An In-Depth Look of BFT Consensus in Blockchain: Challenges and Opportunities

(System)

Suyash Gupta, Jelle Hellings,
Sajjad Rahnama, Mohammad Sadoghi

ExpoLab Creativity Unfolded
ResilientDB

UCDAVIS
UNIVERSITY OF CALIFORNIA

Agenda

Session I

1) Blockchain 101
    1) What is Blockchain, Applications and Components?
    2) Permissionless and Permissioned Blockchain.
2) Transactions and Consensus
3) Primer on Byzantine Fault-Tolerant Consensus
4) Existing Optimizations for BFT Consensus.

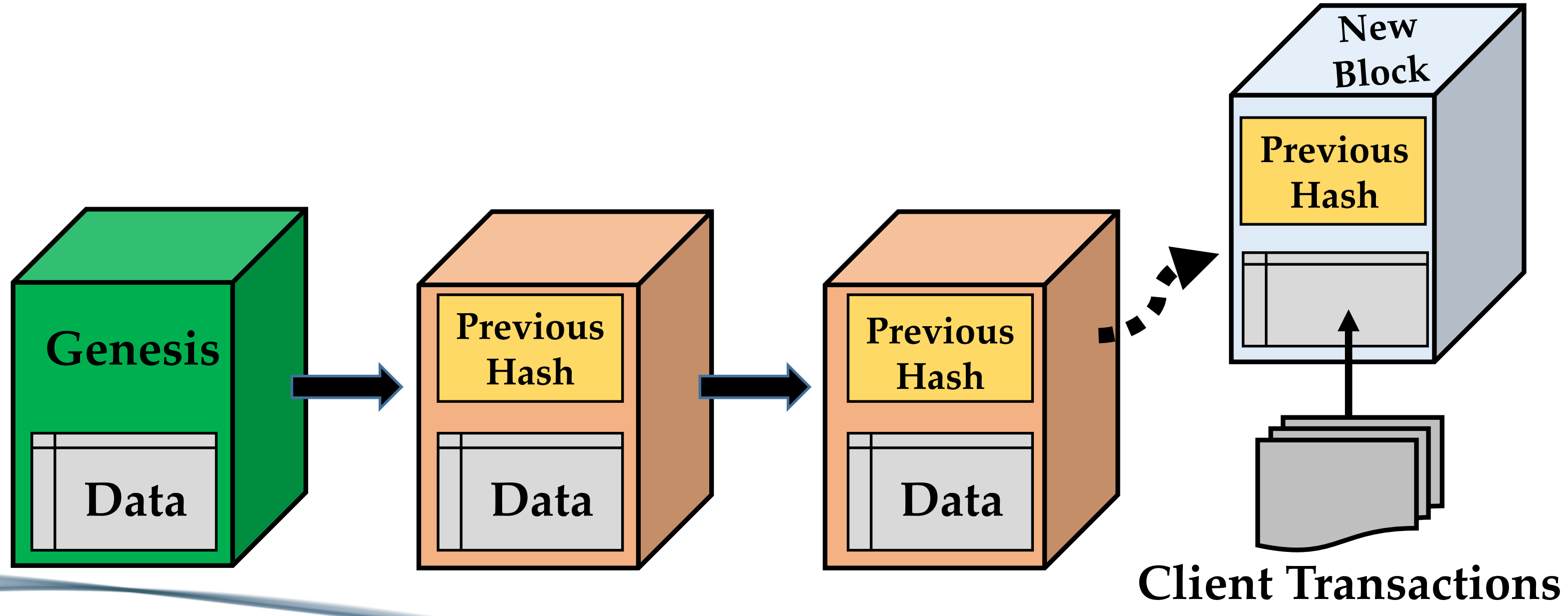
Session II

1) PoE: Two-Phases Resilient Consensus.
2) MultiBFT: Parallel and Wait-free Consensus
3) GeoBFT: Global Scale Consensus
4) Reducing Communication between Clusters

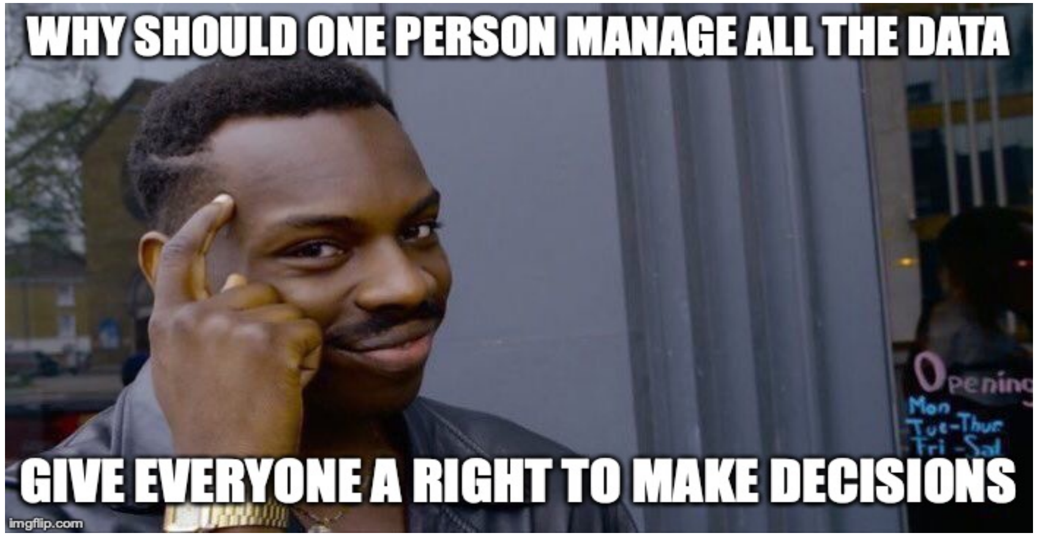
Hands-on ResilientDB

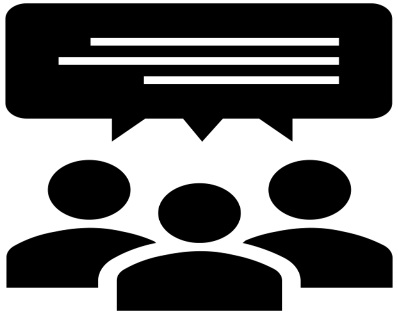
What is Blockchain?

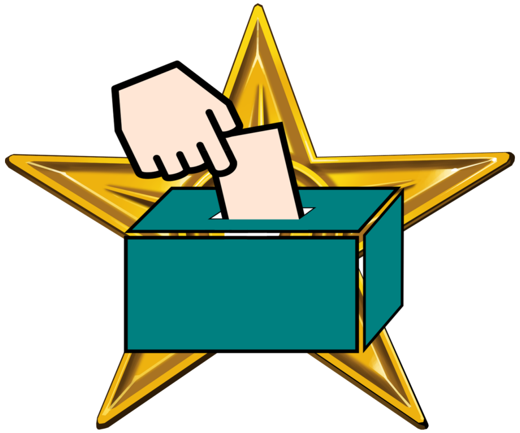
- A linked list of blocks.
- Each block contains hash of the previous block.
- A block contains information about some client transactions.



Why Blockchain?

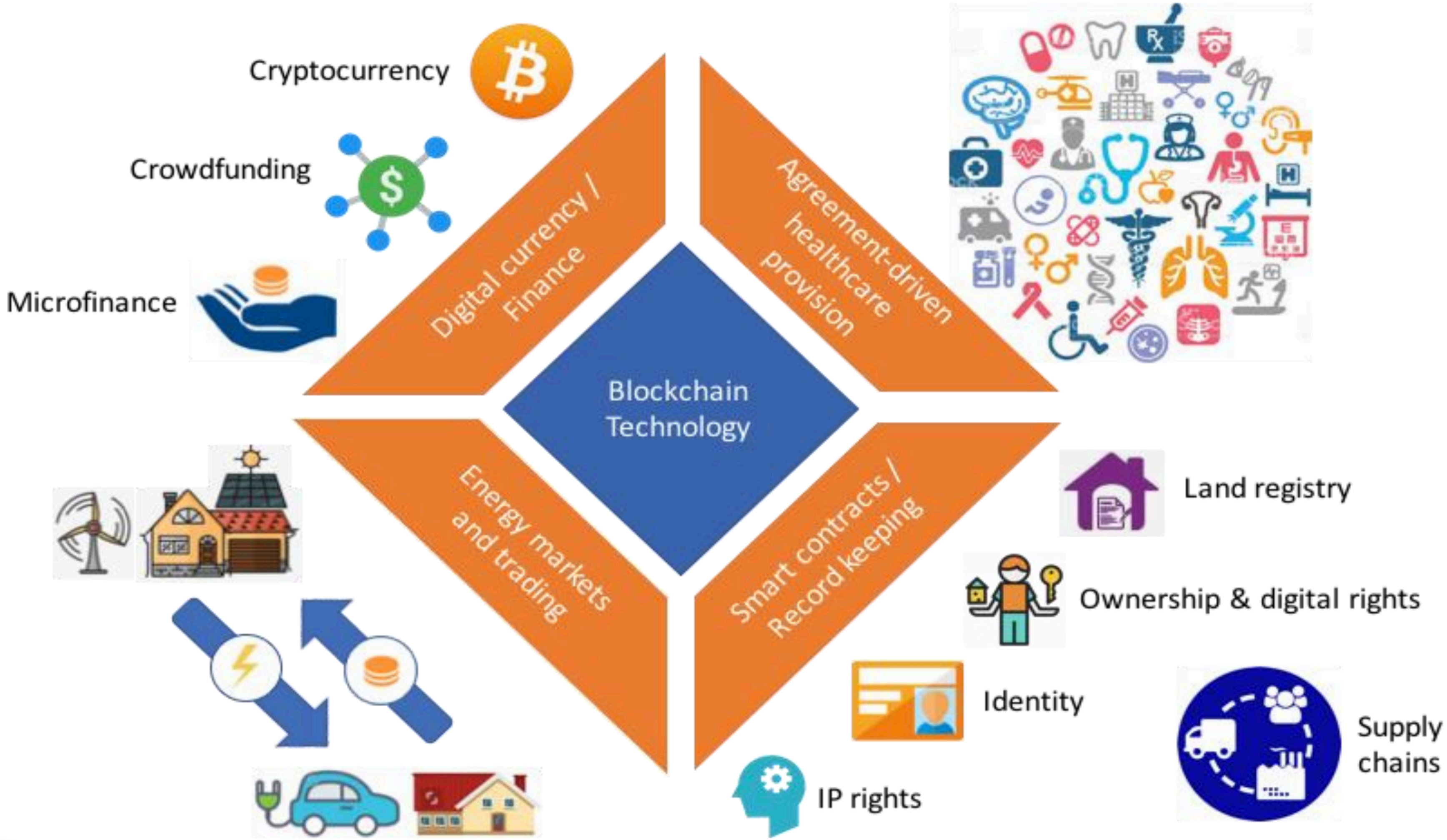






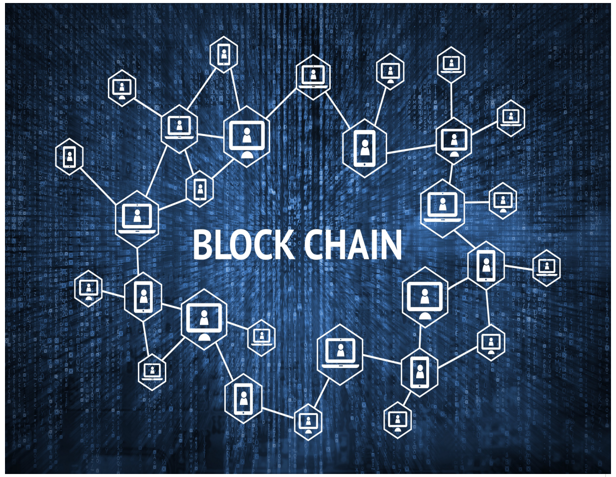
- By User:Pedant, User:Wapcaplet, User:Antonu, User:Vanderlindenma, User:.js. - Composition of File:Barnstar of Diligence Hires.png + File:Voting hand.svg., CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=45960536
- https://blog.devolutions.net/2017/10/whats-the-difference-between-2fa-and-mfa

Blockchain Applications?



- Jacobsen et al., Blockchain Landscape and AI Renaissance: The Bright Path Forward. Middleware (Tutorials) 2018: 2:1

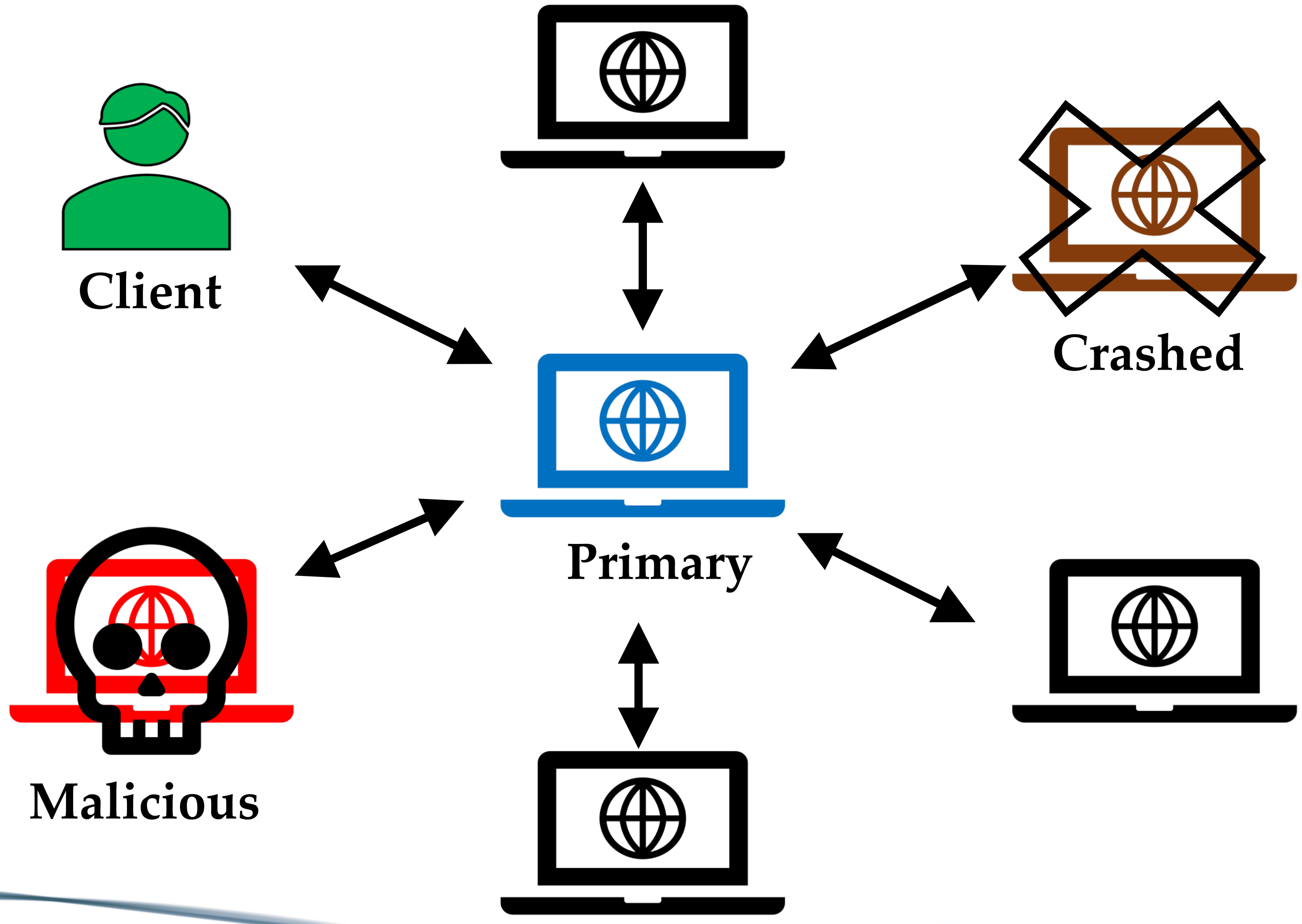
Components of a Blockchain System



- Replicas → Store all the data.
- Client → Sends transactions to process.
- Consensus Protocol → Helps ordering transactions.
- Cryptographic Constructs → Authenticate replicas and clients.
- Ledger → Records transactions.

- https://medium.com/@blake_hall

Consensus



Types of Blockchain Systems

- **Permissionless → Open Access**
  - Anyone can participate.
  - Identities of the replicas unknown.
  - Applications include crypto-currency and money exchange.
- **Permissioned → Restricted Access**
  - Only a selected group of replicas, although untrusted can participate.
  - Identities of the replica known a priori.
  - Applications include health-care and energy trading.

BITCOIN

- First Crypto-currency → a monetary application.
- Uses Nakamoto consensus → Proof-of-Work beneath the skin.
- Supports permissionless access.
- Requires solving hard cryptographic puzzles.
- Any replica that wants to create a new block proves that it did solve the puzzle.
- Difficulty of the puzzle helps prevent malicious attacks.

ETHEREUM

- Another Crypto-currency → a token used in variety of applications.
- Uses Proof-of-Work but plans to start using Proof-of-Stake.
- Supports permissionless access.
- Allows programmers to design their transactions or “*smart contracts*”.
- Hard dependency on Ethereum Virtual machine (EVM).
- Envisions design of Permissioned applications.

The New York Times

Terrorists Turn to Bitcoin for Funding, and They're Learning Fast

TC

Login

Search

Disrupt Berlin 2019

Binance says more than $40 million in bitcoin stolen in 'large scale' hack

Zack Whittaker, Catherine Shu / 6:10 pm PDT • May 7, 2019

Comment

THE VERGE TECH REVIEWS SCIENCE CREATORS ENTERTAINMENT VIDEO FEATU

REPORT \ TECH \ CYBERSECURITY

Why the Ethereum Classic hack is a bad omen for the blockchain

The 51 percent attack is real, and it's easier than ever

By Russell Brandom | Jan 9, 2019, 8:47am EST

COINTELEGRAPH
The future of money

BTC	ETH	XRP	BCH	LTC
$7,424	$152	$0.22	$216	$47.98
-1.98%	+0.03%	-0.75%	-1.00%	+1.24%

News Features Price Analysis Market Tools Cryptopedia Industry

By William Suberg NOV 16, 2019

Bitcoin Cash Hard Fork Sees Miners 'Waste' Money on 14 Invalid Blocks

coindesk

Bitcoin 24h	Ethereum 24h	XRP 24h
$7,355.40 -2.58%	$151.13 -0.62%	$0.223687

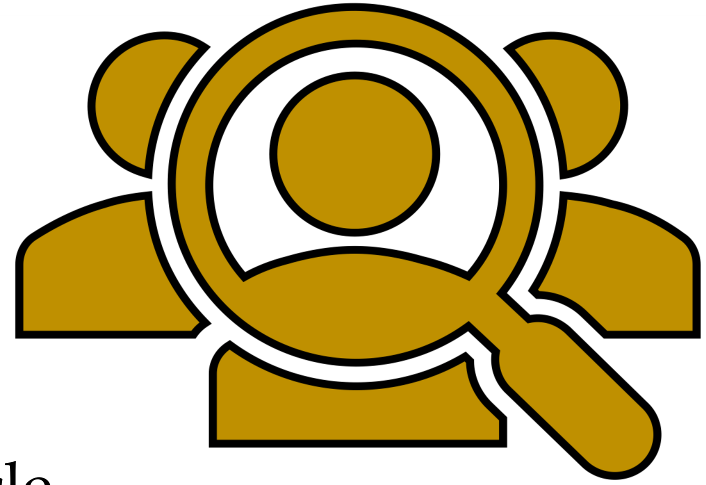
Story from Tech →

Bitcoin Cash Miners Undo Attacker's Transactions With '51% Attack'

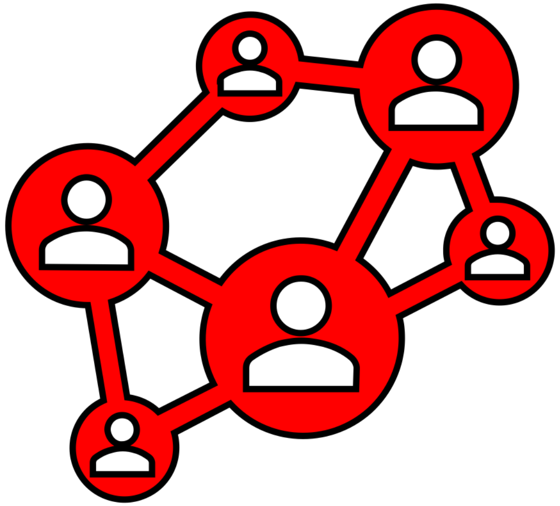
May 24, 2019 at 21:17 UTC • Updated May 25, 2019 at 10:39 UTC

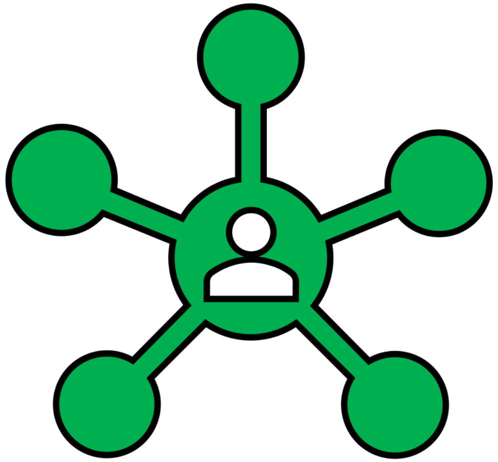
Permissioned Blockchain Systems

- Require identities of the participating replicas to be known a priori.
- Replicas still untrusted → Consensus through traditional BFT protocols.
- Computationally in-expensive.
- Communication intensive.
- Prevent chain forks.
- Suitable for needs of an industry → JP Morgan, IBM, Oracle
- Advent path for *Blockchain Databases.*



Transactions, Agreement and Consensus





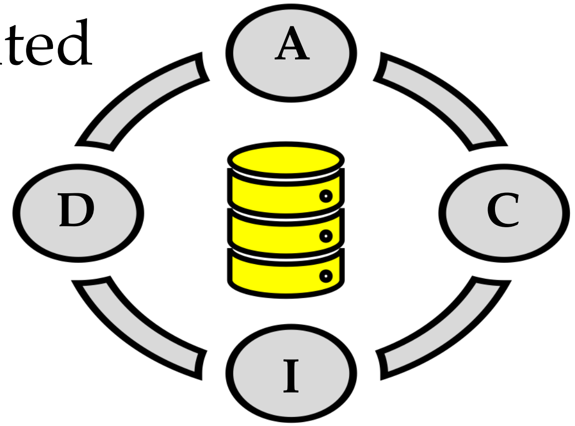
ResilientDB ExpoLab Creativity Unfolded

The Omniscient Transaction

- A transformation from a *consistent* state to another consistent state.
- A *contract* between two or more parties.
- A collection of *Read* or *Write* operations.
- Types of transactions: nested, compensating, multi-operation etc.

ACID Properties

- **Atomicity**: A transaction either completes fully or none of its changes take place.
- **Consistency:** The transaction must obey legal protocols
- **Isolation:** The intermediate state of a transaction is invisible to other transactions
- **Durability**: Once a transaction is committed, it cannot be abrogated

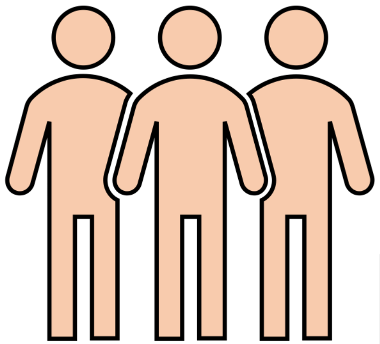


Consistency vs Availability

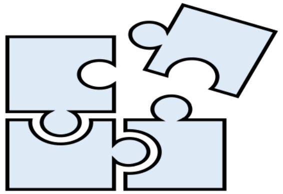
- An ongoing struggle that causes *performance tradeoffs*.
- **Availability** → Database needs to be always available for use.
  - Solution? Replication
  - Issues? Faults, Failures and Attacks.
- **Consistency** → Database needs to be correct.
  - Solution? All replicas should have same state.
  - Issues? Expensive.

- By Ryan Child - http://www.navy.mil/view_image.asp?id=24509, Public Domain, https://commons.wikimedia.org/w/index.php?curid=182472

Partitioning vs Replicating



- Distributed Databases can be partitioned, replicated or both.
- Partitioning → Split database into multiple disjoint partitions.
- Replication → Multiple full copies of the database.
- Partitioned Replication → Multiple partitions, where each partition employs replication.



Agreement in Partitioned Databases

- Partitioned Databases receive client transactions that may access multiple partitions.
- Deciding the fate of *multi-partition transactions* requires coordination among the partitions.
- *Coordination* is costly but necessary.
- Coordination or agreement among the partitions should be both *safe* and *live*.

A Deep Dive into BFT Consensus (Theory Slides Continue)

ExpoLab
Creativity Unfolded

ResilientDB

UCDAVIS
UNIVERSITY OF CALIFORNIA

ResilientDB ExpoLab Creativity Unfolded

Modern BFT Consensus Optimizations

ExpoLab Creativity Unfolded

ResilientDB

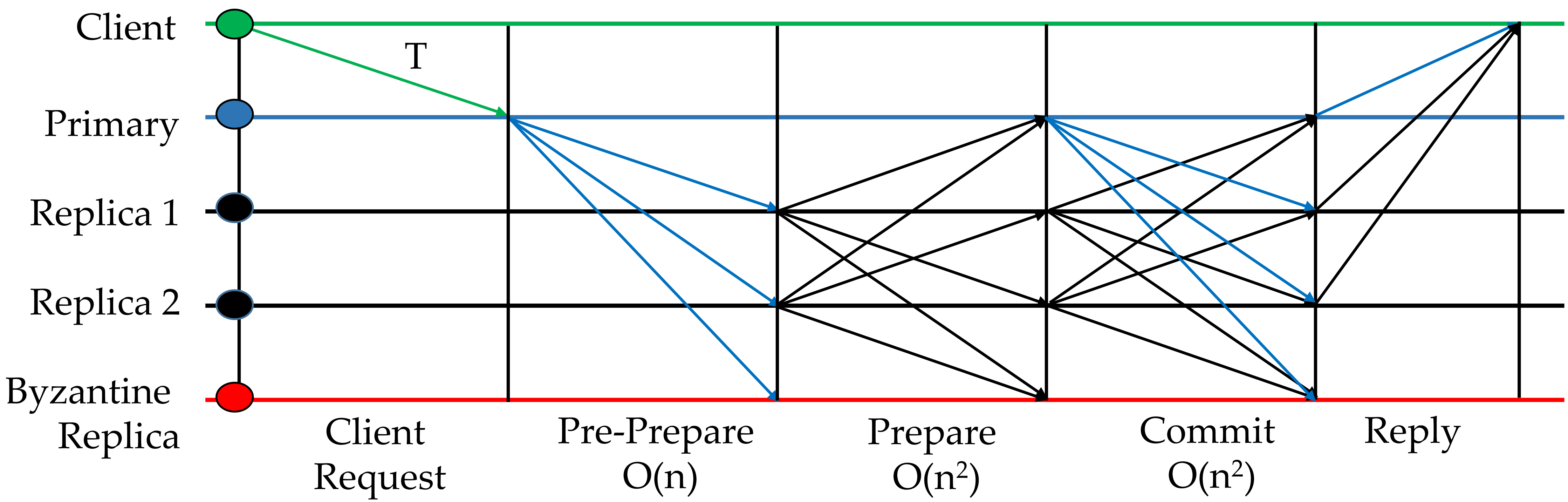
UC DAVIS
UNIVERSITY OF CALIFORNIA

ResilientDB ExpoLab Creativity Unfolded

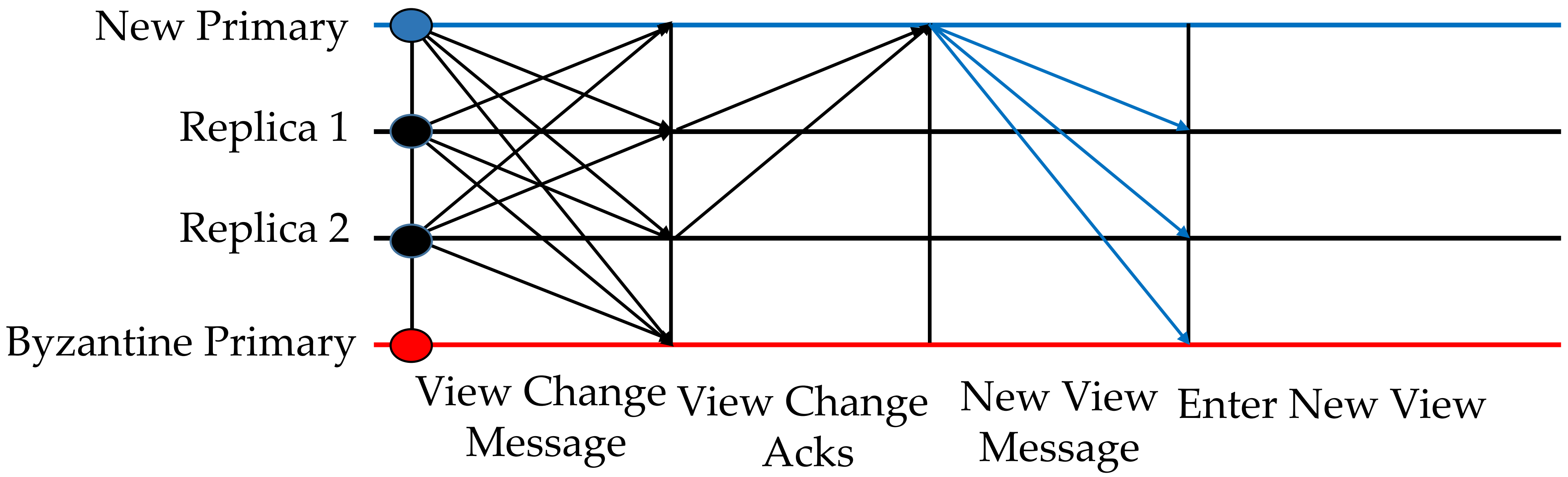
PBFT: Practical Byzantine Fault Tolerance

- First practical Byzantine Fault Tolerant Protocol.
- Tolerates up to **f** failure out of 3f+1 replicas
- Three phases of which two require quadratic communication complexit.
- Safety is always guaranteed and Liveness is guaranteed in periods of partial synchrony.
- View-Change protocol for replacing malicious primary

PBFT Failure-Free Flow



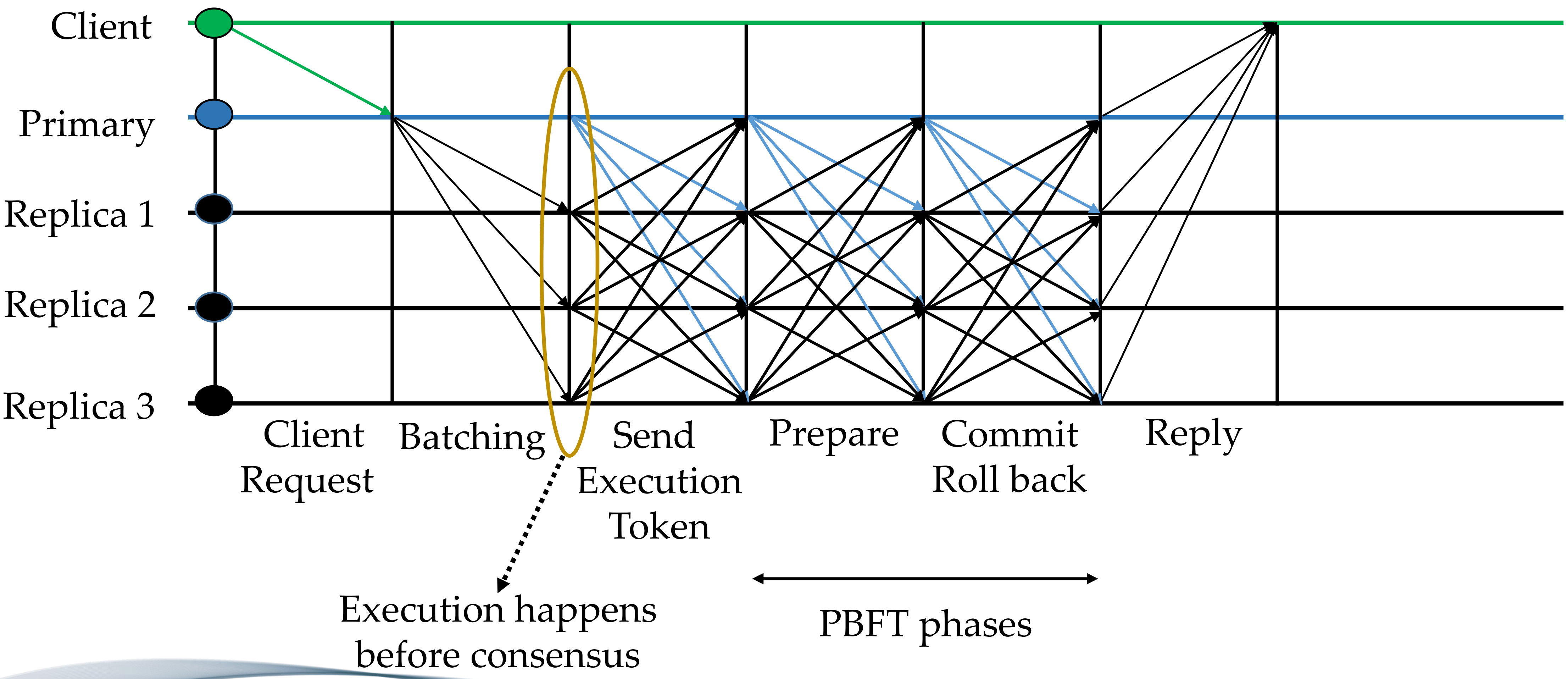
PBFT Primary Failure (View Change)



All about Eve: Execute-Verify Replication for Multi-Core Servers [OSDI'12]

- **New Architecture:** **Execute/verify** instead of Agree/Execute.
- Execute multiple requests concurrently and then verify the output.
- Takes advantage of **parallel hardware** to improve performance.
- Non-deterministic multi-threaded execution
- The **Byzantine agreement is on the output** instead of sequence.
- Allows **divergence** in execute and roll back in case

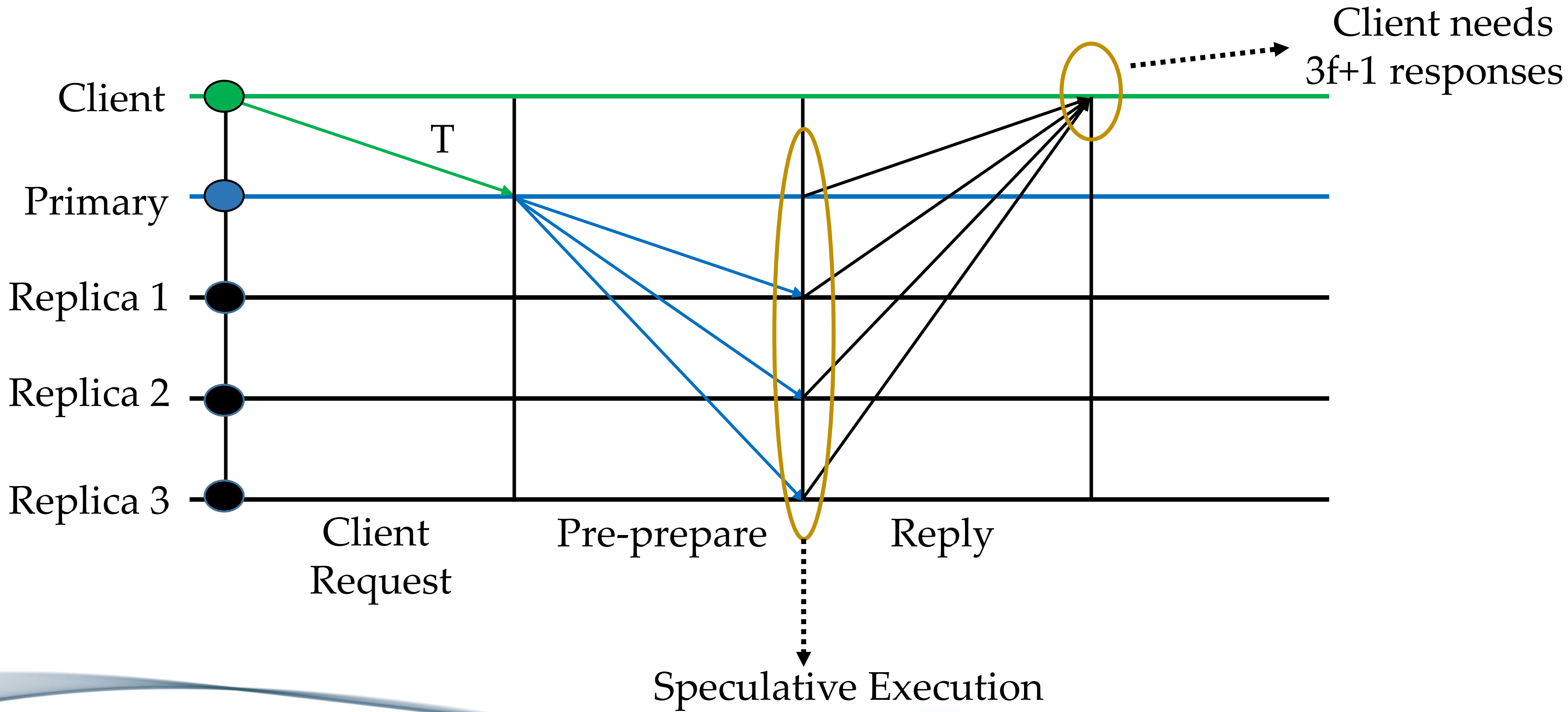
Eve’s Execute-Verify Flow



Zyzzyva: Speculative Byzantine Fault Tolerance [SOSP'07]

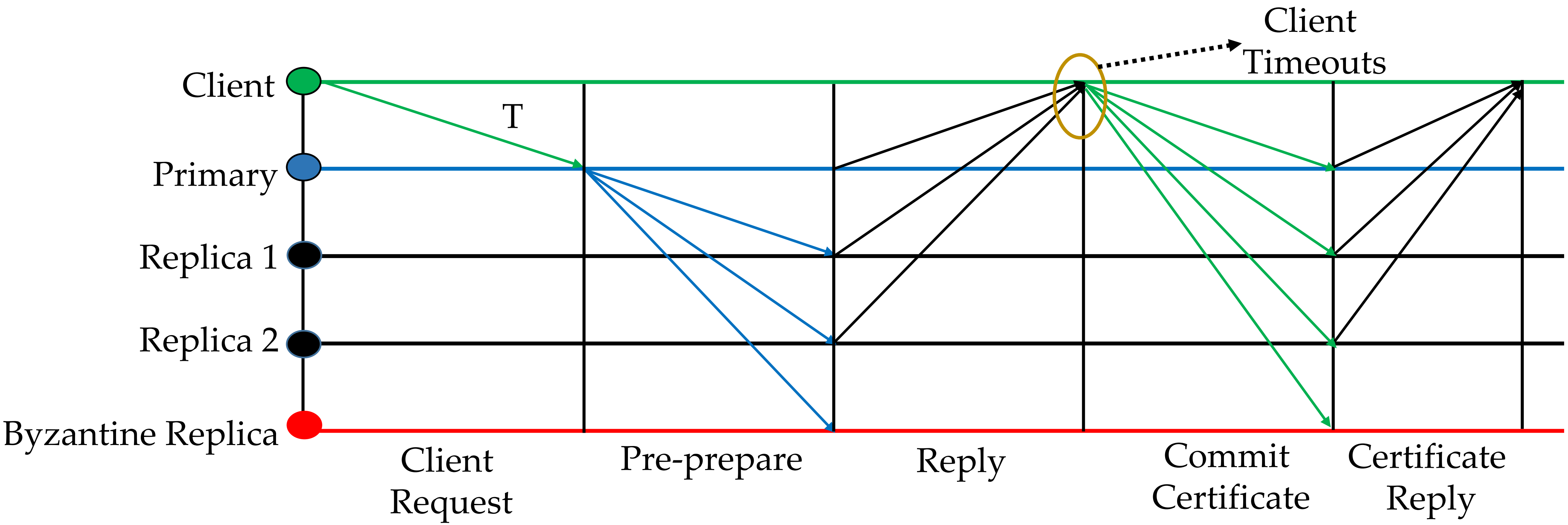
- Employs Speculation to achieve consensus in a single phase.
- In the best case (no failures), it only requires linear communication complexity.
- Depends on its good clients, for achieving common order among the replicas.
- Client needs identical response from all the 3f+1 replicas.
- With just one crash fault it faces severe throughput degradation.
- Recently, proven unsafe.

Zyzzyva Failure-Free Flow

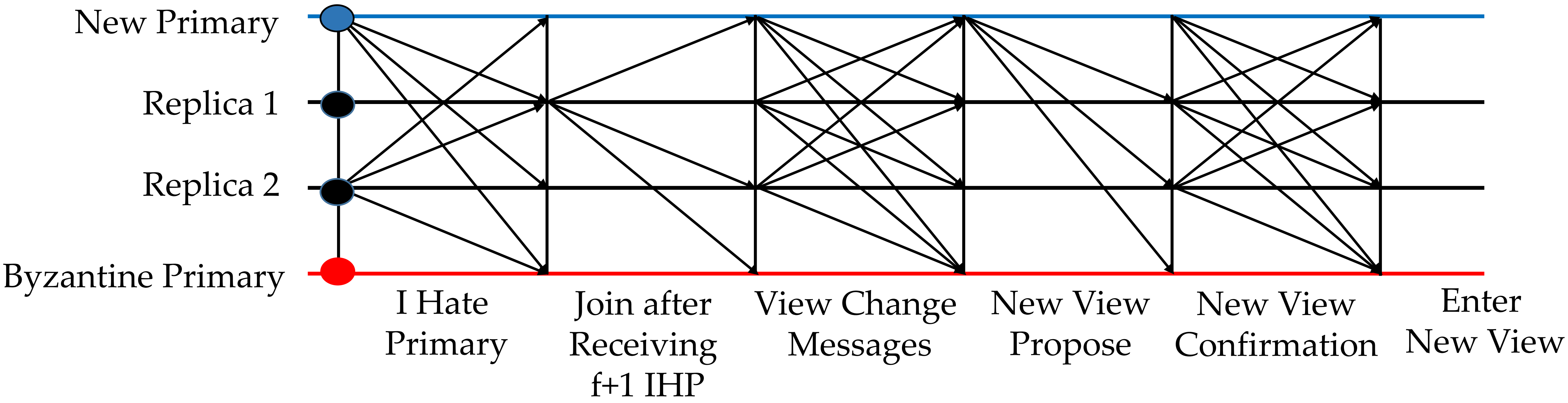


ResilientDB ExpoLab Creativity Unfolded

Zyzzyva Flow with the Failure of One Non-Primary Replica



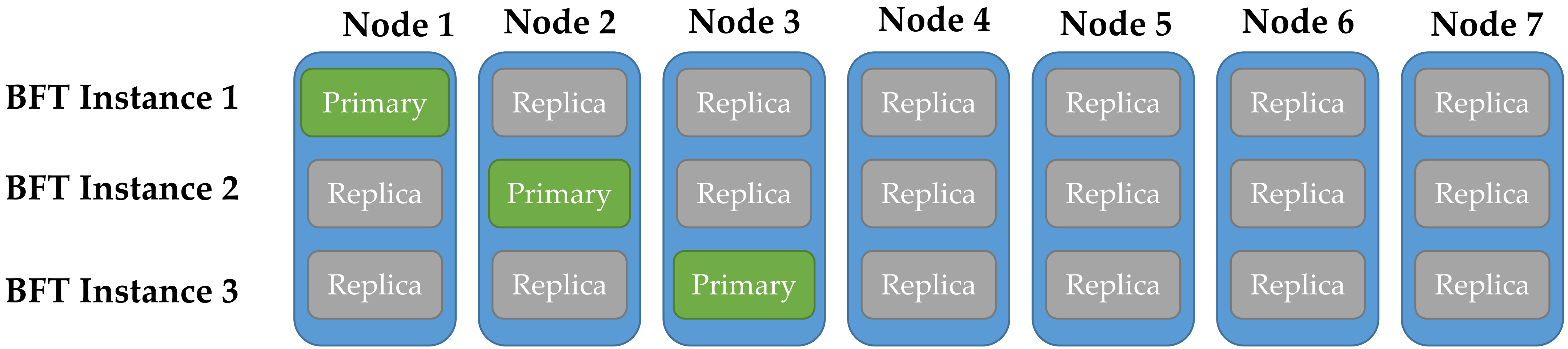
Zyzzyva View Change Protocol



RBFT: Redundant Byzantine Fault Tolerance [ICDCS'13]

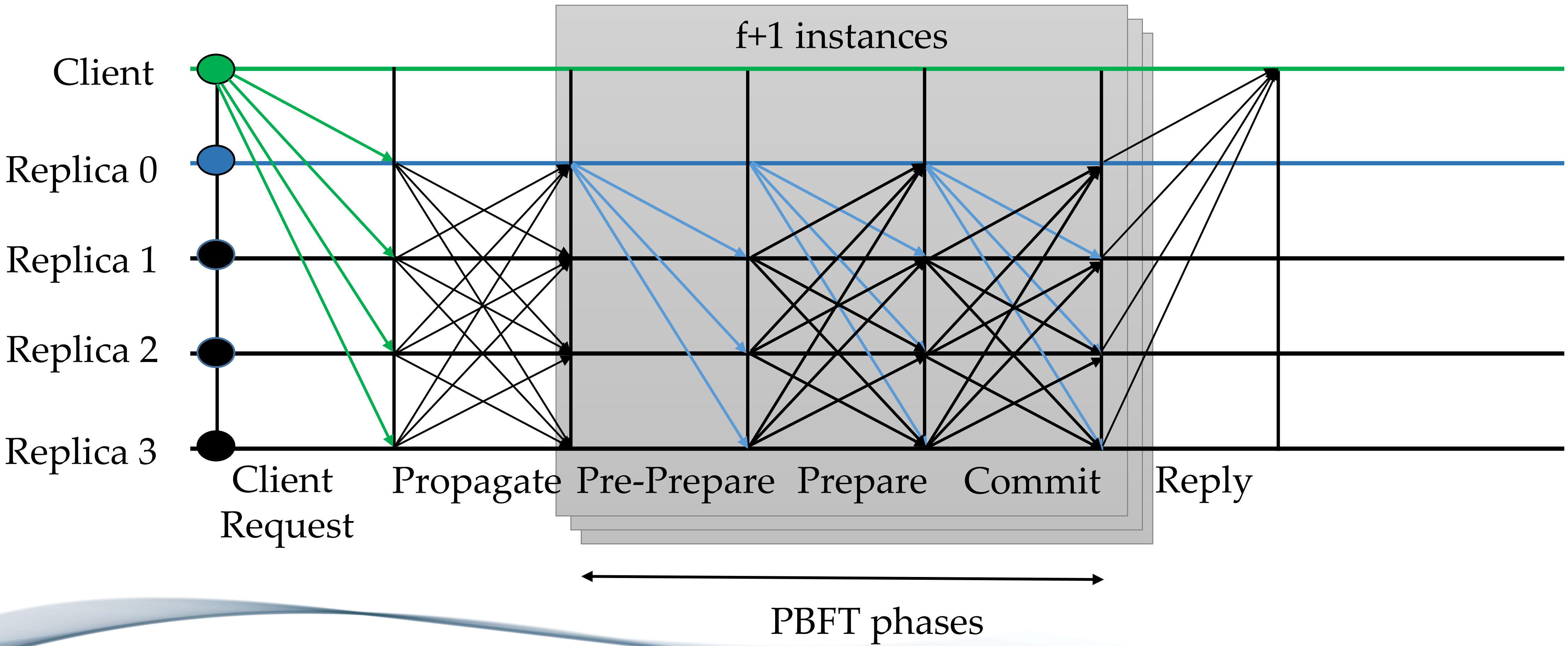
- **Robust** BFT protocol, perform well in the presence of smart malicious primary.
- Runs **f+1 instance of BFT** protocol to monitor best performance .
- Do not rely on **one specific primary**.
- One **Master primary** and **f** backup instance of protocol always being executed.
- **Goal**: replicas monitor the throughput of the primary and replace it with one of the backups when it is slow to achieve robustness

RBFT's Multiple Redundant Primary Design



n=7 replicas, f=1 and f+1=3 instances

RBFT Failure-Free Flow

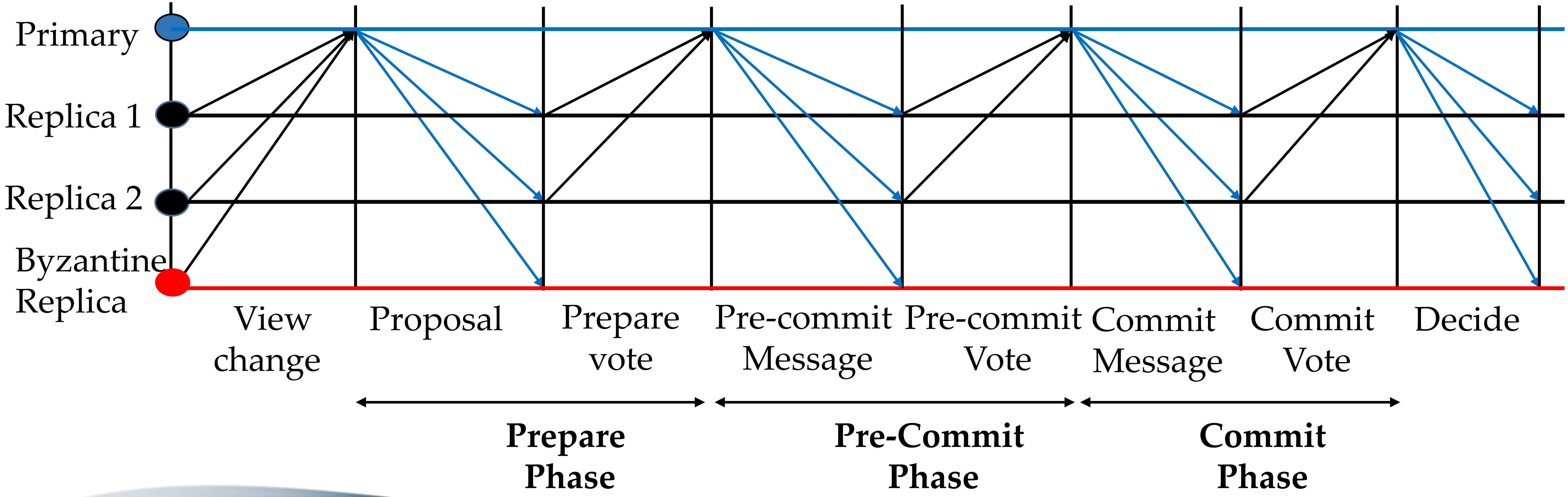


Hotstuff: BFT Consensus in the Lens of Blockchain [PODC'19]

- Splits each O(n^2) phase of PBFT into two linear phases.
- Changes leader at the end of each consensus.
- Employs expensive threshold signatures to linearize consensus.
- **Two versions:**
  - **Basic Hotstuff:** Unfolding each phase of PBFT into two round and make it linear.
  - **Chained Hotstuff:** Pipelined version of basic one, each phase different role for different view.

ResilientDB ExpoLab Creativity Unfolded

Hotstuff Protocol Flow



Attested Append-Only Memory: Making Adversaries Stick to their Word [SOSP'07]

- Uses a trusted component to reduce the hard-limit of one-third byzantine failures.
- Trusted component removes equivocation → Primary cannot lie about the order.
- A2M → Set of trusted, undeniable, ordered logs.
- Messages in the log can be verified by everyone using attestation.
- Attestation overheads: log writing, verifying.

Attested Append-Only Memory

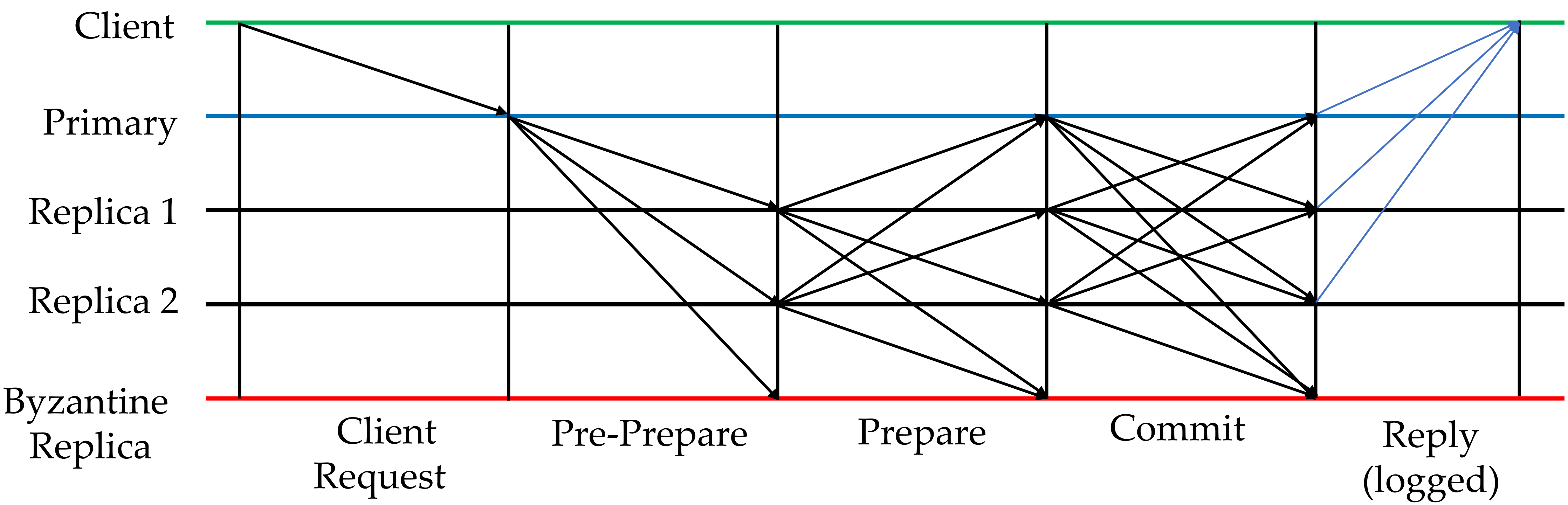
A2M-PBFT-E

- Protecting the execution by adding attestation to client reply
- Safety and liveness when $f<n/3$
- Just Safety when $f<n/2$

A2M-PBFT-EA

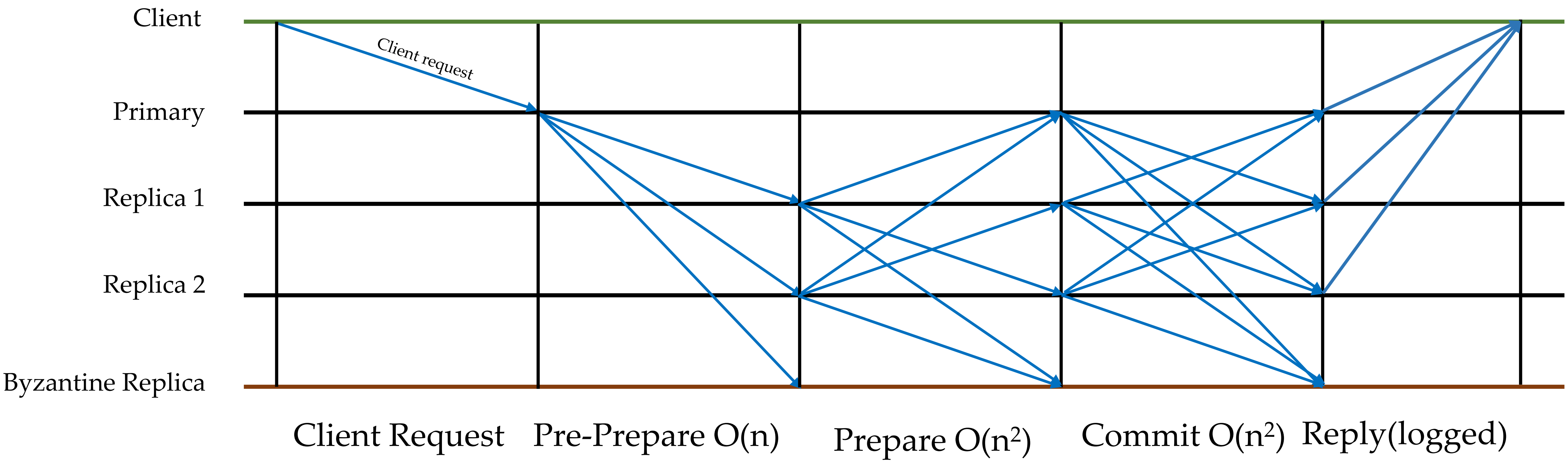
- Append all messages to the A2M Log
- Protect Execution and Agreement with log
- Safety and liveness when $f<n/2$

Attested Append-Only Memory: PBFT with Execution Protection



Attested Append-Only Memory:
Making Adversaries Stick to their Word

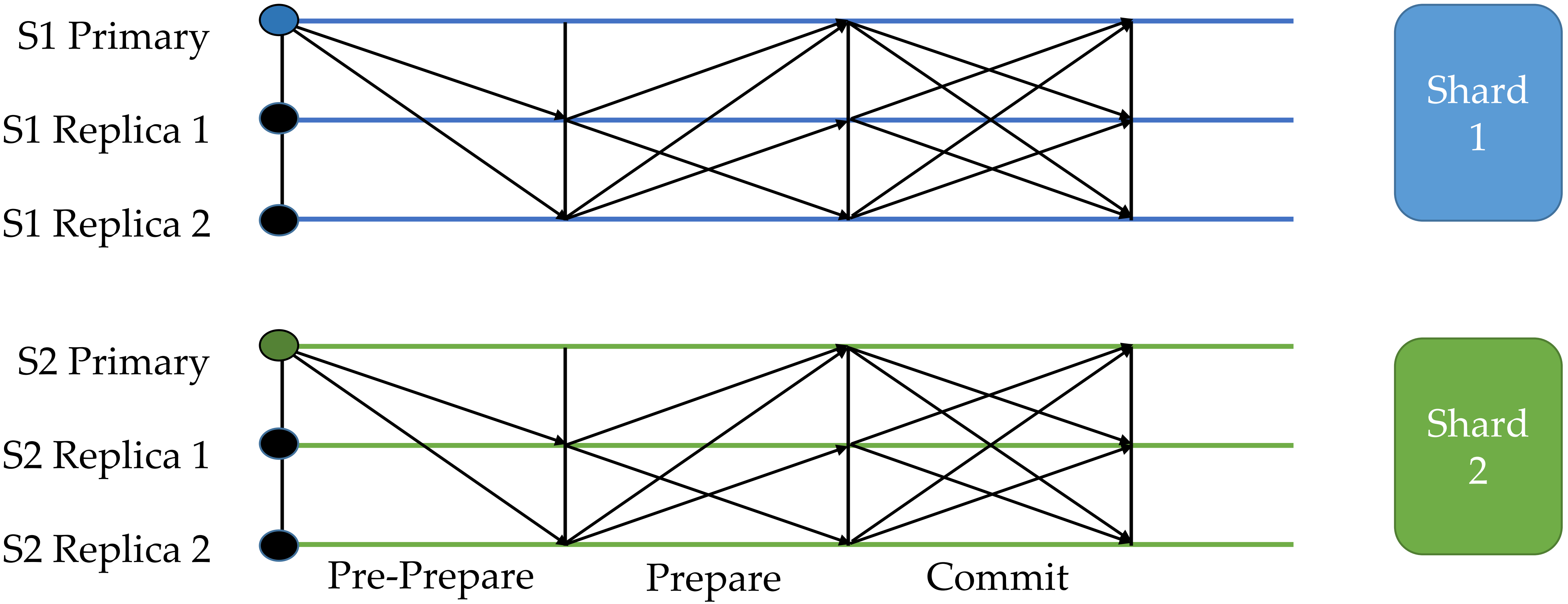
Attested Append Only Memory PBFT with Execution and Agreement Protection:



Towards Scaling Blockchain Systems via Sharding [SIGMOD'19]

- Introduces the notion of multiple chains.
- Data is partitioned into shards → Each shard uses 3f+1 replication.
- PBFT within each shard to provide Byzantine Fault-Tolerance.
- Multi-shard transactions require Two-Phase Commit protocol.
- Authors use SGX Trusted Hardware to reduce costs (n =2f+1)

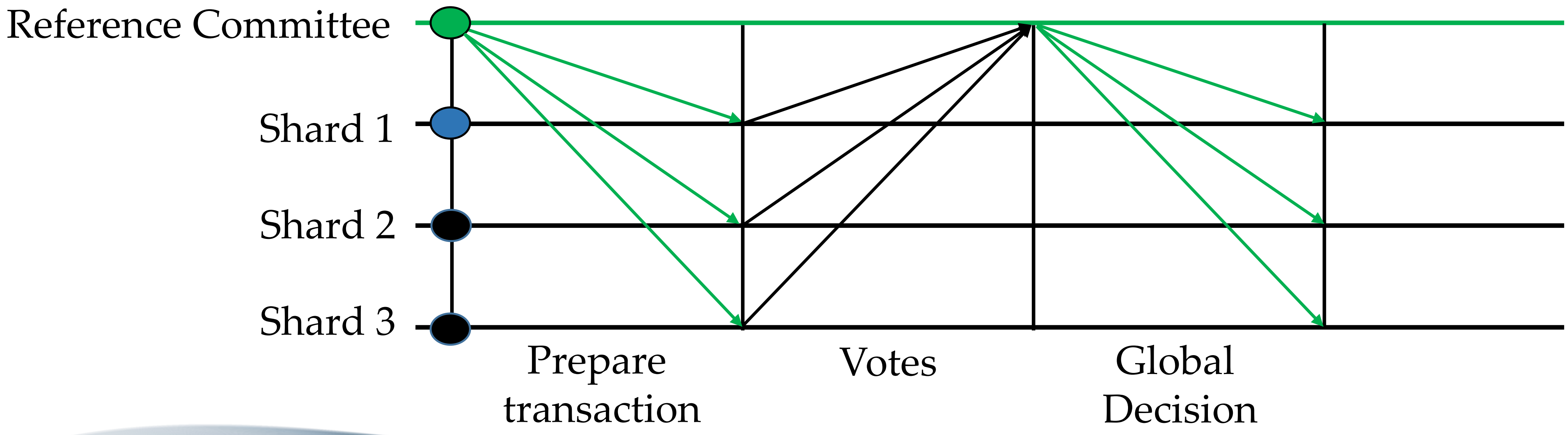
Towards Scaling Blockchain Systems via Sharding



Single-shard transactions are fast!

Towards Scaling Blockchain Systems via Sharding

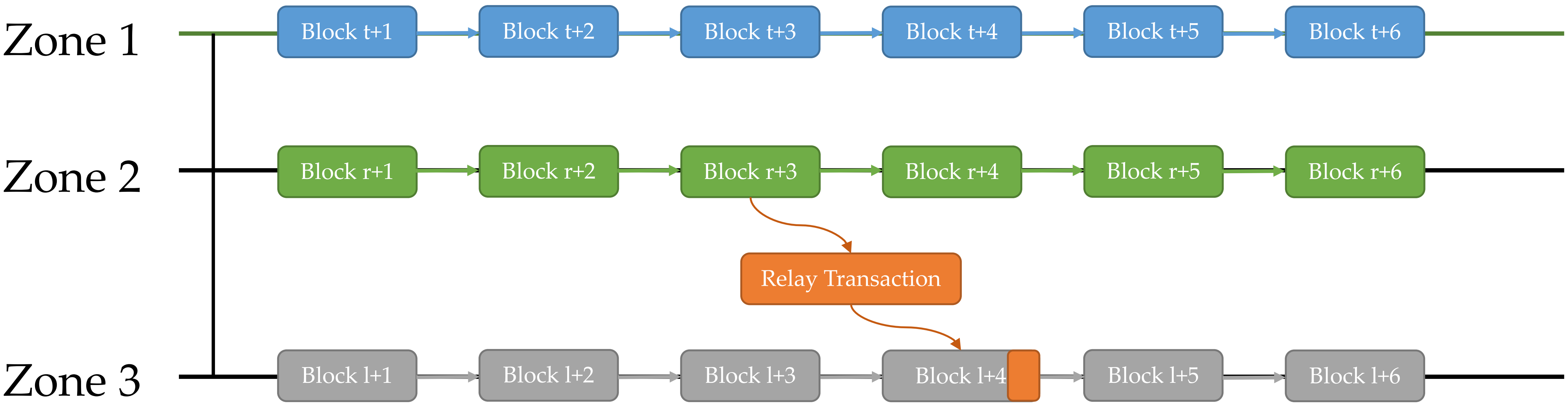
Multi-shard Transactions need 2PC protocol → Initiated by the Reference Committee.



Monoxide: Scale out Blockchains with Asynchronous Consensus Zones (NSDI'19)

- Apply sharding on public blockchain cryptocurrency with Asynchronous Consensus Zone.
- Eventual Atomicity: First withdraw transaction, Later the deposit transaction.
- Asynchronous Consensus Zone: Parallel chains in different zones working independently.
- Mining power amplification: Chu-ko-nu Mining distributes mining power evenly across zones.
- Cross Zone Transactions: broadcasting relay transactions for the inter-zone part of the TXNs.

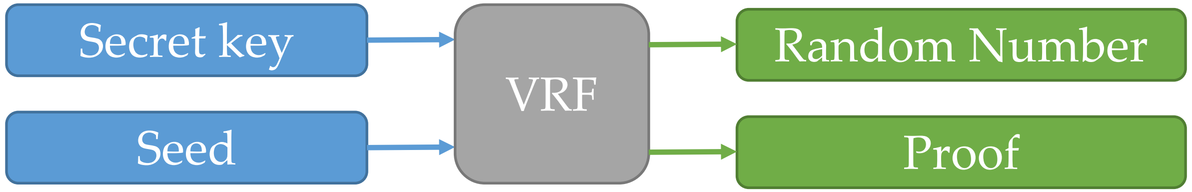
Monoxide: Scale out Blockchains with Asynchronous Consensus Zones



Normal Case with Good Primary and one fault

Algorand: Scaling Byzantine Agreement for Cryptocurrencies [SOSP'17]

- **Committee Based Consensus**: Scalability through consensus among selected users.
- **Proof of Stake:** Block proposers selected based on their stake.
- **Cryptographic Sortition:** Committee selection done independently at each node.
- **Verifiable Random Function**: Takes a secret key and a value and produces a pseudorandom output, with a proof



Algorand: Scaling Byzantine Agreement for Cryptocurrencies

Committe → Block Proposal → Reduction → Binary Byzantine Agreement

Committe: Using VRF and Gossip

Block Proposal: Selected members will propose blocks

Reduction: Reduce to one block to consider using priority

Binary Byzantine Agreement: Choose selected block or empty block

2 – 11 round of voting

Final Agreement

Tentative Agreement

- Final consensus: consensus is reached and block is accepted
- Tentative consensus: other users may reached tentative consensus on different block,
  - Fixed during recovery process or final block after

SESSION II

An In-Depth Look of BFT Consensus in Blockchain: Challenges and Opportunities

ExpoLab
Creativity Unfolded

ResilientDB

UCDAVIS
UNIVERSITY OF CALIFORNIA

ResilientDB ExpoLab Creativity Unfolded

Requirements of Existing BFT Protocols

1) Require three phases of communication, of which two necessitate quadratic communication (PBFT).
2) Expect no failures or dependence on clients (Zyzzyva).
3) Incur high client latencies due to many phases of communication (PBFT, HotStuff).
4) Require threshold signatures, which are computationally expensive (HotStuff).
5) Require more than 3f+1 replicas (Q/U, HQ).
6) Need trusted components (AHL, Attested Append-only memory).

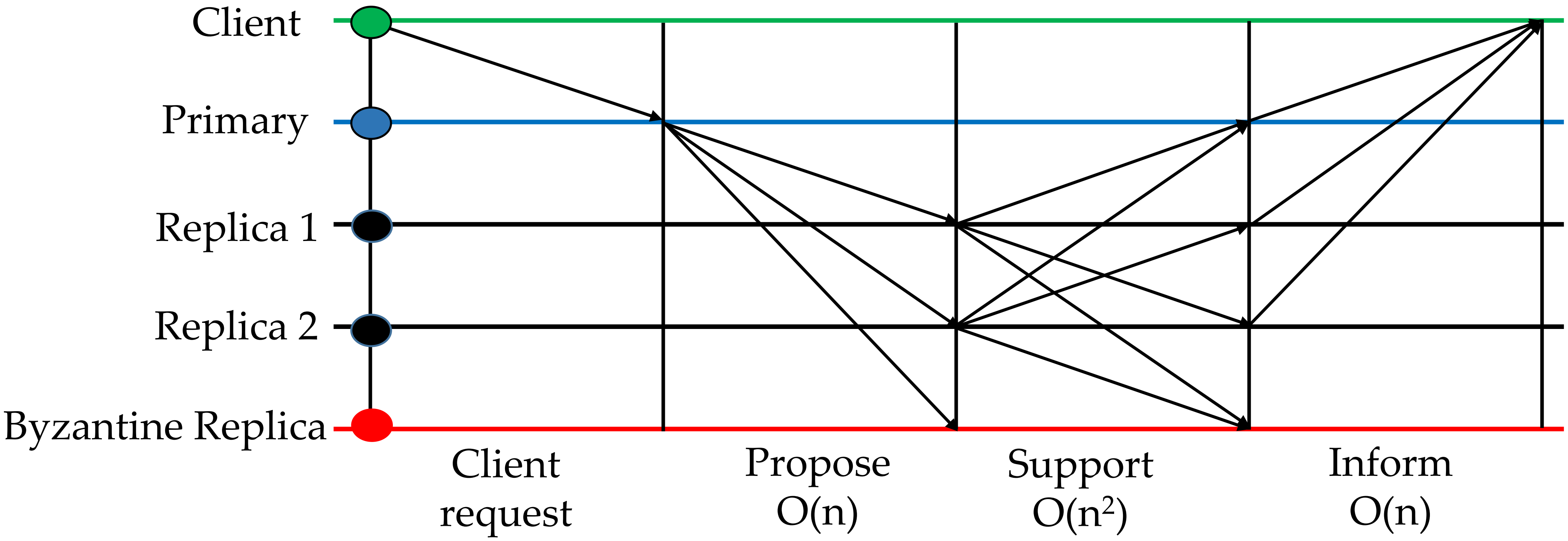
Proof-of-Execution (PoE):
Reaching Consensus through Fault-Tolerant Speculation

- *Speculative Execution* to reduce the client latency.
- *Out-of-Order message processing* for transactions.
- *Two-Phases* of communication.
- *No Dependence* on Clients or requirement of expensive cryptographic primitives.

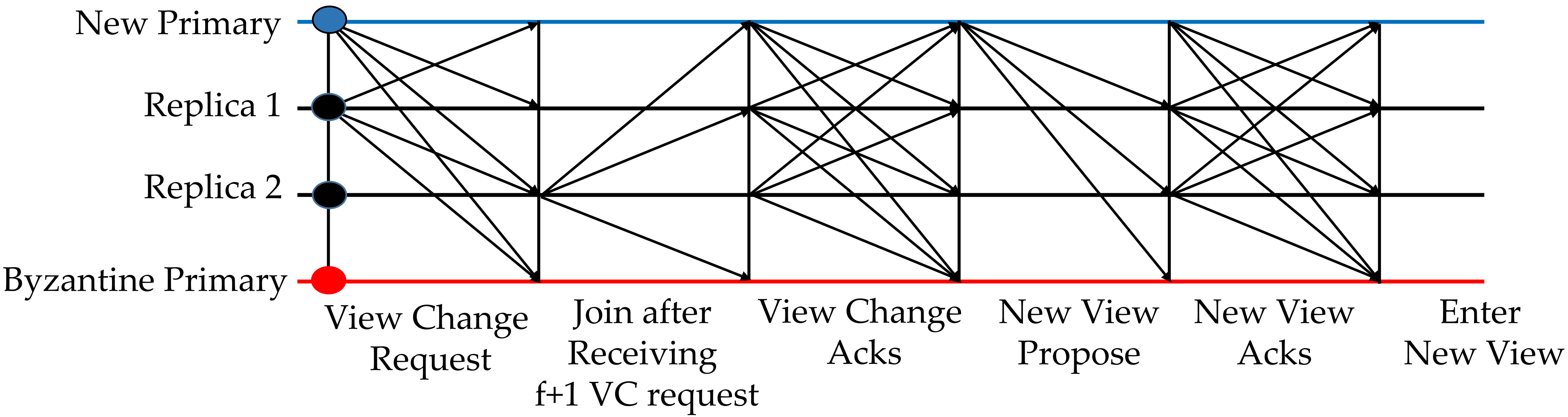
PoE vs Other Protocols

Protocol	Phases	Messages	Computation	Resilience	Requirements
Zyzzyva	1	$\mathcal{O}(\mathbf{n})$	high	0	reliable clients
PoE (our paper)	2	$\mathcal{O}(\mathbf{n} + \mathbf{n}^2)$	low	$\mathbf{f}$	
Pbft	3	$\mathcal{O}(\mathbf{n} + 2\mathbf{n}^2)$	low	$\mathbf{f}$	
HotStuff	4	$\mathcal{O}(\mathbf{n} + 3\mathbf{n}^2)$	high	$\mathbf{f}$	
HotStuff-ts	8	$\mathcal{O}(4\mathbf{n})$	very high	$\mathbf{f}$	threshold sign.

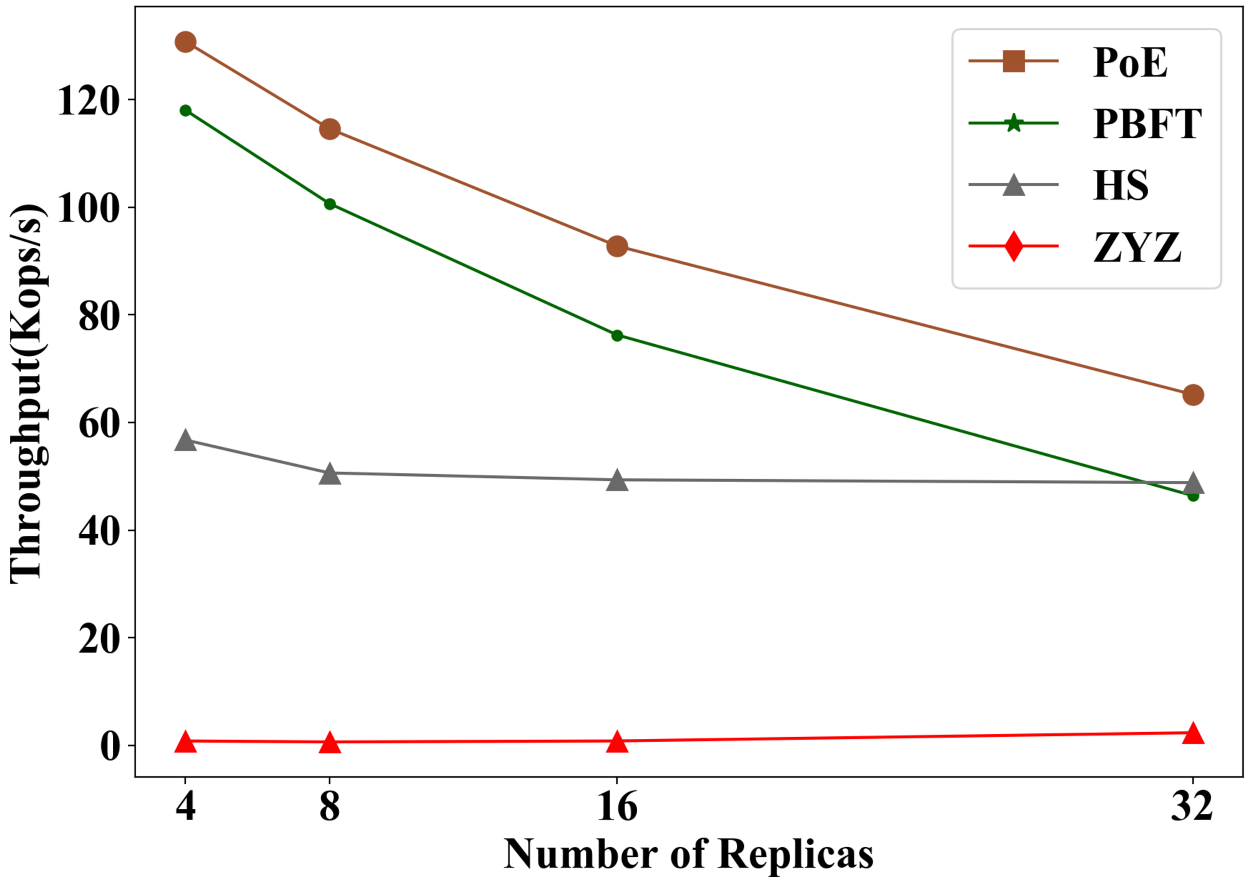
PoE Protocol (Non-Faulty Primary)



PoE View Change Protocol:
Replacing Malicious Primary



PoE Scalability under Single Failure



Scaling Blockchain Databases through Parallel Resilient Consensus Paradigm

- Why should BFT protocols rely on just ***one*** primary replica?
- Malicious primary can ***throttle*** the system throughput.
- Malicious primary requires ***replacement*** —> fall in throughput.

ResilientDB ExpoLab Creativity Unfolded

Multiple Byzantine Fault-Tolerance (MultiBFT) Paradigm

- Designate multiple replicas as Primaries!
- Run multiple parallel consensuses on each replica.

BFT protocol →

Number of Instances (z) →

Replication

Create z instances at each replica and run z parallel BFT instances.

→

Unification

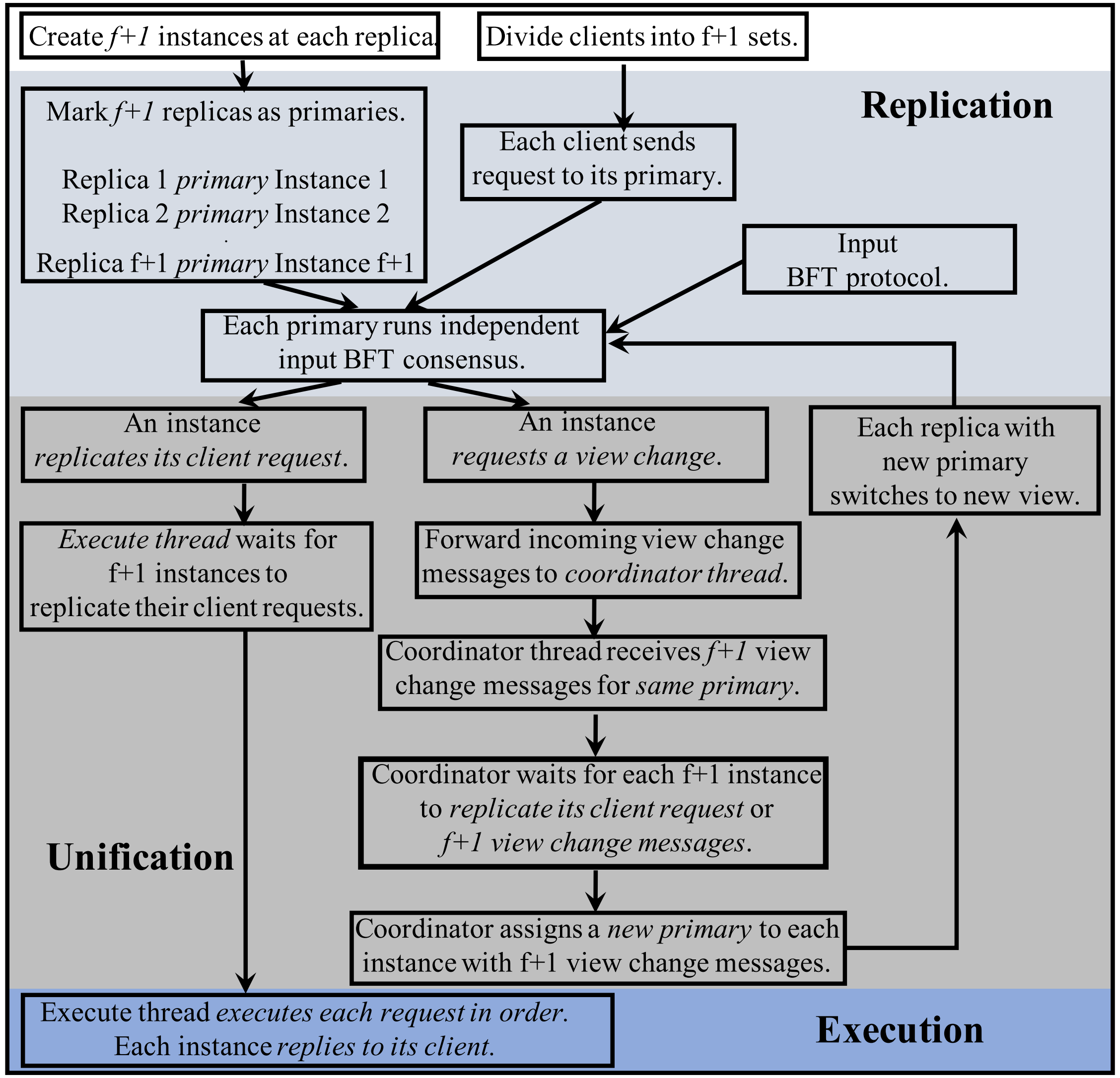
Create a global order of all the requests.

→

Execution

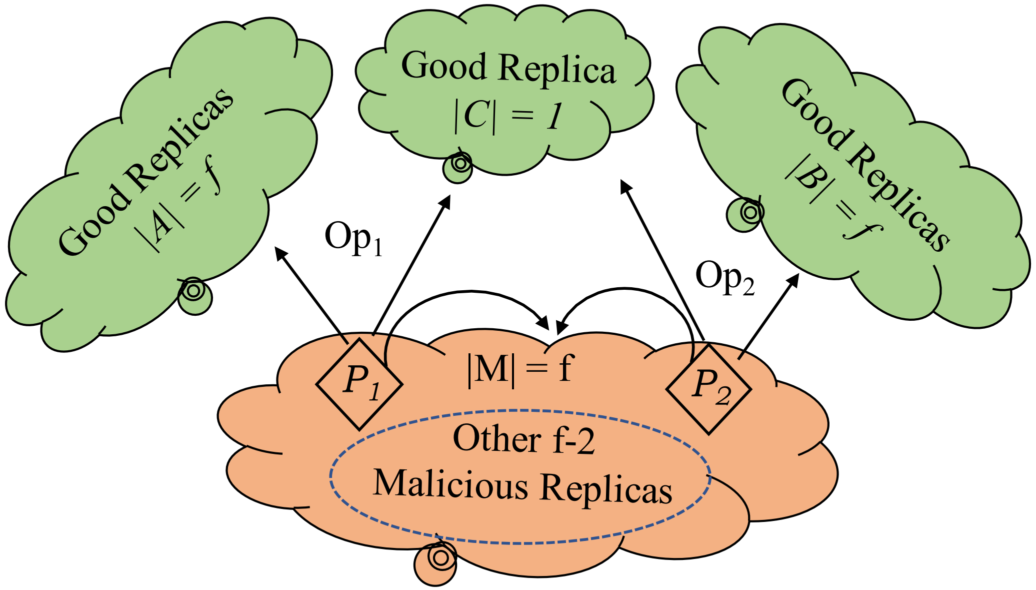
Execute the requests in a global order and reply to clients.

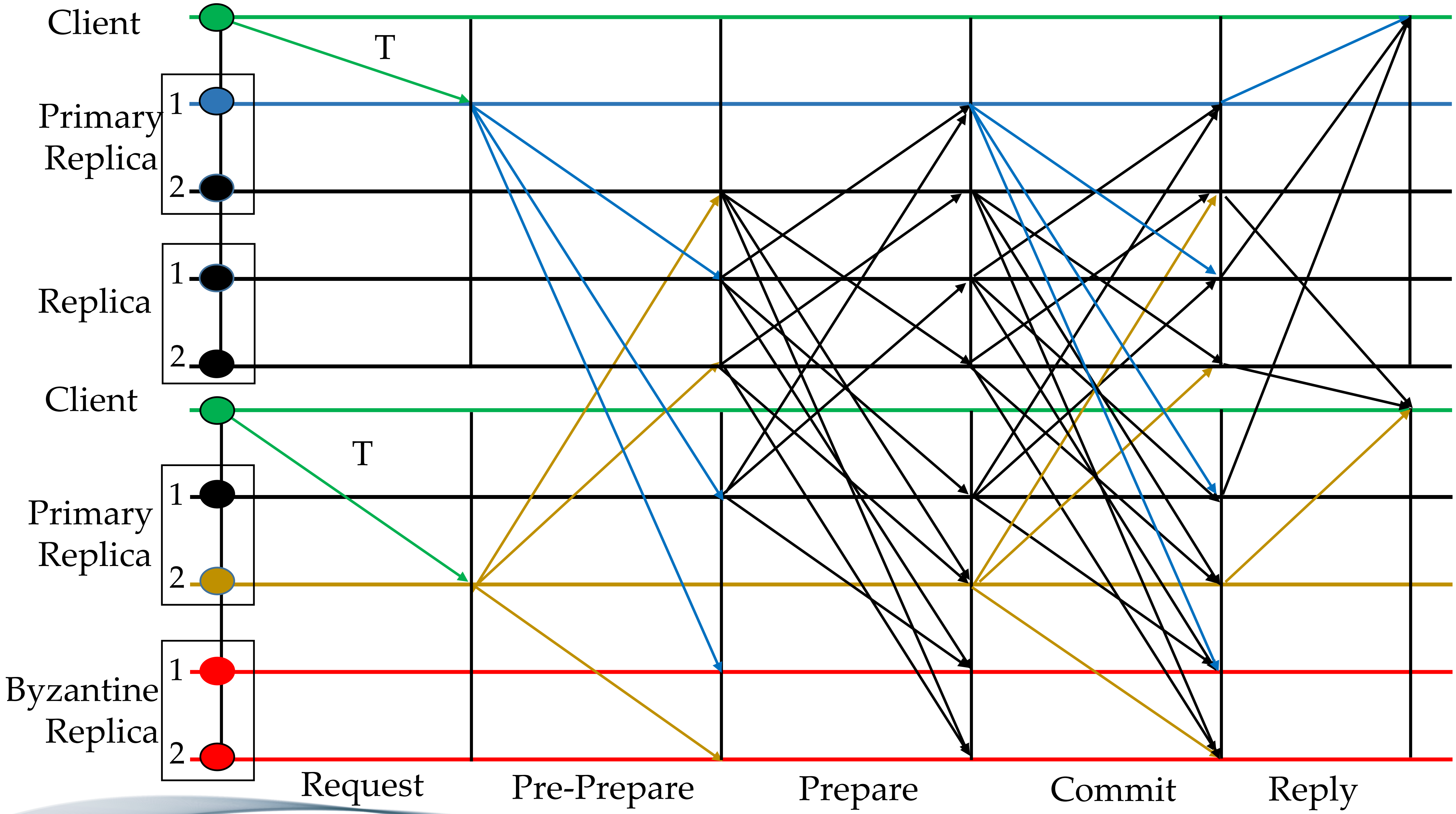
MultiBFT Flow



Malicious Primaries Collusion

- Multiple malicious primaries can prevent liveness!
- Solution → Optimistic Recovery through State Exchange.

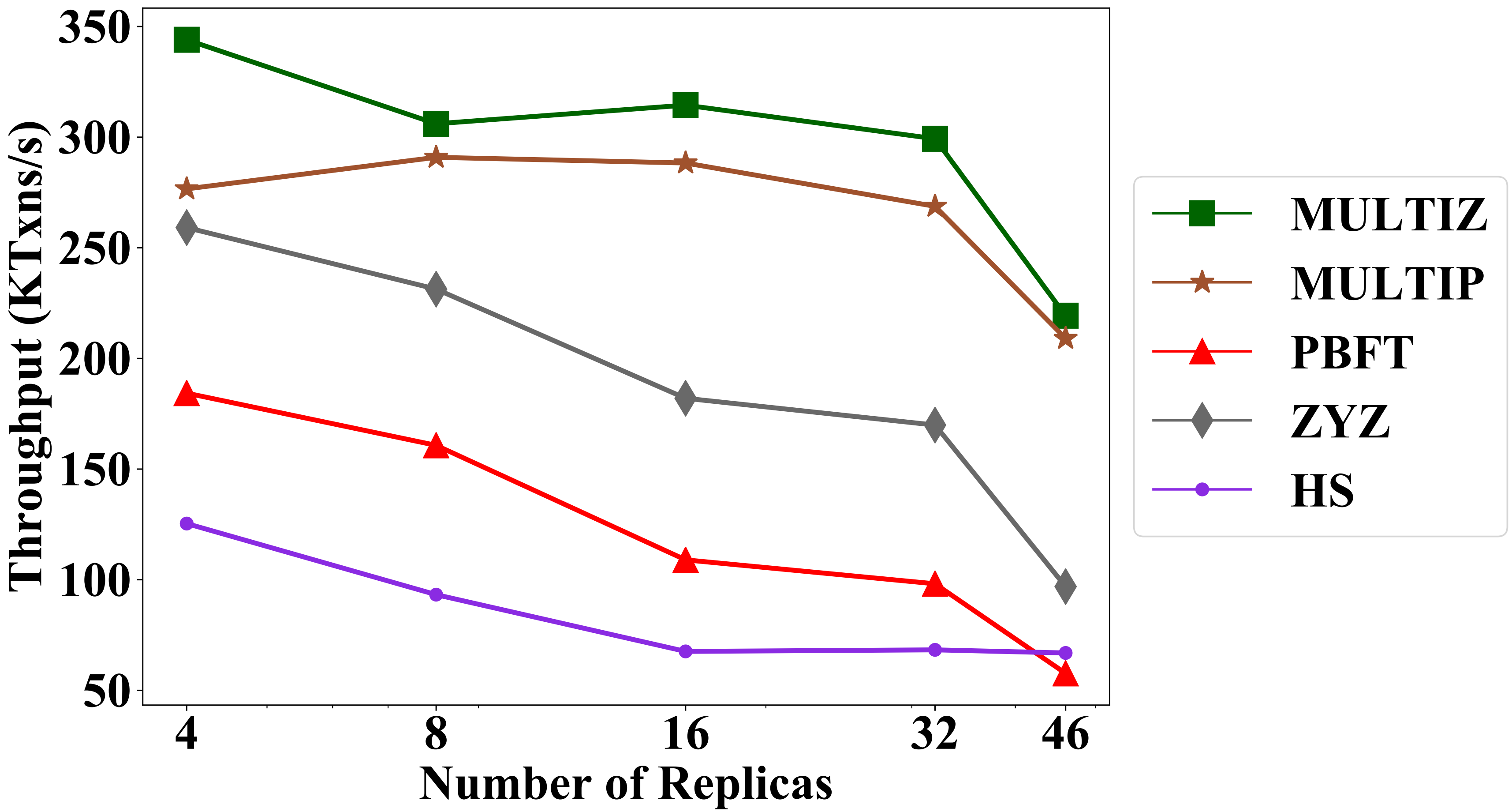




MultiBFT using PBFT with 2 parallel instances on each replica

ResilientDB ExpoLab Creativity Unfolded

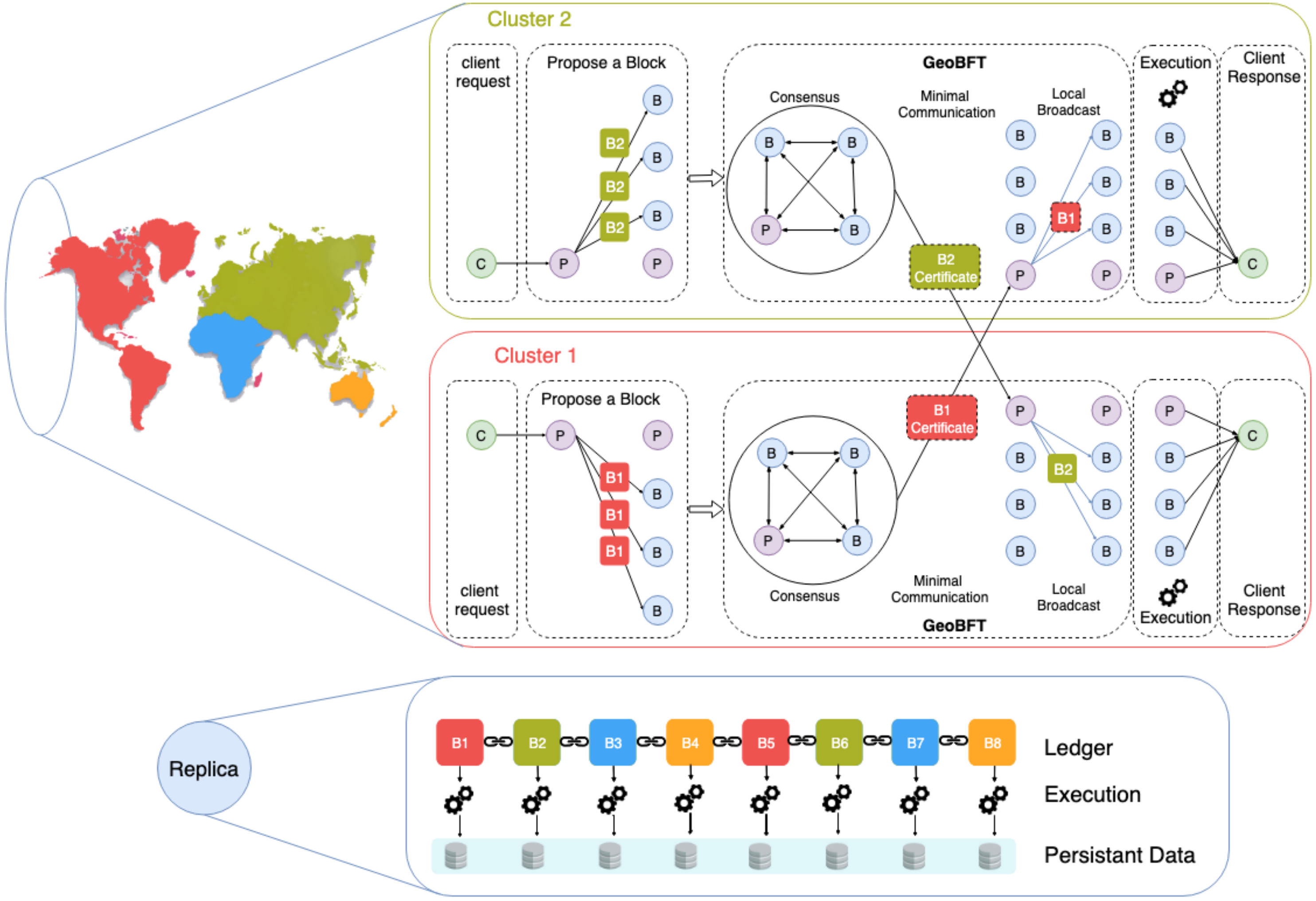
MultiBFT Scalability



Global Scale Resilient Blockchain Fabric

- Traditional BFT protocols do not scale to geographically large distances.
- Blockchain requires decentralization → replicas can be far apart → expensive communication!
- The underlying BFT consensus protocol should be topology-aware.

Vision Geo-Scale Byzantine Fault-Tolerance



GeoBFT Protocol

- GeoBFT groups replicas into clusters based on the distance between these replicas.
- Each cluster runs the PBFT consensus protocol, in parallel and independently.

Replication

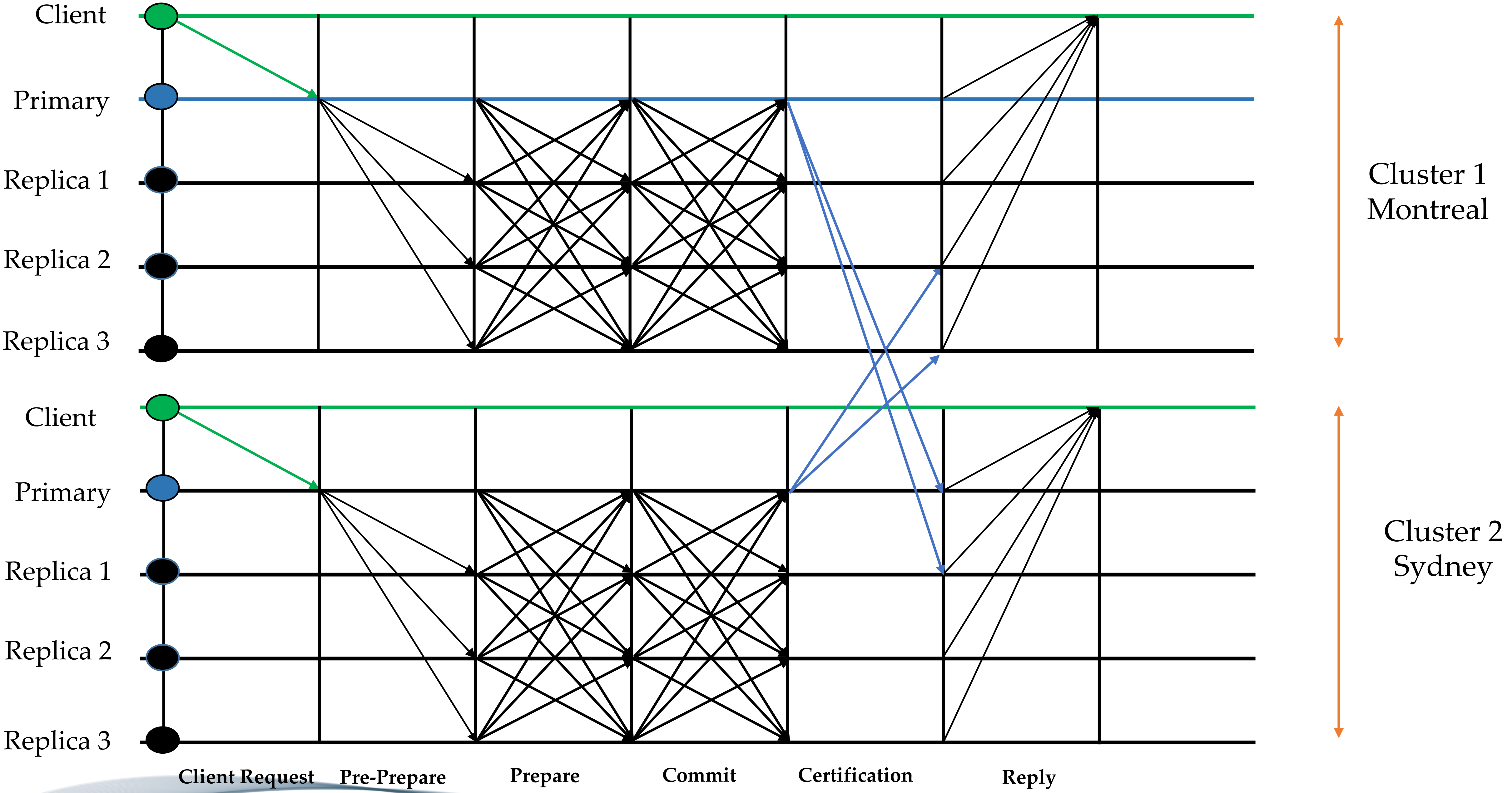
Each cluster runs PBFT protocol in parallel on its client request.

Certification

Primary at each cluster creates a certificate and sends to f+1 replicas of all the other clusters.

Execution

Execute the requests in a global order and reply to clients.

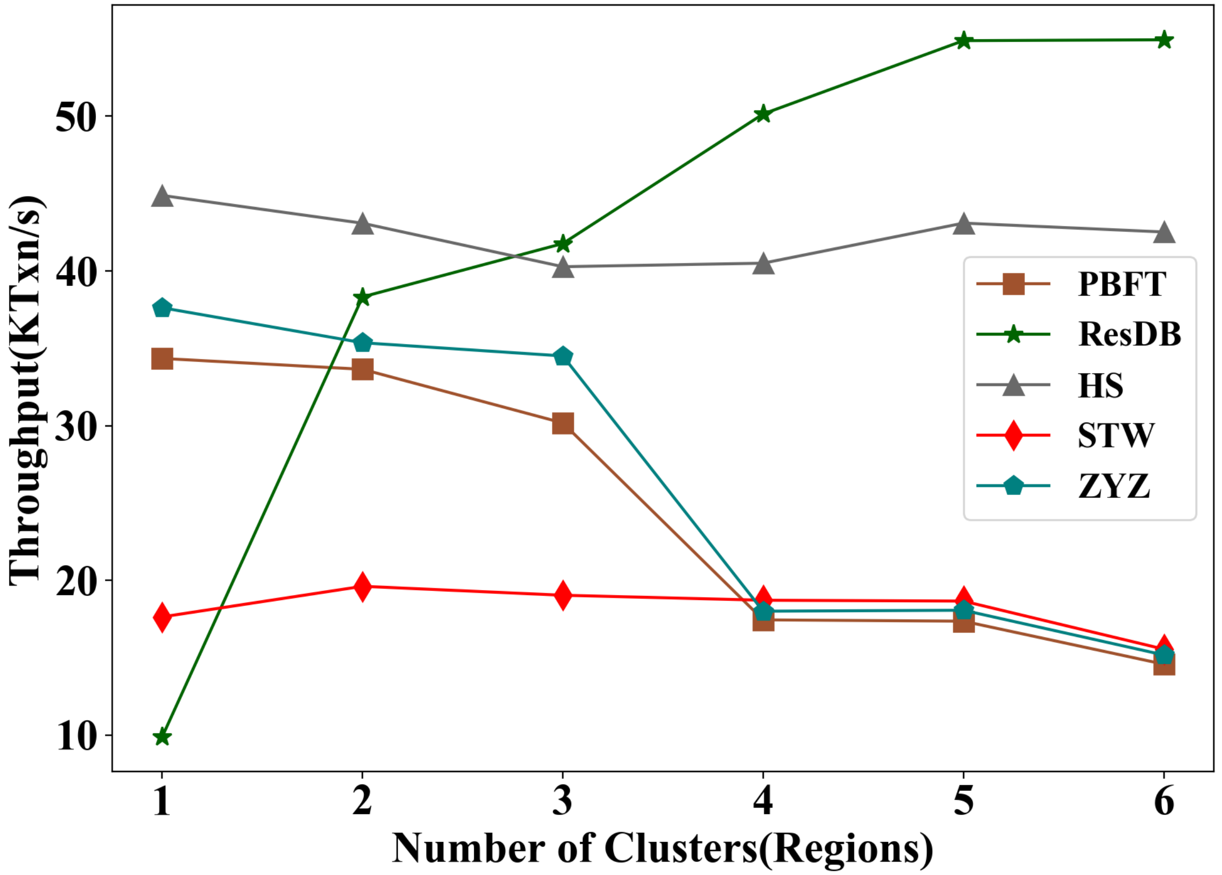


ResilientDB ExpoLab Creativity Unfolded

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.

GeoBFT Scalability



ResilientDB: High Throughput Yielding, Scalable Permissioned Blockchain Fabric

Visit at: https://resilientdb.com/

ExpoLab
Creativity Unfolded

ResilientDB

UCDAVIS
UNIVERSITY OF CALIFORNIA

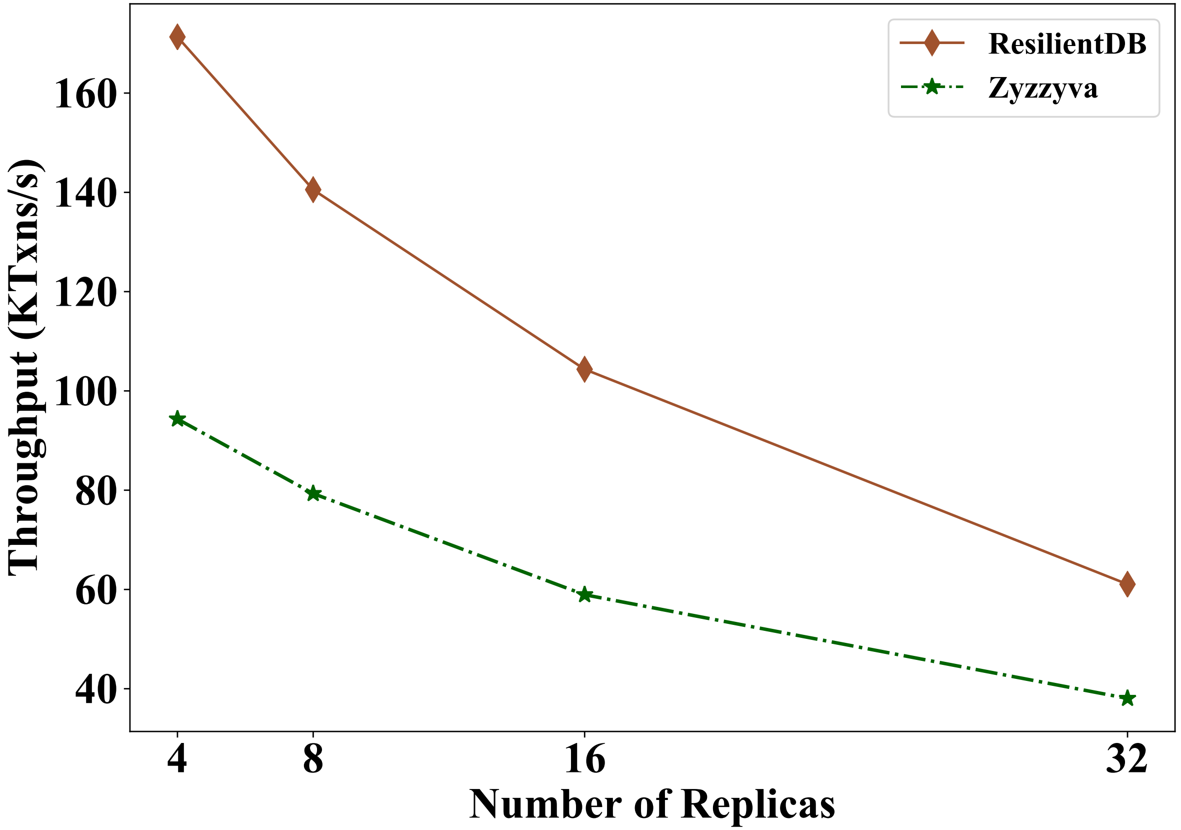
Why Should You Chose ResilientDB?

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.

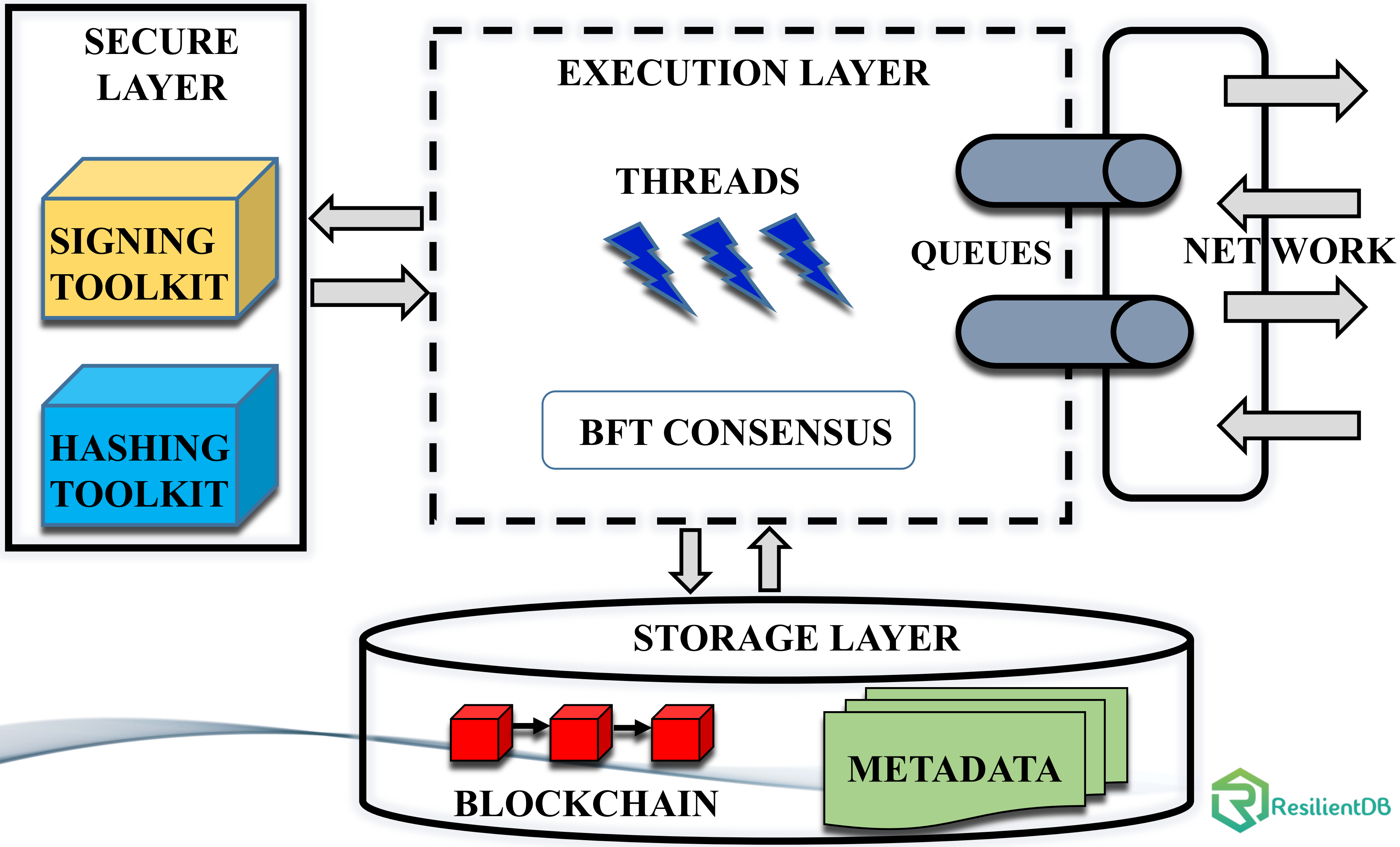
Dissecting Existing Permissioned Blockchains

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

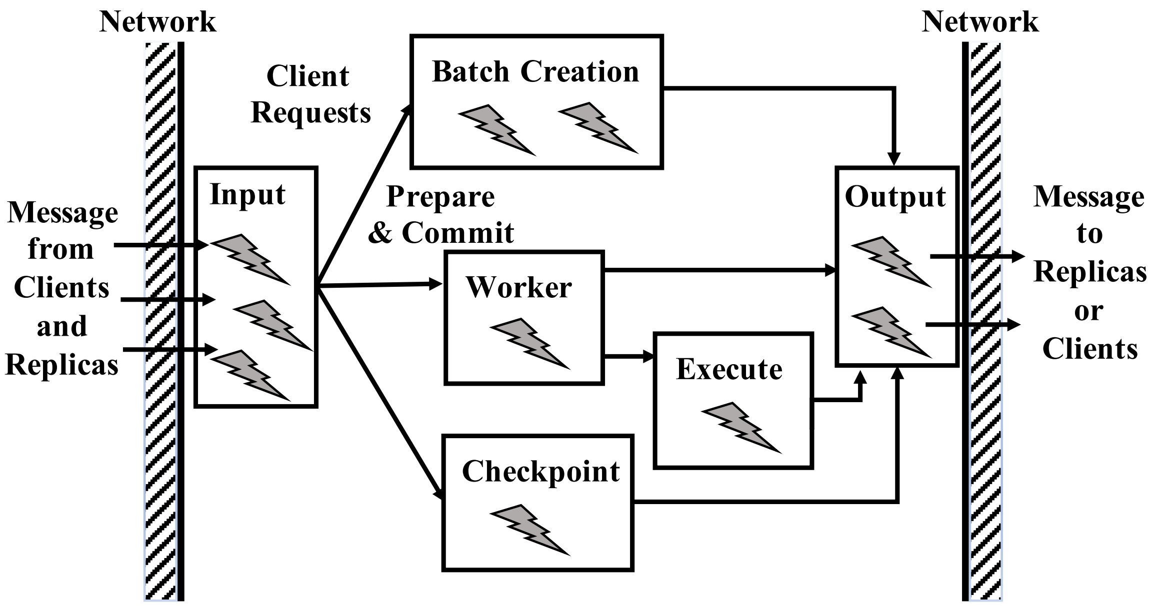
Can a well-crafted system based on a classical BFT protocol outperform a modern protocol?



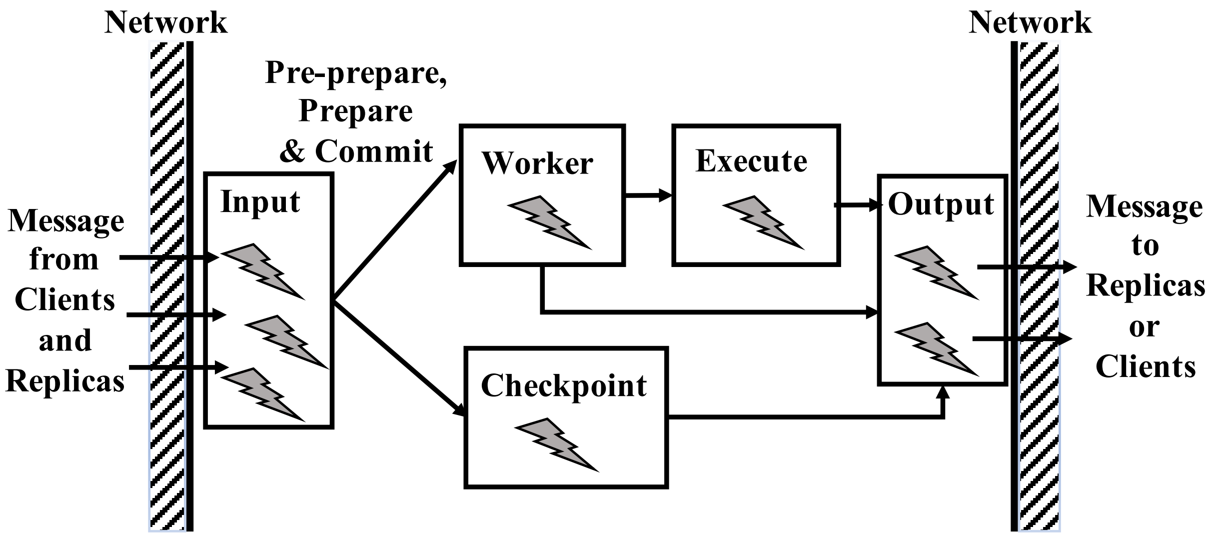
ResilientDB Architecture



ResilientDB Multi-Threaded Deep Pipeline

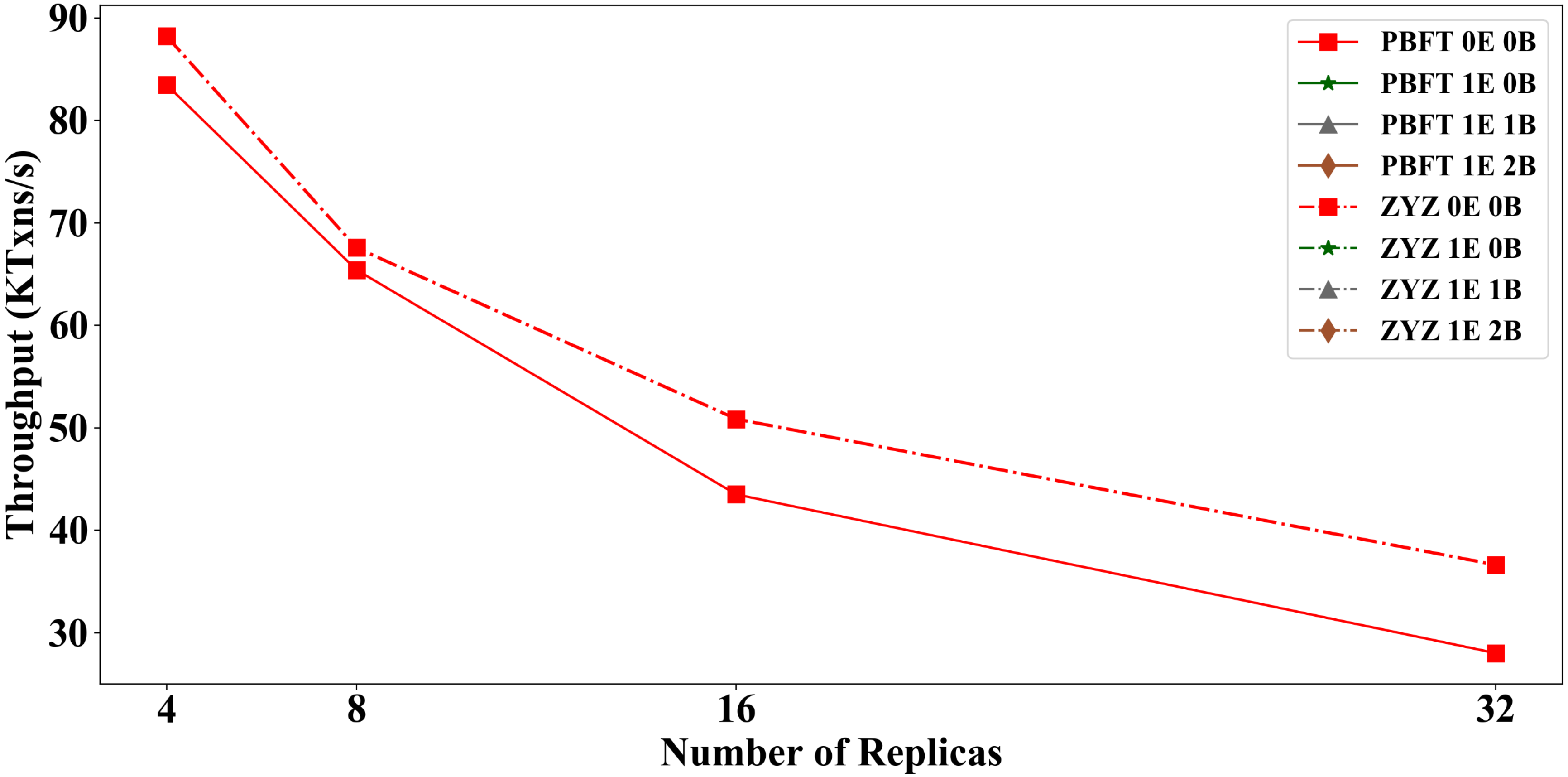


Pipeline at Primary Replica



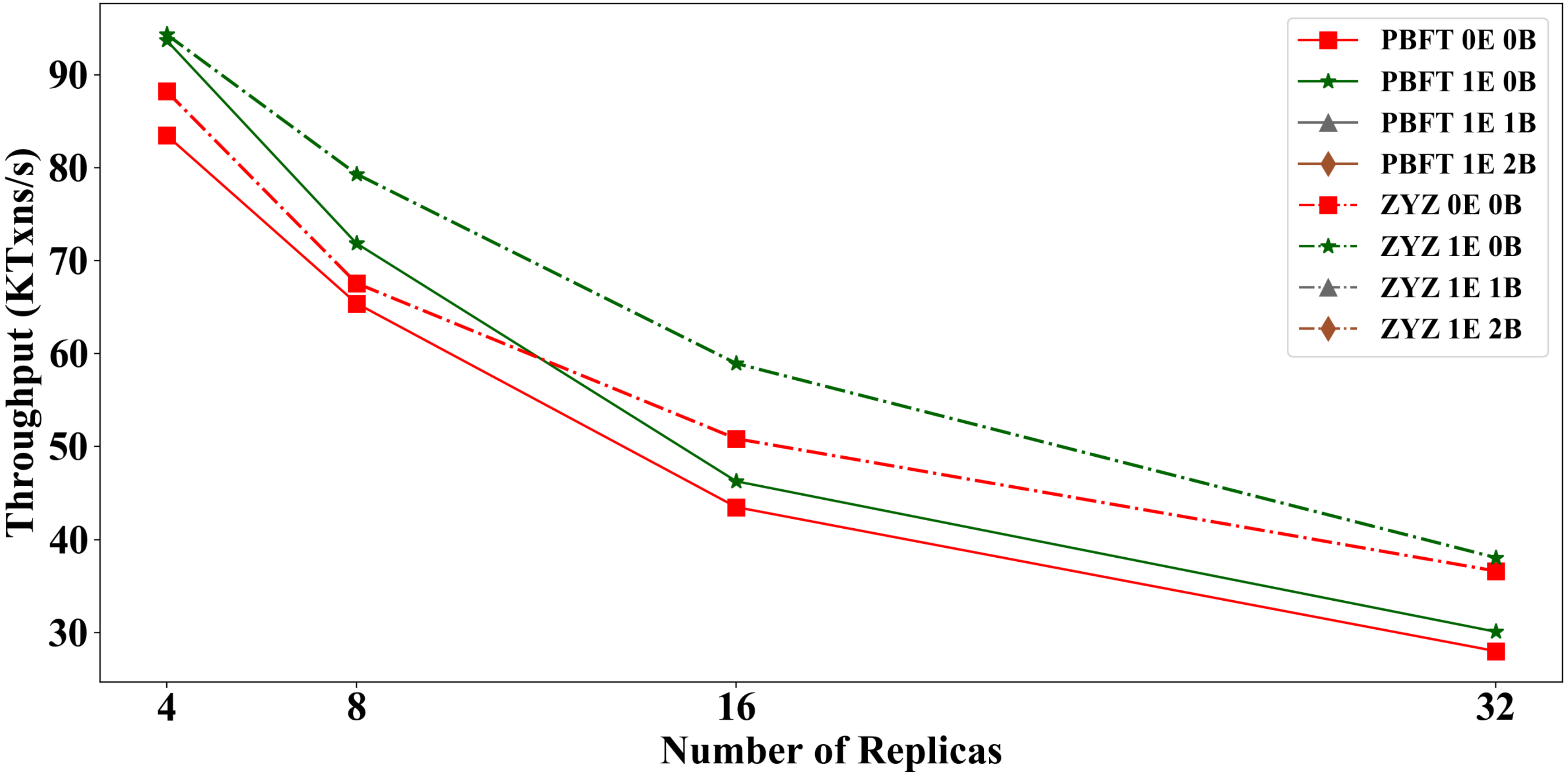
Pipeline at Non-Primary Replica

Insight 1: Multi-Threaded pipeline Gains



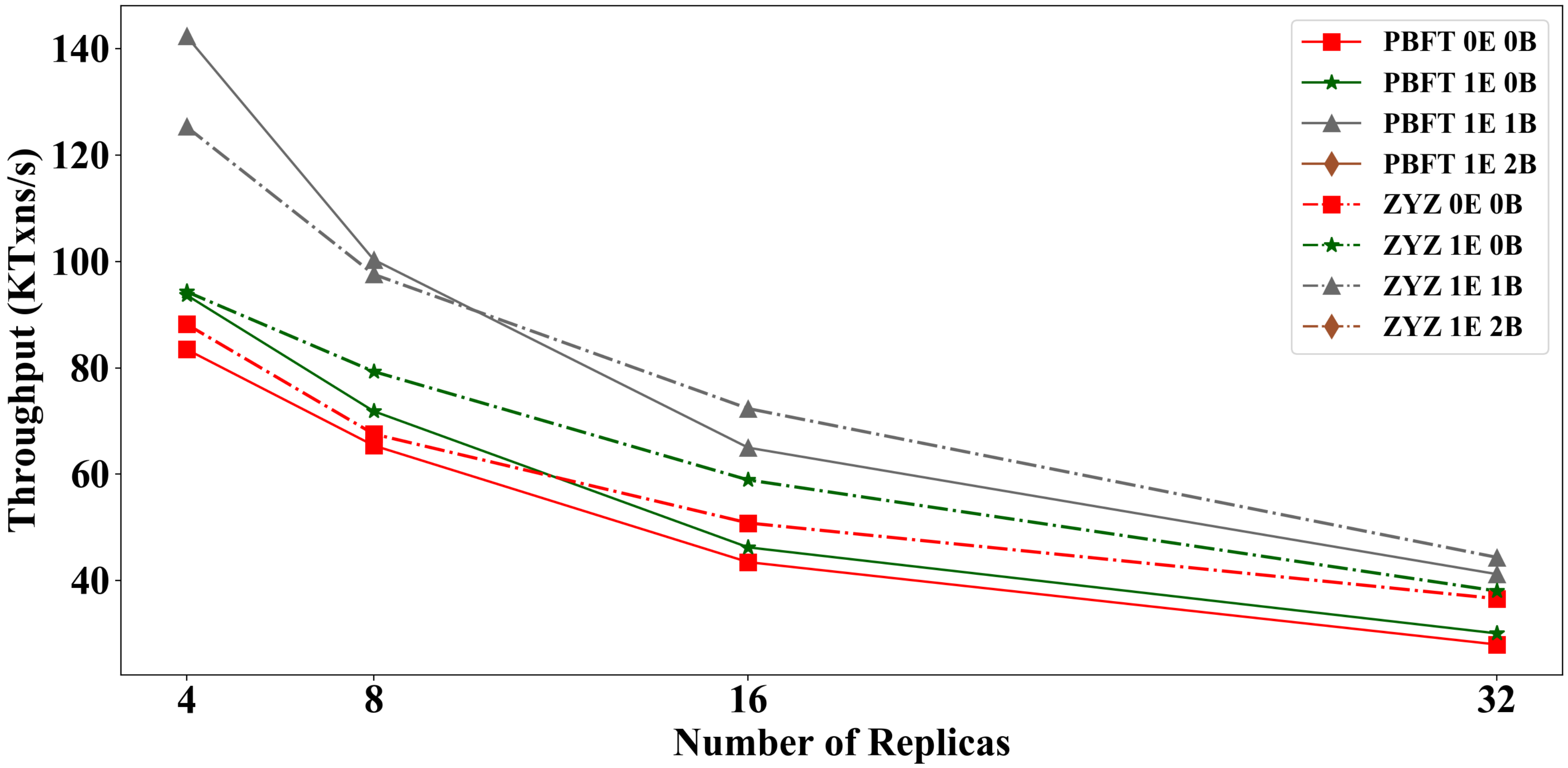
Parallelizing and Pipelining tasks across worker, execution (E) and batch-threads (B).

Insight 1: Multi-Threaded pipeline Gains



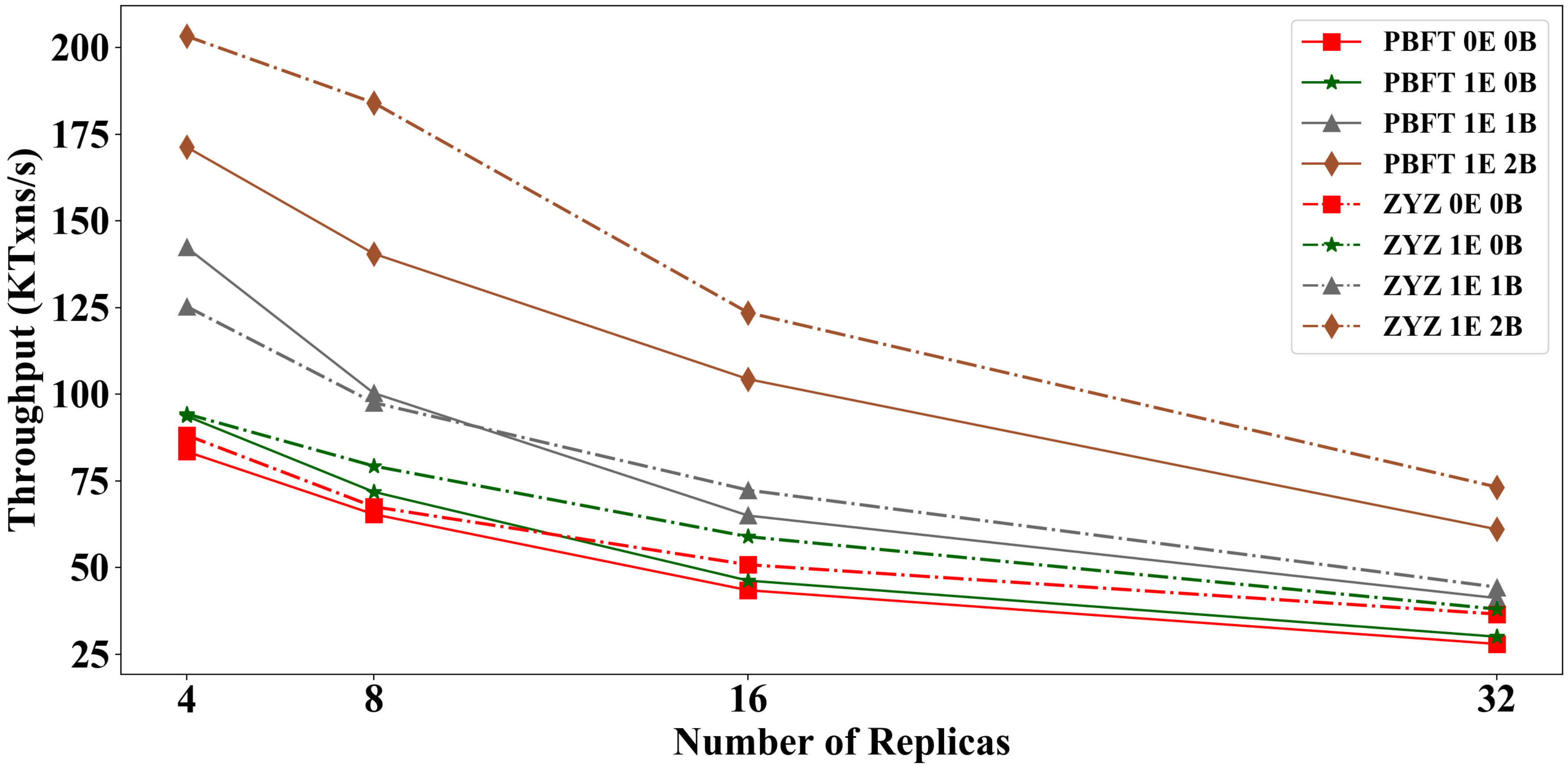
Parallelizing and Pipelining tasks across worker, execution (E) and batch-threads (B).

Insight 1: Multi-Threaded pipeline Gains



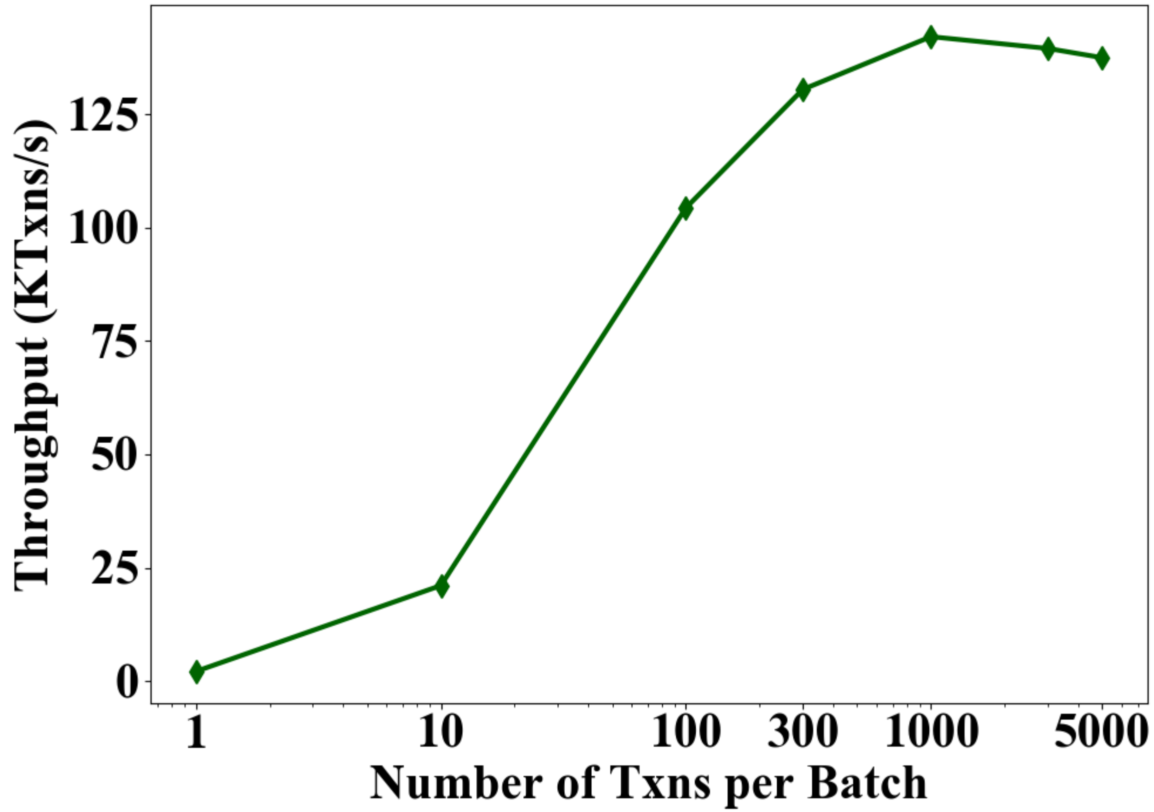
Parallelizing and Pipelining tasks across worker, execution (E) and batch-threads (B).

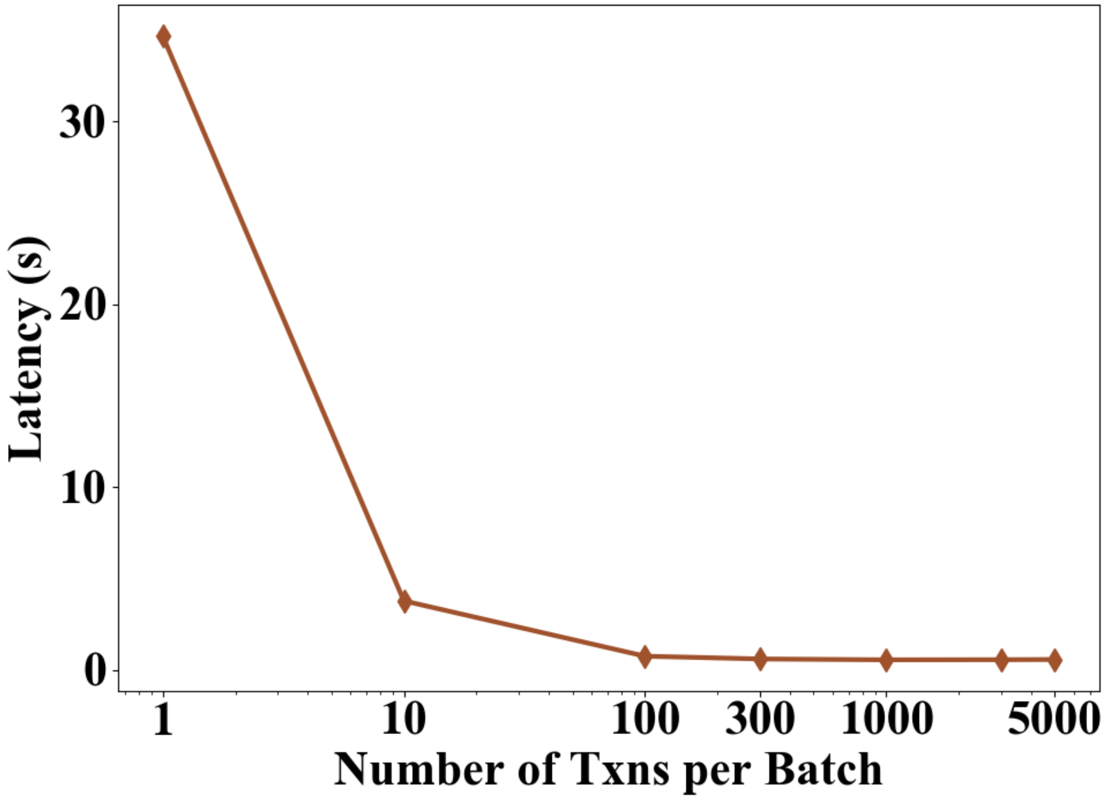
Insight 1: Multi-Threaded pipeline Gains



Parallelizing and Pipelining tasks across worker, execution (E) and batch-threads (B).

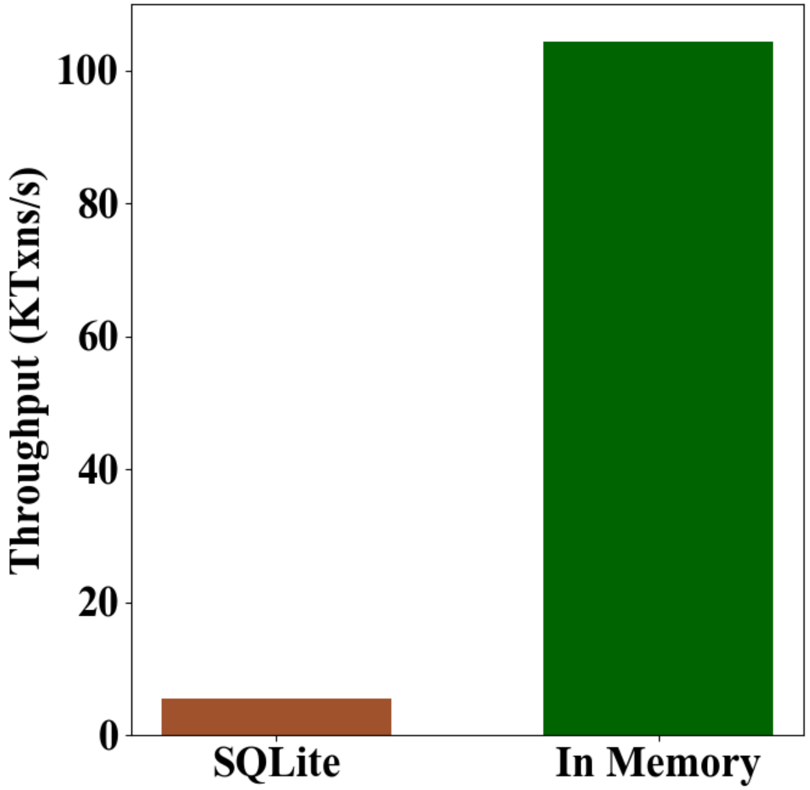
Insight 2: Optimal Batching Gains

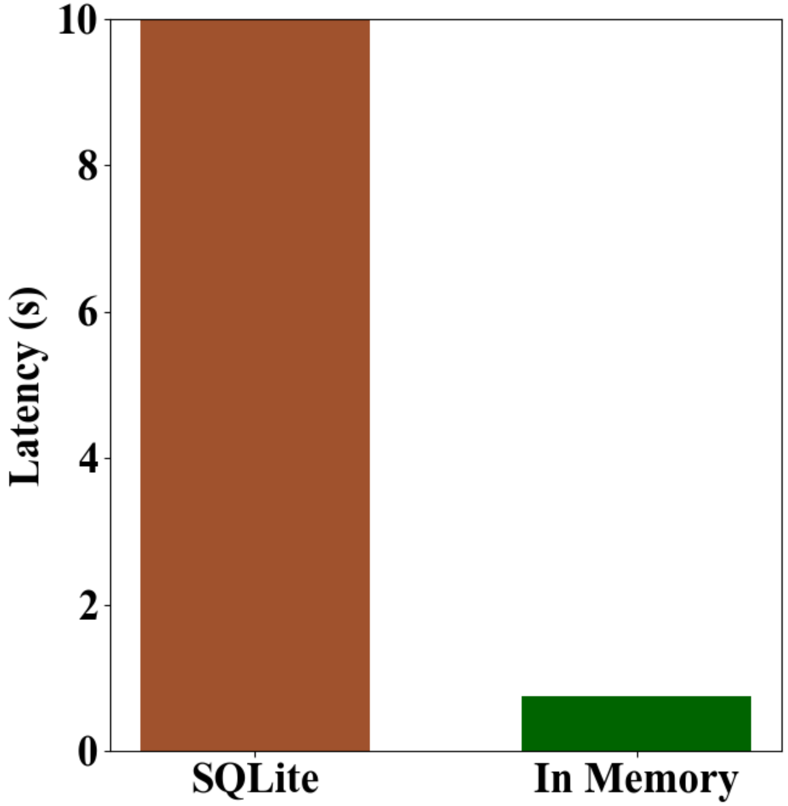




More transactions batched together → increase in throughput →reduced phases of consensus.

Insight 3: Memory Storage Gains





In-memory blockchain storage → reduces access cost.

ResilientDB: Hands On

Visit at: https://github.com/resilientdb/resilientdb

UCDAVIS
UNIVERSITY OF CALIFORNIA

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.

How to Run ResilientDB?

resilientdb / resilientdb

Watch 5 | Unstar 11 | Fork 13

Code | Issues 1 | Pull requests 0 | Actions | Projects 0 | Wiki | Security | Insights

ResilientDB: A scalable permissioned blockchain fabric

46 commits | 1 branch | 0 packages | 2 releases | 4 contributors | MIT

Branch: master | New pull request | Create new file | Upload files | Find file | Clone or download

gupta-suyash readme updated — Latest commit f2302e6 3 days ago

benchmarks	Initial Commit	16 days ago
blockchain	ledger archiecture defined	4 days ago
client	Initial Commit	16 days ago
deps	Initial Commit	16 days ago
scripts	added -e to handle multiple clients in docker-ifconfig	13 days ago
statistics	Initial Commit	16 days ago
system	ledger archiecture defined	4 days ago
transport	Initial Commit	16 days ago
.gitignore	Initial Commit	16 days ago
CHANGELOG.md	changelog added	3 days ago
CODE_OF_CONDUCT.md	Create CODE_OF_CONDUCT.md	15 days ago
LICENSE.md	Initial Commit	16 days ago
Makefile	Initial Commit	16 days ago
README.md	readme updated	3 days ago
config.cpp	Initial Commit	16 days ago
config.h	ledger archiecture defined	4 days ago
resilientDB-docker	Initial Commit	16 days ago

ResilientDB ExpoLab Creativity Unfolded

Docker CE

What is Docker?

> *an open-source project that automates the deployment of software applications inside* ***containers*** *by providing an additional layer of abstraction and automation of* ***OS-level virtualization*** *on Linux.*

- Run a distributed program on one machine
- Simulate with lightweight virtual machines

Docker CE

What is Docker?

> *an open-source project that automates the deployment of software applications inside **containers** by providing an additional layer of abstraction and automation of **OS-level virtualization** on Linux.*

- Run a distributed program on one machine
- Simulate with lightweight virtual machines

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

```
sajjad@sajjad-xps:~/WS/expo/resilientdb|master⚡
>  ./resilientDB-docker -d
Number of Replicas:     4
Number of Clients:      1
Stopping previous containers...
Stopping s3 ... done
Stopping s1 ... done
Stopping s4 ... done
Stopping c1 ... done
Stopping s2 ... done
Removing s3 ... done
Removing s1 ... done
Removing s4 ... done
Removing c1 ... done
Removing s2 ... done
Removing network resilientdb_default
Successfully stopped
Creating docker compose file ...
Docker compose file created --> docker-compose.yml
Starting the containers...
Creating network "resilientdb_default" with the default driver
Creating s4 ... done
Creating c1 ... done
Creating s1 ... done
Creating s2 ... done
Creating s3 ... done
ifconfig file exists... Deleting File
Deleted
Server sequence --> IP
c1 --> 172.21.0.3
s1 --> 172.21.0.4
s2 --> 172.21.0.6
s3 --> 172.21.0.2
s4 --> 172.21.0.5
Put Client IP at the bottom
ifconfig.txt Created!

Checking Dependencies...
Installing dependencies..
/home/sajjad/WS/expo/resilientdb
Dependencies has been installed
```

Resilient DB

- Throughput
  - Transaction per second
- Average Latency
  - The 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 Theread

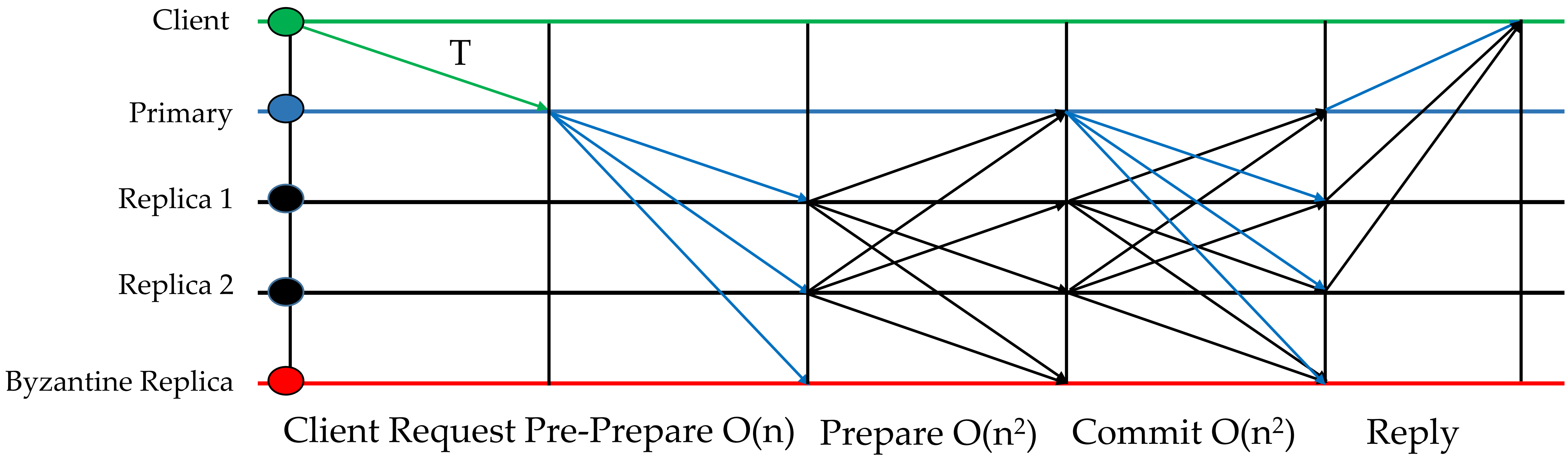
```
Throughputs:
0: 38525
1: 38530
2: 38558
3: 38551
4: 38564
Latencies:
latency 4: 0.505870

idle times:
Idleness of node: 0
Worker THD 0: 116.227
Worker THD 1: 62.0772
Worker THD 2: 62.2130
Worker THD 3: 105.098
Worker THD 4: 74.9193
Idleness of node: 1
Worker THD 0: 39.3157
Worker THD 1: 0.00000
Worker THD 2: 0.00000
Worker THD 3: 104.700
Worker THD 4: 74.8603
Idleness of node: 2
Worker THD 0: 35.0847
Worker THD 1: 0.00000
Worker THD 2: 0.00000
Worker THD 3: 102.415
Worker THD 4: 78.1078
Idleness of node: 3
Worker THD 0: 38.4452
Worker THD 1: 0.00000
Worker THD 2: 0.00000
Worker THD 3: 107.512
Worker THD 4: 77.6965
Memory:
0: 172 MB
1: 156 MB
2: 155 MB
3: 156 MB
4: 812 MB

avg thp: 4: 38541
avg lt : 1: .505
Code Ran successfully ---> res.out
```

PBFT: Practical Byzantine Fault Tolerance

Normal Case (Non-faulty Primary)



PBFT: Practical Byzantine Fault Tolerance

Client Request

- Client/client_main.cpp
- System/client_thread.cpp
- ClientQueryBatch Class
- Process ClientBatch in primary

client_main.cpp ×

client > client_main.cpp > ...

```cpp
int main(int argc, char *argv[])
{
    printf("Running client...\n\n");
    // 0. initialize global data structure
    parser(argc, argv);
    assert(g_node_id >= g_node_cnt);
    uint64_t seed = get_sys_clock();
    srand(seed);
    printf("Random seed: %ld\n", seed);

    int64_t starttime;
    int64_t endtime;
    starttime = get_server_clock();
    // per-partition malloc
    printf("Initializing stats... ");
    fflush(stdout);
    stats.init(g_total_client_thread_cnt);
    printf("Done\n");
    printf("Initializing transport manager... ");
    fflush(stdout);
    tport_man.init();
    printf("Done\n");
    printf("Initializing client manager... ");
    Workload *m_wl = new YCSBWorkload;
    m_wl->Workload::init();
```

client_thread.cpp ×

system > client_thread.cpp > ...

```cpp

RC ClientThread::run()
{

    tsetup();
    printf("Running ClientThread %ld\n", _thd_id);

    while (true)
    {
        keyMTX.lock();
        if (keyAvail)
        {
            keyMTX.unlock();
            break;
        }
        keyMTX.unlock();
    }

    BaseQuery *m_query;
    uint64_t iters = 0;
    uint32_t num_txns_sent = 0;
    int txns_sent[g_node_cnt];
    for (uint32_t i = 0; i < g_node_cnt; ++i)
        txns_sent[i] = 0;

    run_starttime = get_sys_clock();
```

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

worker_thread.cpp

system > worker_thread.cpp > WorkerThread::run()

```cpp
/**
 * Starting point for each worker thread.
 *
 * Each worker-thread created in the main() starts here. Each worker-thread is alive
 * till the time simulation is not done, and continuousy perform a set of tasks.
 * Thess tasks involve, dequeuing a message from its queue and then processing it
 * through call to the relevant function.
 */
RC WorkerThread::run()
{
    tsetup();
    printf("Running WorkerThread %ld\n", _thd_id);

    uint64_t agCount = 0, ready_starttime, idle_starttime = 0;

    // Setting batch (only relevant for batching threads).
    next_set = 0;

    while (!simulation->is_done())
    {
        txn_man = NULL;
        heartbeat();
        progress_stats();

#if VIEW_CHANGES
        // Thread 0 continously monitors the timer for each batch.
        if (get_thd_id() == 0)
        {
            check_for_timeout();
        }

        if (g_node_id != get_current_view(get_thd_id()))
        {
            check_switch_view();
        }
#endif

        // Dequeue a message from its work_queue.
        Message *msg = work_queue.dequeue(get_thd_id());
```

worker_thread.cpp

system > worker_thread.cpp > WorkerThread::process(Message *)

```cpp
void WorkerThread::process(Message *msg)
{
    RC rc __attribute__((unused));

    switch (msg->get_rtype())
    {
    case KEYEX:
        rc = process_key_exchange(msg);
        break;
    case CL_BATCH:
        rc = process_client_batch(msg);
        break;
    case BATCH_REQ:
        rc = process_batch(msg);
        break;
    case PBFT_CHKPT_MSG:
        rc = process_pbft_chkpt_msg(msg);
        break;
    case EXECUTE_MSG:
        rc = process_execute_msg(msg);
        break;
#if VIEW_CHANGES
    case VIEW_CHANGE:
        rc = process_view_change_msg(msg);
        break;
    case NEW_VIEW:
        rc = process_new_view_msg(msg);
        break;
#endif
    case PBFT_PREP_MSG:
        rc = process_pbft_prep_msg(msg);
        break;
    case PBFT_COMMIT_MSG:
        rc = process_pbft_commit_msg(msg);
        break;
    default:
        printf("Msg: %d\n", msg->get_rtype());
        fflush(stdout);
        assert(false);
        break;
    }
}
```

ResilientDB ExpoLab Creativity Unfolded

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

worker_thread_pbft.cpp ×

system > worker_thread_pbft.cpp > ...

```cpp
/**
 * Processes an incoming client batch and sends a Pre-prepare message to al rep
 *
 * This function assumes that a client sends a batch of transactions and
 * for each transaction in the batch, a separate transaction manager is create
 * Next, this batch is forwarded to all the replicas as a BatchRequests Messag
 * which corresponds to the Pre-Prepare stage in the PBFT protocol.
 *
 * @param msg Batch of Transactions of type CientQueryBatch from the client.
 * @return RC
 */
RC WorkerThread::process_client_batch(Message *msg)
{
    //printf("ClientQueryBatch: %ld, THD: %ld :: CL: %ld :: RQ: %ld\n",msg->txr
    //fflush(stdout);

    ClientQueryBatch *clbtch = (ClientQueryBatch *)msg;

    // Authenticate the client signature.
    validate_msg(clbtch);

#if VIEW_CHANGES
    // If message forwarded to the non-primary.
    if (g_node_id != get_current_view(get_thd_id()))
    {
        client_query_check(clbtch);
        return RCOK;
    }

    // Partial failure of Primary 0.
    fail_primary(msg, 9);
#endif

    // Initialize all transaction mangers and uint64_t Message::txn_id e.
    create_and_send_batchreq(clbtch, clbtch->txn_id);

    return RCOK;
}
```

worker_thread.cpp ×

system > worker_thread.cpp > WorkerThread::create_and_send_batchreq(ClientQueryBatch *, uint64_t)

```cpp
 * This function is used by the primary replicas to create and set
 * transaction managers for each transaction part of the ClientQueryBatch message
 * by the client. Further, to ensure integrity a hash of the complete batch is
 * generated, which is also used in future communication.
 *
 * @param msg Batch of transactions as a ClientQueryBatch message.
 * @param tid Identifier for the first transaction of the batch.
 */
void WorkerThread::create_and_send_batchreq(ClientQueryBatch *msg, uint64_t tid)
{
    // Creating a new BatchRequests Message.
    Message *bmsg = Message::create_message(BATCH_REQ);
    BatchRequests *breq = (BatchRequests *)bmsg;
    breq->init(get_thd_id());

    // Starting index for this batch of transactions.
    next_set = tid;

    // String of transactions in a batch to generate hash.
    string batchStr;

    // Allocate transaction manager for all the requests in batch.
    for (uint64_t i = 0; i < get_batch_size(); i++)
    {
        uint64_t txn_id = get_next_txn_id() + i;

        //cout << "Txn: " << txn_id << " :: Thd: " << get_thd_id() << "\n";
        //fflush(stdout);
        txn_man = get_transaction_manager(txn_id, 0);

        // Unset this txn man so that no other thread can concurrently use.
        while (true)
        {
            bool ready = txn_man->unset_ready();
            if (!ready)
            {
                continue;
            }
            else
            {
                break;
            }
        }
```

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

worker_thread_pbft.cpp ×

system > worker_thread_pbft.cpp > WorkerThread::process_batch(Message *)

```
/**
 * Process incoming BatchRequests message from the Primary.
 *
 * This function is used by the non-primary or backup replicas to process an incoming
 * BatchRequests message sent by the primary replica. This processing would require
 * sending messages of type PBFTPrepMessage, which correspond to the Prepare phase of
 * the PBFT protocol. Due to network delays, it is possible that a repica may have
 * received some messages of type PBFTPrepMessage and PBFTCommitMessage, prior to
 * receiving this BatchRequests message.
 *
 * @param msg Batch of Transactions of type BatchRequests from the primary.
 * @return RC
 */
RC WorkerThread::process_batch(Message *msg)
{
    uint64_t cntime = get_sys_clock();

    BatchRequests *breq = (BatchRequests *)msg;

    //printf("BatchRequests: TID:%ld : VIEW: %ld : THD: %ld\n",breq->txn_id, breq->view, get_
    //fflush(stdout);

    // Assert that only a non-primary replica has received this message.
    assert(g_node_id != get_current_view(get_thd_id()));

    // Check if the message is valid.
    validate_msg(breq);

```

ResilientDB

ExpoLab Creativity Unfolded

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

worker_thread_pbft.cpp ×

system > worker_thread_pbft.cpp > ...

```cpp
/**
 * Processes incoming Prepare message.
 *
 * This functions precessing incoming messages of type PBFTPrepMessage. If
 * received 2f identical Prepare messages from distinct replicas, then it c
 * and sends a PBFTCommitMessage to all the other replicas.
 *
 * @param msg Prepare message of type PBFTPrepMessage from a replica.
 * @return RC
 */
RC WorkerThread::process_pbft_prep_msg(Message *msg)
{
    //cout << "PBFTPrepMessage: TID: " << msg->txn_id << " FROM: " << msg->
    //fflush(stdout);

    // Start the counter for prepare phase.
    if (txn_man->prep_rsp_cnt == 2 * g_min_invalid_nodes)
    {
        txn_man->txn_stats.time_start_prepare = get_sys_clock();
    }

    // Check if the incoming message is valid.
    PBFTPrepMessage *pmsg = (PBFTPrepMessage *)msg;
    validate_msg(pmsg);

    // Check if sufficient number of Prepare messages have arrived.
    if (prepared(pmsg))
    {
        // Send Commit messages.
        txn_man->send_pbft_commit_msgs();

        // End the prepare counter.
        INC_STATS(get_thd_id(), time_prepare, get_sys_clock() - txn_man->tx
    }

    return RCOK;
}
```

worker_thread_pbft.cpp ×

system > worker_thread_pbft.cpp > WorkerThread::process_pbft_commit_msg(Message *)

```cpp
/**
 * Processes incoming Commit message.
 *
 * This functions precessing incoming messages of type PBFTCommitMessage
 * received 2f+1 identical Commit messages from distinct replicas, then
 * execute-thread to execute all the transactions in this batch.
 *
 * @param msg Commit message of type PBFTCommitMessage from a replica.
 * @return RC
 */
RC WorkerThread::process_pbft_commit_msg(Message *msg)
{
    //cout << "PBFTCommitMessage: TID " << msg->txn_id << " FROM: " << m
    //fflush(stdout);

    if (txn_man->commit_rsp_cnt == 2 * g_min_invalid_nodes + 1)
    {
        txn_man->txn_stats.time_start_commit = get_sys_clock();
    }

    // Check if message is valid.
    PBFTCommitMessage *pcmsg = (PBFTCommitMessage *)msg;
    validate_msg(pcmsg);

    txn_man->add_commit_msg(pcmsg);

    // Check if sufficient number of Commit messages have arrived.
    if (committed_local(pcmsg))
    {
#if TIMER_ON
        // End the timer for this client batch.
        server_timer->endTimer(txn_man->hash);
#endif

        // Add this message to execute thread's queue.
        send_execute_msg();

        INC_STATS(get_thd_id(), time_commit, get_sys_clock() - txn_man->
```

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

worker_thread.cpp

system > worker_thread.cpp > WorkerThread::process_execute_msg(Message *)

```
/**
 * Execute transactions and send client response.
 *
 * This function is only accessed by the execute-thread, which executes the transactions
 * in a batch, in order. Note that the execute-thread has several queues, and at any
 * point of time, the execute-thread is aware of which is the next transaction to
 * execute. Hence, it only loops on one specific queue.
 *
 * @param msg Execute message that notifies execution of a batch.
 * @ret RC
 */
RC WorkerThread::process_execute_msg(Message *msg)
{
    //cout << "EXECUTE " << msg->txn_id << " :: " << get_thd_id() <<"\n";
    //fflush(stdout);

    uint64_t ctime = get_sys_clock();

    // This message uses txn man of index calling process_execute.
    Message *rsp = Message::create_message(CL_RSP);
    ClientResponseMessage *crsp = (ClientResponseMessage *)rsp;
    crsp->init();

    ExecuteMessage *emsg = (ExecuteMessage *)msg;

    // Execute transactions in a shot
    uint64_t i;
    for (i = emsg->index; i < emsg->end_index - 4; i++)
    {
        //cout << "i: " << i << " :: next index: " << g_next_index << "\n";
        //fflush(stdout);

        TxnManager *tman = get_transaction_manager(i, 0);

        inc_next_index();

        // Execute the transaction
        tman->run_txn();

        // Commit the results.
        tman->commit();

        crsp->copy_from_txn(tman);
```

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 IOThread
- Dequeue in Worker Thread

work_queue.cpp

system > work_queue.cpp > ...

```
void QWorkQueue::enqueue(uint64_t thd_id, Message *msg, bool busy)
{
    uint64_t starttime = get_sys_clock();
    assert(msg);
    DEBUG_M("QWorkQueue::enqueue work_queue_entry alloc\n");
    work_queue_entry *entry = (work_queue_entry *)mem_allocator.align_alloc(sizeof(work_queue_ent
    entry->msg = msg;
    entry->rtype = msg->rtype;
    entry->txn_id = msg->txn_id;
    entry->batch_id = msg->batch_id;
    entry->starttime = get_sys_clock();
    assert(ISSERVER || ISREPLICA);
    DEBUG("Work Enqueue (%ld,%ld) %d\n", entry->txn_id, entry->batch_id, entry->rtype);

    if (msg->rtype == CL_QRY || msg->rtype == CL_BATCH)
    {
        if (g_node_id == get_current_view(thd_id))
        {
            //cout << "Placing \n";
            while (!new_txn_queue->push(entry) && !simulation->is_done())
            {
            }
        }
        else
        {
            assert(entry->rtype < 100);
            while (!work_queue[0]->push(entry) && !simulation->is_done())
            {
            }
        }
    }
```

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

io_thread.cpp ×

system > io_thread.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());
```

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

ResilientDB ExpoLab Creativity Unfolded

Thank You