Paxos

- Fault-tolerant distributed consensus algorithm
  - Does not block if a majority of processes are working
  - The algorithm needs \((2P+1)\) processors survive the simultaneous failure of \(P\) processors

- Goal: provide a consistent ordering of events from multiple clients
  - All machines running the algorithm agree on a proposed value from a client
  - The value can be associated with an event or action
  - Paxos ensures that no other machine associates the value with another event

- Abortable consensus
  - A client’s request may be rejected
  - It then has to re-issue the request
while (submit_request(R) != ACCEPTED) ;
Paxos players

- **Client**: makes a request

- **Proposers**:
  - Get a request from a client and run the protocol
  - **Leader**: elected coordinator among the proposers (not necessary but simplifies message numbering and ensures no contention) – we don’t count on the presence of a single leader

- **Acceptors**:
  - Multiple processes that remember the state of the protocol
  - **Quorum** = any majority of acceptors

- **Learners**:
  - When agreement has been reached by acceptors, a Learner executes the request and/or sends a response back to the client
What Paxos does

• Paxos ensures a consistent ordering in a cluster of machines
  – Events are ordered by sequential event IDs (N)

• Client wants to log an event: sends request to a Proposer
  – E.g., value, v = “add $100 to my checking account”

• Proposer
  – Increments the latest event ID it knows about
  – Asks all the acceptors to reserve that event ID
  – The algorithm has to handle the cases of:
    • Another proposer asking to reserve the same event ID
    • Another proposer already reserved the same ID

• Acceptors
  – A majority of acceptors have to accept the requested event ID
Proposal Numbers

• Each proposal has a number (created by proposer)
  – Must be unique (e.g., <sequence #>.<process_id>)

• Newer proposals take precedence over older ones

• Each acceptor
  – Keeps track of the largest number it has seen so far

• Lower proposal numbers get rejected
  – Acceptor sends back the {number, value} of the currently accepted proposal
  – Proposer has to “play fair”:
    • It will ask the acceptors to accept the {number, value}
    • Either its own or the one it got from the acceptor
Paxos in action

Goal: have all acceptors agree to a value \( v \) associated with a proposal

Paxos nodes: one machine may serve several roles

Client

Proposer

Proposer

Proposer

Leader

Learner

Learner

Acceptor

Acceptor

Acceptor

Quorum

Acceptor

Acceptor

Acceptor

October 29, 2014

© 2014 Paul Krzyzanowski
Paxos in action: Phase 0

Client sends a request to a proposer

Client sends a request to a proposer

Client

Proposer

request(v)

Learner

Quorum

Acceptor

Acceptor

Acceptor
**Paxos in action: Phase 1a**

**Proposer:** creates a proposal \#\(N\) (\(N\) acts like a Lamport time stamp), where \(N\) is greater than any previous proposal number used by this proposer

**Send to Quorum of Acceptors** (however many you can reach – but a majority)

\[
N = \langle \text{seq#} \cdot \text{process_ID} \rangle
\]
**Paxos in action: Phase 1b**

**Acceptor:** if proposer’s ID > any previous proposal
   promise to ignore all requests with IDs < N
   reply with info about highest past proposal: \{ N, value \}
Paxos in action: Phase 2a

Proposer: if proposer receives promises from the quorum (majority):
   - Attach a value $v$ to the proposal (the event).
   - Send *Accept* to quorum with the *chosen* value
If promise was for another $\{N, v\}$, proposer MUST accept that

![Diagram of Paxos phases](image)

- **Proposer**
- **Accept** ($N, v$)
- **Quorum**
- **Accept**
- **Client**
- **Learner**
Paxos in action: Phase 2b

**Acceptor**: if the promise still holds, then announce the value $v$

Send **Accepted** message to Proposer and every Learner
else ignore the message (or send **NACK**)
Paxos in action: Phase 3

**Learner**: Respond to client and/or take action on the request

```
Client 

Proposer

Quorum

Acceptor

Announce(N, v)

Learners

Do (N, v)

Server

Server

Server
```

© 2014 Paul Krzyzanowski
Paxos: Keep trying

• A proposal $N$ may fail because
  – The acceptor may have made a new promise to ignore all proposals less than some value $M > N$
  – A proposer does not receive a quorum of responses: either *promise* (phase 1b) or *accept* (phase 2b)

• Algorithm then has to be restarted with a higher proposal #
Paxos summary

• Paxos allows us to ensure consistent (total) ordering over a set of events in a group of machines
  – Events = commands, actions, state updates

• Each machine will have the latest state or a previous version of the state
Paxos summary

To make a change to the system:

– Tell the proposer (leader) the event/command you want to add
  • Note: these requests may occur concurrently
  • Leader = one elected proposer. Not necessary for Paxos algorithm but an optimization to ensure a single, increasing stream of proposal numbers. Cuts down on rejections and retries.

– The proposer picks its next highest event ID and asks all the acceptors to reserve that event ID
  • If any acceptor sees has seen a higher event ID, it rejects the proposal & returns that higher event ID
  • The proposer will have to try again with another event ID

– When the majority of acceptors accept the proposal, accepted events are sent to learners, which can act on them (e.g., update system state)
  • Fault tolerant: need $2k+1$ servers for $k$ fault tolerance
Paxos Commit
What about Paxos?

• Interface to Paxos
  – Client proposes a value and sends it to the Paxos leader
  – Paxos acceptors will send out the totally-ordered value

• What does Paxos consensus offer?
  – Total ordering of proposals
  – Fault tolerance: proposal is accepted if a majority of acceptors accept it
    • There is always enough data available to recover the state of proposals
  – Is provably resilient in asynchronous networks

• Paxos-based commit is a generalization of 2PC and 3PC
Using Paxos for Commit

The cast:
- One instance of Paxos per participant ($N$ participants)
- $2F+1$ acceptors (we can withstand the failure of $F+1$ acceptors)
- One elected Leader (Proposer) = Coordinator

A leader will get at least $F+1$ messages for each instance
Commit iff every participant’s instance of Paxos chooses Prepared
Paxos commit = 2PC if one acceptor
Virtual Synchrony vs. Transactions vs. Paxos

• **Virtual Synchrony**
  – Fastest & most scalable
  – Focuses on group membership management & reliable multicasts

• **Two-Phase & Three-Phase Commit**
  – Most expensive – requires extensive use of stable storage
  – Designed for transactional activities
  – Not suitable for high speed messaging

• **Paxos**
  – Performance limited by its two-phase protocol
  – Great for consensus & fault tolerance
  – Adds ordering of proposals over 2PC
Back to Mutual Exclusion…
Fault-tolerant mutual exclusion
Leasing versus Locking

• Common approach:
  – Get a lock for exclusive access to a resource

• But: locks are not fault-tolerant

• It’s safer to use a lock that expires instead
  – Lease = lock with a time limit
  – Example:
    • three-phase commit vs. two-phase commit
    • Remote objects in .NET or Java

• Trade-off
  – Long leases with possibility of long wait after failure
  – Or short leases that need to be renewed frequently
Hierarchical Leases

• For fault tolerance, leases should be granted by consensus
• But consensus protocols aren’t super-efficient
• Compromise: use a hierarchy
  – Use consensus as an election algorithm to elect a coordinator
  – Coordinator is granted a lease on a large set of resources
    • Coarse-grained locking: large regions; long time periods
  – Coordinator hands out sub-leases on those resources
    • Fine-grained locking: small regions (objects); short time periods
• When the coordinator’s lease expires
  – Consensus algorithm is run again
The End