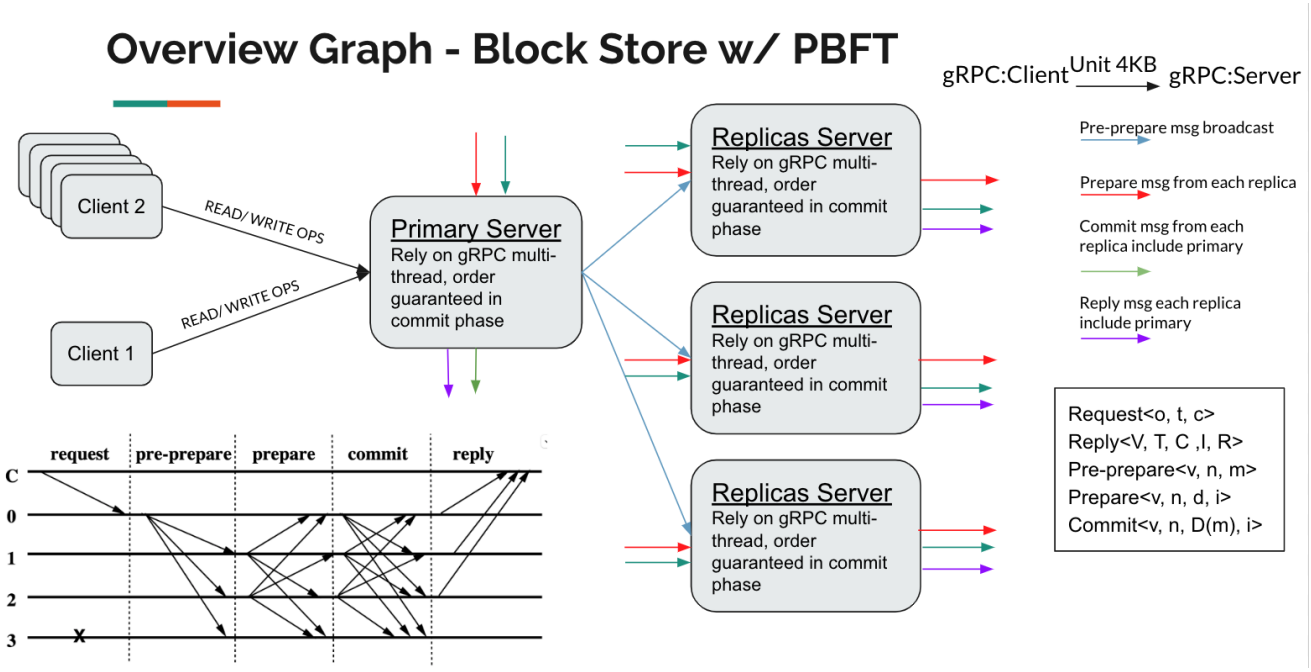
Project IV Report: Replicated Block Store w/ PBFT

Domen Su, Hao-Yu Shih, Shutao Wang, Yatharth Bindal

Project Overview

The idea of this project was to design and implement a replicated network-based block storage system, using PBFT to guarantee strong consistency. It is a simplified form of cloud services such as Amazon EBS, and also with strong guarantee of correctness to prevent malicious servers. The objective of this project was to extend previous project's understanding of the CAP theorem and the trade-off of strong consistency, to include the Practical Byzantine Fault Tolerance consensus algorithm. On the server side, we follow closely with the PBFT protocol described in the paper. As a block store service we simply offer read/write operations to the clients, also we provide guarantee that the correct operation will be carried out under system has more than 3*faulty node +1 total nodes. As mentioned above, the key challenge this paper addresses is to prevent malicious replicas including faulty primary, and also guarantee strong consistency as a whole.

Design Overview



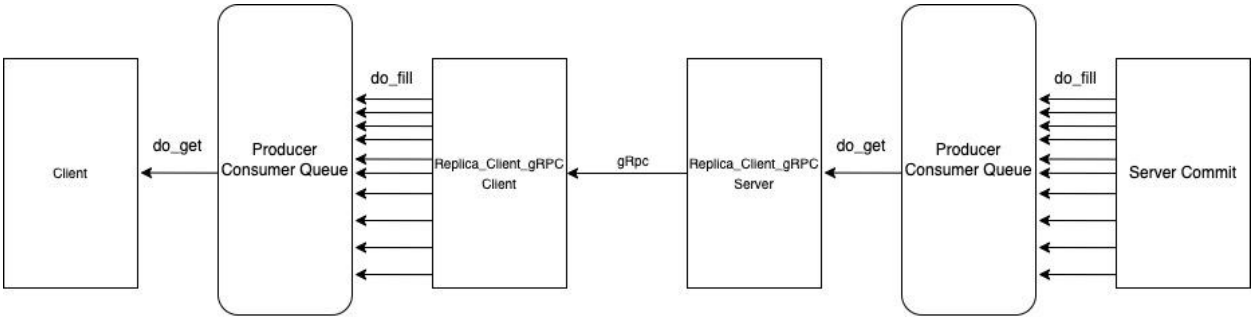
Rationale and Assumptions

- Only export two operations APIs to Client: read/write
- Only two four nodes for Replicas
- Clients knows all replicas' IP address and public keys

Design Choices

Producer and Consumer Queue

To avoid establishing gRCP channels from replica to client at everytime a commit phase has been reached, we decided to utilize gRPC server streaming service call, which runs infinitely of the lifetime of replica. This mechanism helped us alleviate the issue of non bi-directional communication nature of gRPC service calls. To incorporate better with the multi-thread nature of gRPC, we implemented a producer and consumer queue for reply messages to be filled and client gets data more efficiently. Client and service side can process request and reply information asynchronously, as another benefit of this implementation choice.



Digital Signatures

PBFT requires digital signatures for authentication to handle Byzantine Faults. We can either use the built-in gRPC authentication or implement them by ourselves. We choose **OpenSSL** to sign the message because of the flexibility.

- gRPC authentication
  - It is easy to use.
  - It is not flexible enough for our use case, since we need to relay and store the signed messages.
- OpenSSL - RSA
  - It is more flexible.
  - We need to sign and verify the messages by ourselves.
  - This makes the gRPC interface become obscure. We use a generic "SignedMessage" as an argument for most gRPC functions.

Crash Recovery

There are 3 different ways to implement crash recovery. In our implementation, we choose the 3rd option for simplicity.

1. Treat newly recovery replica as a client, send request to sync data with PBFT protocol
    - Advantage
        - Correctness & safety: The consensus protocol can guarantee the total order.
        - It is easy to implement.
    - Disadvantage
        - Not efficient: It requires a full PBFT protocol.
2. Contact all replicas
    - Advantage
        - Safety: It contacts at least one non-faulty node.
        - More efficient than the 1st option.
    - Disadvantage
        - It requires another complex protocol.
3. Contact any replica to sync content
    - Advantage
        - It is most efficient.
    - Disadvantage
        - Require an additional RPC call,
        - Safety: The client may not trust the contacted replica. However, this can be solved by providing a set (2f+1) of signatures

Crash Recovery - State Comparison

After contacting the replica to sync content, we have 3 ways to sync the data. We choose the 3rd option because it is most easy to implement.

1. Use Merkle Tree to compare the state
    - Advantage
        - It is efficient: O(log(storage size)).
    - Disadvantage
        - It is an interactive process, which is more difficult to provide atomicity.
2. Use hashes of blocks to compare the state
    - Advantage
        - It is a non-interactive process, which is easy to provide atomicity.
    - Disadvantage
        - Not as efficient as the 1st option: O(sqrt(storage size))
3. Replay the history
    - Advantage
        - It is easy
        - Can recover efficiently
    - Disadvantage
        - The replicas have to remember all the operation history.

Other Design Choice

1. Predefined Replica ID: We use a json configuration file to indicate the replica id and

keys. We cannot dynamically add nodes, which follows the original PBFT design.
2. We use client public key as client ID since it reduces the server state.
3. Since gRPC is multi-threaded, we use multiple conditional variables to guarantee the total of operations.

Fault Tolerance

Our implementation handles a number of failures originating from both faulty primary and faulty replica servers. The primary server holds some power in PBFT as it is the server that relays client requests to the replica servers. The primary is also responsible for generating credentials for the requests that the client sends, which include the sequence number and the digest of the client's message. Therefore, it is essential for our protocol to be able to detect any malicious behavior by the primary. Below are some scenarios where the primary behaves maliciously along with brief descriptions of how our implementation will detect and deal with them.

Scenario	**Protocol Behavior**
Primary ignores the client's message	The client's request times out and it detects that the primary is faulty
Primary modifies client's message before pre-prepare	Replicas are not able to verify the signature for the modified message and reject it.
Primary assigns an incorrect sequence number to a message	Replicas detect the inconsistency after referring to their operation history and reject the message
Primary sends a prepare message	Replicas detect the origin of the message and reject the prepare messages from the primary
Primary sends different messages to different replicas	Replicas fail to reach consensus

Replica servers also play an important role in PBFT, which can handle F faulty replicas out of a total of 3F + 1 replicas. Our implementation also provides this guarantee and we are able to handle various scenarios where a replica server, or a group of replicas working together attempt to relay incorrect information. Below are some examples of what our protocol would do in scenarios where the replica server(s) are faulty.

Scenario	Protocol Behavior
Replica sends a pre-prepare request.	Other replicas detect it and reject the request
Replica modifies some request information in the prepare stage	The modified information gets rejected by other replicas
F replicas team together and attempt to commit an incorrect request	The incorrect message fails to reach consensus as at least F+1 replicas have the correct request.

Evaluation methodology & Experiment designs:

Hardware Testing Environment:

Our environment uses 6 nodes on Cloudlab, and all nodes are running Ubuntu 20.04LTS. All the nodes have X86_64 architecture with model name Intel(R) Xeon(R) CPU E5-2660 v3@2.60GHz. Every system has 16GiB DIMM DDR4 2133 MHz memory and uses I350 Gigabit Network Connection.

Critical Implementation Details

We implemented the whole project in C++, and used the OpenSSL library for generating client and replicas' private and public key, also encryption and verification process, and we use gRPC for client replica remote procedure calls as communications. The above mentioned consumer and producer queue with an infinite running server streaming reply service was one of our main interesting engineering designs to bypass the limited ability of non bi-directional support nature of gRPC.

Availability

The availability will be heavily limited by the speed of how our implementation of PBFT processes all the client requests.

Strong Consistency

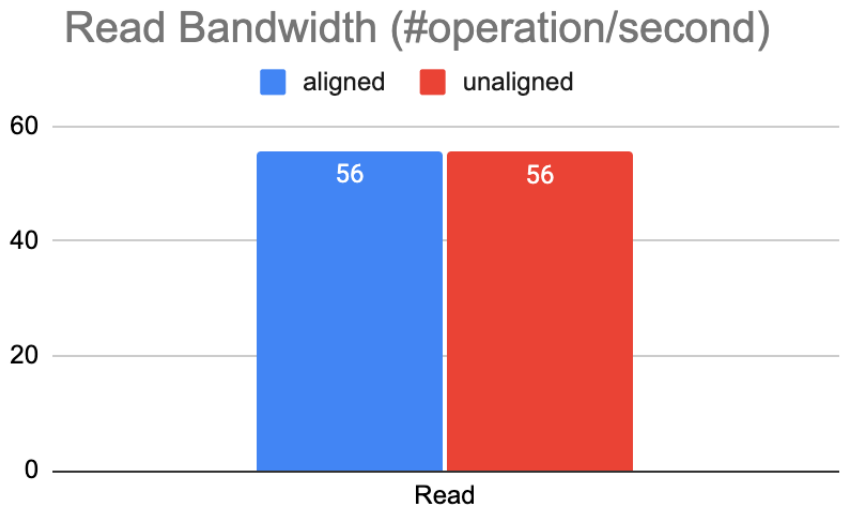
We use PBFT consensus protocol that provides us with strong consistency guarantees where our system of 3F + 1 nodes can handle F faulty replicas while having no impact on our consensus.

Testing Strategy

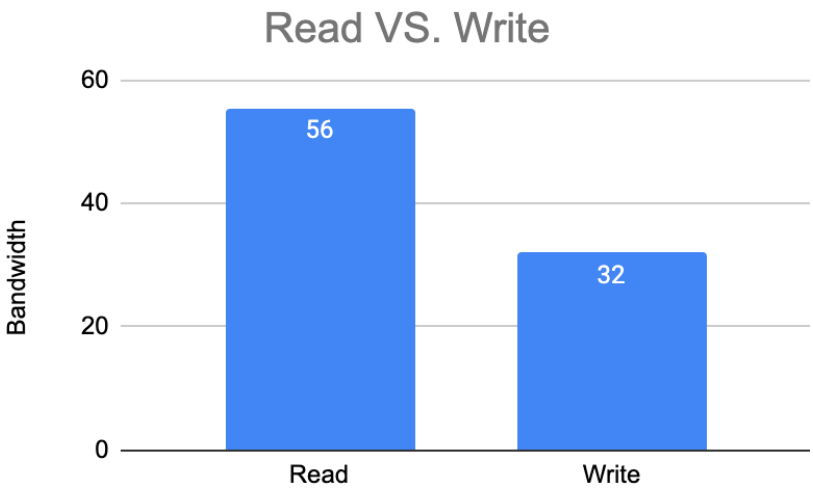
Unlike fail-stop models, in which we can inject crashes to the service for testing, we have to implement some malicious behavior to test BFT systems. We have come up with a few Byzantine Faults that our system is able to tolerate as shown in the previous section. However, due to the time constraint, we implemented one Byzantine Fault and tested other fail-stop situations. We modified the primary replica implementation to create a fake client request. The modified primary replica will send the legitimate request to half of the replicas and the fake request to others. The PBFT system should not reach a consensus in such a situation and eventually causes a view-change. We did not implement the view-change, so we test if our system can correctly detect such a situation.

Performance

1. **Bandwidth for Aligned address/ unaligned address read**

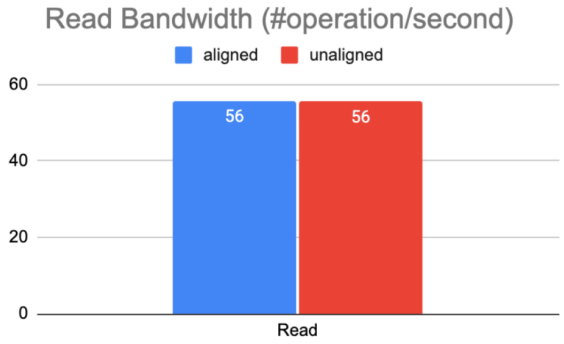


2. **Bandwidth for Read/write**

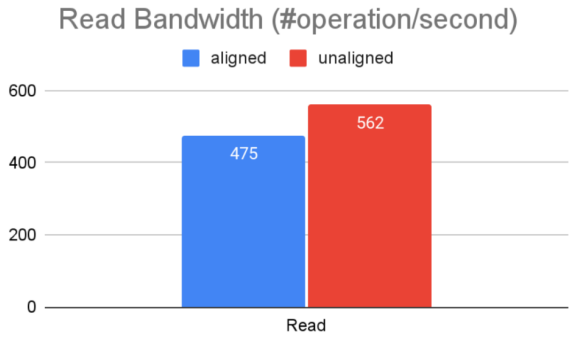


3. **Read performance comparison between PBFT and Primary_Backup**

PBFT

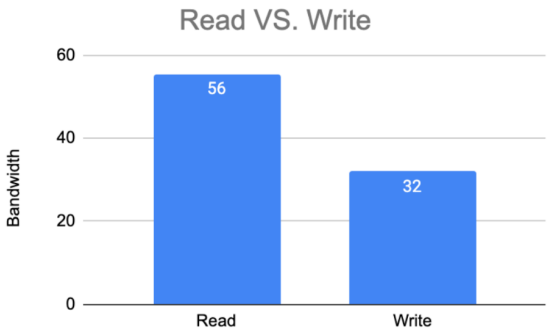


Primary_Backup

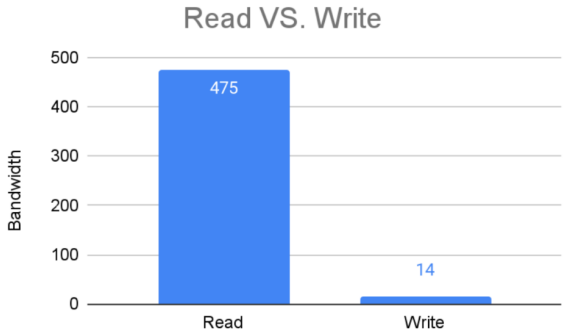


4. **Performance comparison between PBFT and Primary_Backup**

PBFT



Primary_Backup



Conclusion & Lessons learned:

In this project, we design and implement a replicated network-based block storage system, using PBFT to guarantee strong consistency and guarantee correctness by preventing malicious servers. In the commit phase of our PBFT implementation, the debugging process is quite challenging with various conditional variables to guarantee total ordering of the operations. We utilized producer and consumer queue to provide bi-directional communication of gRPC from Reply request from replica to clients. Nevertheless, we heavily rely on gRPC's underlying multithreading scheme to optimize our PBFT protocol system efficiently, since our system implementation relies on gRPC message triggering; sometimes without going deep into the gRPC source code or a version of documentation, we would misinterpreted and have the wrong assumption, this has cost us heavily during our developing and debugging stage of this project.

Reference:

[PBFT]: Miguel Castro and Barbara Liskov. 1999. Practical Byzantine fault tolerance. In Proceedings of the third symposium on Operating systems design and implementation (OSDI '99). USENIX Association, USA, 173–186.

[ALICE]: Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI'14). 433–448.