Efficient and Dynamic Byzantine Fault-tolerant Protocol

Chengyun Zhang¹,a, Pei Lin²,b

¹college of communication and information system Communication University of China Beijing, China
²Jilin University Jilin, China
a1247941864@qq.com, b18770709@qq.com

Abstract: With the development of blockchain, distributed computing and other technologies, the problem of Byzantine fault tolerance has been paid more and more attention. After Lamport raised this problem and gave the corresponding solution, more and more people optimized the solution to the problem in terms of algorithm efficiency and security. After studying the previous algorithm, we used the idea of "predictive execution" and "resource minimization" to convert the original three-phase protocol into two phases and under normal circumstances only 2f+1 nodes can be used to complete the work. Greatly improved the efficiency of the algorithm. At the same time, we consider the total order problem and propose a system to execute the request made by the client in chronological order to prevent the correct client request from being stolen by the enemy. Finally, we consider that the previous BFT protocol did not fully consider the problem of external nodes dynamically joining the system, and optimized the previous solution to enable external nodes to join the BFT system safely and quickly.

1. INTRODUCTION
With the development of blockchain technology and the optimization of distributed technologies, we are paying more and more attention to technologies that have reached consensus on the error of some nodes. Lamport et al. proposed a Byzantine problem and formed a preliminary solution for the formation of a consensus in the presence of wrong nodes, and in 1998 proposed a BFT system that can be implemented - PBFT [2]. Then people continue to optimize on this basis to improve efficiency. The paper [1] [3] [4] [5] [6] [7] transfers some important steps to the client to complete the loss of communication between nodes, The paper [8] proposes a "predictive execution" idea, combining the consensus node and the execution node to convert the original PBFT three-item protocol into two protocols, which greatly reduces the loss of communication between nodes. The paper [9] considers that it does not fully consider the client's error on the basis of [8], which will lead to a sharp increase in system traffic, making the consensus efficiency much lower than the previous PBFT. Therefore, the fault tolerance mechanism of the client is optimized, and the whole system is more efficient. The paper [10] proposes a resource optimization scheme, considering that the previous BFT scheme uses 3f+1 nodes for each request to tolerate f node errors. We can use the completely correct 2f+1 nodes to complete the request. Once there is a node exception, it will enter the original consensus protocol to achieve the tolerance of the wrong node. The paper [10] proposed two sets of BFT systems combined with resource optimization based on the traditional BFT scheme after proposing resource optimization ideas. But these systems use the client to perform some important steps, but do not fully consider the client's error.
At the same time, we also noticed that the software errors of the server in the BFT system and some malicious attacks will cause the server to behave confusingly[13]. However, in the previous BFT protocol, the server that dynamically removed these errors and added to the new server was not fully considered [12]. Therefore, we propose the mobility of the BFT system, that is, the dynamic joining sub-protocol of the node, to ensure that the node can dynamically join the system without restarting the system.

The Total order question [11] is a new issue in the BFT consensus program. The client sends a request to enter the node for broadcast. Due to the broadcast delay factor, some nodes preferentially get the relevant information of the client. If the node is invaded by the adversary, it will resend the content of interest with the compromised client. If this hacked request is executed first by the node, the correct request will be revoked. This will cause damage to the correct client.

In order to optimize the above problems, in our paper, a BFT scheme based on predictive execution and resource optimization ideas is proposed, and the dynamic joining of nodes and the consideration of the total order problem are realized, which makes our scheme achieve the consensus speed Faster, better node dynamics, and better protection for the right client. There are the following innovations in our program:

1.1. Faster consensus speed
We use the idea of predictive execution and resource optimization to convert the traditional three-item BFT protocol into two protocols and consider the resource optimization. We use 2f+1 node areas to implement the consensus of inter-node requests.

1.2. total order protection
In order to protect the request sent by the correct client, we need to protect the total order between nodes. We refer to the global serial number and the local serial number method to globally and locally sort the requests sent by different clients so that the client's request is ordered Execution. no old request will occur before the new request is executed

1.3. dynamically join
We propose a dynamic join sub-protocol for the system node to enable it to join the node without rebooting the system.

2. BACKGROUND
Our agreement to improve the hBFT protocol to join the concept of dynamic join, total order and resource minimization makes it more secure and efficient. As shown in Figure 1, hBFT uses the idea of predictive execution to convert the previous three-phase protocol into a two-phase protocol, improving consensus efficiency. It transfers some of the operations that were previously done in the node to the client for execution, and detects the client's error by communicating between the nodes. And use PBFT-like three-phase checkpoint protocol to ensure that the node can recover stable state after the error occurs and deal with the inconsistency between nodes.

Fault-free cases and normal cases [13] are two states in which the BFT system operates. Among them, Fault-free cases indicate that all nodes can work normally, and join and client in our system can work normally. Normal-cases indicates that the primary node is incorrect and the number of backup node errors does not exceed the set threshold.

3. THE PROTOCOL
In the traditional PBFT-based protocol, we need 3f+1 nodes to prevent f node errors. In our model, we only consider the consensus under "Fault-free cases", not considering the wrong nodes. In this case we only need 2f+1 nodes to get our consensus. Below we detail the process of the consensus algorithm.Before calling the consensus node, we first need to send each node their public and private
keys. This operation is done by a trusted server through the key exchange protocol. After that, each node gets its own public key PKj and its own private key SKj.

The client sends a message \( m = \{ \text{Request}, o, t, c \} | c \) to all nodes by calling the application of the consensus mechanism, where \( o \) represents the required operation of the client request, and \( t \) represents the timestamp sent by the request, \( c \) represents the ID of the client. After the client generates this message, it needs to be digitally signed by the public key of the client. We describe our consensus algorithm through the behavior of the primary node, backup node, and client. We can show in Figure 1.

![Figure 1. EdBFT](image)

3.1. Mater node

After receiving the request from the client, the master node Pj first arranges a sequence number seq for the message and then sends a \( \{ \text{Prepare}, v, \text{seq}, D(m), m, c \} | Pj \) to all backup nodes. Where \( v \) represents the current view number, \( D(m) \) represents the digest of the message, and the message is signed with its own secret key. Below we describe the behavior of the master node in pseudo-code:

1. initialization
2. A  \{ All replicas \}
3. seq = 0  \{ serial number \}
4. When the master node receives \( \{ \text{Request}, o, t, c \} | c \)
5. seq = seqi+1  \{ seqi is last seq \}
6. send \( \{ \text{Prepare}, v, \text{seq}, D(m), m, c \} | Pj \) to all backup nodes
7. send \( \{ \text{Reply}, v, t, \text{seq}, Rseq, c \} | Pj \) to the client
8. When the serial number seq is found to be an integer multiple of \( k \), start a Checkpoint request (to implement each \( k \) message once)
9. When a request for a checkpoint from the \( f+1 \) node is received, a checkpoint request is started.

When a node receives a \( \{ \text{Prepare} \} \) message from the master node, it will authenticate the message:

1. Verify the signature of the \( \{ \text{Prepare} \} \) message
2. The message \( m \) is consistent with the message it received and the message digest is correct.
3. \( V \) is the current view number
4. seq is the seqi+1 of the previous message
5. If the authentication is unsuccessful, the node will send \( \{ \text{Panic}, \text{seq}, v, D(m), c \} | Pj \) to the client indicating that the reliability of the master node is not recognized.

Once the node Pj accepts the \( \{ \text{Prepare} \} \) message sent by the master node, it stores the authenticated \( \{ \text{Prepare} \} \) in the local stack in the order of seq, and executes the \( \{ \text{Prepare} \} \) operation in order to send it to other nodes. The other nodes save the messages sent by the other node and save them in their own cache area, and then perform the corresponding operations in the order of seq. Such a mechanism prevents the node from knowing the client's related request and then in conjunction with the action that the compromised client made after executing the node before the previous operation was performed.

The backup node predicts the O operations in the execution message m in order, and sends a \( \{ \text{Reply}, v, t, \text{seq}, Rseq, v, c \} | Pj \) to the corresponding client and a \( \{ \text{Commit}, v, \text{seq}, Rseq, m, D(m), c \} | Pj \) to all nodes, where Rseq represents the result of the node prediction execution.
3.2. backup node
In order to verify the correctness of the prediction execution results, a node needs to collect 2f+1 matching \{Commit\} messages to determine that the result is correct. When a node receives F+1 \{commit\} messages but does not receive any \{Prepare\} or \{request\} messages, the node can execute the message first and also send a \{reply\} to the client and \{commit\} message to all nodes. When a node receives f+1 matching \{Commit\} messages different from the \{Prepare\} message it receives, the node sends a \{Panic\} message to the client.

1. initialization
2. A \{All replicas\}
3. Seqi = 0
4. Cm \{a collection of \{Commit\} messages\}
5. If you receive \{Request,o,t,c\}|c, send this request to the master node.
6. If you receive \{Prepare,v,seq,D(m),m,c\}|P
7. Whether the seq is judged to be the smallest, if not stored in the cache of the node in order, the execution is performed.
8. If seq is the smallest, judge the correctness of \{Prepare\}
9. If you send \{Commit\} and \{Reply\} correctly
10. If seq is an integer multiple of k, send \{checkpoint\} and stop accepting messages, except for checkpoint messages sent by other nodes.
11. otherwise send \{Panic\} message
12. If a commit message is received, the commit message is counted, if 2f+1, and the result is confirmed.
13. If less than 2f+1 send a panic message
The forwarding of the \{commit\} message ensures that at least f+1 correct nodes have executed the request. If any correct node receives inconsistent information, then the primary node must be wrong and will stop accepting messages.

3.3. client
We have transferred some key steps to the client to execute, so the correctness of the client is also very important. Many BFT protocols are not very strict with the client. In the paper [8], it is only a simple judgment of whether the \{panic\} message is a checkpoint message, and there is no error \{panic\} that can prevent the client from arbitrarily initiating.

We make each node aware of the execution of other nodes through the \{commit\} message sent between the nodes. When the client sends a \{panic\} message, it needs to carry the execution result in 2f+1 related \{reply\} messages. If the f+1 nodes find that the results of the two are inconsistent, it will be judged that the client is wrong.

The client has a time setting for each request, and the client will match the \{Reply\} message. If 2f+1 messages are received before the time expires, it will complete the request. If we receive less than f+1 requests, we will resend the request. In other cases, a \{Panic,v,seq,AllReply,t\} message is sent, where AllReply represents all of the Reply messages received by the client.

1. initialization
2. A \{All replicas\}
3. Re \{set of \{Reply\} messages\}
4. Start 1 \{Start timer\}
5. If Receive receives 2f+1 \{reply\} requests, it will accept the request, transfer the result to the application, and stop timing.
6. If \{reply\} is received less than f+1, the message will be retransmitted and the timing will be restarted, twice as long.
7. If the received message f+1 and 2f+1 will be sent after the timeout expires, a \{panic\} message will be sent to all nodes.
3.4. dynamic join

Previous BFT protocols used some static nodes to complete consensus and execution. However, when a node has an error, it cannot dynamically join the new node. It can only resend the server key by restarting the system and through the key distribution center. Such a mechanism is not conducive to the dynamic joining of nodes. We consider that when a node fails, we need to complete the replacement of the error without restarting the system and re-join a new node.

Below we describe our dynamic join algorithm, join the node as a consensus request, and complete it in the consensus platform. Through the collaboration of the primary node, the backup node, and the client, consensus can be completed and operations can be performed. Our algorithm's full All_replicas represents the number of all participating consensus nodes and Join_pk and Join_sk represent the key pairs that join the node.

The node that wants to access sends \{Join, Join_pk, t, ID\} to all nodes, where Join_pk represents the public key of the node, t represents the timestamp, and the ID is generated by the corresponding hardware device of the server to ensure that the server can be executed uniquely. After receiving such a request, the master node generates a \{Join_prepare, v, seq, All_replicas, Join_pk, D(Join_pk), ID, id\} to broadcast this message to all nodes. Where id represents the id number of the primary node for the newly joined node. After receiving the \{Join_prepare\} message, the backup node compares its received \{Join\} request with related parameters. If the authentication correct backup node sends a \{Join_reply, v, seq, All_replicas, id, ID, list, t\} to the node to be joined. Where list represents the collection of all node public keys. And send a \{Join Commit, v, seq, All Replicas, ID, id, list, t\} to all nodes. After the node to be joined receives \{Join_reply\}, if there are 2f+1 matching \{Join_reply\} messages, the list, id, and All_replicas in the message are saved as the credentials of the subsequent consensus. If less than 2f+1 are received, \{panic,v,seq,All_replicas,id,list,t\} is sent to all nodes to all nodes, and the node judges \{panic\} by the corresponding \{Join_commit\} message. the judgment is successful, the Model Switch is executed. If it is wrong, the join request of this node will no longer be accepted.

4. CHECKPOINT

We added the checkpoint mechanism to ensure that the seq sequence number of the active node can be separated by k to ensure the stability of the system, and also to ensure the state of the active node and the inactive node. Since so far, active nodes are not able to determine that their status updates have been transmitted to inactive nodes, and the status of the inactive nodes is determined by the checkpoint mechanism. We reference the periodic checkpoint protocol to accomplish this task.

When the consensus sequence number seq received by the node is an integer multiple of the global fixed variable k (such as k=100), a \{checkpoint, R, t\}|Pj will be broadcast to all nodes. Where R represents all operations and related messages that this node received after the last checkpoint steady state. And stop receiving any non-checkpoint requests. After receiving the \{checkpoint\} request, the master node will issue a \{checkpoint_prepare, R, seq, v, t\} to all nodes if it also completes an integer multiple of seq or has received f+1 checkpoint requests. As long as a node determines that the received R has the same sequence number and the digest value of the message, then the node reaches the checkpoint stability and sends a \{checkpoint_commit, R, true, seq, t\}|Pj to all nodes. After a node receives 2f+1 \{checkpoint commit\}, it can confirm that the checkpoint has reached a stable state, and the previous information can be deleted.

5. MODEL SWITCH

When the node detects the error of the master node or the client finds that the results of the execution of each node are inconsistent and the client acknowledges the consensus result, the client triggers the Model Switch sub-protocol.

In the conversion, the node provides a consensus history containing the messages that the node has performed so far. Based on the history of the local nodes, design a transformation manager to create and distribute a global consensus history to ensure that hBFT is on a common basis at the beginning.
5.1. Initialization model switch
This protocol is a trigger condition for the conversion to be completed by the client. If the client gets a valid result of operation \( o \) within a valid time, the client will broadcast a \( \{ \text{Panic, Nc, c} \} | \text{C} \) message to all nodes, where \( \text{Nc} \) represents the inconsistency information obtained by the client. Usually, there are different reasons for the results to become inconsistent, such as active node errors, and can not correctly participate in the consensus.

Once the Panic message is received, the correct node will not enter the mode transition immediately, and try to stay in fault-free cases. For example, the node will ignore the old request broadcast by the same client if the client has already broadcast a new request. In addition, when the node receives the panic message, it will send the message to the master node, and only if it receives another panic message under the same request. Another mechanism to protect the wrong client to send a panic message: the node receives a continuous panic request from the same client. If the request is already in the previous checkpoint, the node will retransmit the relevant reply. If these conditions are met, the node will compare the Nc part of the client \( \{ \text{panic} \} \) message with the \( \{ \text{commit} \} \) message it received. If the node authenticates that Nc is correct, the node begins to create a local commit history and sends a panic message to all nodes to ensure that other nodes are also subject to panic messages. If the node authentication Nc is wrong, \( \{ \text{Error}_{\text{client}}, \text{Nc}, \text{c}, t \} | P_j \) will be sent. When other nodes receive the \( f+1 \{ \text{Error}_{\text{client}} \} \) message, they will judge that the client is wrong and will no longer accept messages from this client. In addition to manual debugging.

5.2. Create a local commit history
When running the mode conversion protocol, a correct node will not perform consensus services and all consensus services will stop. At the beginning of the pattern transition, each correct node creates its own local commit history. The node's local history includes a series of matching checkpoints to provide the most recent stable state. In addition, in order for other nodes to know the request submitted by themselves, a correct node inserts the message already prepared into his local history. If a request has been prepared then the node should also send a commit message, which may cause the request to have been committed or executed. For such a request, we will add the corresponding \( 2f+1 \) prepare message we received when building the local history.

Once the local commit history \( h \) is completed, an active node \( A \) sends \( \{ \text{History}, A, h \} | A \) to all nodes and the transition coordinator, and the role of the transition coordinator is arranged according to the node's id.

5.3. Create a global commit history
When the transition coordinator receives \( \{ \text{History} \} \) from the active node, it only processes messages whose signatures are reliable. Based on the local commit history and the commit history of the other \( f \) active nodes received, the transition coordinator creates a global commit history by the following method.

Because in the consensus phase we use the order of the total order to execute the relevant request, each correct node is in accordance with the seq serial number when saving the client request. When the transition coordinator gets a reliable local commit history, the seq serial number will be installed from low to high. A global history is formed based on the history of the highest seq and vacancy in each local history.

The global commit history is formed by the local commit history of different nodes. We refer to each seq in the commit history as a slot. For example, there are three local history slots. From the checkpoint state, the highest seq is 217 but the middle 210 is empty, the highest seq of node 2 is 219, and the highest seq of node 3 is 211. The transition coordinator then decides which slot to join the global commit history, and he will follow the following rules:

1. If \( f+1 \) reliably local commit history contains the request \( o \) then \( o \) will be submitted into the slot.
Otherwise, when the first rule is not met, the slot will be set to null if there is no reliable local history containing the proof of the slot or the proof is unreliable. In both cases, each request is submitted in a protocol-related example, at least \( f+1 \) reliably local commit history.

When the global variable is finished, the transition coordinator broadcasts a \{switch, H, t\}|Pi to all nodes, where Pi is the id number of the transition coordinator node.

In this way, we can generate a global commit history. After receiving a global commit, each active and inactive node re-runs the previous process to verify that the global commit is reliable. In order to achieve this goal, the local commit history must be authorized to be digitally signed. Otherwise, if the authorization mechanism is not used, the correct node does not have to go through the transition coordinator's decision on the local history.

### 5.4. Handling errors in Model Switch

In case the transition coordinator is wrong, it will not be able to establish a reliable global commit history. In order to locate such a related problem, the node sets the mode conversion time, and cannot obtain a reliable global commit history within the time range, then the transition coordinator is considered wrong. This time is set long enough that the transition coordinator is considered incorrect because of a transmission error.

When the time expires, an active node A changes its id from p to p' and sends the local commit history \( h \) to \( P' \), in addition to which the node will set twice the expiration time for the new transition coordinator.

### 6. SAFETY AND ACTIVITY

How a BFT protocol can guarantee a trusted result requires ensuring the security and activity of the protocol. Below we demonstrate the security and activity in our agreement.

#### 6.1. Safety

A transition coordinator requires \( f+1 \) reliable active node local commit history to create a global commit history. In the normal-case case, only all nodes participate in the work. This mechanism ensures that each subset of \( f+1 \)'s local history contains all the history submitted by at least one correct node. If a request \( o \) is committed by one or more of the correct nodes, then the request is at least partially prepared by some nodes. Conversely, node A inserts a history that reliably proves that operation \( o \) enters the local, and eventually these will be included in the global commit history. Only node A does not need to prove the operation \( o \) when that request is included in the last stable checkpoint. In that case, however, all the correct nodes either acknowledge the request they have executed or have implemented a corresponding status update (if it is an inactive node). Otherwise checkpoint will not be recognized.

For the same reason, the global commit history should also be kept secure. If one or more nodes do not acknowledge the execution of the consensus request (or an update of the state), then the global commit should contain proof of the reliability of those requests.

#### 6.2. Activity

If an active node is not able to participate in the consensus due to failure without a node error, then the consensus will be stopped. Conversely, the wrong consensus node cannot directly prevent the system from working. If at least one active node does not acknowledge the arrival of the checkpoint, the checkpoint cannot become stable. As a result, the consensus of the protocol will be stopped because the node cannot upgrade the serial number, but the new request does not get the legal serial number. In addition, a consensus protocol that has been stopped prevents the system from executing other requests, especially client-enforced mode transitions due to the lack of nodes. Since the correct node transmits the panic message, eventually all nodes will initialize the node transition.

Since the mode conversion protocol is triggered locally, the transition coordinator requires \( f+1 \) nodes to reliably localize the commit history from different node regions to create a global commit history. Since there are at most \( f \) erroneous nodes in \( 2f+1 \) active nodes, this will ensure that the
transition coordinator will eventually receive f+1 or more reliable local commit history. Further, by adjusting the expiration time, ensure that the transition coordinator arranges different nodes to prevent errors in generating the global commit history.

7. CONCLUSION

We use the "predictive execution" and "resource optimization" ideas to improve the BFT system. The improved algorithm can achieve two-phase consensus in 2f+1 nodes. This result makes the time complexity of the consensus algorithm greatly reduced. Compared with the paper [6], the number of nodes in the reception in our algorithm is reduced by 1/3. Compared with the paper [7], our algorithm can change from the consensus of three phases to two. The consensus of the phase. At the same time, we consider the total order problem, so that each request is executed in order, preventing some wrong nodes from colluding with the wrong client to infringe the request of the normal client. Finally, on the dynamic joining problem of nodes, we propose an external node that can join the BFT system without restarting the system. This implementation increases the fluidity of the system while increasing the security and activity of the system.

REFERENCES

[1] Cowling, James, et al. "HQ replication: a hybrid quorum protocol for byzantine fault tolerance." Usenix Symposium on Operating Systems Design and Implementation USENIX Association, 2006:13-13.

[2] Castro, Miguel, and B. Liskov. "Practical Byzantine fault tolerance." Symposium on Operating Systems Design & Implementation ACM, 1999:173-186.

[3] Guerraoui, Rachid. "The Next 700 BFT Protocols." European Conference on Computer Systems Design & Implementation ACM, 2010:363-376.

[4] Hendricks, James, et al. "Zzyzx: Scalable fault tolerance through Byzantine locking." Ieee/Ifip International Conference on Dependable Systems and Networks IEEE, 2010:363-372.

[5] Kotla, Ramakrishna, et al. "Zzyzzyva: Speculative Byzantine fault tolerance." Acm Transactions on Computer Systems 27.4(2009):7.

[6] Zielinski, Piotr. "Optimistically Terminating Consensus: All Asynchronous Consensus Protocols in One Framework." International Symposium on Parallel and Distributed Computing IEEE, 2006:24-33.

[7] Zielinski, Piotr. "Optimistically Terminating Consensus: All Asynchronous Consensus Protocols in One Framework." International Symposium on Parallel and Distributed Computing IEEE, 2006:24-33.

[8] Kotla, Ramakrishna, et al. "Zzyzzyva: Speculative Byzantine fault tolerance." Acm Transactions on Computer Systems 27.4(2009):7.

[9] Duan, Sisi, S. Peisert, and K. N. Levitt. "hBFT: Speculative Byzantine Fault Tolerance with Minimum Cost." IEEE Transactions on Dependable & Secure Computing 12.1(2015):58-70.

[10] Kapitza, Rüdiger, et al. "Resource-Efficient Byzantine Fault Tolerance." IEEE Transactions on Computers 65.9(2016):2807-2819.

[11] Duan, Sisi, M. K. Reiter, and H. Zhang. "Secure Causal Atomic Broadcast, Revisited." Ieee/Ifip International Conference on Dependable Systems and Networks IEEE, 2017:61-72.

[12] Ding, Yihua, J. Z. Wang, and P. K. Srimani. "Churn Tolerance Algorithm for State Machine Replication." Ieee/wic/acm International Joint Conferences on Web Intelligence and Intelligent Agent Technology IEEE Computer Society, 2012:356-360.

[13] Tsai, Wei Tek, X. Bai, and L. Yu. "Design Issues in Permissioned Blockchains for Trusted Computing." Service-Oriented System Engineering IEEE, 2017:153-159.