T-PBFT: An EigenTrust-Based Practical Byzantine Fault Tolerance Consensus Algorithm

Original Authors: Sheng Gao, Tianyu Yu, Jianming Zhu, Wei Cai
Implemented and presented by: Fares Kechid, Mashal Zainab and Ribiea Ramzan
on 9th January 2024.

UNIVERSITÉ DU LUXEMBOURG

CONTENTS OF THIS PRESENTATION

01

INTRODUCTION TO THE PAPER

BLOCKCHAINS AND CONSENSUS ALGORITHMS

Problems with existing blockchain consensus algorithms:

- Proof of Work (PoW) and Proof of Stake (PoS) experience challenges related to power consumption and operational inefficiency.
- PBFT and HoneyBadgerBFT fall short of fulfilling scalability demands within a large-scale network.

CONTRIBUTIONS OF THE PAPER

- The paper introduces a novel framework T-PBFT, a multi-stage consensus algorithm.
- To reduce the probability of view change, T-PBFT proposes to replace a single primary node with a primary group.
- EigenTrust is proposed for constructing a reliable consensus group with elevated trust values to enhance consensus efficiency.
- T-PBFT is compared with other BFT-type algorithms, analyzing communication complexity, Byzantine fault tolerance, and theoretical view change probability.

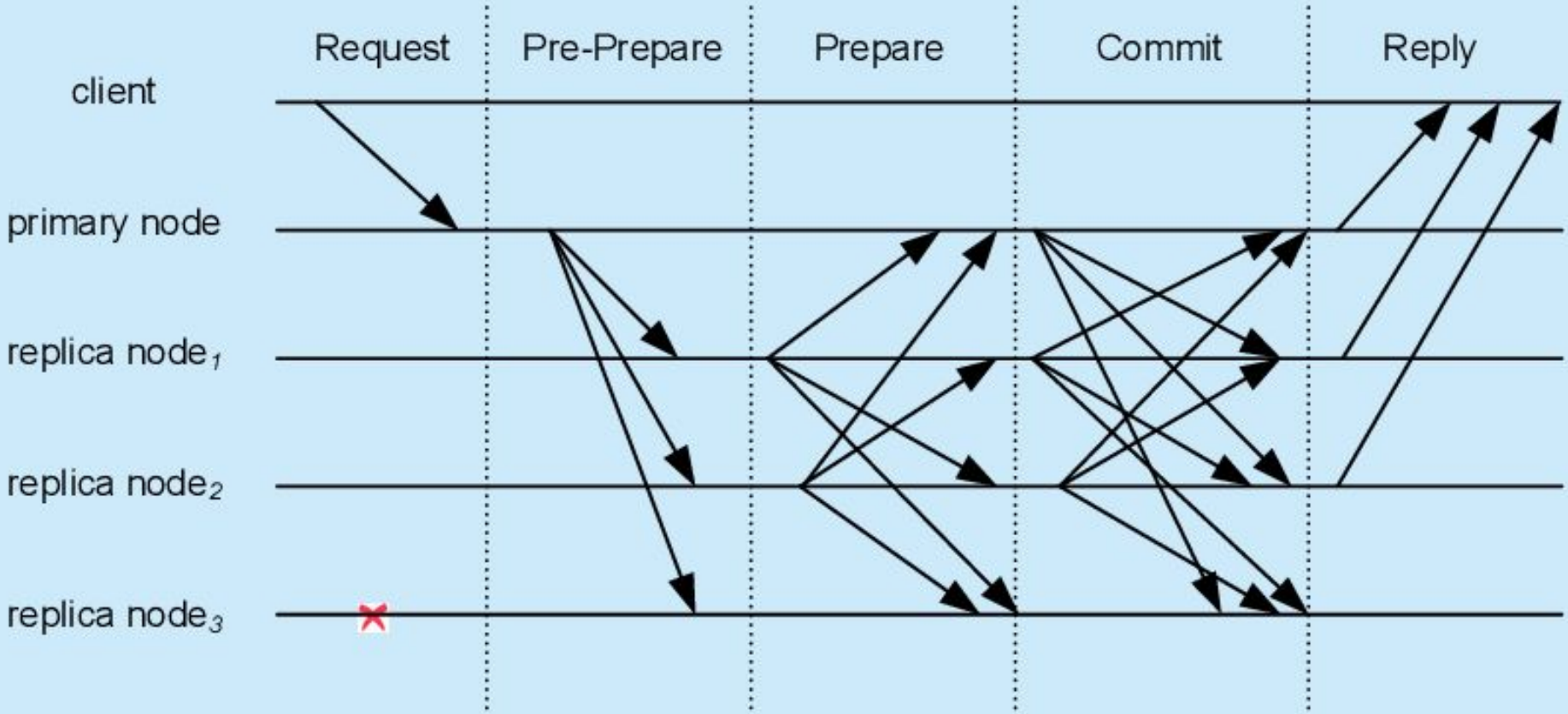
02

T-PBFT

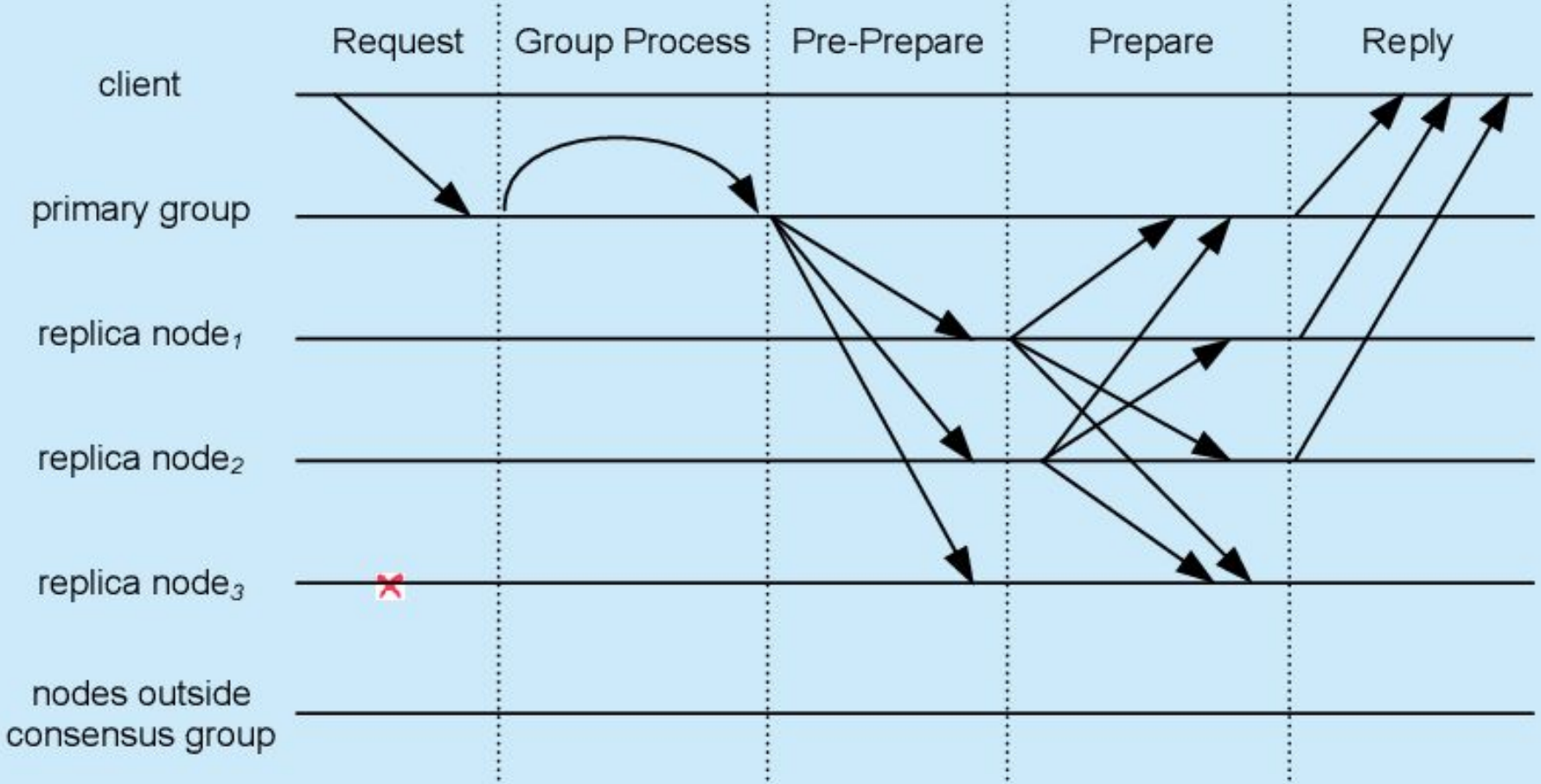
Foundations for T-PBFT

EigenTrust

PBFT



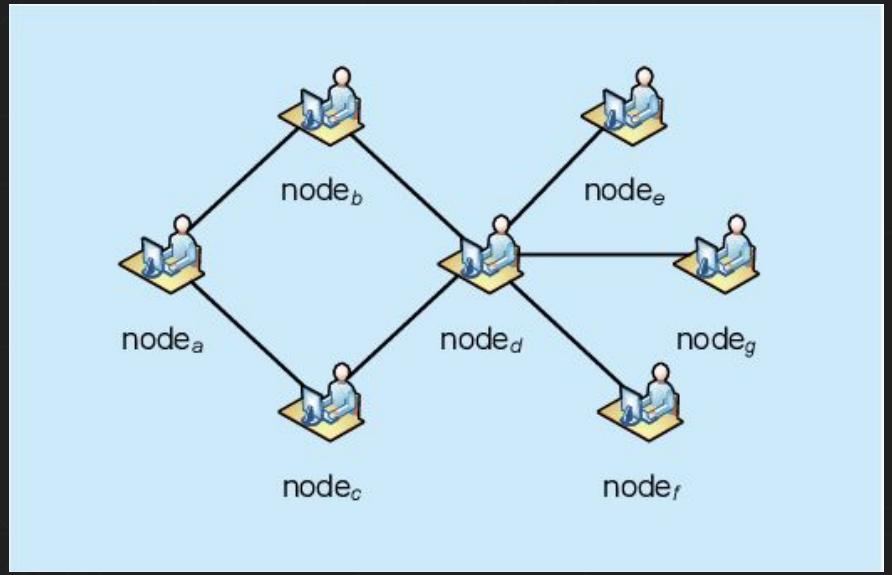
The operation process of PBFT.



The operation process of T-PBFT.

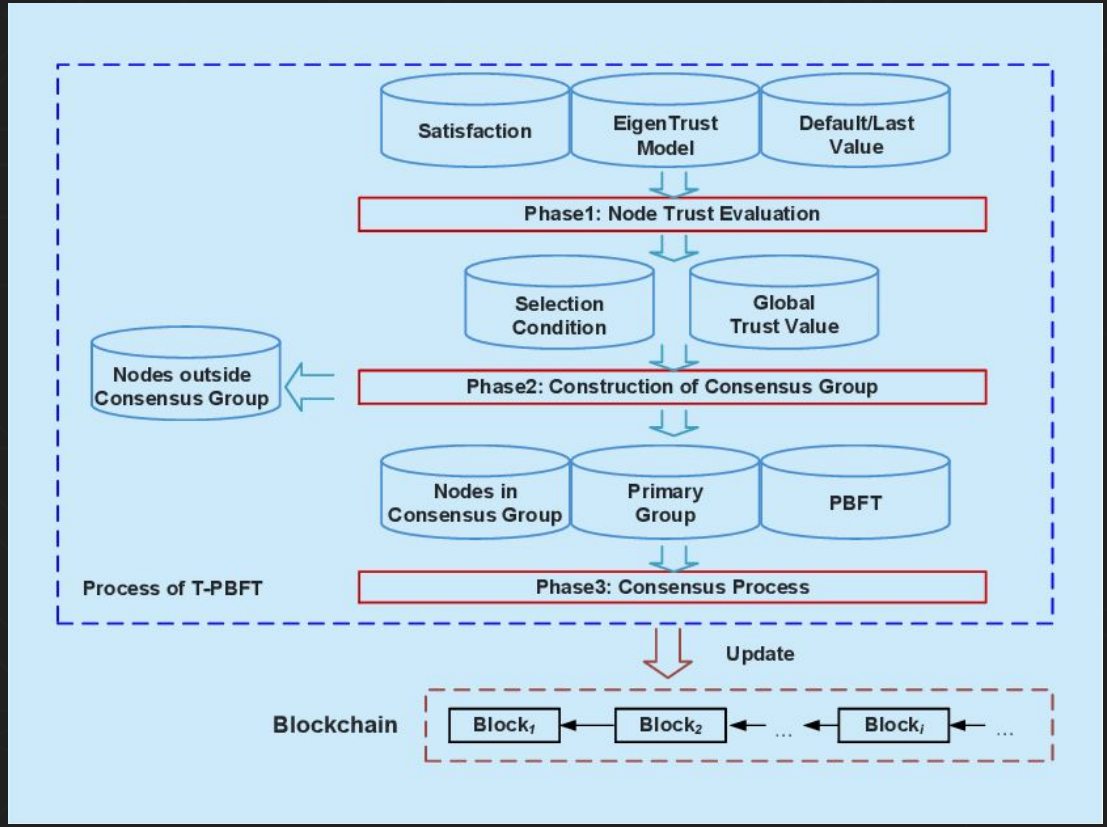
ASSUMPTIONS FOR T-PBFT

1. Behavior Consistency.
2. Limited Transaction time.
3. Additional assumption in our implementation: All nodes are directly connected to each other.



T-PBFT FRAMEWORK AND THE PROCESS

1. Node Trust Evaluation
2. Construction of Consensus Group
3. Consensus Process

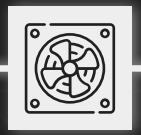


03

IMPLEMENTATIONAL DETAILS

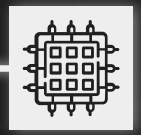
TRUST MANAGER

INCLUDES ALL THE ALGORITHMS USED



DATA STRUCTURES

Bringing abstract theoretical concepts through practical implementation.



SINGULAR ENTITY

The trust manager is charge of everything related to trust

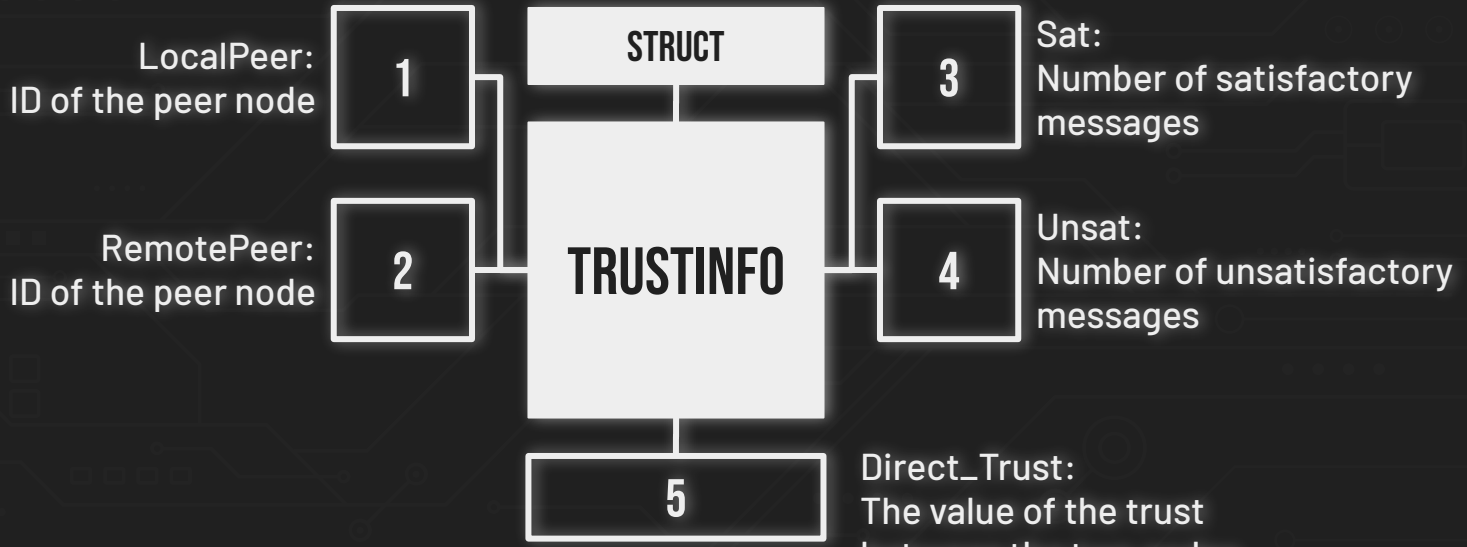
ABSTRACTION

It could be used in an abstract way by other element of the protocol

ENCAPSULATION

Every information should be retrieved in within the trust manager

TRUST MANAGER CORE COMPONENTS: DATA TYPE



TRUST MANAGER CORE COMPONENTS: TRUST MAPS

TRUST MAP

- Captures the direct trust relationship between nodes.
- Key: the value pair of the two nodes

GLOBAL TRUST MAP

- Maintains the global trust values of each node.
- Key: the node ID

TRUST MANAGER CORE COMPONENTS: VECTORS

CONSENSUS GROUP

Nodes forming the consensus group based on global trust percentages.

PRIMARY GROUP

Subset of nodes selected from the consensus group.

TRUST MANAGER: KEY OPERATIONS

	PURPOSE	PARAMETERS
UPDATE TRUST	Updates the trust information between two nodes based on message satisfaction.	Id of LocalPeer and RemotePeer, Boolean indicating message satisfaction.
UPDATES DIRECT TRUST	Updates the direct trust values between a local node and its immediate neighbors.	
UPDATE GLOBAL TRUST	Calculates and updates the global trust values for each node in the network	
GET CONSENSUS GROUP / PRIMARY GROUP	Forms groups of nodes based on specified percentage criteria from global trust values.	Percentage D and M from the total nodes that we have in order to create a group

CALCULATION OF F (BYZANTINE NODES)

$$f = \frac{dN - 1}{3} + (1 - d)N$$

SIGNATURES

KEYS	PURPOSE
SERVER PRIVATE KEY	Local Private Key is used to Sign the Message
SERVER PUBLIC KEY	Public Key is used to verify the signature made by private key. Available to all nodes/servers using a map **publicKeyIds**
GROUP PRIVATE KEY	The Group Private Key is used by the member of primary group to create **Group Signature** for pre-prepare message. Updated every time primary group is changed
GROUP PUBLIC KEY	The Group Public Key is used by consensus group node to verify the group signature. Updated every time primary group is changed

QUEUE

- Every server maintains a queue.
- Queue stores a pair value of client message and client id.
- A bool variable storing if the server is busy or not.
- Assumptions:
  - All Queues are in sync.
- Client Message Received:
  - Add message in the queue.
  - Check if server is busy.
  - If not, get the message and client id from queue and start operation process
  - When receive ack from client after client confirms f+1 messages, remove the request from queue.
  - Check if queue has other requests.
  - If yes, start the operation process

CREATING TRUST GROUPS

```
            // Initialize globalTrustMap with random Sat and Unsat values

        for (size_t i = 0; i < NUM_NODES; i++) {

            salticidae::PeerId node = nodes[i].peerId;
            TrustManager::updateGlobalTrust(node);

        }

    TrustManager::printGlobalTrustMap() ;

    TrustManager::getConsensusGroup(D2) ;
    TrustManager::printConsensusGroup() ;

    TrustManager::getPrimaryGroup(M2) ;
    TrustManager::printPrimaryGroup() ;
    RSAKeyGenerator::randomValueGenerator();
    //queueRequest::randomValueGenerator();
    RSAKeyGenerator::changeGroupKey();
}
```

OPERATION PROCESS

- GROUP REQUEST HANDLER
- PROCESS GROUP REQUEST
- PRIMARY CONSENSUS HANDLER
- PRIMARY VERIFIED HANDLER
- PREPREPARE HANDLER
- PREPARE HANDLER
- SEND REPLY HANDLER
- ACK HANDLER
- PROCESS NEXT REQUEST

```
88  void Node::group_request_handler(MsgGroup &&msg, const Net::conn_t &conn) {
89      cout <<"Node " << get_id() << "got client request message from " << conn->get_peer_id().to_hex() << "\n";
90      std::pair<salticidae::PeerId, std::string> client_request(conn->get_peer_id(), msg.message);
91      requestQueue.push_back(client_request);
92      peerNet->send_msg(MsgAck(), conn->get_peer_id());
93      //}
94      bool check = requestQueue.empty();
95      cout << "check : " << check << std::endl;
96      if (check == false && busy == false) {
97          std::pair<salticidae::PeerId, std::string> nextRequest(requestQueue.front());
98          salticidae::PeerId client_conn_id = nextRequest.first;
99          std::string client_conn_request = nextRequest.second;
100         cout << client_conn_request << " is message \n";
101         cout << client_conn_id.to_hex() << "is conn id \n";
102         Node::process_group_request(client_conn_request, client_conn_id);
103         busy=true;
104     }
105 }
```

OPERATION PROCESS

- GROUP REQUEST HANDLER
- **PROCESS GROUP REQUEST**
- PRIMARY CONSENSUS HANDLER
- PRIMARY VERIFIED HANDLER
- PREPREPARE HANDLER
- PREPARE HANDLER
- SEND REPLY HANDLER
- ACK HANDLER
- PROCESS NEXT REQUEST

```
106 void Node::process_group_request(const std::string msg, const salticidae::PeerId conn) {
107     // resetTimer();
108     // startTimer();
109         cout <<"Node " << get_id() << " processing client request message from " << conn.to_hex() << "\n";
110         RSAKeyGenerator keyManager;
111         std::string signature;
112         client_request = msg;
113         client_id = conn;
114         std::vector<salticidae::PeerId> primary = TrustManager::primaryGroup;
115         int val = RSAKeyGenerator::getRandomValue();
116         // cout << "Random value is " << val << "\n";
117         if (peerId == primary[val])
118         {
119             cout << "\tMessage: " << client_request << "\n";
120             RSAKeyGenerator::incrementOrderNumber();
121             int currentOrderNumber = RSAKeyGenerator::getOrderNumber();
122             std::string orderNumber = std::to_string(currentOrderNumber);
123             //std::string message = msg.message;
124             std::string combinedData = orderNumber + client_request;
125             signature = keyManager.SignMessage(combinedData, privateKey);
126             for (const auto &pid: primary) {
127                 if (pid != peerId) {
128                     peerNet->send_msg(MsgPrimaryConsensus(client_request, signature, orderNumber), pid);
129                 }
130             }
131         }
132         cout << "\tMessage: " << client_request << "\n";
133         //peerNet->send_msg(MsgAck(), conn);
134 }
```

OPERATION PROCESS

- GROUP REQUEST HANDLER
- PROCESS GROUP REQUEST
- **PRIMARY CONSENSUS HANDLER**
- PRIMARY VERIFIED HANDLER
- PREPREPARE HANDLER
- PREPARE HANDLER
- SEND REPLY HANDLER
- ACK HANDLER
- PROCESS NEXT REQUEST

```
void Node::primary_consensus_handler(MsgPrimaryConsensus &&msg, const Net::conn_t &conn) {
    cout <<"Node " << get_id() << "got signed message from " << conn->get_peer_id().to_hex() << "\n";
    RSAKeyGenerator keyManager;
    std::string orderNumber = msg.orderNumber;
    salticidae::PeerId connect = conn->get_peer_id();
    CryptoPP::RSA::PublicKey pubKey = publicKeysID[connect];
    std::string combinedData = orderNumber + client_request;
    //cout<< "client request " << client_request << "\n";
    bool verify = keyManager.VerifySignature(combinedData, msg.signature, pubKey);
    if (verify == true)
    {
        peerNet->send_msg(MsgPrimaryVerified(verify), conn);
    } else {
        cout << "Verification failed, node might be faulty \n";
        peerNet->send_msg(MsgPrimaryVerified(false), conn);
    }
}
```

OPERATION PROCESS

- GROUP REQUEST HANDLER
- PROCESS GROUP REQUEST
- PRIMARY CONSENSUS HANDLER
- **PRIMARY VERIFIED HANDLER**
- PREPREPARE HANDLER
- PREPARE HANDLER
- SEND REPLY HANDLER
- ACK HANDLER
- PROCESS NEXT REQUEST

```
void Node::primary_verified_handler(MsgPrimaryVerified &&msg, const Net::conn_t &conn) {
    cout <<"Node " << get_id() << "got verification message from " << conn->get_peer_id().to_hex() << "\n";
    verified_message.push_back(msg.verified);
    int verifyTrue = 0;
    int verifyFalse = 0;
    RSAKeyGenerator keyManager;
    std::vector<salticidae::PeerId> primary = TrustManager::primaryGroup;
    int primary_size = primary.size();
    if (verified_message.size() == primary.size() - 1) {
        for (bool v : verified_message) {
            if (v == true) {
                verifyTrue++;
            } else {
                verifyFalse++;
            }
        }
    }
    if (verifyTrue >= verifyFalse && verifyTrue != 0) {
        //add send msg
        cout << "Verified \n";
        int currentOrderNumber = RSAKeyGenerator::getOrderNumber();
        cout << currentOrderNumber << " is the order number \n";
        currentOrderNumber = RSAKeyGenerator::getOrderNumber();
        std::string orderNumber = std::to_string(currentOrderNumber);
        std::string combinedData = orderNumber + client_request;
        try {
            CryptoPP::RSA::PrivateKey groupPrivateKey = RSAKeyGenerator::getGroupPrivateKey(peerId);
            std::string signature = keyManager.SignMessage(combinedData, groupPrivateKey);
            std::vector<salticidae::PeerId> consensus = TrustManager::consensusGroup;
            for (const auto &pid: consensus) {
                if (std::find(primary.begin(), primary.end(), pid) == primary.end()) {
                    peerNet->send_msg(MsgPreprepare(client_request, signature, orderNumber), pid);
                }
            }
            verified_message.clear();
```

OPERATION PROCESS

- GROUP REQUEST HANDLER
- PROCESS GROUP REQUEST
- PRIMARY CONSENSUS HANDLER
- **PRIMARY VERIFIED HANDLER**
- PREPREPARE HANDLER
- PREPARE HANDLER
- SEND REPLY HANDLER
- ACK HANDLER
- PROCESS NEXT REQUEST

```
} else if (verifyFalse == primary_size-1) {
    requestQueue.pop_front();
    requestQueue.size();
    client_request = "";
    busy = false;
    bool check = requestQueue.empty();
    TrustManager::getConsensusGroup(D2) ;
    TrustManager::getPrimaryGroup(M2) ;
    RSAKeyGenerator::changeGroupKey();
    peerNet->multicast_msg(MsgNextRequest(true), peers);
    if(!check && !busy) {
        std::pair<salticidae::PeerId, std::string> nextRequest(requestQueue.front());
        salticidae::PeerId client_conn_id = nextRequest.first;
        std::string client_conn_request = nextRequest.second;
    // MsgGroup receivedMsg = client_conn_request;
        Node::process_group_request(client_conn_request, client_conn_id);
        //queueRequest::ifBusy = true;
        busy=true;
    }
} else if (verifyFalse > verifyTrue) {
    requestQueue.pop_front();
    requestQueue.size();
    client_request = "";
    busy = false;
    bool check = requestQueue.empty();
    TrustManager::getConsensusGroup(D2) ;
    TrustManager::getPrimaryGroup(M2) ;
    RSAKeyGenerator::changeGroupKey();
    peerNet->multicast_msg(MsgNextRequest(true), peers);
    if(!check && !busy) {
        std::pair<salticidae::PeerId, std::string> nextRequest(requestQueue.front());
        salticidae::PeerId client_conn_id = nextRequest.first;
        std::string client_conn_request = nextRequest.second;
    // MsgGroup receivedMsg = client_conn_request;
        Node::process_group_request(client_conn_request, client_conn_id);
        //queueRequest::ifBusy = true;
        busy=true;
    }
```

OPERATION PROCESS

- GROUP REQUEST HANDLER
- PROCESS GROUP REQUEST
- PRIMARY CONSENSUS HANDLER
- PRIMARY VERIFIED HANDLER
- **PREPREPARE HANDLER**
- PREPARE HANDLER
- SEND REPLY HANDLER
- ACK HANDLER
- PROCESS NEXT REQUEST

```
void Node::preprepare_handler(MsgPreprepare &&msg, const Net::conn_t &conn) {
    cout << "Node " << get_id() << " got pre-prepare message from " << conn->get_peer_id().to_hex()<<"\n";
    pauseTimer();
    cout << "\tMessage: " << msg.message << "\n";
    cout << "\tFrom " << conn->get_peer_id().to_hex() << "\n";
    RSAKeyGenerator keyManager;
    std::string message = msg.message;
    std::string orderNumber = msg.orderNumber;
    std::string combinedData = orderNumber + client_request;
    CryptoPP::RSA::PublicKey groupPublicKey = RSAKeyGenerator::getGroupPublicKey();
    bool verify = keyManager.VerifySignature(combinedData, msg.signature, groupPublicKey);
    cout << "bool value verify : " << verify << std::endl;
    if (verify == true) {
        cout << "Verified, sending to peers \n";
        std::string peer_signature = keyManager.SignMessage(combinedData, privateKey);
        std::vector<salticidae::PeerId> consensus = TrustManager::consensusGroup;
        for (const auto &pid: consensus) {
            if(pid != peerId) {
                peerNet->send_msg(MsgPrepare(message, orderNumber, msg.signature, peer_signature), pid);
            }
        }
    } else {
        cout << "Verification failed \n";
    }
}
```

OPERATION PROCESS

- GROUP REQUEST HANDLER
- PROCESS GROUP REQUEST
- PRIMARY CONSENSUS HANDLER
- PRIMARY VERIFIED HANDLER
- PREPREPARE HANDLER
- **PREPARE HANDLER**
- SEND REPLY HANDLER
- ACK HANDLER
- PROCESS NEXT REQUEST

```
void Node::prepare_handler(MsgPrepare &&msg, const Net::conn_t &conn){
    cout << "Node " << get_id() << " got prepare message from " << conn->get_peer_id().to_hex()<<"\n";
    RSAKeyGenerator keyManager;
    std::string combinedData = msg.orderNumber + client_request;
    CryptoPP::RSA::PublicKey pubKey = publicKeysID[conn->get_peer_id()];
    bool verify_peer = keyManager.VerifySignature(combinedData, msg.peerSign, pubKey);
    CryptoPP::RSA::PublicKey groupPublicKey = RSAKeyGenerator::getGroupPublicKey();
    bool verify_group = keyManager.VerifySignature(combinedData, msg.groupSign, groupPublicKey);
    if(verify_group == true && verify_peer == true) {
        //cout << "verified \n";
        RSAKeyGenerator::incrementPreparedMessages();
        // cout << RSAKeyGenerator::getPreparedMessages() << "\n";
    } else if (verify_group == true && verify_peer == false) {
        cout << conn->get_peer_id().to_hex() <<" might be faulty \n";
    } else if (verify_group == false && verify_peer == true) {
        cout << "Primary group node might be faulty \n";
    } else {
        cout << conn->get_peer_id().to_hex() << " and Primary group node both might be faulty \n";
    }
```

OPERATION PROCESS

- GROUP REQUEST HANDLER
- PROCESS GROUP REQUEST
- PRIMARY CONSENSUS HANDLER
- PRIMARY VERIFIED HANDLER
- PREPREPARE HANDLER
- **PREPARE HANDLER**
- SEND REPLY HANDLER
- ACK HANDLER
- PROCESS NEXT REQUEST

```
int prepared_message = RSAKeyGenerator::getPreparedMessages();
int fault_tolerance = 2 * ((((D2*nodes) - 1) / 3)+((1-D2) * nodes));
std::vector<salticidae::PeerId> consensus = TrustManager::consensusGroup;
std::vector<salticidae::PeerId> primary = TrustManager::primaryGroup;
int consensus_size = consensus.size();
int primary_size = primary.size();
int max_prepared = (consensus_size - primary_size) * (consensus_size -1);
if (prepared_message >= fault_tolerance) {
    cout<< "Total prepared messages: " << prepared_message << "messages \n";
    //Msg Reply will come here
    RSAKeyGenerator::clearPreparedMessages();
    //std::vector<salticidae::PeerId> consensus = TrustManager::consensusGroup;
    for( const auto &pid :consensus) {
        try
        {
            if(pid != peerId) {
                peerNet->send_msg(MsgSend(true, msg.orderNumber), pid);
            }
        }
        catch(const std::exception& e)
        {
            std::cerr << e.what() << '\n';
        }
        cout<<"sent \n";
    }
    peerNet->send_msg(MsgReply(true, msg.orderNumber), client_id);
    // Create PeerId for local and remote peers.
    salticidae::PeerId localPeer = peerId;
    salticidae::PeerId remotePeer = conn->get_peer_id();
    // Update trust information using TrustManager.
    TrustManager::updateTrust(localPeer, remotePeer, (verify_peer)); //add verify here
```

OPERATION PROCESS

- GROUP REQUEST HANDLER
- PROCESS GROUP REQUEST
- PRIMARY CONSENSUS HANDLER
- PRIMARY VERIFIED HANDLER
- PREPREPARE HANDLER
- **PREPARE HANDLER**
- SEND REPLY HANDLER
- ACK HANDLER
- PROCESS NEXT REQUEST

```
} else if (prepared_message == max_prepared){
    RSAKeyGenerator::clearPreparedMessages();
    requestQueue.pop_front();
    requestQueue.size();
    client_request = "";
    busy = false;
    TrustManager::getConsensusGroup(D2) ;
    TrustManager::getPrimaryGroup(M2) ;
    RSAKeyGenerator::changeGroupKey();
    bool check = requestQueue.empty();
    peerNet->multicast_msg(MsgNextRequest(true), peers);
    cout << "Check queue 2: " << check << ".Size is:  " <<requestQueue.size()<< "\n";
    if(!check && !busy) {
        std::pair<salticidae::PeerId, std::string> nextRequest(requestQueue.front());
        salticidae::PeerId client_conn_id = nextRequest.first;
        std::string client_conn_request = nextRequest.second;
    // MsgGroup receivedMsg = client_conn_request;
        Node::process_group_request(client_conn_request, client_conn_id);
        //queueRequest::ifBusy = true;
        busy=true;
    }
}
```

OPERATION PROCESS

- GROUP REQUEST HANDLER
- PROCESS GROUP REQUEST
- PRIMARY CONSENSUS HANDLER
- PRIMARY VERIFIED HANDLER
- PREPREPARE HANDLER
- PREPARE HANDLER
- **SEND REPLY HANDLER**
- ACK HANDLER
- PROCESS NEXT REQUEST

```
//send reply handler
void Node::send_reply_handler(MsgSend &&msg, const Net::conn_t &conn) {
    cout << "Node " << get_id() << " got send reply message from " << conn->get_peer_id().to_hex()<<"\n";
    peerNet->send_msg(MsgReply(msg.sendReply, msg.orderNumber), client_id);
    RSAKeyGenerator::clearPreparedMessages();
}
```

OPERATION PROCESS

- GROUP REQUEST HANDLER
- PROCESS GROUP REQUEST
- PRIMARY CONSENSUS HANDLER
- PRIMARY VERIFIED HANDLER
- PREPREPARE HANDLER
- PREPARE HANDLER
- SEND REPLY HANDLER
- **ACK HANDLER**
- PROCESS NEXT REQUEST

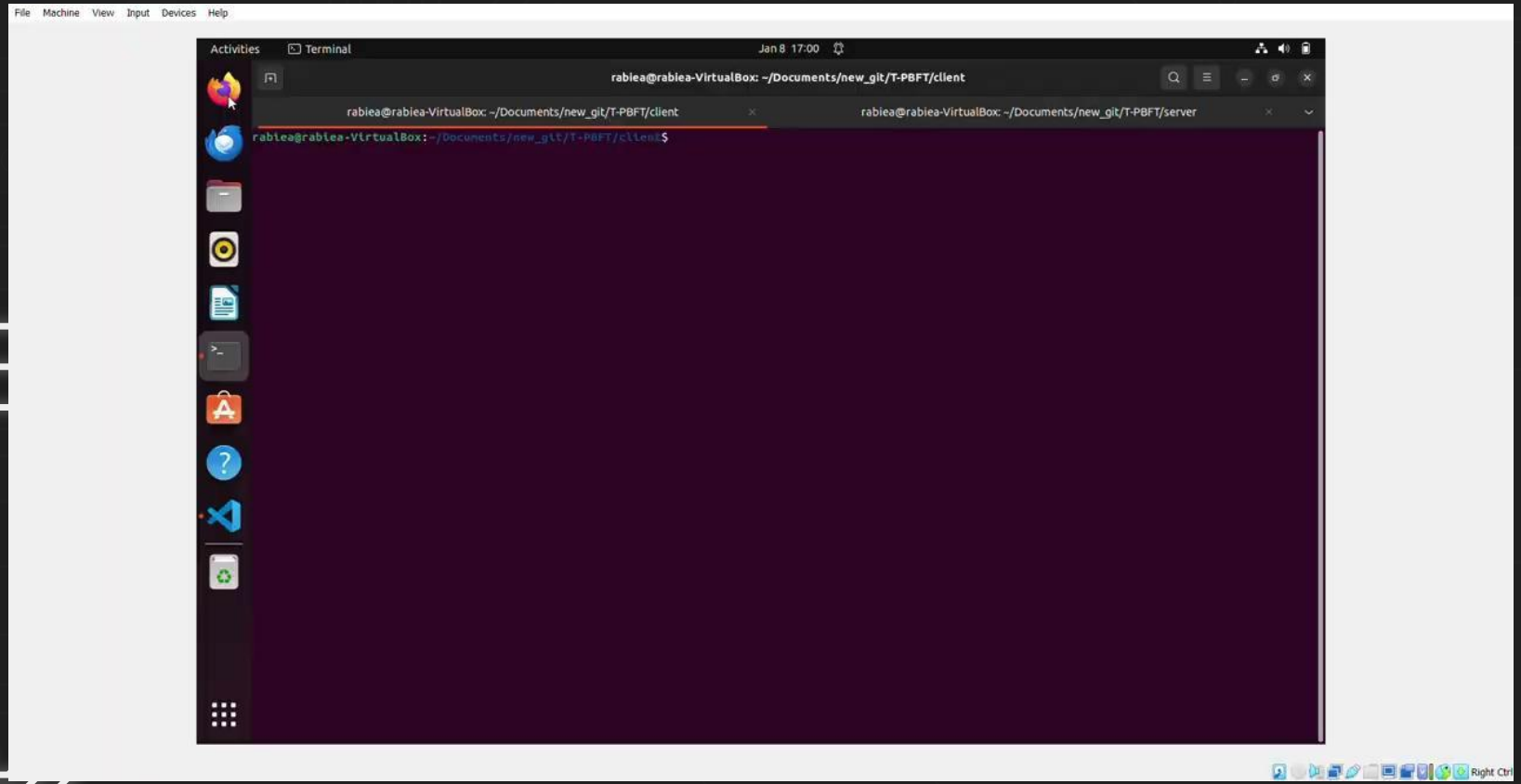
```
void Node::ack_handler(MsgAck &&msg, const Net::conn_t &conn) {
    cout << "Node " << get_id() << " got ACK:\n";
    cout << "\tFrom " << conn->get_peer_id().to_hex() << "\n";
        // Update Direct Trust Value
        for (size_t i = 0; i < NUM_NODES; i++) {
            salticidae::PeerId localPeer = allNodes[i];
            TrustManager::updateDirectTrust(localPeer);
        }
        //Update Global Trust Value
        for (size_t i = 0; i < NUM_NODES; i++) {
            salticidae::PeerId node = allNodes[i];
            TrustManager::updateGlobalTrust(node);
        }

        TrustManager::printGlobalTrustMap() ;
        TrustManager::getConsensusGroup(D2) ;
        TrustManager::printConsensusGroup() ;
        TrustManager::getPrimaryGroup(M2) ;
        TrustManager::printPrimaryGroup() ;
        RSAKeyGenerator::changeGroupKey();
        RSAKeyGenerator::clearPreparedMessages();
        RSAKeyGenerator::randomValueGenerator();
    requestQueue.pop_front();
    client_request = "";
    busy = false;
    bool check = requestQueue.empty();
    peerNet->multicast_msg(MsgNextRequest(true), peers);
    if(!check && !busy) {
        std::pair<salticidae::PeerId, std::string> nextRequest(requestQueue.front());
        salticidae::PeerId client_conn_id = nextRequest.first;
        std::string client_conn_request = nextRequest.second;
        Node::process_group_request(client_conn_request, client_conn_id);
        busy=true;
    }
    //num_acks += 1;
}
```

OPERATION PROCESS

- GROUP REQUEST HANDLER
- PROCESS GROUP REQUEST
- PRIMARY CONSENSUS HANDLER
- PRIMARY VERIFIED HANDLER
- PREPREPARE HANDLER
- PREPARE HANDLER
- SEND REPLY HANDLER
- ACK HANDLER
- **PROCESS NEXT REQUEST**

```
void Node::process_next_request(const MsgNextRequest &&msg, const Net::conn_t &conn) {
    requestQueue.pop_front();
    busy = false;
    client_request = " ";
    bool check = requestQueue.empty();
    if(check == false && busy == false) {
        std::pair<salticidae::PeerId, std::string> nextRequest(requestQueue.front());
        salticidae::PeerId client_conn_id = nextRequest.first;
        std::string client_conn_request = nextRequest.second;
        Node::process_group_request(client_conn_request, client_conn_id);
        busy=true;
    }
}
```

DEMO - SINGLE MESSAGE



DEMO - MULTIPLE MESSAGES

04

CONCLUSION

CONCLUSIÓN AND OUR ANALYSIS

- The paper was largely vague and did not provide any implementational details about many phases such as
  - Group process
  - Transactions and blocks
  - Client request send to primary or all
  - Triggering of the view change
  - Number of prepared messages that should be received
- The evaluation of the paper is done only theoretically, so we did not have any practical benchmarks to compare our implementation with.

FRAMEWORK PROBLEMS

- The problem with this protocol is that the nodes are not totally async. Like if node 1 is waiting for a message and we apply a while(something doesn't get true) it will not move on from this and other nodes won't do anything or won't receive messages during this phase.
- Node/Server should have used threads.
- There is no way we can access Nodes/Server object outside handler file. There is a struct function returning pointer NodeVector *get_nodes() but it was not working as it should. we tried to call this in main.cpp file because we have handler object there and tried to get access to nodes object but it was not working.
- If you add some structures in node.h file, handler.cpp file will start giving error when creating node object. For example std::queue, std::chrono.

THANK YOU!