TÉCNICO LISBOA

Consensus protocols

Highly dependable systems

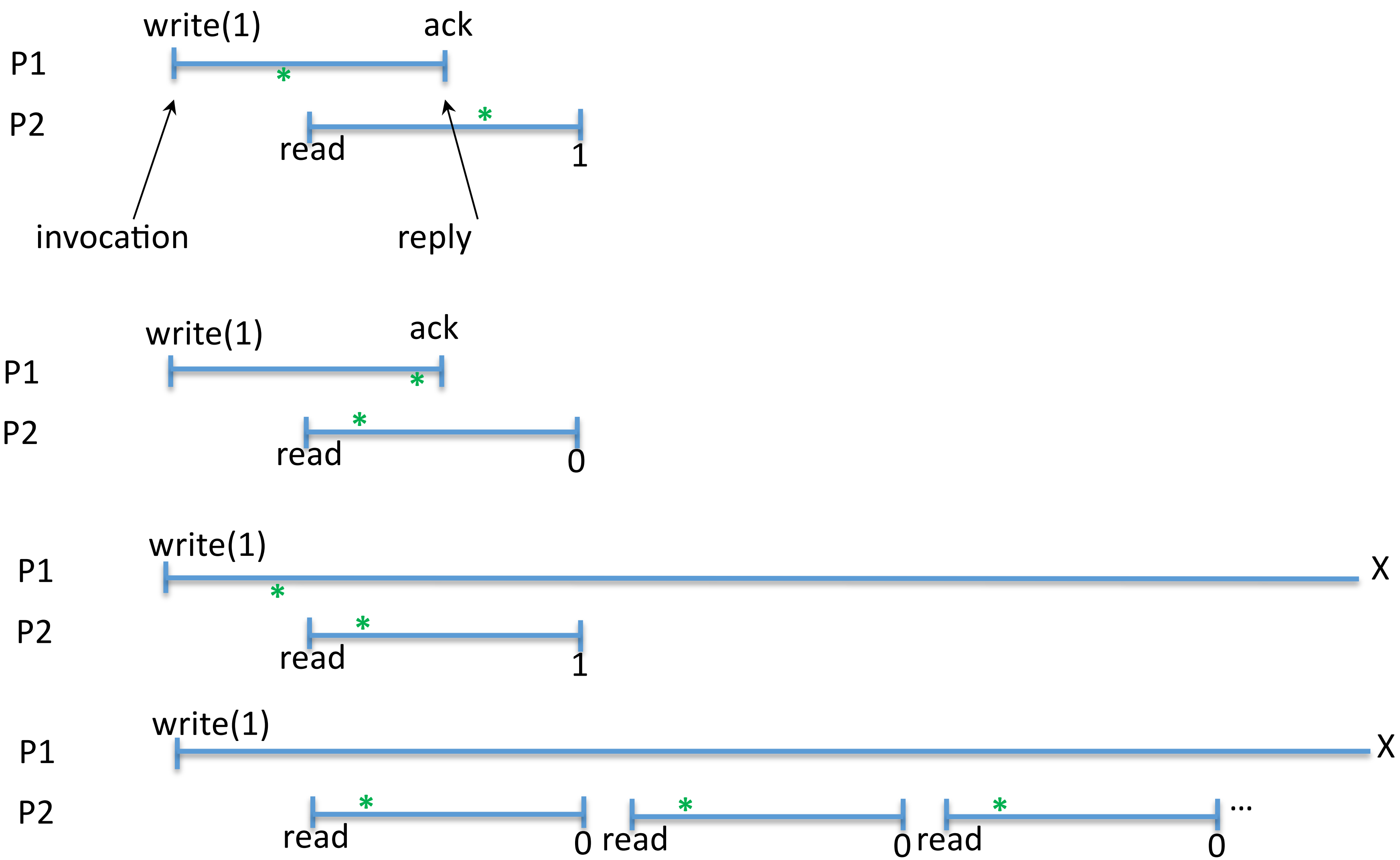
Lecture 6

Lecturers: Miguel Matos and Paolo Romano

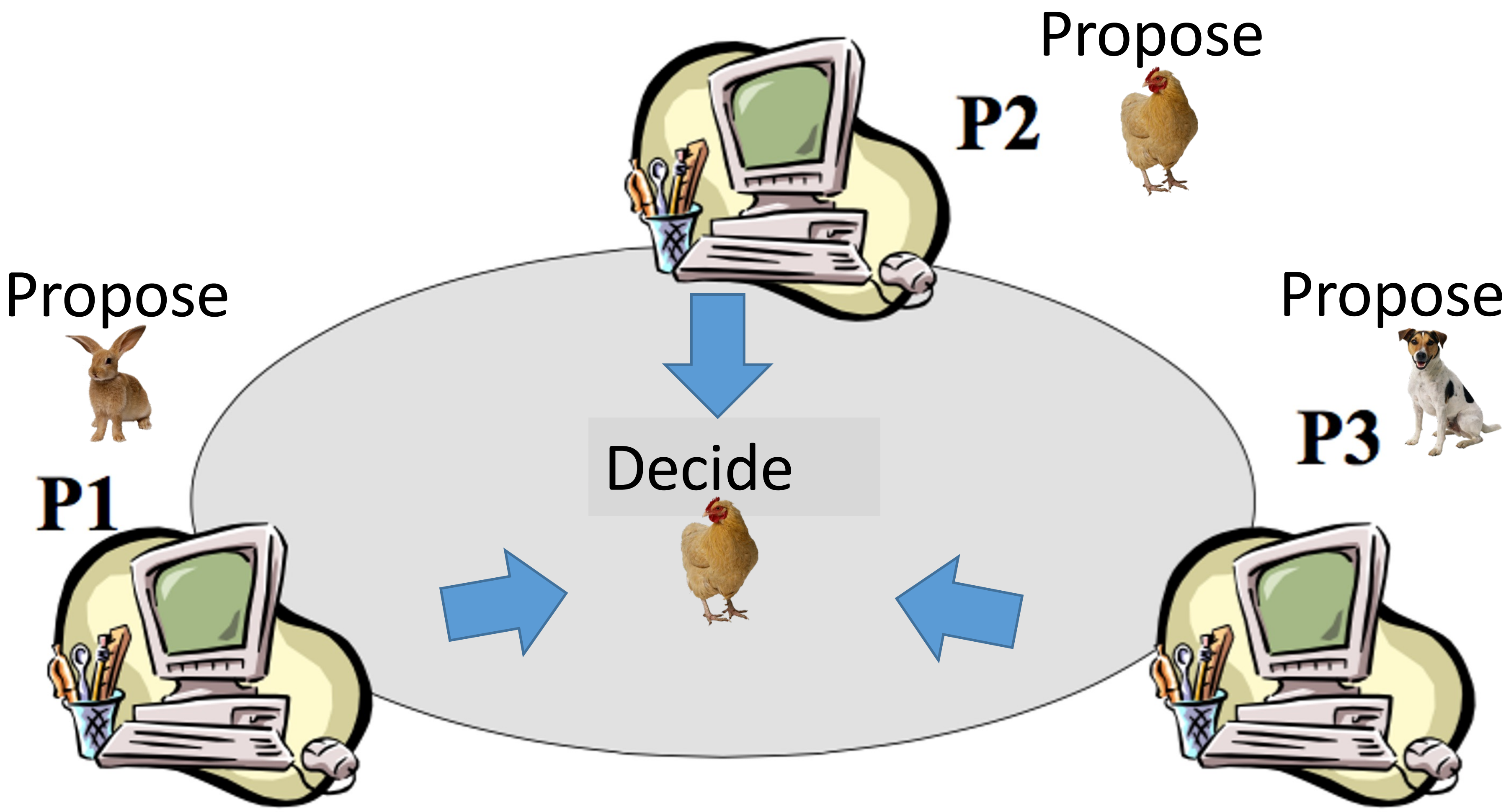
Last lecture: atomicity / linearizability (works for (N,N)-atomic registers as well)

- For any operation, there exists a **serialization point**, between the invocation and the reply, such that if we move the invocation and the reply to that point, the resulting execution obeys the sequential specification of a read/write register (operations appear to be executed at some instant between its invocation and reply time)
  - If the last operation does not return, the serialization point may or may not be included
  - (failed writes may or may not complete)

Examples of atomic executions



Consensus



The consensus problem

- Basic idea: each process has an input proposal
- All processes must reach the same output decision
- Must be safe despite faults, asynchrony
- This is a key building block in many systems
  - generic state machine replication
  - coordination systems like Apache ZooKeeper (CFT)
  - permissioned blockchains or permissionless side chains (BFT)

Specification in the crash model: Uniform consensus

- Events:
  - Request: <Propose, v>
  - Indication: <Decide, v'>
- Properties:

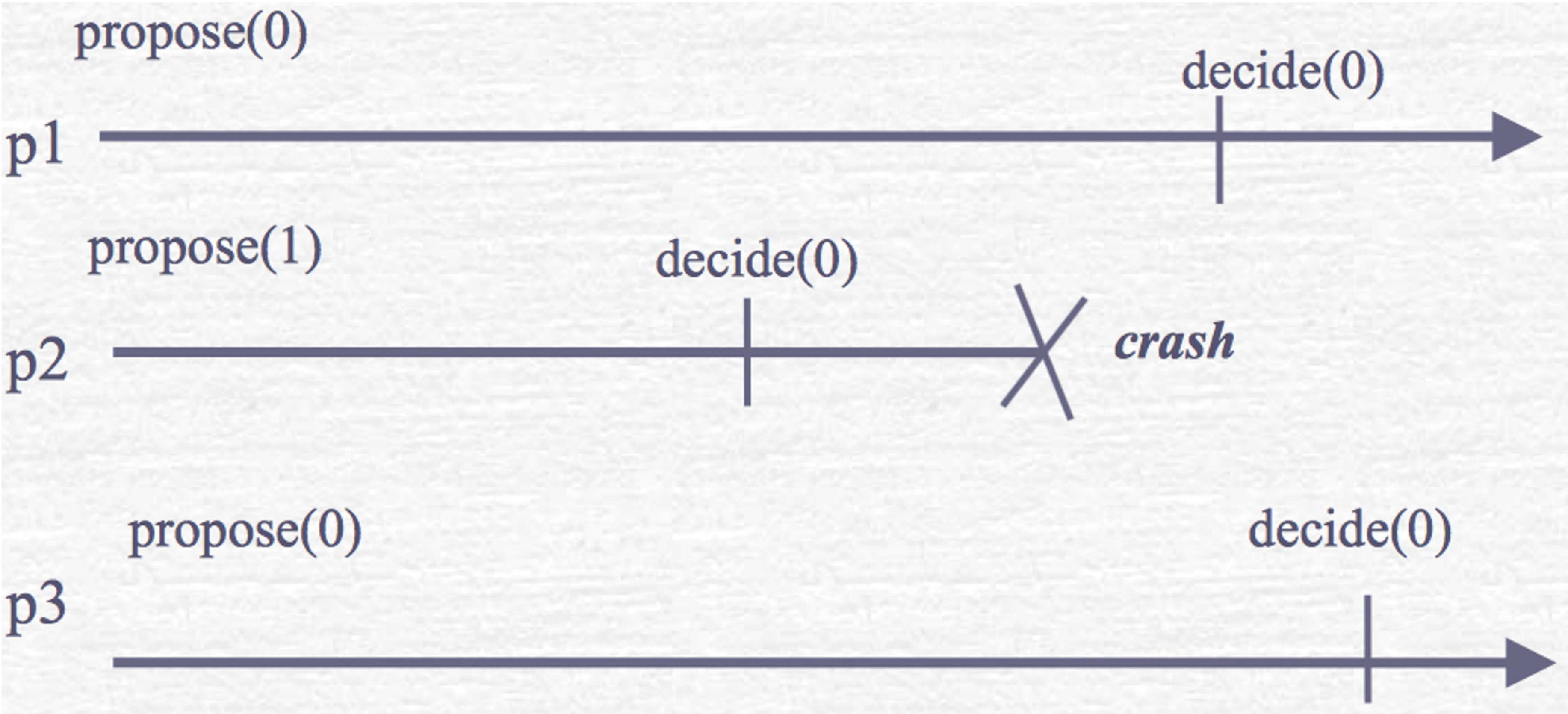
C1. Validity: Any value decided is a value proposed

C2. [Uniform] Agreement: No two processes decide differently

C3. Termination: Every correct process eventually decides

C4. Integrity: No process decides twice

Example of a valid trace



Algorithm to solve consensus in the crash model: Paxos

- Submitted for publication in 1990
- Reviewers said it was mildly interesting, though not very important – and that the presentation was distracting
- Paper was rejected and shelved
- Eventually published after a decade
- Then adopted at Google (published in 2006)
- Now a standard building block used by many systems

Paxos in a nutshell

- This is covered in another course, plus our focus is not on the crash model
- Here, we give a brief outline

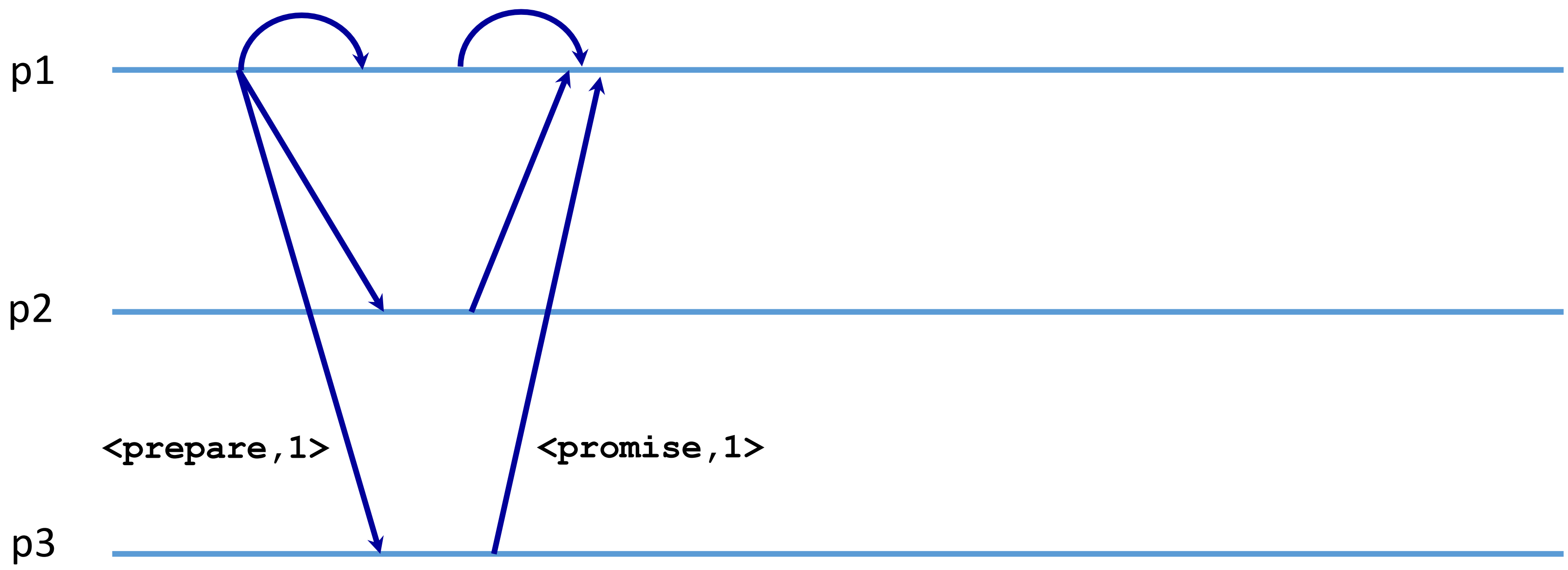
Overview

- Any process can propose **v**, first to reach a majority wins
- How do we select among multiple proposals?
- Associate timestamp <seqno, process id> with **v**
- Protocol has two phases:
  - First, processes read the state of others to form proposal
  - Second, try to convince others to accept their proposal

Protocol steps (first phase)

1. Process p chooses a proposal timestamp n = [sn,p]
2. All processes keep track of:
   - timestamp accepted and associated value $<n_a , v_a>$, and
   - most recent promise not to accept lower timestamps, n_h
3. p sends `prepare` msg, asking all processes if they already accepted any proposals with $n_a < n$
4. if so, reply $<n_a , v_a>$ else set $n_h = n$ (and return this `promise` not to accept anything below n)

First phase example run



Protocol steps (second phase)

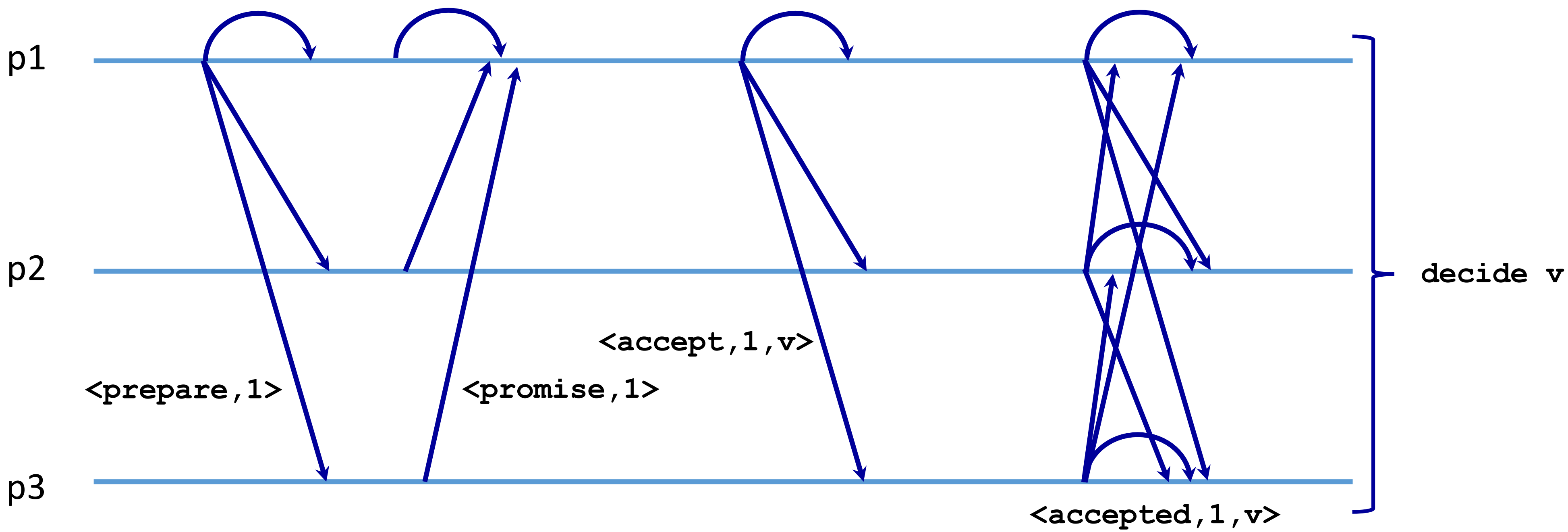
1. After p collects quorum of replies, send either a previously accepted value (if it was received) or its own proposal in an <`accept`, <n,v>> message
2. Processes accept proposal if $n \geq n_h$ setting:

   $n_h = n_a = n$

   $v_a = v$

(Then convey decision to all processes through `accepted` message)

Second phase example run

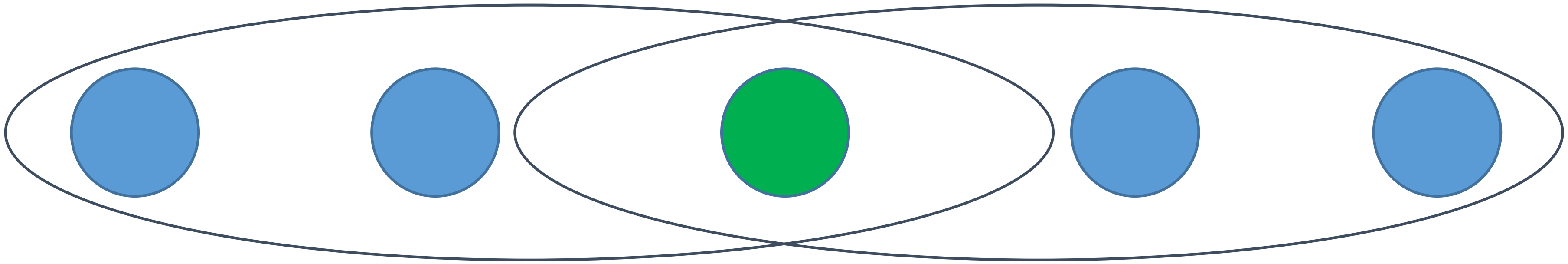


Paxos in practice – multi-Paxos

- Instead of running two phases for every “operation”:
  - use phase 1 to nominate a leader (run phase 1 for all possible operations / instances of consensus)
  - let the leader run phase 2 each time an operation is executed (thus concluding one of the consensus instances)
  - if leader is non-responsive, then goto first step
- Parallel to IBFT (phase 1 is a round change, phase 2 is the normal case operation)

Why is Paxos safe?

- Agreement is guaranteed by the fact that if a proposal with v is accepted (majority of `accept`s were issued), then any higher-numbered proposal must have value v



v is chosen →
Quorum accepted <n,v>

An attempt to decide in
n+1 will have to propose v

But is it live?

Impossibility of consensus (FLP)

- There is no **deterministic** protocol that solves consensus in an asynchronous system where even a single process may suffer a crash fault
  - Fisher, Lynch, and Paterson. Impossibility of distributed consensus with one faulty process. JACM, Vol. 32, no. 2, April 1985, pp. 374-382
- We will present a simple and elegant proof for consensus among two processes
  - The main result applies to an arbitrary number of processes

Proof of the impossibility of consensus

- By contradiction, let's consider that there exists an algorithm that solves consensus
- We consider three different executions of that algorithm, with varying network conditions
  - Note that any behavior from the network is possible in an asynchronous system
- The two processes executing consensus are called A and B

Execution #1

- Both processes propose 0 initially
- Process B crashes as soon as the execution starts
- By the validity condition of the specification, process A must decide 0
- And by the termination property it must eventually decide → let's say it decides at some instant t1

Execution #2

- Both processes propose 1 initially
- Process A crashes as soon as the execution starts
- By the validity condition of the specification, process B must decide 1
- And by the termination property it must eventually decide → let's say it decides at some instant t2

Execution #3

- Process A proposes 0 and process B proposes 1 initially
- Messages between A and B (in both directions) are delayed such that they are never delivered up until max(t1,t2)
- Process A decides 0 by t1, since its execution is indistinguishable from execution #1
- Process B decides 1 by t2, since its execution is indistinguishable from execution #2
- We found a contradiction (which?)

Byzantine fault tolerant consensus

- Recall previous specification (crash model):
  - Termination: Every correct process eventually decides
  - Validity: Any value decided is a value proposed
  - Integrity: No process decides twice
  - Agreement: No two processes decide differently
- Which property needs to be revisited in the Byzantine model?

Weak Byzantine consensus

- Termination: Correct processes eventually decide.
- Weak validity: If all processes are correct and some process decides v, then v was proposed by some process.
  - If some processes are faulty, any value may be decided
- Integrity: No correct process decides twice.
- Agreement: No two correct processes decide differently.

Strong Byzantine consensus

- Strong validity: If all correct processes propose the same value v, then no correct process decides a value different from v;

  otherwise, a correct process may only decide a value that was proposed by **some** correct process or the special value □

Weak vs Strong Byzantine consensus

- *Strong validity* does *not* imply weak validity
- Strong validity allows to decide □
- *Weak validity* requires (only if all processes are correct) that the decided value was proposed by some (correct) process
- The two Byzantine consensus notions are not directly comparable
- For this class, we **focus on weak validity**

Implementing BFT consensus

- Strategy is similar to Paxos, i.e., modularize into:
- EpochChange
  - Choose a leader, and make sure any previously decided value carries over to the new epoch
- EpochConsensus
  - Try to reach decision within an epoch
  - May fail, in which case it aborts and returns state to initialize new EpochConsensus

Byzantine Epoch Change

- Leverage Byzantine leader election protocol from Lecture 3
- Recap: if the consensus algorithm is not making progress (timeout), process i broadcasts a NEWEPOCH message to all processes.
- If a process receives more than f NEWEPOCH messages, also broadcasts NEWEPOCH
  - Prevents unwanted epoch change. Why?
- If a process receives more than 2f NEWEPOCH messages it changes epoch.
  - Cannot wait for more. Why?

EpochConsensus: interface

- Tries to achieve consensus within an epoch, but may abort unless leader is correct and network behaves synchronously
- Interface (events):
  - **Request**: ⟨ bep, Propose | v ⟩: Proposes value v for epoch consensus. Executed only by the leader l.
  - **Request**: ⟨ bep, Abort ⟩: Aborts epoch consensus.
  - **Indication**: ⟨ bep, Decide | v ⟩: Outputs a decided value v of epoch consensus.
  - **Indication**: ⟨ bep, Aborted | st ⟩: Signals that epoch consensus has completed the abort and outputs internal state st.

EpochConsensus: specification (for epoch with timestamp *ts*)

- *Validity:*
  If (all processes are correct and) a process ep-decides v, then v was ep-proposed by a leader of epoch consensus with timestamp ts′ ≤ ts.
- *Uniform agreement:*
  No two correct processes ep-decide different values.
- *Lock-in:*
  If a correct process ep-decided v in an epoch consensus with timestamp ts′ < ts, processes cannot decide a value v'≠v.
- *Termination*:
  If the leader is correct, has ep-proposed a value, and no correct process aborts this epoch consensus, then every correct process eventually ep-decides

Byzantine Epoch Consensus (read phase)

- Leader sends READ to all processes
- Processes reply with STATE message containing its local state <valts, val, writeset>:
  1. (**valts, val**) - a timestamp/value pair with the value that the process received most recently in a **Byzantine quorum of WRITE messages**
  2. **writeset** - a set of timestamp/value pairs with one entry for every value that this process has ever written (where timestamp == most recent epoch where the value was written).

Outcome of the read phase

- Read phase obtains the states from a byz. quorum of processes to determine whether there exists a value that may have been epoch-decided (if so, it must be written, to ensure lock-in property)
- If so, send this value in the subsequent WRITE
- What are the required conditions to be able to affirm that a value may have been epoch-decided?

Outcome of the read phase

1. The value corresponds to the highest timestamp in a byzantine quorum of (timestamp,value) pairs reported in distinct STATE messages
   - This is the most recent value for which a process claims to have received a Byzantine quorum of WRITEs
2. The value appears in the writeset of at least f+1 processes
   - This ensures value occurs in the writeset of a correct process

- If no value meets these two conditions, then outcome is *unbound*

Read phase: coping with byzantine leaders

- Leader sends the STATEs collected in the read phase to all
  - processes send their states digitally signed, to prevent tampering
- All processes independently check, based on information in state messages, if some value may have already been ep-decided in a previous epoch (lock-in property)

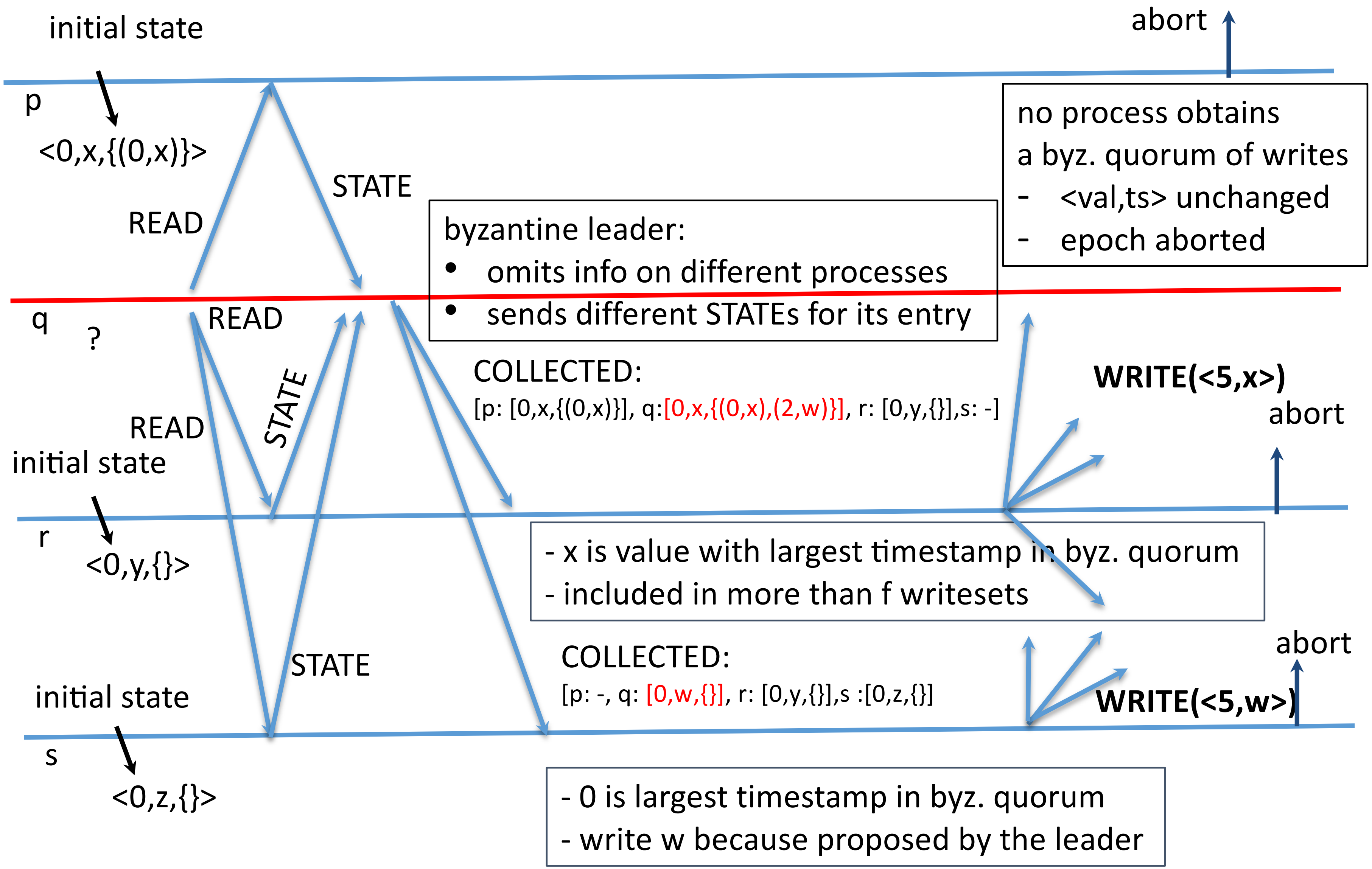
Read phase: coping with byzantine leaders

- A leader cannot forge STATE values of other processes, thanks to the use of digital signatures
  - but it can omit information from some process
  - or send different values regarding its state to different processes
- However, the conditions governing the outcome of the read phase prevent safety violations

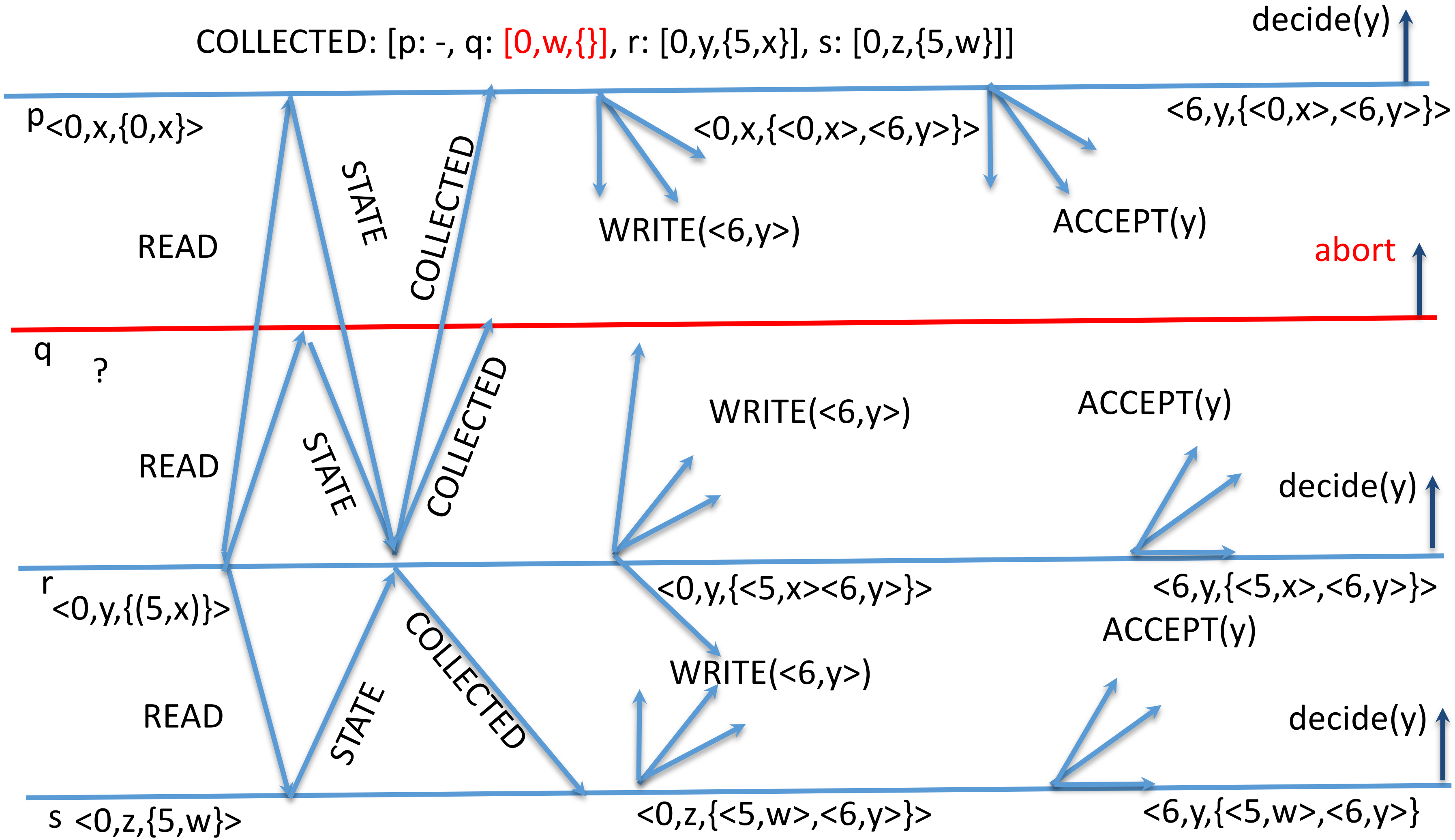
Write phase

- If a process receives a Byzantine quorum of WRITE messages from distinct processes containing the same value v, it sets its state to (current_epoch, v) and broadcasts an ACCEPT message
- When a process receives a Byz. quorum of ACCEPT messages from distinct processes containing the same value v, it epoch-decides v

Example execution: byzantine leader q in epoch 5



Example execution: correct leader r in epoch 6



r is correct → all processes get the same COLLECTED unless they time out (asynchrony), they will all write the same value, and accept it

Correctness sketch

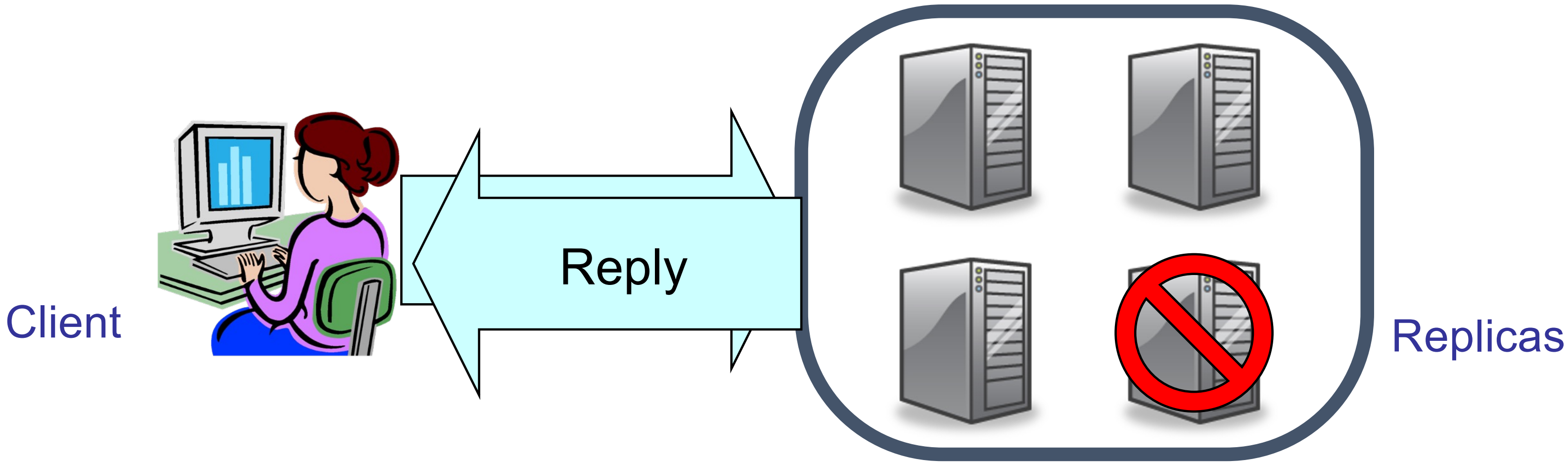
- Agreement property:
  - Usual contradiction proof based on collecting 2f+1 ACCEPTs, and the fact that correct processes do not send conflicting ACCEPT messages
- Validity property:
  - Weak validity applies only to executions with only correct processes, simplifying the proof
- Termination and abort behavior property:
  - Follows from sequence of steps after correct leader starts the protocol

Correctness sketch (lock-in property)

- assume process p ep-decided v in consensus instance ts’ < ts
- then, p collected 2f+1 ACCEPTs for v, at least f+1 from correct processes, who set value and timestamp to <v,ts’>
- those ACCEPTS follow from receiving 2f+1 WRITEs, at least f+1 from correct processes, who added (ts’,v) to their writeset
- now let’s consider the first subsequent instance ts* where a correct process receives COLLECTED, we prove that the outcome of the read phase has to be v
  - Between ts’ and ts* no correct process received COLLECTED, thus did not send write, thus state variables valts, val, and writeset did not change
  - Thus the f+1 correct processes use (ts’,v) as the starting value of ts* and include it in writeset
  - By construction of the outcome of the read phase, its output must be bound to ts’
  - Therefore, all correct processes that write will write v, implies that correct processes that decide will decide v in ts*
  - Recursively using the same argument until round ts establishes the property

State machine replication (SMR)

1. Take an arbitrary service, make it deterministic
   Example: an append-only sequence of blocks of transactions
2. Replicate the server
3. Enforce that correct replicas execute request in the same order (follow the same sequence of state transitions)
4. Use voting to guarantee that client sees correct output



From consensus to state machine replication

- Consensus protocol is at the heart of solving point number 3
  - Clients issue several requests independently of each other
  - Each request is assigned a sequence number, thus defining order by which they are executed
  - Instantiate one consensus instance per sequence number, to determine which request gets executed at that point in the sequence
- Can optimize the EpochConsensus protocol for this setting:
- When instantiating new epoch, read phase of the protocol can be executed only once for requests in the interval [current, +∞)

Acknowledgements

- Rachid Guerraoui, EPFL