ResilientDB

Scalable, Resilient, and Configurable Permissioned Blockchain Fabric

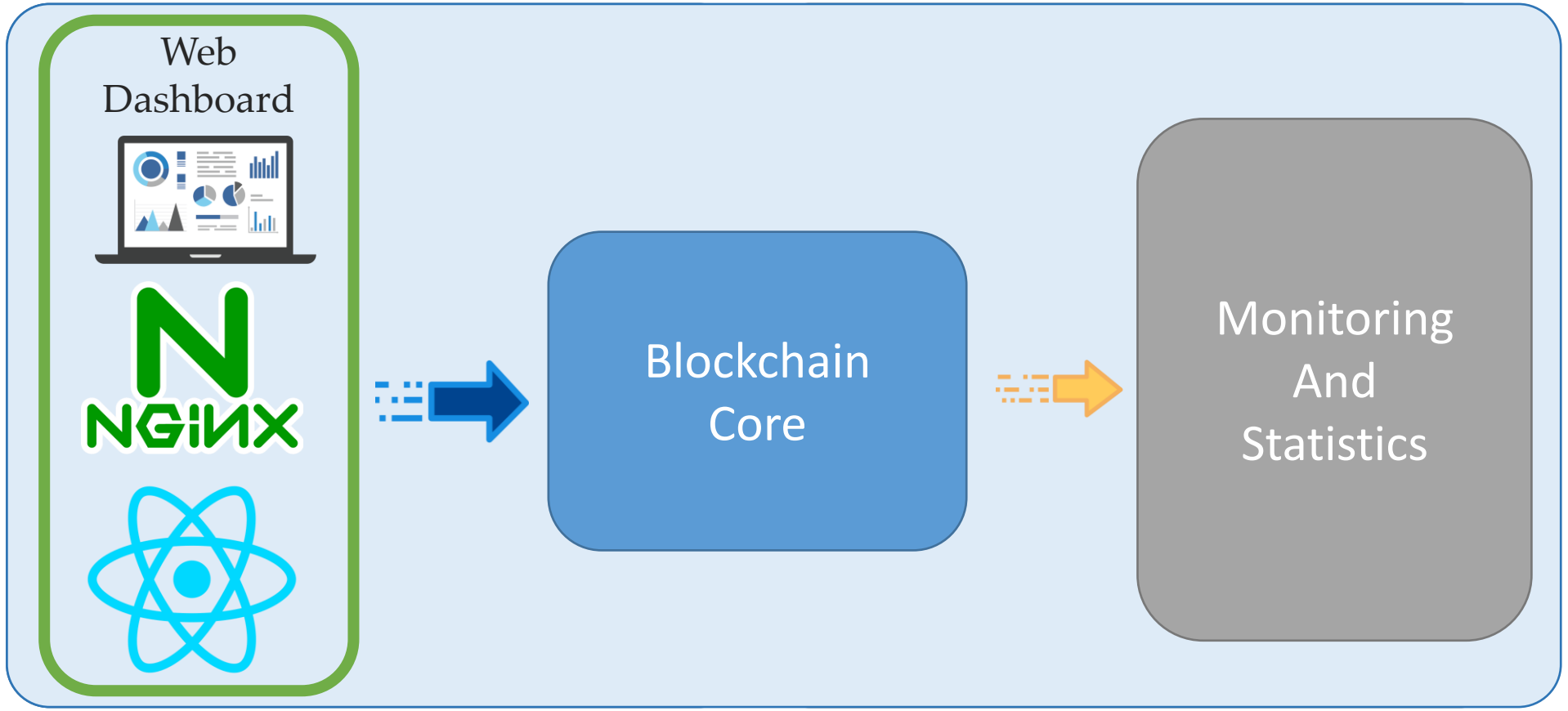
Sajjad Rahnama, Suyash Gupta, Mohammad Sadoghi

March 8th, 2020

ResilientDB

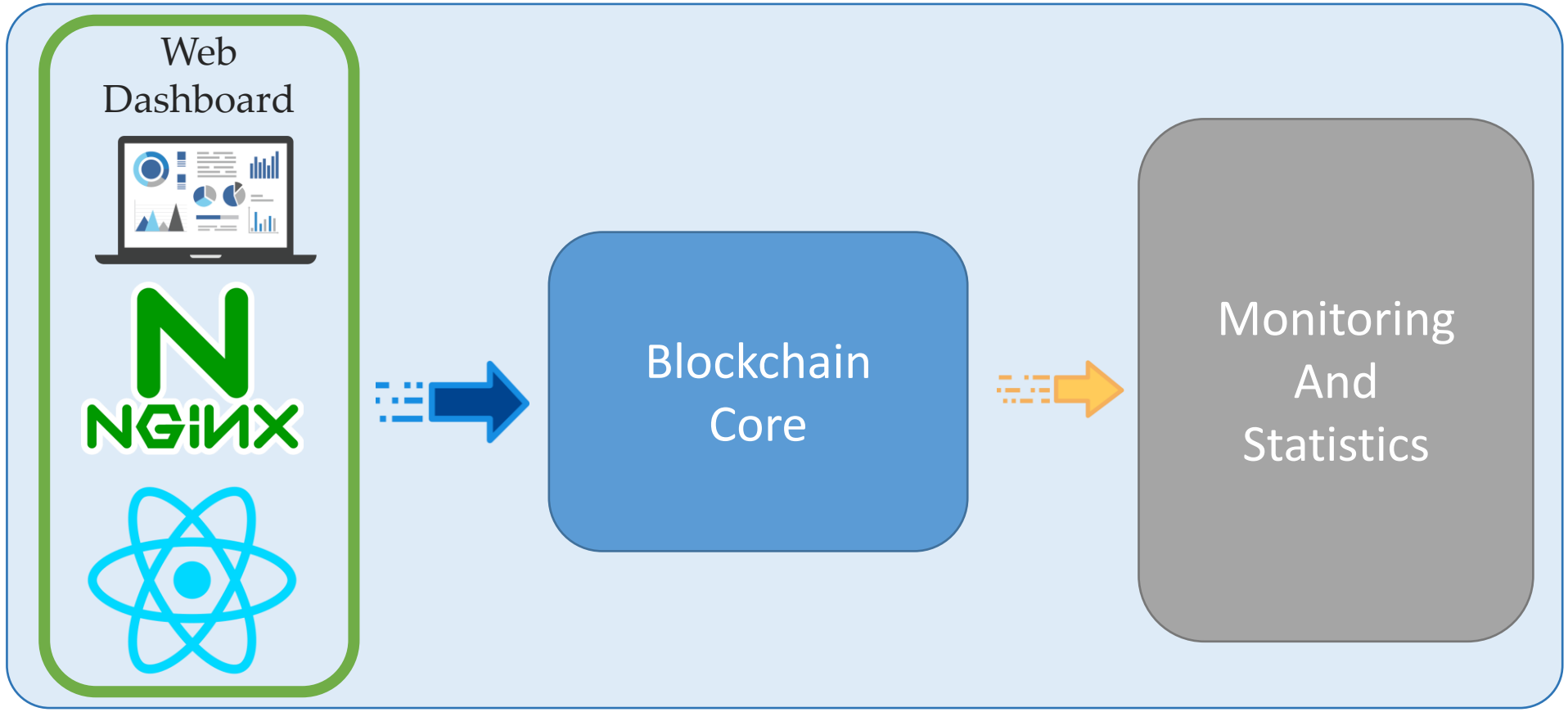
- Modular Design
- Fully pipelined multi threaded system
- Out of order message processing
- Transparent network layer
- High throughput and low latency
- Blockchain Core and Web Interface

Architecture

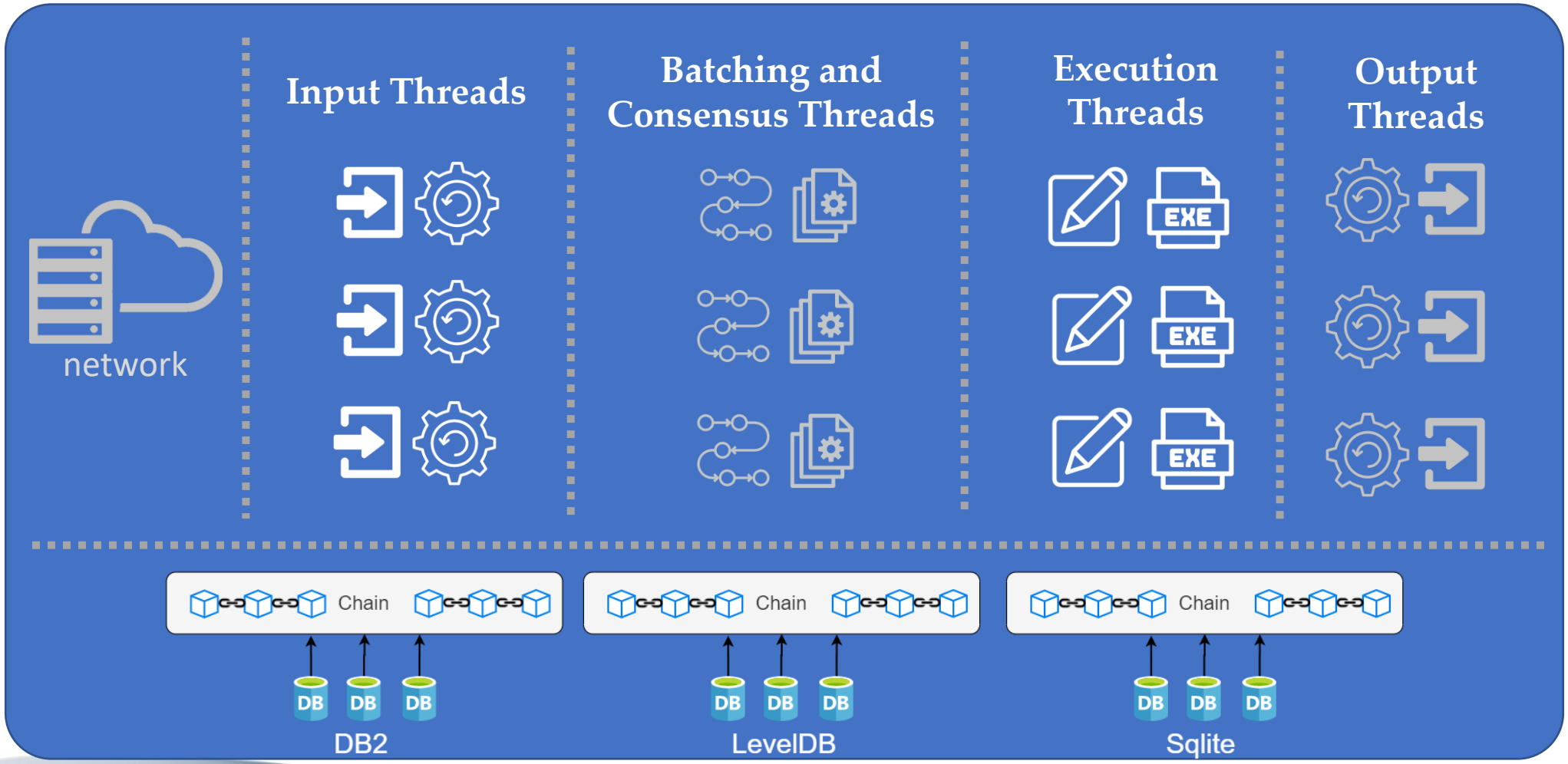


Interface and Monitoring

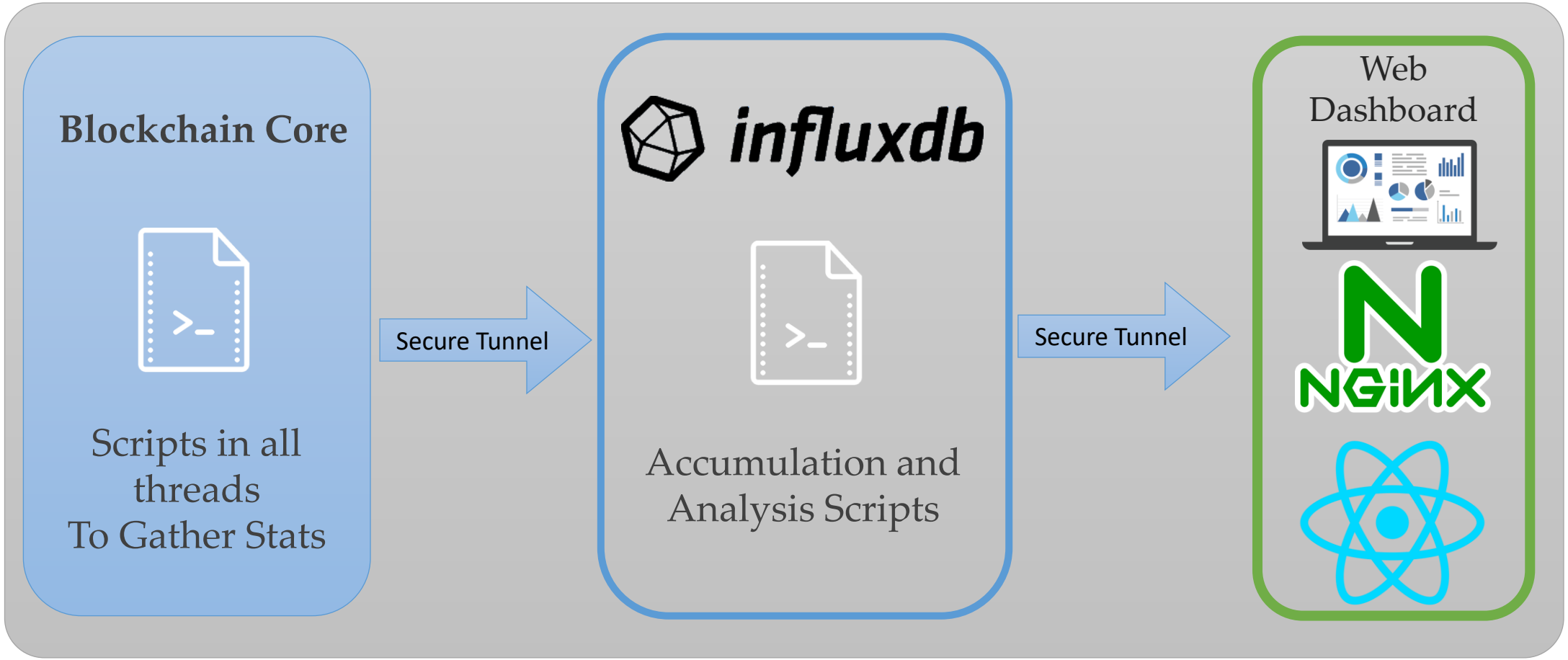
Architecture



Blockchain Core



Monitoring And Statistics



Blockchain Core Components

- Transport Layer (nanomsg)
- Input/Output Threads (Message Queues)
- Consensus Threads (Worker, Checkpoint, Batching)
- Execution and the chain
- Data structures and Transaction Managers
- Statistics Classes

Transport Layer

- Using Nanomsg Next Generation (NNG)
- Between each pair of nodes constant number of **pair sockets**
- Sockets will be **always open** throughout the run
- Sockets get created in in the **initialization phase** in replicas and clients
- **Ports** will get selected **deterministically** in all nodes
- Provides **send** and **receive API** to input and output layer based on **node_id**

Input/Output Threads

- Input Threads
  - While loop on sockets to receive messages
  - Place received messages in **works queues**
  - Each Input thread is responsible for **certain nodes**
- Output Threads
  - While loop on **message queues** to send messages
  - **Buffer messages** to put data on link efficiently
  - Each Output thread is responsible for **certain nodes**

```
class InputThread : public Thread
{
public:
    RC run();
    RC client_recv_loop();
    RC server_recv_loop();
    void check_for_init_done();
    void setup();
    void managekey(KeyExchange *keyex);
};

class OutputThread : public Thread
{
public:
    RC run();
    void check_and_send_batches();
    void send_batch(uint64_t dest_node_id);
    void copy_to_buffer(mbuf *sbuf, RemReqType type, BaseQuery *qry);
    uint64_t get_msg_size(RemReqType type, BaseQuery *qry);
    uint64_t get_thd_id() { return _thd_id; }
    uint64_t idle_starttime = 0;
};
```

Consensus Threads

- Responsible for **processing consensus messages**
- Based on the messages type they are called **worker**, **batching** or **checkpoint thread**
- They loop on **Work Queues** to pick up messages and process
- **Worker threads** process general consensus messages: prepare, commit,…
- **Batching threads** process client requests in the primary to create batches
- **Checkpoint threads** process checkpoint messages

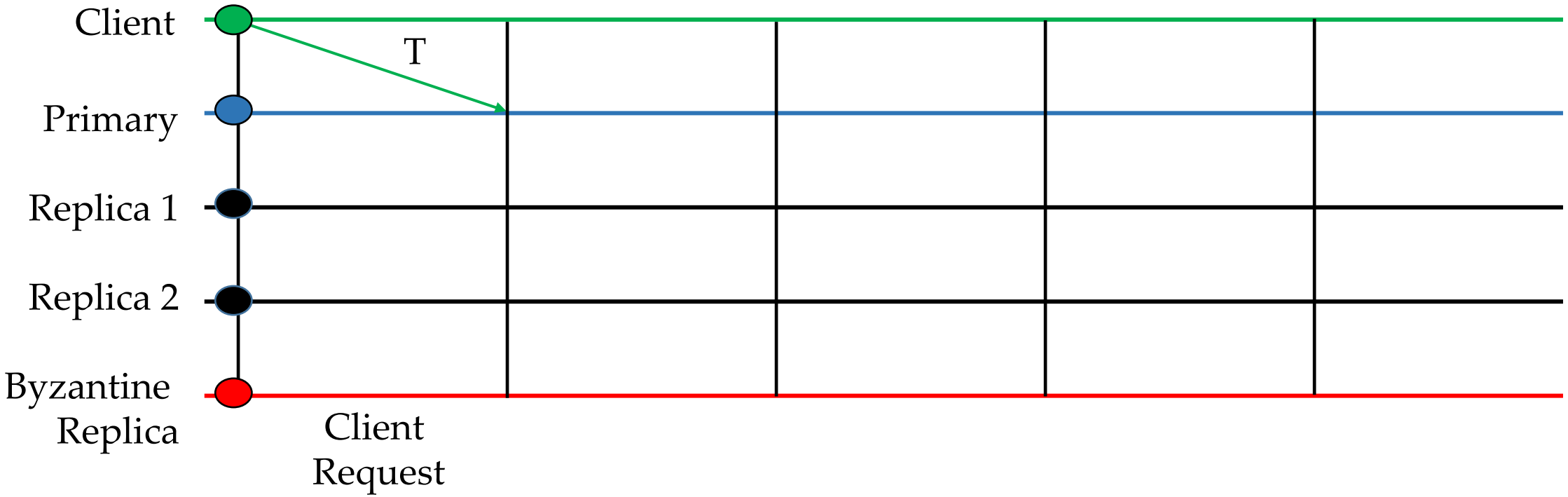
Consensus Threads

```
class WorkerThread : public Thread
{
public:
    RC run();
    void setup();
    void send_key();
    void process(Message *msg);
    void release_txn_man(uint64_t txn_id, uint64_t batch_id);
    void create_and_send_batchreq(ClientQueryBatch *msg, uint64_t tid);
    void set_txn_man_fields(BatchRequests *breq, uint64_t bid);
    bool validate_msg(Message *msg);
    void send_checkpoints(uint64_t txn_id);
    RC process_key_exchange(Message *msg);
    RC process_client_batch(Message *msg);
    RC process_batch(Message *msg);
    RC process_pbft_chkpt_msg(Message *msg);
    RC process_view_change_msg(Message *msg);
    RC process_new_view_msg(Message *msg);
    RC process_pbft_prep_msg(Message *msg);
    RC process_pbft_commit_msg(Message *msg);
    bool prepared(PBFTPrepMessage *msg);
    bool committed_local(PBFTCommitMessage *msg);

private:
    uint64_t _thd_txn_id;
    TxnManager *txn_man;
};
```

Execution & Chain

- Execute the transactions in linearizable order
- Transition in the **state machine** via **DB instance**
- **SQlite, In-Memory , LevelDB** Support
- Banking **Smart Contract** and **YCSB** support
- Ledger contains **chained hash** of **transactions**

Request Lifetime in ResilientDB



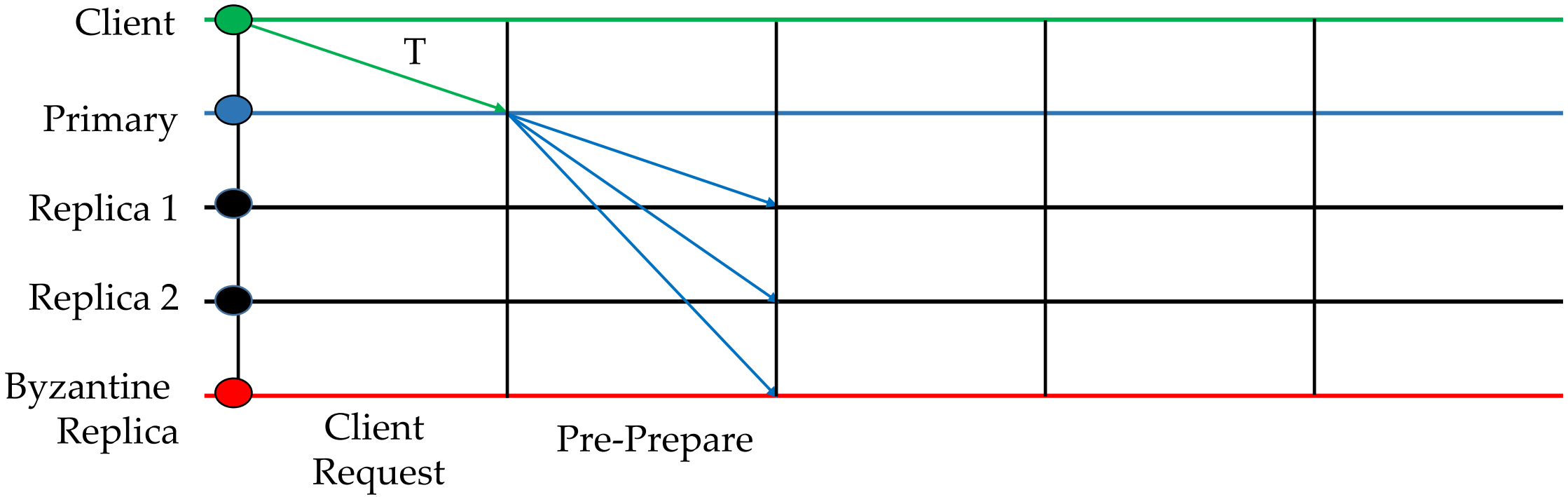
Request Lifetime in ResilientDB

Client Request

- An instance of **ClientQueryBatch** Message
- Gets Created in **client thread** (client loop)
- Client thread puts it into **message queue**
- Output threads pick it up pass it to **transport layer**
- It will be sent to the primary

```
  vector<string> signatures;
  signatures.push_back(client_batch->signature);

  vector<uint64_t> destinations;
  destinations.push_back(primary_id);\

  msg_queue.enqueue(get_thd_id(), client_batch,
      signatures, destinations);

  destinations.clear();


private:
    uint64_t last_send_time;
    uint64_t send_interval;
};

#if BANKING_SMART_CONTRACT
    Array<BankingSmartContractMessage *> cqrySet;
#else
    Array<YCSBClientQueryMessage *> cqrySet;
#endif
};
```

Request Lifetime in ResilientDB



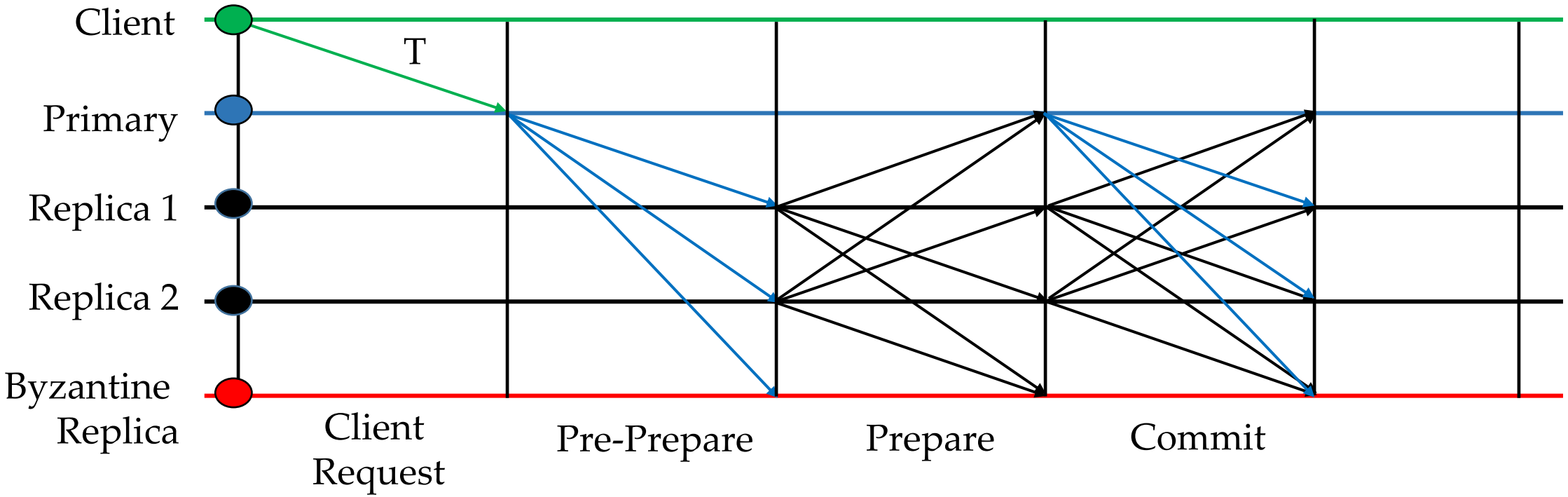
Request Lifetime in ResilientDB

Client Batch in Primary

- Input Threads in replicas and clients wait to receive messages from other nodes
- Batching Threads pick up ClientBatch from work queue
- Create Transactions and Transaction Managers
- Create digest, assign sequence number
- Send Pre-Prepare message to all nodes

```
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);


    return RCOK;
}
```

Request Lifetime in ResilientDB



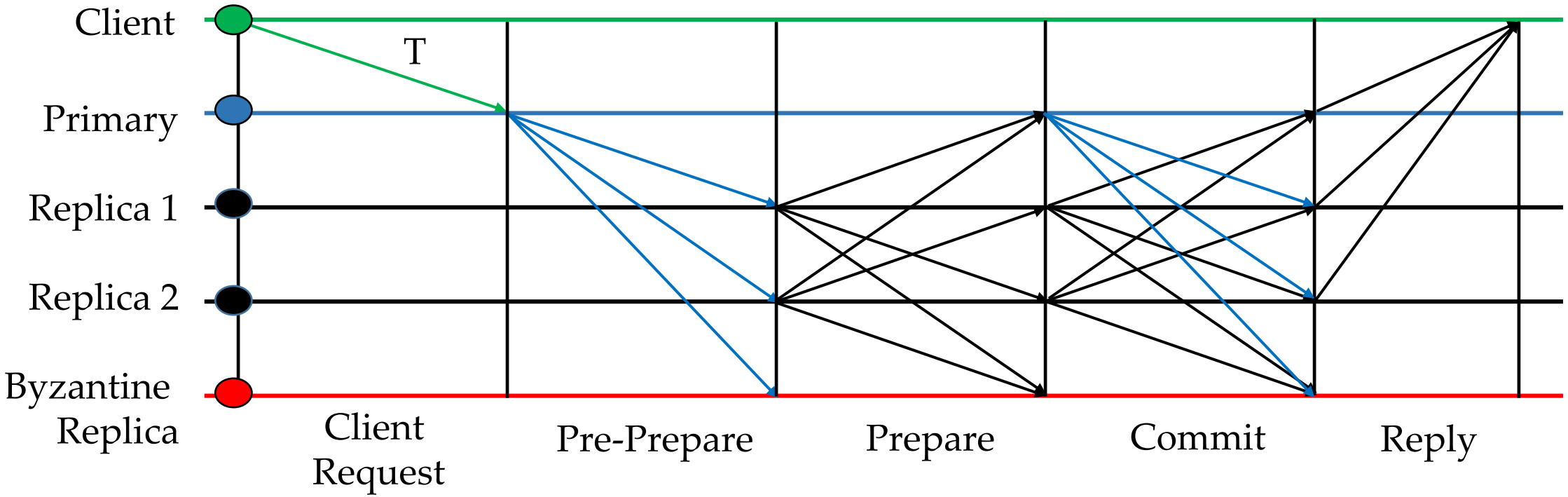
Request Lifetime in ResilientDB

Prepare and Commit Messages

- For Prepare → **process_prepare** function
  - Validate and count the number of messages
  - Send Commit Messages
- For Commit → **process_commit** function
  - Validate and count the number of messages
  - Check Committed local condition
  - Send Execute message

```
/**
 * Processes incoming Commit message.
 *
 * This functions precessing incoming messages of type PBFTCommitMessage. If a replica
 * received 2f+1 identical Commit messages from distinct replicas, then it asks the
 * 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_commit(Message *msg)
{
    // 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))
    {

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

        INC_STATS(get_thd_id(), time_commit, get_sys_clock() - txn_man->txn_stats.time_start_commit);
    }

    return RCOK;
}
```

Request Lifetime in ResilientDB



Request Lifetime in ResilientDB

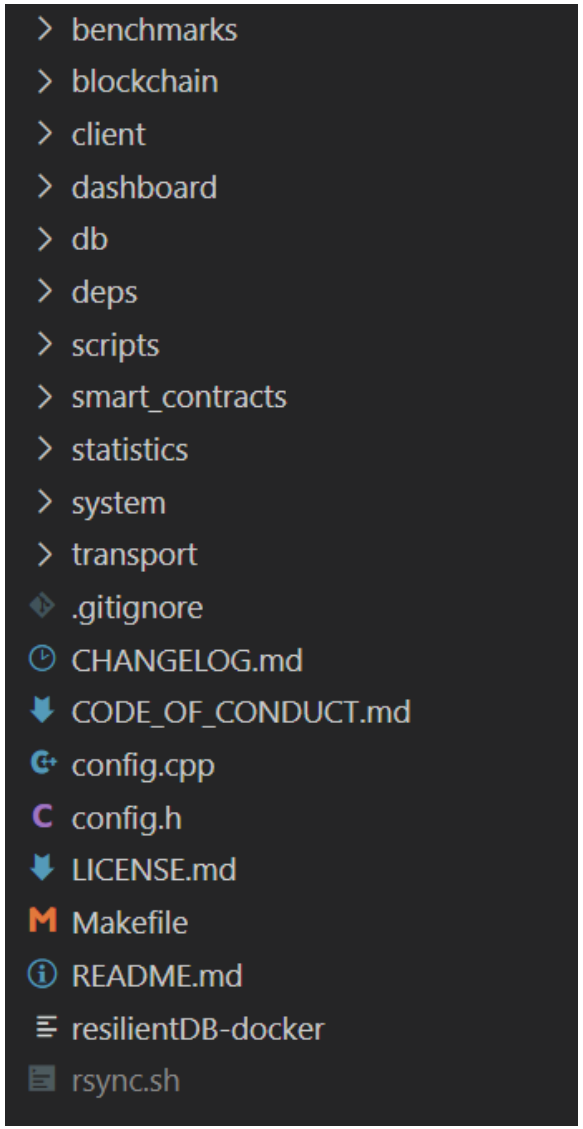
Execute and Reply

- Internal **Execute message** to execute thread
- Execute queues to force **linearizability**
- Execute YSCB or Smart Contract
- Add to chain
- Create and send Client Reply
- Send Checkpoint Messages

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

  crsp->copy_from_txn(txn_man);

  vector<string> signatures;
  vector<uint64_t> destinations;
  dest.push_back(txn_man->client_id);
  msg_queue.enqueue(get_thd_id(), crsp, signatures, destinations);
  dest.clear();

  INC_STATS(_thd_id, tput_msg, 1);
  INC_STATS(_thd_id, msg_cl_out, 1);

  // Check and Send checkpoint messages.
  send_checkpoints(txn_man->get_txn_id());

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

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

      crsp->copy_from_txn(tman);
  }
```

Directory Structure of ResilientDB

- Messages and Transport layer exist in **transport** folder
- Chain and database instances are in **benchmark** and **db** directories
- Worker threads and data structures are in **systems** folder
- Static smart contracts reside in **smart_contract** folder
- **Statistics** and **dashboard** folder contain UI and logging classes
- **Scripts** contains run, deploy, and gathering result scripts
- Client directory contains client threads and main functions



Implement Consensus Protocols

- Define your flow
- Define your messages
- Define process functions
- Modify clients for requests
- Define your execution model

```
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);
```

Questions?