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

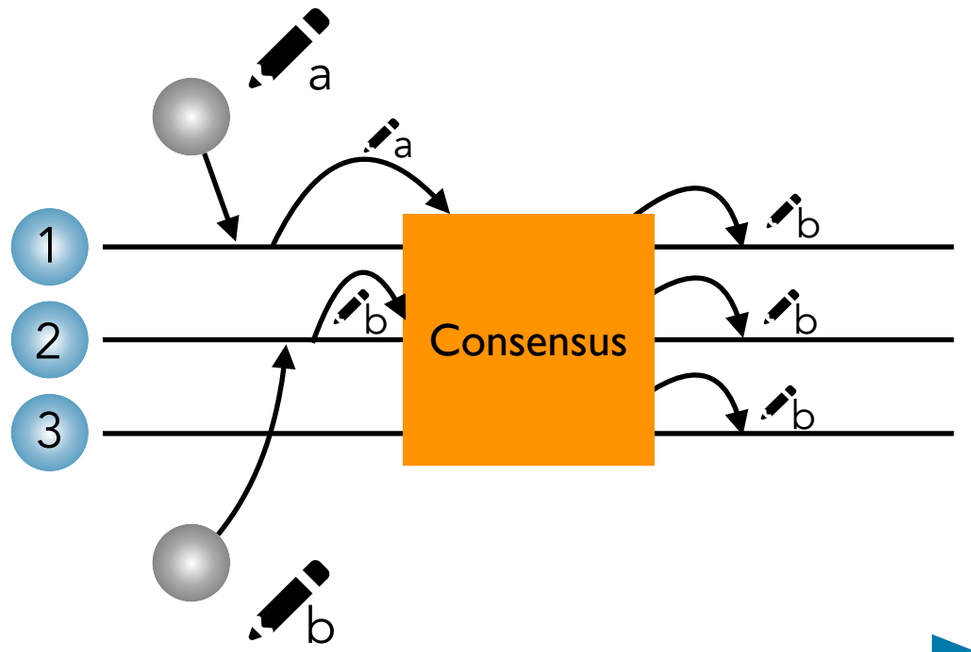
Towards a replicated state-machine - Atomic broadcast

- 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**
  - If not the proposal of a single operation would not be accepted and therefore never chosen 😰
  - However, if several operations are proposed we can run into a deadlock 😱 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 *uniquely numbered*: (number, operation)
- **P2: If a proposal with operation *op* is chosen, then every *higher-numbered proposal* that is chosen has operation *op***

Distributed Consensus

The Paxos algorithm

- **P2: If a proposal with operation *op* is chosen, then every higher-numbered proposal that is chosen has operation *op***
- **P2a: If a proposal with operation *op* is chosen, then every higher-numbered proposal that is accepted has operation *op***
  - Hmmm… 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 *op* is chosen, then every higher-numbered proposal that is proposed has value *op***
  - P2b (proposed) $\Longrightarrow$P2a (accepted) $\Longrightarrow$ P2 (chosen)

Distributed Consensus

The Paxos algorithm

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

Distributed Consensus

The Paxos algorithm

- From (1) and (2):
  - Every acceptor in C has accepted a proposal with a number in *[m, n -1]*, and every proposal with a number in *[m, n -1]* accepted by any acceptor has operation *op*
  - 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 *op* by ensuring that the following invariant is maintained:
- **P2c: For any *n* and *op*, if a proposal with (*n*, *op)* 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) *op* 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 a number less than *n*, if any, that **has been** or **will be accepted** by each acceptor in some majority of acceptors.
  - Learning about proposals that have been accepted is easy enough, just ask
  - Predicting future acceptances is usually hard… 🙄
- Instead, the proposer requests that any acceptors promise not to 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 a number *n* greater than that of any prepare request to which it has already responded, then it responds to the request with the highest-numbered proposal *(h, x)* that it has accepted (if any) and a promise not to accept any more proposals numbered less than *n*.
- **Phase 2**
  - (a) If the proposer receives a response to its **prepare request** (numbered n) from a majority of acceptors, then it sends an **accept request** to each of those acceptors for a proposal *(n, op)*, where *op* is the operation *x* of the highest-numbered proposal among the responses, or is any operation if the responses reported no accepted 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

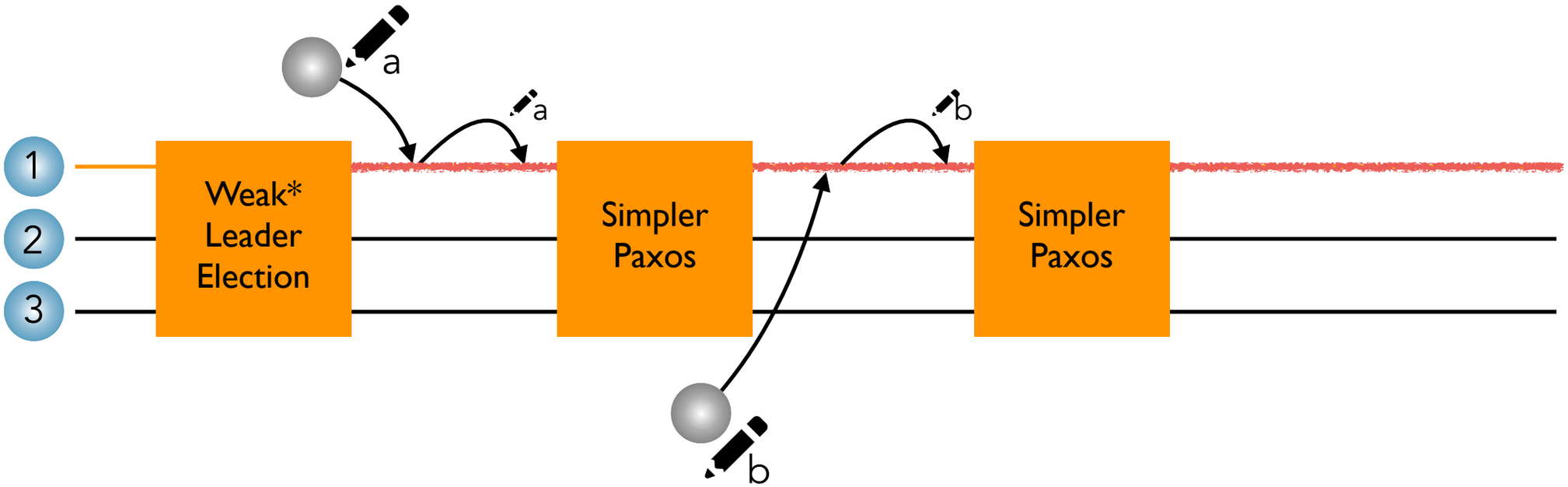
- It's easy to construct a scenario in which two proposers each keep issuing a sequence of proposals with increasing numbers, none of which are ever chosen. 😳

- Proposer *p* completes Phase 1 for 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



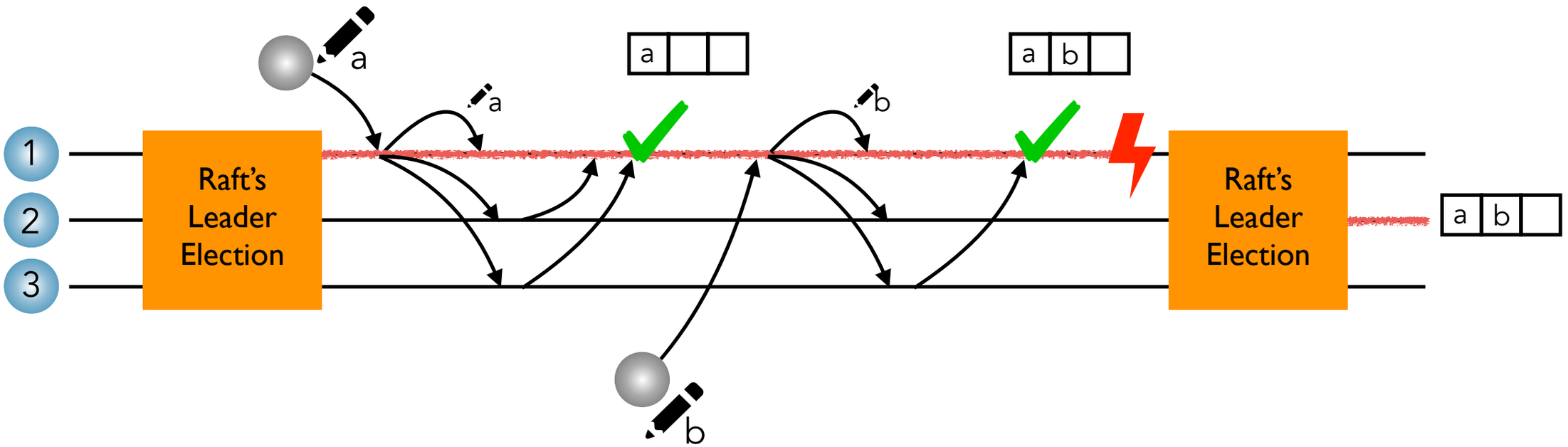
- Ensuring a unique leader is harder than solving Consensus so a weaker form of leader election is actually adopted, and sufficient

- 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

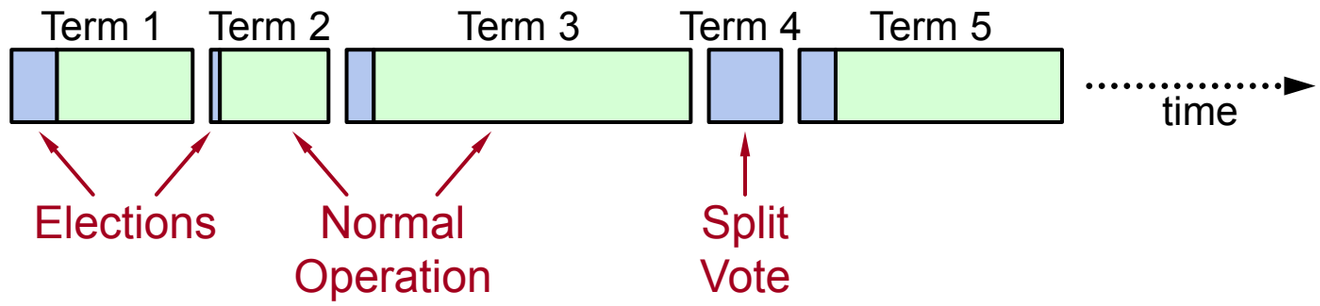
The Raft algorithm - A distinguished proposer to guarantee progress

- 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's* (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 heartbeats then it starts an election to choose a new one: it increments **its** term counter 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 state* and starts sending heartbeats.
  - (b) it receives heartbeats from a leader from a term greater or equal than its term counter: transitions to *follower state*.
  - (c) a pre-configured timeout expires with no winner: starts a new election.
- A candidate wins an election if it receives a **majority of votes for the same term**. Each replica **votes at most once per term**, on a FCFS 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

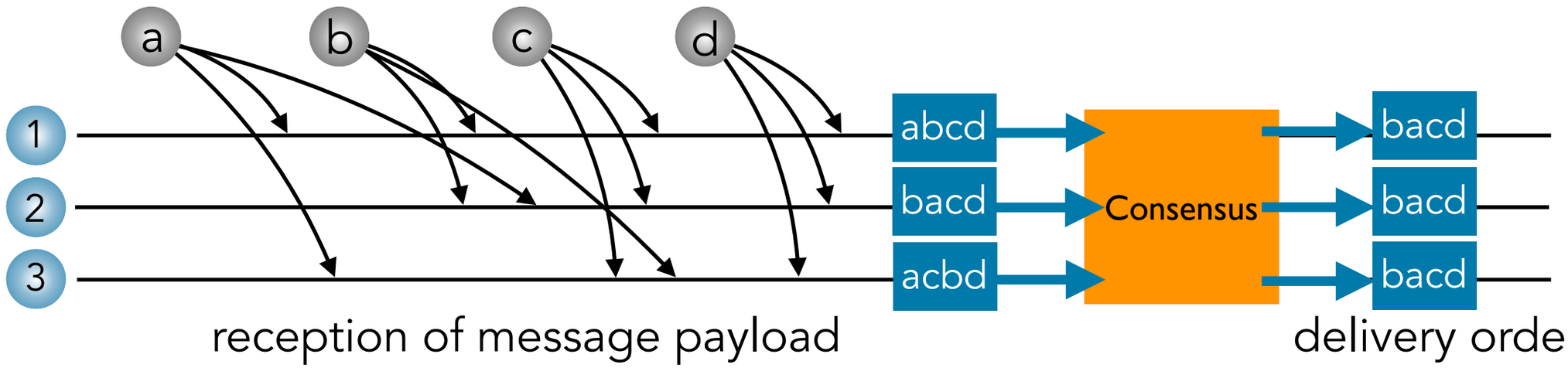
The Raft algorithm - Log replication

- Raft ensures — Log Matching Property — that:
  - **If two entries in different logs have the same index and term, then:**
    - **they store the same operation**
    - **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 phase* as a process cannot become a leader before having all the committed entries from previous terms in its log.

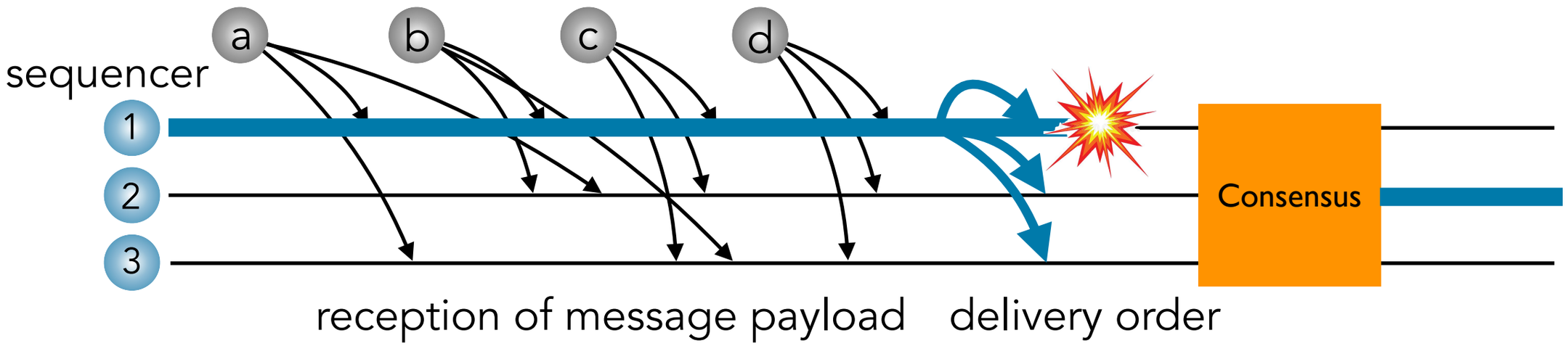
Distributed Consensus

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)