Byzantine fault tolerance

Group Members:
Anqi Hu        Huijuan Huang    Xiaoxin Li
Zheng Qin      Yingfei Zeng     Zilin Zhang
Chaofeng Zhou
What is Byzantine Generals' Problem?
Byzantine Generals' Problem

Attacking VS Retreat
Byzantine Generals' Problem

attacking

retreat

attacking

retreat
When a group is trying to make a collective decision about how it will act...

- The traitors may tell some members that they wish to do one thing, and tell other members the opposite.
- This can cause problems for the group's ability to act in unison and further cause the environment to break apart.
- E.g., a server cluster won't work well if some servers within it fail to pass data consistently to other servers; a computer network will fail if the devices on it do not agree on a common networking protocol to use when exchanging information.
Definition

1. Byzantine Fault: a fault that presents different symptoms to different observers -> the faulty node exhibits arbitrary behavior. Such fault can occur, e.g., due to a software bug, a hardware malfunction, or a malicious attack.

2. Byzantine Failure: the loss of a system component due to a Byzantine Fault in a distributed system that requires consensus.

3. Byzantine fault tolerance: the characteristic of a system that tolerates Byzantine failures.
Real World Scenarios

- Space shuttle data bus standing wave
- Space shuttle mission STS-124
- Mid-value select
- Command / Monitor wrap-back
- Time-Triggered Protocol (TTP/C) heavy ion fault injection
- Multi-Microprocessor Flight Control System
- Potential grounding of an entire aircraft fleet
- A pushbutton input to the command and monitor lanes of an airplane brake system caused the system to fail

Reference: https://c3.nasa.gov/dashlink/static/media/other/ObservedFailures1.html
The shuttle data bus example

A technician used the wrong resistor value for bus termination.

Reflections from the impedance mismatch caused a standing wave on the bus.

The bus used the quad computer system — Two: attached to the bus at antinodes of this standing wave; The rest two: attached at nodes

Due to the influence of the wave, these two computers might output different values in different time, which caused the loss of the system.
Failure Models and Related Work
## Failure Model in Network

<table>
<thead>
<tr>
<th>Type of failure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Omission failure</td>
<td>A server fails to respond to incoming requests (cannot receive or respond the request)</td>
</tr>
<tr>
<td>Timing failure</td>
<td>A server's response lies outside the specified time interval</td>
</tr>
<tr>
<td><strong>Crash failure or fail-stop</strong></td>
<td>A server halts, but is working correctly until it halts</td>
</tr>
<tr>
<td>Response failure</td>
<td>The server’s response is incorrect (malicious behavior or software problems)</td>
</tr>
<tr>
<td><strong>Byzantine</strong></td>
<td>A server may produce arbitrary responses at arbitrary times</td>
</tr>
</tbody>
</table>
How to Handle Failure

Failures might easily occur without warning.

It’s hard to prevent failures or repair failures in a short time.

A good strategy is that we tolerate failure.

How can we tolerate failure?

Replication!

If one node fails other good nodes can keep the system functioning well.
Fail-Stop Failure & Byzantine Failure

Two most common failure:

- Fail-stop failure: A node is either correct or dead (Can be detected). Correct node always obey the protocols. (Related work: Paxos, RAFT.)
- Byzantion failure: We cannot detect whether a node is faulty or not. The most serious!!!!!

Questions:

How can we handle these two types of failure?

What’s the difference?

Can we use Paxos to handle Byzantine failure?
Fail-Stop Failure

Attracted a lot of research interest in 80's.
The main problems of network come from hardware. Servers usually crash.

Require:

- $2f + 1$ replicas to tolerate for $\leq f$ faulty nodes.

- quorums$(f + 1)$ out of $2f + 1$ nodes in every operation.

Why $2f + 1$?

Why don’t we just have $f + 1$?
Fail-Stop Failure

Let $N$ be the number of nodes, and let a quorum be any $N - f$ nodes. For safety, we require that any two quorums have a non-empty intersection, which is true if $N - f > f$. So we have at least $2f + 1$ nodes.

We only need a quorum $(f + 1)$ of nodes to reach the consensus:

Any two majority sets of nodes will have at least one node in common.

if two proposals are processed by two majority, there must be at least one node that agreed to both.
Fail-Stop Failure

Two main roles: Proposer(1), Acceptor(2f + 1).

Two phase protocol:

1) Prepare phase: the proposer sends prepare message with sequence number N to all acceptors, wait for answer from a quorums of acceptors. The acceptors will accept the prepare message with sequence higher than what they have received before.

2) Accept phase: the proposer sends a value to acceptors and wait for it to be accepted by a quorum. If proposed value is accepted by majority, proposer announces the consensus value.
Paxos under Byzantine Faults

Can Paxos work with Byzantine nodes?

- Can’t rely on the proposer (primary) to assign seqno
  Malicious primary can assign the same seqno to different requests! It can lie!

- Can’t use Paxos for view change
  Does the intersection of two quorums always contain one honest node?
  No, Bad node tells different things to different quorums!

Other nodes can be fooled by the malicious node.
Two majority will conflicts with each other.
Paxos under Byzantine Faults

Paxos under Byzantine faults

(f = 1)
Paxos under Byzantine Faults

Paxos under Byzantine faults

(f = 1)

N0
n_t=N0:1

N1
n_t=N2:1

Prepare(N2:1)

OK

Decide
xyz

N2
Paxos under Byzantine Faults

Conflicting decisions!
We need a new approach to handle the Byzantine fault!

Main Ideas of Practical Byzantine Fault Tolerance

- 3f+1 replicas
- To deal with malicious primary (proposer)
  Use a 3-phase protocol to agree on sequence number
- To deal with loss of agreement
  Use a bigger quorum (2f+1 out of 3f+1 nodes)
Assumptions

- Asynchronous system and network failures.
- Uses a Byzantine failure model.
  - Allows faulty nodes collude, delay communication or delay correct nodes.
- Independent node failures.
  - Each node should run different implementations of the service code and operating system and should have a different root password and a different administrator.
- Strong cryptography and the adversary is computationally bound.
  - Unable to subvert the cryptographic techniques.
Service Properties

- Provides both safety and liveness assuming no more than \( f = \left\lfloor \frac{n-1}{3} \right\rfloor \) replicas are faulty.
  - Safety means that the replicated service behaves like a centralized implementation that executes operations atomically one at a time.
  - Liveness means clients eventually receive replies to their requests.

- The algorithm does not address the problem of fault-tolerant privacy.
  - A faulty replica may leak information to an attacker.
Views

- Operations occur within views.

- For a given view, a particular node is designated the primary node, and the others are backup nodes.
  - Primary picks the order of operations by assigning an increasing sequence number.
  - Backup ensure primary behaves correctly.

- Primary = v mod n
  - v is the view number
  - n is the number of node
A client sends a request for a service to the primary.
**Rough Overview Of Algorithm**

- The primary multicasts the request to the backups.
Rough Overview Of Algorithm

- Backups verify the request.
Rough Overview Of Algorithm

- All execute the request and send the reply to the client.
- The clients waits for $f + 1$ replies from different replicas with the same result, which will be the result of the operation.
Phase 1: Pre-Prepare (primary to replicas)

1. Primary sends \( \langle \text{PRE-PREPARE}, \text{view}, \text{seqn}, \text{digest} \rangle \), req to all other replica

2. A replica accepts PRE-PREPARE:
   - Digest and signatures are correct
   - It is in the same view
   - It has not accepted the same view and seqn containing a different digest
   - seqn is in a certain range

3. If it accepts, then multicast PREPARE to all other replicas
Phase 2: Prepare(replicas to replicas)

1. Each replica wait for $2f$ matching PREPARE messages
2. Puts these messages in log
3. We say $(req, v, n, i)$ is true, where $i$ is node ID.

Above can guarantee:
If $\text{prepared}(req, v, n, i)$ is TRUE for honest replica $ri$, then $\text{prepared}(req', v, n, j)$ where $req' \neq req$ FALSE for any honest $rj$.
Why no double prepares?

Remember that: Each replica wait for 2f matching PREPARE messages.

Honest intersection of maximally disjoint 2f+1 sets is non-empty.
Phase 3: Commit(replicas to client)

1. Make sure op doesn’t execute until prepared(req, view, seqn, i) is True for f+1 non-faulty replicas.
2. We say committed(req, view, seqn) is true when this property holds.

How does replica know committed(req, v, n) holds?

- Add one more message: ri \rightarrow R
  \{COMMIT, view, seqn, req, id\}
- Once 2f+1 COMMITs at a node, then apply op and respond to client.
View Changes
View Changes

The view change mechanism
- Protects against faulty primaries

Backups propose a view change when a timer expires
- The timer runs whenever a backup has accepted some message & is waiting to execute it.
- Once a view change is proposed, the backup will no longer accept messages (except view-change, new-view) in the current view.
View Changes – Step 1

Step 1: Backups which timeout multicast

$<\text{VIEW-CHANGE}, \text{view} + 1, i>$
Step 1: Backups which timeout multicast

\(<\text{VIEW-CHANGE}, \text{view } + 1, i>\)

Step 2: When the primary of \(v + 1\) receives \(2f\) view-change messages for view \(v + 1\), it multicasts

\(<\text{NEW-VIEW}, \text{view } + 1, V, O>\)

\(V\): a set containing the view-change messages received by the primary plus the view-changes messages it sent

\(O\): a set of pre-prepare messages
View Changes – Step 2

Recipients of NEW-VIEW:
- Verify the pre-prepare messages
- Send a prepare message for each pre-prepare
- Enter the new view
Correctness
Safety & Liveness

**Safety** -- nothing bad happens (no reachable ERROR/STOP state)

**Liveness** -- something good eventually happens (an action is eventually executed properly)
Safety

1. Non-faulty replicas agree on the sequence numbers of requests that commit locally in the same view.

If prepared \((m, v, n, i)\) is true, prepared \((m', v, n, j)\) is false for any non-faulty replica \(j\) (including \(i = j\)) and any \(m'\) such that \(D(m') \neq D(m)\).
2. Non-faulty replicas also agree on the sequence number of requests that commit locally in different views at different replicas.

Set R1 contains at least $f + 1$ non-faulty replicas such that prepared $(m, v, n, i)$ is true for every replica $i$ in the set.

Any correct new-view message for view $v' > v$ contains correct view-change messages from every replica $i$ in a set R2 of at least $2f + 1$ replicas.

R1 and R2 must intersect in at least one replica $k$ that is not faulty. $k$’s view-change message will ensure that the fact that $m$ prepared in a previous view is propagated to subsequent views.
Liveness – View Changes

1. Can’t let a single node start a view change!
   • Why? Could livelock the system by spamming view changes

2. The timeout period increases exponentially until some requested operation executes
   • Why? To avoid starting a view-change too soon

3. If any node that gets more than \( f + 1 \) view-change requests, it sends a view-change message for the smallest view in the set, even if its timer has not expired.
   • Why? To avoid starting a view change too late when they time out slowly and then the oldest view-change issuer rolls over to view-change \( v + 2 \).
Communication Optimization
Communication Optimization

- Digest replies: send only one reply to client with result
Communication Optimization

- Digest replies: send only one reply to client with result
- Optimistic execution: execute prepared requests
Communication Optimization

- Digest replies: send only one reply to client with result
- Optimistic execution: execute prepared requests

- Read-only operations: executed in current state
Performance Evaluation
Micro-benchmark

Replication v. No Replication

The micro-benchmark provides a service-independent evaluation of the performance of the replication library.

- Overhead for read-only operations is significantly lower.
- Overhead is lower for the 4/0 and 0/4 operations.

Table 1: Micro-benchmark results (in milliseconds); the percentage overhead is relative to the unreplicated case.
Andrew Benchmark

BFS v. BFS-nr v. NFS-std(NFS-v2)

1. creates subdirectories recursively
2. copies a source tree
3. examines the status of all the files in the tree without examining their data
4. examines every byte of data in all the files
5. compiles and links the files
Andrew Benchmark

<table>
<thead>
<tr>
<th>phase</th>
<th>BFS strict (%)</th>
<th>r/o lookup (%)</th>
<th>BFS-nr</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.55 (57%)</td>
<td>0.47 (34%)</td>
<td>0.35</td>
</tr>
<tr>
<td>2</td>
<td>9.24 (82%)</td>
<td>7.91 (56%)</td>
<td>5.08</td>
</tr>
<tr>
<td>3</td>
<td>7.24 (18%)</td>
<td>6.45 (6%)</td>
<td>6.11</td>
</tr>
<tr>
<td>4</td>
<td>8.77 (18%)</td>
<td>7.87 (6%)</td>
<td>7.41</td>
</tr>
<tr>
<td>5</td>
<td>38.68 (20%)</td>
<td>38.38 (19%)</td>
<td>32.12</td>
</tr>
<tr>
<td>total</td>
<td>64.48 (26%)</td>
<td>61.07 (20%)</td>
<td>51.07</td>
</tr>
</tbody>
</table>

Table 2: Andrew benchmark: BFS vs BFS-nr. The times are in seconds.

The overhead of Byzantine fault tolerance for this service is low.
Andrew Benchmark

<table>
<thead>
<tr>
<th>phase</th>
<th>BFS strict</th>
<th>BFS r/o lookup</th>
<th>NFS-std</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.55 (-69%)</td>
<td>0.47 (-73%)</td>
<td>1.75</td>
</tr>
<tr>
<td>2</td>
<td>9.24 (-2%)</td>
<td>7.91 (-16%)</td>
<td>9.46</td>
</tr>
<tr>
<td>3</td>
<td>7.24 (35%)</td>
<td>6.45 (20%)</td>
<td>5.36</td>
</tr>
<tr>
<td>4</td>
<td>8.77 (32%)</td>
<td>7.87 (19%)</td>
<td>6.60</td>
</tr>
<tr>
<td>5</td>
<td>38.68 (-2%)</td>
<td>38.38 (-2%)</td>
<td>39.35</td>
</tr>
<tr>
<td>total</td>
<td>64.48 (3%)</td>
<td>61.07 (-2%)</td>
<td>62.52</td>
</tr>
</tbody>
</table>

Table 3: Andrew benchmark: BFS vs NFS-std. The times are in seconds.

3% more time to run benchmark than a commercial system that is used daily by many users
Conclusion

- Byzantine-Fault-tolerant File System is a state-machine replication algorithm that is able to tolerate Byzantine faults.
- It can be used in practice: it is the first to work in an asynchronous system like internet. And it is only 3% slower than the standard NFS implementation in Digital Unix.
- It can not survive all errors. It can only masks errors that occur independently at different replicas.