Building High Throughput Permissioned Blockchain Fabrics: Challenges and Opportunities

Suyash Gupta
Jelle Hellings
Sajjad Rahnama
Mohammad Sadoghi

Exploratory Systems Lab
University of California Davis
At the core of *any* Blockchain application is a Byzantine Fault-Tolerant (BFT) consensus protocol.
Practical Byzantine Fault-Tolerance (PBFT)
[OSDI’99]

• First *practical* Byzantine Fault-Tolerant Protocol.

• Tolerates up to \( f \) failures in a system of \( 3f+1 \) replicas

• Requires three *phases* of which two necessitate *quadratic* communication complexity.

• *Safety* is always guaranteed and *Liveness* is guaranteed in periods of partial synchrony.
PBFT Civil Executions

Client
Replica 1
Replica 2
Byzantine Replica

Client Request
Pre-Prepare O(n)
Prepare O(n^2)
Commit O(n^2)
Reply

T
Execute
PBFT Uncivil Execution: Primary Failure

*View Change*

Replica 1

Replica 2

Byzantine Primary

New Primary

View Change Message

New View Message

Enter New View

Got VC message from Majority?
Speculative Byzantine Fault Tolerance (Zyzzyva) [SOSP’07]

- Speculation to achieve consensus in a single phase.
- Under no failures, it only requires linear communication complexity.
- Requires good clients, for ensuring same order across the replicas.
- Clients need matching responses from all the 3f+1 replicas.
- Just one crash failure is sufficient to severely impact throughput.
- Recently, proven unsafe!
Zyzzyva Civil Executions

Client needs 3f+1 responses
Zyzyva under Failure of \textit{one} Non-Primary Replica

On client timeout $\rightarrow$ switches to \textit{slow-path}.
SBFT: A Scalable and Decentralized Trust Infrastructure [DSN’19]

- A safe alternate to Zyzzyva.
- Employs threshold signatures to linearize consensus → Splits each $O(n^2)$ phase of PBFT into two linear phases.
- Requires twin-paths → fast-path and slow-path.
- Introduces notion of collectors and executors.
Either no failures or $c+1$ crash failures for $c > 0$ collectors if $n = 3f+2c+1$
Hotstuff: BFT Consensus in the Lens of Blockchain [PODC’19]

• Splits each $O(n^2)$ phase of PBFT into two linear phases.

• Advocates leaderless consensus $\rightarrow$ Frequent primary replacement.

• Employs threshold signatures to linearize consensus $\rightarrow$ enforces sequential processing.

• Two versions:
  • Basic Hotstuff: Primary switched at the end of each consensus.
  • Chained Hotstuff: Employs pipelining to ensure each phase run by a distinct primary.
Other Proposed Byzantine-Fault Tolerant Designs

1) System consisting of $n >> 3f+1$.
   
   ➢ Q/U [SOSP’05] expects $5f+1$ replicas.

2) Use of trusted components to prevent primary equivocation.
   
   ➢ AHL [SIGMOD’19]
Novel Byzantine Fault-Tolerant Protocols
Proof-of-Execution (PoE)

Three-phase Linear protocol

Speculative Execution

Out-of-Order Message Processing

No dependence on clients or trusted component.

No reliance on a twin-path design.
## PoE vs Other Protocols

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Phases</th>
<th>Messages</th>
<th>Resilience</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZYZZYVA</td>
<td>1</td>
<td>$\mathcal{O}(n)$</td>
<td>0</td>
<td>reliable clients and unsafe</td>
</tr>
<tr>
<td><strong>POE (our paper)</strong></td>
<td>3</td>
<td>$\mathcal{O}(3n)$</td>
<td>f</td>
<td>sign. agnostic</td>
</tr>
<tr>
<td>PBFT</td>
<td>3</td>
<td>$\mathcal{O}(n + 2n^2)$</td>
<td>f</td>
<td></td>
</tr>
<tr>
<td>HOTSTUFF</td>
<td>4</td>
<td>$\mathcal{O}(n + 3n^2)$</td>
<td>f</td>
<td></td>
</tr>
<tr>
<td>HOTSTUFF-TS</td>
<td>8</td>
<td>$\mathcal{O}(8n)$</td>
<td>f</td>
<td>threshold sign.</td>
</tr>
<tr>
<td>SBFT</td>
<td>5</td>
<td>$\mathcal{O}(5n)$</td>
<td>0</td>
<td>threshold sign. and twin path</td>
</tr>
</tbody>
</table>
Proof-of-Execution (PoE)

n = 4 replicas and f <= 1
PoE View Change Protocol

Byzantine Primary → View Change Request → Join after Receiving f+1 VC requests → New View Propose → Enter New View

New Primary
Replica 1
Replica 2
PoE Scalability under Single Failure

(a) Scalability (Single Failure)

(b) Scalability (Single Failure)
Resilient Concurrency Control (RCC) Paradigm

Democracy → Give all the replicas the power to be the primary.

Parallelism → Run multiple parallel instances of a BFT protocol.

Decentralization → Always there will be a set of ordered client requests.
Why should BFT protocols rely on just one primary replica?

Malicious primary can throttle the system throughput.

Malicious primary requires replacement \(\rightarrow\) fall in throughput.
Resilient Concurrency Control Paradigm

BFT protocol

Number of Instances (z)

Parallelism
Run z parallel BFT instances.

Ordering
Create a global order of all the requests.

Execution
Execute the requests in a global order.

RCC can employ several BFT protocols: PBFT, Zyzzyva, SBFT and PoE.
RCC using PBFT with 2 parallel instances on each replica
Colluding Primaries

Multiple malicious primaries can prevent liveness!

Solution → Optimistic Recovery through State Exchange.

Good Replicas
\(|A| = f\)

Good Replicas
\(|B| = f\)

Good Replica
\(|C| = 1\)

Other \(f-2\)
Malicious Replicas

\(P_1\)
\(m_1\)

\(P_2\)
\(m_2\)
Scalability

Throughput (txn/s) vs. Number of replicas (n)

- RCC(3)
- RCC(f + 1)
- RCC(n)
- PBFT
- SBFT
- HOTSTUFF
- ZYZZYVA

Single Failure Experiments

25x
Global Scale Resilient Blockchain Fabric

- Traditional BFT protocols do not scale to geographically large distances.
- Blockchain requires decentralization \(\rightarrow\) replicas can be far apart \(\rightarrow\) expensive communication!
- The underlying BFT consensus protocol should be topology-aware.
Vision Geo-Scale Byzantine Fault-Tolerance
GeoBFT Protocol

GeoBFT is a topology-aware protocol, which groups replicas into clusters. Each cluster runs the PBFT consensus protocol, in parallel and independently.

**Local Replication**
Each cluster runs PBFT to select, locally replicate, and certify a client request.

**Inter-cluster Sharing**
Primary at each cluster shares the certified client request with other clusters.

**Ordering and Execution**
Order the certified requests, execute them, and inform local clients.
Cluster 1 $C_1$

Cluster 2 $C_2$

Local PBFT Consensus on $T_1$
GeoBFT Takeaways

• To ensure common ordering → linear communication among the clusters is required.
• Primary replica at each cluster sends a secure certificate to f+1 replicas of every other cluster.
• Certificates guarantee common order for execution.
• If primary sends invalid certificates → will be detected as malicious.
Permissioned Blockchain Through the Looking Glass: Architectural and Implementation Lessons Learned

Visit at: https://resilientdb.com/

*Proceedings of the 40th IEEE ICDCS 2020.*
Why Should You Choose ResilientDB?

1) Bitcoin and Ethereum offer low throughputs of 10 txns/s.

2) Existing Permissioned Blockchain Databases still have low throughputs (20K txns/s).

3) Prior works blame BFT consensus as expensive.

4) System Design is mostly overlooked.

5) ResilientDB adopts well-researched database and system practices.
Dissecting Existing Permissioned Blockchains

1) Single-threaded Monolithic Design
2) Successive Phases of Consensus
3) Integrated Ordering and Execution
4) Strict Ordering
5) Off-Chain Memory Management
6) Expensive Cryptographic Practices
Can a well-crafted system based on a classical BFT protocol outperform a modern protocol?
ResilientDB Architecture

SECURE LAYER
- SIGNING TOOLKIT
- HASHING TOOLKIT

EXECUTION LAYER
- THREADS
- BFT CONSENSUS
- QUEUES
- NETWORK

STORAGE LAYER
- BLOCKCHAIN
- METADATA
ResilientDB Multi-Threaded Deep Pipeline

- **Client Requests**: Input
- **Prepare & Commit**: Batch Creation
- **Checkpoint**: Worker
- **Execute**: Output

- **Network**: Message from Clients and Replicas
- **Network**: Message to Replicas and Clients

Flowchart:
- Input
- Batch Creation
- Worker
- Execute
- Output

Diagram labels:
- Messages from Clients and Replicas
- Messages to Replicas and Clients

Diagram components:
- Input
- Batch Creation
- Worker
- Execute
- Output
Insight 1: Multi-Threaded pipeline Gains

Parallelizing and Pipelining tasks across worker, execution (E) and batch-threads (B).
Insight 1: Multi-Threaded pipeline Gains

Parallelizing and Pipelining tasks across worker, execution (E) and batch-threads (B).
Insight 1: Multi-Threaded pipeline Gains

Parallelizing and Pipelining tasks across worker, execution (E) and batch-threads (B).
Insight 1: Multi-Threaded pipeline Gains

Parallelizing and Pipelining tasks across worker, execution (E) and batch-threads (B).
Insight 2: Optimal Batching Gains

More transactions batched together $\rightarrow$ increase in throughput $\rightarrow$ reduced phases of consensus.
Insight 3: Memory Storage Gains

In-memory blockchain storage → reduces access cost.
Insight 4: Number of Clients

Too many clients → increases average latency.
ResilientDB: Hands On

Visit at: https://github.com/resilientdb/resilientdb
How to Run ResilientDB?
How to Run ResilientDB?

• Go to https://github.com/resilientdb/resilientdb and Fork it!

• Install Docker-CE and Docker-Compose (Links on git)

• Use the Script “resilientDB-docker” as following:

  ./resilientDB-docker --clients=1 --replicas=4

  ./resilientDB-docker -d [default 4 replicas and 1 client]

• Result will be printed on STDOUT and stored in res.out file.
Docker CE

What is Docker?

- Run a distributed program on one machine
- Simulate with lightweight virtual machines
How to Run ResilientDB?

• Go to https://github.com/resilientdb/resilientdb and Fork it!

• Install Docker-CE and Docker-Compose (Links on git)

• Use the Script "resilientDB-docker" as following:

  ./resilientDB-docker --clients=1 --replicas=4

  ./resilientDB-docker -d [default 4 replicas and 1 client]

• Result will be printed on STDOUT and stored in res.out file.
Resilient DB

./resilientDB-docker -d

• Remove old Containers
• Create new Containers
• Create IP address settings
• Install dependencies
• Compile Code
• Run binary files
• Gather the results
Resilient DB

- Throughput
  - Transaction per second

- Average Latency
  - The time from client request to client reply

- Working Thread idleness
  - The time that thread is waiting

- WT0: Consensus Messages
- WT1 and WT2: Batch Threads
- WT3: checkpointing Thread
- WT4: Execute Thread
Configuration Parameters to Play

- **NODE_CNT**: Total number of replicas, minimum 4, that is, $f=1$.
- **THREAD_CNT**: Total number of threads at primary (at least 5).
- **CLIENT_NODE_CNT**: Total number of clients (at least 1).
- **MAX_TXN_IN_FLIGHT**: Multiple of Batch Size.
- **DONE_TIMER**: Amount of time to run the system.
- **BATCH_THREADS**: Number of threads at primary to batch client transactions.
- **BATCH_SIZE**: Number of transactions in a batch (at least 10).
- **TXN_PER_CHKPT**: Frequency at which garbage collection is done.
- **USE_CRYPTO**: To switch on and off cryptographic signing of messages.
- **CRYPTO_METHOD_ED25519**: To use ED25519 based digital signatures.
- **CRYPTO_METHOD_CMAC_AES**: To use CMAC + AES combination for authentication.
PBFT: Practical Byzantine Fault Tolerance

Main Functions

- Client/client_main.cpp
- System/client_thread.cpp
- System/main.cpp
PBFT: Practical Byzantine Fault Tolerance

Process Messages

- Transport/message.cpp
- System/worker_thread.cpp
- System/worker_thread_pbft.cpp
- Worker Thread: Run function
- Worker Thread: Process function
PBFT Failure-Free Flow

Client
Primary
Replica 1
Replica 2
Byzantine Replica

Client Request
Pre-Prepare $O(n)$
Prepare $O(n^2)$
Commit $O(n^2)$
Reply

$T$
PBFT: Practical Byzantine Fault Tolerance

Process Client Message

- System/worker_thread_pbft.cpp
- process_client_batch Function
- Create and Send Batch Request
  - create_and_send_batchreq Function
  - Create Transactions
  - Create Digest
- BatchRequest Class
  - Pre-Prepare Message
PBFT Failure-Free Flow

Client
Primary
Replica 1
Replica 2
Byzantine Replica

Client Request
Pre-Prepare O(n)
Prepare O(n²)
Commit O(n²)
Reply

Preparation:
- Pre-prepare: $O(n)$
- Prepare: $O(n^2)$
- Commit: $O(n^2)$

Decision:
- Client
- Primary
- Replica 1
- Replica 2
- Byzantine Replica

Stage T
PBFT: Practical Byzantine Fault Tolerance

Process Batch Request (Prepare)

- System/worker_thread_pbft.cpp
- process_batch Function
- Create and Send Prepare Message
  - Create Transactions
  - Save Digest
- PBFTPrepare Class
  - Prepare Message
PBFT Failure-Free Flow

Client -> Primary

Pre-Prepare: O(n)

Prepare: O(n^2)

Commit: O(n^2)

Byzantine Replica

T

Client Request

Pre-Prepare

Prepare

Commit

Reply
PBFT: Practical Byzantine Fault Tolerance

Process Prepare and Commit Messages (Prepare)

- System/worker_thread_pbft.cpp
- process_pbft_prepare Function
  - Count Prepare Messages
  - Create and Send commit Message
  - PBFTCommit Message
- process_pbft_commit Function
  - Count commit messages
  - Create and Send execute Message
  - ExecuteMessage Class

```c
// worker_thread_pbft.cpp

void *worker_thread_pbft(void *param) {
  // ... (code snippet)
}
```

```c
// worker_thread_pbft.cpp

void create_and_send_commit_message(uint32_t txid) {
  // ... (code snippet)
}
```

```c
// worker_thread_pbft.cpp

void execute_message(uint32_t txid) {
  // ... (code snippet)
}
```
PBFT: Practical Byzantine Fault Tolerance

Process Execute Message

- System/worker_thread.cpp
- Internal Message
- process_execute Function
- Execute the Transactions in batch in order
- Create and send Client Response
- ClientResponse Class
PBFT: Practical Byzantine Fault Tolerance

Work Queue

• Lock Free queues
• All the messages are being stored in these queues
• System/work_queue.cpp
• Multiple queues for different Threads
• Dequeue and Enqueue Interfaces
• Enqueue in IOTThread
• Dequeue in Worker Thread
PBFT: Practical Byzantine Fault Tolerance

IO Thread and Transport Layer

- Multiple Input Threads
- Multiple Output Threads
- System/io_thread.cpp
- Transport Layer: TCP Sockets
- Nano Message Library
- Transport/transport.cpp

```cpp
RC InputThread::server_recv_loop()
{
    myrand rdm;
    rdm.init(get_thd_id());
    RC rc = RCOK;
    assert(rc == RCOK);
    uint64_t starttime = 0;
    uint64_t idle_starttime = 0;
    std::vector<Message> msgs;
    while (!simulation->is_done())
    {
        heartbeat();
        #if VIEW_CHANGES
        if (g_node_id != get_current_view(get_thd_id()))
        {
            uint64_t tid = get_thd_id() - 1;
            uint32_t nchange = get_newView(tid);
            if (nchange)
            {
                set_current_view(get_thd_id(), get_current_view(get_thd_id()) + 1);
                set_newView(tid, false);
            }
        }
        #endif
        msgs = tport_man.recv_msg(get_thd_id());
    }
}
```
Thank You