Distributed Algorithms
Practical Byzantine Fault Tolerance

Alberto Montresor

University of Trento, Italy

2016/05/03

Acknowledgments: Lorenzo Alvisi

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.
Contents

1. Beyond PBFT
   - Overview

2. Zyzzyva
   - Introduction
   - Three cases
   - The case of the missing phase
   - View changes

3. Aardvark

4. UpRight
After PBFT, several others papers started to appear:

  In *Proc. of the Symposium on Operating systems design and implementation, OSDI’06*, Oct. 2005


The end results has been to complicate the adoption of Byzantine solutions.
“In the regions we studied (up to $f = 5$), if contention is low and low latency is the main issue, then if it is acceptable to use $5f + 1$ replicas, Q/U is the best choice, else HQ is the best since it outperforms PBFT with a batch size of 1.”

“Otherwise, PBFT is the best choice in this region: It can handle high contention workloads, and it can beat the throughput of both HQ and Q/U through its use of batching.”

“Outside of this region, we expect HQ will scale best: HQ’s throughput decreases more slowly than Q/U’s (because of the latter’s larger message and processing costs) and PBFT’s (where eventually batching cannot compensate for the quadratic number of messages).”
One protocol to rule them all!

Zyzzyva is the last word on BFT!

(Is it?)
Replica coordination

- All correct replicas execute the same sequence of commands
- For each received command $c$, correct replicas:
  - Agree on $c$’s position in the sequence
  - Execute $c$ in the agreed upon order
  - Reply to the client
How it is done now
The engineer’s Rule of thumb

Citation

*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*

Butler Lampson, “Hints for Computer System Design”
How Zyzzyva does it

![Diagram of Zyzzyva's replication process]

- Request
- Primary
- Replica 1
- Replica 2
- Replica 3
Specification for State Machine Replication

Stability
A command is stable at a replica once its position in the sequence cannot change.

Safety
Correct clients only process replies to stable commands.

Liveness
All commands issued by correct clients eventually become stable and elicit a reply.
Enforcing safety

- Safety requires:
  - Correct **clients** only process replies to stable commands

- ...but RSM implementations enforce instead:
  - Correct **replicas** only execute and reply to commands that are stable

- Service performs an output commit with each reply
Speculative BFT (Trust, but verify)

- Replicas execute and reply to a command without knowing whether it is stable
  - trust order provided by primary
  - no explicit replica agreement!

- Correct client, before processing reply, verifies that it corresponds to stable command
  - if not, client takes action to ensure liveness
Verifying stability

- Necessary condition for stability in Zyzzyva:
  - A command $c$ can become stable only if a majority of correct replicas agree on its position in the sequence

- Client can process a response for $c$ iff:
  - a majority of correct replicas agrees on $c$’s position
  - the set of replies is incompatible, for all possible future executions, with a majority of correct replicas agreeing on a different command holding $c$’s current position
History

- **History** $H_{i,k}$ is the sequence of the first $k$ commands executed by replica $i$

- On receipt of a command $c$ from the primary, replica appends $c$ to its command history

- Replica reply for $c$ includes:
  - the application-level response
  - the corresponding command history

- **Additional details:**
  - Can be hashed through **incremental hashing**
Case 1: Unanimity

- Client processes response if all replies match:

\[ r_1 = \ldots = r_4 \land H_{1,k} = \ldots = H_{4,k} \]
Case 1: Unanimity

Some comments:

- Note that although a client has a proof that the request position in the command history is irremediately set, no server has such a proof.
- Comparison of histories may be based on incremental hash.
- Three message hops to complete the request in the good case.

Is it safe to accept the reply in this case?

- All processes have agreed on ordering.
- Correct processes cannot change their mind later.
- New primary can ask $n - f$ replicas for their histories.
Case 2: A majority of correct replicas agree

Is it safe to accept such a message?
Case 2: A majority of correct replicas agree

Consider this case...
Case 2: A majority of correct replicas agree

Client sends to all a commit certificate containing $2f + 1$ matching histories
Case 2: A majority of correct replicas agree

Client processes response if it receives at least $2f + 1$ acks
Case 2: A majority of correct replicas agree

Safe?

- Certificate proves that a majority of correct processes agree on its position in the sequence
- Incompatible with a majority backing a different command for that position

Stability

- Stability depends on matching command histories
- Stability is prefix-closed:
  - If a command with sequence number $k$ is stable, then so is every command with sequence number $k' < k$
Case 3: None of the above

- Fewer than $2f + 1$ replies match
- Clients retransmits $c$ to all replicas – hinting primary may be faulty
The case of the missing phase

- Where did the third phase go?
- Why was it there to begin with?
The missing phase – COMMIT

Consider this scenario:

- $f$ malicious replicas, including the primary
- The primary stops communicating with $f$ correct replicas
- They go on strike – they stop accepting messages in this view, ask a view change
- $f + f$ replicas stop accepting messages, $f + 1$ replicas keep working
- The remaining $f + 1$ replicas are not enough to conclude the PRE-PREPARE and PREPARE phases
- The $f$ correct processes that are asking a view change are not enough to conclude one, so there is no opportunity to regain liveness by electing a new primary
The case of the missing phase

The missing phase – COMMIT

The third phase of PBFT breaks this stalemate:

- The remaining $f + 1$ replicas
  - either gather the evidence necessary to complete the request,
  - or determine that a view change is necessary

- Commit phase needed for liveness
Where the third phase go?

In PBFT

What compromises liveness in the previous scenario is that the PBFT view change protocol lets correct replicas commit to a view change and become silent in a view without any guarantee that their action will lead to the view change.

In Zyzzyva

A correct replica does not abandon view $v$ unless it is guaranteed that every other correct replica will do the same, forcing a new view and a new primary.
View change

- Two phases:
  - Processes unsatisfied with the current primary sent a message \langle i\text{-HATE\text{-THE\text{-PRIMARY}}, v} \rangle to all
  - If a process collects \( f + 1 \) i-HATE-THE-PRIMARY messages, sends a message to all containing such messages and starts a new view change (similar to the traditional one)

- Extra phase of agreement protocol is moved to the view change protocol
Optimizations

- Checkpoint protocol to garbage collect histories
- Replacing digital signatures with MAC
- Replicating application state at only $2f + 1$ replicas
- Batching
Performance

Fig. 4. Realized throughput for the 0/0 benchmark as the number of client varies for systems configured to tolerate $f = 1$ faults.
Discussion

- What have you learned?
- Do you agree on the principles?
Aardvark

Aardvark

**NSDI’09**


- A new beginning!
- http://en.wikipedia.org/wiki/File:
  Porc_formiguer.JPG

---

2 Aardvark is the first word of the English dictionary – Oritteropo in Italian
Surviving vs tolerating

Although current BFT systems can survive Byzantine faults without compromising safety, we contend that a system that can be made completely unavailable by a simple Byzantine failure can hardly be said to tolerate Byzantine faults.
Conventional wisdom

- Handle normal and worst case separately
  - remain safe in worst case
  - make progress in normal case

- Maximize performance when
  - the network is synchronous
  - all clients and servers behave correctly
Conventional wisdom

- Misguided
  - encourages systems that fail to deliver BFT

- Maximize performance when
  - the network is synchronous
  - all clients and servers behave correctly
Conventional wisdom

- Misguided
  - encourages systems that fail to deliver BFT

- Dangerous
  - it encourages fragile optimizations
Conventional wisdom

- Misguided
  - encourages systems that fail to deliver BFT

- Dangerous
  - it encourages fragile optimizations

- Futile
  - it yields diminishing return on common case
Blueprint

- Build the system around execution path that:
  - provides acceptable performance across the broadest set of executions
  - it is easy to implement
  - it is robust against Byzantine attempts to push the system away from it
**Revisiting conventional wisdom**

- Signatures are expensive — use MACs
  - Faulty clients can use MACs to generate ambiguity
  - Aardvark requires clients to sign requests

- View changes are to be avoided
  - Aardvark uses regular view changes to maintain high throughput despite faulty primaries

- Hardware multicast is a boon
  - Aardvark uses separate work queues for clients and individual replicas
  - Aardvark uses fully connected topology among replicas (separate NICs)
MAC Attack

- Primary
- Replica 1
- Replica 2
- Replica 3

- $c$
- $<c,k>$
- $<c,k>$
- $<c,k>$

✔
✔
✔
✔
MAC Attack

- Primary
- Replica 1
- Replica 2
- Replica 3

- \( c \)
- \( <c,k> \)
# Throughput

<table>
<thead>
<tr>
<th></th>
<th>Best case</th>
<th>Faulty client</th>
<th>Client flood</th>
<th>Faulty primary</th>
<th>Faulty replica</th>
</tr>
</thead>
<tbody>
<tr>
<td>PBFT</td>
<td>62K</td>
<td>0</td>
<td>crash</td>
<td>1k</td>
<td>250</td>
</tr>
<tr>
<td>QU</td>
<td>24K</td>
<td>0</td>
<td>crash</td>
<td>NA</td>
<td>19k</td>
</tr>
<tr>
<td>HQ</td>
<td>15K</td>
<td>NA</td>
<td>4.5K</td>
<td>NA</td>
<td>crash</td>
</tr>
<tr>
<td>Zyzzyva</td>
<td>80K</td>
<td>0</td>
<td>crash</td>
<td>crash</td>
<td>0</td>
</tr>
<tr>
<td>Aardvark</td>
<td>39K</td>
<td>39K</td>
<td>7.8K</td>
<td>37K</td>
<td>11K</td>
</tr>
</tbody>
</table>
Bibliography

A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche.

**UpRight cluster services.**


- A new (B)FT replication library
- Minimal intrusiveness for existing apps
- Adequate performance

Goal:
- ease BFT deployment
- make explicit incremental cost of BFT
- switching to BFT: simple change in a config file
- $u =$ max number of failures to ensure liveness
- $r =$ max number of commission failures to preserve safety
- Exposes incremental cost of BFT
  - Byzantine agreement
  - if $r << u$, $\text{BFT} \approx \text{CFT}$ in replication cost

- Allows richer design options
  - Byzantine faults are rare: $u > r$
  - Safety more critical than liveness: $r > u$
Reality Check

- UpRight\textsuperscript{3} (Java; latest update Oct. 2009)
- ArchiStar-BFT\textsuperscript{4} (Java; latest update May 2015)
- Bft-SMaRt\textsuperscript{5} (Java; latest update Apr. 2016)

\textsuperscript{3}https://code.google.com/archive/p/upright/
\textsuperscript{4}https://github.com/archistar/archistar-bft
\textsuperscript{5}http://bft-smart.github.io/library/
For (far in the) future lectures

Reading Material
