An exciting decade

State machine replication
- Practical Byzantine Fault Tolerance
- Reuse of existing (non-deterministic) implementations [SOSP 01]
- Reduced replication cost [SOSP 03]
- Low-overhead confidentiality [SOSP 03]
- High throughput [DSN 04]
- Applications: Farsite[OSDI 02], Oceanstore [FAST 03]

Quorums
- Fault Scalability (Q/U) [SOSP 05]
- Improved performance under contention (HQ) [OSDI 06]

Zyzzyva

Why then another BFT protocol?

Complex decision tree hampers BFT adoption

“Simplify, simplify”
H.D. Thoreau
“Simplify, simplify”
H.D. Thoreau

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
  - Replies to the client

One protocol that matches or tops its competitors in:
- ✔ latency
- ✔ throughput
- ✔ cost of replication

How it is done now

How Zyzzyva does it
Stability

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

**RSM Safety**
Correct clients only process replies to stable commands.

**RSM Liveness**
All commands issued by correct clients eventually become stable and elicit a reply.

Enforcing safety

- RSM 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”

- **Insight**: output commit at the client, not at the service!
- 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.
**Command History**

- $H_{i,k} =$ a hash of 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

**Case 1: Unanimity**

Client processes response if all replies match:

$$r_1 = \ldots = r_4 \land H_{1,k} = \ldots = H_{4,k}$$

---

**Safe?**

- A majority of correct replicas agrees on $c$'s position (all do!)
- If primary fails
  - New primary determines $k$-th command by asking $n-f$ replicas for their $H$

---

**Safe?**

- A majority of correct replicas agrees on $c$'s position (all do!)
- If primary fails
  - New primary determines $k$-th command by asking $n-f$ replicas for their $H$
Safe?

✓ A majority of correct replicas agrees on $c$’s position (all do!)

إجراء

If primary fails

☐ New primary determines $k$-th command by asking $n-f$ replicas for their $H$

Case 2: A majority of correct replicas agree

At least $2f+1$ replies match

Safe?

✓ A majority of correct replicas agrees on $c$’s position (all do!)

إجراء

If primary fails

☐ New primary determines $c$’s position by asking $n-f$ replicas for their $H$

✓ It is impossible for a majority of correct replicas to agree on a different command for $c$’s position
Safe?

✓ A majority of correct replicas agrees on $c$’s position

🔗 If primary fails

□ New primary determines $k$-th command by asking $n - f$ replicas for their $H$

Safe?

✓ A majority of correct replicas agrees on $c$’s position

🔗 If primary fails

□ New primary determines $k$-th command by asking $n - f$ replicas for their $H$
Safe?

✓ A majority of correct replicas agrees on c’s position

⚠️ If primary fails

□ New primary determines k-th command by asking n−f replicas for their H

Not safe!

Safe?

✓ A majority of correct replicas agrees on c’s position

⚠️ If primary fails

□ New primary determines k-th command by asking n−f replicas for their H

Safe?

✓ A majority of correct replicas agrees on c’s position

⚠️ If primary fails

□ New primary determines k-th command by asking n−f replicas for their H

Safe?

✓ A majority of correct replicas agrees on c’s position

⚠️ If primary fails

□ New primary determines k-th command by asking n−f replicas for their H
Case 2: A majority of correct replicas agree

Client sends to all a commit certificate containing $2f+1$ matching histories

Safe?

- Certificate proves that a majority of correct replicas agreed on $c$’s position
- If primary fails
  - New primary determines $k$-th command by contacting $n-f$ replicas
  - This set contains at least one correct replica with a copy of the certificate
- Incompatible with a majority backing a different command for that position

Stability and command histories

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

Fewer than \(2f+1\) replies match

Clients retransmits \(c\) to all replicas—hinting primary may be faulty

Zyzzyva recap

- Output commit at the client, not the service
- Replicas execute requests without explicit agreement
- Client verifies if response corresponds to stable command
- At most 2 phases within a view to make command stable

The Case of the Missing Phase

Client processes response if it receives at least \(f+1\) matching replies after commit phase
The Case of the Missing Phase

Pre-prepare  Prepare  Majority

Where did the third phase go?
Why was it there to begin with?

View-Change: replacing the primary

In PBFT, a replica that suspects primary is faulty goes unilaterally on strike
- Stops processing messages in the view
- Third “Commit” phase needed for liveness

In Zyzzyva, the replica goes on “Technion strike”
- Broadcasts “I hate the primary” and keeps on working
- Stops when sees enough hate mail to ensure all correct replica will stop as well

Extra phase is moved to the uncommon case
Faulty clients can’t affect safety

- Faulty clients cannot create inconsistent commit certificates
- Clients cannot fabricate command histories, as they are signed by replicas
- It is impossible to generate a valid commit certificate that conflicts with the order of any stable request
  - Stability is prefix closed!

“Olly Olly Oxen Free!”
or, faulty clients can’t affect liveness

- Faulty client omits to send CC for $c$
- Replicas commit histories are unaffected!
- Later correct client who establishes $c' > c$ is stable “frees” $c$ as well
  - Stability is prefix closed

Optimizations

- Checkpoint protocol to garbage collect histories
- Optimizations include:
  - Replacing digital signatures with MAC
  - Replicating application state at only $2f+1$ replicas
  - Batching
  - Zyzzyva5
Batching

Only one history digest for all requests in the batch—amortizes crypto operations

Speeding up the faulty case

Zyzzyva is safe despite \( f \) Byzantine faults...

...but a single Byzantine replica can force two rounds

\( \text{Zyzzyva}^5 \): one-round stability despite \( f \) non-primary Byzantine replicas

\( 5f+1 \) replicas (only \( 2f+1 \) application replicas)

\( \square \) command is stable if client receives at least \( 4f+1 \) matching replies

\( \square \) Fast despite \( e < f \) Byzantine failures with \( n > 3f+2e+1 \)

Evaluation

20 Pentium 4, 3.0 GHz, Linux 2.6, 1 Gbps LAN

Stress-tested Zyzzyva with BFT micro-benchmarks

\( \square \) Report results for no payload

\( \square \) Compared performance against:

- Unreplicated server
- PBFT[SOSP 01]
- Q/U[SOSP 03]
- HQ[OSDI 05]
Speculation improves throughput significantly.

Batching is crucial for high throughput.
Zyzzyva within 35% of unreplicated service.

Zyzzyva vs. Optimal RSM

<table>
<thead>
<tr>
<th></th>
<th>Optimal</th>
<th>Zyzzyva</th>
</tr>
</thead>
<tbody>
<tr>
<td>Replication cost</td>
<td>$3f+1$</td>
<td>$3f+1$</td>
</tr>
<tr>
<td>Total replicas</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Replication cost</td>
<td>$2f+1$</td>
<td>$2f+1$</td>
</tr>
<tr>
<td>App. replicas</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Throughput Overhead:</td>
<td>$2$</td>
<td>$2+3f/b$</td>
</tr>
<tr>
<td>Crypto. ops</td>
<td></td>
<td>$b$ a batch size</td>
</tr>
<tr>
<td>Latency</td>
<td>$3$</td>
<td>$3$</td>
</tr>
<tr>
<td>Message delays</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>
Quorum-based protocols

<table>
<thead>
<tr>
<th></th>
<th>Zyzzyva</th>
<th>Q/U</th>
</tr>
</thead>
<tbody>
<tr>
<td>Replication cost</td>
<td>$2f+1$</td>
<td>$5f+1$</td>
</tr>
<tr>
<td>App replicas</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Latency (Updates)</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Message delays</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If latency is paramount, is Q/U the better choice?

Still, Q/U is only 15% faster than Zyzzyva5!
Zyzzyva outperforms Q/U under contention and for reads
No batching increases queue delay at servers in Q/U