VLDB2020Tokyo

Building High Throughput Permissioned Blockchain Fabrics: Challenges and Opportunities


Suyash Gupta


Jelle Hellings


Sajjad Rahnama


Mohammad Sadoghi

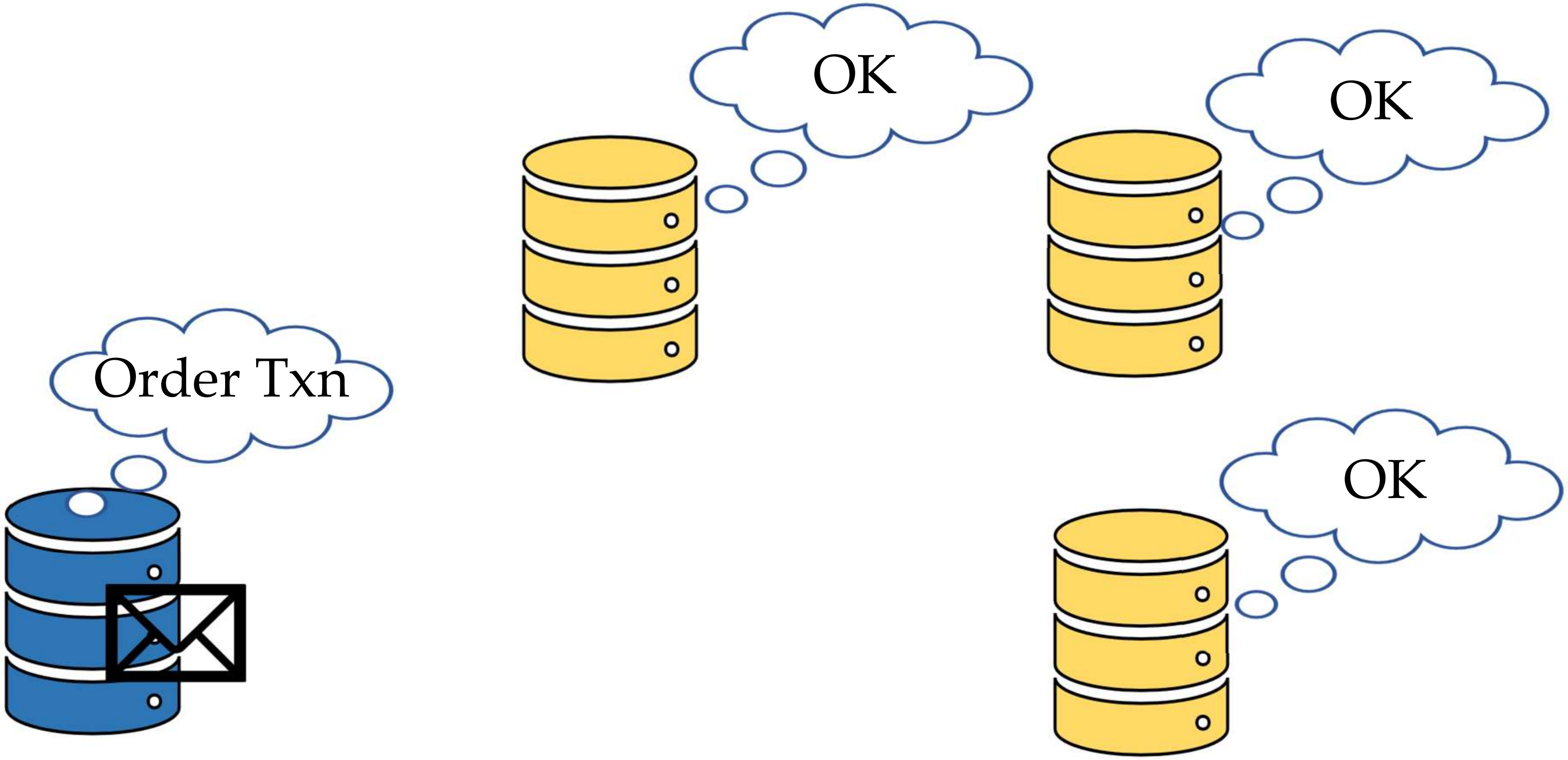
Exploratory Systems Lab
University of California Davis

UCDAVIS UNIVERSITY OF CALIFORNIA

ExpoLab Creativity Unfolded

ResilientDB

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



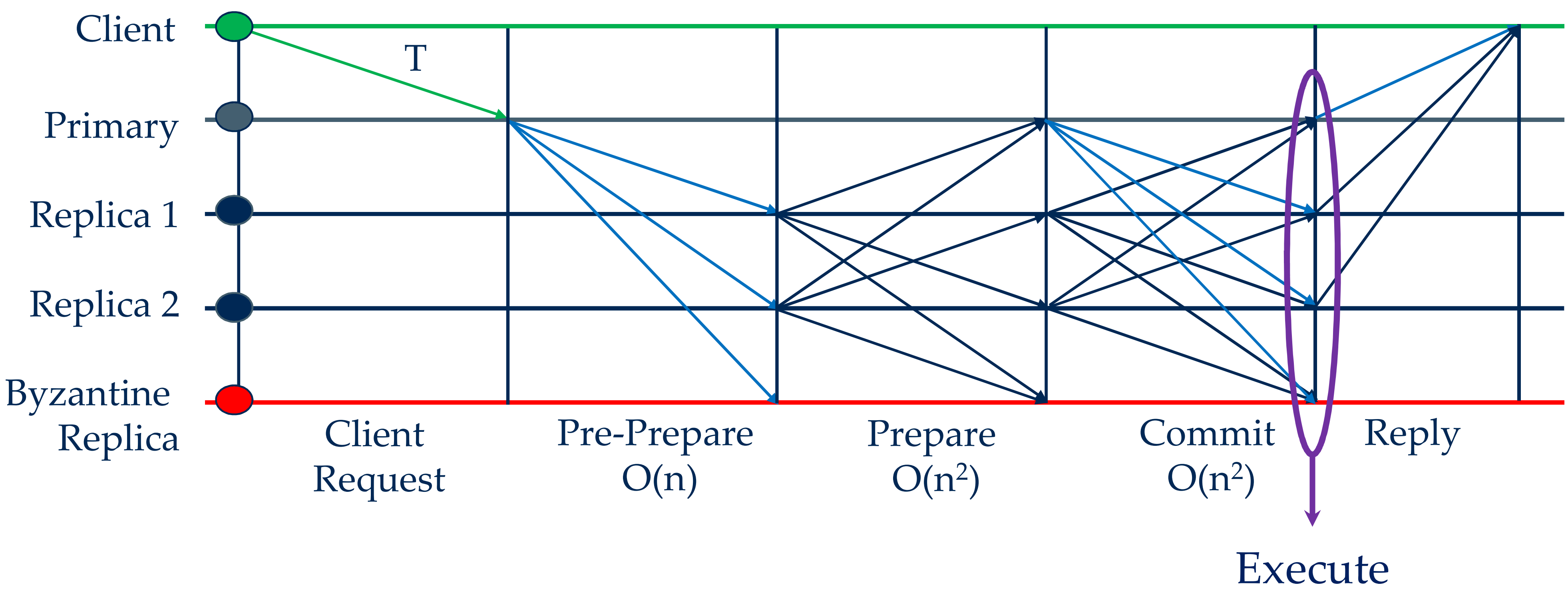
Practical Byzantine Fault-Tolerance (PBFT)

[OSDI'99]

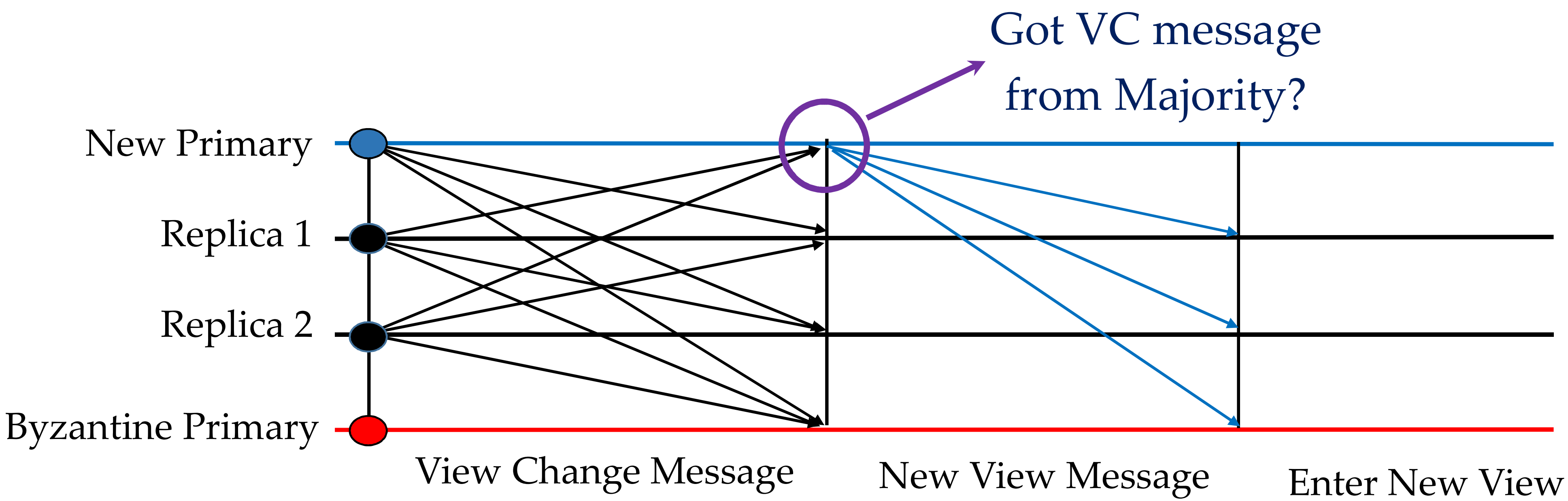
- First *practical* Byzantine Fault-Tolerant Protocol.
- Tolerates up to **f** failures in a system of 3f+1 replicas
- Requires three phases of which two necessitate quadratic communication complexity.
- Safety is always guaranteed and Liveness is guaranteed in periods of partial synchrony.

ResilientDB ExpoLab Creativity Unfolded

PBFT Civil Executions



PBFT Uncivil Execution: Primary Failure
(*View Change*)



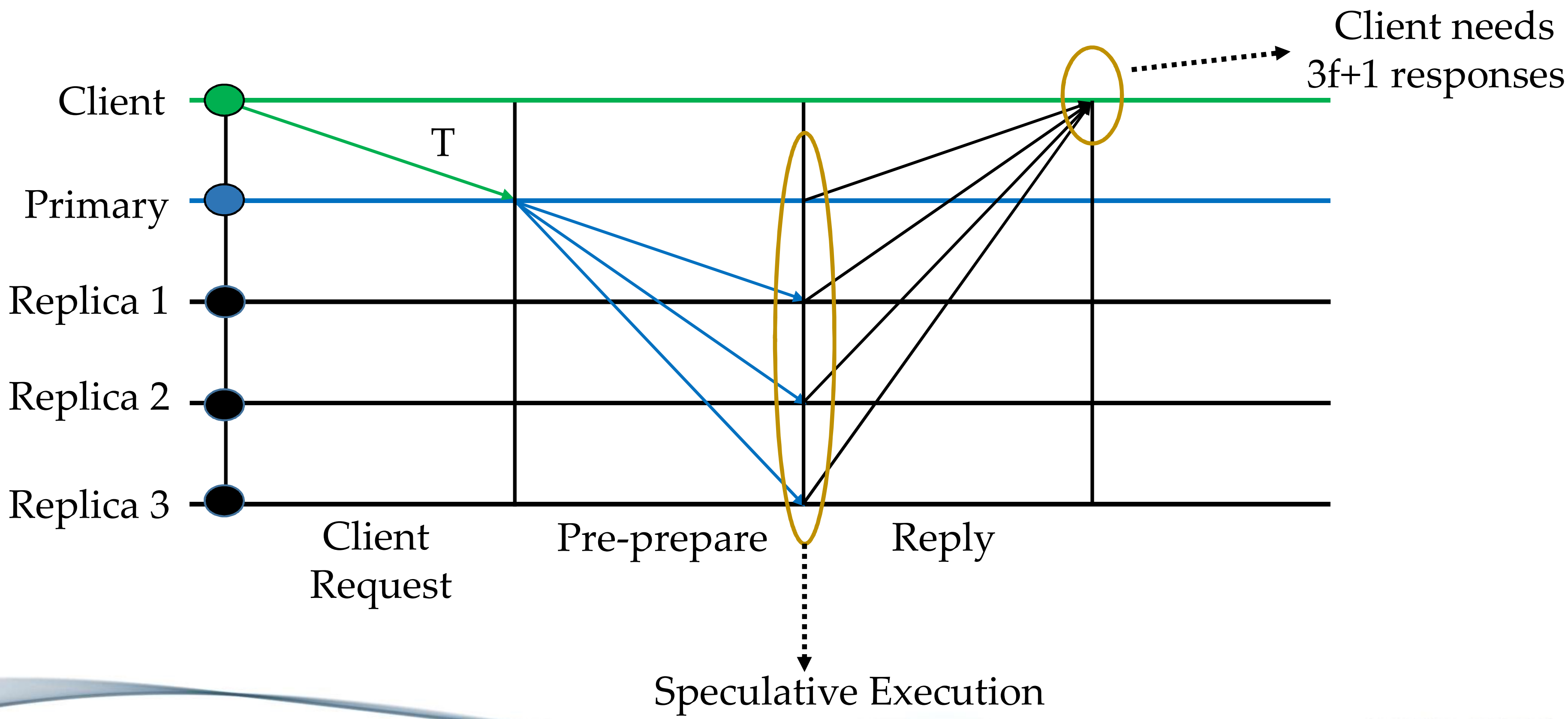
ResilientDB ExpoLab Creativity Unfolded

Speculative Byzantine Fault Tolerance (Zyzzyva) [SOSP'07]

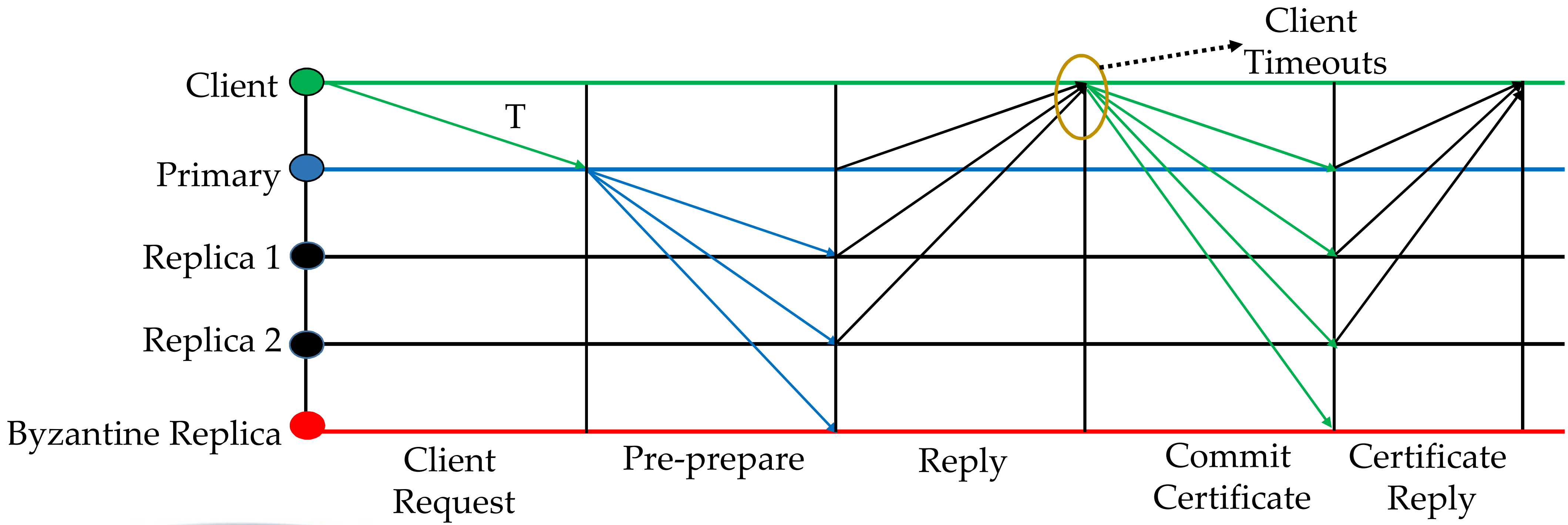
- Speculation to achieve consensus in a single phase.
- Under ***no failures***, it only requires linear communication complexity.
- Requires good clients, for ensuring same order across the replicas.
- Clients need matching responses from all the 3f+1 replicas.
- Just one crash failure is sufficient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

ResilientDB ExpoLab Creativity Unfolded

Zyzzyva Civil Executions



Zyzzyva under Failure of *one* Non-Primary Replica



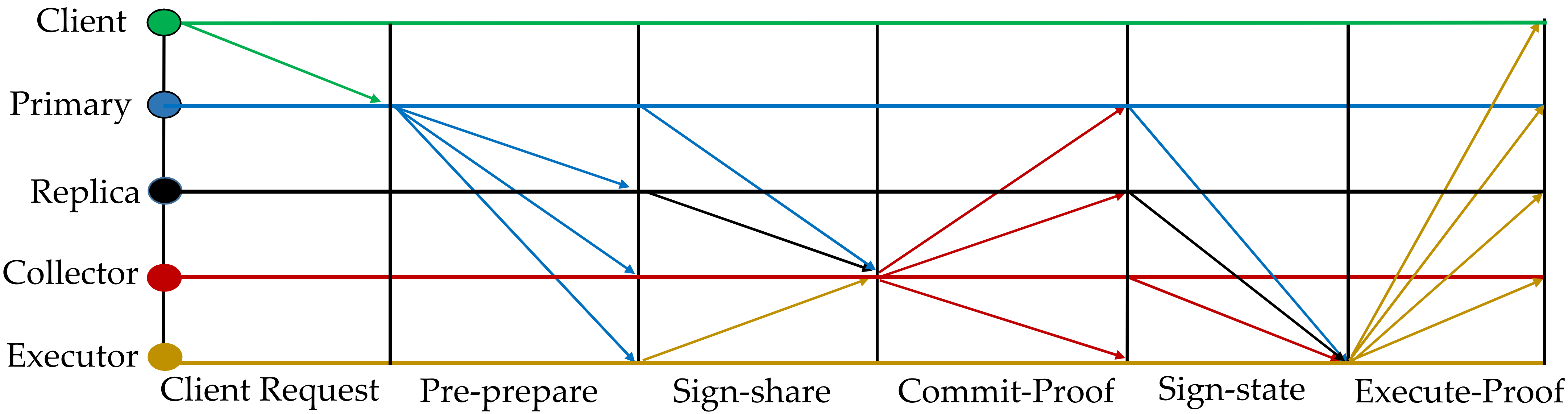
On client timeout → switches to *slow-path*.

ResilientDB ExpoLab Creativity Unfolded

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

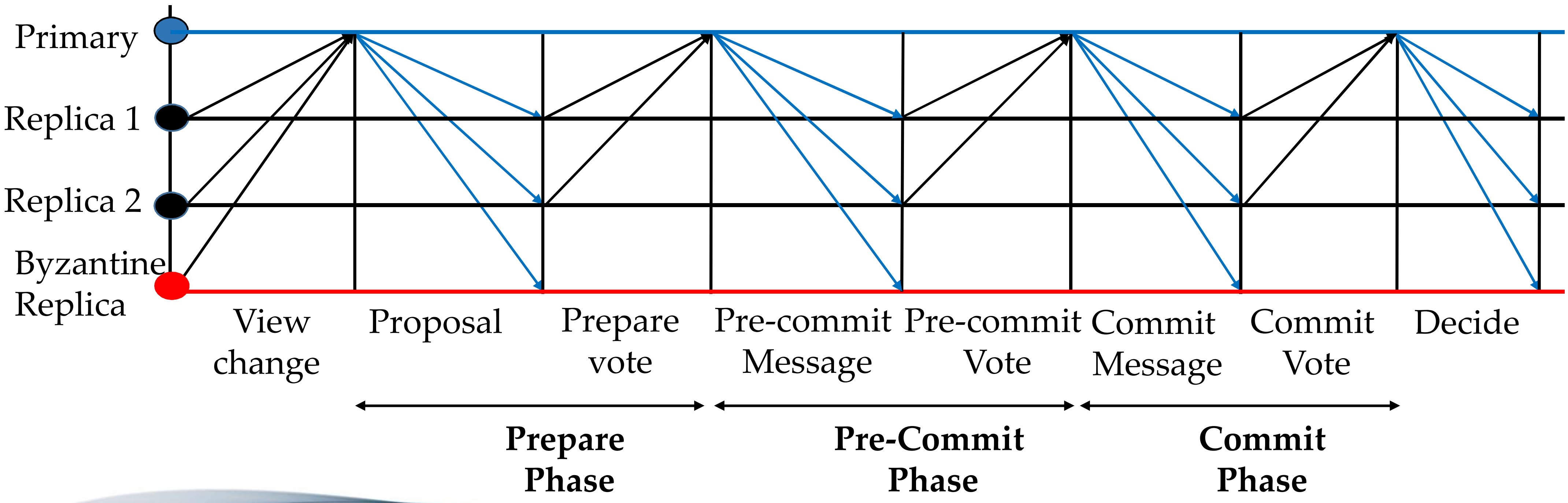


Either no failures or c+1 crash failures for c > 0 collectors if n = 3f+2c+1

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 >> 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

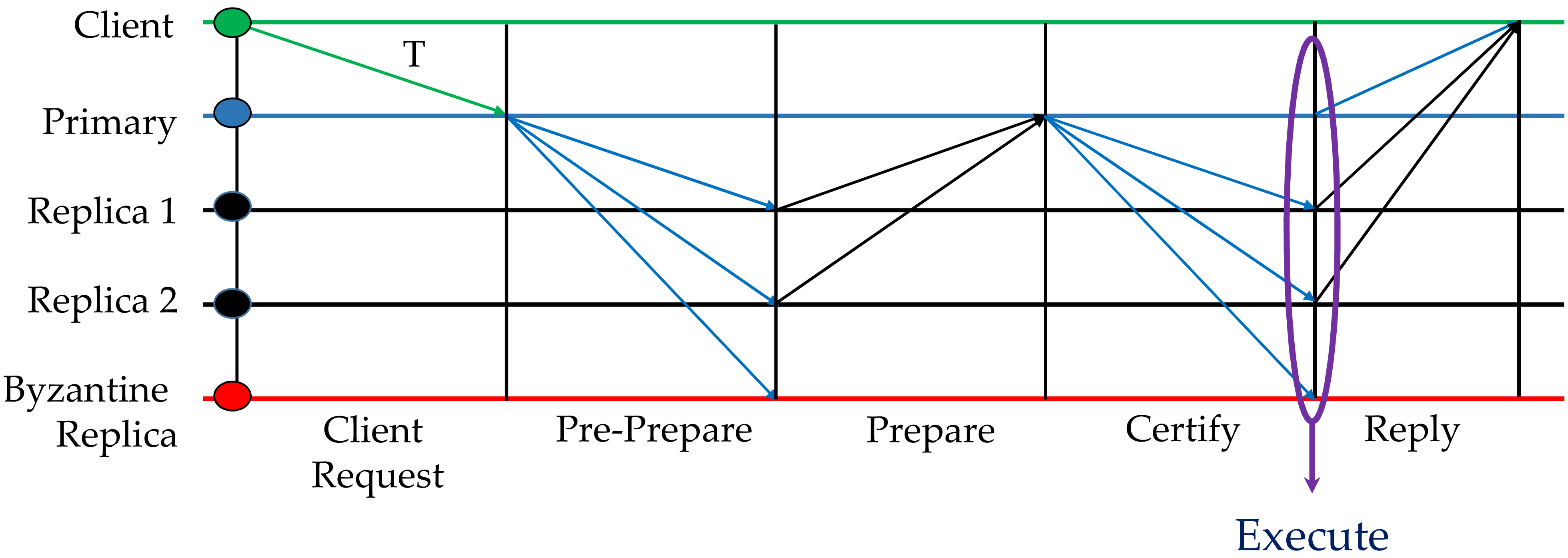
No dependence on clients or trusted component.

No reliance on a twin-path design.

PoE vs Other Protocols

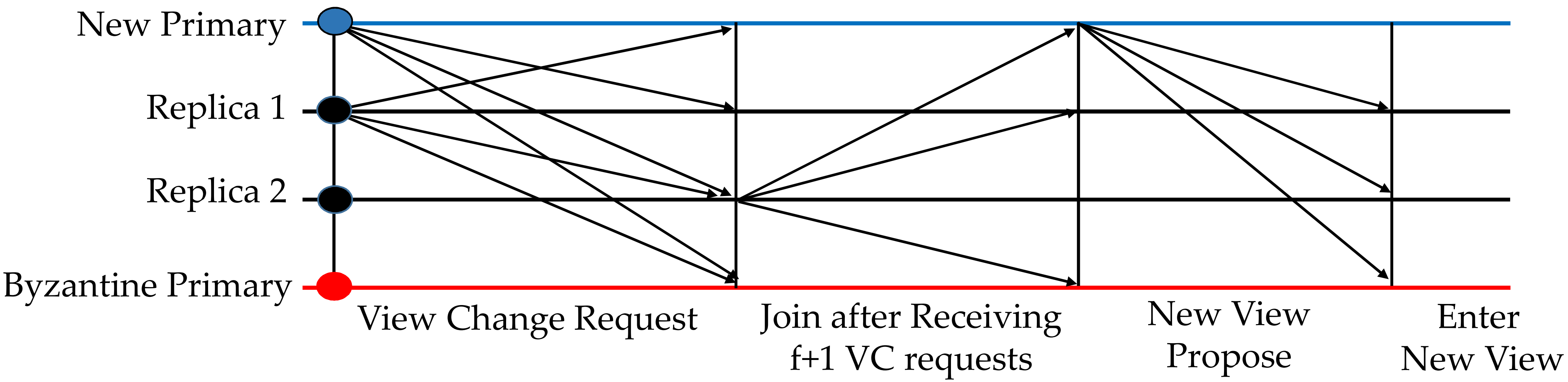
Protocol	Phases	Messages	Resilience	Requirements
ZYZZYVA	1	$\mathcal{O}(\mathbf{n})$	0	reliable clients and unsafe
POE (our paper)	3	$\mathcal{O}(3\mathbf{n})$	$\mathbf{f}$	sign. agnostic
PBFT	3	$\mathcal{O}(\mathbf{n}+2\mathbf{n}^2)$	$\mathbf{f}$	
HOTSTUFF	4	$\mathcal{O}(\mathbf{n}+3\mathbf{n}^2)$	$\mathbf{f}$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

Proof-of-Execution (PoE)



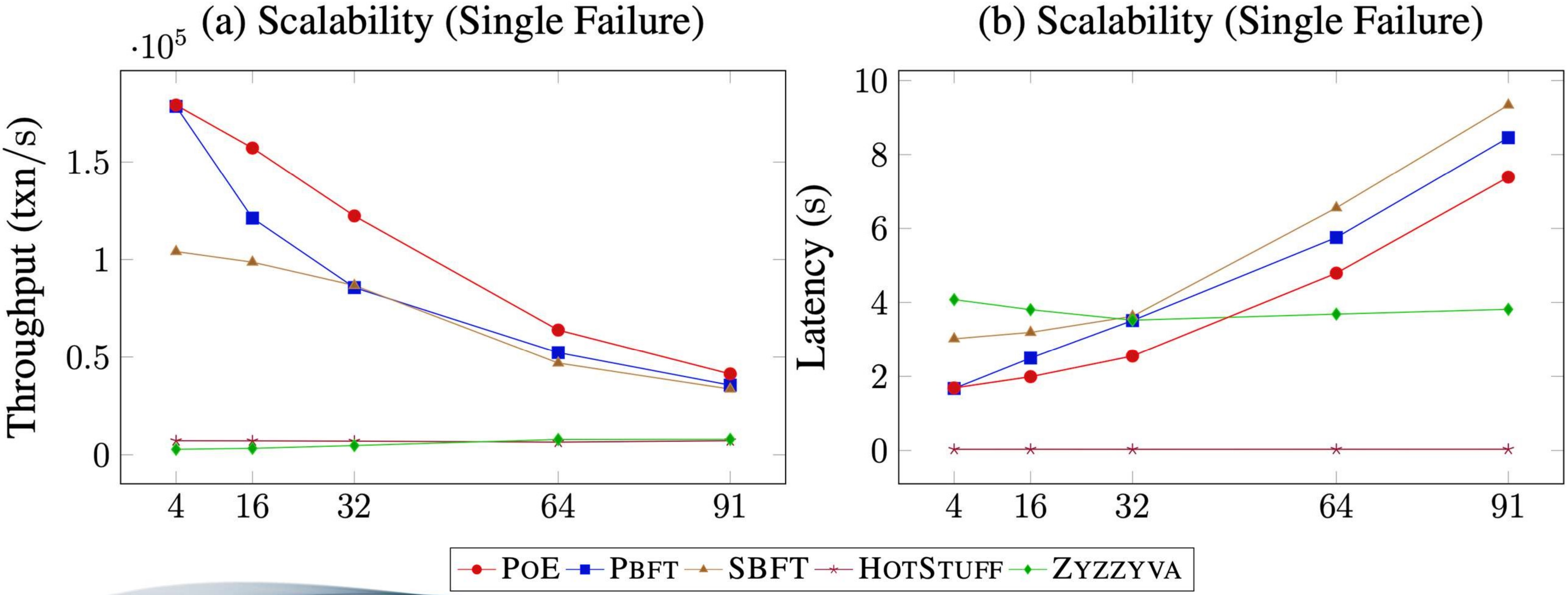
n = 4 replicas and f <= 1

PoE View Change Protocol



ResilientDB ExpoLab Creativity Unfolded

PoE Scalability under 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.

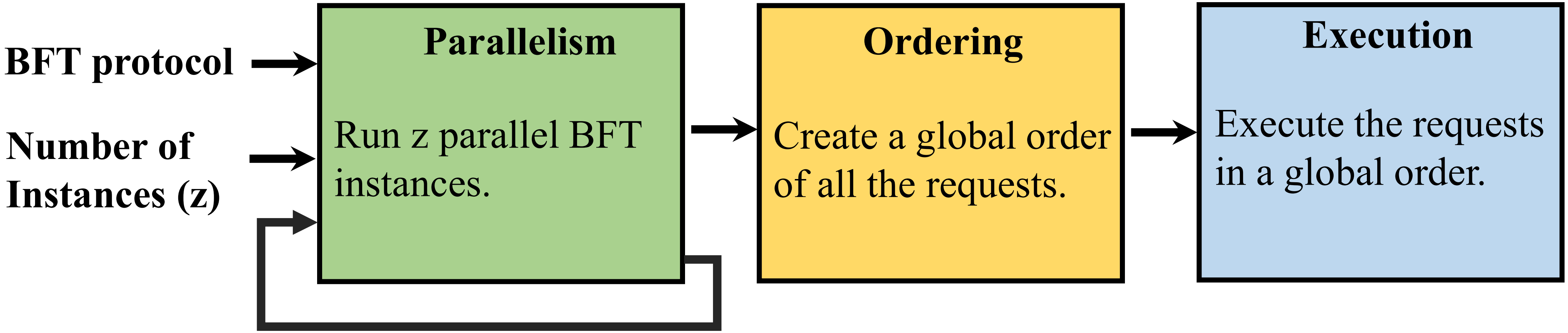
RCC Defense

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

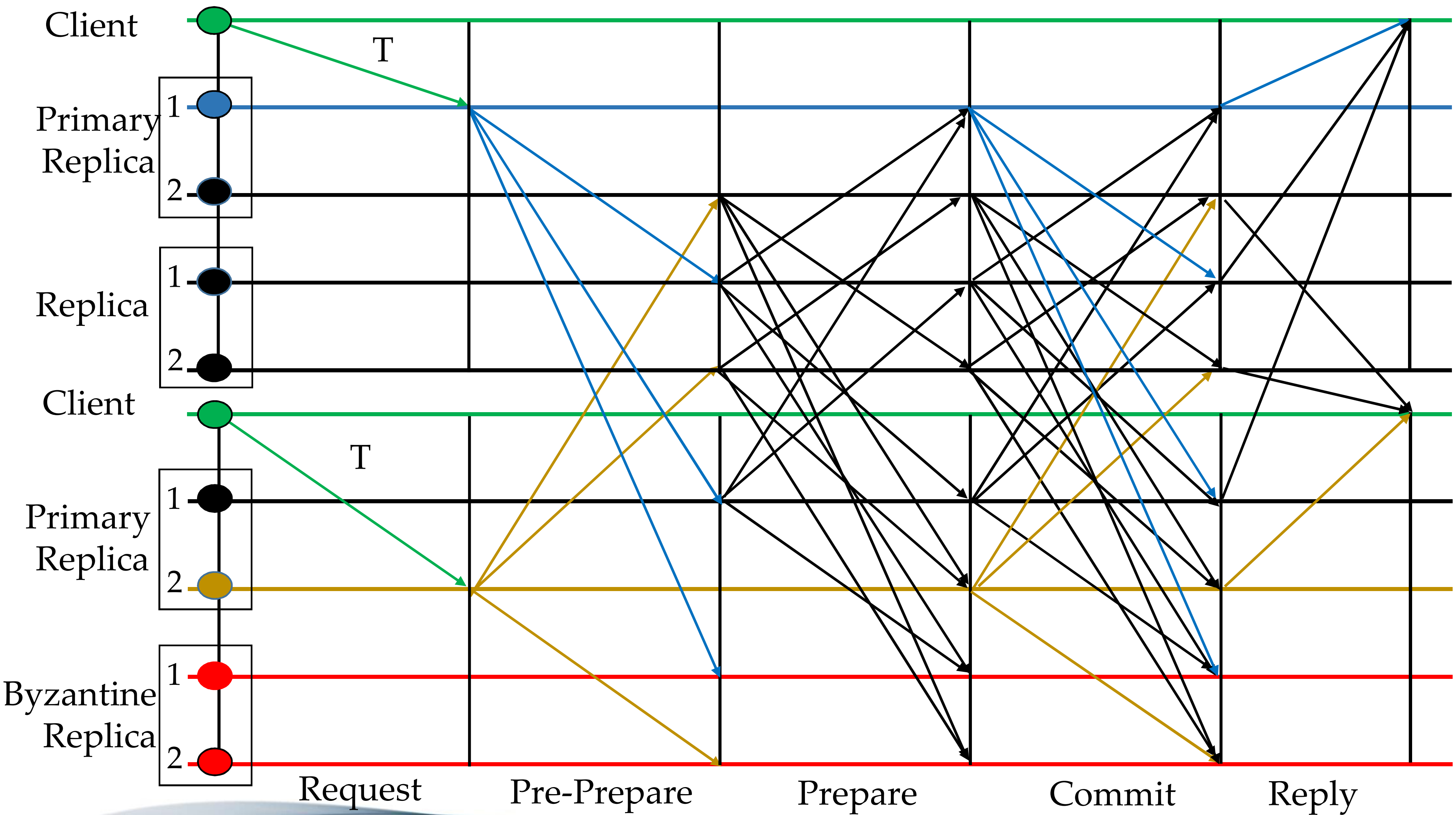
Malicious primary can ***throttle*** the system throughput.

Malicious primary requires ***replacemenat*** → fall in throughput.

Resilient Concurrency Control Paradigm



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

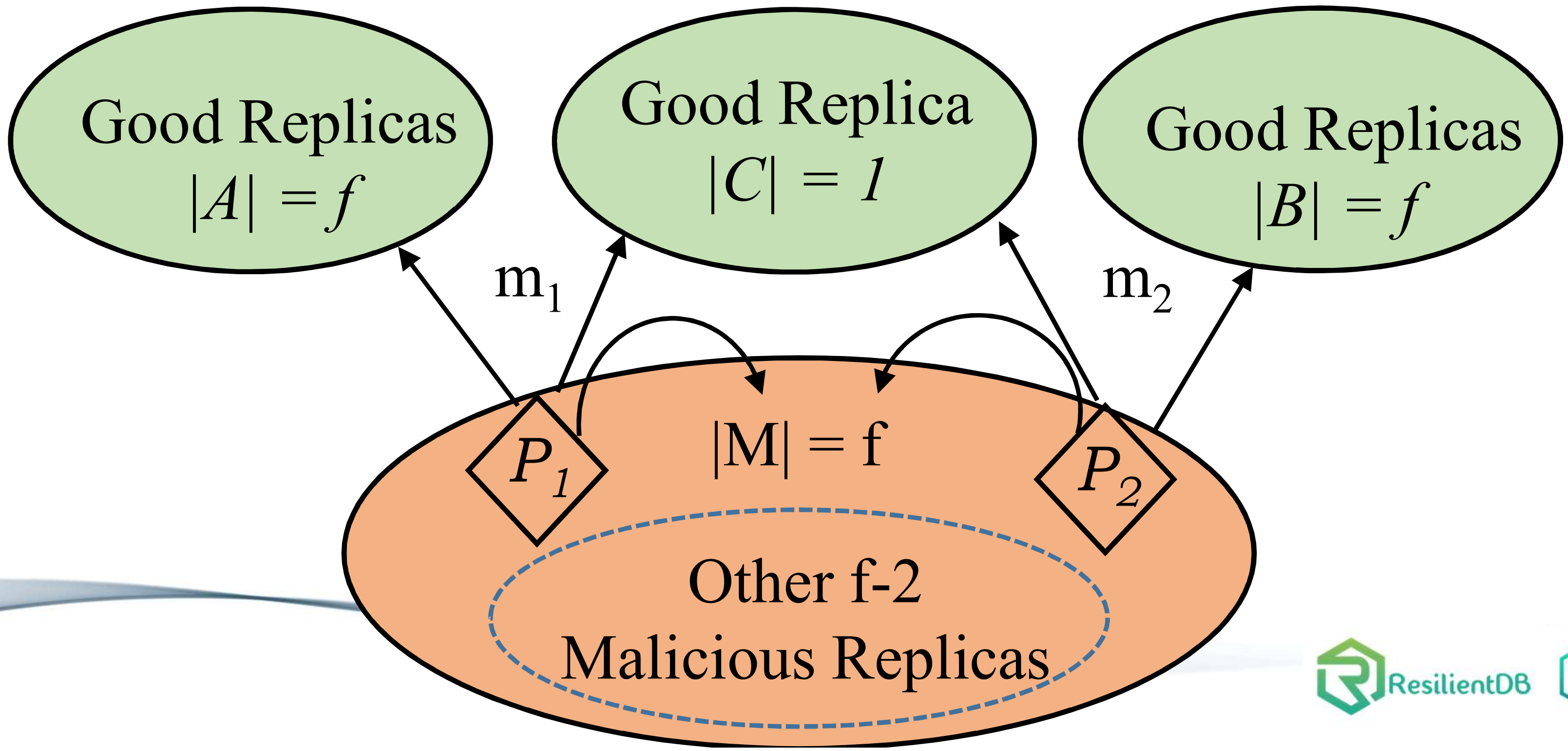


RCC using PBFT with 2 parallel instances on each replica

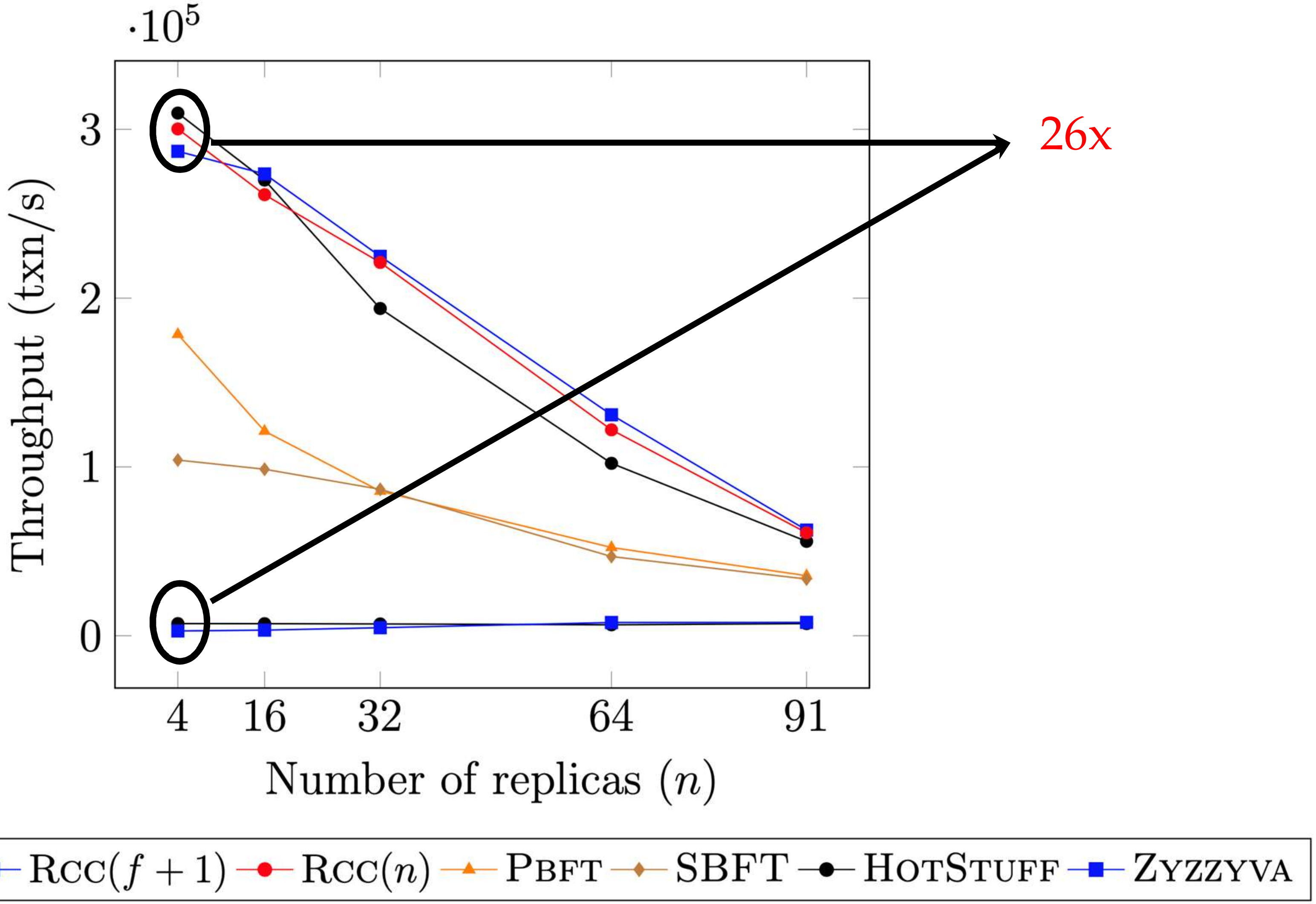
Colluding Primaries

Multiple malicious primaries can prevent liveness!

Solution → Optimistic Recovery through State Exchange.



Scalability



Single Failure Experiments

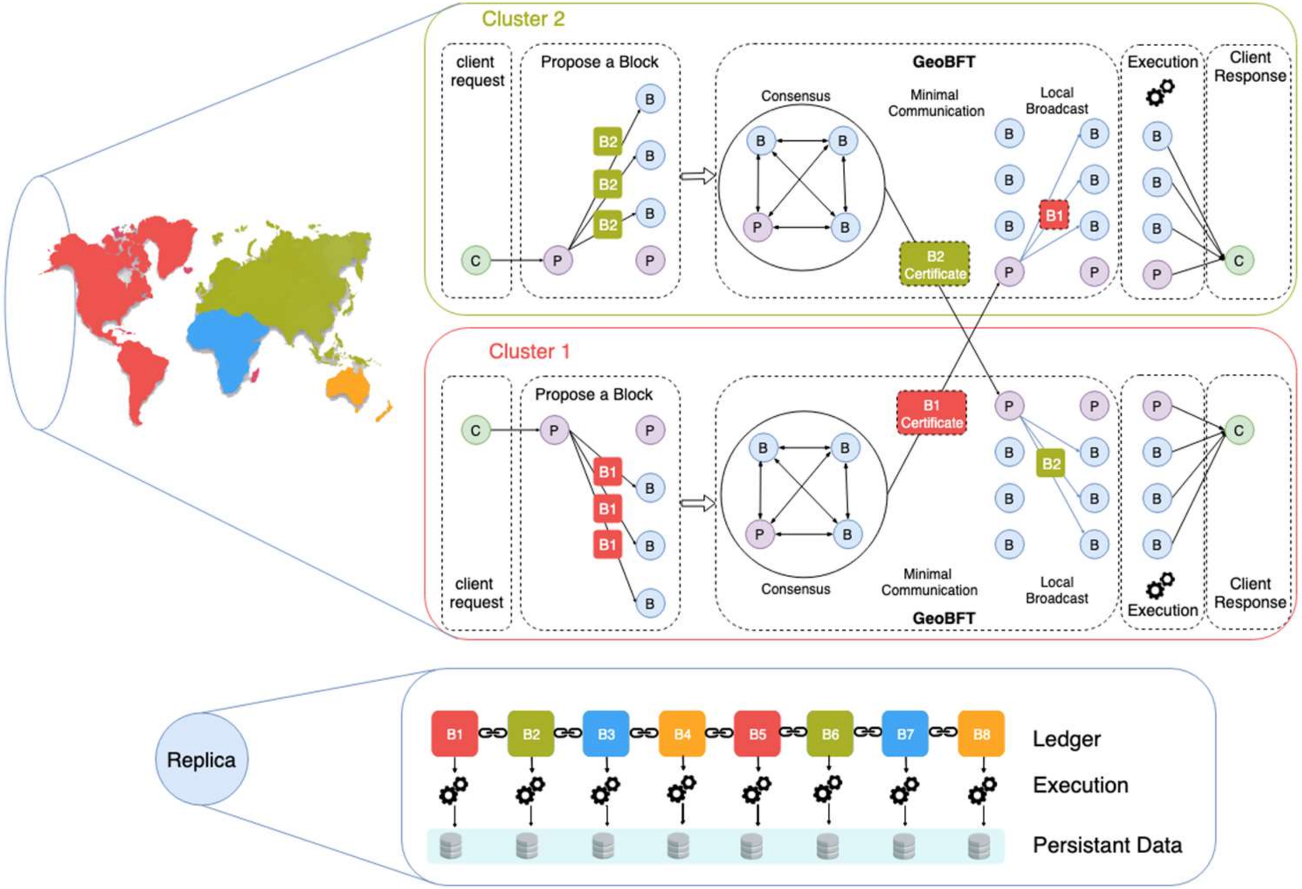
ResilientDB ExpoLab Creativity Unfolded

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.

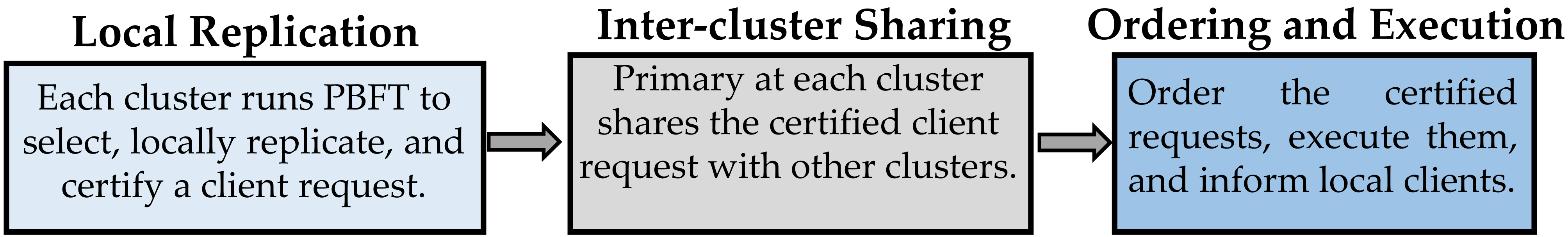
Proceedings of the VLDB Endowment 2020.

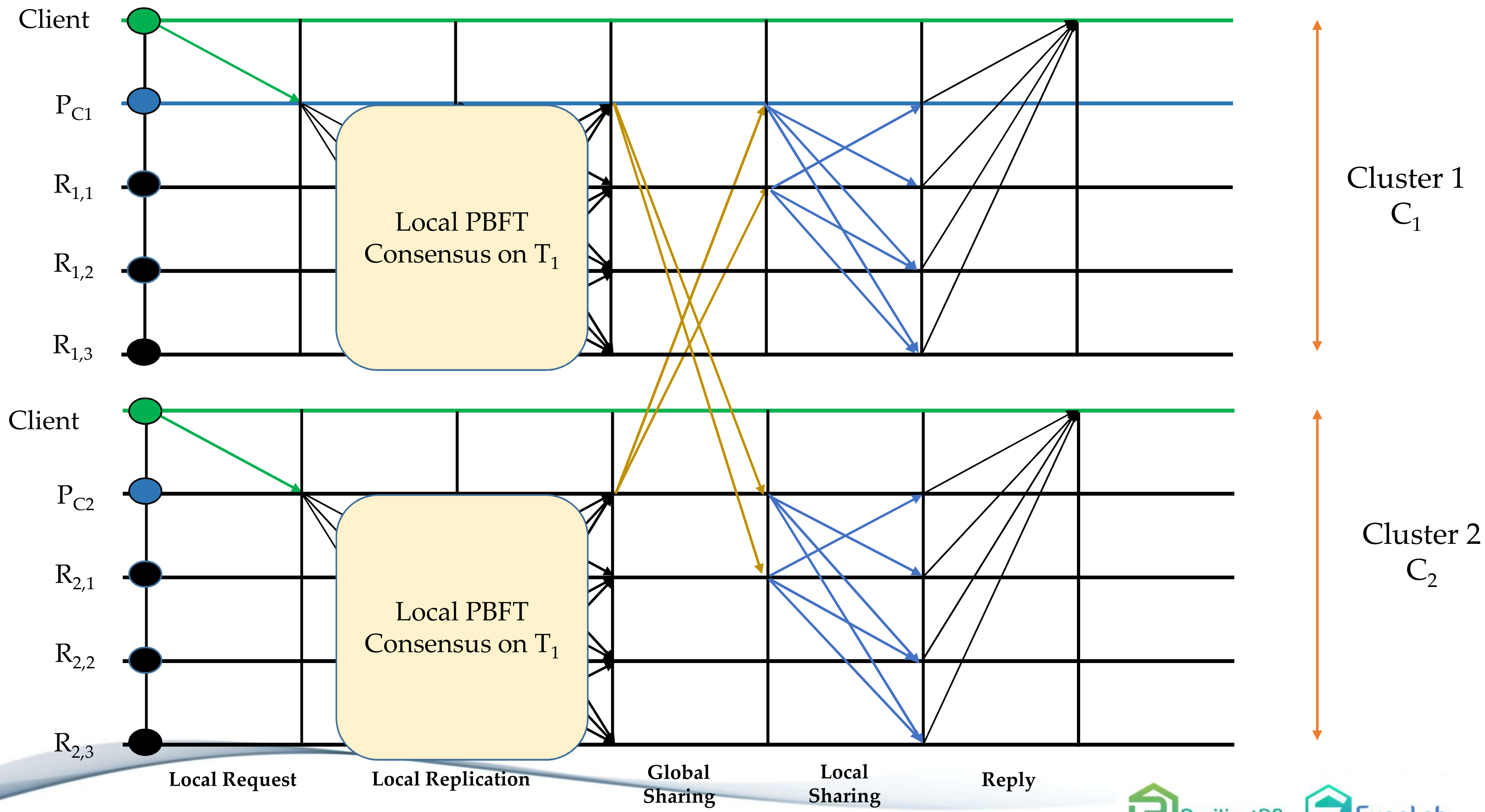
Vision Geo-Scale Byzantine Fault-Tolerance



GeoBFT Protocol

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



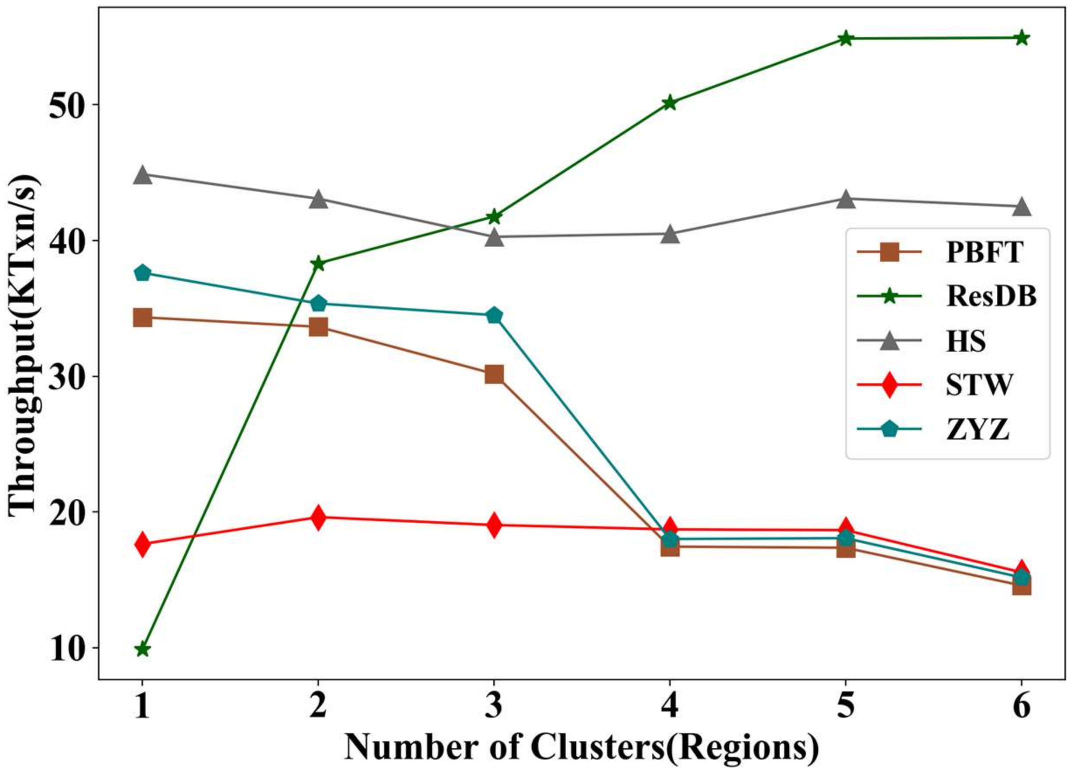


ResilientDB ExpoLab Creativity Unfolded

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



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

Visit at: https://resilientdb.com/

UC DAVIS
UNIVERSITY OF CALIFORNIA

*Proceedings of the 40th IEEE ICDCS 2020.

Why Should You Chose ResilientDB?

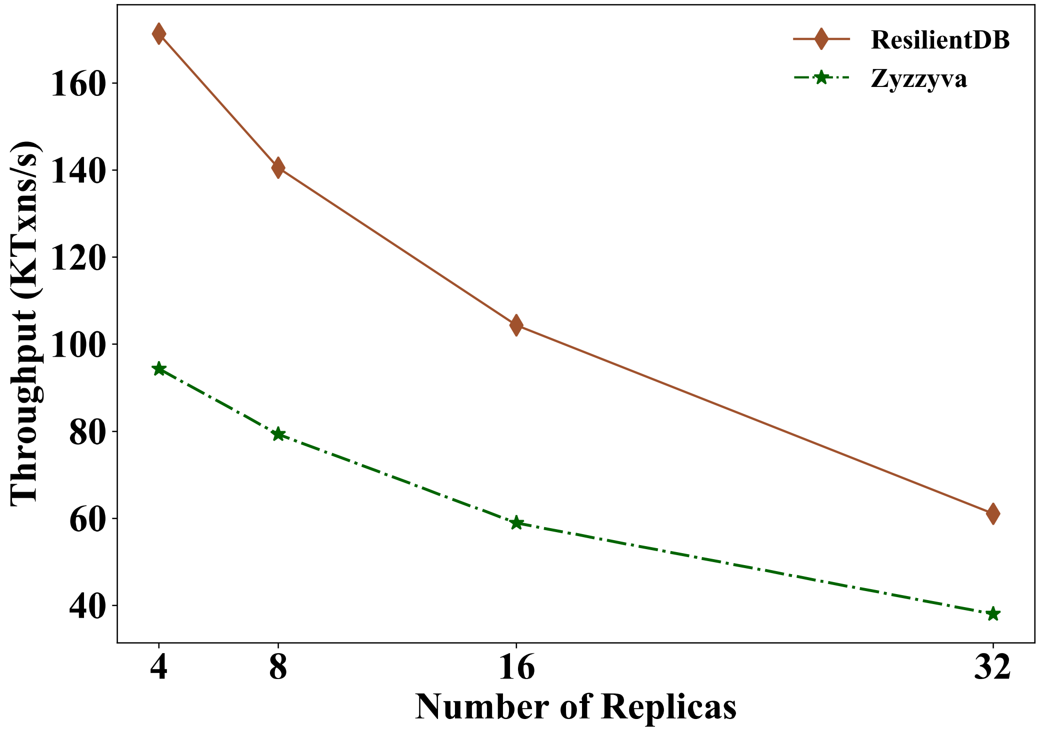
1) Bitcoin and Ethereum offer low throughputs of *10 txns/s*.
2) Existing Permissioned Blockchain Databases still have low throughputs (*20K txns/s*).
3) Prior works blame BFT consensus as *expensive*.
4) System Design is mostly *overlooked*.
5) ResilientDB adopts *well-researched* database and system practices.

Dissecting Existing Permissioned Blockchains

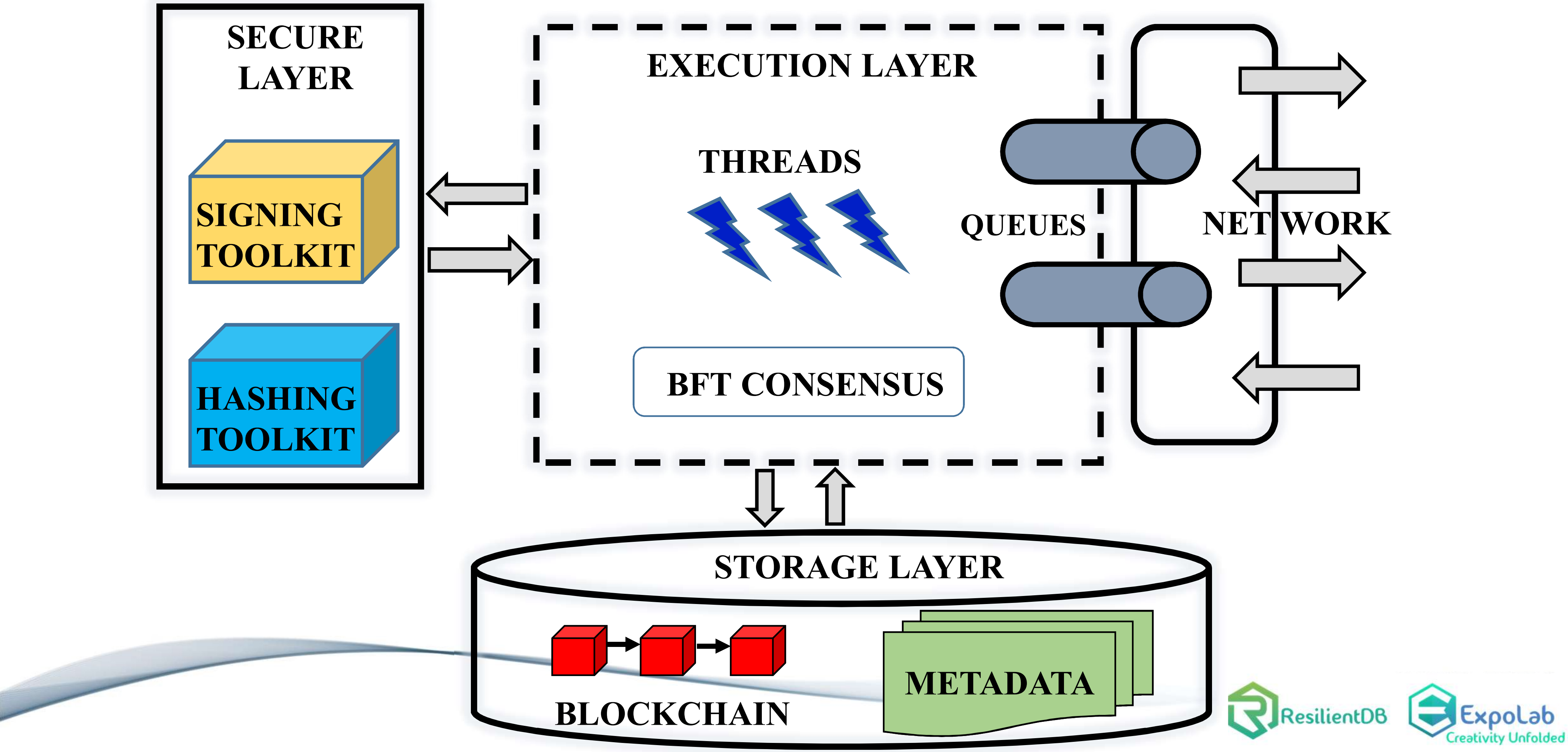
1) Single-threaded Monolithic Design
2) Successive Phases of Consensus
3) Integrated Ordering and Execution
4) Strict Ordering
5) Off-Chain Memory Management
6) Expensive Cryptographic Practices

ResilientDB ExpoLab Creativity Unfolded

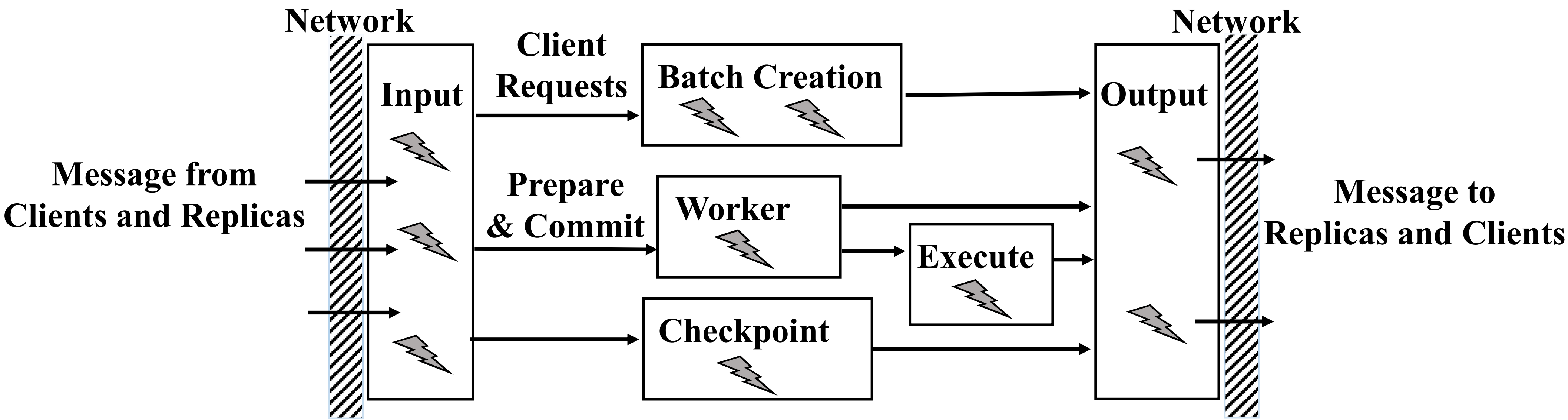
Can a well-crafted system based on a classical BFT protocol outperform a modern protocol?



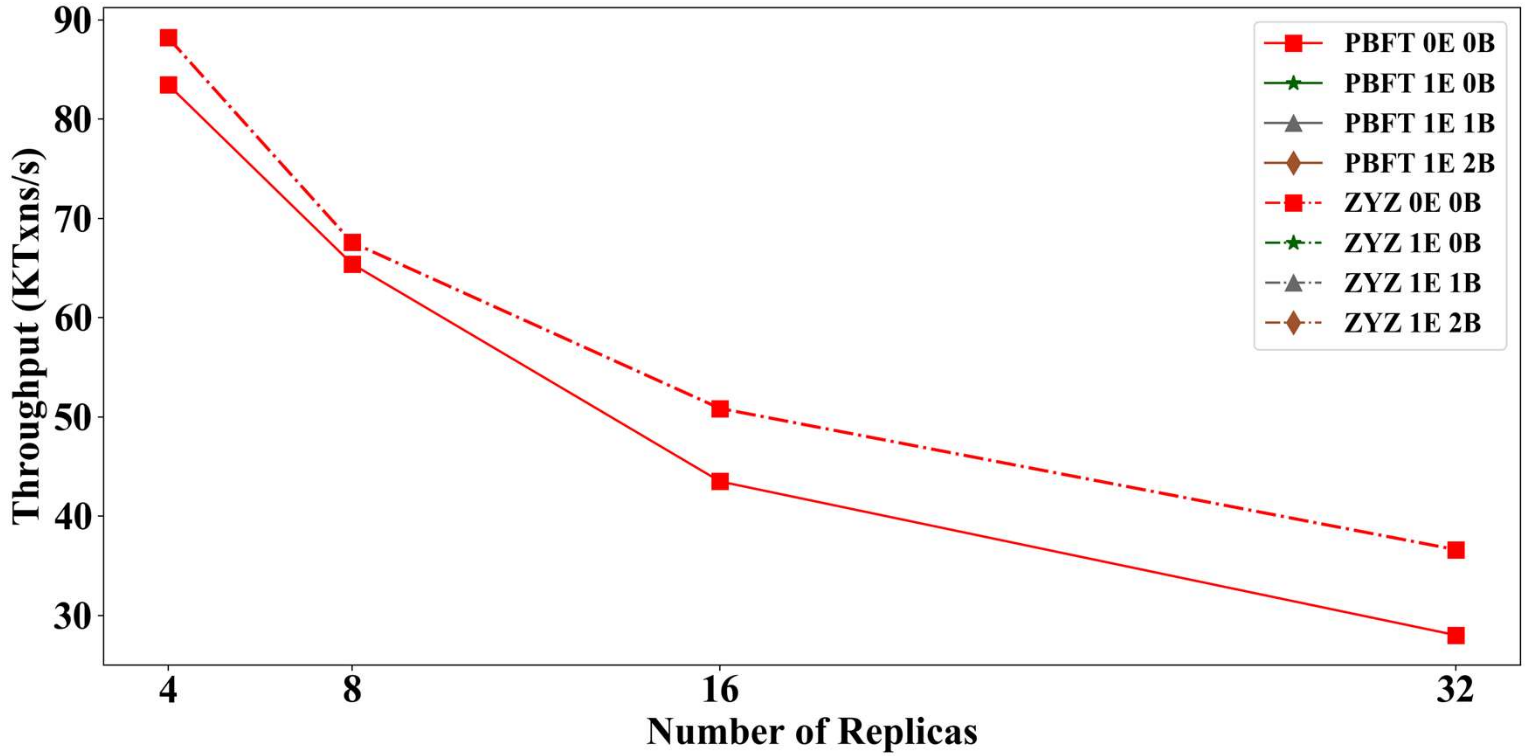
ResilientDB Architecture



ResilientDB Multi-Threaded Deep Pipeline



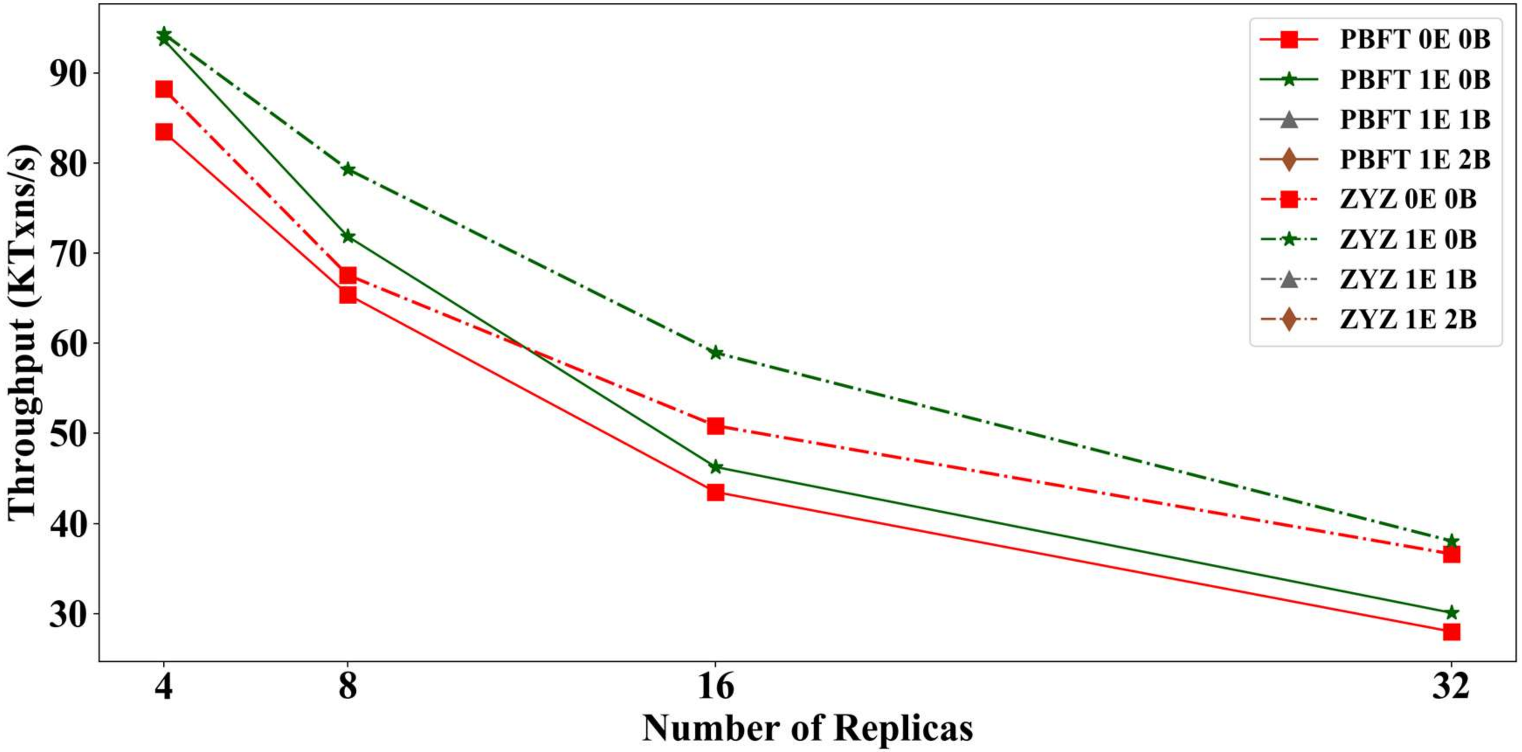
Insight 1: Multi-Threaded pipeline Gains



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

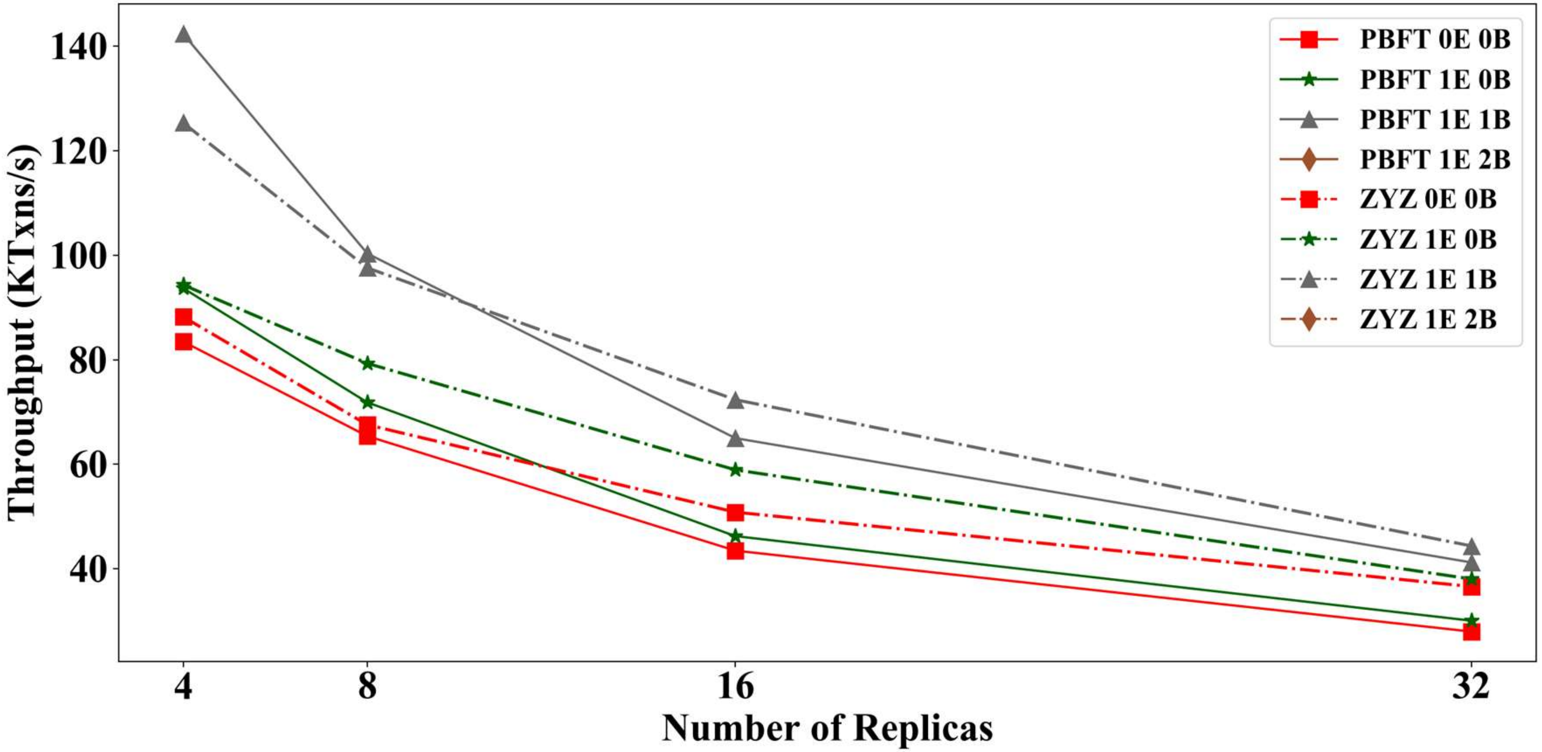
ResilientDB ExpoLab Creativity Unfolded

Insight 1: Multi-Threaded pipeline Gains



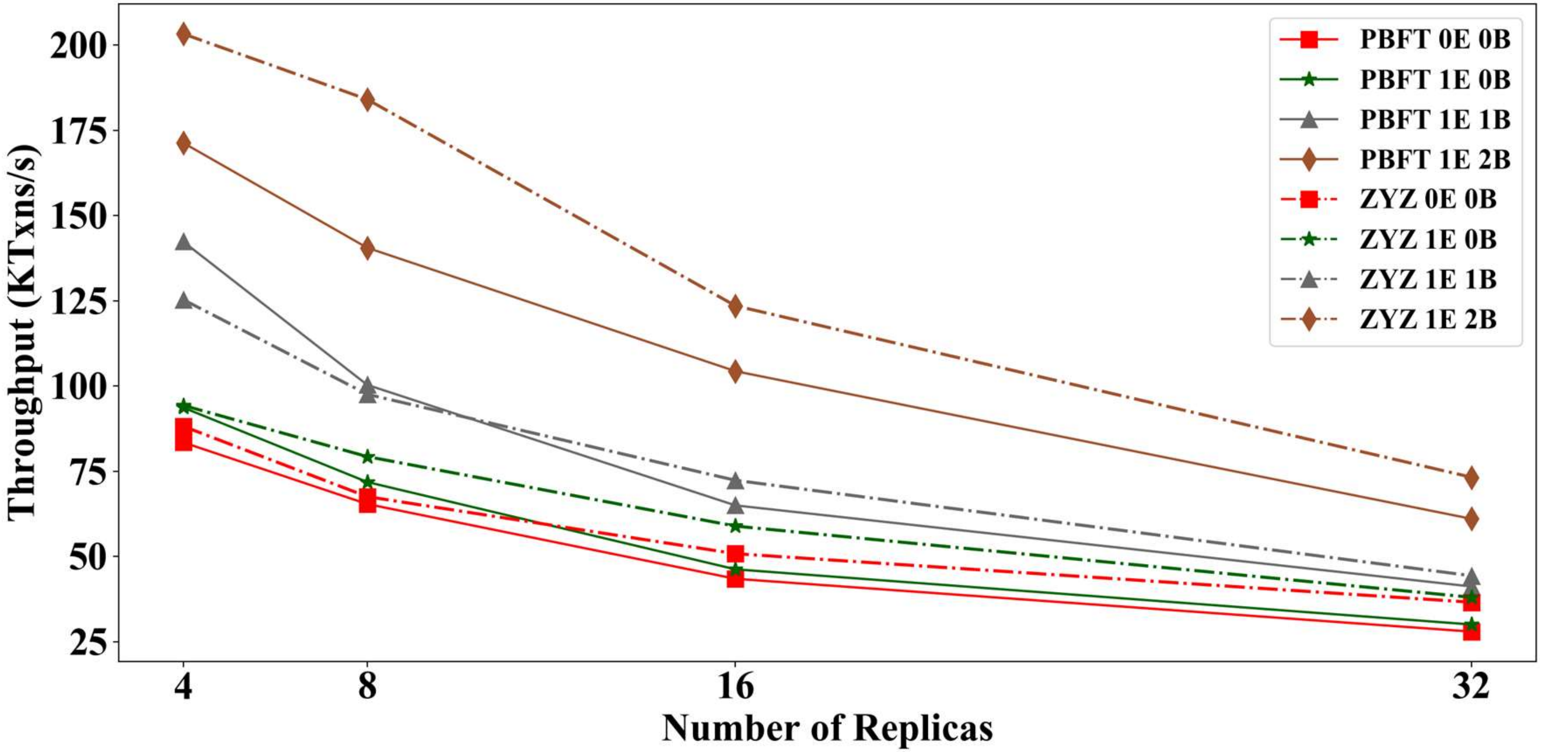
Parallelizing and Pipelining tasks across worker, execution (E) and batch-threads (B).

Insight 1: Multi-Threaded pipeline Gains



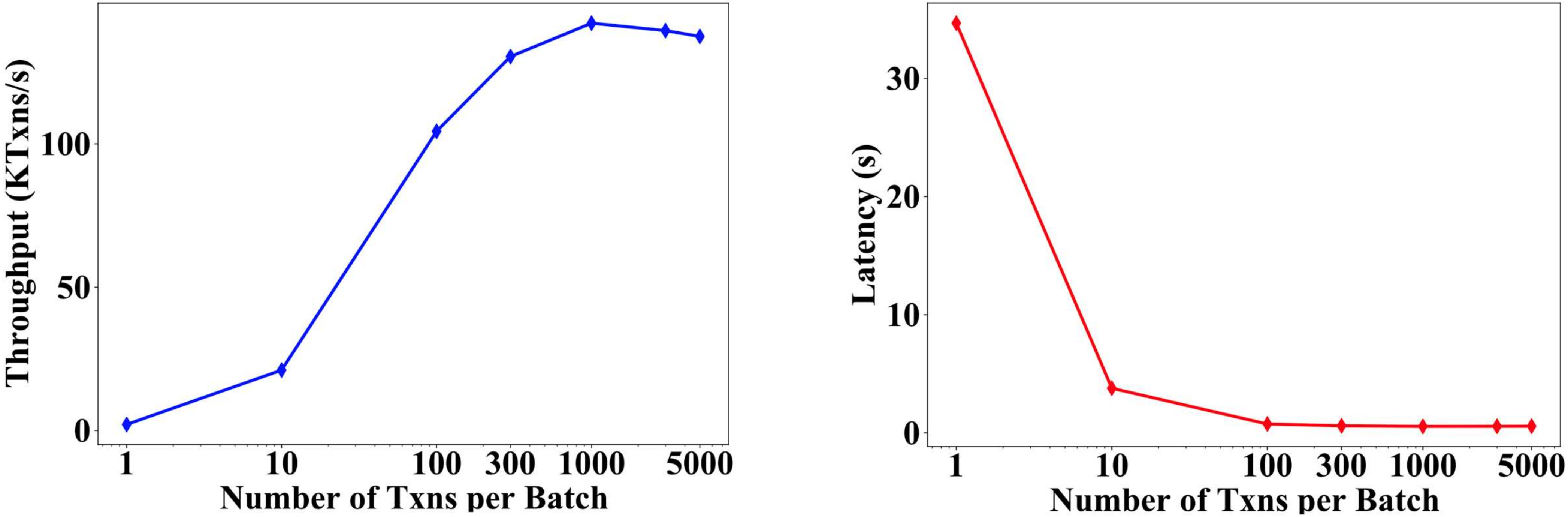
Parallelizing and Pipelining tasks across worker, execution (E) and batch-threads (B).

Insight 1: Multi-Threaded pipeline Gains



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

Insight 2: Optimal Batching Gains

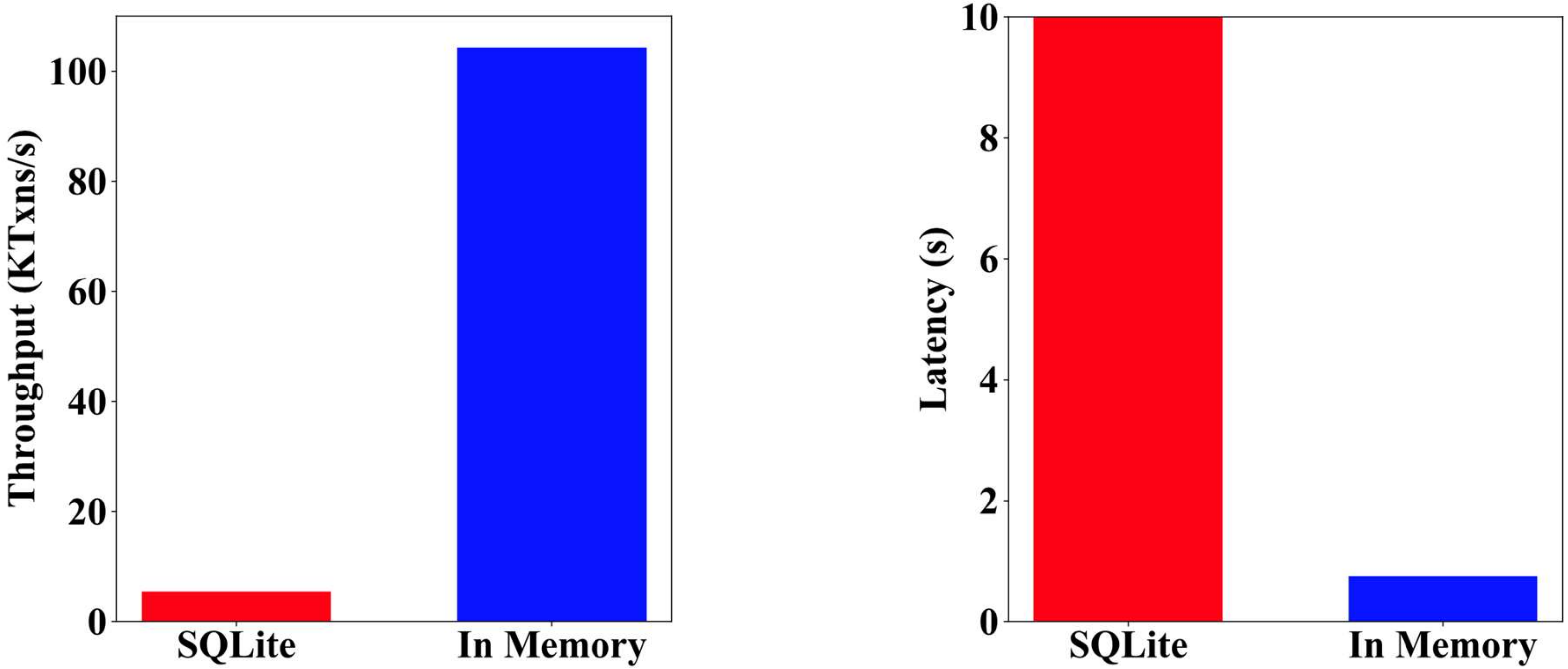


More transactions batched together → increase in throughput

→reduced phases of consensus.

ResilientDB ExpoLab Creativity Unfolded

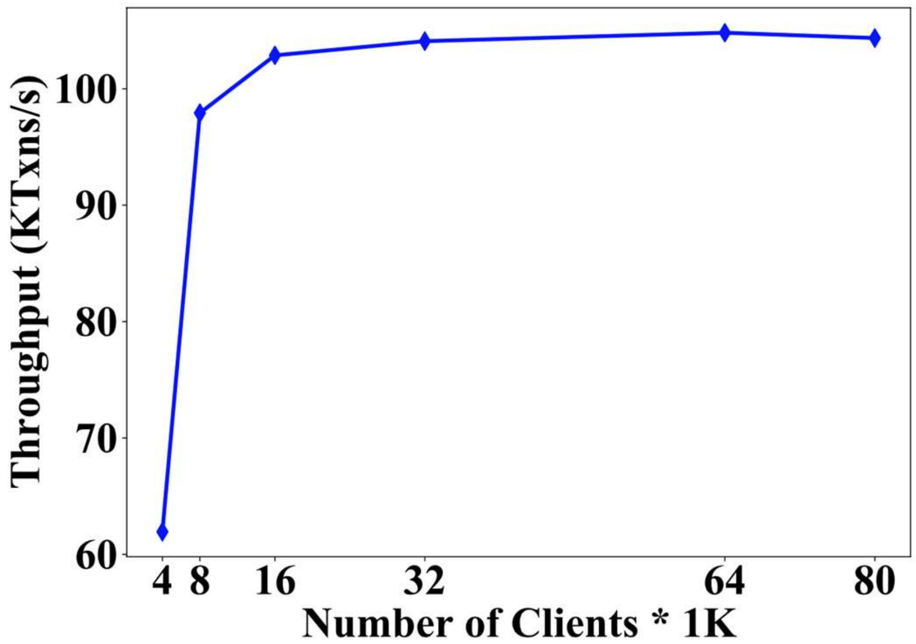
Insight 3: Memory Storage Gains

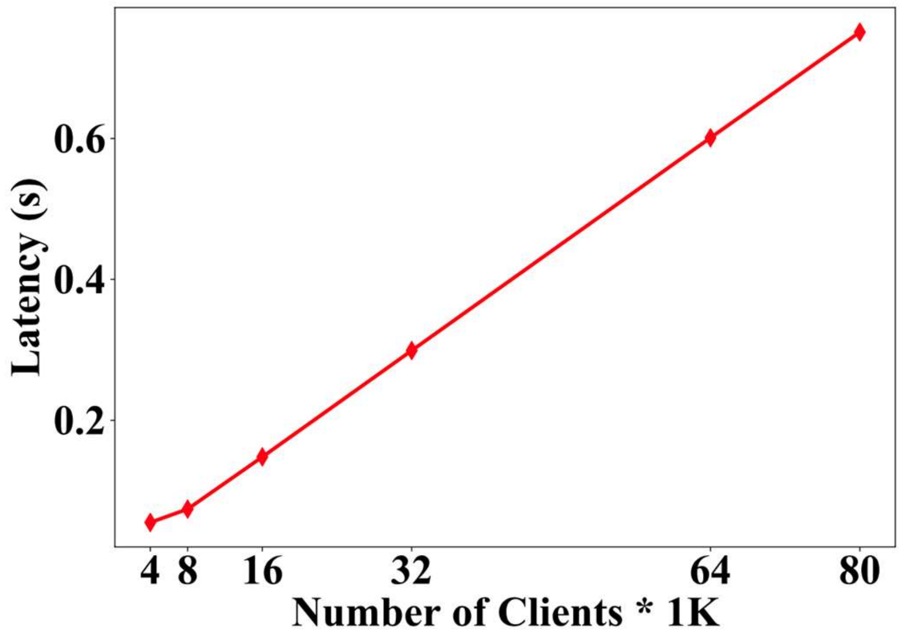


In-memory blockchain storage → reduces access cost.

ResilientDB ExpoLab Creativity Unfolded

Insight 4: Number of Clients





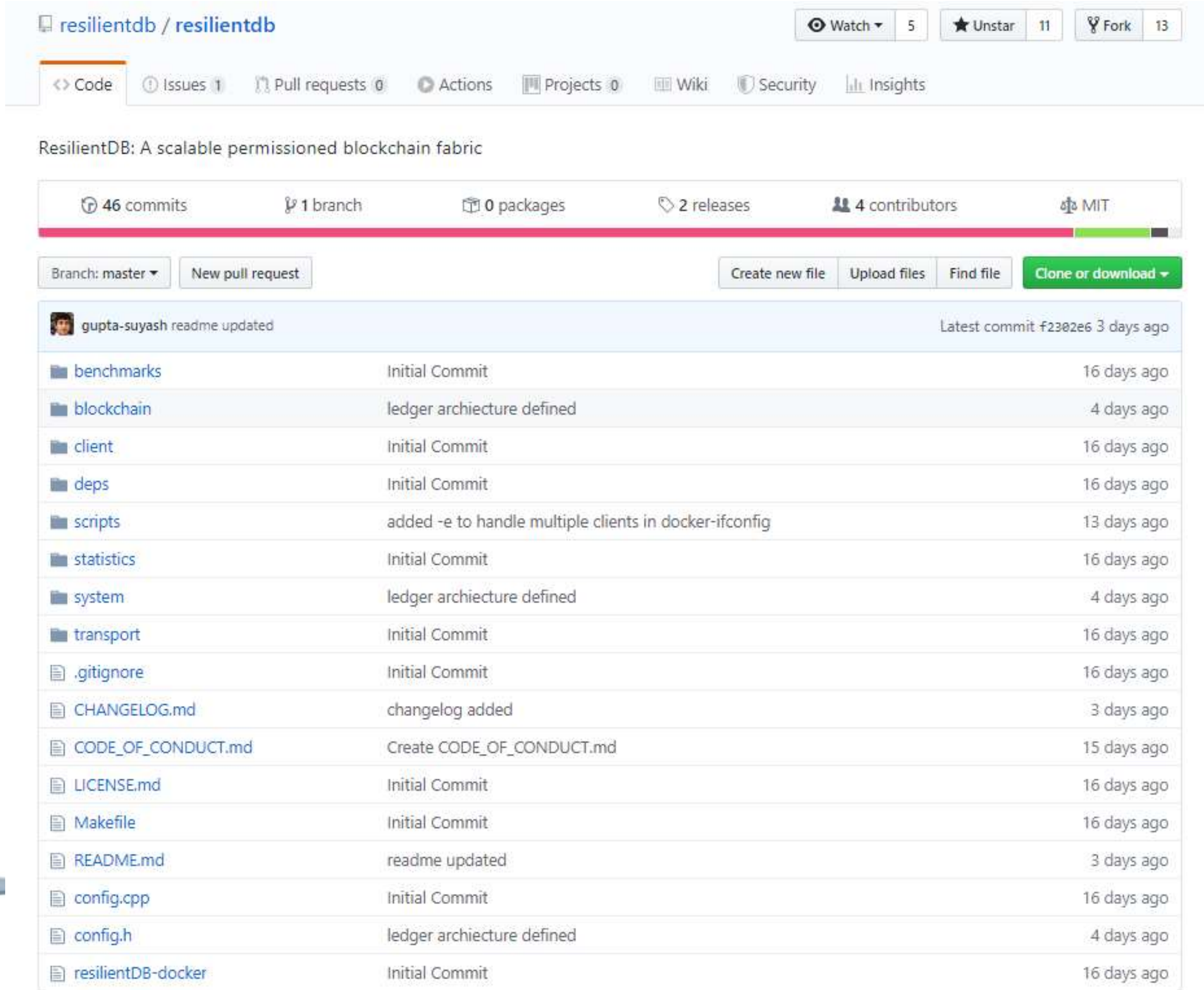
Too many clients → increases average latency.

ResilientDB ExpoLab Creativity Unfolded

ResilientDB: Hands On

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

How to Run ResilientDB?



ResilientDB ExpoLab Creativity Unfolded

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

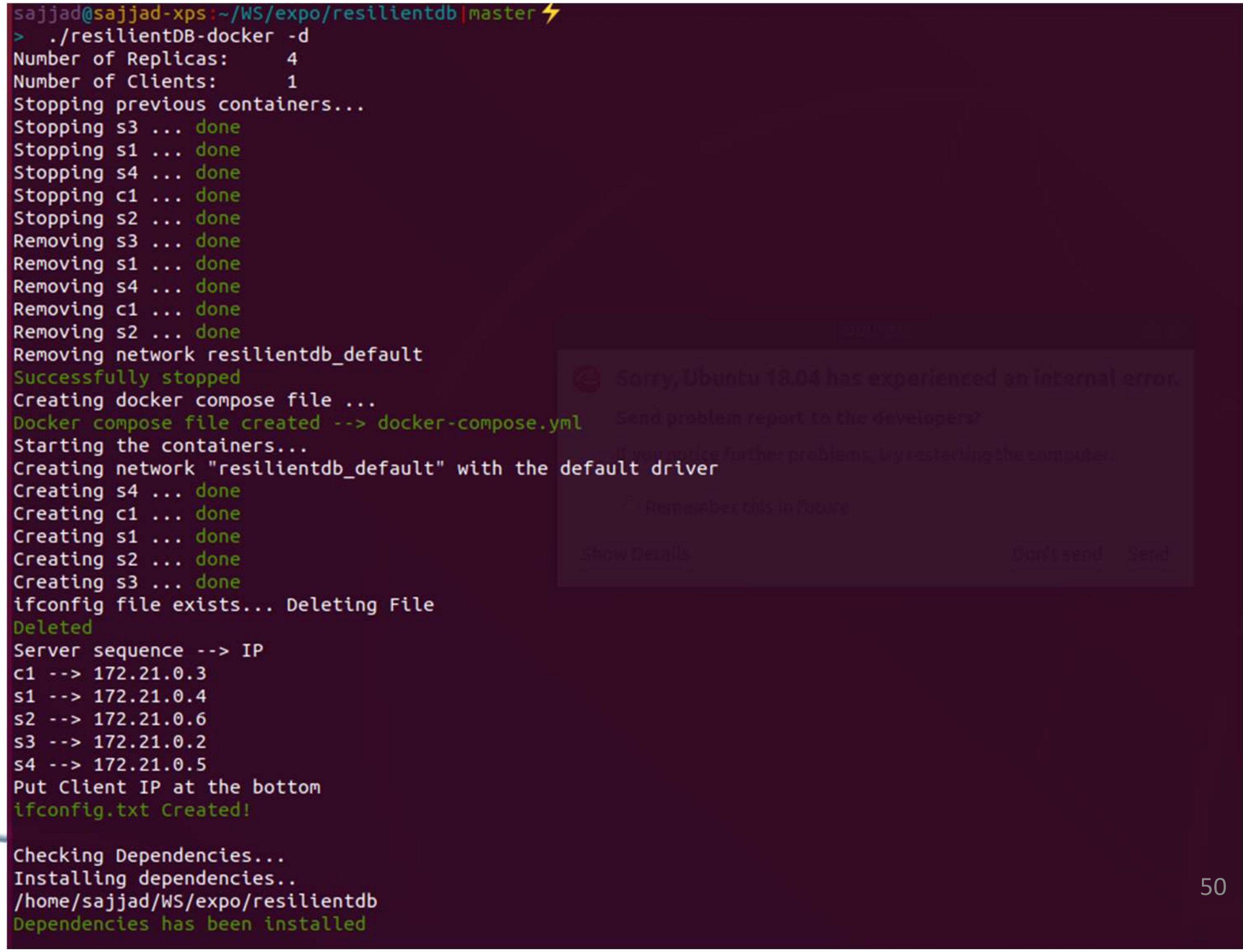
    ./resilientDB-docker -d [default 4 replicas and 1 client]

- 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



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

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();
```

ResilientDB ExpoLab Creativity Unfolded

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()

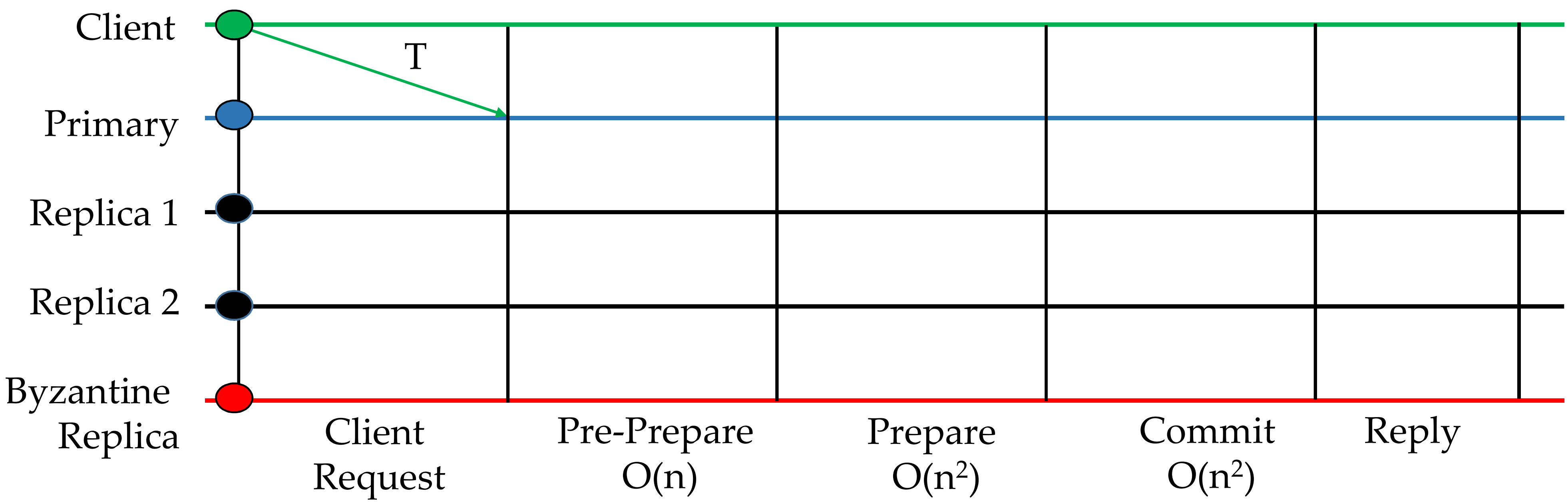
```cpp
/**
 * Starting point for each worker thread.
 *
 * Each worker-thread created in the main() starts here. Each worker-thread is alive
 * till the time simulation is not done, and continuousy perform a set of tasks.
 * Thess tasks involve, dequeuing a message from its queue and then processing 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;
    }
}
```

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

worker_thread_pbft.cpp

system > worker_thread_pbft.cpp > ...

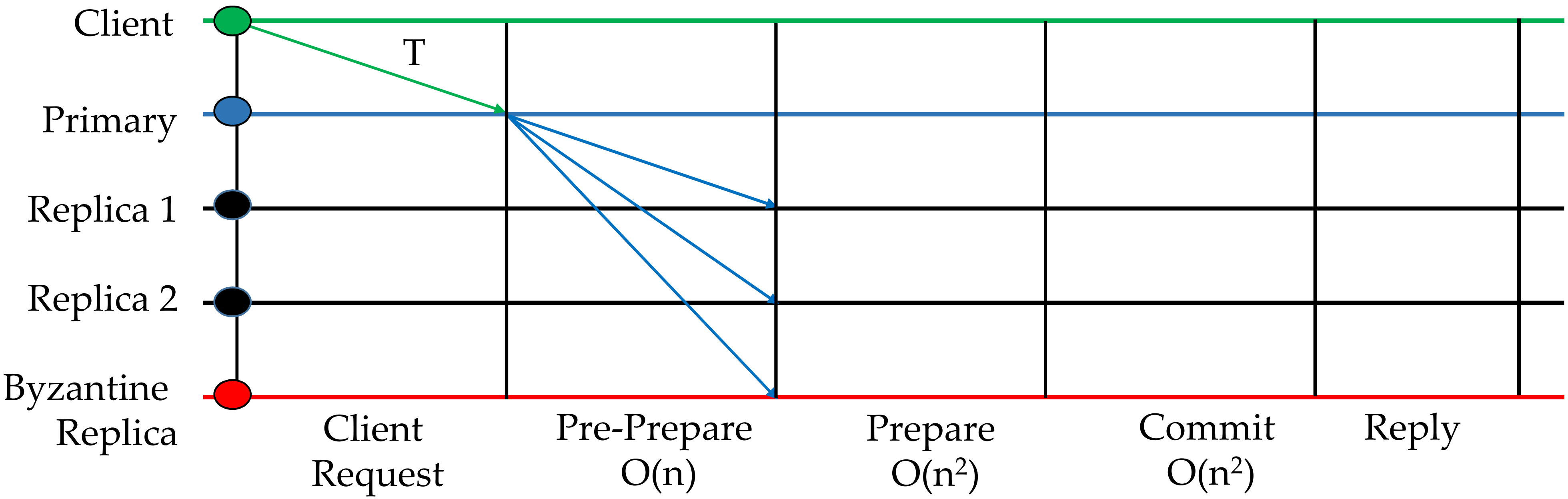
```cpp
/**
 * Processes an incoming client batch and sends a Pre-prepare message to al rep
 *
 * This function assumes that a client sends a batch of transactions and
 * for each transaction in the batch, a separate transaction manager is createc
 * Next, this batch is forwarded to all the replicas as a BatchRequests Message
 * 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->txn
    //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 Failure-Free Flow



PBFT: Practical Byzantine Fault Tolerance

Process Batch Request (Prepare)

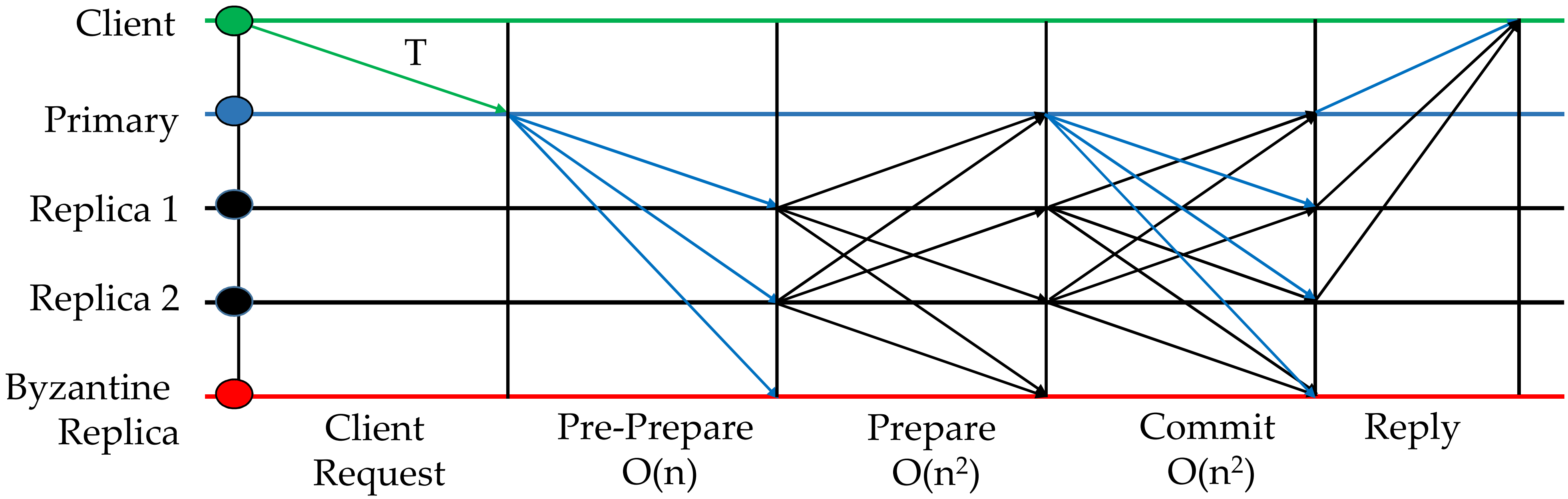
- System/worker_thread_pbft.cpp
- process_batch Function
- Create and Send Prepare Message
  - Create Transactions
  - Save Digest
- PBFTPrepare Class
  - Prepare Message

worker_thread_pbft.cpp

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

```cpp
/**
 * 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);

```

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

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 > ...
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

UCDAVIS