EECS 591
Distributed Systems

Manos Kapritsos
Fall 2018
THE GENERAL IDEA

A sends data to the primary node, which then replicates the data to multiple replicas.
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
THE FIRST TWO PHASES OF PBFT

Primary

Replica 1

Replica 2

Replica 3

Pre-prepare  Prepare
**P-Prepare Certificate**

- P-Certificate: a collection of $2f + 1$ prepares

- P-Certificates ensure consistent order of requests within views

- 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 \( <\text{COMMIT}, v, n, d, k>_{\sigma_k} \) to all replicas.
**Commit Certificate**

- **C-Certificate**: a collection of $2f + 1$ commits

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

- 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 $V$, 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.

<table>
<thead>
<tr>
<th>P</th>
<th>P</th>
<th>P</th>
<th>P</th>
<th>P</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>h=12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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 $\mathcal{V}$ valid VIEW-CHANGE messages for $v+1$
  - it can verify locally that $\mathcal{O}$ is correct (repeating the primary’s computation)

- Adds all entries in $\mathcal{O}$ to its log (as did $p'$)

- Sends a PREPARE to all replicas for each message in $\mathcal{O}$

- Adds all PREPARE messages to its log and enters new view
On the other hand: Google is implementing BFT as we speak.
ADMINISTRIVIA

Research part
- Starts on Wednesday with Fast Paxos
- Submit reviews through HotCRP account

Midterm
- Wednesday 10/31, 3-5pm, in class
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 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
SMR IMPLEMENTATION

Agree

Server

Server

Server
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?

Dependability and Performance
Eve (OSDI ’12)

Scaling replication to multithreaded execution
SMR requires replica convergence

Agree-Execute enforces sequential execution
**EXECUTE-VERIFY**

First execute...
(multithreaded and without agreeing on the order)

...then verify
(that replicas agree on the outcome)
On convergence

Verify

match?
Repair: rollback and re-execute sequentially
Eve’s logic at a glance

if (converged)
commit
else
repair divergence

1. Make divergence uncommon

2. Detect divergence efficiently

3. Repair divergence efficiently
MAKING DIVERGENCE UNCOMMON

if (converged)
  commit
else
  repair divergence
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
### 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

```plaintext
if (converged)
    commit
else
    repair divergence
```

Application state

Merkle tree
Efficient divergence repair

Need to rollback application states after every divergence

Application state

Copy-on-write

if (converged)
    commit
else
    repair divergence
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 ❤️ Non-deterministic order of requests

Replication of multithreaded services

Bonus: mask concurrency bugs
Masking concurrency bugs
EXECUTE-VERIFY: AN ARCHITECTURAL CHANGE

Synchronous

Asynchronous

Arbitrary failures

Crash failures
Configurations

Asynchronous BFT
- Execution:
  - Tolerates 1 arbitrary fault
- Verification:
  - Tolerates 1 arbitrary fault

Synchronous primary-backup
- Primary:
  - Tolerates 1 omission fault
- Backup:
  - Tolerates 1 omission fault
EVALUATION

What is the performance benefit of Eve compared to traditional SMR systems?
Application: H2 Database Engine
Workload: TPC-W (browsing)