DISTRIBUTED SYSTEMS

CSCI 4510/6510

PAXOS

The Problem

- We need to store a log of events (decrees)

Event 1	Event 2	Event 3	Event 4	Event 5	….

- Want to replicate this log at multiple sites.
  - Need the event to appear in the same order in every log.
  - Need every event to eventually appear in every log.
- Challenges
  - Concurrency – different processes want to write to log at same time
  - Failures – processes and channels

System Model

- Processes may fail by crashing and may restart.
  - Both crash failures and crash/recovery processes.
  - Processes do not lose state when they crash and recover.
  - No Byzantine failures.

- Messaging is asynchronous.
  - Messages may take arbitrarily long to be delivered.
  - Messages can be duplicated.
  - Messages can be lost.
  - Messages cannot be corrupted.

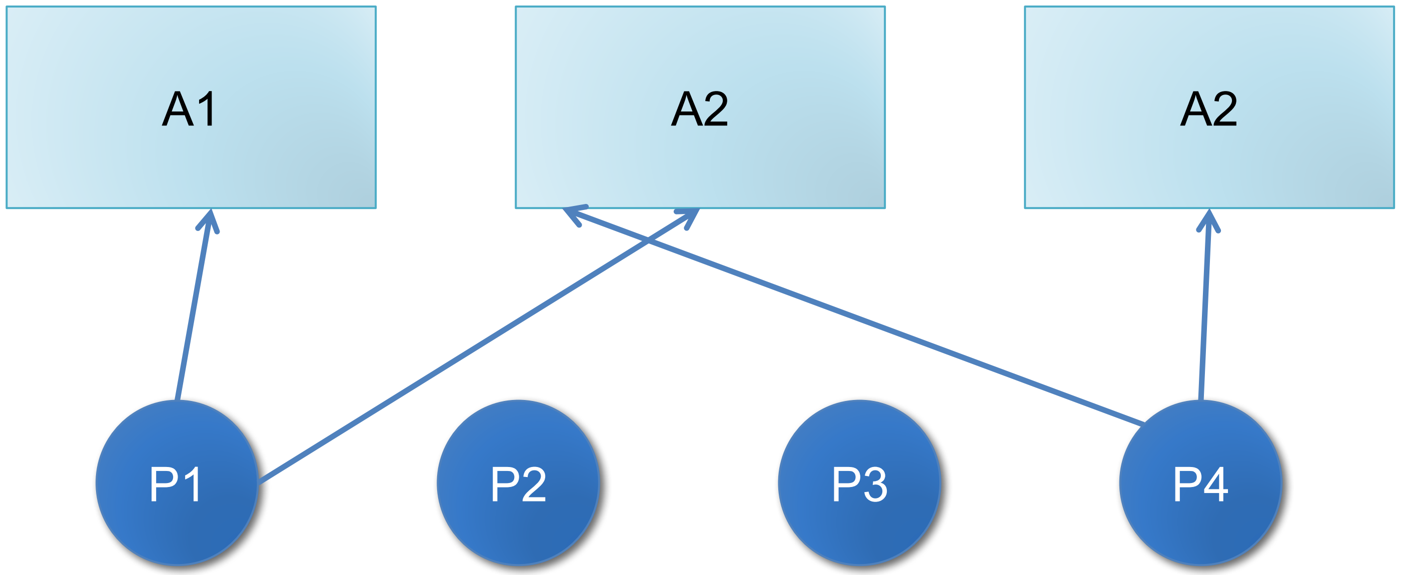
The Paxos Algorithm

- Initial approach: consider the replicated log problem as a sequence of consensus problems, one per log position.

Event 1	Event 2	Event 3	Event 4	Event 5	….

- The algorithm used to reach consensus on a single log entry is called the **Synod Algorithm**.
- Later: show how to make this initial approach more efficient – this is **Paxos**.

Roles in the Consensus Algorithm



- **Proposers**: propose value to chosen (the consensus value)
- **Acceptors**: decide whether to accept a proposed value
- The value is **chosen** when enough acceptors accept it.
  - Enough = a majority
- **Learners**: must all learn the consensus value (after it is chosen)
  - For our discussion, Acceptors = Learners

Requirements for Consensus Algorithm

- Safety
  - Only a value that has has been proposed may be chosen (decided as consensus value).
  - Only a single value is chosen.
  - A process never learns that a value has been chosen unless it actually has been (can’t change the log).
- Liveness
  - No precise requirement
  - Goal is that eventually some proposed value is chosen, and then eventually a process can learn the chosen value.

Desired Requirements

- (P1) An acceptor must accept the first value it receives.
  - Means an acceptor must be able to accept more than one proposal
  - Proposal has number and value (n,v)

- (P2) If a proposal with value v is chosen, then every higher numbered proposal that is also chosen has value v.

Final Requirement for Proposer (Invariant)

- (P2) If a proposal with value v is chosen, then every higher numbered proposal that is also chosen has value v.
- (P2c) For any v and n, if proposal number n is issued with value v, then there is a majority set S such that either
  - (1) No accepter in S has accepted any value OR
  - (2) v is the value of the highest-numbered proposal with number less than n that was accepted by any acceptor in S.

Algorithm for Proposer

- To propose a value:
  - Picks a new proposal number n
  - Sends “prepare(n)” message to set of acceptors, requesting:
    - promise not to accept any proposals with lower number than n
    - response with highest-numbered proposal (less than n) that is has accepted

- If proposer receives response from majority of acceptors:
  - Create proposal (n,v)
    - If no responding acceptor has accepted a value, v its own value
    - Otherwise, v is value from with highest-numbered proposal accepted by a responding acceptor
  - Send “accept(n,v)” message to set of acceptors

Final Requirement for Acceptor (Invariant)

- (P1) An acceptor must accept the first value it receives.
- An acceptor can accept a proposal only if it has not promised not to.
- (P1a) An acceptor can accept a proposal numbered n iff it has not responded to a prepare request having proposal number greater than n.

Algorithm for Acceptor

- Acceptor can receive: “prepare(n)” and “accept(n,v)”

- On receiving “prepare(n)”:
  - Respond only if n is greater than proposal number of any it has already responded to
  - Send (m,w) for highest-numbered proposal it has accepted, if any.
  - This response is “promise” not to accept any proposal numbered less than n.

- On receiving “accept(n,v)”:
  - If not responded to proposal with number greater than n
    - Accept proposal (n,v)

Synod Algorithm Implementation

- Proposers are stateless
  - Safety is not violated if a proposer crashes
  - If proposer recovers, it can start over
  - To ensure liveness, we need some proposer to stay alive "long enough"

- Each acceptor stores
  - maxPrepare: largest proposal number for which it has responded to a prepare message (initially 0)
  - accNum: largest proposal number of proposal it has accepted (initially null)
  - accVal: value of proposal numbered accNum (initially null)

  - The acceptor state must survive crash/recovery

Execution of Synod Algorithm

Proposer | **Acceptors**

1. Choose new proposal number *n.*
   Send *prepare(n)* to **all** acceptors.

$\xrightarrow{\textit{propose(n)}}$

2. If *n > maxPrepare*
      maxPrepare = n,
      reply with *(accNum, accVal)*

$\xleftarrow{\textit{promise (accNum, accVal)}}$

3. If receive response from majority
      choose value *v,*
      send *accept(n,v)* to **all** acceptors
   Else, start over

v = value with largest *accNum* number

Only if all *accVal* values are *null*, choose own value.

$\xrightarrow{\textit{accept(n, v)}}$

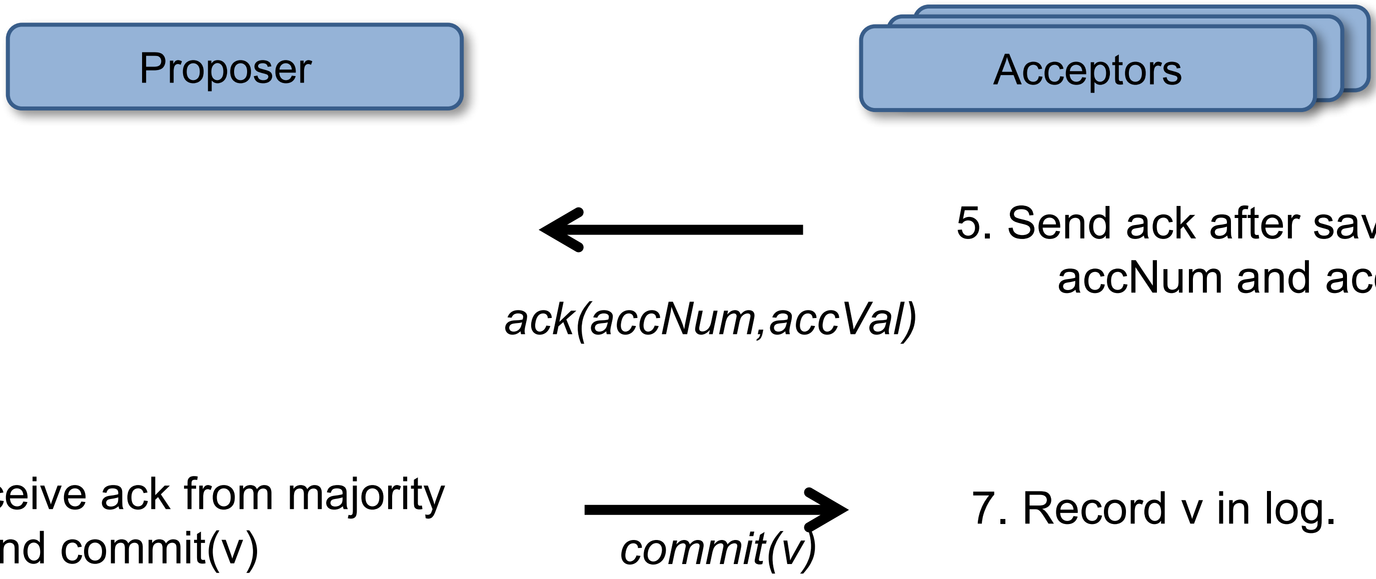
4. If *n ≥ maxPrepare*
      accNum = n
      accVal = v
      maxPrepare = n

Learning the Accepted Value

- A value is chosen when a majority of acceptors accept it.
- How can the learners (acceptors) determine when this has happened?

- Only once a value has been chosen can it be written in the log.

Execution of Paxos Algorithm (2)



5. Send ack after saving accNum and accVal

6. If receive ack from majority send commit(v)

7. Record v in log.

Log is stored in stable storage.

With message loss, there is no guarantee that all learners will eventually learn the chosen value.

Correctness of Synod Algorithm

- Safety is guaranteed by algorithm construction.

- What about liveness?
  - Do a majority of acceptors eventually accept a value?
  - Even if acceptors do not fail and messages are not lost?

- Need majority of acceptors to be alive for progress.

- Multiple proposers can keep issuing prepare requests with higher numbers.
  - Possible no proposal gets majority of “promises”.

Ensuring Progress

- Can ensure liveness by having a **distinguished proposer**.
  - All proposers funnel proposals through this distinguished proposer.
  - Distinguished proposer issues proposals one at a time.
- Use leader election algorithm to elect this distinguished process.

Full Paxos Algorithm

- The Synod algorithm is used to determined consensus value for a single log entry.

- The Paxos algorithm is a sequence of Synod algorithms.
  - With some optimizations.

Full Paxos Algorithm

- Use leader election algorithm of your choice to elect a leader (to act as distinguished proposer).
  - .
- When leader elected, it may have missing log entries.
  - It runs Synod algorithm (acts as proposer and acceptor) to learn consensus value for those entries.
  - Uses a dummy value for these entries.
- Once holes in log are filled, leader can start processing new requests to create log entries.

Paxos Optimization

- If leader is stable, only need to do “prepare-promise” phase once (when first elected)
  - Until leader’s first value is accepted.

- After first value is accepted, leader just uses “accept-ack-commit” phase for subsequent values.
  - All acceptors have an implicit promise to the leader’s proposal.
  - Call this proposal number 0.

PROJECT 2

Basic Implementation (no leader)

- All tweet, block, and unblock events should be replicated in every log.
- Event creation:
  - When user creates new event, client (proposer) initiates full Synod algorithm to replicate this event at every site.
  - If there are competing proposals or site failures, the proposal may fail – no event created in log.
  - System should report to client whether event was created in the log or not.

Basic Implementation (no leader)

- Support for crash failures and recovery:
  - When a site recovers, it needs to learn the log entries it may have missed.
  - It does this by executing full Synod algorithm for log entries after its last log entry – propose dummy value.
  - Don't want to execute infinitely many Synod instances.
  - How can site discover when to stop?
- If you are using UDP, may end up with "holes" in log
  - Hole created if "commit" message is lost.
  - Need to periodically check for holes and execute Synod algorithm to fill in the hole (propose dummy value).
- No log entries can be created unless a majority of sites are up.

Paxos with a Leader

(minor change from Project Spec to make your life easier)

- The user ID will be stored as part of the log entry.
  - Log entry should contain: (user ID, operation details)

- The leader for log entry K will be the site corresponding to the user ID of log entry K-1.
- The leader can issue proposal number 0.
  - All acceptors have an implicit promise to proposal number 0.
  - The leader can skip the propose/promise phase, and begin with "accept(0,v)"
  - If the acceptor has prepareNum > 0, it will ignore this accept message.
  - If leader times out waiting for acks for "accept(0,v)", it should start again with higher proposal number, and execute full Synod algorithm.
- All other sites should use full Synod algorithm.

Choosing a Proposal Number

- Proposal numbers must be unique (and increasing)
- How can proposers choose proposal number?

Demonstration Details

- Your application should show “debug” information in the UI.
  - What messages are sent.
  - What messages are received.
  - When a log entry is created
  - Whether a client’s proposal is accepted or not.

- Please make this debug information compact and readable.