Final Report - Implementation of RingBFT

Zaoyi Zheng
University of California, Davis
royzheng@ucdavis.edu

Xiaoxi Yu
University of California, Davis
xxiyu@ucdavis.edu

Fuming Fu
University of California, Davis
fufu@ucdavis.edu

Jiangnan Chen
University of California, Davis
jnchen@ucdavis.edu

Haochen Yang
University of California, Davis
yhcyang@ucdavis.edu

Weijia Wang
University of California, Davis
wjawang@ucdavis.edu

ABSTRACT

Federated data management has been widely used in various scientific data management applications[10]. Due to the increasing number of byzantine attacks[1], Byzantine Fault-Tolerant (BFT) consensus protocols have become essential in distributed databases and related fields such as blockchain. When client transactions require access to a single-shard, the traditional BFT protocols like PBFT[3], are efficient on sharded-replicated blockchains. However, in many situations, cross-shard transactions are required[4, 9], and when client transactions require access to a cross-shard, the performance will face degradation.

This paper[11] presents a novel meta-BFT protocol for sharded blockchains, RingBFT, which can significantly reduce the costs with cross-shard transactions while supporting large scale shards, improving throughput performance, and guaranteeing safety and liveliness. Our goal for the final project is to implement and evaluate the proposed protocol, and compare the results and performance demonstrated in the paper on the ResilientDB platform[5].

1 INTRODUCTION

1.1 Background of RingBFT

In the late 90s, Practical Byzantine Fault Tolerance(PBFT) consensus was introduced by [2], which is the first protocol that survives Byzantine faults in asynchronous network. Based on PBFT consensus, the original Byzantine gathering problem is greatly improved from exponential to polynomial time complexity. After PBFT consensus is built, a lot of advanced protocols are gradually presented. For instance, GeoBFT, a geo-scale consensus protocol, is presented to deal with geo-scale deployments in which many replicas spread across a geographically large area participate in consensus [6]. At its core, it adopts PBFT consensus in each cluster to help replicas reach an agreement at local level as shown in the following figure[1][7].

Similarly, PBFT also plays a pivotal role in RingBFT [12], whose local consensus is also built on PBFT. With respect to RingBFT, it aims to tackle issues in federated database management. As the recent surge in federated data-management applications, it has brought forth concerns about the security of underlying data and the consistency of replicas in the presence of malicious attacks. This leads to the rise of sharded replicated blockchains. However, the existing BFT protocols for these sharded blockchains are efficient if client transactions require access to a single-shard, but face performance degradation if there is a cross-shard transaction that requires access to multiple shards.

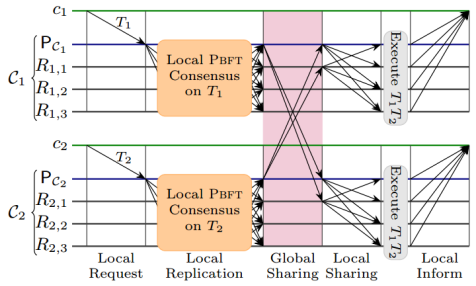

Figure 1: GeoBFT consensus protocol schematic [7]

To mitigate this issue, a strategy is to employ the sharded-replicated paradigm. In a sharded-replicated database, the data is distributed across a set of shards where each shard manages a unique partition of the data that may be much less than the whole data. Furthermore, each shard replicates its partition of data at local to ensure availability under failures. If each transaction accesses only one shard, these sharded systems can fetch high throughput as consensus is restricted to a subset of replicas. Nevertheless, a normal transaction is more than a single-shard transaction. Cross-shard transactions are more common in practice and is of great importance to be considered. To handle cross-shard transaction efficently, [12] proposed RingBFT that significantly reduces the costs associated with cross-shard transaction. RingBFT guarantees consensus for each cross-shard transaction in at most two rotations around the ring. In RingBFT, each shard may participate in concurrent rotational flows, where each rotation maps to the processing of a transaction. For each cross-shard transaction, RingBFT follows the principle of process, forward, and re-transmit. This implies that each shard performs consensus on the transaction and forwards it to the next shard. This flow continues until each shard is aware of the fate of the transaction. However, the real challenge with cross-shard transactions is managing conflicts and preventing deadlocks, which RingBFT achieves by requiring cross-shard transactions to travel in ring order. Despite all of this, RingBFT ensures that communications between the shards are linear to the number of shards. This minimalistic thought has allowed RingBFT to achieve unprecedented gains in throughput and has allowed us to scale BFT protocols to nearly 500 nodes.

In addition, RingBFT also resembles GeoBFT to some degree. It is clear that replicas will all be grouped either in GeoBFT or RingBFT.

[1]Modified from ResilientDB: Global Scale Resilient Blockchain Fabric

And the only difference is the group of replicas in GeoBFT is termed as clusters while it is called shards in RingBFT. Meanwhile, GeoBFT is designed for excellent scalability to tackle geographically large area participate in consensus. Hence, it inspires us to develop RingBFT from GeoBFT after taking their common ground into account. As shown below, To be specific, RingBFT can be regarded as a advanced version of GeoBFT. First, It requires that a transaction is travelled in ring order. Second, it introduces lock mechanism to handle conflicts and prevent deadlocks. Finally, a linear communication beweeen shards is achieved in RingBFT. Therefore, these improvements gives us a hint to implement RingBFT protocol.

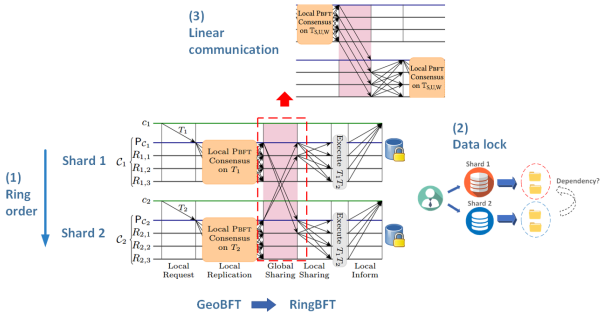

Figure 2: Similarity between GeoBFT and RingBFT protocol(Modified from ResilientDB: Global Scale Resilient Blockchain Fabric [7])

1.2 Market Opportunity of RingBFT

As stepping into the era of information, the rapid increase in the volume of data opens up new possibilities for enterprises. Traditionally, a user can only index a single data source at a time. But federated database, which could be implemented with RingBFT protocol, improves the likelihood of users finding what they need on the first attempt by including more data sources.

In the field of business, the ability to increase customer engagement and satisfaction rates is regarded as the ace in the hole, which has been an impediment for a company to progress. However, this can be easily resolved by employing a federated database that provides the most relevant information in an intuitive format and even, at times, surfacing content a user did not even know. Instead of relying on a traditional single database, visitors can reach their desired destination with fewer clicks through the federated database. As a result, users stay on your site longer, rather than leaving due to dissatisfaction. This ultimately increases the chances of converting your visitor into a customer and increases customer stickiness. In this regard, this exercise would be much more helpful especially for e-commerce businesses since they face more competitions with their counterparts in terms of customer share.

However, the security and privacy concerns are major obstacles for a federated database due to the fact that several parties maintain a common database in a federated database. The leakage of private data can lead to serious issues beyond the financial loss of providers[8]. Thus, to deal with this issue, RingBFT is proposed, a meta-BFT protocol for sharded blockchains, to necessitate all those involved parties to reach a consensus on each transaction. As a result, it will eventually maximize the benefit of federated database usage by utilizing globally-distributed nodes to guard data's security and increasing customer stickiness with improved throughput by the protocol.

2 ACTIVITIES

2.1 Client

Clients are the generators of transaction requests. There are many existing protocols that aim to deal with the single-shard transaction. However, when clients require access to multiple shards, existing BFT protocols are facing performance degradation. The Ring-BFT protocol aims to reach consensus more efficiently when facing cross-shard transactions.

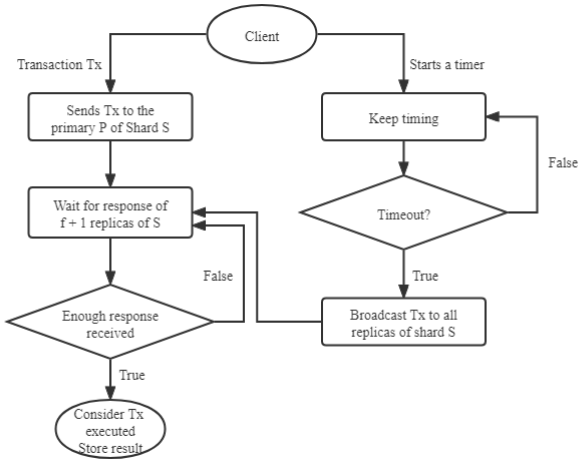

Figure 3: Workflow of Client

In the cross-shard scenario, the client has the following activities:

(1) When a client c wants to process a cross-shard transaction $T_{\mathfrak{J}}$, it creates a $<T_{\mathfrak{J}}>_C$ message and sends it to the primary of the first shard S in ring order. As part of this transaction, client c specifies the information regarding all the involved shards, such as their identifiers and the necessary read-write sets of each shard.
(2) After client c sends out the request of transaction, it awaits receipt of response messages from $f + 1$ replicas of shard S.
(3) When client c receives enough identical responses, it considers $<T_{\mathfrak{J}}>_C$ executed, with result r, as the k-th transaction.
(4) The client cannot wait indefinitely to receive valid responses, so each client c will start a timer when it sends its transaction to the primary of shard S. If the timer timeouts prior to c receiving at least $f + 1$ identical responses, it broadcasts $<T_{\mathfrak{J}}>_C$ to all the replicas of shard S.

Due to faulty primary or unreliable networks, such transactions may be lost in the middle. Instead of waiting indefinitely, the client will start a timer when it sends the transactions to primary in the shard. If the timer reaches timeout before the client receives $f + 1$ identical reply message, the client will re-transmit the transactions to the primary.

2.2 Primary

Primary is a special kind of Replicas. Primary in the RingBFT is decentralized in that it is not responsible for coordinating all consensus decisions, as such a centralized design limits the throughput to the outgoing global bandwidth and latency of this single replica or cluster[7].

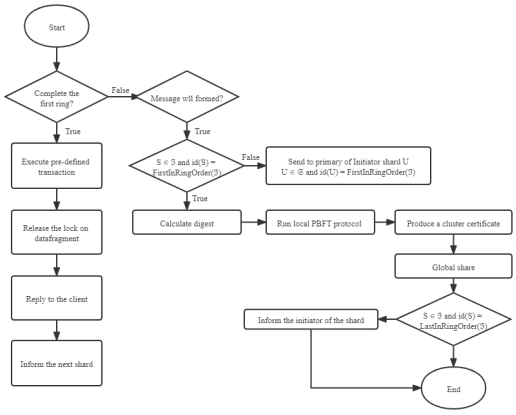

Figure 4: Workflow of Primary

Generally speaking, primary has the following activities:

(1) The primary will check if the message is well-formed at the reception of the client request and confirm it is the initiator of the shards or forwards it to the initiator.
(2) The sequence number will be assigned to the transaction, the digest will be calculated to reduce the cost of communication and the primary replicates the transaction to all local replicas using the PBFT.
(3) At the end of local replication, the primary can produce a cluster certificate for the transaction. These are shared with other clusters via local communication. However, the transaction cannot be executed unless this transaction is simple.
(4) If it has completed one complete ring order, then the last shard of transaction involved shards in ring order will inform the initiator to execute the transaction, release the lock of data and reply to the client. The primary will communicate with local replicas to ensure the transaction is executed in every replica.

In the implementation, the primary's task is to open the port and listen to the message from clients, replicas or the primary from other shards, choosing one of the activities mentioned above.

2.3 Replica

With respect to replicas, there are two scenarios are considered:(1) Single-shard consensus (2) Cross-shard consensus. In addition, we'll talk about the encivil executions in the end.

2.3.1 Single-shard consensus.

(1) **Pre-prepare** After receiving the Pre-prepare message from its primary p, a replica R will first check if the message is well-formed. The Pre-prepare message includes: (1) sequence number k that specifies the order for this transaction, and (2) digest $\triangle = H(\langle T \rangle c)$ of the client transaction which will be used in future communication to reduce data communicated across the network.
(2) **Prepare** In Prepare phase, a replica R will send Prepare($\triangle$, k) to all the replicas of S.
(3) **Commit** When R receives identical Prepare messages (and are also well-formed) from at least nf replicas of S, it achieves a weak guarantee that the majority of non-faulty replicas have also agreed to support p's order for m. Hence, it marks this request as prepared, creates a Commit($\triangle$, k) message, and broadcasts this message.
(4) **Reply** When R receives identical Commit messages (and are also well-formed) from at least nf replicas of S, it achieves a strong guarantee that the majority of non-faulty replicas have also prepared this request. Hence, it executes transaction T_T, after $(k-1)^{th}$ all the transactions have been executed and replies to the client c.

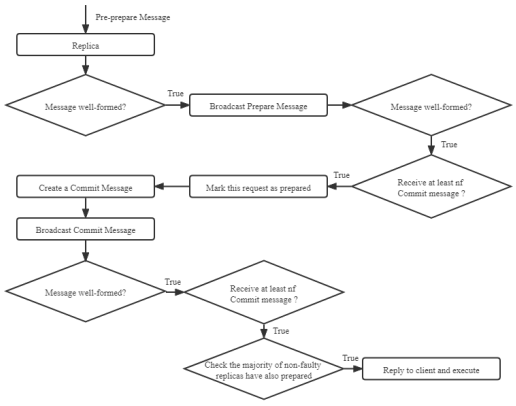

Figure 5: Single-shard Consensus

2.3.2 Cross-shard consensus.

(1) **Pre-prepare** When a replica R ∈ S receives the Pre-prepare message from P_S, it checks if the request is well-formed. If this is the case and if R has not agreed to support any other request from P_S as the k^{th} request, then it broadcasts a Prepare message in its shard S
(2) **Prepare** When a replica R receives identical Prepare messages from nf distinct replicas, it gets an assurance that a majority of non-faulty replicas are supporting this request. At this point, each replica r broadcasts a Commit message to all the replicas in S.Once a transaction passes this phase, the replica R marks it prepared.
(3) **Commit and Data locking** When a replica R receives well-formed identical Commit messages from nf distinct replicas in S, it checks if it also prepared this transaction at the same sequence number. If this is the case, RingBFT requires each replica r to lock all the read-write sets that transaction T needs to access in shard S.

(4) **Foward to next shard** Once a replica R in S locks the data corresponding to CST T, it sends a Forward message to only one replica q of the next shard in ring order.

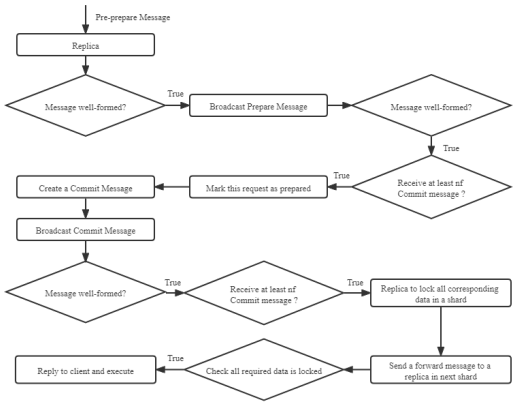

Figure 6: Cross-shard Consensus

2.4 Uncivil Executions

In order to achieve liveness through periods of synchrony, RingBFT adds 3 timers(local timer, remote timer, and transmit timer) in each replica.

(1) If replica does not receive nf identical Commit messages from distinct replicas, or primary fails to propose a request from the client, both situations will cause local timer timeouts and will invoke local view change protocol.
(2) If malicious primary causes in-the-dark attack, RingBFT will ensure the progress of such replicas with the help of *Checkpoint* message.
(3) If there's no network communication between shards, it will trigger the timeout of transmit timers in the replicas, which will cause the retransmission of *Forward* messages.
(4) If only partial network works or primary in the previous shard is byzantine, it will trigger timeout of remote timers in replicas. Affected replicas will send *ViewChange* message to other replicas in the same shard.

3 IMPLEMENTATION ON RESILIENT DB

RingBFT aims to scale permissioned blockchains to hundreds of replicas through efficient sharding. To argue the benefits of our RingBFT protocol, we need to first implement it in a permissioned blockchain fabric. For this purpose, we employed a state-of-the-art permissioned blockchain fabric, ResilientDB. ResilientDB offers an optimal system-centric design that eases implementing novel BFT consensus protocols. ResilientDB presents an architecture that allows even classical protocols like PBFT to achieve high throughputs and low latencies.

NetworkLayer : ResilientDB provides a network layer to manage communication among clients and replicas. The network layer provides TCP/IP capabilities through Nanomsg-NG to communicate messages. To facilitate uninterrupted processing of millions of messages, at each replica, ResilientDB offers multiple input and output threads to communicate with the network.

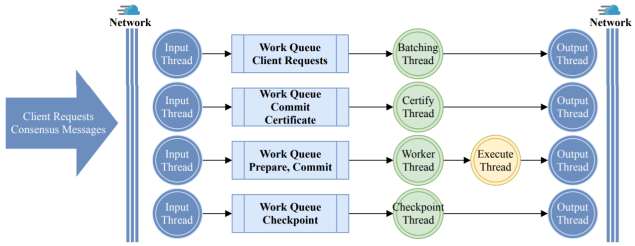

Figure 7: The parallel-pipelined architecture provided by ResilientDB fabric for efficiently implementing RingBFT[7].

PipelinedConsensus : Once a message is received from the network, the key challenge is to process it efficiently. If all the ensuing consensus tasks are performed sequentially, the resulting system output would be abysmally low. Moreover, such a system would be unable to utilize the available computational and network capabilities. Hence, ResilientDB associates with each replica a parallel pipelined architecture, which we illustrate in Figure 7.

Blockchain : To securely record each successfully replicated transaction, we also implement an immutable ledger–blockchain. For systems running fully-replicated BFT consensus protocols like PBFT and Zyzzyva, blockchain is maintained as a single linked-list of all transactions where each replica stores a copy of the blockchain. However, in the case of sharding protocols like RingBFT, each shard maintains its own blockchain. As a result, no single shard can provide a complete state of all the transactions. Hence, we refer to the ledger maintained at each shard as a partial-blockchain.

3.1 Redesign

We have added two classes inherited from the Message class:

- *RingBFTForwardMessage*
- *RingBFTExecuteMessage*

RingBFTForwardMessage includes variables such as forwardOrder that indicates the indexes of shards in forwarding order, executionOrder that indicates the indexes of shards in execution order. It also has a method get_next_node_id(), which will return the id of the next node in forwarding order.

```
class RingBFTForwardMessage : public Message
{
public :
    ...
    deque<uint64_t> executeOrder;
    deque<uint64_t> forwardOrder;
    // node id of next shard to forward
    uint64_t get_next_node_id(deque<uint64_t> ringOrder,
    deque<uint64_t> executeOrder);
}
```

```
// get next node id in forwarding order
uint64_t RingBFTForwardMessage :: get_next_node_id(
```

```
deque<uint64_t> ringOrder, deque<uint64_t> executeOrder){
    uint64_t next_shard = ringOrder.front();
    ringOrder.pop_front();
    if (ringOrder.empty()){
        executeOrder.push_front(next_shard);
    } else {
        executeOrder.push_back(next_shard);
    }
    return next_shard;
}
```

We also defined a RingBFTExecuteMessage class, one of its variables is executeOrder which indicates the indexes of shards in execution order, the method get_next_node_id() will return the node id of next shard to forward in execution order.

```
class RingBFTExecuteMessage : public Message
{
public:
    ...
    RingBFTExecuteMessage(deque<uint64_t> eo){
        this.executeOrder = eo;
    };

    // indexes of shards in execution order
    deque<uint64_t> executeOrder;

    // node id of next shard to forward
    uint64_t get_next_node_id(deque<uint64_t> executeOrder);
}
```

```
// get next node id in execution order
uint64_t RingBFTExecuteMessage::get_next_node_id
    (deque<uint64_t> executeOrder){
    uint64_t next_executeion_node_id = executeOrder.front();
    executeOrder.pop_front();
    return next_executeion_node_id;
}
```

The transaction is redesigned that it needs to be put in SpinLockMap so as to enable every node of the shard to lock and unlock data fragments. Client threads are supposed to sparse transactions into specific operations which is stored in SpinLockMap, where every possible operation is stored as an entry in the map. In a multi-thread environment, one entry of the map could be executed by only one worker thread and it would be released until that working thread finishes the execution.

The worker thread is redesigned that RingBFT requires nodes to perform different operations at the first and last rounds. Receive the message from the work queue, a node performs different tasks relative to the message type. The fraction of code is shown below:

```
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 RBFT_LAST_ROUND_MSG:
        send_last_round_execute_msg(msg);
    case EXECUTE_MSG:
        rc = process_execute_msg(msg);
        break;
     ...
}
```

In the first round execution, local nodes will perform PBFT without executing the transaction to make consensus, lock the corresponding data fragment, check if it is the last shard of involved shards and decide whether to perform the Last Round Execution or not.

In the last Round Execution, local nodes will perform the transaction, unlock the corresponding data fragment and reply to the client. The fraction of code is shown below:

```
RingBFTExecuteMessage rbft_msg
= (RingBFTExecuteMessage *)msg;
    if (msg->rtype != RBFT_LAST_ROUND_MSG)
    {
        // lock corresponding datafragment
        dependentSRC[rbft_msg->executeOrder.front()].lock();
        // If the last shard in transaction - involved shards
        if (rbft_msg->executeOrder.size() == 1)
        {
            //The id of initiator is set to 0
            vector<uint64_t> dest;
            dest.push_back((uint64_t )0);
            lastMsg = Message::
            create_message(RBFT_LAST_ROUND_MSG);
            msg_queue.enqueue(get_thd_id(), lastMsg, dest);
            dest.clear();
        }
    }
    else
    {
        // execute
        tman->run_txn_print(msg);
        // reply to client
        Message *rsp = Message::create_message(CL_RSP);
        ClientResponseMessage *crsp
        = (ClientResponseMessage *)rsp;
        crsp->init();
        crsp->copy_from_txn(txn_man);
        vector<uint64_t> dest;
```

```
        dest.push_back(txn_man->client_id);
        msg_queue.enqueue(get_thd_id(), crsp, dest);
        dest.clear();
        // unlock corresponding datafragment
        dependentSRC[rbft_msg->executeOrder.front()]
        .unlock();
    }
```

3.2 Global Sharing

The main difference between RingBFT and GeoBFT In GeoBFT, the primary sends messages to replicas of other shards, and it sends f+1 messages, while in RingBFT, both replica and primary send one message to the next shard with the same id. This gives the basic idea to modify the RingBFT based on the GeoBFT, and here are the main steps in global sharing:

(1) For cross-shard transactions, the number of nodes should be more than that in a cluster. Otherwise, it is a single-shard transaction instead of a cross-shard transaction. So we need to check if it is a cross-shard transaction before the following steps.
(2) Create a message and coerce it.
(3) Register in transaction manager.
(4) Traverse all nodes and for every node check for these three conditions:
  (a) Not in the same cluster.
  (b) Have the same id.
  (c) In adjacent shards, which means in ring order.
(5) If the node satisfies all the three conditions above, add it to the message queue then clear the destination vector.

```
#if RING
  // Forward message(Global sharing in RingBFT)
  // Not for Single shard
  if (g_node_cnt > ringbft_cluster_size)

  {
   RingBFTForwardMessage *rbm =
   (RingBFTForwardMessage *)
   Message::create_message(txn_man, RINGBFT_MSG);
   rbm->txn_id = txn_man->get_txn_id();

   vector<uint64_t> dest;
   for (uint64_t i = 0; i < g_node_cnt; i++)
   {
    // Not in the same shard and having the same id
    if (! is_in_same_cluster(g_node_id, i) &&
    i % ringbft_cluster_size == g_node_id %
     ringbft_cluster_size)
    {
     dest.push_back(i);
     break;
    }
   }

   msg_queue.enqueue(get_thd_id(), rbm, dest);
   dest.clear();
  }
#endif
```

3.3 Execution

To visualize the process of RingBFT, printouts are used to identify every step of the protocol:

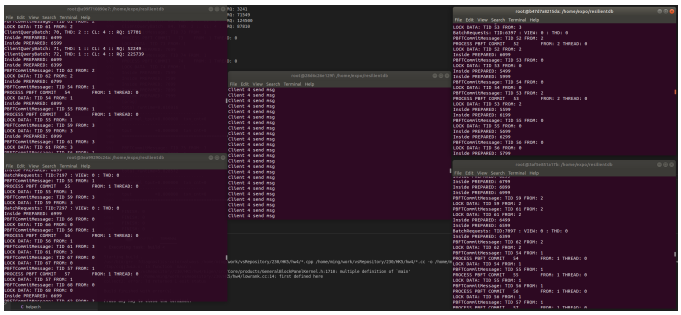

Figure 8: RBFT Printouts

4 IMPLEMENTATION IN GO

Because of the difficulty of running and debugging code on Resilient DB, we try to implement the demo in Go.

4.1 PBFT in Go

PBFT is greatly used in nowadays real applications. Just for simplicity in our implementation of RingBFT, we choose a PBFT project[2] in Github and continue writing RingBFT code above it.

Here are some implementation details for PBFT:

(1) **Communication:** Messages transmission is achieved by *http* server with the help of *net/http* library in Go. In each stage of PBFT, each node only needs to send a *http* request to the target node.
(2) **Node:** Each node maintains its own *MsgEntrance* channel to handle the incoming messages from other nodes. After decoding the http request and encoded messages, nodes get messages from the *MsgDelivery* channel and perform different tasks based on the type of the message.
(3) **State:** Each node has its own current state to record the current stage and view id, to remember the last sequence ID and to log different messages. All jobs are implemented as a method of a state instance.

4.2 Implementation Details

4.2.1 Ring Order. Ring order is hard coded in the http request message. A representative request message is in the following:

```
    {
     "clientID": "ahnhwi",
     "operation": "GetMyName",
     "timestamp":859381532,
     "RingOrder":[1, 2]
    }
```

[2]https://github.com/bigpicturelabs/simple_pbft

In this example, the global sharing messages are sent from shard 1 to shard 2.

4.2.2 Global Sharing. Finishing global sharing stage in linear time is a great advantage for RingBFT. When try to send global sharing messages, nodes should check 3 conditions:

(1) Sender and receiver have the same ID
(2) Sender and receiver not in the same shard

The ID, in our implementation, is the node index in its shard.

Here's the implementation for that:

```
...
// Get the shard num of the source
re := regexp.MustCompile(`[0-9]+`)
shardNumFrom, shardNumTo := int(re.Find([]byte(node.NodeID))[0])
    -48, 0

// Get the shard num of the target
for index, ringOrder := range msg.ReqMsg.RingOrder {
        if ringOrder == shardNumFrom {
                if index != len(msg.ReqMsg.RingOrder)-1 {
                        shardNumTo = msg.ReqMsg.RingOrder[(index
                            +1)%len(msg.ReqMsg.RingOrder)]
                } else {
                        node.CurrentState.CurrentStage = consensus
                            .Executed
                        shardNumTo = msg.ReqMsg.RingOrder[(index
                            +1)%len(msg.ReqMsg.RingOrder)]
                }
        }
}
...
```

All nodes ID are pre-defined before the execution and follow some rules to distinguish itself from the others. Thus, in order to find the target node in global sharing stage, regular expression is applied to each node ID.

4.2.3 Local Sharing. Local sharing stage is much similar to preprepare stage in PBFT. Instead of sending messages from only the primary, all nodes should forward the global sharing messages to other nodes in the same shard. However, only the primary needs to respond it if it receives sufficient forward messages.

Here's a snippet of the code:

For primary:

```
node.CurrentState.MsgLogs.GlobalForwardMsgs[msg.CommitMsg.
    NodeID] = msg
        if len(node.CurrentState.MsgLogs.GlobalForwardMsgs) >= 2*f
            {
                node.CurrentState.CurrentStage = consensus.Idle
                prePrepareMsg, err := node.CurrentState.
                        StartConsensus(msg.ReqMsg)
                if err != nil {
                        return err
                }

                node.CurrentState.MsgLogs.ReqMsg = msg.ReqMsg
                node.Broadcast(prePrepareMsg, "/preprepare")
                LogStage("Pre-prepare", true)
                return nil
        }
```

4.2.4 Automated Testing. In order to test the RingBFT, our team also writes a automated testing script on speed up our testing process. With the help of Tmux, bash script can run each node in a seperate Tmux window without manually configuring.

4.3 Results

Here's the result for our project:

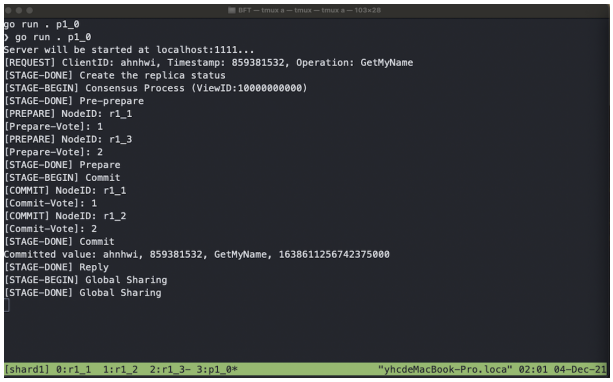

Figure 9: Result of shard 1

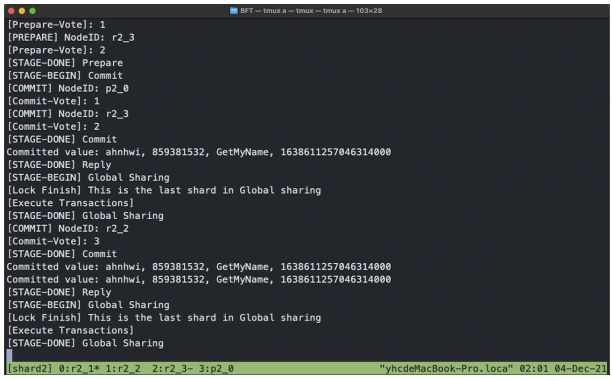

Figure 10: Result of shard 2

4.4 Further work

Due to the limited time budget, here list some parts that not implemented in our project.

(1) **Digital Signature:** For simplicity, our team use common hash function instead of the digital signature in global sharing.
(2) **Unstable execution:** There are some conflicts between goroutines(message dispatcher and message delivery). In some cases, the consensus will be stuck in a stage without a response.
(3) **Client implementation:** For now, we only imitate the behaviour of client by sending HTTP request through curl. Next, we will try to add client nodes in our system.

REFERENCES

[1] Y. Amir, B. Coan, J. Kirsch, and J. Lane. 2008. Byzantine replication under attack. (2008), 197–206.

[2] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (New Orleans, Louisiana, USA) *(OSDI '99)*. USENIX Association, USA, 173–186.

[3] Miguel Castro, Barbara Liskov, et al. 1999. Practical byzantine fault tolerance. In *OSDI*, Vol. 99. 173–186.

[4] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–22.

[5] Suyash Gupta, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. 2020. Resilientdb: Global scale resilient blockchain fabric. *arXiv preprint arXiv:2002.00160* (2020).

[6] Suyash Gupta, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. 2020. ResilientDB: Global Scale Resilient Blockchain Fabric. *Proc. VLDB Endow.* 13, 6 (feb 2020), 868–883. https://doi.org/10.14778/3380750.3380757

[7] Suyash Gupta, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. 2020. ResilientDB: Global Scale Resilient Blockchain Fabric. 13, 6 (2020), 868–883.

[8] Yunlong Lu, Xiaohong Huang, Yueyue Dai, Sabita Maharjan, and Yan Zhang. 2019. Blockchain and federated learning for privacy-preserved data sharing in industrial IoT. *IEEE Transactions on Industrial Informatics* 16, 6 (2019), 4177–4186.

[9] Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. 2016. The challenges of global-scale data management. In *Proceedings of the 2016 International Conference on Management of Data*. 2223–2227.

[10] Peng Peng and Lei Zou. 2019. Survey on Federated RDF Systems. *Frontiers of Data and Domputing* 1, 1 (2019), 73–81.

[11] Sajjad Rahnama, Suyash Gupta, Rohan Sogani, Dhruv Krishnan, and Mohammad Sadoghi. 2021. RingBFT: Resilient Consensus over Sharded Ring Topology. *arXiv preprint arXiv:2107.13047* (2021).

[12] Sajjad Rahnama, Suyash Gupta, Rohan Sogani, Dhruv Krishnan, and Mohammad Sadoghi. 2021. RingBFT: Resilient Consensus over Sharded Ring Topology. *ArXiv* abs/2107.13047 (2021).