

Building High Throughput Permissioned Blockchain Fabrics: Challenges and Opportunities



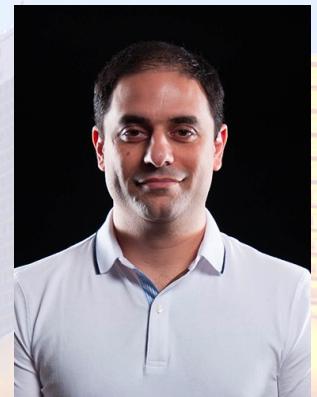
Suyash Gupta



Jelle Hellings



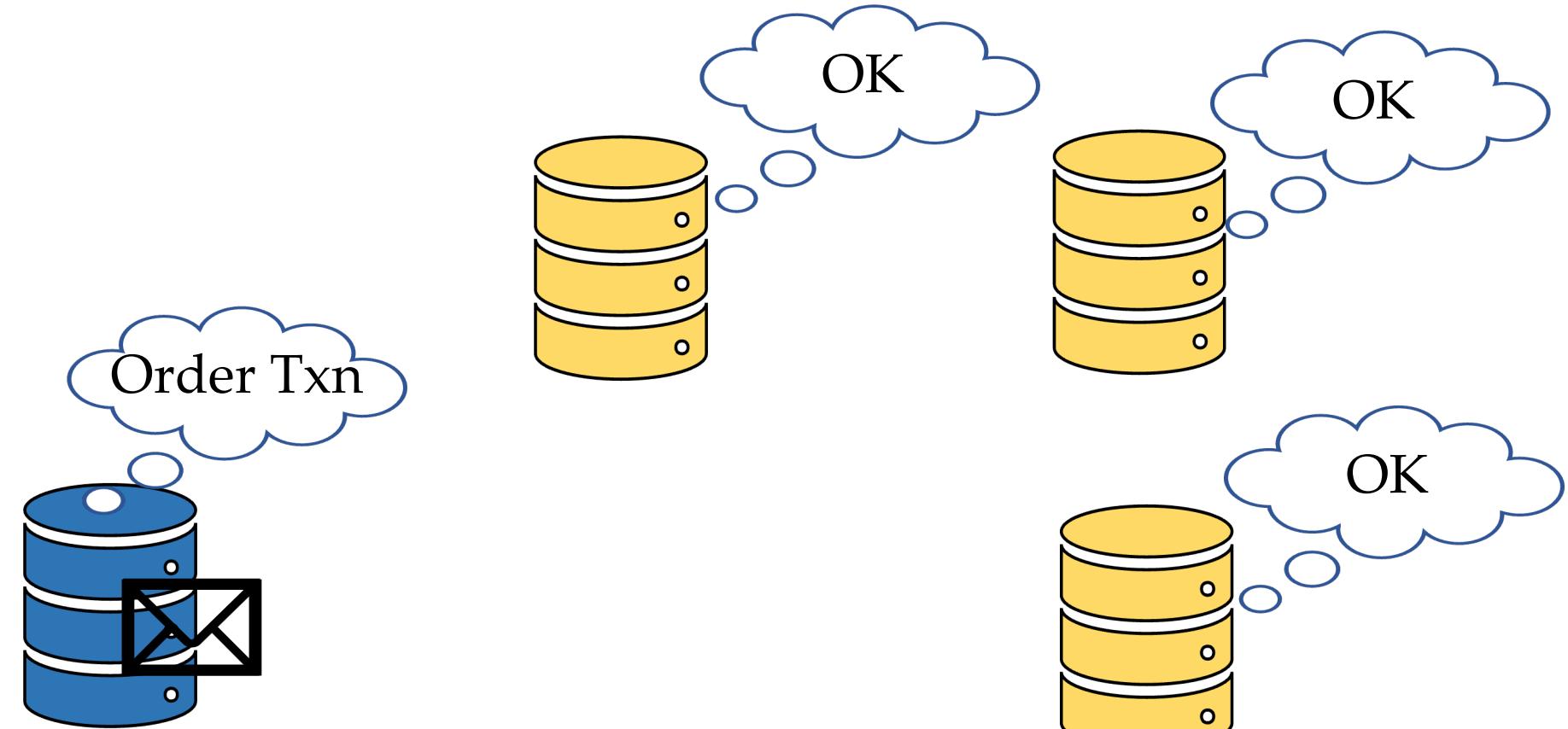
Sajjad Rahnama



Mohammad Sadoghi

Exploratory Systems Lab
University of California Davis

At the core of *any* Blockchain application is a Byzantine Fault-Tolerant (BFT) consensus protocol.

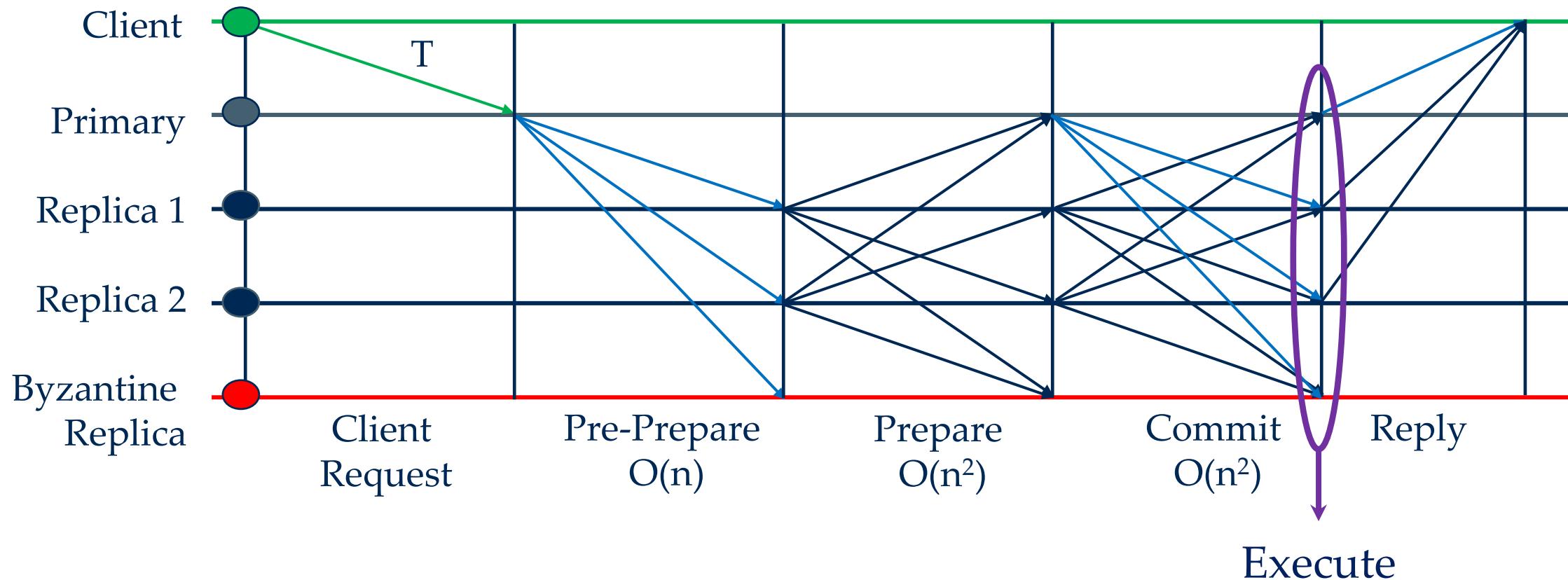


Practical Byzantine Fault-Tolerance (PBFT)

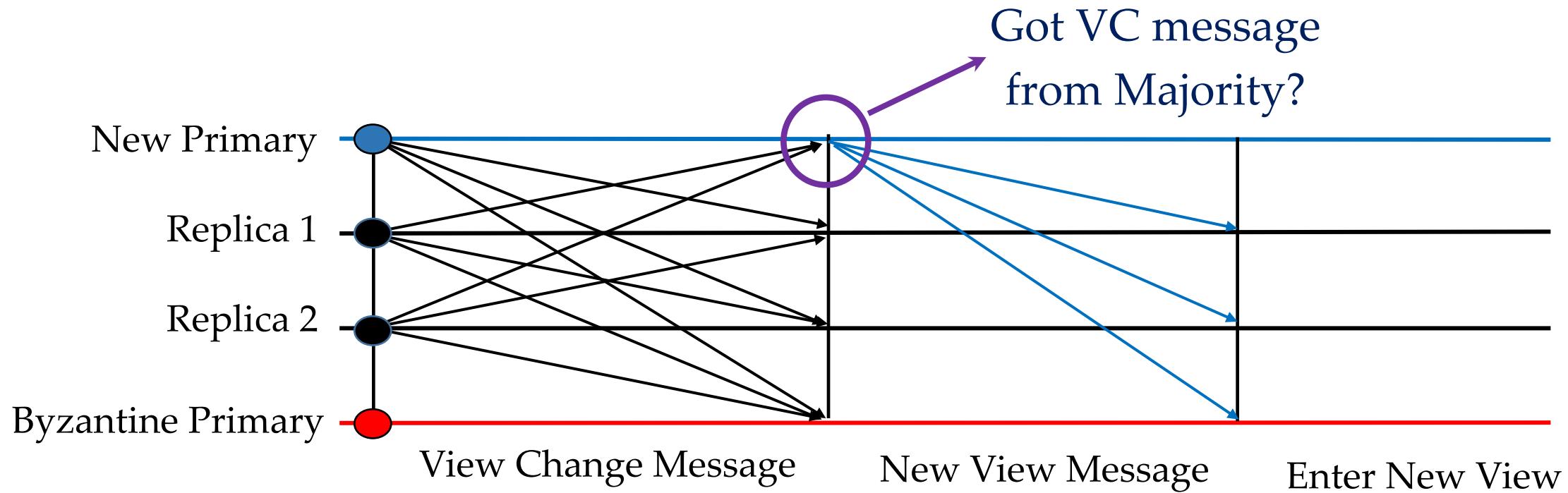
[OSDI'99]

- 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



PBFT Uncivil Execution: Primary Failure (View Change)

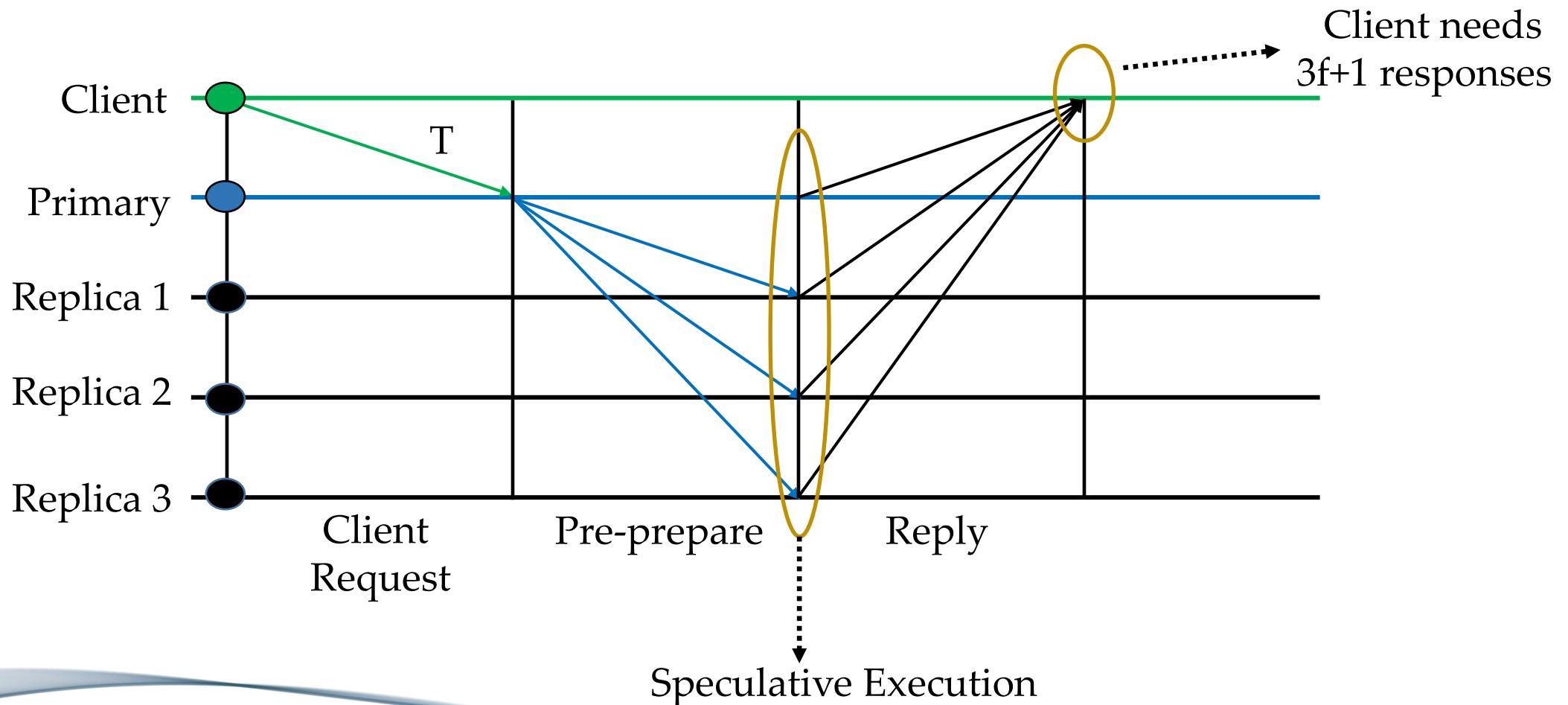


Speculative Byzantine Fault Tolerance (Zyzzyva)

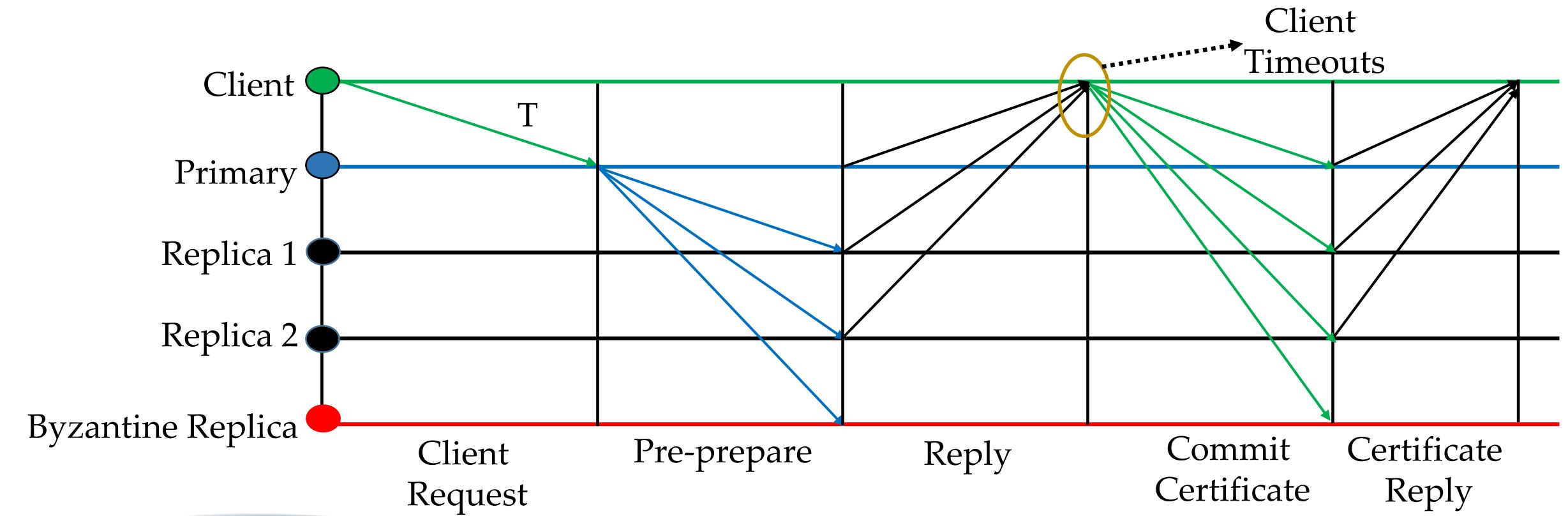
[SOSP'07]

- 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!

Zyzzya Civil Executions



Zyzyva under Failure of *one* Non-Primary Replica



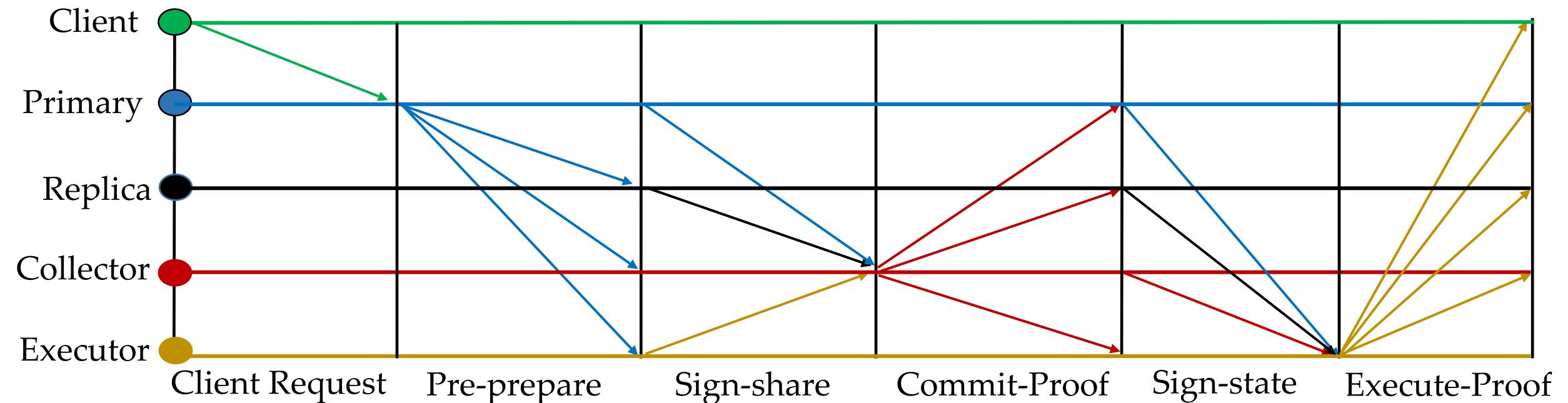
On client timeout → switches to *slow-path*.

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**.

SBFT Civil Execution



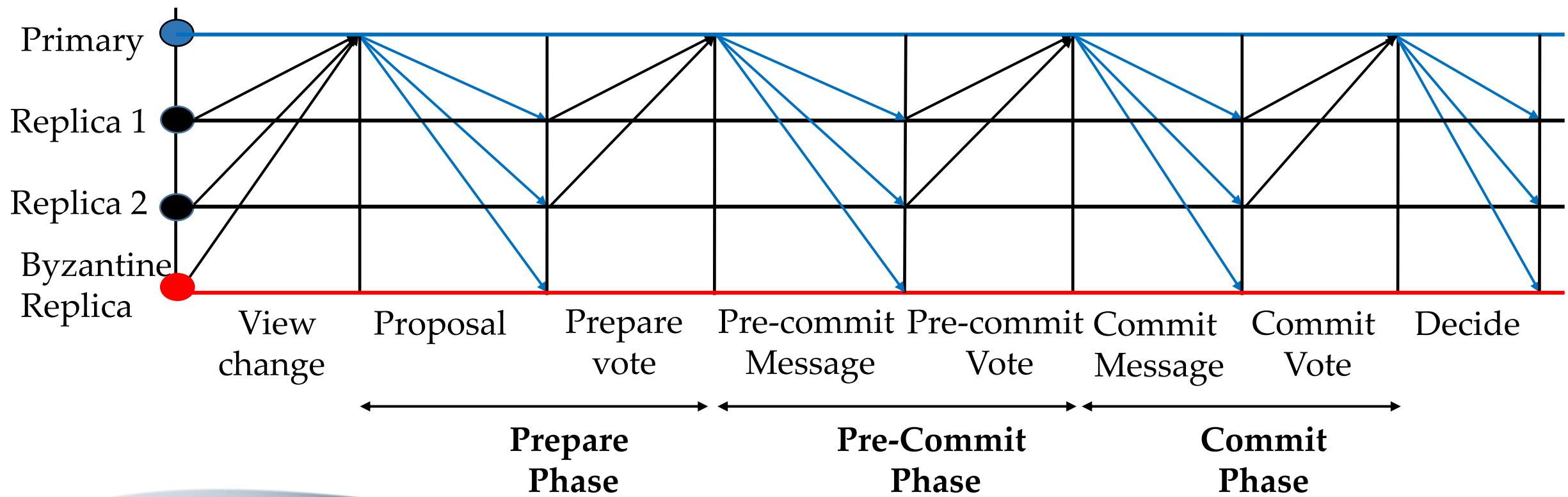
Either no failures or $c+1$ crash failures for $c > 0$ collectors if $n = 3f+2c+1$

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 → Frequent primary replacement.
- Employs threshold signatures to linearize consensus → 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.

Hotstuff Protocol



Other Proposed Byzantine-Fault Tolerant Designs

- 1) System consisting of $n \gg 3f+1$.
 - 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



UCDAVIS
UNIVERSITY OF CALIFORNIA



Proof-of-Execution (PoE)

Three-phase **Linear** protocol

Speculative Execution

Out-of-Order Message Processing

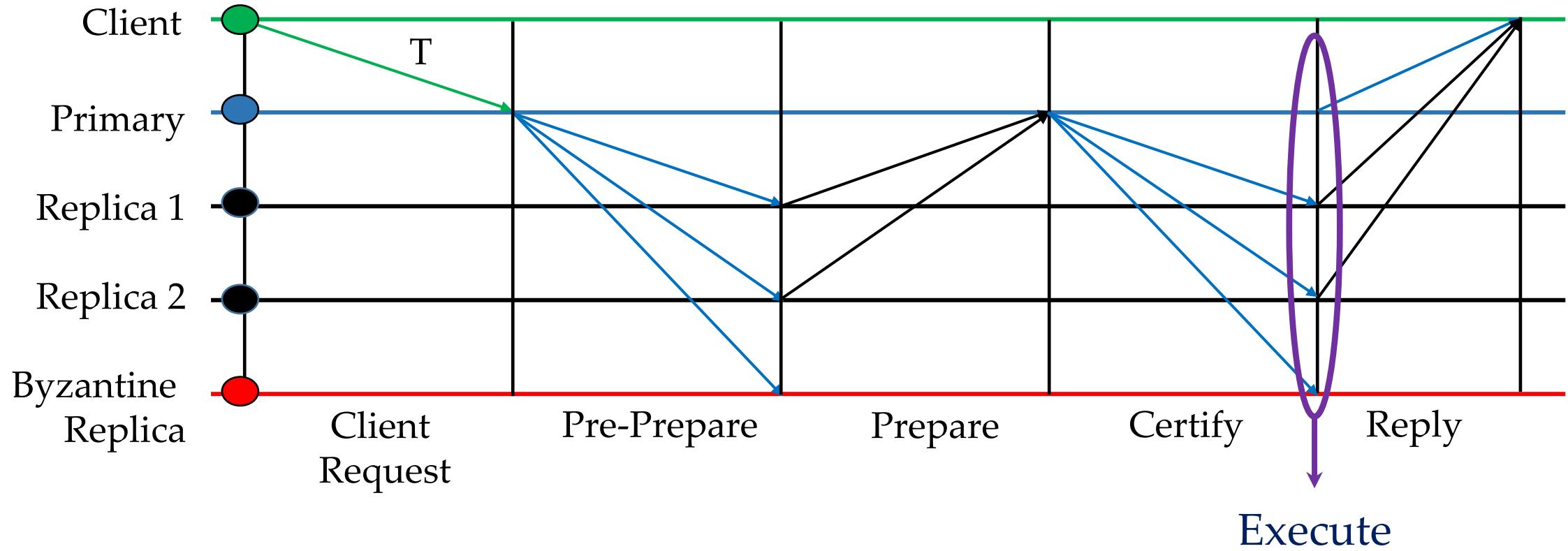
No dependence on clients or trusted component.

No reliance on a twin-path design.

PoE vs Other Protocols

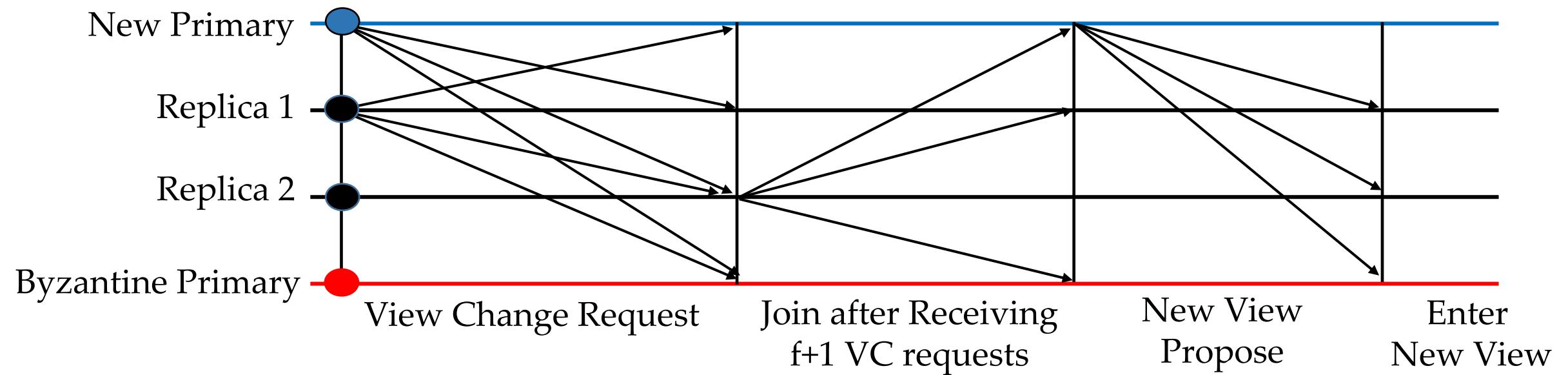
Protocol	Phases	Messages	Resilience	Requirements
ZYZZYVA	1	$\mathcal{O}(n)$	0	reliable clients and unsafe
PoE (our paper)	3	$\mathcal{O}(3n)$	f	sign. agnostic
PBFT	3	$\mathcal{O}(n + 2n^2)$	f	
HOTSTUFF	4	$\mathcal{O}(n + 3n^2)$	f	
HOTSTUFF-TS	8	$\mathcal{O}(8n)$	f	threshold sign.
SBFT	5	$\mathcal{O}(5n)$	0	threshold sign. and twin path

Proof-of-Execution (PoE)



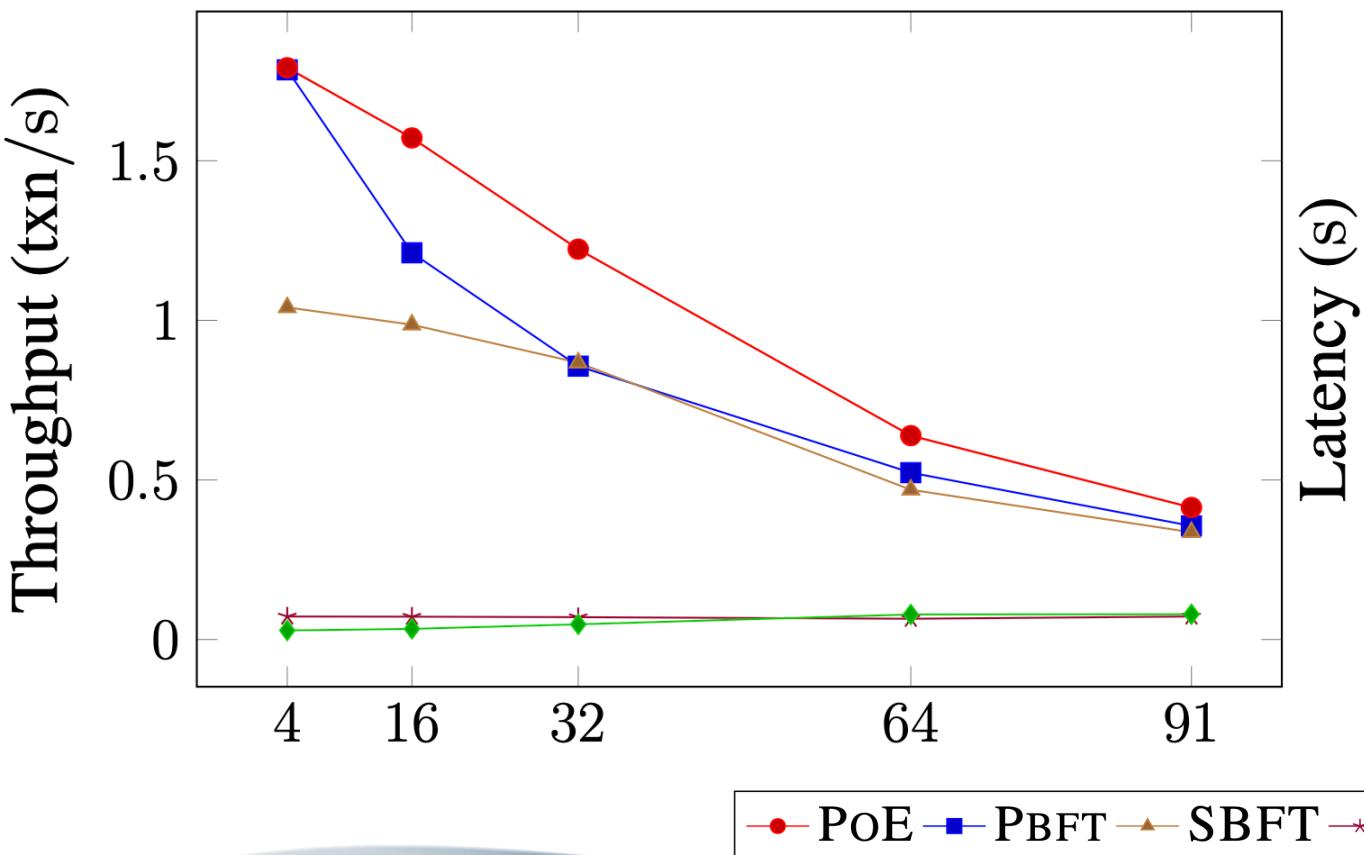
$n = 4$ replicas and $f \leq 1$

PoE View Change Protocol

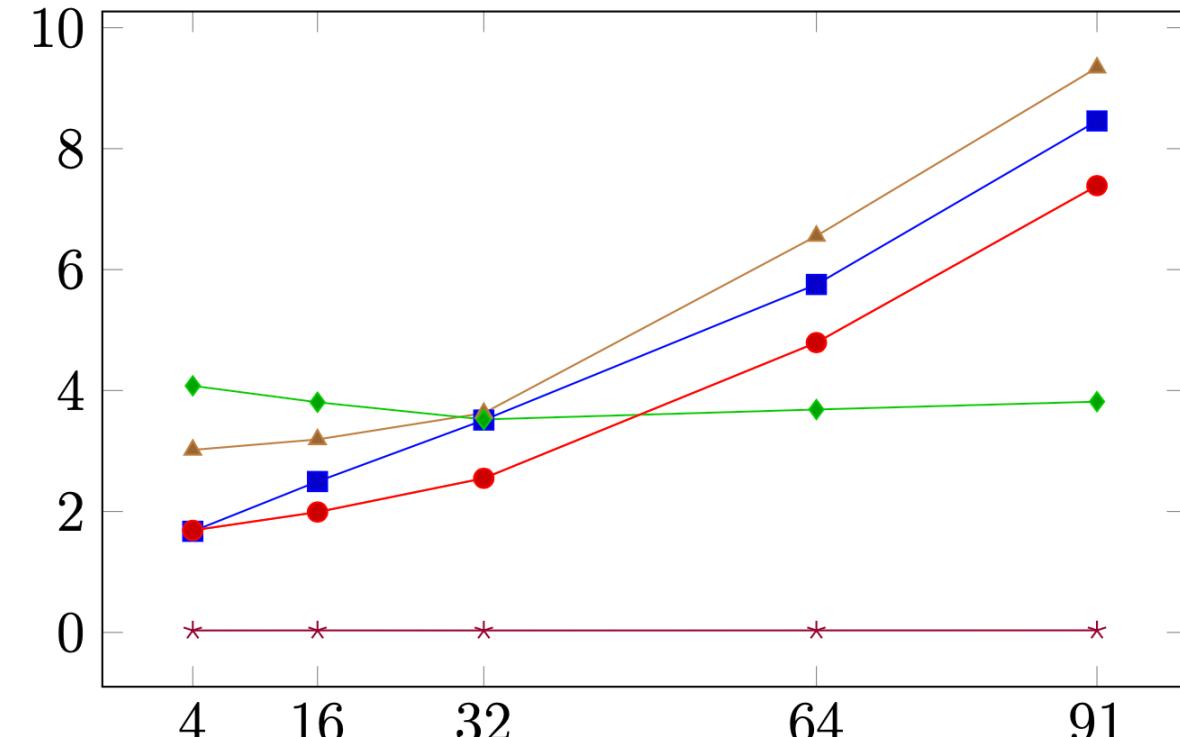


PoE Scalability under Single Failure

. 10^5 (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.

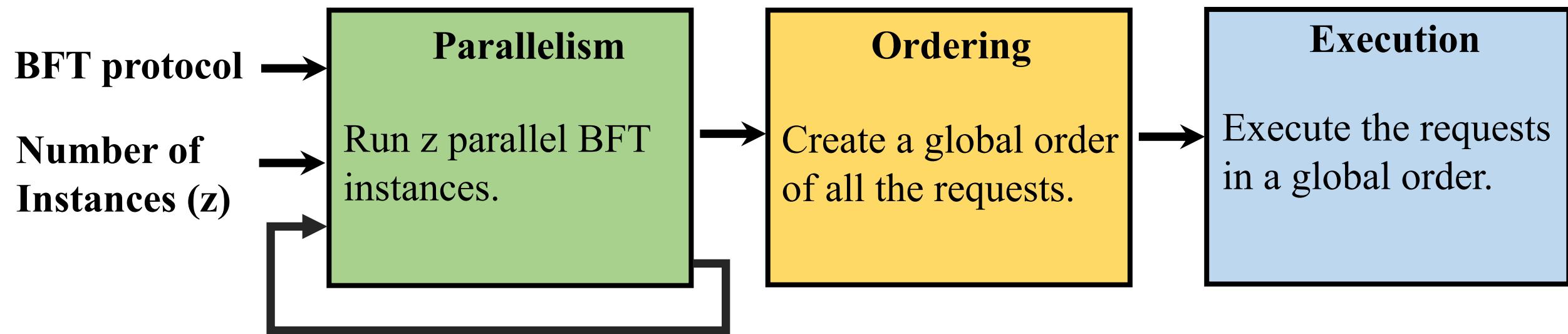
RCC Defense

Why should BFT protocols rely on just *one* primary replica?

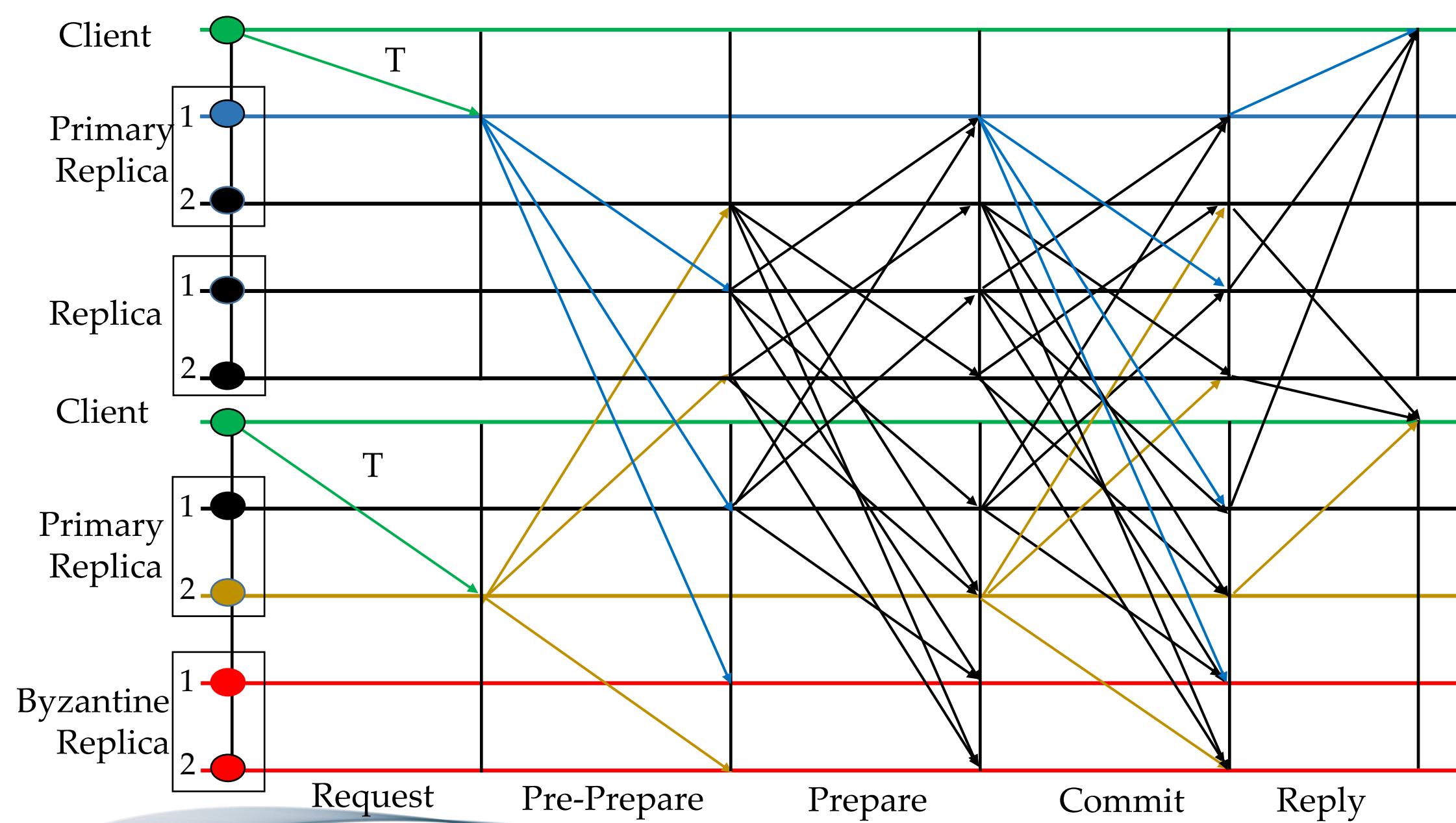
Malicious primary can *throttle* the system throughput.

Malicious primary requires *replacement* → fall in throughput.

Resilient Concurrency Control Paradigm



RCC can employ several BFT protocols: PBFT, Zyzzyva, SBFT and PoE.

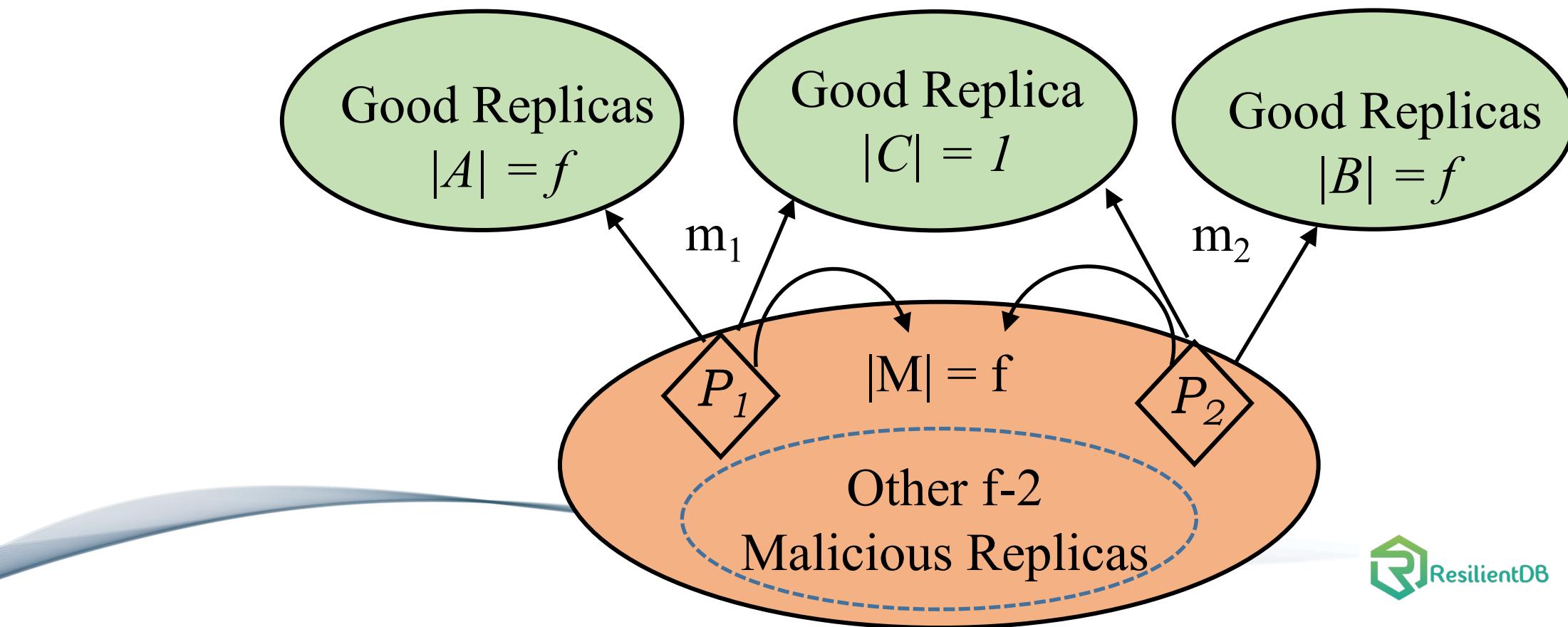


RCC using PBFT with 2 parallel instances on each replica

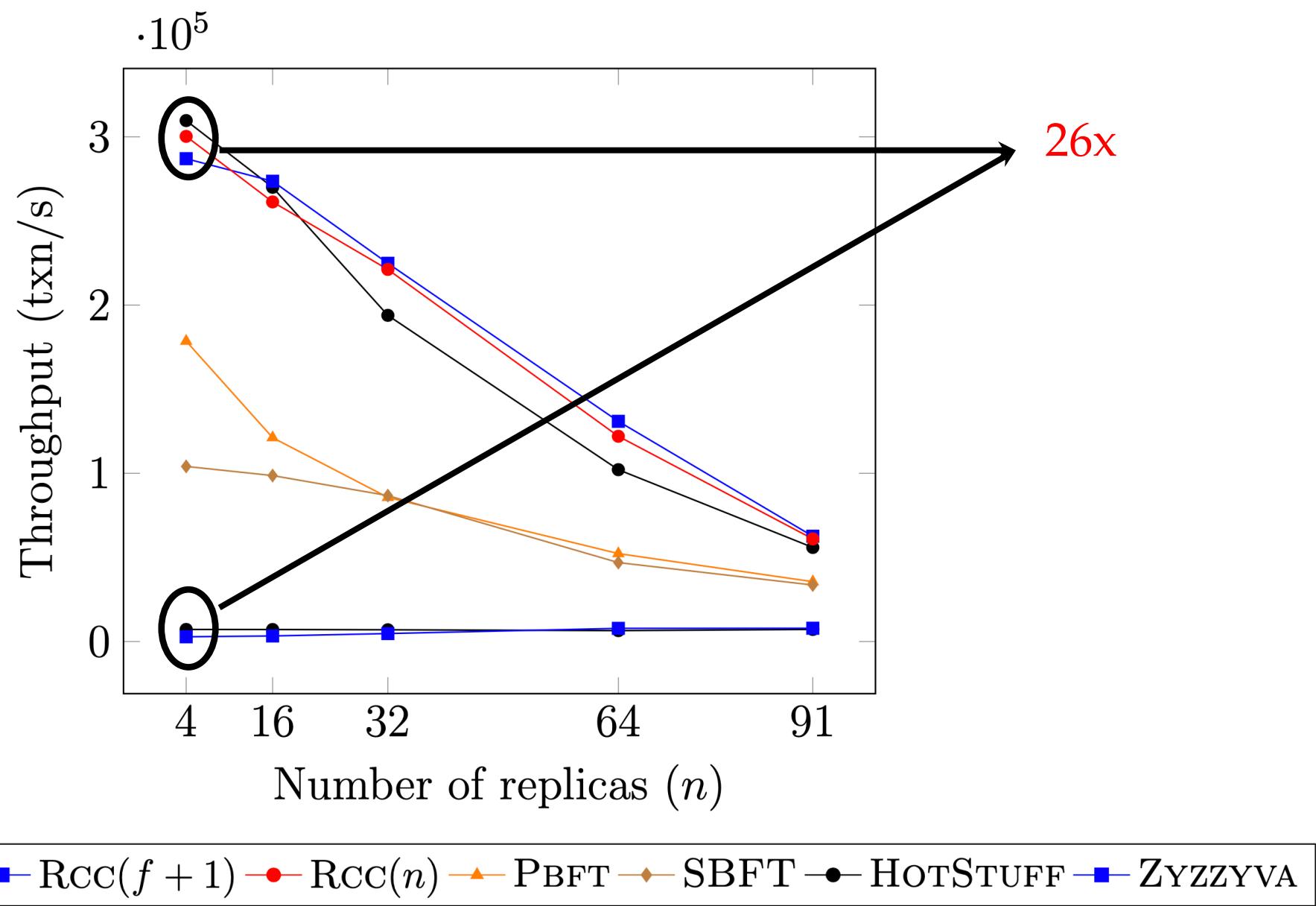
Colluding Primaries

Multiple malicious primaries can prevent **liveness!**

Solution → Optimistic Recovery through State Exchange.



Scalability

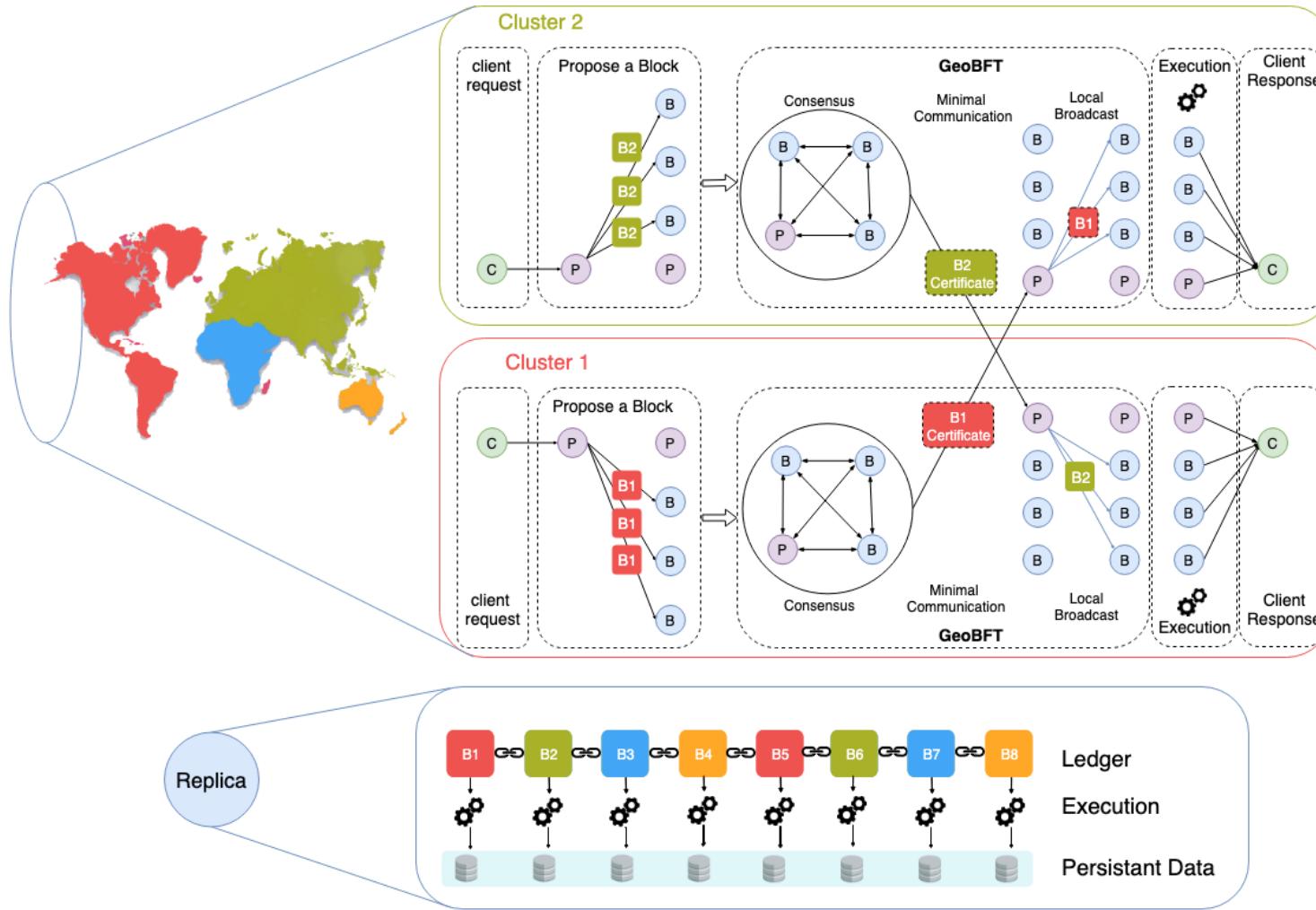


Single Failure Experiments

Global Scale Resilient Blockchain Fabric

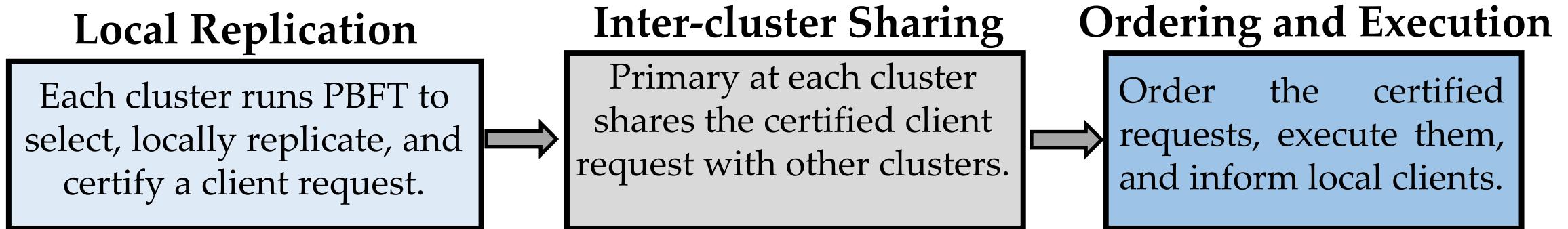
- Traditional BFT protocols do not scale to geographically large distances.
- Blockchain requires decentralization → replicas can be far apart → expensive communication!
- The underlying BFT consensus protocol should be topology-aware.

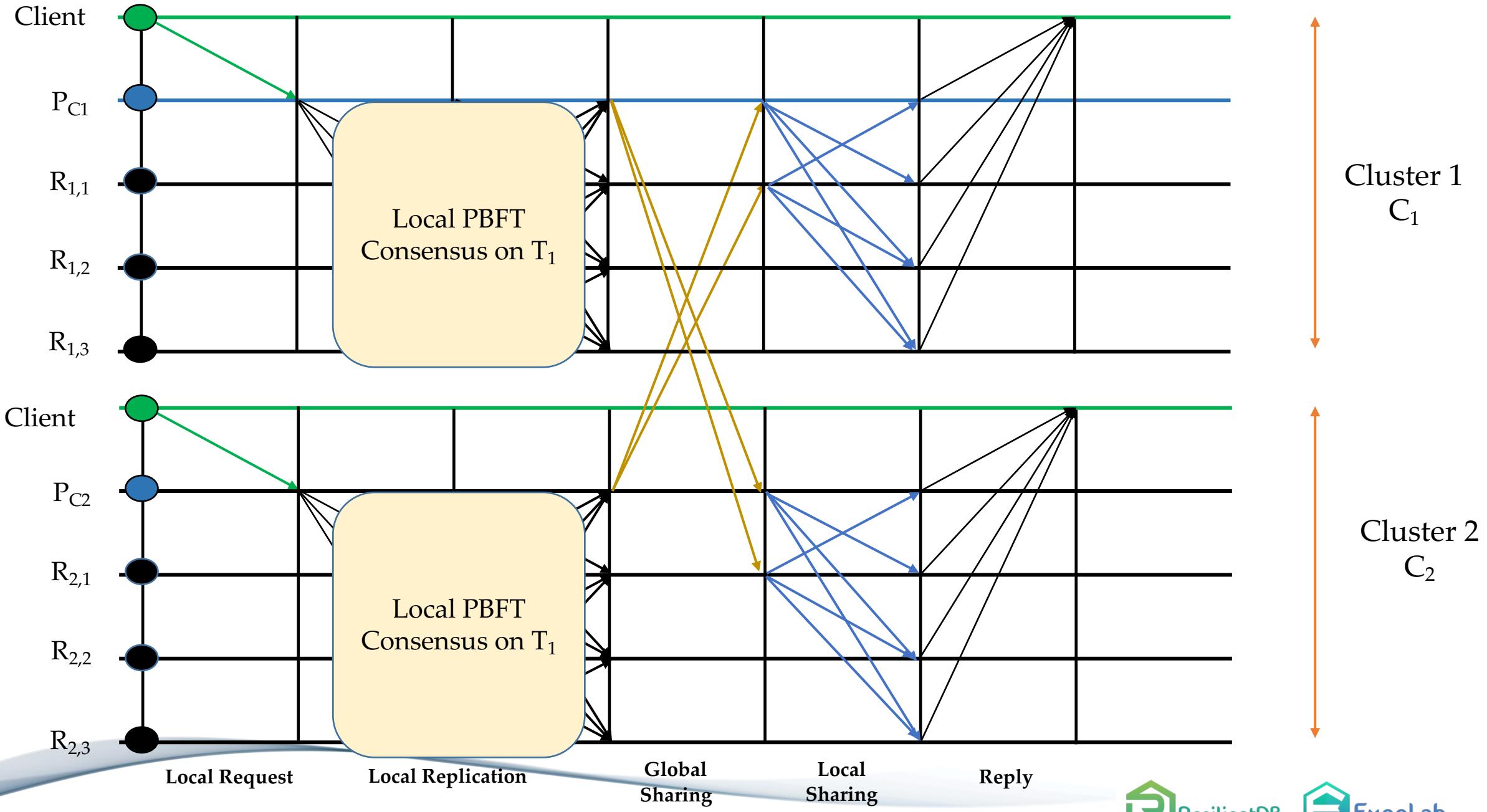
Vision Geo-Scale Byzantine Fault-Tolerance



GeoBFT Protocol

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

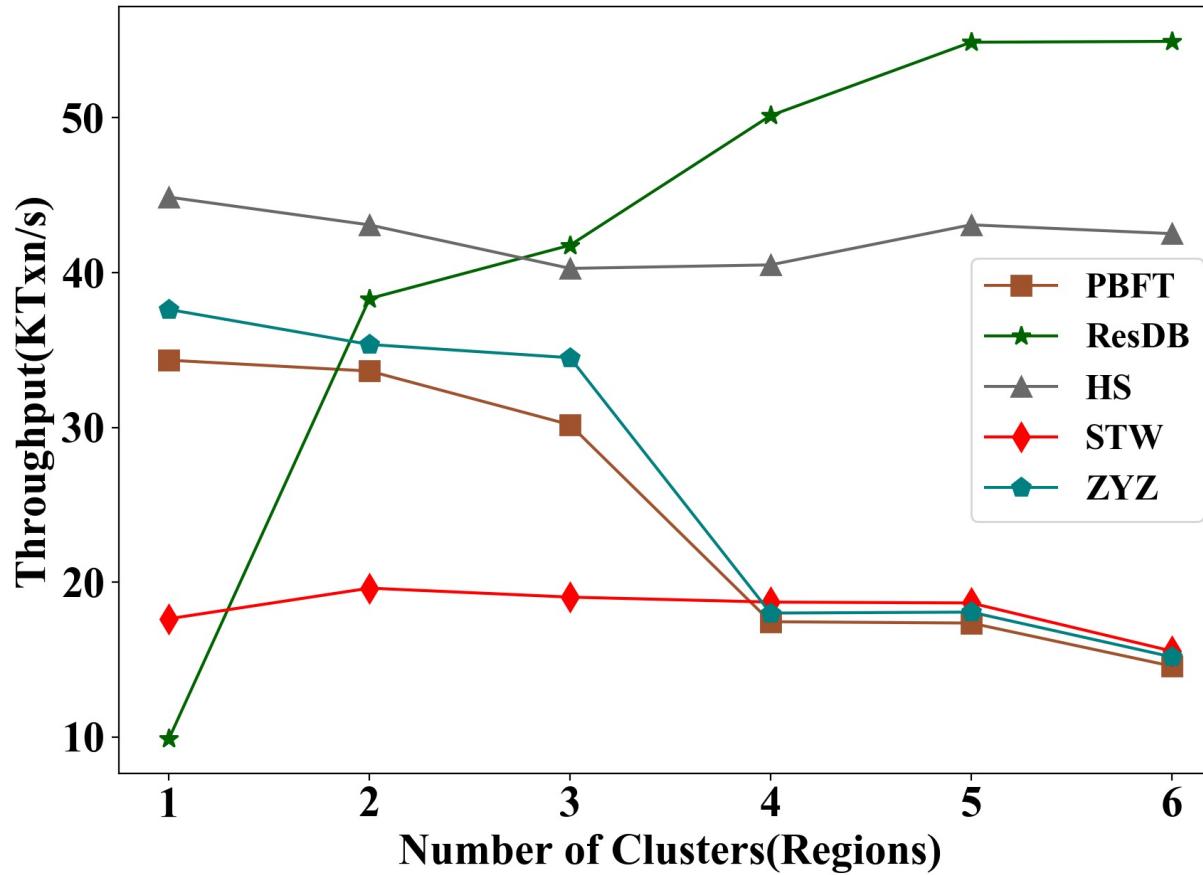




GeoBFT Takeaways

- 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



Permissioned Blockchain Through the Looking Glass: Architectural and Implementation Lessons Learned

Visit at: <https://resilientdb.com/>



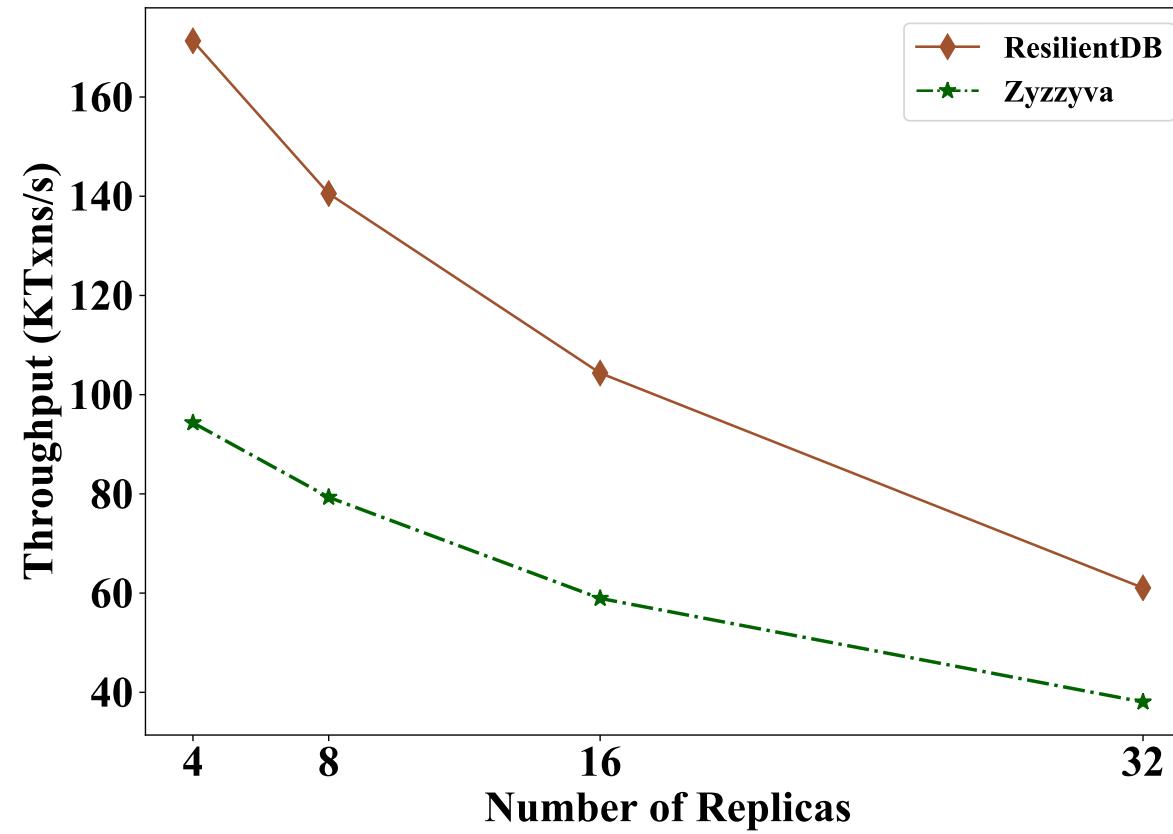
Why Should You Choose ResilientDB?

- 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.

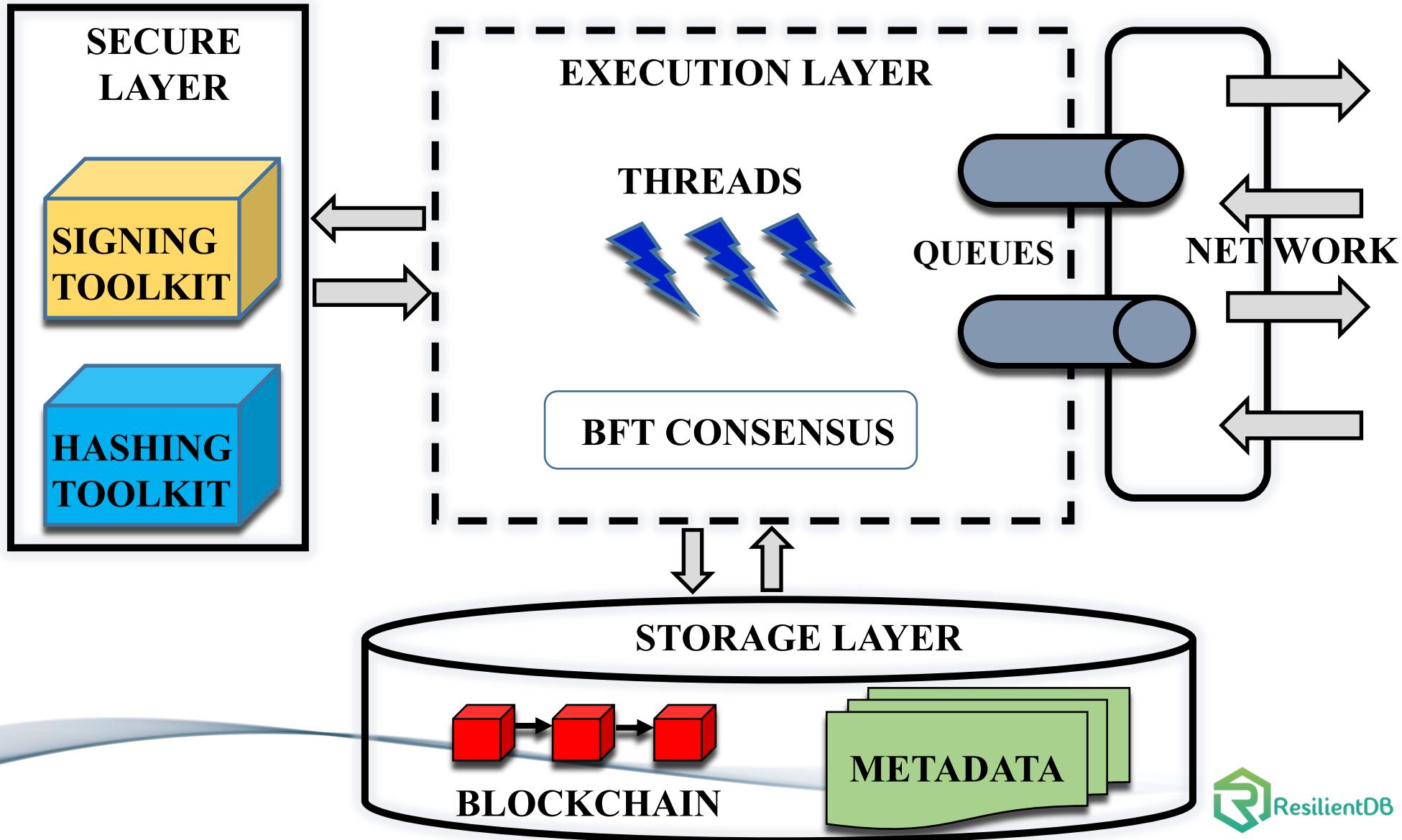
Dissecting Existing Permissioned Blockchains

- 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

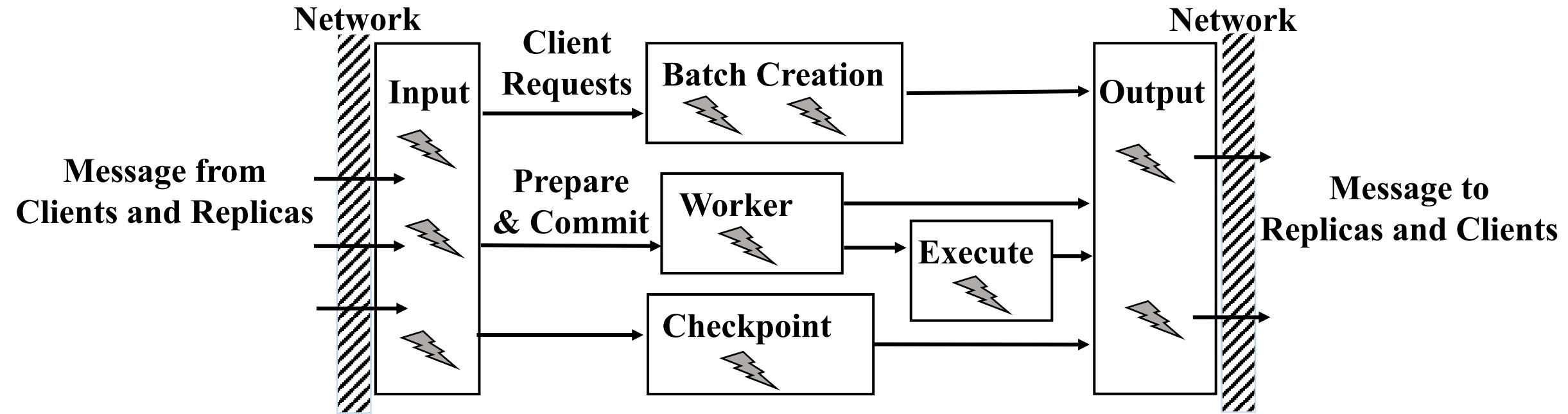
Can a well-crafted system based on a classical BFT protocol outperform a modern protocol?



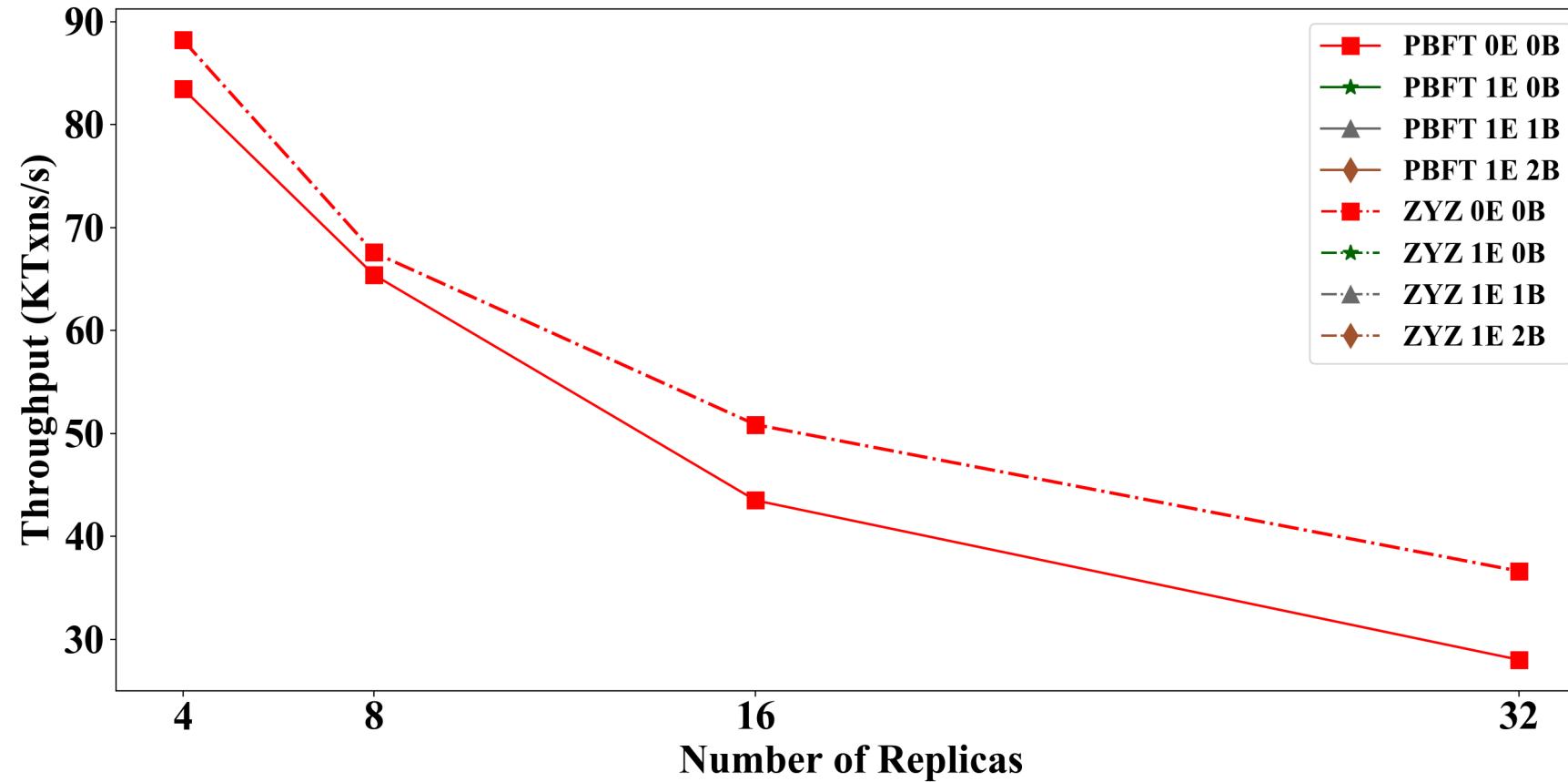
ResilientDB Architecture



ResilientDB Multi-Threaded Deep Pipeline

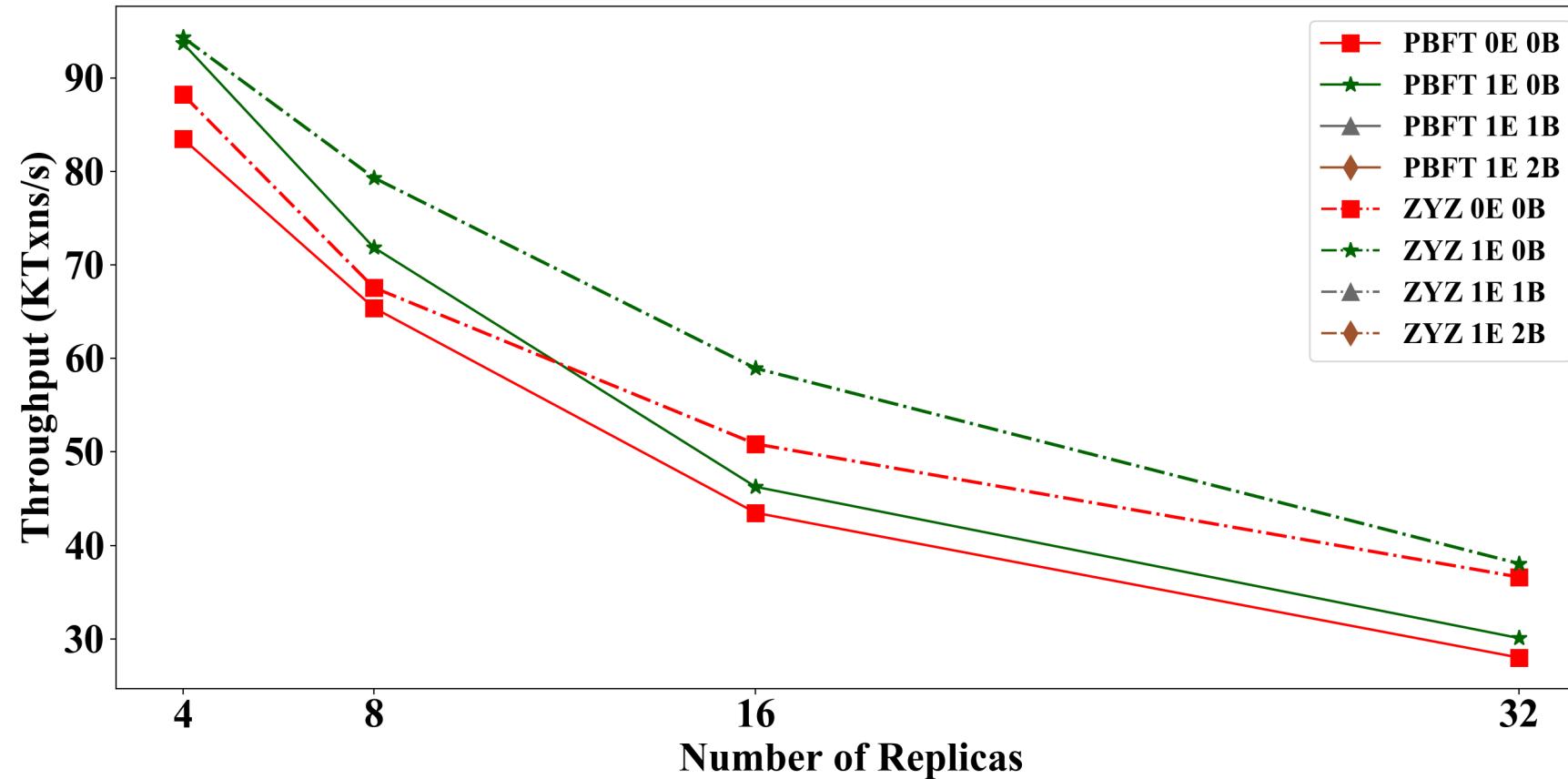


Insight 1: Multi-Threaded pipeline Gains



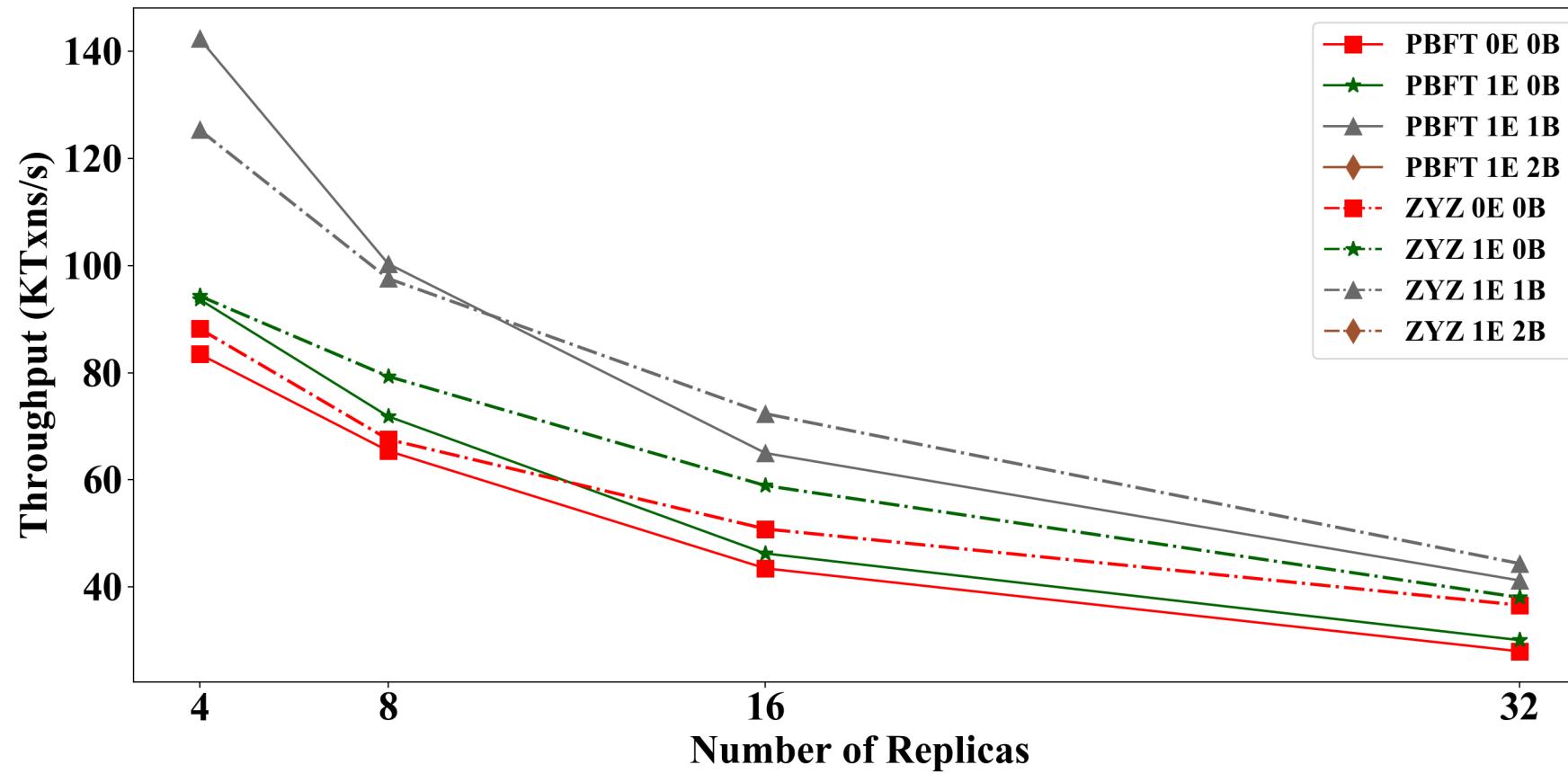
Parallelizing and Pipelining tasks across worker, execution (E) and batch-threads (B).

Insight 1: Multi-Threaded pipeline Gains



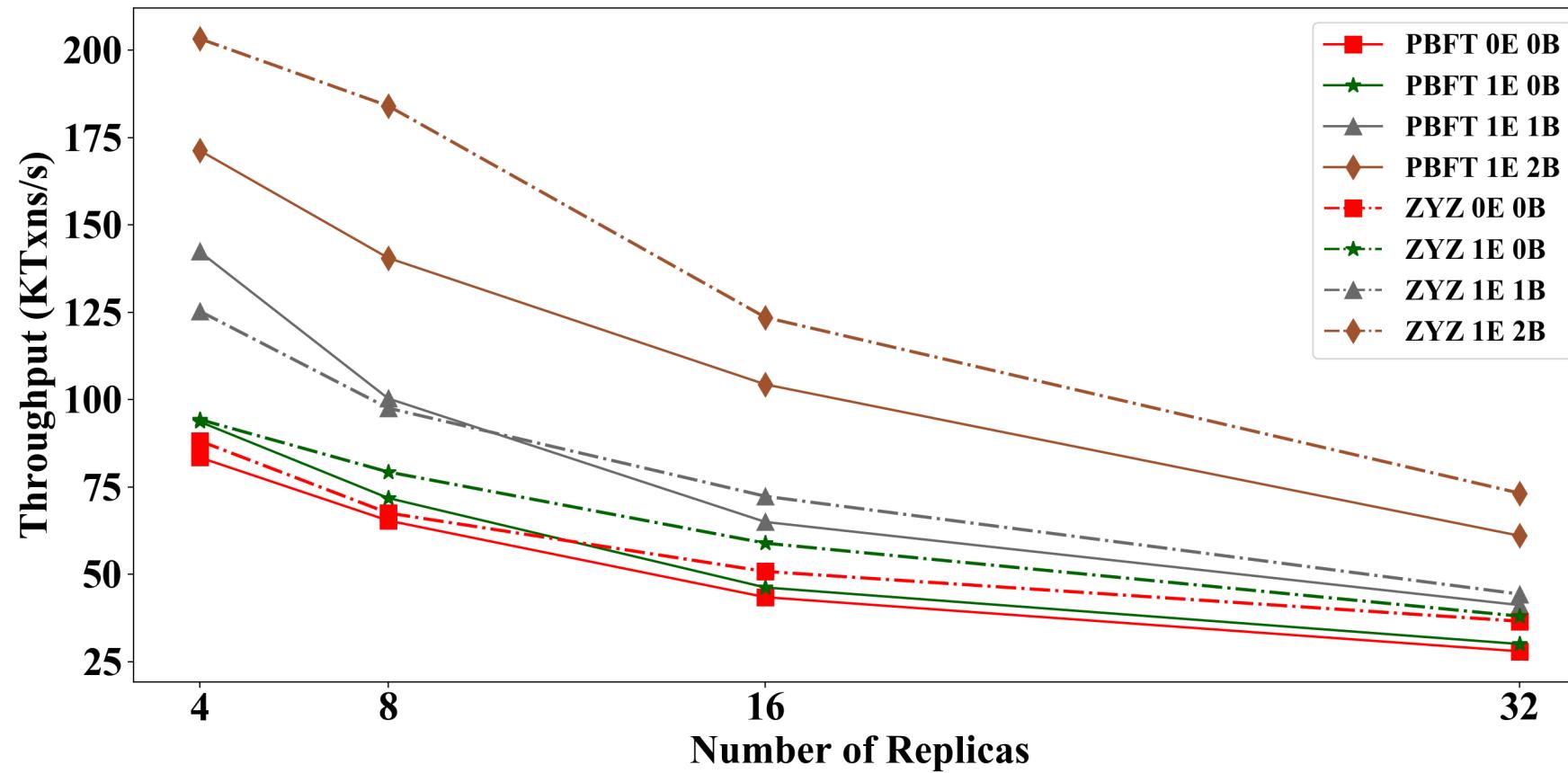
Parallelizing and Pipelining tasks across worker, execution (E) and batch-threads (B).

Insight 1: Multi-Threaded pipeline Gains



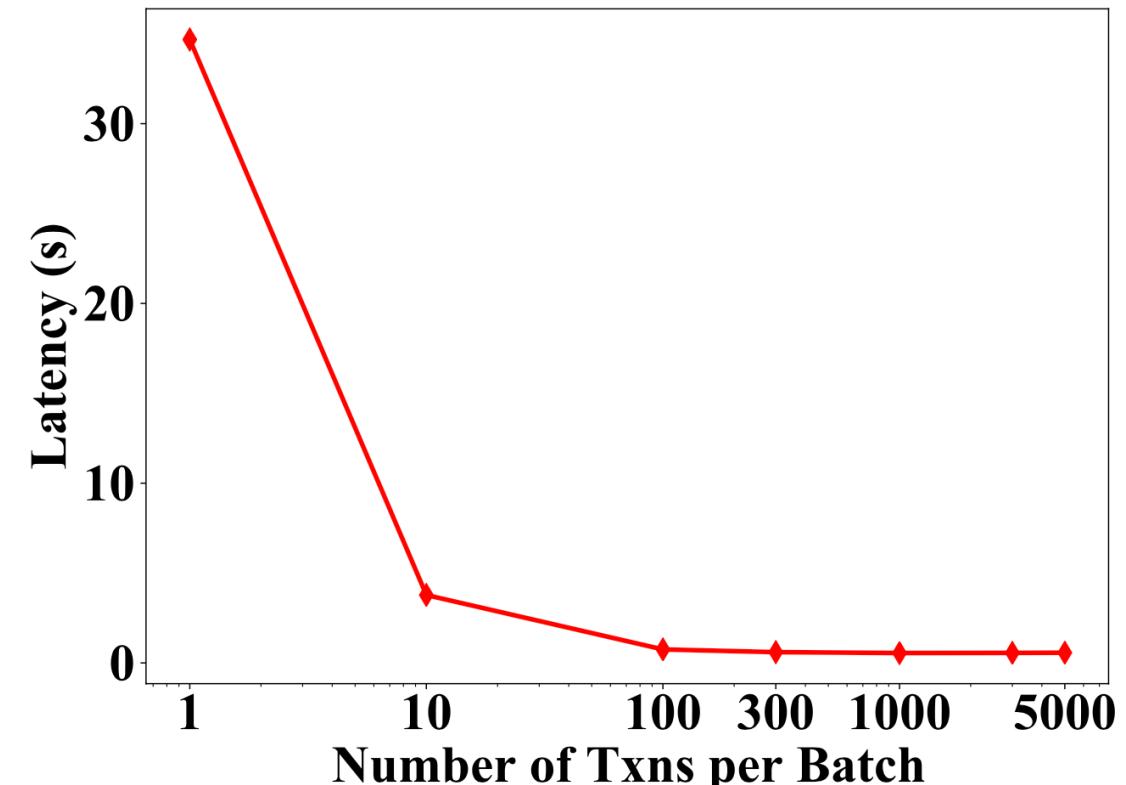
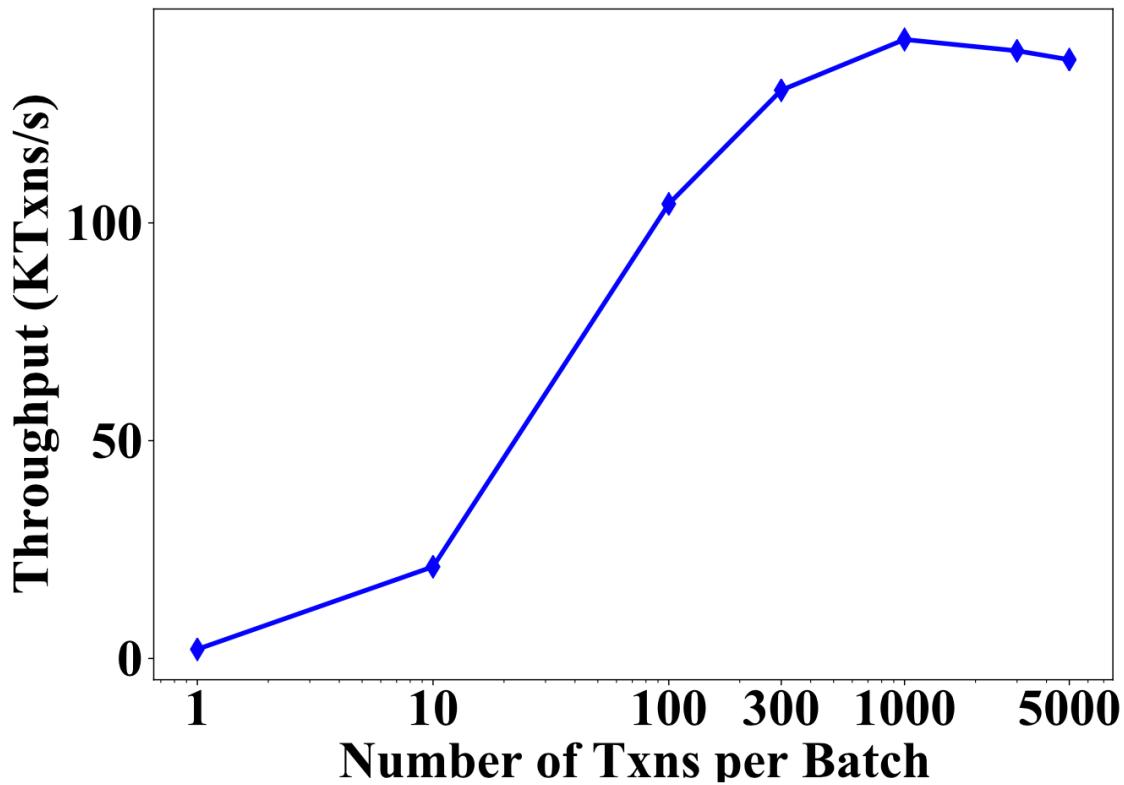
Parallelizing and Pipelining tasks across worker, execution (E) and batch-threads (B).

Insight 1: Multi-Threaded pipeline Gains



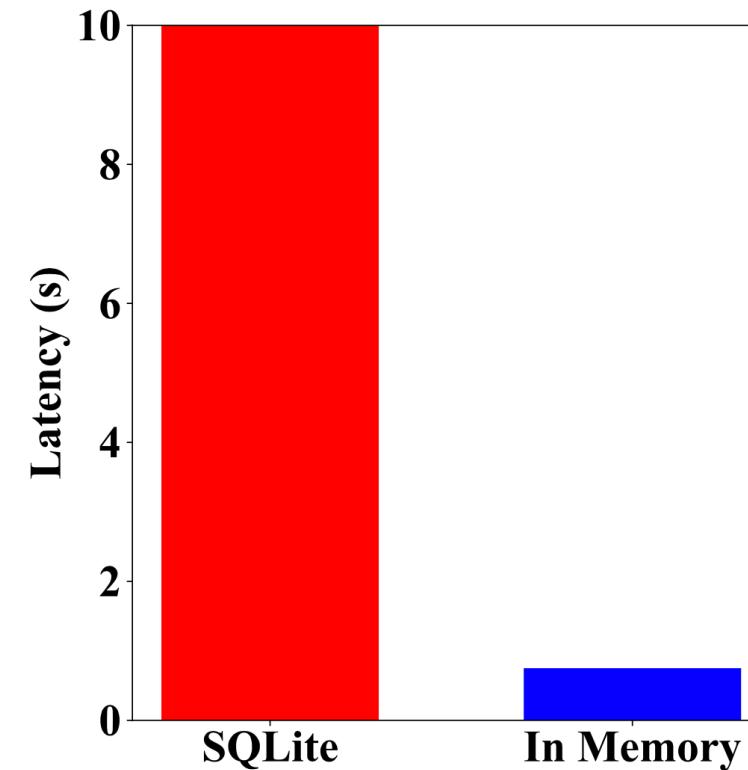
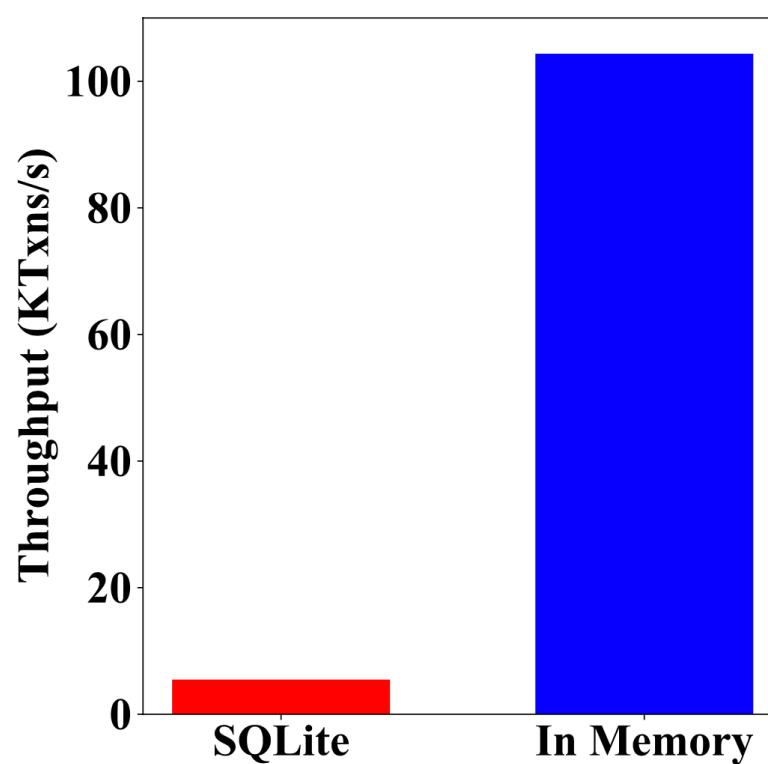
Parallelizing and Pipelining tasks across worker, execution (E) and batch-threads (B).

Insight 2: Optimal Batching Gains



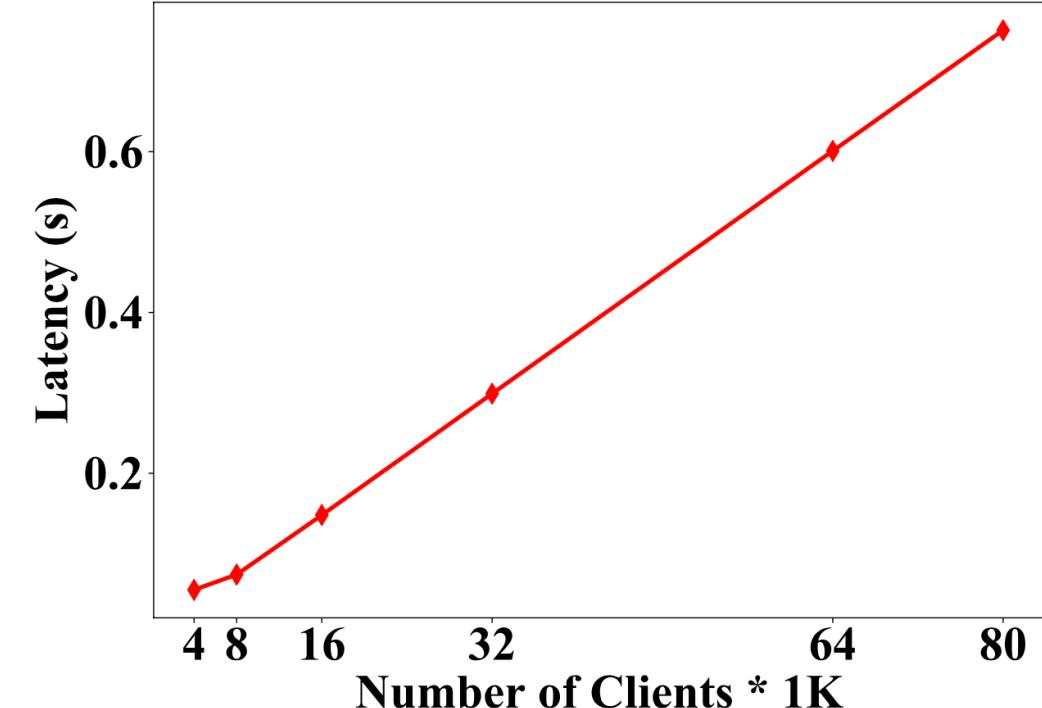
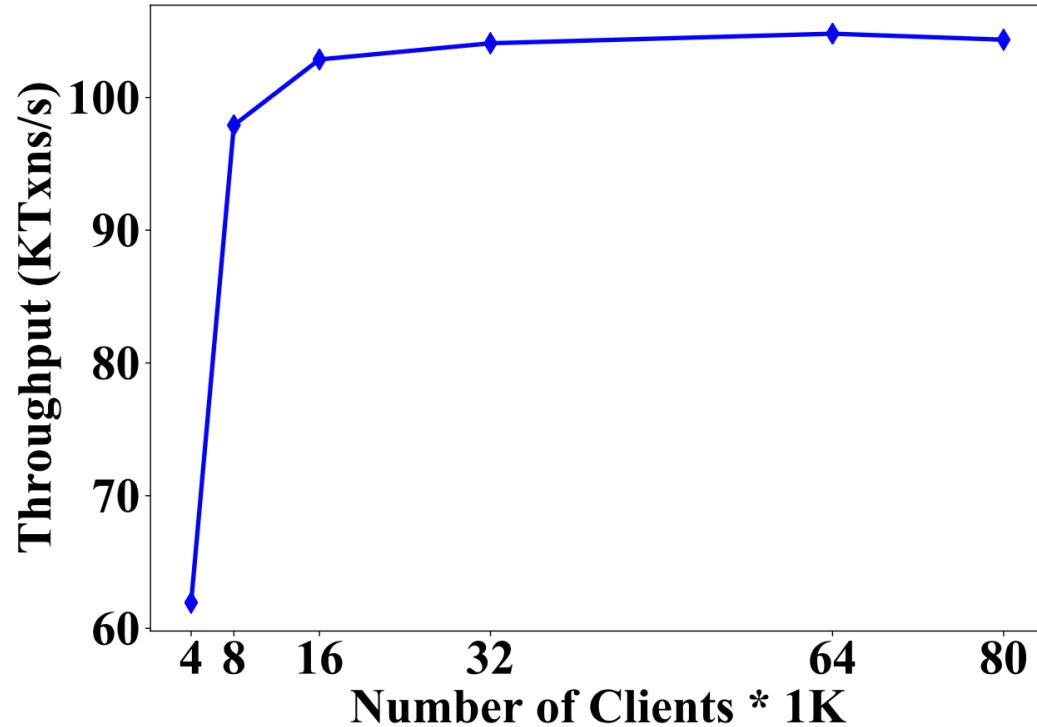
More transactions batched together → increase in throughput
→ reduced phases of consensus.

Insight 3: Memory Storage Gains



In-memory blockchain storage → reduces access cost.

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?

resilientdb / resilientdb

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

Watch 5 Unstar 11 Fork 13

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

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?

*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

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

```
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  Send problem report to the developers?
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
```

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

```
C++ client_main.cpp ×  
client > C++ client_main.cpp > ...  
31 int main(int argc, char *argv[]){  
32     printf("Running client...\n\n");  
33     // 0. initialize global data structure  
34     parser(argc, argv);  
35     assert(g_node_id >= g_node_cnt);  
36     uint64_t seed = get_sys_clock();  
37     srand(seed);  
38     printf("Random seed: %ld\n", seed);  
39  
40     int64_t starttime;  
41     int64_t endtime;  
42     starttime = get_server_clock();  
43     // per-partition malloc  
44     printf("Initializing stats... ");  
45     fflush(stdout);  
46     stats.init(g_total_client_thread_cnt);  
47     printf("Done\n");  
48     printf("Initializing transport manager... ");  
49     fflush(stdout);  
50     tport_man.init();  
51     printf("Done\n");  
52     printf("Initializing client manager... ");  
53     Workload *m_wl = new YCSBWorkload;  
54     m_wl->Workload::init();  
55 }
```

```
C++ client_thread.cpp ×  
system > C++ client_thread.cpp > ...  
79  
80 RC ClientThread::run()  
81 {  
82     tsetup();  
83     printf("Running ClientThread %d\n", _thd_id);  
84  
85     while (true)  
86     {  
87         keyMTX.lock();  
88         if (keyAvail)  
89         {  
90             keyMTX.unlock();  
91             break;  
92         }  
93         keyMTX.unlock();  
94     }  
95  
96     BaseQuery *m_query;  
97     uint64_t iters = 0;  
98     uint32_t num_txns_sent = 0;  
99     int txns_sent[g_node_cnt];  
100    for (uint32_t i = 0; i < g_node_cnt; ++i)  
101        txns_sent[i] = 0;  
102  
103    run_starttime = get_sys_clock();  
104 }
```

PBFT: Practical Byzantine Fault Tolerance

Process Messages

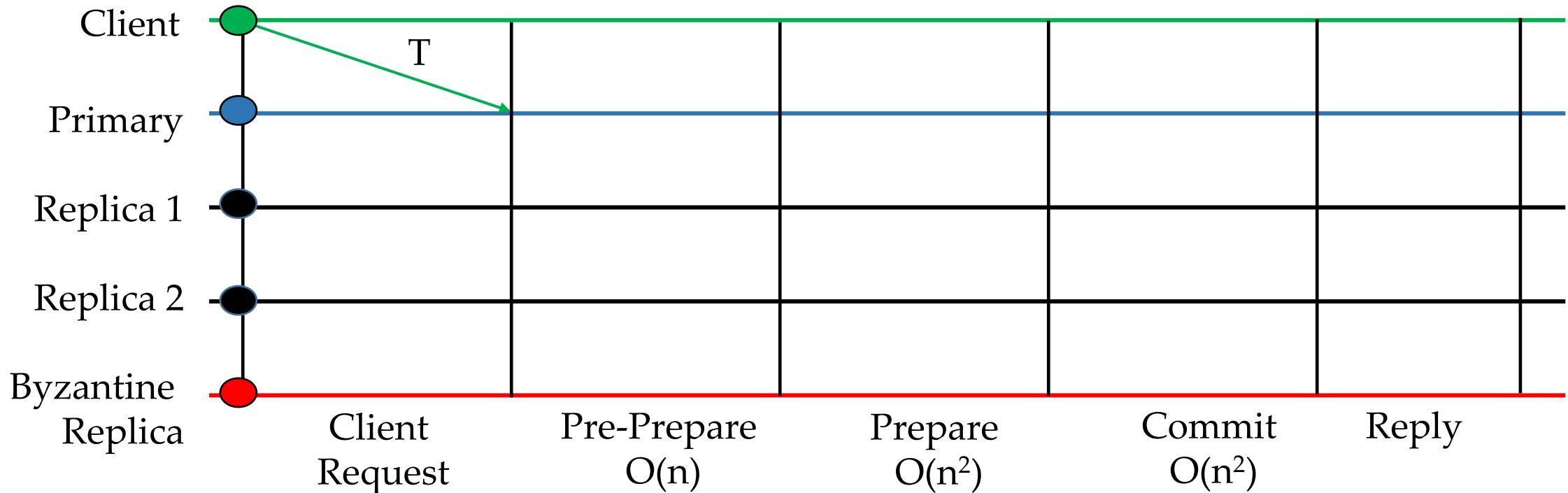
- Transport/message.cpp
- System/worker_thread.cpp
- System/worker_thread_pbft.cpp
- Worker Thread: Run function
- Worker Thread: Process function

```
C++ worker_thread.cpp ×
system > C++ worker_thread.cpp > WorkerThread::run()

626 /**
627  * Starting point for each worker thread.
628 *
629 * Each worker-thread created in the main() starts here. Each worker-thread is alive
630 * till the time simulation is not done, and continuously perform a set of tasks.
631 * These tasks involve, dequeuing a message from its queue and then processing it
632 * through call to the relevant function.
633 */
634 RC WorkerThread::run()
635 {
636     tsetup();
637     printf("Running WorkerThread %d\n", _thd_id);
638
639     uint64_t agCount = 0, readystarttime, idlestarttime = 0;
640
641     // Setting batch (only relevant for batching threads).
642     next_set = 0;
643
644     while (!simulation->is_done())
645     {
646         txn_man = NULL;
647         heartbeat();
648         progress_stats();
649
650 #if VIEW_CHANGES
651         // Thread 0 continuously monitors the timer for each batch.
652         if (get_thd_id() == 0)
653         {
654             check_for_timeout();
655         }
656
657         if (g_node_id != get_current_view(get_thd_id()))
658         {
659             check_switch_view();
660         }
661 #endif
662
663         // Dequeue a message from its work_queue.
664         Message *msg = work_queue.dequeue(get_thd_id());
```

```
C++ worker_thread.cpp ×
system > C++ worker_thread.cpp > WorkerThread::process(Message *)
87 void WorkerThread::process(Message *msg)
88 {
89     RC rc __attribute__((unused));
90
91     switch (msg->get_rtype())
92     {
93     case KEYEX:
94         rc = process_key_exchange(msg);
95         break;
96     case CL_BATCH:
97         rc = process_client_batch(msg);
98         break;
99     case BATCH_REQ:
100        rc = process_batch(msg);
101        break;
102    case PBFT_CHKPT_MSG:
103        rc = process_pbft_chkpt_msg(msg);
104        break;
105    case EXECUTE_MSG:
106        rc = process_execute_msg(msg);
107        break;
108 #if VIEW_CHANGES
109    case VIEW_CHANGE:
110        rc = process_view_change_msg(msg);
111        break;
112    case NEW_VIEW:
113        rc = process_new_view_msg(msg);
114        break;
115 #endif
116    case PBFT_PREP_MSG:
117        rc = process_pbft_prep_msg(msg);
118        break;
119    case PBFT_COMMIT_MSG:
120        rc = process_pbft_commit_msg(msg);
121        break;
122    default:
123        printf("Msg: %d\n", msg->get_rtype());
124        fflush(stdout);
125        assert(false);
126    }
127 }
```

PBFT Failure-Free Flow



PBFT: Practical Byzantine Fault Tolerance

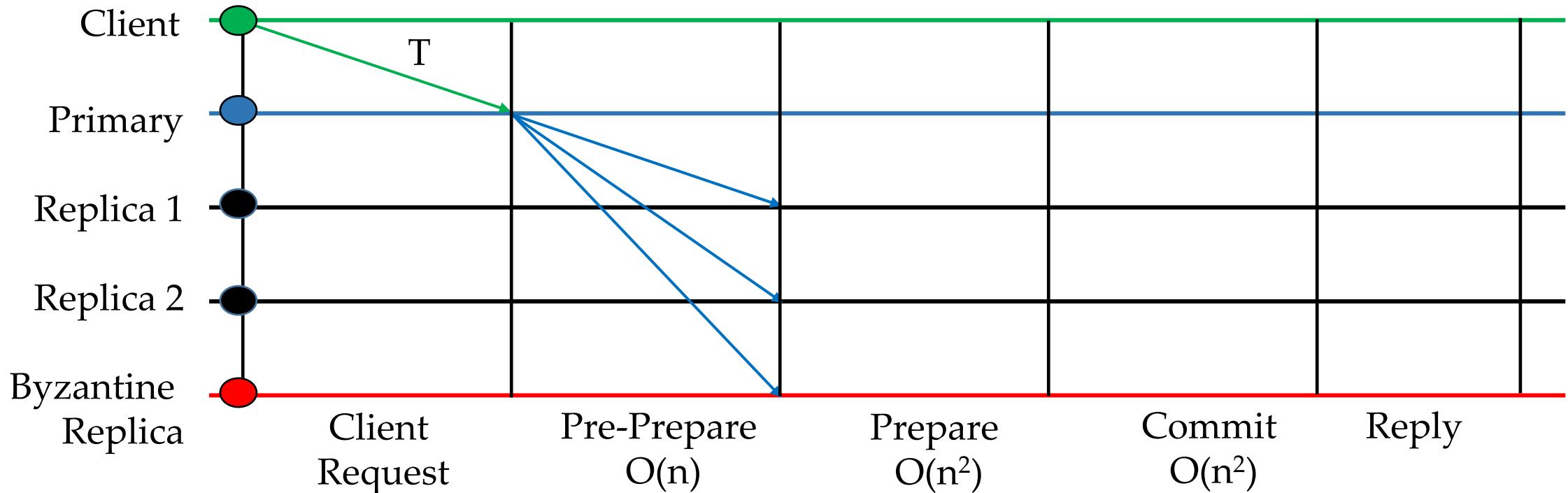
Process Client Message

- 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

```
C++ worker_thread_pbft.cpp ×
system > C++ worker_thread_pbft.cpp > ...
18 /**
19  * Processes an incoming client batch and sends a Pre-prepare message to all replicas.
20  *
21  * This function assumes that a client sends a batch of transactions and
22  * for each transaction in the batch, a separate transaction manager is created.
23  * Next, this batch is forwarded to all the replicas as a BatchRequests Message
24  * which corresponds to the Pre-Prepare stage in the PBFT protocol.
25  *
26  * @param msg Batch of Transactions of type ClientQueryBatch from the client.
27  * @return RC
28 */
29 RC WorkerThread::process_client_batch(Message *msg)
30 {
31     //printf("ClientQueryBatch: %ld, THD: %ld :: CL: %ld :: RQ: %ld\n", msg->txns->size(), msg->txns->begin()->thd_id, msg->txns->begin()->cl_id, msg->txns->begin()->rq_id);
32     //fflush(stdout);
33
34     ClientQueryBatch *clbtch = (ClientQueryBatch *)msg;
35
36     // Authenticate the client signature.
37     validate_msg(clbtch);
38
39 #if VIEW_CHANGES
40     // If message forwarded to the non-primary.
41     if (g_node_id != get_current_view(get_thd_id()))
42     {
43         client_query_check(clbtch);
44         return RCOKE;
45     }
46
47     // Partial failure of Primary 0.
48     fail_primary(msg, 9);
49 #endif
50
51     // Initialize all transaction managers and uint64_t Message::txnid.
52     create_and_send_batchreq(clbtch, clbtch->txnid);
53
54     return RCOKE;
55 }
```

```
C++ worker_thread.cpp ×
system > C++ worker_thread.cpp > WorkerThread::create_and_send_batchreq(ClientQueryBatch *, uint64_t)
1123 /**
1124  * This function is used by the primary replicas to create and set
1125  * transaction managers for each transaction part of the ClientQueryBatch message
1126  * by the client. Further, to ensure integrity a hash of the complete batch is
1127  * generated, which is also used in future communication.
1128 */
1129 * @param msg Batch of transactions as a ClientQueryBatch message.
1130 * @param tid Identifier for the first transaction of the batch.
1131 */
1132 void WorkerThread::create_and_send_batchreq(ClientQueryBatch *msg, uint64_t tid)
1133 {
1134     // Creating a new BatchRequests Message.
1135     Message *bmsg = Message::create_message(BATCH_REQ);
1136     BatchRequests *breq = (BatchRequests *)bmsg;
1137     breq->init(get_thd_id());
1138
1139     // Starting index for this batch of transactions.
1140     next_set = tid;
1141
1142     // String of transactions in a batch to generate hash.
1143     string batchStr;
1144
1145     // Allocate transaction manager for all the requests in batch.
1146     for (uint64_t i = 0; i < get_batch_size(); i++)
1147     {
1148         uint64_t txn_id = get_next_txnid() + i;
1149
1150         //cout << "Txn: " << txn_id << " :: Thd: " << get_thd_id() << "\n";
1151         //fflush(stdout);
1152         txn_man = get_transaction_manager(txn_id, 0);
1153
1154         // Unset this txn man so that no other thread can concurrently use.
1155         while (true)
1156         {
1157             bool ready = txn_man->unset_ready();
1158             if (!ready)
1159             {
1160                 continue;
1161             }
1162             else
1163             {
1164                 break;
1165             }
1166         }
1167     }
1168 }
```

PBFT Failure-Free Flow



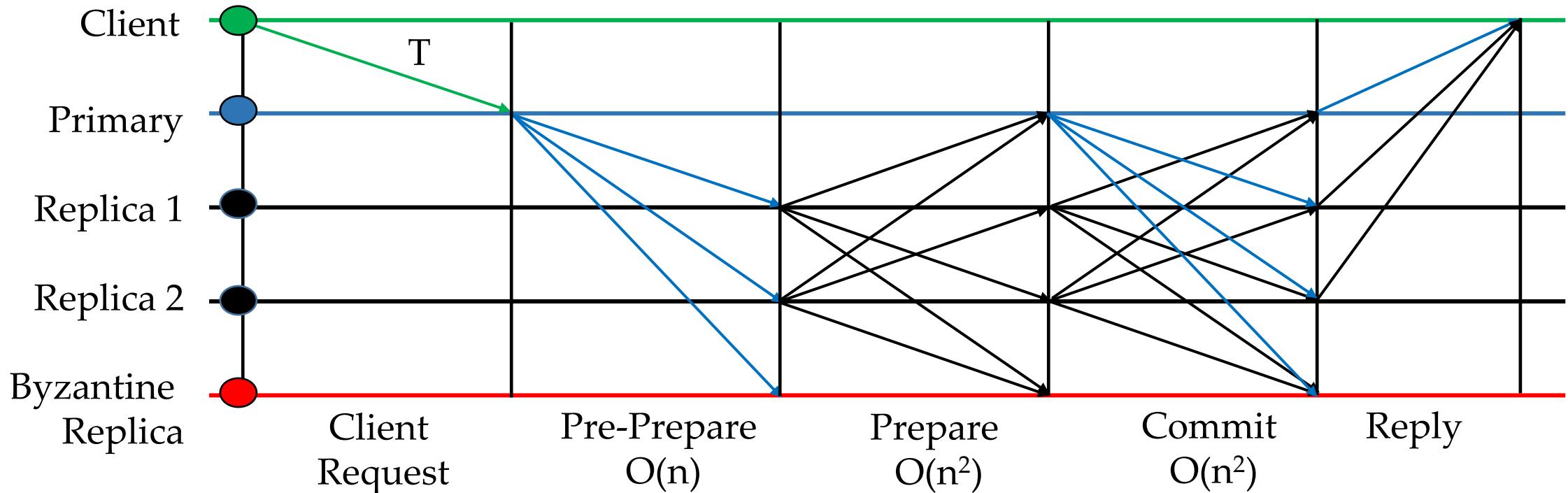
PBFT: Practical Byzantine Fault Tolerance

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

```
C++ worker_thread_pbft.cpp ×
system > C++ worker_thread_pbft.cpp > WorkerThread::process_batch(Message *)
57  /**
58   * Process incoming BatchRequests message from the Primary.
59   *
60   * This function is used by the non-primary or backup replicas to process an incoming
61   * BatchRequests message sent by the primary replica. This processing would require
62   * sending messages of type PBFTPrepMessage, which correspond to the Prepare phase of
63   * the PBFT protocol. Due to network delays, it is possible that a replica may have
64   * received some messages of type PBFTPrepMessage and PBFTCommitMessage, prior to
65   * receiving this BatchRequests message.
66   *
67   * @param msg Batch of Transactions of type BatchRequests from the primary.
68   * @return RC
69   */
70 RC WorkerThread::process_batch(Message *msg)
71 {
72     uint64_t cntime = get_sys_clock();
73
74     BatchRequests *breq = (BatchRequests *)msg;
75
76     //printf("BatchRequests: TID:%ld : VIEW: %ld : THD: %ld\n", breq->txn_id, breq->view, get_
77     //fflush(stdout);
78
79     // Assert that only a non-primary replica has received this message.
80     assert(g_node_id != get_current_view(get_thd_id()));
81
82     // Check if the message is valid.
83     validate_msg(breq);
84 }
```

PBFT Failure-Free Flow



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 ×
system > C++ worker_thread_pbft.cpp > ...
200
186 /**
187 * Processes incoming Prepare message.
188 *
189 * This function processes incoming messages of type PBFTPrepMessage. If
190 * received 2f identical Prepare messages from distinct replicas, then it c
191 * and sends a PBFTCommitMessage to all the other replicas.
192 *
193 * @param msg Prepare message of type PBFTPrepMessage from a replica.
194 * @return RC
195 */
196 RC WorkerThread::process_pbft_prep_msg(Message *msg)
197 {
198     //cout << "PBFTPrepMessage: TID: " << msg->txnid << " FROM: " << msg->
199     //fflush(stdout);
200
201     // Start the counter for prepare phase.
202     if (txn_man->prep_rsp_cnt == 2 * g_min_invalid_nodes)
203     {
204         txn_man->txn_stats.time_start_prepare = get_sys_clock();
205     }
206
207     // Check if the incoming message is valid.
208     PBFTPrepMessage *pmsg = (PBFTPrepMessage *)msg;
209     validate_msg(pmsg);
210
211     // Check if sufficient number of Prepare messages have arrived.
212     if (prepared(pmsg))
213     {
214         // Send Commit messages.
215         txn_man->send_pbft_commit_msgs();
216
217         // End the prepare counter.
218         INC_STATS(get_thd_id(), time_prepare, get_sys_clock() - txn_man->tx
219     }
220
221     return RCOK;
222 }
```

```
C++ worker_thread_pbft.cpp ×
system > C++ worker_thread_pbft.cpp > ⚡ WorkerThread::process_pbft_commit_msg(Message *)
200
275 /**
276 * Processes incoming Commit message.
277 *
278 * This function processes incoming messages of type PBFTCommitMessage
279 * received 2f+1 identical Commit messages from distinct replicas, then :
280 * execute-thread to execute all the transactions in this batch.
281 *
282 * @param msg Commit message of type PBFTCommitMessage from a replica.
283 * @return RC
284 */
285 RC WorkerThread::process_pbft_commit_msg(Message *msg)
286 {
287     //cout << "PBFTCommitMessage: TID: " << msg->txnid << " FROM: " << msg->
288     //fflush(stdout);
289
290     if (txn_man->commit_rsp_cnt == 2 * g_min_invalid_nodes + 1)
291     {
292         txn_man->txn_stats.time_start_commit = get_sys_clock();
293     }
294
295     // Check if message is valid.
296     PBFTCommitMessage *pcmsg = (PBFTCommitMessage *)msg;
297     validate_msg(pcmsg);
298
299     txn_man->add_commit_msg(pcmsg);
300
301     // Check if sufficient number of Commit messages have arrived.
302     if (committed_local(pcmsg))
303     {
304 #if TIMER_ON
305         // End the timer for this client batch.
306         server_timer->endTimer(txn_man->hash);
307    #endif
308
309     // Add this message to execute thread's queue.
310     send_execute_msg();
311
312     INC_STATS(get_thd_id(), time_commit, get_sys_clock() - txn_man->tx
313 }
```

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

```
C++ worker_thread.cpp X
system > C++ worker_thread.cpp > WorkerThread::process_execute_msg(Message *)
795
796 /**
797 * Execute transactions and send client response.
798 *
799 * This function is only accessed by the execute-thread, which executes the transactions
800 * in a batch, in order. Note that the execute-thread has several queues, and at any
801 * point of time, the execute-thread is aware of which is the next transaction to
802 * execute. Hence, it only loops on one specific queue.
803 *
804 * @param msg Execute message that notifies execution of a batch.
805 * @ret RC
806 */
807 RC WorkerThread::process_execute_msg(Message *msg)
808 {
809     //cout << "EXECUTE " << msg->txnid << " :: " << get_thd_id() << "\n";
810     //fflush(stdout);
811
812     uint64_t ctime = get_sys_clock();
813
814     // This message uses txnid of index calling process_execute.
815     Message *rsp = Message::create_message(CL_RSP);
816     ClientResponseMessage *crsp = (ClientResponseMessage *)rsp;
817     crsp->init();
818
819     ExecuteMessage *emsg = (ExecuteMessage *)msg;
820
821     // Execute transactions in a shot
822     uint64_t i;
823     for (i = emsg->index; i < emsg->end_index - 4; i++)
824     {
825         //cout << "i: " << i << " :: next index: " << g_next_index << "\n";
826         //fflush(stdout);
827
828         TxnManager *tman = get_transaction_manager(i, 0);
829
830         inc_next_index();
831
832         // Execute the transaction
833         tman->run_txnid();
834
835         // Commit the results.
836         tman->commit();
837
838         crsp->copy_from_txnid(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

```
C++ work_queue.cpp ×  
system > C++ work_queue.cpp > ...  
44     void QWorkQueue::enqueue(uint64_t thd_id, Message *msg, bool busy)  
45     {  
46         uint64_t starttime = get_sys_clock();  
47         assert(msg);  
48         DEBUG_M("QWorkQueue::enqueue work_queue_entry alloc\n");  
49         work_queue_entry *entry = (work_queue_entry *)mem_allocator.align_alloc(sizeof(work_queue_ent  
50             entry->msg = msg;  
51             entry->rtype = msg->rtype;  
52             entry->txn_id = msg->txn_id;  
53             entry->batch_id = msg->batch_id;  
54             entry->starttime = get_sys_clock();  
55             assert(ISSERVER || ISREPLICA);  
56             DEBUG("Work Enqueue (%ld,%ld) %d\n", entry->txn_id, entry->batch_id, entry->rtype);  
57  
58             if (msg->rtype == CL_QRY || msg->rtype == CL_BATCH)  
59             {  
60                 if (g_node_id == get_current_view(thd_id))  
61                 {  
62                     //cout << "Placing \n";  
63                     while (!new_txn_queue->push(entry) && !simulation->is_done())  
64                     {  
65                         //cout << "Pushing \n";  
66                     }  
67                 else  
68                 {  
69                     assert(entry->rtype < 100);  
70                     while (!work_queue[0]->push(entry) && !simulation->is_done())  
71                     {  
72                         //cout << "Pushing \n";  
73                     }  
74                 }  
75             }  
76         }  
77     }  
78 }
```

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

```
C++ io_thread.cpp ×  
system > C++ io_thread.cpp > ...  
299     RC InputThread::server_recv_loop()  
300     {  
301  
302         myrand rdm;  
303         rdm.init(get_thd_id());  
304         RC rc = RCOK;  
305         assert(rc == RCOK);  
306         uint64_t starttime = 0;  
307         uint64_t idle_starttime = 0;  
308         std::vector<Message *> *msgs;  
309         while (!simulation->is_done())  
310         {  
311             heartbeat();  
312  
313             #if VIEW_CHANGES  
314                 if (g_node_id != get_current_view(get_thd_id()))  
315                 {  
316                     uint64_t tid = get_thd_id() - 1;  
317                     uint32_t nchange = get_newView(tid);  
318  
319                     if (nchange)  
320                     {  
321                         set_current_view(get_thd_id(), get_current_view(get_thd_id()) + 1);  
322                         set_newView(tid, false);  
323                     }  
324                 }  
325             #endif  
326  
327             msgs = tport_man.recv_msg(get_thd_id());  
328         }
```

Thank You