Byzantine Fault Tolerance

Consensus Strikes Back (continued)
Announcements
Lab 2

- Due in approximately 5 hours.
  - If you haven't started yet then ... I don't really know how to help.
- Note, even with late days you must submit by 23:59 on Saturday.
- Unless you have received an extension (over e-mail) from me.
Remaining Classes

• Changed one reading for next week.
  • Substantially reduced the number of pages.
• Week after is Thanksgiving -- no class. 🦃🦃
• No class on November 28th.
Remaining Classes

• No readings for class on December 5.

• Use the time for final projects.

• Spend December 12 on final project presentations.

• Essentially going to do posters, though maybe without the printing.

• Final project write-ups are due on Dec 12. Cannot accept them late.

• Between 5-12 pages single column.
Previously...
Byzantine Fault Tolerance

• Failed nodes can exhibit arbitrary behavior.
  • Send different messages to different nodes, not send messages, etc.

• Want to get consensus despite failures. Specifically ensuring:
  • **Agreement**: All correct nodes agree on a value.
  • **Validity**: Messages sent by a correct node are accepted.
Digests/Hashes

Arbitrary length input $\rightarrow h \rightarrow$ Fixed length output

- Deterministic: $h(x)$ should always be the same value.
- Not invertable -- given $h(x)$ cannot find $x$.
- Output of $h(x)$ is equivalent to a random function.
- Infeasible to find collisions.
Digital Signature

- Need an entire family of these functions, not just one.
- Parametrize with one or more "keys".

- \( f(\text{private}, \text{message}) = \text{signature} \)
- \( g(\text{public}, \text{message}, \text{signature}) = \checkmark \) iff \( f(\text{private}, \text{message}) = \text{signature} \)
AppendEntries(...,
    [], leaderCommit = 4),
    Sig(pr1, success),
    Sig(pr2, success),
    Sig(pr3, success)
Practical Byzantine Fault Tolerance
Requirements

- **Safety**: Provide linearizability.
  - Despite byzantine clients or participants. Safety preserved always.

- **Liveness**: Ensure progress.
  - Requires fewer than floor(n-1/3) failures, and partial synchrony.
  - Specifically require that message delay is bounded.
Client Request

req, sig_c(req)
Initial Client Request

- Why does the client need to sign its request?
pre-prepare(v, n, d) = \text{sig}_0(\text{pre-prepare}(v, n, d))

\langle \text{req}, \text{sig}_c(\text{req}) \rangle

v = \text{view}
\
n = \text{slot}
\n\text{d} = D(\langle \text{req}, \text{sig}_c(\text{req}) \rangle)
Pre-Prepare

• Why does the leader leave the client message out from its signature?

• Why do we need to include \( \text{sig}_c(\text{req}) \) in this case?
Prepare

req, sigₙ(req)

prepare(v, n, d, 1)

sig₁(prepare(v, n, d, 1))

v = view
n = slot
d = D(<req, sigₙ(req)>)
Prepare

• Predicate: $\text{prepared}(m, v, n, i)$

  • Node $i$ received a pre-prepare for message $m$ in view $v$ with slot $n$.
  
  • Node $i$ received $2f$ prepares from different backups.

• If $\text{prepared}(m, v, n, i)$ and $\text{prepared}(m', v, n, j)$ then

  • $D(m) = D(m')$

• Why?
Prepare

Wait for prepared to become true

req, sig_c(req)

f = maximum number of faulty nodes
Prepare

• Why wait for $2f$ prepare messages?
• Why does every node broadcast the prepare message?
  • Why not just send to leader, client, etc.
Commit

req, sig_{c}(req)

commit(v, n, d, i)

sig_{c}(commit(v, n, d, i))

f = maximum number of faulty nodes
Commit

• Predicate: committed(m, v, n)
  • prepared(m, v, n, i) is true for f+1 non-faulty nodes.
  • If committed is true then message m has been committed in slot n.
Commit

• Predicate: committed-local(m, v, n, i)
  • prepared(m, v, n, i) is true.
  • Node i accepts $2f + 1$ commit messages.

• Claim: committed-local(m, v, n, i) => committed(m, v, n)
  • For any non-faulty node i. Why?
Commit

req, sig_c(req)

Wait for committed
Response
Response

- Client waits for f+1 identical responses before accepting the response.

- Why wait for f+1?
Haven't really used those signatures yet?
Violating Liveness

req, sig_c(req)
Violating Liveness

req, $\text{sig}_c(\text{req})$

resp($r, v, 0$) $\text{sig}_1(\text{resp})$

0

1

2

3
Fixing Liveness Problems
Fixing Liveness Problems

req, sig_c(req) → req, sig_c(req)
Fixing Liveness Problems

req, sig(req)

req, sig(req)
Fixing Liveness Problems

req, sig_c(req)

View Change
What is Important for View Change

- Committed log entries remain committed.
- Eventually arrive at a non-faulty leader.
Arriving at a Non-Faulty Leader

View 1
Arriving at a Non-Faulty Leader
Arriving at a Non-Faulty Leader

For view $V$, leader is $V \mod (\# \text{ of peers})$

At most $f$ view changes before arriving at a correct leader.
View Change

\[
\text{view-change}(v+1, \text{prepared}, 1) \\
\text{sig:}(\text{view-change}(v+1, \text{prepared}, i))
\]
View Change

\[
\text{view-change}(v+1, \text{prepared}, 1) \quad \text{sig}_{i}(\text{view-change}(v+1, \text{prepared}, 1))
\]

\[
\text{prepared} = [(\text{pre-prepare}(n, v, d), \quad \text{sig}(\text{pre-prepare}(n, v, d), \quad \text{prepare}(v, n, d, i_{0}), \quad \text{sig}_{0}(\text{prepare}(v, n, d, i_{0}))), \quad \text{prepare}(v, n, d, i_{1}), \quad \text{sig}_{1}(\text{prepare}(v, n, d, i_{1}))), \quad \text{prepare}(v, n, d, i_{2}), \quad \text{sig}_{2}(\text{prepare}(v, n, d, i_{2}))), \quad \ldots), \\
(\text{pre-prepare}(n', v', d'), \quad \ldots), \\
\ldots)]
\]
What is Included in Prepared

\[
\text{prepared} = [(\text{pre-prepare}(n, v, d), \\
\text{sig}(\text{pre-prepare}(n, v, d)), \\
\text{prepare}(v, n, d, i_0), \\
\text{sig}_{i_0}(\text{prepare}(v, n, d, i_0)), \\
\text{prepare}(v, n, d, i_1), \\
\text{sig}_{i_1}(\text{prepare}(v, n, d, i_1)), \\
\text{prepare}(v, n, d, i_2), \\
\text{sig}_{i_2}(\text{prepare}(v, n, d, i_2)), \\
\ldots), \\
(\text{pre-prepare}(n', v', d'), \\
\ldots), \\
\ldots]
\]

At slot n, prepared message with digest d, in view v.
Here is evidence that I received the pre-prepare message.
Received a prepare from i₀
Proof that I did the right thing in preparing this slot.
Send view-change requests on timeout. Why?
Leader of new view waits for $2f$ valid view changes. Why $2f$?
View Change

• Once leader for view v+1 has received 2f valid view-changes it is leader.

• Needs to produce a single consolidated log.

• Must include all committed log entries.
View Change

<table>
<thead>
<tr>
<th>Peer 0</th>
<th>Peer 1</th>
<th>Peer 2</th>
<th>Peer 3</th>
<th>Peer 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, 0, d)</td>
<td>(2, 0, d')</td>
<td>(2, 0, d')</td>
<td>(1, 0, d'')</td>
<td>(1, 0, d'''</td>
</tr>
</tbody>
</table>

Is this possible?

Tuple form: (v, n, d)
**View Change**

<table>
<thead>
<tr>
<th>Peer 0</th>
<th>Peer 1</th>
<th>Peer 2</th>
<th>Peer 3</th>
<th>Peer 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, 0, d)</td>
<td>(2, 0, d')</td>
<td>(2, 0, d')</td>
<td>(1, 0, d)</td>
<td>(1, 0, d)</td>
</tr>
</tbody>
</table>

Is it possible that $d \neq d'$?

Tuple form: $(v, n, d)$
View Change

<table>
<thead>
<tr>
<th>Peer 0</th>
<th>Peer 1</th>
<th>Peer 2</th>
<th>Peer 3</th>
<th>Peer 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, 0, d)</td>
<td>(2, 0, d')</td>
<td>(2, 0, d')</td>
<td>(1, 0, d)</td>
<td>(1, 0, d)</td>
</tr>
<tr>
<td>(1, 1, e)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Tuple form: (v, n, d)
### View Change

<table>
<thead>
<tr>
<th>Peer 0</th>
<th>Peer 1</th>
<th>Peer 2</th>
<th>Peer 3</th>
<th>Peer 4</th>
<th>Merged Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, 0, d)</td>
<td>(2, 0, d')</td>
<td>(2, 0, d')</td>
<td>(1, 0, d)</td>
<td>(1, 0, d)</td>
<td>(3, 0, d')</td>
</tr>
<tr>
<td>(1, 1, e)</td>
<td>(1, 2, f)</td>
<td>(1, 0, d)</td>
<td>(1, 0, d)</td>
<td>(1, 0, d)</td>
<td>(3, 1, e)</td>
</tr>
<tr>
<td></td>
<td>(1, 5, g)</td>
<td>(1, 0, d)</td>
<td>(1, 0, d)</td>
<td>(1, 0, d)</td>
<td>(3, 2, f)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(3, 0, d')</td>
<td>(3, 3, noop)</td>
<td>(3, 3, noop)</td>
<td>(3, 5, g)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(3, 4, noop)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Tuple form: (v, n, d)
# View Change

<table>
<thead>
<tr>
<th>Peer 0</th>
<th>Peer 1</th>
<th>Peer 2</th>
<th>Peer 3</th>
<th>Peer 4</th>
<th>Merged Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, 0, d)</td>
<td>(2, 0, d')</td>
<td>(2, 0, d')</td>
<td>(1, 0, d)</td>
<td>(1, 0, d)</td>
<td>(3, 0, d')</td>
</tr>
<tr>
<td>(1, 1, e)</td>
<td>(1, 2, f)</td>
<td>(1, 5, g)</td>
<td>(3, 4, noop)</td>
<td>(3, 5, g)</td>
<td>(3, 1, e)</td>
</tr>
</tbody>
</table>

**Claim:** This ensures committed entries remain committed. Why?
View Change

```
new-view(v+1, [prepared_1, prepared_2, prepared_3], merged, 1)
```

```
sig_1(new-view(v+1, [prepared_1, prepared_2, prepared_3], merged, 1))
```

```
new-view(v+1, [prepared_1, prepared_2, prepared_3], merged, 1)
```

```
sig_1(new-view(v+1, [prepared_1, prepared_2, prepared_3], merged, 1))
```

```
new-view(v+1, [prepared_1, prepared_2, prepared_3], merged, 1)
```

```
sig_1(new-view(v+1, [prepared_1, prepared_2, prepared_3], merged, 1))
```

```
new-view(v+1, [prepared_1, prepared_2, prepared_3], merged, 1)
```

```
sig_1(new-view(v+1, [prepared_1, prepared_2, prepared_3], merged, 1))
```
Zyzzyva
Observation

• PBFT require $n^2$ messages when deciding on a value.
  • Every node sends prepare messages to every other node.
  • Every node sends commit messages to every other node.

• Can we do better?
  • Have all nodes send prepare/commit messages to one node, that counts.
  • What node to pick: pick the client -- anyways involved in the process.
Observation 2

• So far we have assumed that one cannot undo state machine operations.
• Here we assume there is some way to undo an applied operation.
  • Enable speculatively executing operations before committed.
Mechanism

\[ c \]

\[
\begin{align*}
\text{pre-prepare}(v, n, d) & \quad \text{sig}_0(\text{pre-prepare}(v, n, d)) \\
\langle \text{req}, \text{sig}_c(\text{req}) \rangle & \\
\text{spec-response}(v, n, d, r, i) & \quad \text{sig}_i(\text{spec-response}(v, n, d, r, i))
\end{align*}
\]

\[ r \text{ = computed response} \]
Mechanism

$v = \text{view}$
$n = \text{slot}$
$d = D(<\text{req}, \text{sig}_c(\text{req})>)$
$r = \text{computed response}$

commit-certificate($v$, $n$, $d$, 
$[\text{sig}_0(\text{spec-response}(v, n, d, r, i)), ...])$,
$\text{sig}_c(\text{commit-certificate})$
Problem

- The only nodes which know what is committed are
  - Ones which receive a commit certificate.
  - Clients.
- Clients are not involved in view changes.
  - Challenge: how to preserve committed log entries?
- Abraham, Gueta and Malkhi show it doesn't.