Byzantine Fault Tolerance

Consensus Strikes Back
Announcements
Lab 2
Lab 2

• Hopefully everyone has started by now, maybe even finished large portions.
Lab 2

• Hopefully everyone has started by now, maybe even finished large portions.

• If not ... you should worry.
Lab 2

• Hopefully everyone has started by now, maybe even finished large portions.

• If not ... you should worry.

• Please don't change the protobufs.
Lab 2

- Hopefully everyone has started by now, maybe even finished large portions.
- If not ... you **should worry**.
- Please don't change the protobufs.
- My testing strategy is going to be to write a few clients and check linearizability.
Lab 2

• Hopefully everyone has started by now, maybe even finished large portions.

• If not ... you **should worry**.

• Please don't change the protobufs.

• My testing strategy is going to be to write a few clients and check linearizability.

• Changing the interface doesn't let me do that.
Lab 2

• Hopefully everyone has started by now, maybe even finished large portions.

• If not ... you **should worry**.

• Please don't change the protobufs.

• My testing strategy is going to be to write a few clients and check linearizability.

• Changing the interface doesn't let me do that.

• Feel free to change whatever is not the interface.
BFT
A Note on Terminology

• Byzantine Empire?
A Note on Terminology

• Byzantine Empire?

• Continuation of the Roman Empire, ~400-1450 AD
A Note on Terminology

- Byzantine Empire?
  - Continuation of the Roman Empire, ~400-1450 AD
  - Commonly used as example of bad bureaucracy, in fighting...
A Note on Terminology

• Byzantine Empire?
  • Continuation of the Roman Empire, ~400-1450 AD
  • Commonly used as example of bad bureaucracy, in fighting...
  • Historical records don't entirely agree with this.
What is the Problem?
What is the Problem?

0 1 2 3 4
Concrete Problems

AppendEntries(..., [(index=4)])
Concrete Problems

![Diagram showing a sequence of events involving `AppendEntries` function, with an arrow labeled "Success" pointing to one of the steps.]

- `AppendEntries(..., [(index=4)])`
- Success
Concrete Problems

AppendEntries(..., [(index=4)])

Success

AppendEntries(..., [], leaderCommit=4)
Concrete Problems

0

1

2
Concrete Problems

RequestVote(term=2)
Concrete Problems

RequestVote(term=2)
RequestVote(term=2)
Concrete Problems

0

1

2

RequestVote(term=2)

RequestVote(term=2)

VoteGranted(term=2)
Concrete Problems

0

1

2

RequestVote(term=2)

VoteGranted(term=2)

VoteGranted(term=2)

RequestVote(term=2)

RequestVote(term=2)
Concrete Problems

As far as I understanding, the handling is not part of the raft algorithm, right?
My handling is to have the first timeout reasonably large so that the newly started peer is not overturning the current leader by candidate voting. (because the append entries failed to reach to it in time, and it request vote for higher term and the leader will compromise)

Any other better way to deal with this?

This handling is a part of the Raft algorithm, not working due to network instability would render it incorrect.

The raft algorithm assumes that once the connection is there, the append entries will be delivered to the newly started peer before it get granted true for a higher term. But when I look at the log, the current leader can even reply to the newly started peer’s vote request but the append entries are still failing to delivered. So consider below scenario:

```
peer0 (current leader):
  Index   0   1   2   3   4
  Term    0   3   4   4   5 (committed up to this)

peer1 (follower):
  Index   0   1   2   3   4
  Term    0   3   4   4   5 (committed up to this)

peer2 (restarted, was committed just up index 1)
  Index   0   1   2   3   4
  Term    0   3   -   -   -
```

1. peer2 started, but the connection is not stable (do not retrieve any append entries)
2. peer2 election timeout, ask for vote as candidate
3. get rejected because its current term is smaller, T of the new peer is set to 5 then
4. peer2 election timeout, ask for vote as candidate, T = T + 1 = 6
5. other peer vote true for that as T > their current Term
   (at this point, the new peer is still not receiving any append entries from the current leader. May be it is not the issue of conn stability, not sure why it is not delivered to new peer even the new peer can communicate about voting with the leader.)
6. the new peer become leader but with an outdated log, the log replication will overwrite other peers committed log.
Failure Models

• Until now we have considered **fail-stop** processes.
Failure Models

- Until now we have considered **fail-stop** processes.
  - When failed: stop sending messages and take no steps.
Failure Models

- Until now we have considered **fail-stop** processes.
  - When failed: stop sending messages and take no steps.
- Byzantine faults: when failed do "arbitrary things."
Failure Models

• Until now we have considered **fail-stop** processes.
  
  • When failed: stop sending messages and take no steps.

• Byzantine faults: when failed do "arbitrary things."
  
  • These arbitrary things could even be coordinated with other failed nodes.
However assuming we know participants a priori.
“On the Internet, nobody knows you’re a dog.”
On the internet nobody knows what maps to a user, nor to a machine, ...
Not Considering this Problem

• Live in a centralized environment.
Not Considering this Problem

• Live in a centralized environment.

• All servers/nodes are launched by some centralized entity.
Not Considering this Problem

• Live in a centralized environment.
• All servers/nodes are launched by some centralized entity.
• For example Kubernetes or a human with physical access.
Not Considering this Problem

• Live in a centralized environment.
  • All servers/nodes are launched by some centralized entity.
  • For example Kubernetes or a human with physical access.

• Several ways to solve the decentralized problem.
Not Considering this Problem

• Live in a centralized environment.
  • All servers/nodes are launched by some centralized entity.
  • For example Kubernetes or a human with physical access.
• Several ways to solve the decentralized problem.
  • But largely separable from the discussion at hand.
Is This Still Useful?
Is This Still Useful?

• Yes...
Is This Still Useful?

- Yes...
- Used by Boeing in the 777 to ensure safety.
Is This Still Useful?

- Yes...
- Used by Boeing in the 777 to ensure safety.
- Used in SpaceX Falcon -- "... to meet requirements for approaching the ISS"
Is This Still Useful?

• Yes...

• Used by Boeing in the 777 to ensure safety.

• Used in SpaceX Falcon -- "... to meet requirements for approaching the ISS"

• Generally useful, but cost prohibitive.
Failure Models

• Until now we have considered fail-stop processes.
Failure Models

• Until now we have considered **fail-stop** processes.

• When failed: stop sending messages and take no steps.
Failure Models

• Until now we have considered **fail-stop** processes.
  
  • When failed: stop sending messages and take no steps.

• Byzantine faults: when failed do "arbitrary things."
Failure Models

- Until now we have considered **fail-stop** processes.
  - When failed: stop sending messages and take no steps.
  - Byzantine faults: when failed do "arbitrary things."
  - These arbitrary things could even be coordinated with other failed nodes.
What Can we Do?
What Do We Care about Addressing

0 State

1 State  2 State  3 State  4 State
What Do We Care about Addressing

Can't really peer into the state of a remote node, cannot do much.
Failed nodes can only interfere by sending messages.
What Do We Care about Addressing

Make sure messages sent by all nodes are "correct" before acting.
Why challenging?
Why challenging?

Don't know failed nodes a-priori.
When are Messages Correct?
When are Messages Correct?

- Every correct node receives the same messages (and acts correctly).
When are Messages Correct?

• Every correct node receives the same messages (and acts correctly).
• Same might not necessarily mean "correct".
When are Messages Correct?

- Every correct node receives the same messages (and acts correctly).
  - Same might not necessarily mean "correct".
  - But always accept any message from a correct participant.
When are Messages Correct?

• Every correct node receives the same messages (and acts correctly).

• Same might not necessarily mean "correct".

• But always accept any message from a correct participant.

• Every message is "consistent" with the protocol.
When are Messages Correct?

• Every correct node receives the same messages (and acts correctly).

  • Same might not necessarily mean "correct".

  • But always accept any message from a correct participant.

• Every message is "consistent" with the protocol.

  • Attach some kind of proof that you were supposed to send this message.
When are Messages Correct?

- Every correct node receives the same messages (and acts correctly).
- Every message is "consistent" with the protocol.
Agreeing on Correct Messages
Problem we Want to Solve

```
0
  AppendEntries(...,
               [(index=4)])
    1  2  3  4
```
Problem we Want to Solve

0 <- Success
1 2 3 4
Problem we Want to Solve

AppendEntries(..., [ ], leaderCommit = 4)
Problem we Want to Solve

0

AppendEntries(...,
[(index=4)])

1 2 3 4
Problem we Want to Solve

```
AppendEntries(..., [], leaderCommit = 4)
```

Diagram:
```
0 -> 1, 2, 3, 4
```
Problem we Want to Solve

• Cannot observe messages between individuals.
Problem we Want to Solve

- Cannot observe messages between individuals.
- Hard to judge whether behavior is correct.
Problem we Want to Solve

• Cannot observe messages between individuals.
  • Hard to judge whether behavior is correct.
• New idea: send messages to everyone.
Problem we Want to Solve

• Cannot observe messages between individuals.
  • Hard to judge whether behavior is correct.
• New idea: send messages to everyone.
  • Everyone knows where the state machine should be.
Sending to Everyone

0→1: AppendEntries(..., [(index=4)])
Sending to Everyone
Sending to Everyone is Insufficient

0→1: AppendEntries(..., [(c0, index=4)])
Sending to Everyone is Insufficient

![Diagram]

- Node 0
- Success
- Transition 1 → 2
- Transition 2 → 3
- Transition 3 → 4
- Transition 4 → 1
Sending to Everyone is Insufficient

Slot 4 is c0
Sending to Everyone is Insufficient

![Diagram]

0

1

2

3

4

Success

1 thinks slot 4 is c1

Slot 4 is c0
Sending to Everyone is Insufficient

Slot 4 is c0

1 thinks slot 4 is c1

1 thinks slot 4 is c0
Sending to Everyone is Not Sufficient

- Faulty node can send differing messages to "everyone".
Sending to Everyone is Not Sufficient

- Faulty node can send differing messages to "everyone".
- Run some protocol to detect this problem.
Sending to Everyone

0→1: AppendEntries(..., [(c0, index=4)])
0→1: AppendEntries(..., [(c1, index=4)])
Sending to Everyone
Sending to Everyone
Sending to Everyone

Choose majority, breaking ties deterministically.
Sending to Everyone

0

1

2

3

4

0 0->1: c0, 4
1
2  ???
3 0->1: c0, 4
4 0->1: c0, 4

0 0->1: c0, 4
1 0->1: c0, 4
2  ???
3
4 0->1: c0, 4

0 0->1: c0, 4
1 0->1: c0, 4
2  ???
3 0->1: c0, 4
4
Sending to Everyone

Choose majority, breaking ties deterministically.
Not Possible for 1 failure with 3 participants

0

1

2

0→1: x=2

0→1: x=1

1

0

2

0→1: x=1

0→1: x=1
Not Possible for 1 failure with 3 participants
Not Possible for 1 failure with 3 participants

Cannot distinguish between these two cases. Cannot meet the two requirements state at the beginning.
Limitations

• More generally cannot solve for $m$ failures with $< 3m + 1$ participants.
Limitations

• More generally cannot solve for $m$ failures with $< 3m+1$ participants.

• Proof by reduction to the case with 3.
• However, note that doing this once is not sufficient for more than 1 faults.
• However, note that doing this once is not sufficient for more than 1 faults.
• For example, can force any decision in this case.
Solution: Recursively call again.
When are Messages Correct?

- Every correct node receives the same messages (and acts correctly).
- Every message is "consistent" with the protocol.
Proving Consistency with the Protocol
What Does this Even Mean?

```
AppendEntries(...,
[(index=4)])
```
What Does this Even Mean?

0

Success

1  2  3  4
What Does this Even Mean?

AppendEntries(...,
    [], leaderCommit = 4),
Proof that a majority have accepted entries until 4.
Problem

- How to generate proofs?
Problem

• How to generate proofs?
  • Many possibilities, but just going to include messages here.
Problem

• How to generate proofs?
  • Many possibilities, but just going to include messages here.
• How to prevent failed nodes from misrepresenting messages?
Misrepresenting Messages

$\text{AppendEntries}(\ldots, []), \text{leaderCommit} = 4)$,
Success from $0, 1, 2, 3$

0 -> 1, 2, 3, 4
Misrepresenting Messages

AppendEntries(...,
    [], leaderCommit = 4),
Success from 0, 1, 2, 3
Warning: Cryptography
Digests/Hashes

Arbitrary length input → $h$ → Fixed length output
Digests/Hashes

Arbitrary length input $\xrightarrow{h}$ Fixed length output

- Deterministic: $h(x)$ should always be the same value.
Digests/Hashes

- Deterministic: $h(x)$ should always be the same value.
- Not invertable -- given $h(x)$ cannot find $x$. 

Arbitrary length input $\rightarrow h \rightarrow$ Fixed length output
Digests/Hashes

Arbitrary length input → $h$ → 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.
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

Input $\rightarrow f \rightarrow$ Signature

Input Signature $\rightarrow g \rightarrow $ ✔ or ✗
Digital Signature

\[ g(m, s) = \checkmark \text{ if and only if } f(m) = s. \]
Digital Signature

\[ g(m, s) = \checkmark \text{ if and only if } f(m) = s. \]

Owe you 10 scoops of ice cream.
\[ f(\text{Owe you 10 scoops of ice cream.}) \]
Digital Signature

Input $\rightarrow f \rightarrow$ Signature

Input $\rightarrow g \rightarrow \checkmark$ or $\times$

$g(m, s) = \checkmark$ if and only if $f(m) = s.$
Digital Signature

Input $\rightarrow f \rightarrow$ Signature

Input Signature $\rightarrow g \rightarrow \checkmark$ or $\times$

$g(m, s) = \checkmark$ if and only if $f(m) = s$.

Owe you 10 scoops of ice cream. $f(\text{Owe you 10 scoops of ice cream.})$
Digital Signature

\[ g(m, s) = \checkmark \text{ if and only if } f(m) = s. \]

Owe you 10 scoops of ice cream. \( f(\text{Owe you 10 scoops of ice cream.}) \)

Owe you 0 scoops of ice cream. \( f(\text{Owe you 0 scoops of ice cream.}) \)
Digital Signature

\[ g(m, s) = \checkmark \text{ if and only if } f(m) = s. \]
Digital Signature

\[ g(m, s) = \checkmark \text{ if and only if } f(m) = s. \]
Digital Signature

\[ g(m, s) = \checkmark \text{ if and only if } f(m) = s. \]

You owe Grizzly 10 scoops.

\[ f(\text{You owe Grizzly 10 scoops of ice cream.}) \]

Owe you 0 scoops of ice cream.

\[ g(\text{Owe you 0 scoops of ice cream.}) \]

Owe you 10 scoops of ice cream.

\[ f(\text{Owe you 10 scoops of ice cream.}) \]

Provides a non-repudiation mechanism.
Digital Signature

- Need an entire family of these functions, not just one.
- Why?

\[ g(m, s) = \checkmark \text{ if and only if } f(m) = s. \]
Digital Signature

\[ g(m, s) = \checkmark \text{ if and only if } f(m) = s. \]

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

- Need an entire family of these functions, not just one.
  - Why?
- Parametrize with one or more "keys".
  - \( f(\text{key, message}) = \text{signature} \)

\[
\begin{align*}
g(m, s) &= \checkmark \text{ if and only if } f(m) = s. \\
\end{align*}
\]
Digital Signature

- Need an entire family of these functions, not just one.
- Why?
- Parametrize with one or more "keys".
  - \( f(\text{key}, \text{message}) = \text{signature} \)
  - \( g(\text{key}', \text{message}, \text{signature}) = \sqrt{\text{ }} \) if and only if \( f(\text{key}, \text{message}) = \text{signature} \)
Digital Signature

- Need an entire family of these functions, not just one.
- Parametrize with one or more "keys".
  - \( f(\text{key } \text{private}, \text{message}) = \text{signature} \)
  - \( g(\text{key}^1 \text{ public}, \text{message, signature}) = \checkmark \text{ iff } f(\text{key} \text{private}, \text{message}) = \text{signature} \)
Proving Message Receipt

AppendEntries(...,
    [], leaderCommit = 4),
Sig(pr1, success),
Sig(pr2, success),
Sig(pr3, success)
Proving Message Receipt

```
AppendEntries(...,
    [], leaderCommit = 4),
Sig(pr1, success),
Sig(pr2, success),
Sig(pr3, success)
```

Diagram:
- Node 0 connects to nodes 1, 2, 3, and 4.
- Nodes 1, 2, 3, and 4 are labeled Pr1, Pr2, Pr3, and Pr4, respectively.
Proving Message Receipt

• Looks simple in principle, but tricky to get right in practice.

• PBFT takes this basic idea and packages it into an optimized version.
PBFT

\[ \text{req, sig}_c(\text{req}) \]
PBFT

\[ \text{req, } \text{sig}_c(\text{req}) \]

\[ \text{pre-prep}(v, n, d) \]
\[ \text{sig}_0(\text{pre-prep}(v, n, d)) \]
\[ \langle \text{req, } \text{sig}_c(\text{req}) \rangle \]
PBFT

pre-prep(v, n, d)  \[\text{sig}_0(\text{pre-prep}(v, n, d))\]

<req, \text{sig}_c(\text{req})>
PBFT

req, sig_c(req)

prep(v, n, d, i)

sig_i(prep(v, n, d, i))
Wait for 2m prepare messages

req, sigc(req)
PBFT

req, sig_c(req)

commit(v, n, d, i)

sig_c(commit(v, n, d, i))
PBFT

req, sig_c(req)

Wait for 2m commit messages
Haven't used those signatures yet?
Where are the Signatures Used?

- View change: prove that you are not trying to cheat when consolidating logs.
Where are the Signatures Used?

- View change: prove that you are not trying to cheat when consolidating logs.
- Another important bit: in each view leader is decided by view-number.
Where are the Signatures Used?

- View change: prove that you are not trying to cheat when consolidating logs.
- Another important bit: in each view leader is decided by view-number.
- Not by voting as in Raft.
Where are the Signatures Used?

• View change: prove that you are not trying to cheat when consolidating logs.

• Another important bit: in each view leader is decided by view-number.
  • Not by voting as in Raft.

• Why is this important?
Where are the Signatures Used?

• View change: prove that you are not trying to cheat when consolidating logs.

• Another important bit: in each view leader is decided by view-number.
  • Not by voting as in Raft.
  • Why is this important?

• Not discussing details today, might return to this next week.
Quiz