Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults

Dian Yu
Comparison with PBFT (Traditional BFT protocols)

Similarities:

- Build practical Byzantine fault tolerance systems
- Protocol: Clients → Primary → Replicas → Agreement

Differences: (Robust)

- Signature for authentication
- Regular view change
- Point to point communication
Ideal BFT systems

“Handle normal and worst case separately as a rule because the requirements for the two are quite different. The normal case must be fast. The worst case must make some progress”

Gracious execution: synchronous execution. All clients and servers behave correctly

Uncivil execution: synchronous execution. Up to f servers and any numbers of clients are Byzantine
Problem with PBFT/Zyzzyva

Misguided: current BFT systems can survive Byzantine faults, but completely unavailable by a simple failure

Dangerous: encourages fragile optimizations

Futile: Further improvements have little effect on performance

<table>
<thead>
<tr>
<th>System</th>
<th>Peak Throughput</th>
<th>Faulty Client</th>
</tr>
</thead>
<tbody>
<tr>
<td>PBFT [8]</td>
<td>61710</td>
<td>0</td>
</tr>
<tr>
<td>Q/U [1]</td>
<td>23850</td>
<td>0†</td>
</tr>
<tr>
<td>HQ [12]</td>
<td>7629</td>
<td>N/A‡</td>
</tr>
<tr>
<td>Zyzzyva [18]</td>
<td>65999</td>
<td>0</td>
</tr>
<tr>
<td>Aardvark</td>
<td>38667</td>
<td>38667</td>
</tr>
</tbody>
</table>
Aardvark: RBFT in action

3 stages:

1. Client request transmission
2. Replica agreement
3. Primary view change
Signed client requests - MAC

MAC: Message Authentication Code

If the same MAC is found: then the message is authentic and integrity checked
Else: something is not right.
Digital Signature

**Unsigned certificate:** contains user ID, user's public key, as well as information concerning the CA

- **H**
  - Generate hash code of unsigned certificate
  - Encrypt hash code with CA's private key to form signature
  - Create signed digital certificate

- **E**
  - Signed certificate

- **D**
  - Decrypt signature with CA's public key to recover hash code
  - Use certificate to verify Bob's public key

- **Bob's ID Information**
  - Bob's public key
  - CA information

- **Recipient can verify signature by comparing hash code values**

**Diagram:**
- **H** (Hash)
- **E** (Encrypt)
- **D** (Decrypt)
- **Bob's ID Information**
- **Recipient**

7/16
Signed client requests - digital signatures

Problem with MAC: no non-repudiation property of digital signatures

Solution: Signature

- Valid MAC but not valid signature:
  - Not routine message corruption
  - Significant fault or malicious behavior with client

Denial-of-service attack?

1. Hybrid MAC-signature construct
2. Complete one request first
Resource isolation

Separate network interface controllers (NICs)
Separate work queues for clients and replicas
Hardware parallelism
Regular view changes

System throughput remains high when replicas are faulty (uncivil intervals)

Cost of a view change is similar to the regular cost of agreement
Protocol Description
Client request transmission

Fundamental challenge:

Each replica comes to the same conclusion about the authenticity of the request

Request:

\[ \langle \langle \text{REQUEST}, o, s, c \rangle_{\sigma_c}, c \rangle_{\mu_{c,p}} \]

Analysis:

Signature check: ensures only requests that will be accepted by all correct replicas are processed.

Result: for every \( k \) correct requests submitted by a client, each replica performs at most \( k+1 \) signature verifications.
Replica agreement

Fundamental Challenge:

Ensure each replica can quickly collect the quorums of PREPARE and COMMIT messages necessary to make progress.

Potential solution:

1. Design a protocol so that incorrect messages from faulty replica will not gain quorum
2. If quorum of timely correct replicas exists, a faulty replica cannot impede progress.
Catchup messages

Benefit: allows temporarily slow replicas to avoid becoming permanently non-responsive

Downside: faulty replicas impose significant load on non-faulty counterparts
Primary view changes

Faulty primary: delay processing requests, discard requests, corrupt clients’ MAC authenticators, introduce gaps in the sequence number space, unfairly delay or drop clients’ requests

Past systems: conservative. Only change when the current primary does not allow the system make even minal progress

Aardvark: initiate a view change when delay exceeds heartbeat timer expires.

Fairness: PRE-PREPARES from the same client
Analysis (with proof)

1. Peak throughput during a gracious view
2. During uncivil executions, with a correct primary Aardvark’s throughput at least $g$ times the throughput of a gracious view

<table>
<thead>
<tr>
<th>System</th>
<th>Peak Throughput</th>
<th>Network Flooding UDP</th>
<th>TCP</th>
</tr>
</thead>
<tbody>
<tr>
<td>PBFT</td>
<td>61710</td>
<td>crash</td>
<td>-</td>
</tr>
<tr>
<td>Q/U</td>
<td>23850</td>
<td>23110</td>
<td>crash</td>
</tr>
<tr>
<td>HQ</td>
<td>7629</td>
<td>4470</td>
<td>0</td>
</tr>
<tr>
<td>Zyzzyva</td>
<td>65999</td>
<td>crash</td>
<td>-</td>
</tr>
<tr>
<td>Aardvark</td>
<td>38667</td>
<td>7873</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>System</th>
<th>Peak Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aardvark</td>
<td>38667</td>
</tr>
<tr>
<td>PBFT</td>
<td>61710</td>
</tr>
<tr>
<td>PBFT w/ client signatures</td>
<td>31777</td>
</tr>
<tr>
<td>Aardvark w/o signatures</td>
<td>57405</td>
</tr>
<tr>
<td>Aardvark w/o regular view changes</td>
<td>39771</td>
</tr>
</tbody>
</table>
Conclusion

All previous BFT (PBFT, QU, HQ, Zyzzyva) were broken under Byzantine fault

A system surviving the worst case doesn’t mean it works well. Should make it work well in worst case as well.

A small adaptation for parallelism might improve the performance a lot

A robust system should give adequate performance in any scenario
Questions?