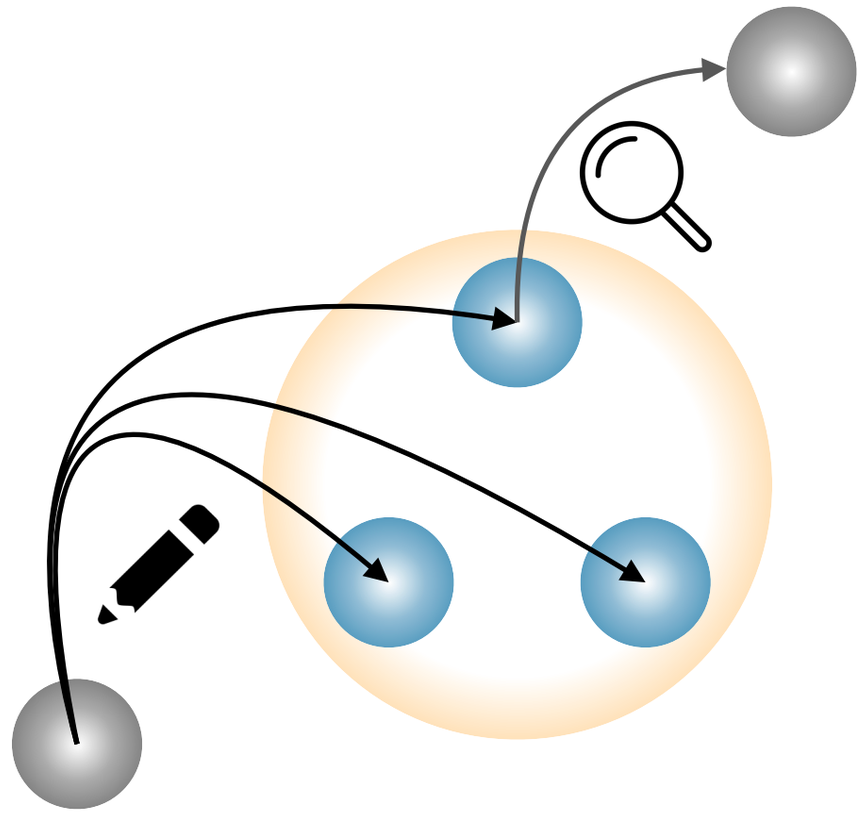
Syllabus

- Introduction to fault-tolerant distributed systems
- Models of distributed systems and related faults
- **Data replication**
- Distributed consensus
- State machine replication
- Database replication

Data replication

ROWA

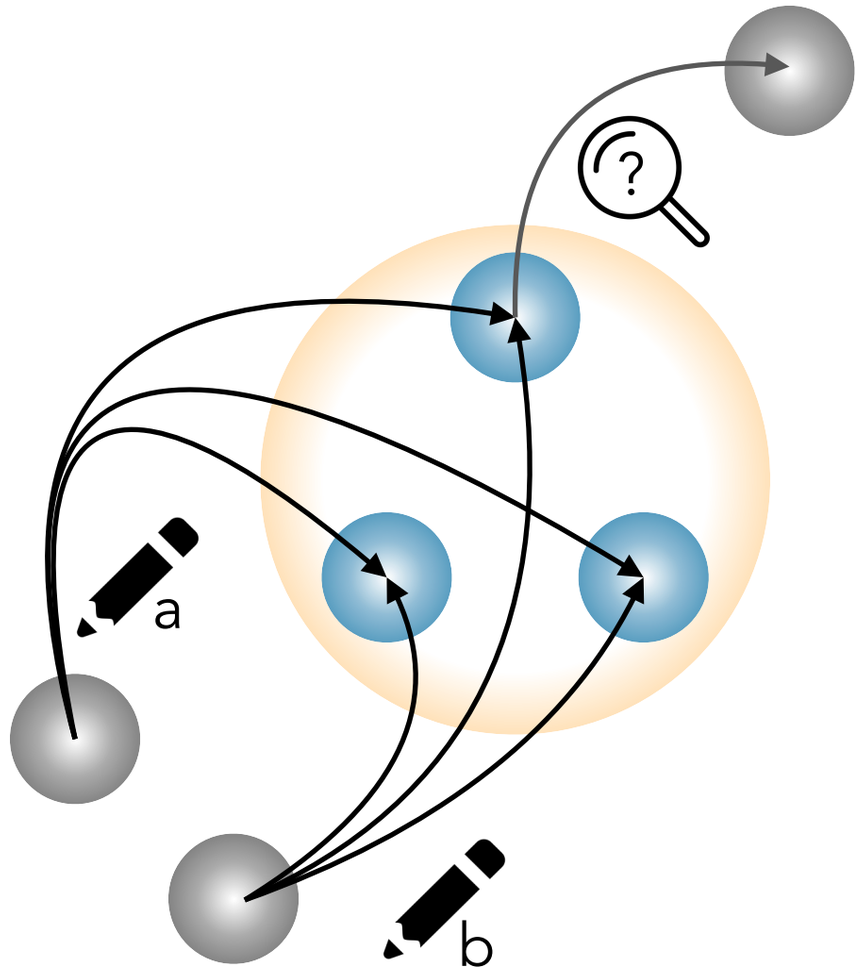
- Let us consider the replication of a basic storage service
  - Elementary atomic read and write operations (mutable servers)
  - Multiple sequential client and server processes l replicas (concurrency model)
  - No faults (fault model)
- Read-One-Write-All (ROWA)
  - Write *value* to all replicas
  - Read can be performed on any replica



Data replication

ROWA

- Let us consider the replication of a basic storage service
  - Elementary atomic read and write operations (mutable servers)
  - Multiple sequential client and server processes / replicas (concurrency model)
  - No faults (fault model)
- Read-One-Write-All (ROWA)
  - Write *value* to all replicas
  - Read can be performed on any replica
- Concurrent writes
  - What does a subsequent read return?
  - We may end up with replicas writing a and then b and others writing b and then a



Data replication

Data versioning

- How can we prevent stale writes?
- Let us add a *version* to the state of the replicas
  - v_i is replica i's version, initially null
- Write operations now become more complex

At client proxies

```
write (value)
    _read (_, vmax) from some replica
    v_write (value, vmax+1) to all replicas
```

At replica i

```
_write (value, version)
    if version > vi then
        xi = value
        vi = version
```

- How to define version? Is a simple scalar enough?
- Eg. version = (counter, pid)

```
vi > vj ::
    vi.counter > vj.counter OR
    vi.counter = vj.counter AND vi.pid > vj.pid
```

Data replication

Quorums

- ROWA is not fault tolerant
- Let us introduce Quorums
  - A quorum is a set of replicas
  - We will refer to two quorums: a write quorum Qw and a read quorum Qr
  - We will write to some Qw set of replicas and will read from some Qr set
  - ROWA is a particular case in which `|Qw| = n` and `|Qr| = 1`
- If we make `|Qw| < n` then the replicated system becomes fault tolerant
- Quorum replication requires that:

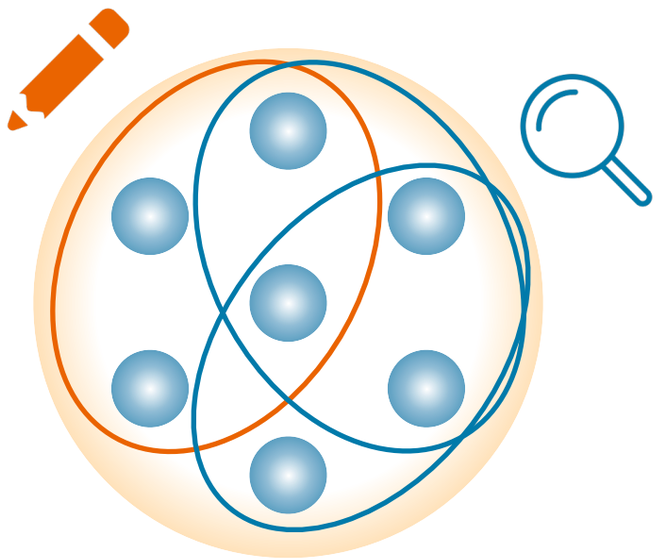
  `|Qr| + |Qw| > n`, read and write quorums always intersect

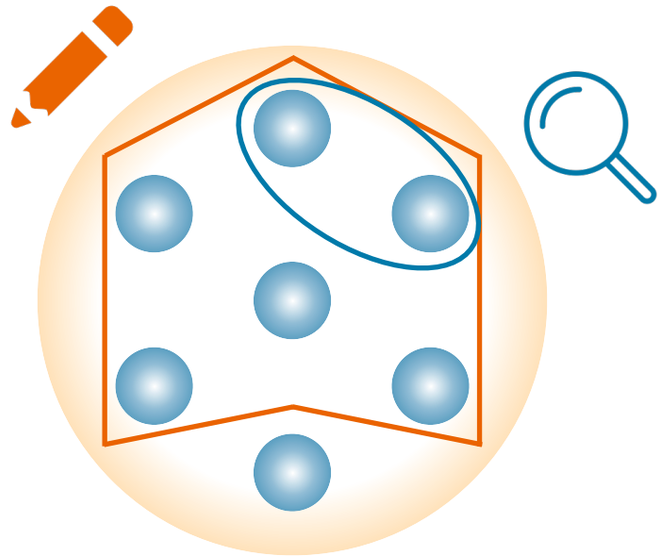
  `2 * |Qw| > n`, any two write quorums always intersect

Data replication

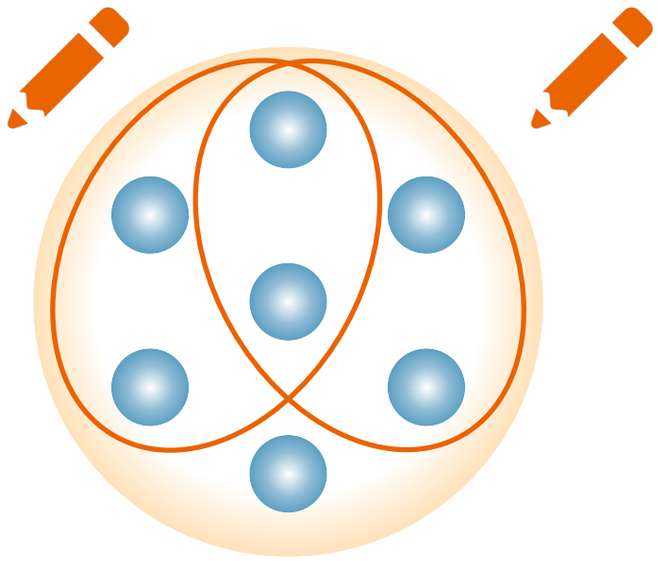
Quorums

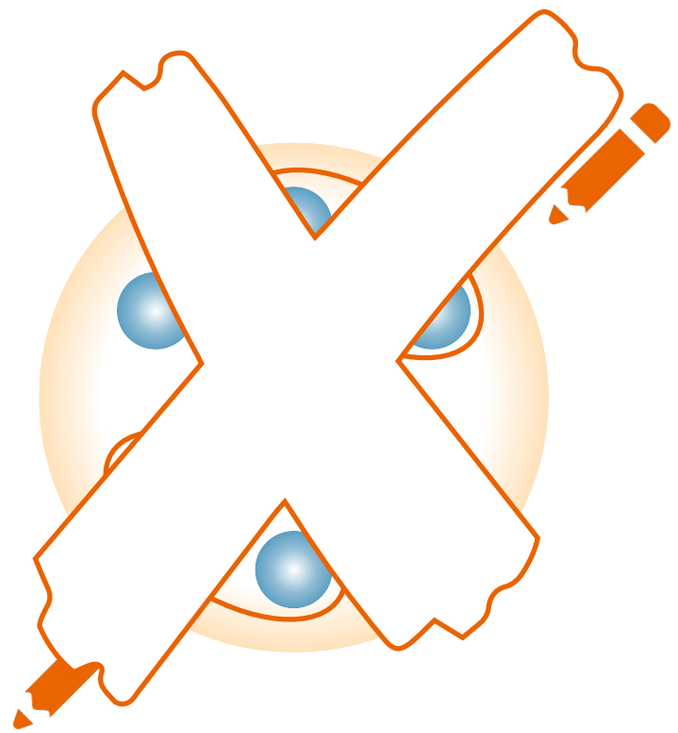
- Read and Write operations take `Qr` and `Qw` into account now





`|Qr| + |Qw| > n`





`2 * |Qw| > n`

Data replication

Quorums

- Read and Write operations take `Qr` and `Qw` into account now
- Write operations now need to read from a `Qr`

At client proxies

```
read (value)
    SetV = _read (x, v) from a Qr
    vmax = largest (_,v) from SetV
    (value, vmax) from SetV
```

```
write (value)
    SetV = _read (_, v) from a Qr
    vmax = largest (_, v) from SetV
    _write (value, vmax+1) to a Qw
```

At replica i

```
_read(value, version)
    value = xi
    version = vi
```

```
_write (value, version)
    if version > vi then
        xi = value
        vi = version
```

Data replication

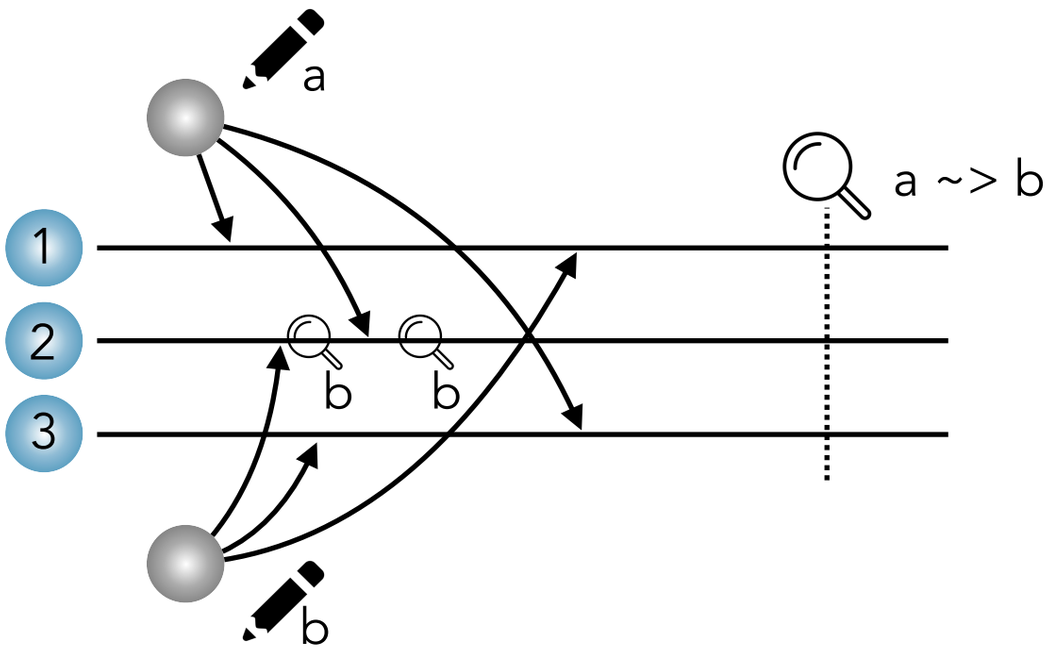
Quorums

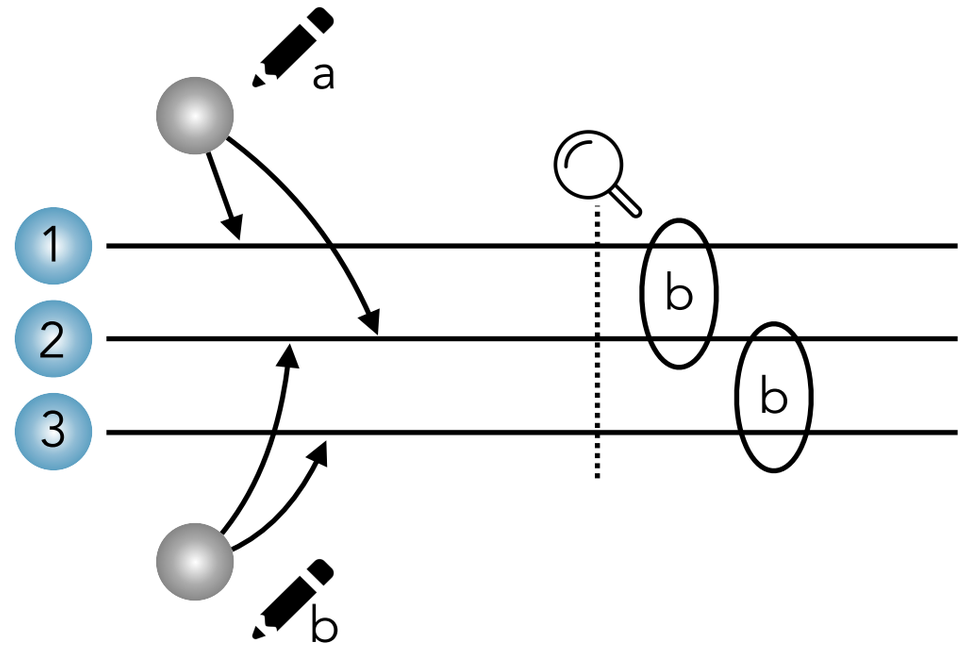
- The use of quorums allows for trade-offs in several system aspects
- With omission faults fault tolerance can be maximised using strict majority quorums:
  - $|Qr| = |Qw| = \lceil (n+1)/2 \rceil$
  - $|Qw| = \lceil (n+1)/2 \rceil$ also leads to the least expensive write operations
- $|Qr|$ can determine workload bias; it determines the cost of reads and impacts the cost of writes
- For how it may impact throughput, latency, and network load see this chapter's reading material

Data replication

Order of writes

- Monotonic increasing versions + PID's



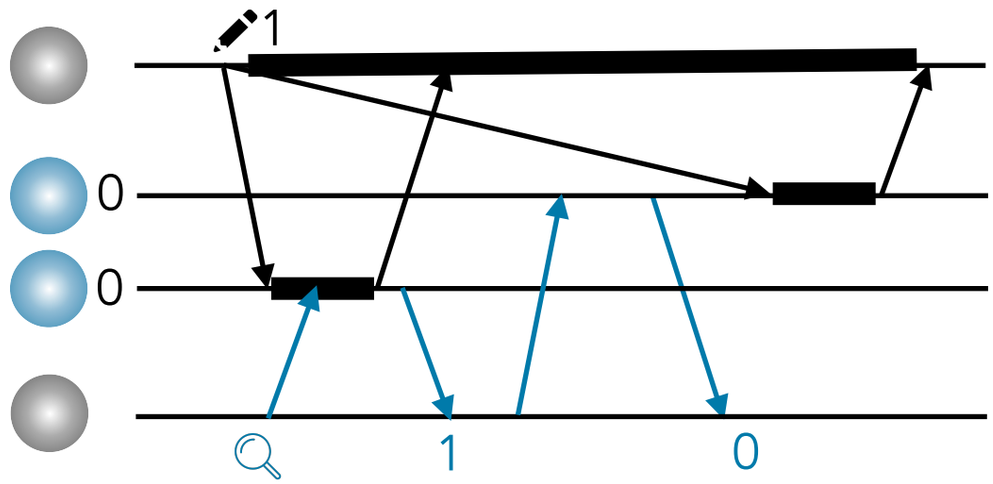


- Globally synchronised clocks
- Causal ordering + PID's

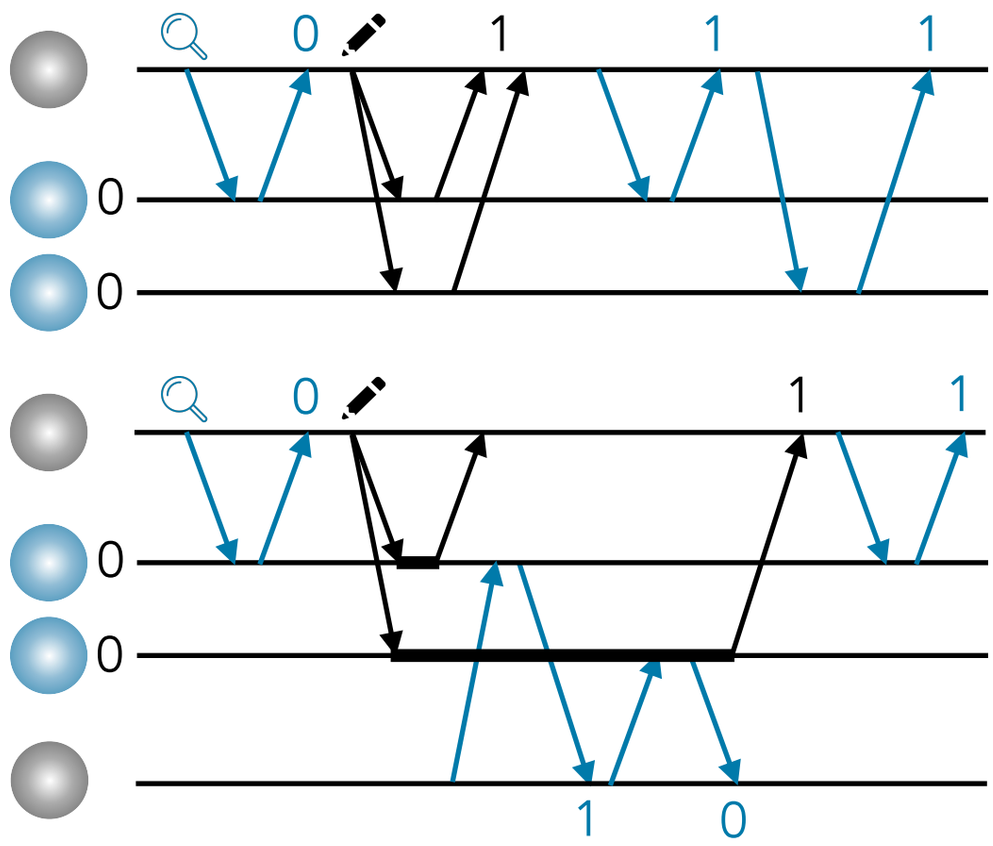
Data replication

Concurrency control

- However, the lack of concurrency control of write operations may lead to basic semantics "inconsistencies"
  - Reading **within** a concurrent write may lead to **unexplainable** results



- Consider an application that manages a non-decreasing variable
  - One process (sequential) execution is OK
  - A concurrent execution may easily violate the application semantics



Data replication

Locked quorums

- Read and Write operations take Qr and Qw into account now

At client proxies

```
read (value)
    SetL = _getlock() from a Qr
    if |SetL| == |Qr|
        SetV = _read (x, v) from a Qr
        vmax = largest (_,v) from SetV
        (value, vmax) from SetV
        res = OK
    else
        res = error
    _freelock() from SetL
    return res

write (value)
    SetL = _getlock() from a Qw
    if |SetL| == |Qw|
        SetV = _read (_, v) from a Qr
        vmax = largest (_, v) from SetV
        _write (value, vmax+1) to a Qw
        res = OK
    else
        res = error
    _freelock() from SetL
    return res
```

At replica i

```
_getlock()
    if lockedi == False
        lockedi = True
        return i
    else
        return error

_freelock()
    lockedi = False

_read(value, version)
    value = xi
    version = vi

_write (value, version)
    if version > vi then
        xi = value
        vi = version
```

Models of distributed systems and related faults

Reading material

- D. Gifford

  “Weighted voting for replicated data”

  SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES, 1979

- M. Whittaker, A. Charapko, J. Hellerstein, H. Howard, I. Stoica

  “Read-Write Quorum Systems Made Practical”

  PAPOC’21, 2021

Syllabus

- Introduction to fault-tolerant distributed systems
- Models of distributed systems and related faults
- Data replication
- **Distributed consensus**
- State machine replication
- Database replication

Distributed Consensus

- Towards a replicated state-machine
- Fault-tolerant Consensus
- (Impossibility of asynchronous fault-tolerant deterministic Consensus)
- The Paxos algorithm
- The Raft algorithm
- (The HotStuff algorithm)

Distributed Consensus

Towards a replicated state-machine

- In last chapter's approach to replicate read and write operations
  - Quorums allow for fault tolerance and performance trade-offs
  - Versioning leads to large overheads, on storage and on write operations
  - Locking per operation is impractical and prone do deadlock
  - Failure of the *locker* might not be simple to deal with
- Provided important insights on fault-tolerant replication
  - No-partitioning: all writes span `Qw` replicas
  - Total order: No two operations appear out-of-order
  - Isolation: No two writes interleave

Distributed Consensus

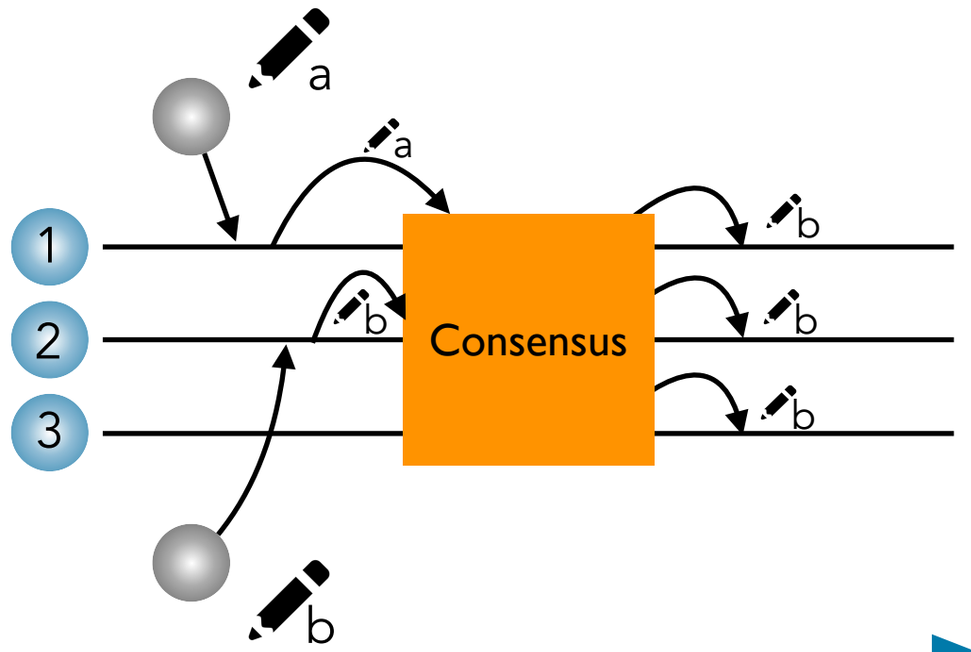
Towards a replicated state-machine - Atomic broadcast

- A well-known and easy to understand conceptual approach to replication is the use of the *atomic broadcast communication protocol*:
- Clients and replicas communicate through *abcast* and *abdeliver* primitives and these ensure that:
  - Integrity: if a process *abdelivers m*, it does so at most once and only if *m* has been previously *abcast*
  - Termination: if a non-faulty process *abcasts m*, then it eventually *abdelivers m*
  - Agreement: If a no-faulty process *abdelivers m*, then all non-faulty processes eventually *abdeliver m*
  - **Total order**: If two processes both *abdeliver* messages *m* and *m'*, then they do so in the same order

Distributed Consensus

The Consensus problem

- Let us see how replicas can *agree* on which operation to execute first
- The Consensus problem: Consider a known finite set of replicas, each can *propose* to execute an operation. Replicas are expected to *choose* an operation satisfying the following properties:
  - Non-triviality: The chosen operation has been proposed
  - Agreement: No two replicas choose differently
  - Termination: All non-faulty replicas eventually choose
- Let us assume that each replica is a proxy to the clients and runs the Consensus algorithm



Distributed Consensus

The Consensus problem - Nonsense variations and their trivial solutions

- Without Non-triviality

```
Operation Consensus (Operation o)
{
   return NOP ;
}
```

- No Agreement

```
Operation Consensus (Operation o)
{
   return o;
}
```

- No Termination

```
Operation Consensus (Operation o)
{
   while(True) ;
}
```

Distributed Consensus

The Paxos algorithm

- The Paxos algorithm, proposed by Leslie Lamport in 1998, focus on the safety properties of Consensus. It, of course, contemplates Termination but does not get down to necessary liveness conditions to ensure it
- Model: Let us consider an asynchronous system, and a non-byzantine adversary:
  - A minority of replicas may crash (and can, possibly, later recover)
  - A subset of messages sent can be lost (and can also be duplicates)
- In the following, we will learn the Paxos algorithm by reasoning about satisfying the validity and agreement properties.
- The next slides shamelessly copy Lamport's "Paxos Made Simple" paper. Any errors are the scriber's.

Distributed Consensus

The Paxos algorithm

- In the Paxos algorithm, each replica may have three simultaneous roles:
  - proposer: proposes an operation to other replicas
  - acceptor: commits to accept a proposed operation
  - learner: learns that a proposal has been chosen
- The gist of the algorithm is as follows:
  - A proposer *may send* a proposed operation to a set of acceptors
  - An acceptor *may accept* the proposed operation
  - The operation is **chosen when a majority of acceptors have accepted it**
  - A learner learns the chosen operation once it knows a majority have accepted it

Distributed Consensus

The Paxos algorithm

- **P1: An acceptor must accept the first proposal that it receives**
  - Assume not: the proposal of a single operation would not be accepted and therefore never chosen
  - If several operations are proposed we can run into a deadlock (for we need a majority of acceptors), unless acceptors can accept more than one proposal
  - However, an acceptor cannot simply accept any two proposals as it could lead to the choice of different operations. We need to guarantee that all *chosen* proposals have the same operation.
    - Let proposals be numbered: (number, operation), and numbered differently
- **P2: If a proposal with operation o is chosen, then every higher-numbered proposal that is chosen has operation o**

Distributed Consensus

The Paxos algorithm

- **P2: If a proposal with operation o is chosen, then every higher-numbered proposal that is chosen has operation o**
- **P2a: If a proposal with operation o is chosen, then every higher-numbered proposal that is accepted has operation o**
  - Because communication is asynchronous, a proposal could be chosen with some particular replica r never having received any proposal. Suppose a new replica "wakes up" and issues a higher-numbered proposal with a different operation, by P1 r would be required to accept this proposal, violating P2a.
  - So, let us strengthen P2a:
- **P2b: If a proposal with operation o is chosen, then every higher-numbered proposal that is proposed has value o**
  - P2b (proposed) $\Longrightarrow$P2a (accepted) $\Longrightarrow$ P2 (chosen)

Distributed Consensus

The Paxos algorithm

- **P2b: If a proposal with operation o is chosen, then every higher-numbered proposal that is <u>proposed</u> has value o**
- How an algorithm can satisfy P2b?
- Let us consider how we would prove that P2b holds
  - Assume (m, o) is chosen. We need to show that for any (n > m, x), x = o
  - (1) For induction, let us assume that for any ([m, n -1], x), x = o
  - (2) Then, for (m, o) to be chosen there must be some set C consisting of a majority of acceptors such that every acceptor in C accepted (m, o).
- From (1) and (2) …

Distributed Consensus

The Paxos algorithm

- From (1) and (2):
  - Every acceptor in C has accepted a proposal with number in [m , n -1], and every proposal with number in [m, n -1] accepted by any acceptor has operation o
  - Since any set S consisting of a majority of acceptors contains at least one member of C, we can conclude that a proposal numbered n has operation o by ensuring that the following invariant is maintained:

- **P2c: For any n and o, if a proposal with number n and operation o is issued, then there is a set S consisting of a majority of acceptors such that either:**
  - **(a) no acceptor in S has accepted any proposal numbered less than n, or**
  - **(b) o is the operation of the highest-numbered proposal among all proposals numbered less than n accepted by the acceptors in S**

Distributed Consensus

The Paxos algorithm

- By P2c, a proposer that wants to issue a proposal numbered n must know the highest-numbered proposal with number less than n, if any, that **has been or will be accepted** by each acceptor in some majority of acceptors.
  - Learning about proposals already accepted is easy enough; predicting future acceptances is hard…
  - Instead, the proposer requests that the acceptors not accept any more proposals numbered less than n.

Distributed Consensus

The Paxos algorithm - Satisfying safety properties

- **Phase 1**
  - (a) A proposer selects a proposal number n and sends a *prepare request* with number n to a majority of acceptors
  - (b) If an acceptor receives a *prepare request* with number n greater than that of any prepare request to which it has already responded, then it responds to the request with a promise not to accept any more proposals numbered less than n and with the highest-numbered proposal (if any) that it has accepted
- **Phase 2**
  - (a) If the proposer receives a response to its prepare requests (numbered n) from a majority of acceptors, then it sends an accept request to each of those acceptors for a proposal numbered n with an operation o, where o is the operation of the highest-numbered proposal among the responses, or is any operation if the responses reported no proposals
  - (b) If an acceptor receives an accept request for a proposal numbered n, it accepts the proposal unless it has already responded to a prepare request having a number greater than n

Distributed Consensus

The Paxos algorithm - Learning chosen operations

- To learn that a value has been chosen, a learner must find out that a proposal has been accepted by a majority of acceptors
- The obvious (but expensive) algorithm is to have each acceptor, whenever it accepts a proposal, respond to all learners, sending them the proposal
- The assumption of non-Byzantine failures makes it easy for one learner to find out from another learner that a value has been accepted. We can have the acceptors respond with their acceptances to a distinguished learner, which in turn informs the other learners when a value has been chosen. Thrifty but slow.
- Because of message loss, a value could be chosen with no learner ever finding out. If a learner needs to know whether a value has been chosen, it can issue a proposal, using the algorithm described above.

Distributed Consensus

The Paxos algorithm - Liveness

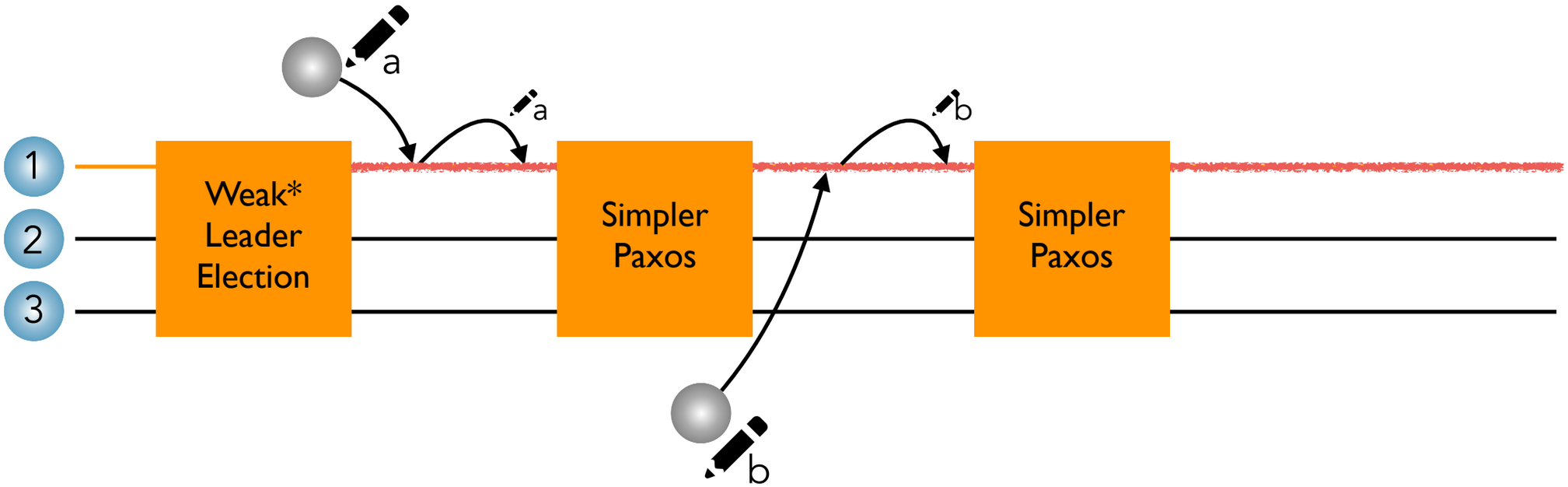
- It's easy to construct a scenario in which two proposers each keep issuing a sequence of proposals with increasing numbers, none of which are ever chosen.

- Proposer *p* completes Phase 1 for a proposal n_1. Proposer *q* then completes Phase 1 for a proposal $n_2 > n_1$. Proposer *p*'s Phase 2 accept requests for a proposal n_1 are ignored because the acceptors have all promised not to accept any new proposal numbered less than n_2. So, proposer *p* then begins and completes Phase 1 for a new proposal number $n_3 > n_2$, causing the second Phase 2 accept requests of proposer *q* to be ignored. And so on.

Distributed Consensus

The Paxos algorithm - Liveness

- To guarantee progress, a distinguished proposer must be selected as the only one to try issuing proposals. If it can communicate successfully with a majority of acceptors, and if it uses a proposal with greater than any already used, then it will succeed with a proposal that is accepted



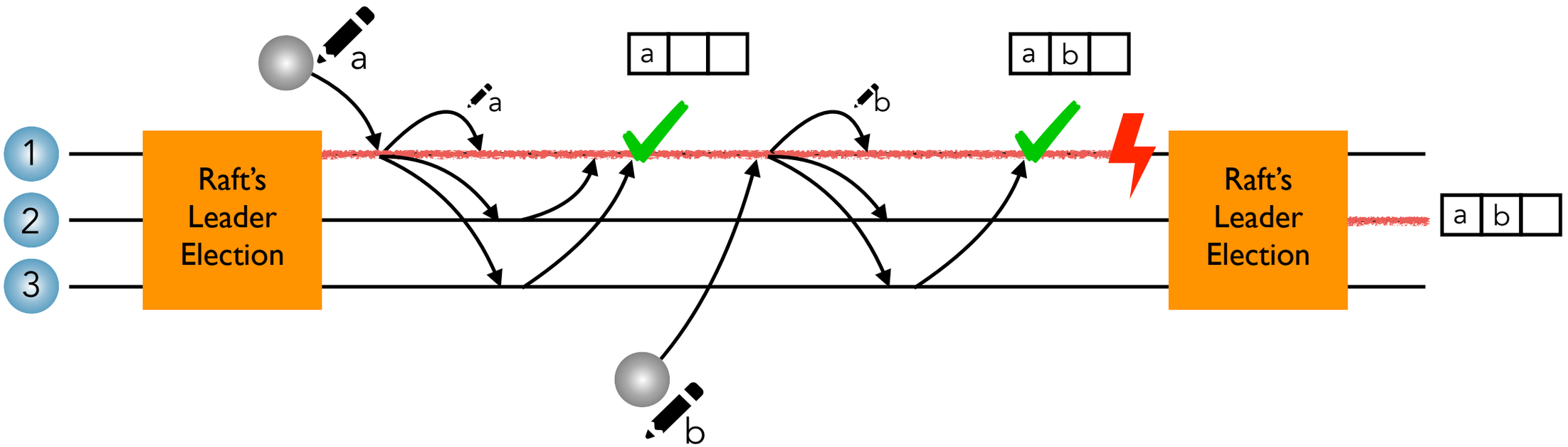
- Ensuring a unique leader is harder than solving Consensus so a weaker form of leader election is actually adopted, and sufficient

- Having a leader allows to simplify the Paxos algorithm

Distributed Consensus

The Raft algorithm - A distinguished proposer to guarantee progress

- To prevent competing proposers to outnumber each other, the Raft algorithm "implements consensus by **first electing a distinguished leader**, then giving the leader complete responsibility for managing a **replicated log** [of operations]."
- The focus of Raft is on the election of a leader. "Once a leader has been elected, it begins servicing client requests."
- Raft's "tailored" leader election allows to streamline replication eschewing a Consensus instance per operation:



Distributed Consensus

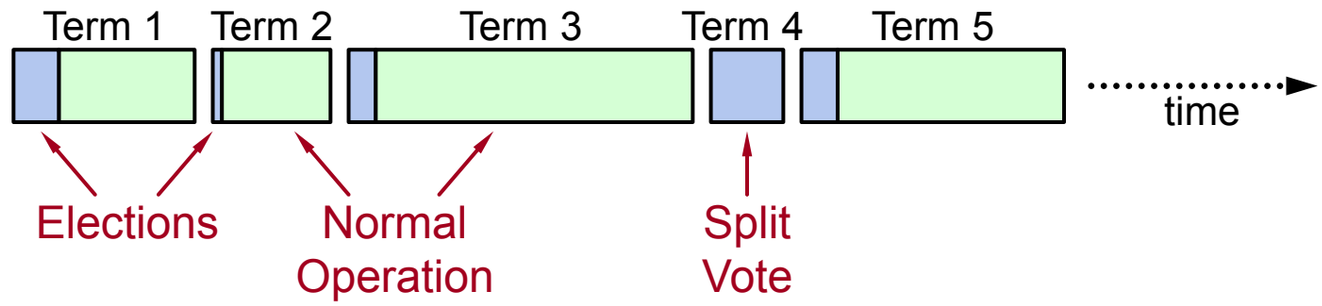
The Raft algorithm - A distinguished proposer to guarantee progress

- As leaders may fail, we need to elect new leaders
- Leadership uniqueness (or lack of) and succession require special attention
- Because we cannot ensure leaders are unique and, due to asynchrony and message loss, processes may differ on the log state, Raft adds election restrictions that resemble Paxos' properties
- Raft uses the voting process to prevent a candidate from winning an election unless it is up-to-date. A candidate must contact a majority of the cluster in order to be elected, which means that at least one of those servers has the most up-to-date state.

Distributed Consensus

The Raft algorithm - Leader election

- At any given time, any replica is in one of three states: *leader, follower or candidate*
- In normal operation, there is exactly one *leader* and all other replicas are *followers* that simply respond to the *leader* (and *candidates*) requests
- Normal operation ceases when the *leader* is deemed to be replaced. The system goes through monotonically increasing "leader terms": leader election, its term, leader election, its term,…



- Due to asynchrony and message loss, different replicas may observe the transitions between terms at different times, and in some situations a replica may not observe an election or even entire terms

Distributed Consensus

The Raft algorithm - Leader election

- If there's a *leader*, it sends periodic heartbeats to all replicas and if they receive them in a timely manner then happily stay as *followers*
- If a *follower* does not receive any leader's heartbeats then it starts an election to choose a new one: it increments **its** term and transitions to *candidate* state and requests votes from all replicas in the system
- A *candidate* stays as such until:
  - (a) it wins the election: transitions to *leader* and starts sending heartbeats
  - (b) it receives heartbeats from a leader in >= term: transitions to *follower*
  - (c) a period of time goes by with no winner: starts a new election
- A candidate wins an election if it receives votes from a **majority of the replicas for the same term**. Each replica will **vote for at most one candidate in a given term**, on a first-come-first-served basis

Distributed Consensus

The Raft algorithm - Log replication

- A major contribution of Raft is that it handles the **replication of a sequence of operations**, a replicated totally ordered log.
- A leader services client operation requests. It appends the operation to its log as a new entry, then sends the entry and the current term to all other replicas.
- Once the entry is acknowledged by a majority of the replicas, it becomes **committed**. The leader executes the operation and replies to the client
- Once a follower learns that a log entry is committed, it considers all preceding entries also committed and **applies** them to its local state machine (in log order)

Distributed Consensus

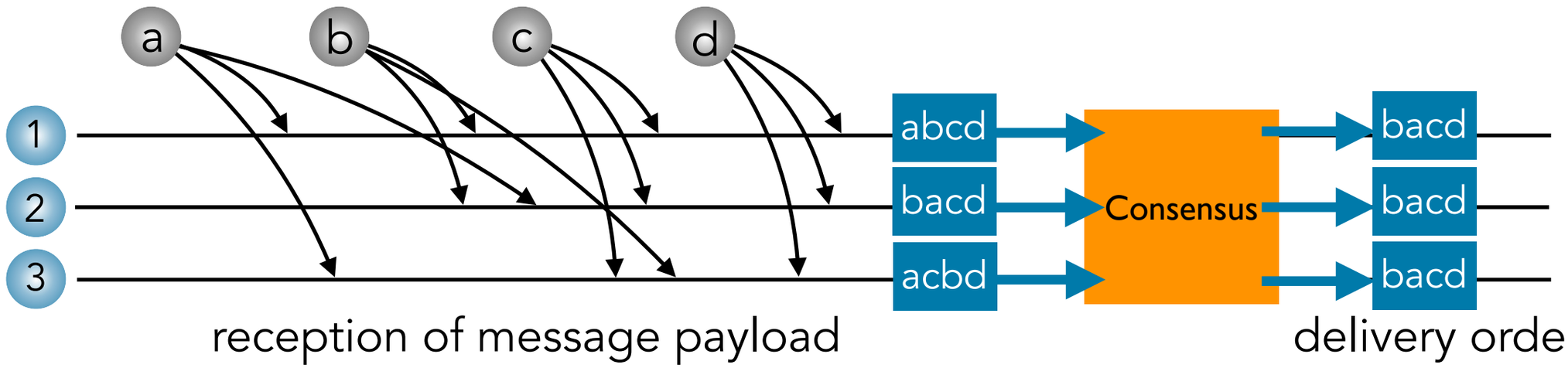
The Raft algorithm - Log replication

- Rafts ensures — Log Matching Property — that:
  - If two entries in different logs have the same index and term, then they store the same operation
  - If two entries in different logs have the same index and term, then the logs are identical in all preceding entries
- Raft uses a simple approach where it guarantees that all the committed entries from previous terms are present on each new leader from the moment of its election, without the need to transfer those entries to the leader.
- The above subsumes a synchronisation instant as a process cannot become leader before having all the committed entries from previous terms in its log.

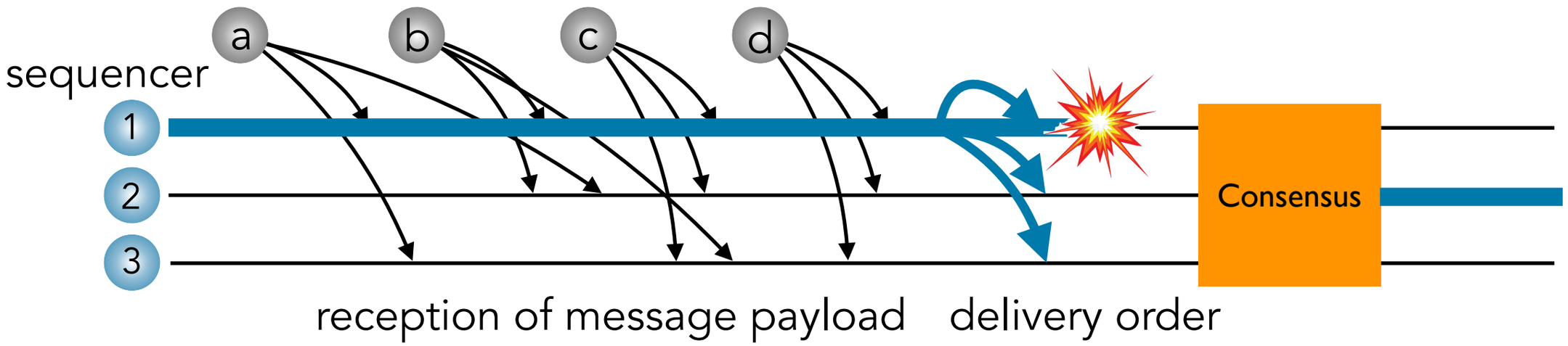
Distributed Consensus

Consensus to Atomic broadcast

- An instance of Consensus can be used to decide on a sequence (set and order) of messages to be delivered:



- Or, by unifying the role of Consensus leader (eg. Raft) and sequencer:



Models of distributed systems and related faults

Reading material

- L. Lamport

  "Paxos Made Simple"

  ACM SIGACT NEWS 32, 4 (DECEMBER 2001)

- D. Ongaro and J. Ousterhout

  "In Search of an Understandable Consensus Algorithm"

  2014 USENIX ANNUAL TECHNICAL CONFERENCE (JUNE 2014)

- L. Lamport

  "The Part-Time Parliament" (entertaining material)

  ACM TRANSACTIONS ON COMPUTER SYSTEMS 16, 2 (MAY 1998)

Syllabus

- Introduction to fault-tolerant distributed systems
- Models of distributed systems and related faults
- Data replication
- Distributed consensus
- **State machine replication**
- Database replication

State machine replication

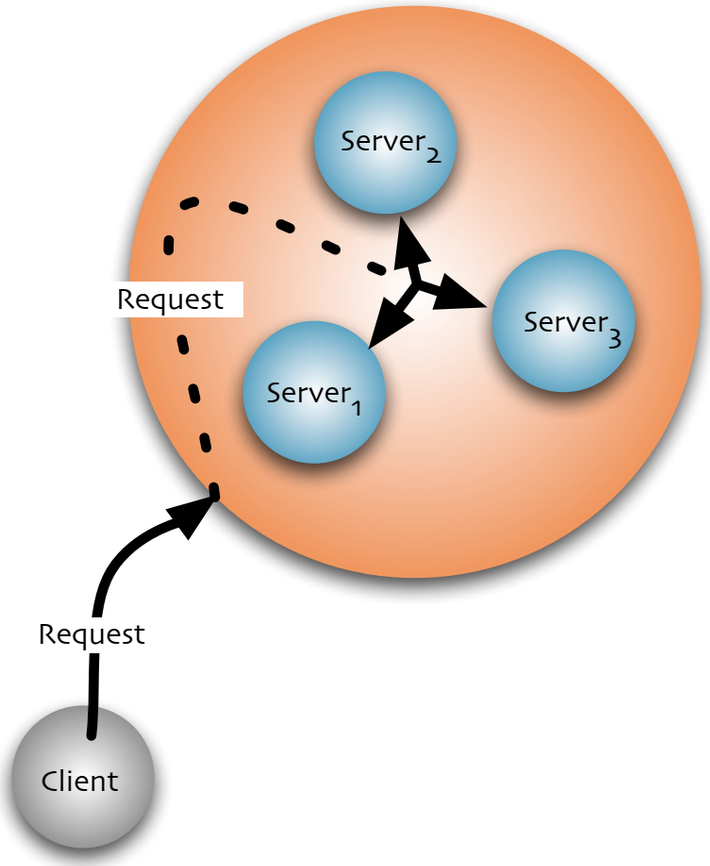
- Active replication
- Passive replication
- A functional model of replication
- Replica integration
- (Replica consistency)

State machine replication

- We now consider the **replication of any generic function:** a function differs from the previous read and write operations as they might read and write any finite set of state variables and return some value as the result of executing some arbitrary code
- Model
  - Asynchronous sequential client and server processes
  - Asynchronous reliable communication channels
  - Omissions faults
  - Consensus is solvable
- In this chapter we review the two major function replication techniques of **state machine replication**: active and passive replication

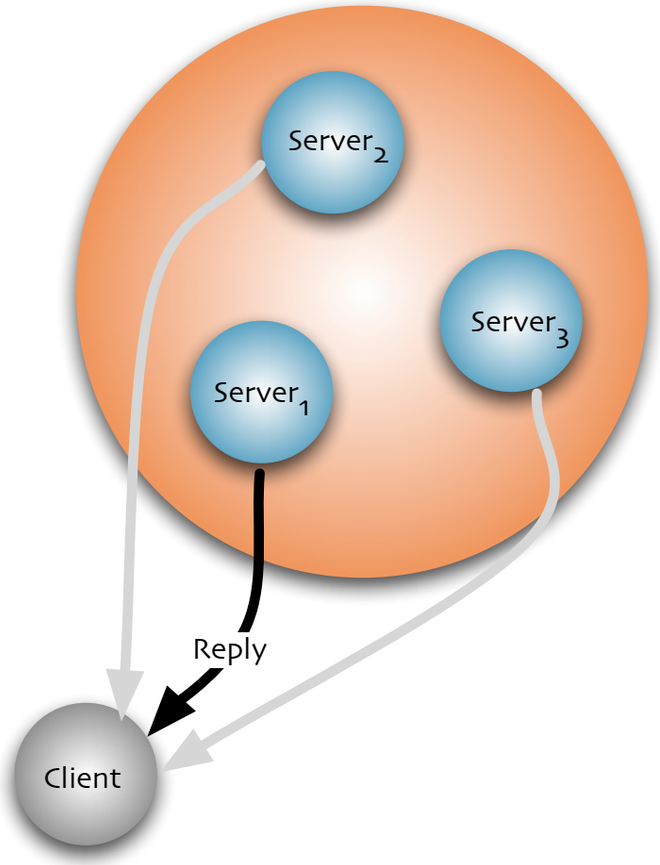
State machine replication

Active replication



1.
Request is sent to all the replicas: client uses **abcast** or uses any replica as proxy.

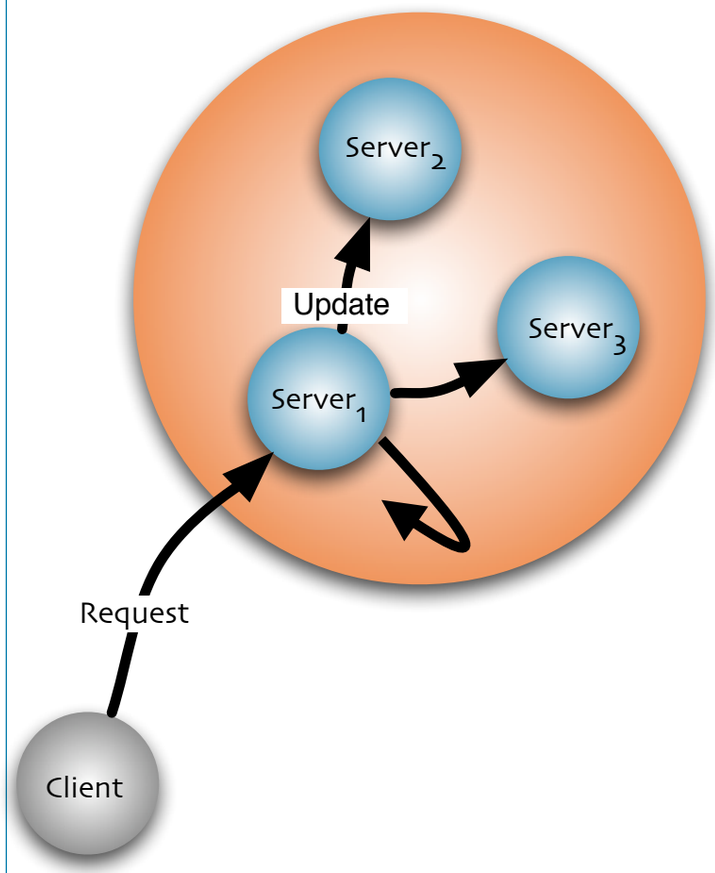
2.
Each replica processes the request, updates its own state, and replies to the client, or its proxy.



3.
Client, or proxy, waits until it receives the first response (or a majority of identical responses the Byzantine fault model)

State machine replication

Passive replication or Primary-Backup replication

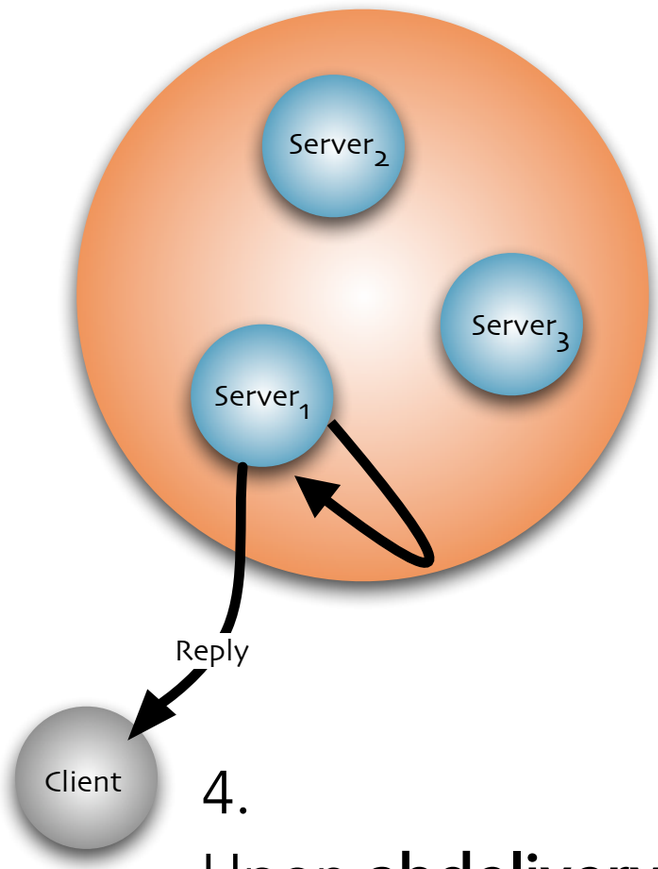


1.

Request is addressed to a distinguished, primary replica: directly or through any replica

2.

The primary executes the request, updates its own state, and **abcasts** a (request id, state update, reply) message to all other replicas



3.

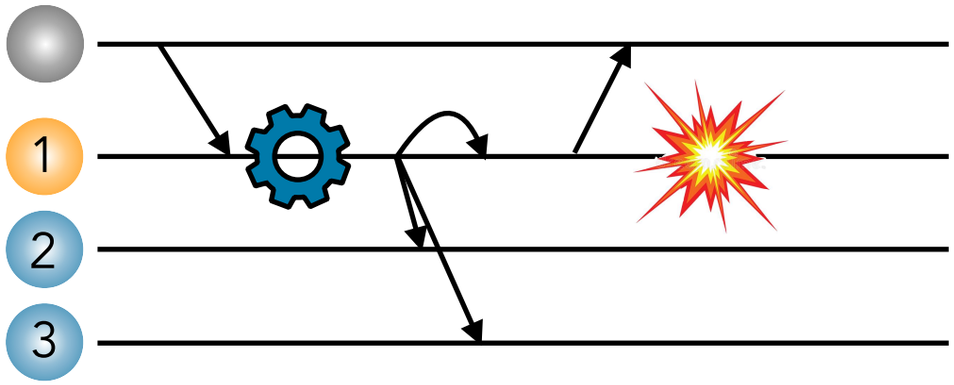
Upon **abdelivery** each replica updates its state

4.

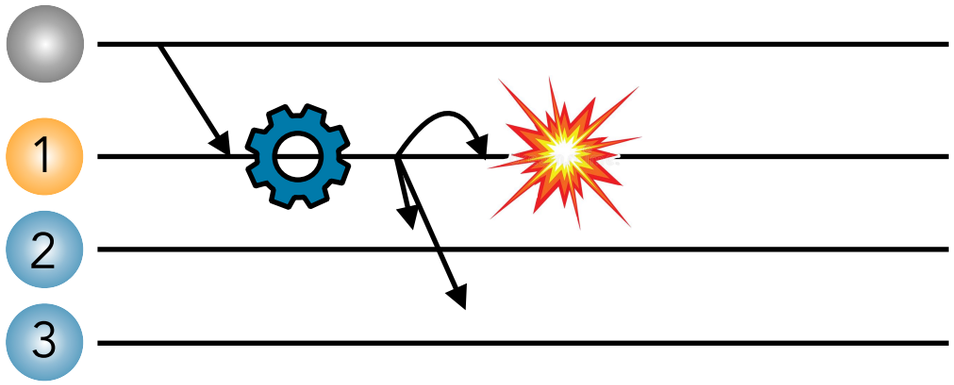
Upon **abdelivery** the primary replies to the client

State machine replication

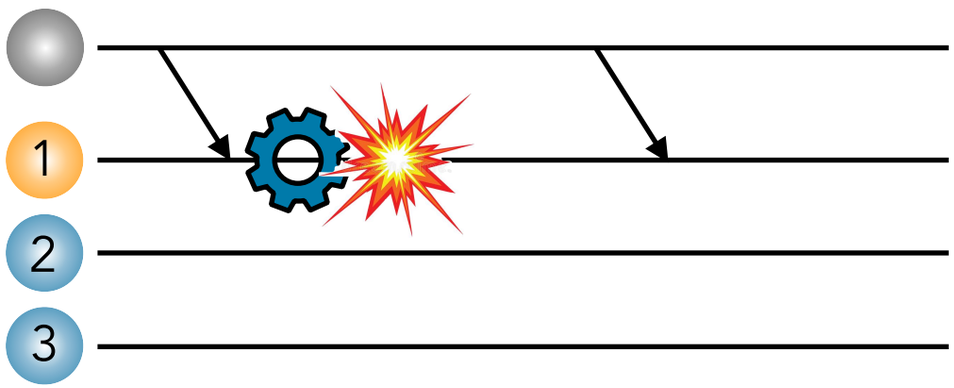
Passive replication or Primary-Backup replication



Transparent for the client



Might not be transparent for the client



Not transparent for the client

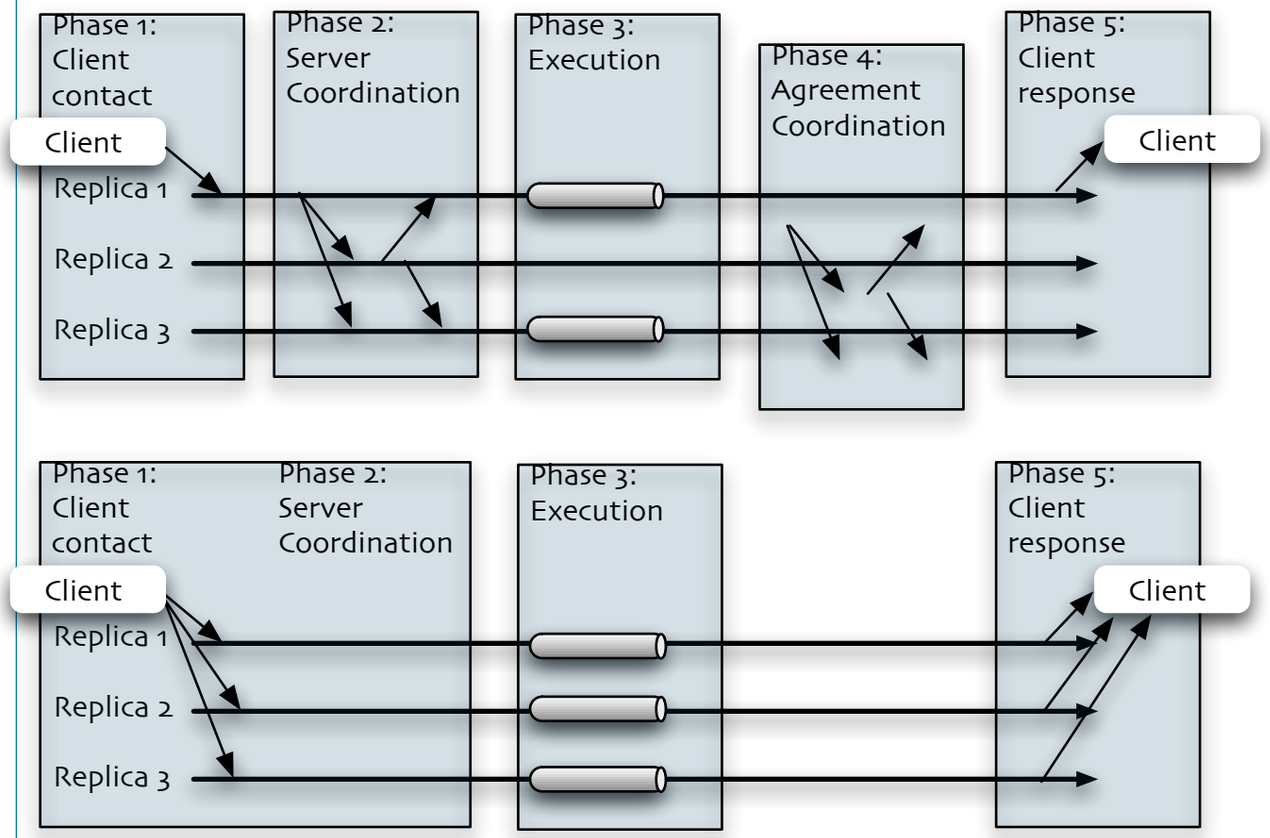
State machine replication

Active vs. Passive replication

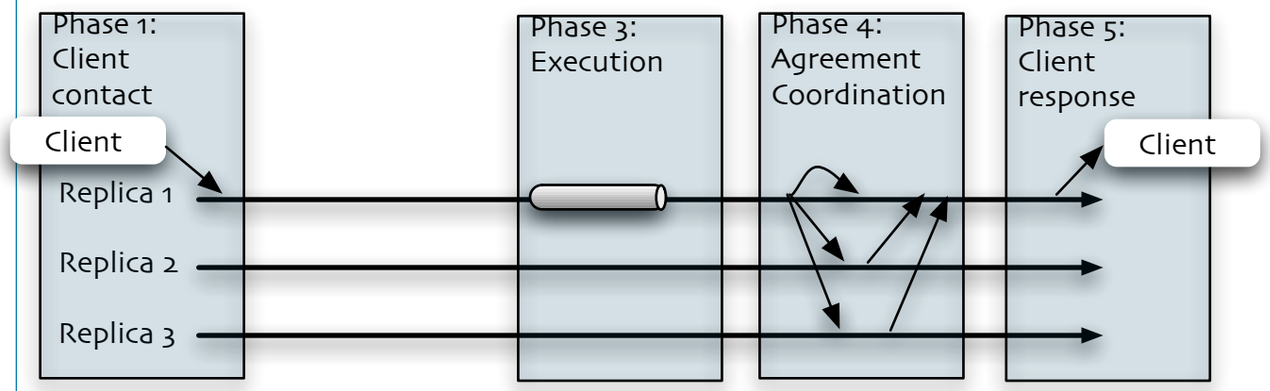
Active	Passive
Only deterministic functions	Any function
Replication is transparent	Awarness of a primary replica
Replica failures are transparent	Primary failure can be perceived

State machine replication

A functional model of replication



Active Replication

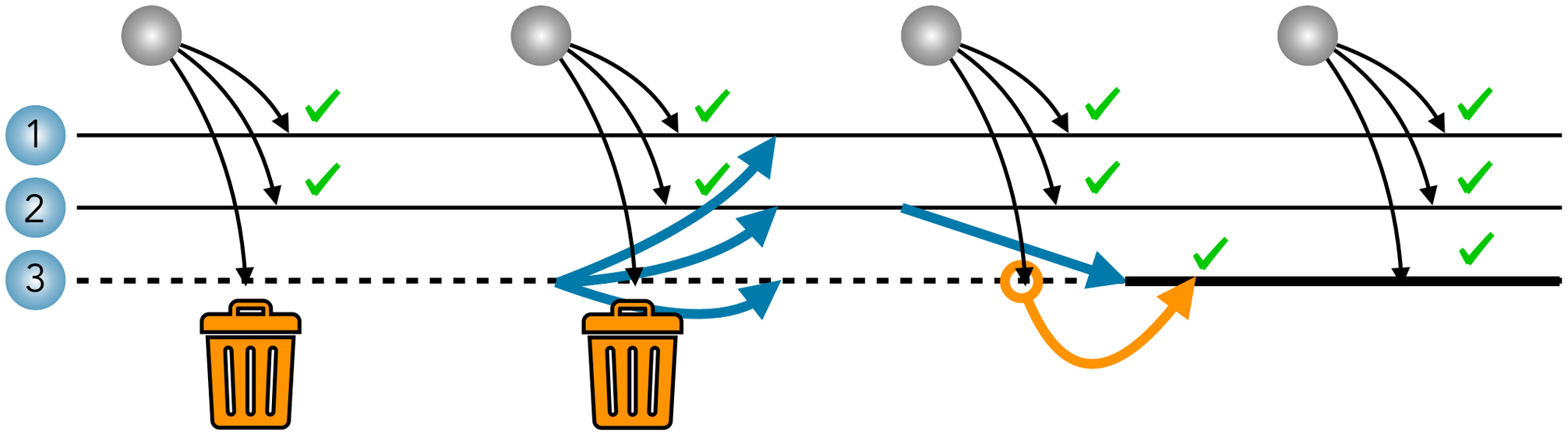


Passive Replication

State machine replication

Replica integration

- To restore or increase the systems resilience we need to add replicas to the system. Before being able to handle any requests the replica's state need to be consistent with the others'.
- Replica integration should be done with minimal disruption to the system's availability and performance. We aim at online integration.



state request state reception

State machine replication

Reading material

- M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, G. Alonso
  "Understanding Replication in Databases and Distributed Systems"
  INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS (2000)

- R. Guerraoui, A. Schiper
  "Software-Based Replication for Fault Tolerance"
  IEEE COMPUTER (1997)

Syllabus

- Introduction to fault-tolerant distributed systems
- Models of distributed systems and related faults
- Data replication
- Distributed consensus
- State machine replication
- **Database replication**

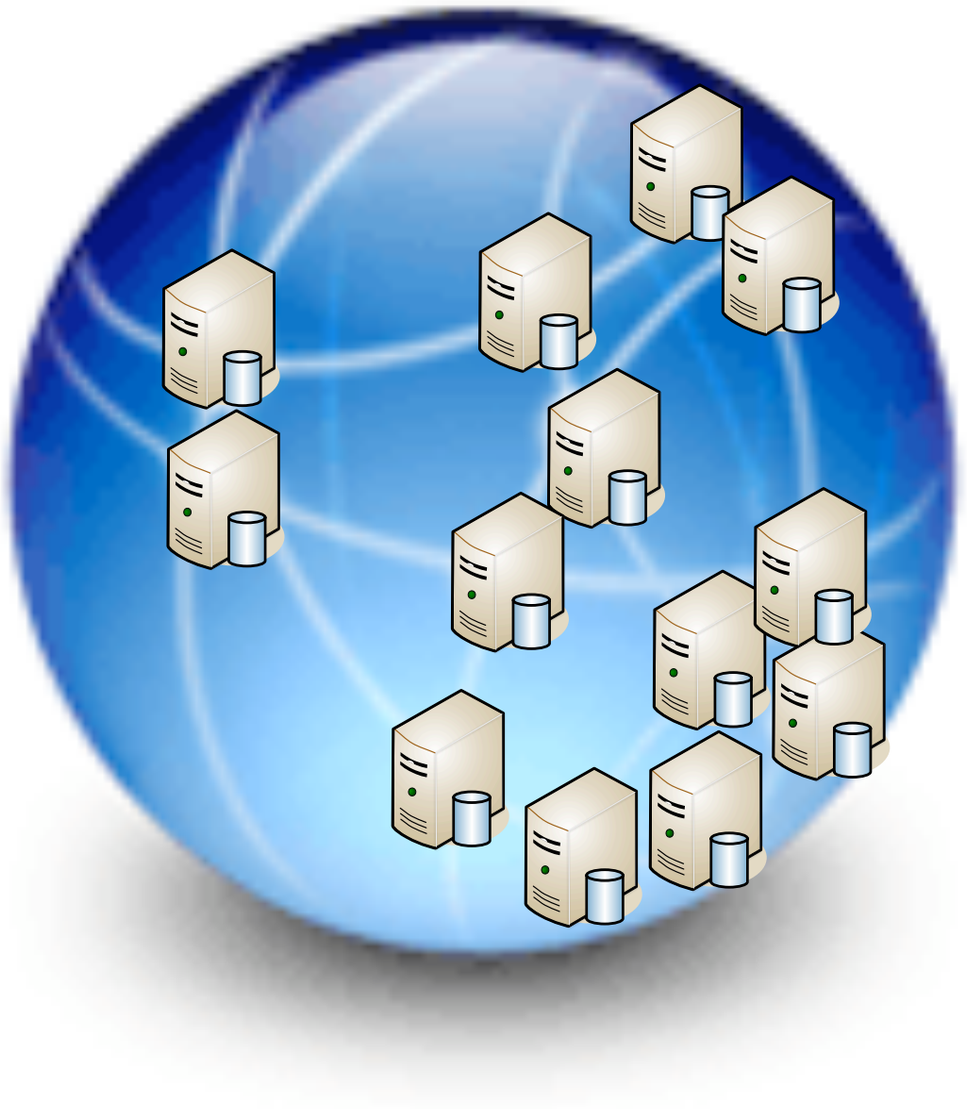
Database replication

- Basic Approaches
- Specific Database Replication Protocols

Database replication

Transactional database management systems

- Transactional requests
- Multi-interaction requests
- Highly concurrent servers



- Replication goals
  - Availability
  - Performance
- Replication challenges
  - Scalability under generic workloads
  - Preserved Consistency Criteria

Database replication

Basic approaches

- The replication of a DBMS is a particular case of state machine replication (not data!)
- DBMS typically process transactions which are special beasts with particular “life styles and social etiquette”
- DBMS tend to be highly concurrent systems and may adopt several concurrency control mechanisms. Concurrency is a source of non determinism!
- The replication of a DBMS tend to be based on the primary/backup technique, ie. passive replication, and is usually distinguished as being either:
  - Synchronous or Asynchronous
  - Single or Multi-master

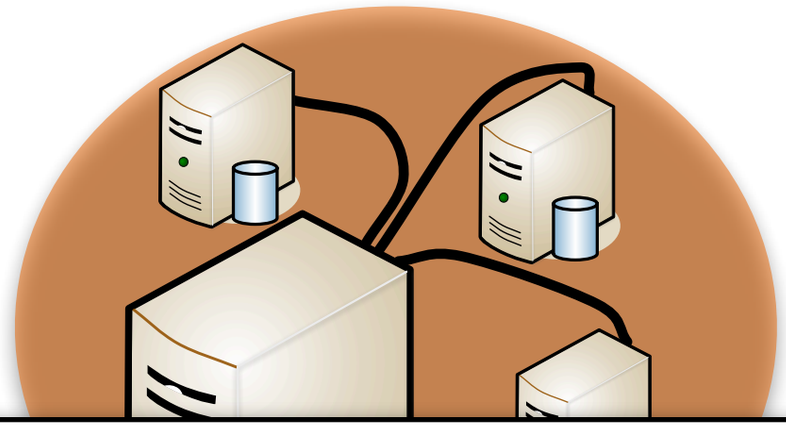
Database replication

Consistent replication requirements

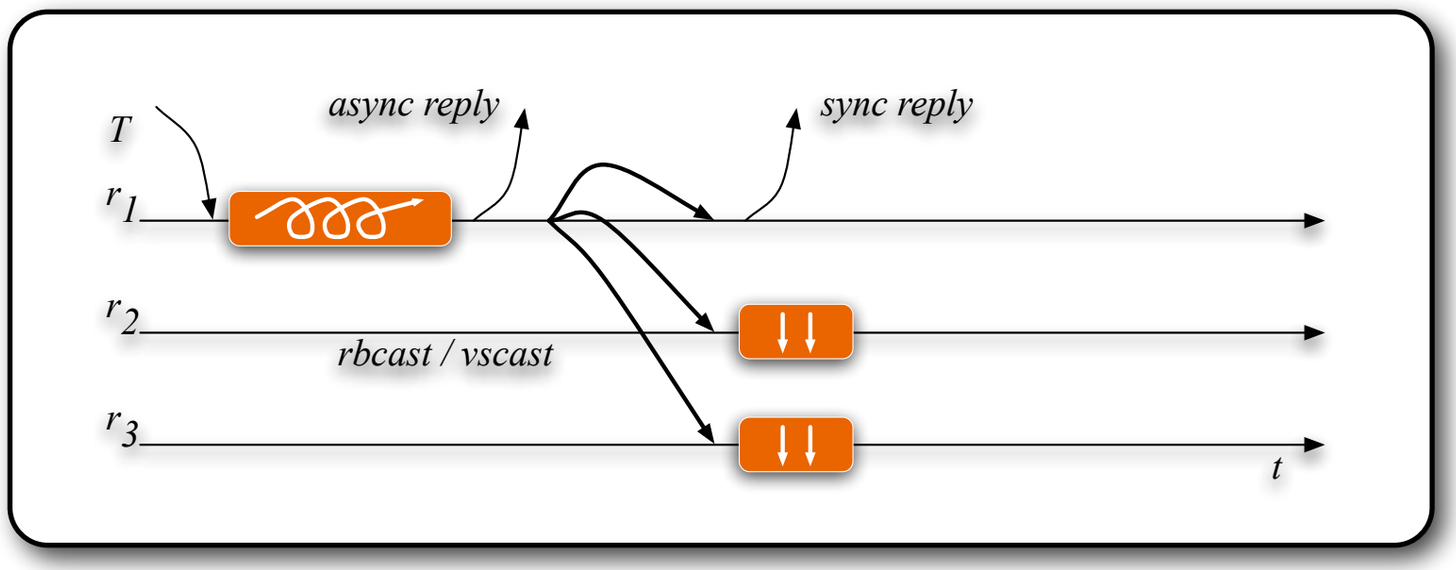
- Strictly consistent database replication to provide transparent distribution and fault tolerance
  - Strictly consistent replicas on transaction boundaries
  - No reconciliation
  - Automatic fail-over without loss of updates
- Focus on group communication based techniques instead of traditional distributed locking and atomic commitment protocols
- Multi-master, update-anywhere protocols

Database replication

Primary-backup

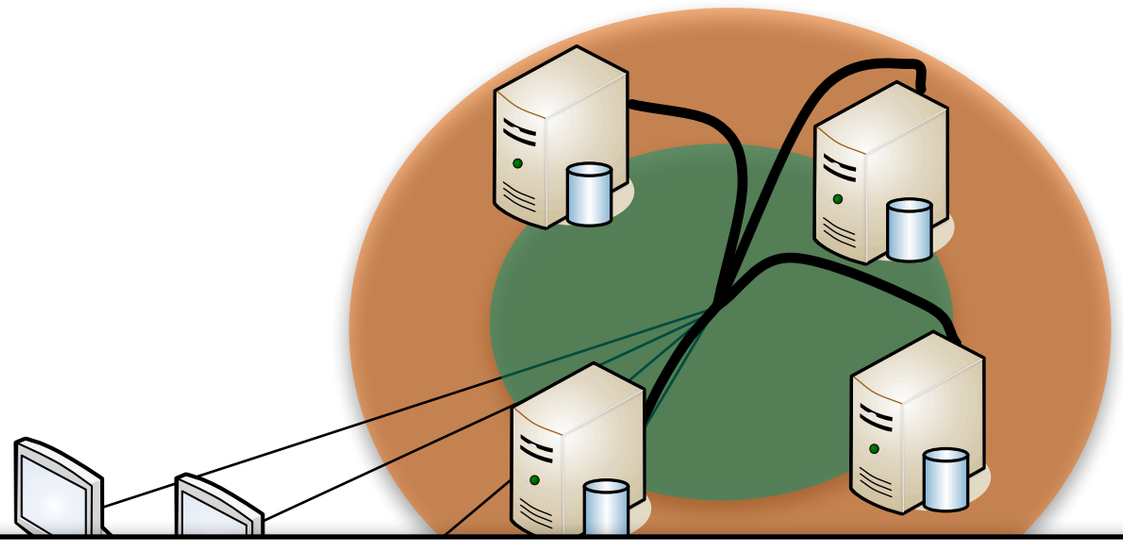


- Asynch PB is standard replication in most DBMS
- Asynch vs. synch update propagation
- Adequate to handle non-deterministic servers
- Extensible to multi-master (partitioning, reconciliation)

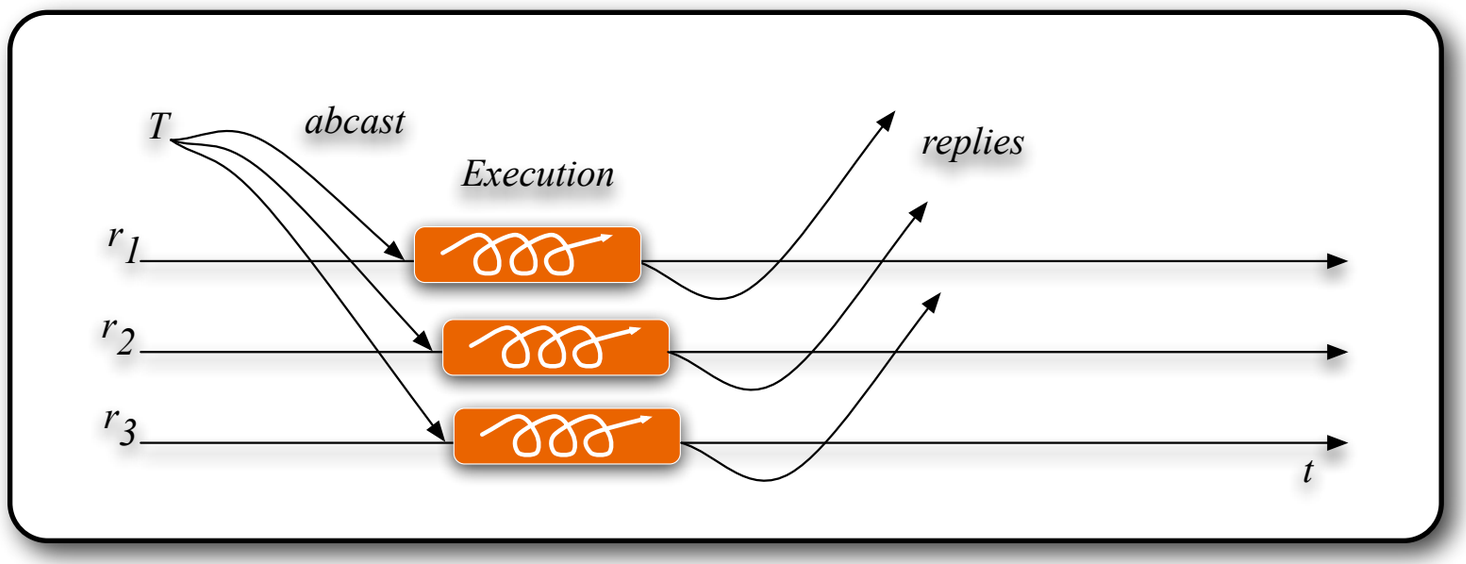


Database replication

Replicated state-machine



- Simplicity and failure transparency
- Requires deterministic processing (SQL, scheduling)



Database replication

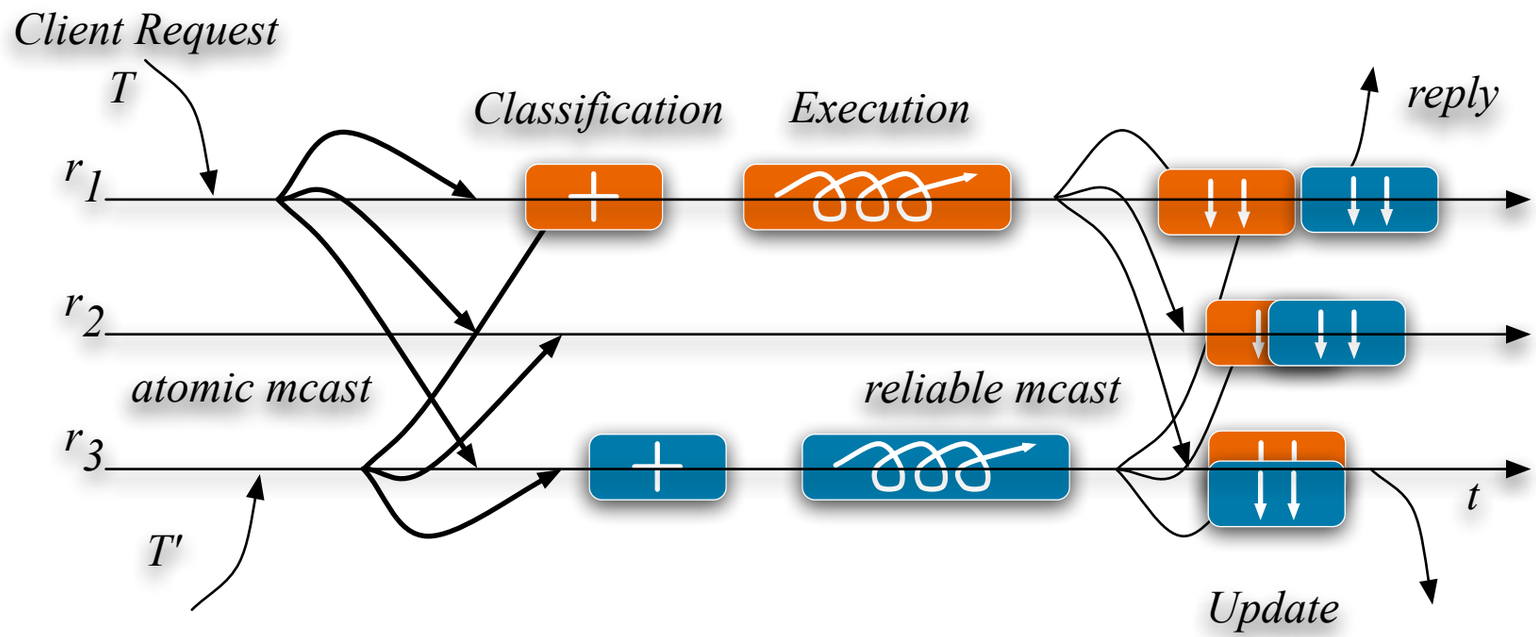
Specific Database Replication Protocols

- Build on the classical Replicated State-Machine
- Adopt the passive replication paradigm
- To globally serialise conflicting concurrent transactions share the use of an atomic multicast primitive
  - Message delivery is Atomic and Totally Ordered
- Ensure 1-copy-equivalence as consistency criterion

Database replication

Conservative execution

- Transactions are classified in conflict classes
- Transactions are ordered before their execution
- Conflicting transactions are sequentially executed

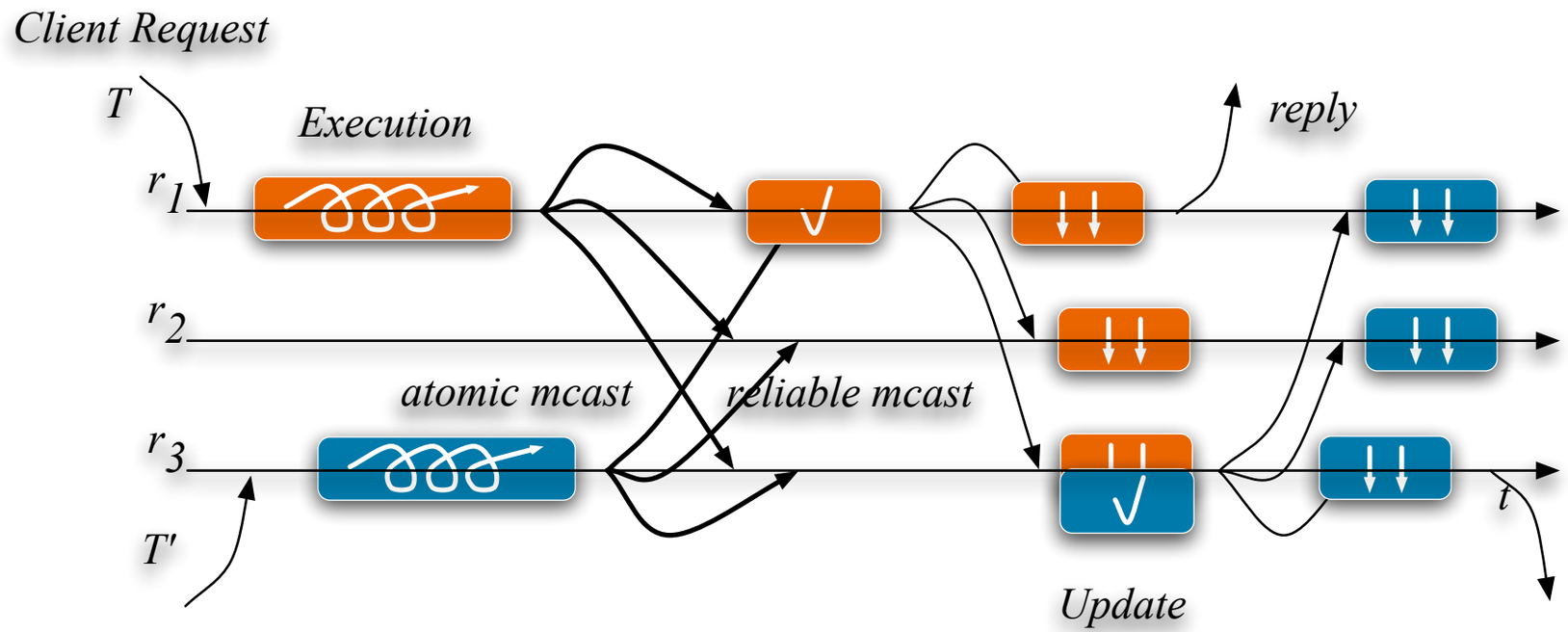


Database replication

Optimistic execution

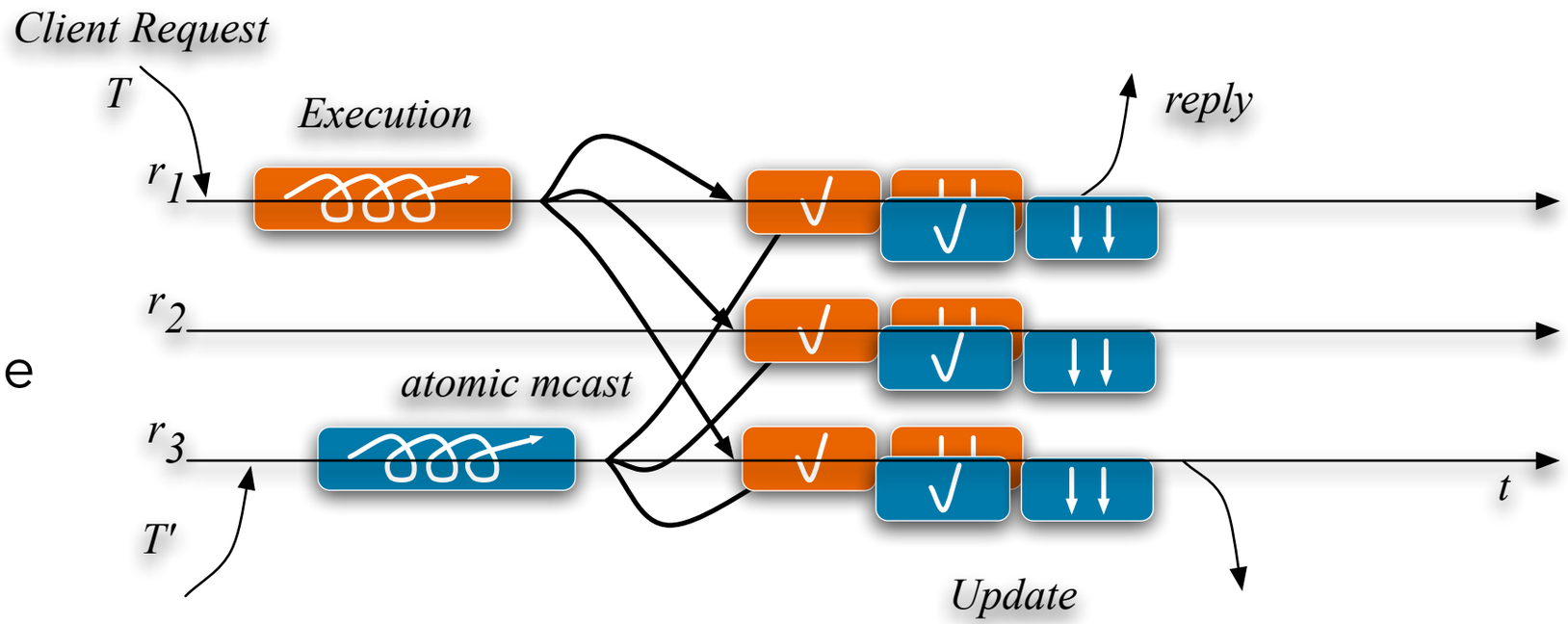
- Transactions are ordered after their execution
- Conflicting transactions may execute concurrently

Postgres-R



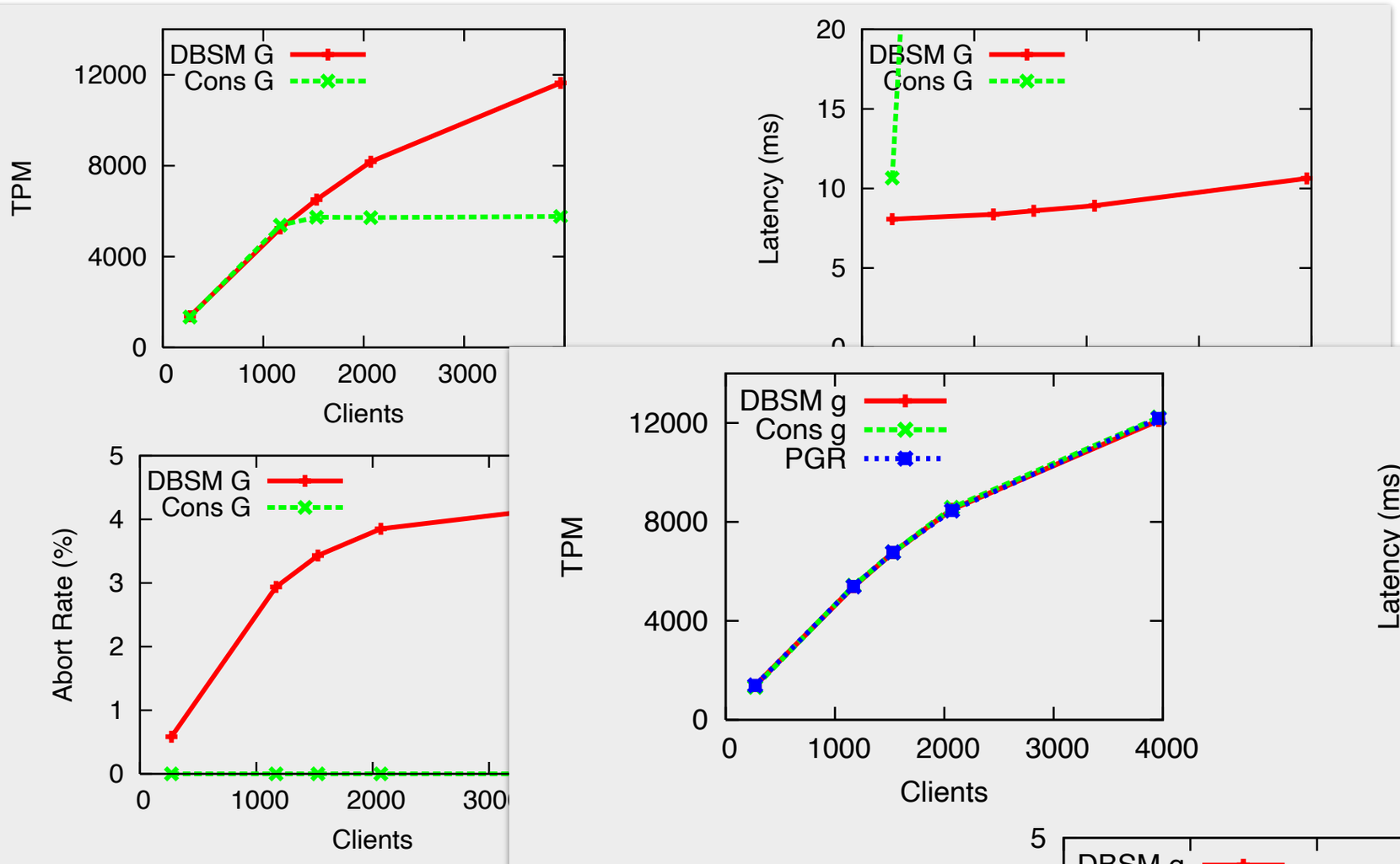
DBSM

DataBase State Machine

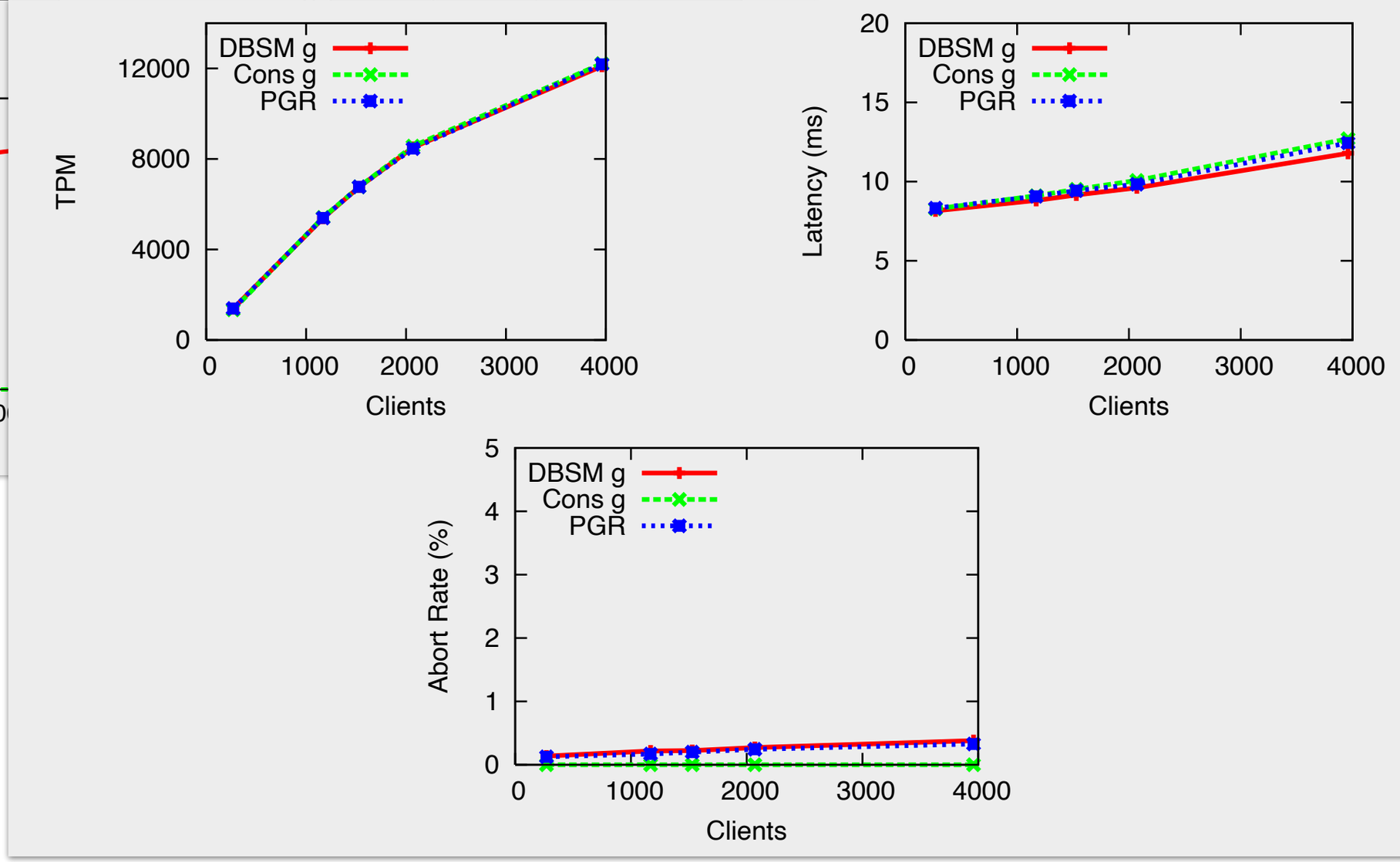


Database replication

Performance comparison



Coarse Grain CC



Fine Grain CC

Database replication

Reading material

- M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, G. Alonso
  "Understanding Replication in Databases and Distributed Systems"
  INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS (2000)

- F. Pedone, R. Guerraoui, A. Schiper
  "The Database State Machine Approach"
  DISTRIBUTED AND PARALLEL DATABASES (2003)

- Y. Lin, B. Kemme, M. Patiño-Martínez, R. Jiménez-Peris
  "Middleware based data replication providing snapshot isolation"
  ACM SIGMOD (2005)