Distributed Systems

11. Consensus: Paxos

Paul Krzyzanowski
Rutgers University
Fall 2015
Consensus Goal

Allow a group of processes to agree on a result

– All processes must agree on the same value
– The value must be one that was submitted by at least one process (the consensus algorithm cannot just make up a value)
We saw versions of this

• Mutual exclusion
  – Agree on who gets a resource or who becomes a coordinator

• Election algorithms
  – Agree on who is in charge

• Other uses of consensus:
  – Manage group membership
  – Synchronize state to manage replicas: make sure every group member agrees on a (key, value) set
  – Distributed transaction commit

• General consensus problem:
  – How do we get unanimous agreement on a given value?
Achieving consensus seems easy!

Designate a system-wide coordinator to determine outcome

BUT ... this assumes there are no failures
          ... or we are willing to wait indefinitely for recovery
Consensus algorithm goal

• Create a fault-tolerant consensus algorithm that does not block if a majority of processes are working

• Goal: agree on one result among a group of participants
  – Processors may fail (some may need stable storage)
  – Messages may be lost, out of order, or duplicated
  – If delivered, messages are not corrupted
Consensus requirements

• Validity
  – Only proposed values may be selected

• Uniform agreement
  – No two nodes may select different values

• Integrity
  – A node can select only a single value

• Termination (Progress)
  – Every node will eventually decide on a value
Consensus: Paxos
Paxos

• Fault-tolerant distributed consensus algorithm
  – Does not block if a majority of processes are working
  – The algorithm needs a majority \((2P+1)\) of 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 will 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
A Programmer’s View

while (submit_request(R) \neq ACCEPTED) ;

Think of R as a key:value pair in a database
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 need to rely 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

These different roles are usually part of the same system.
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
    • ID = sequence number
  – Asks all the acceptors to reserve that event 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

One of the proposers is chosen to be a leader: it gets all the requests.

Paxos nodes: one machine may serve several roles

Leader

Quorum

Client

October 28, 2015
© 2014-2015 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)

Quorum

Acceptors

Learner
Paxos in action: Phase 1a – PREPARE

**Proposer**: creates a proposal \( #N \) \((N \text{ acts like a Lamport time stamp})\), 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)

\[
N = < \text{seq#} \cdot \text{process_ID} >
\]
**Paxos in action: Phase 1b – PROMISE**

**Acceptor**: if proposer’s ID > any previous proposal
- promise to ignore all requests with IDs < N
- reply with info about highest accepted proposal, if there is one: \{ N, value \}
- else *Reject* the proposal

*This step:*
(a) Tells proposers about any value that has already been accepted.
(b) Rejects older proposals.

<table>
<thead>
<tr>
<th>Promise(N, v)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quorum</td>
</tr>
<tr>
<td>Acceptor</td>
</tr>
<tr>
<td>Learner</td>
</tr>
<tr>
<td>Proposer</td>
</tr>
</tbody>
</table>

Promise to ignore all proposals < N
Promise contains the previous N
**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 instead

If the acceptor returned any \( (N, v) \) sets then the proposer must agree to accept one of those values instead of the value it proposed. It picks the \( v \) for the highest \( N \).
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**)

---

October 28, 2015

© 2014-2015 Paul Krzyzanowski
Paxos in action: Phase 3

**Learner**: Respond to client and/or take action on the request
Paxos: A Simple Example – All Good
Client sends a request to a proposer

Think of the request as an update to a specific \{key, value\} set, such as a field in a database that may be propagated to multiple instances of that database. Our value of “e” here might be something like a request to set name=“e”.
Proposer: picks a sequence number: 5
Send to Quorum of Acceptors

**Propose**($5$:“e”)
Acceptor: Suppose 5 is the highest sequence # any acceptor has seen.
Each acceptor PROMISES not to accept any lower numbers.
**Proposer:** Proposer receives the promise from a majority of acceptors. Proposer must accept that $<$seq, value$>$.
Paxos in action: Phase 2b – **ANNOUNCE**

**Acceptors**: Acceptors state that they accepted the request

- **Proposer**
- **Acceptors**
- **Learners**
- **Client**

**Announce(5,"e")**

**Accepted(5,"e")**
Paxos: A Simple Example – Higher Offer
Paxos in action: Phase 0

Client sends a request to a proposer

Client

Proposer

Request(“e”)

Learner

Quorum

Acceptor

Acceptor

Acceptor
Paxos in action: Phase 1a – PREPARED

**Proposer:** picks a sequence number: 5
Send to Quorum of Acceptors

One acceptor receives a higher offer
BEFORE it gets this PREPARE message

Proposer: picks a sequence number: 5

Send to Quorum of Acceptors

**Prepare(5:“e”)**

**Prepare(7:“g”)**
Acceptor: Suppose 5 is the highest sequence # any acceptor has seen
Each acceptor PROMISES not to accept any lower numbers

Proposer

Learner

Client

Quorum

Promise(5:“e”)

Promise(7:“g”)

Promise(5:“e”)

Acceptors
Proposer: Proposer receives the higher # offer and **MUST** change its mind and accept the highest offer that it received from any acceptor.
**Acceptor**: Acceptors state that they accepted the request. A learner can propagate this information.

Diagram:
- Client
- Proposer
- Learners
- Acceptors

Messages:
- `Announce(7, "g")`
- `Accepted(7, "g")`

Date: October 28, 2015

© 2014-2015 Paul Krzyzanowski
• 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 used in:
  – Cassandra lightweight transactions
  – Google Chubby lock manager / name server
  – Google Spanner, Megastore
  – Microsoft Autopilot cluster management service from Bing
  – VMware NSX Controller
  – Amazon Web Services
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
The End