EECS 591
Distributed Systems

Manos Kapritsos
Fall 2019
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 $<$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

\[ \begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
\hline
\end{array} \]

- two sets O and N:
  - if there is a P-certificate for n, m in V, where n \leq h
    add <PRE-PREPARE, v+1, n, m>\textsubscript{σp’} to O
  - otherwise, if n \leq h but there is no P-Certificate
    add <PRE-PREPARE, v+1, n, null>\textsubscript{σp’} to N

- p’ sends <NEW-VIEW, v+1, V, O, N>\textsubscript{σp’} 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
On the other hand:

Google has used BFT in its datacenters and so do many blockchain approaches.

Figure 1: Typical Figure 2 from Byzantine fault paper: Our network protocol

BFT: A PERSPECTIVE

[Mickens 2013]
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
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

Agree

Execute

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?
On divergence

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

Frequent

if (converged)
commit
else
repair divergence

Uncommon

1. Make divergence uncommon

2. Detect divergence efficiently

3. Repair divergence efficiently
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

Application state

Merkle tree
**Efficient divergence repair**

Need to rollback application states after every divergence

```python
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
Masking concurrency bugs
EXECUTE-VERIFY: AN ARCHITECTURAL CHANGE

- Synchronous
- Asynchronous

Arbitrary failures
Crash failures
Configurations

Asynchronous BFT

Execution

Verification

Tolerates 1 arbitrary fault

Synchronous primary-backup

Primary

Backup

Tolerates 1 omission fault
TWO GENERALS’ PROBLEM

Both generals must attack together or face defeat

Communication is only by messengers sneaking through the valley

Messengers may not make it through…
Question 1 (true or false)

a. $e \rightarrow d$

b. $a \rightarrow j$

c. $g \rightarrow b$
Lamport clocks

\[ p \rightarrow q \Rightarrow LC(p) < LC(q) \]

the Clock condition
Vector clocks

\[ VC(e_i)[j] = \text{number of events executed by process j that causally precede } e_i \]

\[ p \rightarrow q \iff \theta(p) \subset \theta(q) \]

Strong clock condition
VECTOR CLOCKS

$$VC(e_i)[j] = \text{number of events executed by process j that causally precede } e_i$$

Question 2: what is the VC of:

a. event $d$

b. event $g$
Cristian’s algorithm

\[ t = x \]

\[ t = x \]

\[ \alpha, \beta \geq 0 \]

\[ P(t) \]

\[ Q(t) \]

\[ \text{"time=?"} \]

\[ \text{"time=} T \text{"} \]

\[ Q(x) = ? \]
2-Phase Commit

Coordinator $c$:
1. sends VOTE-REQ to all participants

Participant $p_i$:
2. sends $vote_i$ to Coordinator
   if $vote_i = \text{No}$ then
     $decision_i := \text{Abort}$
     halt
3. if (all votes are Yes) then
   $decision_c := \text{Commit}$
   send Commit to all
   else
   $decision_c := \text{Abort}$
   send Abort to all who voted Yes
4. if received Commit then
   $decision_i := \text{Commit}$
   else
   $decision_i := \text{Abort}$
   halt
3-Phase Commit

Coordinator $c$

1. sends VOTE-REQ to all participants

2. sends $\text{vote}_i$ to Coordinator
   - if $\text{vote}_i = \text{No}$ then $\text{decision}_i := \text{Abort}$
   - halt

Participant $p_i$

3. if (all votes are Yes) then
   - send Precommit to all

   - else
     - $\text{decision}_c := \text{Abort}$
     - send Abort to all who voted Yes
     - halt

4. if received Precommit then
   - send Ack

5. collect Ack from all participants
   - When all Ack’s have been received:
     - $\text{decision}_c := \text{Commit}$
     - send Commit to all

6. When $p_i$ receives Commit, sets $\text{decision}_i := \text{Commit}$ and halts
3PC: Which states are compatible?

**Question 3:**

<table>
<thead>
<tr>
<th></th>
<th>Aborted</th>
<th>Uncertain</th>
<th>Committable</th>
<th>Committed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aborted</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncertain</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Committable</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Committed</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A Hierarchy of Failure Models

- Byzantine
- Omission failures
- Crash
- Commission failures
Ingredients: a server

1. Make server deterministic (state machine)
2. Replicate server
3. Ensure that all replicas go through the same sequence of state transitions
4. Vote on replica outputs
A primary-backup protocol

(request) reply

new primary

$\tau$

sync

$\text{ack}$
**Chain replication**

Tail can respond immediately, without waiting for the new update.
**Consensus**

**Validity**
If all processes that propose a value propose \( v \), then all correct processes eventually decide \( v \)

**Agreement**
If a correct process decides \( v \), then all correct processes eventually decide \( v \)

**Integrity**
Every correct process decides at most one value, and if it decides \( v \), then some process must have proposed \( v \)

**Termination**
Every correct process eventually decides some value
Our algorithm implementing consensus in a synchronous setting is correct! That is, it is both safe and live.
BAD NEWS

The FLP result:
There is no protocol that solves consensus in an asynchronous system where one process may crash

Fischer, Lynch, Paterson 1985
Abstract

The Paxos algorithm, when presented in plain English, is very simple.
### ACCEPTOR STATES
(as leader #50 comes to power)

<table>
<thead>
<tr>
<th>Acceptors</th>
<th>Value</th>
<th>By leader</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( x )</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>( y )</td>
<td>41</td>
</tr>
</tbody>
</table>

**Question 4:**
What is the set of possible values that leader #50 can propose?
THE THREAT TO LIVENESS: DUELING PROPOSERS

Greetings, peasants! I am your fearless leader #1! Grant me your blessing!

Greetings, peasants! I am your fearless leader #3! Grant me your blessing!

Greetings, peasants! I am your fearless leader #5! Grant me your blessing!

Greetings, peasants! I am your fearless leader #7! Grant me your blessing!

Greetings, peasants! I am your fearless leader #2! Grant me your blessing!

Greetings, peasants! I am your fearless leader #4! Grant me your blessing!

Greetings, peasants! I am your fearless leader #6! Grant me your blessing!

Greetings, peasants! I am your fearless leader #8! Grant me your blessing!
Paxos/SMR in real life

Proposers, acceptors and learners are all collocated on $2f + 1$ replicas.
PBFT

- Primary
- Replica 1
- Replica 2
- Replica 3

Phases:
- Pre-prepare phase
- Prepare phase
- Commit phase
- Reply phase
**EXECUTE-VERIFY**

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

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