EECS 591
Distributed Systems

Manos Kapritsos
Winter 2018
PBFT: A Byzantine Renaissance

Practical Byzantine Fault Tolerance
(Castro, Liskov 1999-2000)

• First practical protocol for asynchronous BFT replication

• Like Paxos, PBFT is safe all the time, and live during periods of synchrony
The setup

System model
- Asynchronous system
- Unreliable channels

Crypto
- Public/private key pairs
- Signatures
- Collision-resistant hashes

Service
- Byzantine clients
- Up to $f$ Byzantine servers
- $n = 3f + 1$ total servers

System goals
- Always safe
- Live during periods of synchrony
The general idea

![Diagram showing a user A interacting with a primary node and three replicas. Each replica is connected to the primary with an arrow, indicating a relationship or process flow.]
PBFT: Normal Operation

Three phases:

- **Pre-prepare** assigns sequence number to request
- **Prepare** ensures consistent ordering of requests within views
- **Commit** ensures consistent ordering of requests across views

Each replica maintains the following state:

- Service state
- A **message log** with all messages sent or received
- An integer representing the replica's current view
Primary sends \( \langle \langle \text{PRE-PREPARE}, v, n, d \rangle_{\sigma_p}, m \rangle \) to all replicas
Replica \( k \) sends \(<\text{PREPARE, } v, n, d, k>\sigma_k\) to all replicas

- Replicas that send a PREPARE accept the assignment of \( m \) to sequence number \( n \) in view \( v \)
- Each accepted PREPARE message is stored in the accepting replica’s message log
P-Certificates ensure consistent order of requests within views

A replica produces a P-Certificate($m,v,n$) iff its log holds:
- the request $m$
- A PRE-PRE-PREPARE for $m$ in view $v$ with sequence number $n$
- $2f$ PREPARE from distinct backups that match the PRE-PRE-PREPARE

A P-Certificate($m,v,n$) means that a quorum agrees to assign $m$ to sequence number $n$ in view $v$
- No two non-faulty replicas with P-Certificate($m,v,n$) and P-Certificate($m',v,n$)
P-CERTIFICATES ARE NOT ENOUGH

• A P-Certificate proves that a quorum of \(2f + 1\) replicas has agreed to assign \(m\) to sequence number \(n\) in view \(v\).

• Yet that assignment could be modified if a **view change** happens (the primary changes)
  • The new primary may not be convinced to assign \(m\) to \(n\) in the new view \(v'\).
**P-Certificates are Not Enough**

- Yet that assignment could be modified if a **view change** happens (the primary changes)
  - The new primary may not be convinced to assign \( m \) to \( n \) in the new view \( v' \)
  - \( 2f + 1 \) prepares means at least \( f + 1 \) correct replicas received a pre-prepare for \( (m,v,n) \)
After collecting a P-Certificate, replica k sends $\langle\text{COMMIT}, v, n, d, k\rangle_{\sigma_k}$ to all replicas.
Commit Certificate

- C-Certificates ensure consistent order of requests across views
  - Cannot miss a P-Certificate during view change

- A replica has a C-Certificate \((m,v,n)\) iff:
  - it had a P-Certificate \((m,v,n)\)
  - its log contains \(2f + 1\) matching COMMIT messages from distinct replicas (including itself)

- A replica executes a request when:
  - it gets a C-Certificate for it
  - it has executed all requests with smaller sequence numbers
After executing a request, replica k replies to the client with \(<\text{REPLY}, v, t, c, k, r>\) \(\sigma_k\).
TO ARMS, REPLICAS!!

- A disgruntled replica mutinies:
  - Stops accepting messages (except for VIEW-CHANGE and NEW-VIEW messages)
  - Sends $<\text{VIEW-CHANGE}, v+1, P>_{\sigma_k}$
  - $P$ contains all P-Certificates known to replica $k$
- A replica joins mutiny after seeing $f + 1$ distinct VIEW-CHANGE messages
- Mutiny succeeds if the 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" $p'$ (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.

  ![Diagram showing sequence numbers and P-Certificates]

  $h = 12$

- Two sets $\mathcal{O}$ and $\mathcal{N}$:
  - if there is a P-certificate for $n, m$ in $V$, where $n \leq h$
    add $<\text{PRE-PREPARE}, v+1, n, m>_{\sigma_p}$ to $\mathcal{O}$
  - otherwise, if $n \leq h$ but there is no P-Certificate
    add $<\text{PRE-PREPARE}, v+1, n, \text{null}>_{\sigma_p}$ to $\mathcal{N}$

- $p'$ sends $<\text{NEW-VIEW}, v+1, V, \mathcal{O}, \mathcal{N}>_{\sigma_p}$ to all replicas
On to view v+1: The Replica

- A replica accepts a NEW-VIEW message for v+1 if
  - it is signed properly
  - it contains in V valid 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 (as did p’)

- Sends a PREPARE to all replicas for each message in O

- Adds all PREPARE messages to its log and enters new view
BFT: A PERSPECTIVE

On the other hand:
Google is implementing BFT as we speak
Homework #1 graded
- Ask us if you have questions about the solutions
- Regrade requests in writing

Research part
- Starts on Wednesday: Fast Paxos, Flexible Paxos
- Submit reviews through HotCRP account

Midterm
- Postponed until one week after the Spring Break
Eve: Replicating Multithreaded Servers

Kapritsos, Wang, Quema, Clement, Alvisi, Dahlin
The Achilles’ heel of replication

Birth of most dependability techniques

Challenge: scale to multithreaded execution
How do we build dependable multithreaded services?

Answer: State Machine Replication
**STATE MACHINE REPLICA**

**Ingredients:** a server

1. Make server deterministic (state machine)
2. Replicate server
3. Provide all replicas with the same input

**Guarantee:** correct replicas will produce the same output
**State Machine Replication**

**Ingredients:** a server

1. Make server deterministic (state machine)
2. Replicate server
3. Provide all replicas with the same input

Guarantee: correct replicas will produce the same output
How do we build dependable multithreaded services?

Maybe use deterministic multithreading?

Nope. Won’t support modern replication protocols
How do we build dependable multithreaded services?
How do we build dependable multithreaded services?
Eve (OSDI ’12)

Scaling replication to multithreaded execution
SMR requires replica convergence

Execute
SMR requires replica convergence

Agree-Execute enforces sequential execution
First execute...  
(multithreaded and without agreeing on the order)  

...then verify  
(that replicas agree on the outcome)
ON CONVERGENCE

Server → Commit

Server → Commit

Server → Commit

Verify

match?

YES

YES

YES
ON DIVERGENCE

Repair: rollback and re-execute sequentially
if (converged)
  commit
else
  repair divergence

1. Make divergence uncommon

2. Detect divergence efficiently

3. Repair divergence efficiently

Eve’s logic at a glance
Making divergence uncommon

if (converged)
   commit
else
   repair divergence

Idea: identify commutative requests

Mixer: group together commutative requests
   • Execute requests within a group in parallel

Mixer is a hint, not an oracle
Making divergence uncommon

if (converged)
  commit
else
  repair divergence

Idea: identify commutative requests

Mixer: group together commutative requests
  • Execute requests within a group in parallel

Mixer is a hint, not an oracle
## Example: TPC-W Mixer

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Read tables</th>
<th>Write tables</th>
</tr>
</thead>
<tbody>
<tr>
<td>getBestSellers</td>
<td>item, author, order_line</td>
<td></td>
</tr>
<tr>
<td>doCart</td>
<td>item</td>
<td>shopping_cart_line, shopping_cart</td>
</tr>
<tr>
<td>doBuyConfirm</td>
<td>customer, address</td>
<td>order_line, item, cc_xacts, shopping_cart_line</td>
</tr>
</tbody>
</table>

3 frequent transactions of the TPC-W browsing workload
Efficient divergence detection

Need to compare application states & responses frequently

if (converged)
  commit
else
  repair divergence

Merkle tree

Application state
Efficient divergence repair

Need to rollback application states after every divergence

if (converged)
  commit
else
  repair divergence

Application state

Rollback

Copy-on-write
if (converged)
  commit
else
  repair divergence

1. Make divergence uncommon
   Mixer

2. Detect divergence efficiently
   Merkle tree

3. Repair divergence efficiently
   Copy-on-Write
Dependability ❤️ Performance

- Independent execution
- Replication of multithreaded services
- Non-deterministic order of requests
- Bonus: mask concurrency bugs
MASKING CONCURRENCY BUGS

Server

Server

Server

Verify

token

token

token
EXECUTE-VERIFY: AN ARCHITECTURAL CHANGE

Synchronous

Arbitrary failures

Crash failures

Asynchronous
**Configurations**

**Asynchronous BFT**

- **Execution**
  - 1 arbitrary fault tolerated

- **Verification**
  - 1 arbitrary fault tolerated

**Synchronous primary-backup**

- **Primary**
  - 1 omission fault tolerated

- **Backup**
  - 1 omission fault tolerated