Paxos

- **Always safe**
- Live during periods of synchrony
- Leader (primary) responsible for proposing the consensus value
- Features Dijkstra as a cheese inspector

Somewhat popular...
- Part-time Parliament [L98]
- Frangipani [TML97]
- Byzantine Paxos [CL99]
- Disk Paxos [GL00]
- Reconstructing Paxos [BDFG01]
- Active Disk Paxos [CM02]
- Separating Agreement & Execution [YMAD 03]
- Byzantine Disk Paxos [ACKM04]
- Fast Byzantine Paxos [MA05]
- Fast Paxos [LO5]
- Hybrid Quorums [CMLRS06]
- Chubby [BO06]
- Paxos Register [LCAA07]
- Zyzzyva [KADCW07]

The Game of Paxos

Processes are competing to write a value in a write-once register

To learn the final value:
1. Push the read button and examine the token that falls into the tray
2. If the token is green, GAME OVER - the final value of the register is stamped on the token!
3. If the token is red and stamped with a value, set the dial to the same value, and push the write button
4. If the token is red and not stamped, set the dial to any value, and push the write button

Quorum Systems

Given a set of servers \( U, |U| = n \)

A quorum system is a set \( Q \subseteq 2^U \) such that
\[ \forall Q_1, Q_2 \in Q : Q_1 \cap Q_2 \neq \emptyset \]

Each \( Q \) in \( Q \) is a quorum
A R/W Register

store at each server a \((v, ts)\) pair

Write\((x, d)\)
- Ask servers in some \(Q\) for their \(ts\)
- Set \(ts_c > \max\{\{ts\} \cup \text{any previous } ts_c\}\)
- Update some \(Q'\) with \((d, ts_c)\)

Read\((x)\)
- Ask servers in some \(Q\) for their \((v, ts)\)
- Select most recent \((v, ts)\)
- Update some \(Q'\) with \((d, ts_c)\)

PBFT:
A Byzantine Renaissance

- Practical Byzantine Fault-Tolerance (CL99, CL00)
- first to be safe in asynchronous systems
- live under weak synchrony assumptions −Byzantine Paxos
- fast! PBFT uses MACs instead of public key cryptography
- uses proactive recovery to tolerate more failures over system lifetime; now need no more than \(f\) failures in a “window”

BASE (RCL 01)
- uses abstraction to reduce correlated faults
The Setup

- **System Model**
  - Asynchronous system
  - Unreliable channels

- **Crypto**
  - Public/Private key pairs
  - MACs
  - Collision-resistant hashes
  - Unbreakable

- **Service**
  - Byzantine clients
  - Up to $f$ Byzantine servers
  - $N > 3f$ total servers

- **System Goals**
  - Always safe
  - Live during periods of synchrony

The General Idea

- **Primary-backup + quorum system**
  - Executions are sequences of views
  - Clients send signed commands to primary of current view
  - Primary assigns sequence number to client's command
  - Primary writes sequence number to the register implemented by the quorum system defined by all the servers (primary included)

What could possibly go wrong?

- **The Primary could be faulty!**
  - Could ignore commands; assign same sequence number to different requests; skip sequence numbers; etc
  - Backups monitor primary's behavior and trigger view changes to replace faulty primary

- **Backups could be faulty!**
  - Could incorrectly store commands forwarded by a correct primary
    - Use dissemination Byzantine quorum systems [MR98]

- **Faulty replicas could incorrectly respond to the client!**
  - Client waits for $f+1$ matching replies before accepting response
What could possibly go wrong? 😊

- The Primary could be faulty!
  - could ignore commands, assign same sequence number to different requests, skip sequence numbers, etc.
  - Backups monitor primary’s behavior and trigger view changes to replace faulty primary.

- Backups could be faulty!
  - could incorrectly store commands forwarded by a correct primary.
  - use dissemination Byzantine quorum systems [MR98]

- Faulty replicas could incorrectly respond to the client!
  - Client waits for $f+1$ matching replies before accepting response.

- Carla Bruni could start singing!

Me, or your lying eyes?

- Algorithm steps are justified by certificates
  - Sets (quorums) of signed messages from distinct replicas proving that a property of interest holds.

- With quorums of size at least $2f+1$
  - Any two quorums intersect in at least one correct replica.
  - Always one quorum contains only non-faulty replicas.

PBFT: The site map

- Normal operation
  - How the protocol works in the absence of failures – hopefully, the common case.

- View changes
  - How to depose a faulty primary and elect a new one.

- Garbage collection
  - How to reclaim the storage used to keep certificates.

- Recovery
  - How to make a faulty replica behave correctly again.

Normal Operation

- Three phases:
  - Pre-prepare assigns sequence number to request.
  - Prepare ensures fault-tolerant consistent ordering of requests within views.
  - Commit ensures fault-tolerant consistent ordering of requests across views.

- Each replica $i$ maintains the following state:
  - Service state
  - A message log with all messages sent or received.
  - An integer representing $i$’s current view.
Client issues request

<REQUEST<state_machine_operation>><timestamp><client_id>

Primary
Backup 1
Backup 2
Backup 3
Client issues request

\[ \text{Primary multicasts } \langle \text{PRE-PREPARE}, v, u, d, \sigma_p, m \rangle \]

Pre-prepare

\[ \text{View} \]

Primary multicasts \( \langle \text{PRE-PREPARE}, v, u, d, \sigma_p, m \rangle \)
Pre-prepare

Primary multicasts \(<\text{PRE-PREPARE}, v, n, d > \sigma_p, m>\)

Correct backup $i$ accepts PRE-PREPARE if:

1. PRE-PREPARE is well formed
2. $i$ is in view $v$
3. $i$ has not accepted another PRE-PREPARE for $v, n$ with a different $d$
4. $n$ is between two water-marks $l$ and $h$
   (to prevent sequence number exhaustion)

Each accepted PRE-PREPARE message is stored in the accepting replica's message log (including the Primary's).
Correct replica \( i \) accepts \texttt{PREPARE} if:

- \texttt{PREPARE} is well formed
- \( i \) is in view \( v \)
- \( v \) is between two water-marks \( L \) and \( H \)

- Replicas that send \texttt{PREPARE} accept seq.# \( n \) for \( m \) in view \( v \)
- Each accepted \texttt{PREPARE} message is stored in the accepting replica's message log

- \( P \)-certificates ensure total order within views

- Replica produces \( P \)-certificate(\( m,v,n \)) iff its log holds:
  - The request \( m \)
  - A \texttt{PRE-PREPARE} for \( m \) in view \( v \) with sequence number \( n \)
  - 2f \texttt{PREPARE} from different backups that match the pre-prepare
**Prepare Certificate**

- **P-certificates** ensure total order within views
- Replica produces P-certificate \((m,v,n)\) iff its log holds:
  - The request \(m\)
  - A PRE-PREPARE for \(m\) in view \(v\) with sequence number \(n\)
  - \(2f\) PREPARE from different backups that match the pre-prepare
- A P-certificate \((m,v,n)\) means that a quorum agrees with assigning sequence number \(n\) to \(m\) in view \(v\)
- NO two non-faulty replicas with P-certificate \((m_1,v,n)\)
  and P-certificate \((m_2,v,n)\)

**P-certificates are not enough**

- A P-certificate proves that a majority of correct replicas has agreed on a sequence number for a client's request
- Yet that order could be modified by a new leader elected in a view change

---

**Commit**

- After collecting a P-certificate, replica \(i\) multicasts \(<\text{COMMIT},v,n,d,i>\) to replicas
- **Commit Certificate**
  - **C-certificates** ensure total order across views
    - can't miss P-certificate during a view change
  - A replica has a C-certificate \((m,v,n)\) if:
    - it had a P-certificate \((m,v,n)\)
    - log contains \(2f+1\) matching \(\text{COMMIT}\) from different replicas (including itself)
  - Replica executes a request after it gets C-certificate for it, and has cleared all requests with smaller sequence numbers
After executing request, replica \( i \) replies with
\[
\text{<REPLY, v, t, c, i, r\_\sigma^i>}
\]

A disgruntled backup mutinies:
- stops accepting messages (but for VIEW-CHANGE & NEW-VIEW)
- multicasts \(<\text{VIEW-CHANGE} v+1, \mathcal{P} \_\sigma^i>\)
- \( \mathcal{P} \) contains all P-Certificates known to replica \( i \)

A backup joins mutiny after seeing \( f+1 \) distinct VIEW-CHANGE messages

Mutiny succeeds if new primary collects a new-view certificate \( \mathcal{V} \), indicating support from \( 2f+1 \) distinct replicas (including itself)

On to view \( v+1 \): the new primary

The “primary elect” \( \hat{p} \) (replica \( v+1 \mod N \)) extracts from the new-view certificate \( \mathcal{V} \):
- the highest sequence number \( h \) of any message for which \( \mathcal{V} \) contains a P-certificate

On to view \( v+1 \): the new primary

The “primary elect” \( \hat{p} \) (replica \( v+1 \mod N \)) extracts from the new-view certificate \( \mathcal{V} \):
- the highest sequence number \( h \) of any message for which \( \mathcal{V} \) contains a P-certificate
- two sets \( \mathcal{O} \) and \( \mathcal{N} \):
  - If there is a P-certificate for \( n,m \) in \( \mathcal{V} \), \( n \leq h \)
    \[ \mathcal{O} = \mathcal{O} \cup <\text{PRE-PREPARE}, v+1, n, m \_\sigma^p> \]
  - Otherwise, if \( n < h \) but no P-certificate:
    \[ \mathcal{N} = \mathcal{N} \cup <\text{PRE-PREPARE}, v+1, n, \text{null} \_\sigma^p> \]
On to view $v+1$: the new primary

- The “primary elect” $\hat{p}$ (replica $v+1 \mod N$) extracts from the new-view certificate $V$:
  - the highest sequence number $h$ of any message for which $V$ contains a $P$-certificate
  - two sets $O$ and $N$:
    - If there is a $P$-certificate for $n,m$ in $V$, $n \leq h$
      $$O = O \cup \text{PRE-PREPARE},v+1,n,m_{\sigma_p}$$
    - Otherwise, if $n \leq h$ but no $P$-certificate:
      $$N = N \cup \text{PRE-PREPARE},v+1,n,\text{null}_{\sigma_p}$$
- $\hat{p}$ multicasts $\text{NEW-VIEW},v+1,V,O,N_{\sigma_p}$

On to view $v+1$: the backup

- Backup accepts $\text{NEW-VIEW}$ message for $v+1$ if
  - it is signed properly
  - it contains in $V$ a valid $\text{VIEW-CHANGE}$ messages for $v+1$
  - it can verify locally that $O$ is correct (repeating the primary’s computation)
- Adds all entries in $O$ to its log (so did $\hat{p}$!)
- Multicasts a $\text{PREPARE}$ for each message in $O$
- Adds all $\text{PREPARE}$ to log and enters new view

Garbage Collection

- For safety, a correct replica keeps in log messages about request $o$ until it
  - $o$ has been executed by a majority of correct replicas, and
  - this fact can proven during a view change
- Truncate log with Certificate
  - Each replica $i$ periodically (after processing $k$ requests) checkpoints state and multicasts $\text{CHECKPOINT},n,d,i$

Garbage Collection

- For safety, a correct replica keeps in log messages about request $o$ until it
  - $o$ has been executed by a majority of correct replicas, and
  - this fact can proven during a view change
- Truncate log with Certificate
  - Each replica $i$ periodically (after processing $k$ requests) checkpoints state and multicasts $\text{CHECKPOINT},n,d,i$
  - last executed request reflected in state
Garbage Collection

For safety, a correct replica keeps in log messages about request \( o \) until it
- \( o \) has been executed by a majority of correct replicas, and
- this fact can be proven during a view change.

Truncate log with Certificate
- Each replica \( i \) periodically (after processing \( k \) requests) checkpoints state and multicasts
  \( \text{CHECKPOINT} \langle n, d, i \rangle \)
  state's digest

Truncate log with Stable Certificate
- Each replica \( i \) periodically (after processing \( k \) requests) checkpoints state and multicasts
  \( \text{CHECKPOINT} \langle n, d, i \rangle \)
- \( 2f+1 \) CHECKPOINT messages are a proof of the checkpoint's correctness

View change, revisited

A disgruntled backup multicasts
- \( \text{VIEW-CHANGE} \langle v+1, n, s, C, P, i \rangle \sigma \),
View change, revisited

A disgruntled backup multicasts

\(<\text{VIEW-CHANGE}, v+1, n, s, C, P, i >_{\pi}, \rangle \)

last stable checkpoint

stable certificate for 

P certifies for requests with sequence number > n

\hat{p} multicasts

\(<\text{NEW-VIEW}, v + 1, n, V, \hat{O}, \hat{N}, >_{\hat{\pi}}, \rangle \)

sequence number of last stable checkpoint
Citius, Altius, Fortius: Towards deployable BFT

- Reducing the costs of BFT replication
- Addressing confidentiality
- Reducing complexity

Reducing the costs of BFT replication

- Who cares? Machines are cheap...
  - Replicas should fail independently in software, not just hardware
  - How many independently failing implementations of non-trivial services do actually exist?

Back the old conundrum

A: voter and client share fate!

Not so fast...
Rethinking State Machine Replication

Not Agreement + Order
but rather Agreement on Order + Execution

Benefits:
- 3f+1 state machine replicas
Rethinking State Machine Replication

Not Agreement + Order
but rather Agreement on Order + Execution

Benefits:
- 2f+1 state machine replicas
- Separation reduces replication costs

Nodes in E: expensive
(different across applications and within same application)
Nodes in A: cheap
(simple and reusable across applications)

Separation enables confidentiality

Three design principles:
1. Use redundant filters for fault tolerance
2. Restrict communication
3. Eliminate nondeterminism
The Privacy Firewall

- \((h+1)^2\)-filter grid tolerates h Byzantine failures
- A filter only communicates with filters immediately above or below
- Each filter checks both reply and request certificates
- Safe
  - \(h+1\) rows \(\rightarrow\) one is correct
- Live
  - \(h+1\) columns \(\rightarrow\) one is correct
- Restricts nondeterminism
  - threshold cryptography for replies
  - cluster A locks rsn
  - controlled message retransmission

Inside the PF

- \((h+1)^2\)-filter grid tolerates h Byzantine failures
- A filter only communicates with filters immediately above or below
- Each filter checks both reply and request certificates
- Safe
  - \(h+1\) rows \(\rightarrow\) one is correct
- Live
  - \(h+1\) columns \(\rightarrow\) one is correct
- Restricts nondeterminism
  - threshold cryptography for replies
  - cluster A locks rsn
  - controlled message retransmission
Inside the PF

- \((h+1)^2\)-filter grid tolerates \(h\) Byzantine failures
- A filter only communicates with filters immediately above or below
- Each filter checks both reply and request certificates
- Safe
  - \(h+1\) rows \(\rightarrow\) one is correct
- Live
  - \(h+1\) columns \(\rightarrow\) one is correct
- **Restricts nondeterminism**
  - threshold cryptography for replies
  - cluster \(A\) locks rsn
  - controlled message retransmission

Privacy Firewall guarantees

Output-set confidentiality

Output sequence through correct cut is a legal sequence of outputs produced by a correct node accessed through an asynchronous, unreliable link.