Project #1 is out

Due Friday, March 9th, 23:59:59pm

Submit by email to me

Maximum team size: 2
After executing a request, replica $k$ replies to the client with $\sigma_k <\text{REPLY}, v, t, c, k, r>$.
TO ARMS, REPLICAUX!!
(a.k.a the view change)

- A disgruntled replica mutinies:
  - Stops accepting messages (except for VIEW-CHANGE and NEW-VIEW messages)
  - Sends $\langle$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

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

• two sets $O$ and $N$:
  ▶ 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 $O$
  ▶ otherwise, if $n \leq h$ but there is no P-Certificate
    add $<\text{PRE-PREPARE}, v+1, n, \text{null}>_{\sigma_{p'}}$ to $N$

• $p'$ sends $<\text{NEW-VIEW}, v+1, V, O, N>_{\sigma_{p'}}$ 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
Eve: replicating multithreaded servers

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

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

Answer:
State Machine Replication
STATE MACHINE REPLICACTION

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

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?
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?
ON CONVERGENCE

Verify

Server

Server

Server

YES

YES

YES
On convergence

Server → Commit

Server → Commit

Server → Commit

Verify
ON DIVERGENCE

Verify

Server

1
2
3
token

Server

1
2
3
token

Server

1
2
3
token
ON DIVERGENCE

Verify

Server

Server

Server
ON DIVERGENCE

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

1. Make divergence uncommon
2. Detect divergence efficiently
3. Repair divergence efficiently

if (converged)
    commit
else
    repair divergence
Making divergence uncommon

if (converged)
commit
else
repair divergence
**Making divergence uncommon**

```java
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
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
**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

if (converged)
   commit
else
   repair divergence

Application state

Merkle tree
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

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

- **Arbitrary failures**
- **Crash failures**

<table>
<thead>
<tr>
<th>Synchronous</th>
<th>Asynchronous</th>
</tr>
</thead>
</table>
**Configurations**

**Asynchronous BFT**
- Execution
- Verification

- Tolerates 1 arbitrary fault

**Synchronous primary-backup**
- Primary
- 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)
Review of Part One: Fundamentals
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…
Ordering events without physical clocks

Question 1 (true or false)

a. $e \rightarrow d$  true
b. $a \rightarrow j$  true
c. $g \rightarrow b$  false
Lamport clocks

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

the Clock condition
**VECTOR CLOCKS**

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

\[
[1,0,0] \\
a \rightarrow b \rightarrow c \rightarrow d
\]

\[
[0,0,1] \\
h \rightarrow i \rightarrow j
\]

\[
p \rightarrow q \iff \theta(p) \subset \theta(q) \\
\text{Strong clock condition}
\]
**Vector clocks**

\[ VC(e_i)[i] = \text{number of events executed by process } i \text{ (including } e_i) \]

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

**Question 2:** what is the VC of:

a. event \( d \)  \( \text{Answer: } [4,2,0] \)

b. event \( g \)  \( \text{Answer: } [2,3,1] \)
Cristian's algorithm

\[ t = x \]

\[ \text{slave} \quad P(t) \quad \text{master} \quad Q(t) \]

\[ \text{"time=?"} \quad 2D \quad \text{"time=}\, T\text{"} \]

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

\[ \min + \alpha \quad 2d \quad \min + \beta \]

\[ \frac{2D}{T} \]

\[ Q(x) = ? \]
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

   halt

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

Participant $p_i$

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

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
**State Machine Replication**

**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

\((f = 1)\)
Chain replication

Tail can respond immediately, without waiting for the new update

update  

Head  $f + 1$ replicas  

Tail

query  
reply
**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
Paxos

Abstract

The Paxos algorithm, when presented in plain English, is very simple.
PAXOS AT WORK

Proposer

Acceptors

IAmLeader YouAreLeader Decree

Learner

Accept

PAXOS
### 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?

**Answer:**
\{x, y, a\} where a is the proposer’s value
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 #2! Grant me your blessing!

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

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

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

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

Greetings, peasants! I am your fearless leader #7! 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

Pre-prepare phase | Prepare phase | Commit phase | Reply phase
First execute...
(multithreaded and without agreeing on the order)

Execute

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

Verify