ECS 235A Final Project Report: Visualizing pBFT

Lily Bhattacharjee

Demo Link

Github Repo

1 Introduction

Practical Byzantine Fault Tolerance (pBFT) is a seminal consensus protocol designed to ensure strong data consistency and high availability under certain constraints regarding the number of good vs. malicious replicas, as well as network communication reliability [1]. Modern distributed systems often split data (sharding – for an example framework for extending pBFT principles to a multi-shard system, see ByShard) and transaction processing across multiple nodes, replicating records so requests don't have a single point of failure in case a node containing the target crashes [2].

In a singly-sharded relational database system containing n initially identical replicas, executing a transaction synchronously across all copies of a record is a high-communication task [3]. Network bandwidth, latency, and error rate can be unreliable, with packets being dropped, corrupted, or replayed [3]. Waiting for all replicas to respond with a success message after reaching consensus can result in the slowest replica(s) creating bottlenecks, reducing throughput as the client waits for confirmations before proceeding to the next, possibly dependent transaction [3]. The strength of pBFT isn't just in its ability to ignore up to a certain ratio ($n/3$, where n is the total number of replicas) of malicious or unsuccessful behavior, but also in its recovery mechanisms – each replica maintains a log that can be synced and used to regain consistency if nodes go down and come back later [3].

pBFT in the normal case isn't overly complex, comprised of four largely sequential stages: pre-prepare, prepare, commit, inform [3]. Messages corresponding to different transactions or views can be interspersed with each other, as the protocol doesn't wait for all replicas to respond at each stage, but recipients can drop delayed or unnecessary communications, preventing processing errors [3]. However, replicas can initiate view changes to switch primaries, and the resulting communications and their results can be difficult to visualize, in response to malicious behaviors including the following:

- *Bad primary*: insert forged client transactions [3].
- *Bad primary*: prevent replication of client transactions from some or all clients e.g. by not sending the transaction to good replicas [3].
- *Bad replica*: send invalid transaction results to client [3].
- *Bad replica*: interfere with correct working of distributed system e.g. convince replicas other good replicas / primaries are malicious [3].
- *Bad replica*: other methods of disrupting the consensus protocol e.g. by replaying messages from previous views [3].
- *Bad primary / replica*: not responding to other replicas / the client [3].

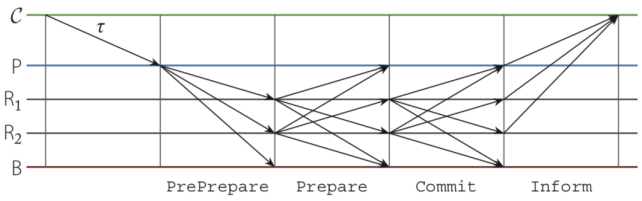
The objective of this project is to facilitate an understanding of pBFT and how it recovers from / handles malicious replicas or primaries to reach consensus and enforce consistency across good replicas. To this end, visualize-pbft is a full-stack web application that can be run locally and corresponds to a deployed mirror on Heroku, allowing the user to enter system parameters before running a simulation to model consensus for t transactions, displaying node-to-node communications as directed edges in a force graph.

By mapping communications to phases and database states (in this case, a bank with 3 customers), users can identify how different stages are affected by malicious behavior, and how this differs from the normal case.

2 Background

This section briefly dives into the core concepts of pBFT – normal case behavior, and view change / what can trigger one. Optimizations include validating new views, dealing with unreliable communication, the replica-specific checkpoint algorithm and recovery, and limiting message sending redundancy / network traffic, but were outside the scope of implementation for this final project [3].

2.1 Normal Case



On a high level, pBFT operates by sequentially moving through views $v = 0, 1, ...$ In each view, the steps below occur:

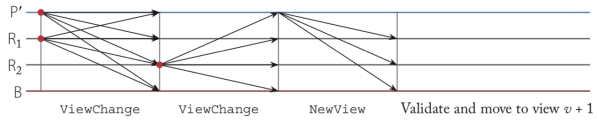
- Replica $p = v \mod n$, where n is the total number of replicas, is elected primary [1].
- Client c sends a transaction request $< T >_c$ to the primary, signed for request validity [1]. A malicious primary shouldn't be able to modify T without being detected by other replicas. The signature also ensures non-repudiation by the client [1].
- p proposes the ρth transaction request to all replicas by creating a $m =$ PrePrepare$(< T >_c, v, \rho)$ message [1].
- Upon receiving m from p, each replica R generates Prepare(m), and broadcasts to every other replica [1].
- R waits until it receives Prepare messages from at least g distinct senders, after which it enters a prepared state [1].
- R enters the commit phase, broadcasting Commit(m) to all other replicas [1].
- R waits until it receives Commit messages from at least g distinct senders, after which it enters a committed state [1]. The purpose of the Prepare all-to-all broadcast is so each R can ensure that at least g other replicas have also received a pre-prepare message [1]. The purpose of the Commit all-to-all broadcast is so R is now aware that at least g other replicas have received g other prepare messages i.e. $\geq g$ replicas are definitively "good" and ready to commit [1].
- R enters the committed state, after which it schedules the transaction for local execution [1]. After all previous transactions are executed and appended to its log, R executes and appends T, sending an Inform$(< T >_c, \rho, r)$, where r is the optional result of T directly to the client [1].

To be robust to not just node failure / unresponsive behavior ($n > 2f$, where f is the number of faulty nodes), but malicious

nodes, the following conditions must hold: $n > 3f$ and $g > 2f$ [3]. Assuming that there are f actively bad nodes, we know there must be more than f good nodes to balance them out so transactions can proceed and be executed [3]. However, those nodes could crash, for completely non-malicious reasons [3].

In that case, to mitigate the f faulty nodes, we need at minimum f more good nodes; in the case of overlapping transactions due to the asynchronous nature of pBFT's processing, at least f nodes must be executing the same transaction to gain a simple majority [3]. Hence, $g \geq 2f + 1$, and as $n = g + f$, $n \geq 3f + 1$ [3]. If these relationships don't hold, pBFT is not guaranteed to reach consensus [3].

2.2 View Change



Failure detection in pBFT is difficult, particularly for malicious primaries that send pre-prepare messages to $\leq f$ good replicas, keeping them out of sync with the rest of the system [3]. However, if replica R monitors the commit algorithm and determines that forward progress hasn't been made for some specified amount of time e.g. missing / delayed pre-prepare messages, R will assume primary failure [3]. It will stop the algorithm and take the following steps:

- R will broadcast ViewChange(E, v), where E is the set of R's past PrePrepare messages, to all other replicas [3].
- If some other replica R' receives ViewChange requests from at least g distinct senders, it joins the view change quorum, sending its own [3]. By stopping its participation in the commit algorithm when it detects failure, R guarantees a failure detection cascade if at least g good replicas in total agree [3]. This is because if R is a good replica, stopping participation will prevent the other stages from proceeding, causing all other good replicas to eventually detect failure [3].
- The new primary $(v + 1) \mod n$ is elected, broadcasting NewView$(v + 1, V, N)$ to all other replicas, where V is the set of g ViewChange requests received by the new primary and N is a set of optional no-ops to fill in request holes [3]. pBFT proceeds with the transaction in a new view [3].

3 Implementation

To accept user input from a frontend interface, on load, visualize-pbft displays a dashboard page built with Javascript / jQuery / d3, HTML, and CSS, and sends a POST request to backend containing form data when a submit button is pressed. The Flask backend asynchronously calls a simulation function and waits for a specified timeout (20s, configurable) for results to return, including log, transaction, replica database state data, etc. This is hardcoded because for certain input configurations, e.g. $n = 6, f = 4$, pBFT will never succeed, failing at an intermediate phase to proceed forward. In these cases, the simulation will return data up to the communication after which the program hangs.

The backend then calculates / wrangles data to be displayed to the user, reloads the index route, and passes objects to a Jinja template, displaying the dashboard with simulation data corresponding to the latest inputs.

Note: For more detailed local setup instructions, please see the project README.

3.1 Node-to-node Communication

Upon entering the simulation function, the program selects the specified number of Byzantine replicas from the set of n, and models inter-node communication by forking $n + 1$ threads, one for each of the replicas, and one for the client. The threads synchronize between phases and transactions by communicating with the main process. Each replica / client has two queues: one for passing objects to main, and the other for receiving objects from any other thread. For example, if replica a wants to pass a message to replica b, it will push it into b's receiving queue.

Between phases, to maintain separation in this basic model of pBFT / to avoid race conditions, all threads send an object to main, which hangs until each thread responds before moving to the next phase. This is not a built-in feature of pBFT and reduces throughput – in a more advanced design, replicas would use the view number of received messages to decide whether to drop or enqueue for future processing. The following is the structure for the client and replica threads, determining how they interact with each other. The replica thread function is that same as that used for the primary because primary is a temporary state that alternates between replicas.

Client

- Note: the contents of this thread are running indefinitely in a while loop.
- Receives transaction from main.
- While the transaction t has not been successfully completed, the client executes the following steps.
- Sends t to the current primary.
- Waits for acknowledgment that the primary has been informed.
- Wait until $f + 1$ identical Inform messages are received. Notify main via queue if this does not succeed and move to the next iteration of the inner while loop, starting the next attempted primary notification for the transaction. If it does, send a different message to main, and move to the next iteration of the outer while loop, waiting for the next transaction to be sent.

Replica (in Primary mode)

- Note: the contents of this thread are running indefinitely in a while loop.
- Receives transaction message from client.
- Sends PrePrepare message to all backups.
- Sends Prepare message to all other replicas.
- Waits to receive identical Prepare messages from g other replicas.
- Sends Commit message to all other replicas.
- Waits to receive identical Commit messages from g other replicas.
- Executes transaction on local database.
- Sends Inform message to client.

Replica (in Backup mode)

- Note: the contents of this thread are running indefinitely in a while loop.
- Receives PrePrepare message from primary.
- Sends Prepare message to all other replicas.
- Waits to receive identical Prepare messages from g other replicas.
- Sends Commit message to all other replicas.

- Waits to receive identical Commit messages from g other replicas.
- Executes transaction on local database.
- Sends Inform message to client.

Main (Simulation Manager / Orchestrator)

- Initializes mock databases.
- Selects Byzantine replica indices to uniquely identify replica threads with Byzantine status.
- Initializes replica logs, communication queues, pairwise session keys, and signatures.
- Iteratively manages all transactions, executing the following steps.
- Elects primary, communicating the index to the client.
- Waits until the PrePrepare phase is done before sending a message to all of the replicas to move to the next phase.
- Waits until the Prepare phase is done before sending a message to all of the replicas to move to the next phase.
- Waits until the Commit phase is done before sending a message to all of the replicas to move to the next phase.
- Waits until the Inform phase is done before sending a message to all of the replicas to move to the next phase.
- Clear all communication queues, pass back data, and terminate all threads.

3.2 Cryptography

Assume some replica a is receiving a message from replica b. To ensure that the communication is reliable, pBFT's specifications attempt to guarantee authenticity and integrity – that replica b did send the message and that no one has tampered with the original communication [1].

At different stages, the protocol uses message authentication codes (MACs), message digests, and digital signatures [1]. In the original formulation of pBFT, all messages were signed (Option 3), and each node had an associated public and private key [1]. The receiving node could use the public key to ensure that the signature was consistent with the sending node and the received message [1]. However, public key encryption / decryption is computationally inefficient and doesn't scale well [1].

A way to mitigate this issue, exploiting the fact that transmitted messages are either one-to-many (e.g. replica to all other replicas) or one-to-one (e.g. replica to client), a pBFT optimization modifies MACs, which would otherwise be unverifiable by recipients, to be verifiable [1]. The underlying idea involves generating a unique session key for each pair of communicators (client and all replicas), and appending that onto a transmitted message before hashing [1]. The ten least significant bytes of the message digest are appended onto the original message before it is sent [1]. This method is used to ensure no tampering for every communication that is not a ViewChange or NewView message [1]. For those, pBFT uses digital signatures, because they are generally transmitted rarely [1].

Another modification is imposed upon one-to-many communications like Prepare messages, which are sent by one replica to all others. Because the sending replica isn't sure which replica will receive a particular sent message, instead of using a MAC message digest, the sender appends a vector of digests, known as an *authenticator* [1]. The receiving replica then uses its index to find its corresponding digest for verification. Before processing any received message in the simulation, the receiving replica determines whether the sender is consistent with the digest [1].

Generating a digest

- Get the session key uniquely shared by the sender and the recipient.
- Create the input to the digest – concatenating the message bytes with the session key bytes.
- Hash the input (with MD5).
- Get the ten least significant bytes of the resulting hash.

Example usage: generating a PrePrepare message

- Create the JSON object representing the PrePrepare message.
- For each other replica r, generate the ten-byte digest using the key shared by r and the current primary.
- Generate the final communication, including the resulting authenticator, the transaction message sent by the client (also including an authenticator with digests for every single replica, not just the primary), and the PrePrepare message. Note: If the primary tampers with the transaction sent by the client, at least one good replica will detect the change. This is because its corresponding digest will not match, as it was created by a key shared only between the good replica and the client, unknown to any malicious primary.

Verifying a digest

- Take as input the received message, shared key, and received digest.
- Recompute the expected digest using the message and key.
- Verify that the expected and received digests match.

Generating a digital signature

- Create the communication input and convert to bytes.
- Use the private signing key specific to each sender to sign the communication bytes.

Example usage: generating a NewView message

- Create the NewView communication, including the new view number, replica index, etc.
- Sign the communication bytes before sending.

Verifying a digital signature

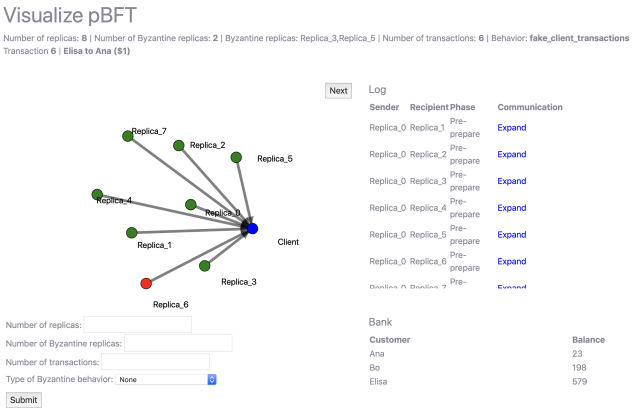
- Use the sender's public verify key to check that an authentication error isn't raised.

3.3 Heroku Deployment

The Heroku deployment only has access to one free dyno including 256 threads. The simulation should not hit the limit because threads are killed between runs. Due to resource limitations and the blocking nature of the simulation, which currently prevents the frontend from loading until the execution is finished, timeouts are set to 20s, which should be in the ballpark of the maximum time the dashboard should take to reload.

However, as the number of transactions increases, it is possible that 20s will not be enough for the simulation to fully run. If it exits early, not all steps will be displayed, but a rerun should usually fix the issue. To modify timeouts so this never occurs, running locally is the best option. Currently, timeouts are used to differentiate between situations when pBFT should hang (e.g. $g \leq 2f, n \leq 3f$) and long-running inputs.

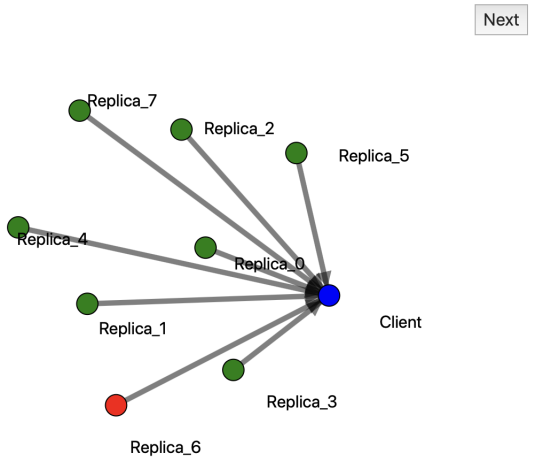
4 Visualization



The visualization includes 4 major components:

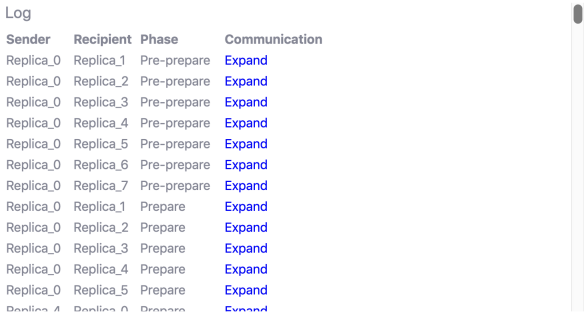
- force graph (upper left): displays nodes and the messages they transmit to each other as edges.
- communication log (upper right): traces through the messages transmitted during each phase / chunk of messages with the same type. Clicking on an "Expand" link will toggle a larger view of the message transmitted along the edge.
- options form (lower left): allows user to submit options for the next simulation.
- bank state (lower right): displays changing bank state as the user clicks "Next" through the phases, and the replicated banks reach consensus.

Force graph

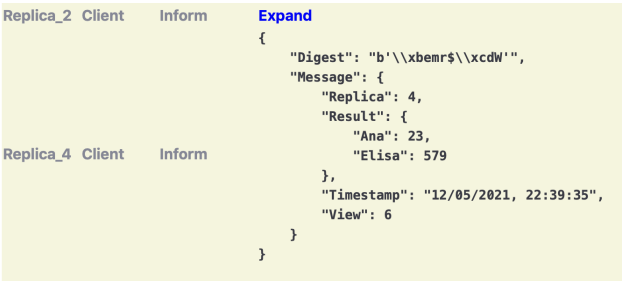


In the displayed force graph, the red replica (6) is the current primary, and the others are in backup mode. The client is blue. The system is in the Inform phase.

Communication log

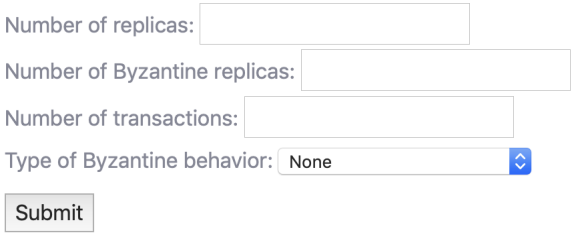


The "Expand" options in the communication column can be clicked to reveal the full message sent:



Clicking the expanded cell again will toggle visibility, shrinking it again.

Options form



The form allows users to enter (1) number of replicas, (2) number of Byzantine replicas, (3) number of transactions to run, (4) type of Byzantine behavior, before a simulation run is initiated on backend. Currently, on submit, the frontend validates that the following conditions are satisfied:

- all textbox inputs are valid integers.
- the number of replicas is between 2 and 8 inclusive.
- the number of Byzantine replicas is between 0 and 8 inclusive.
- the number of transactions is between 1 and 10 inclusive.
- if the number of Byzantine replicas is set to a nonzero number and "None" is specified for Byzantine behavior, the input is invalid.
- the number of Byzantine replicas must be strictly less than the number of replicas total.

Additionally, the visualization supports 5 types of Byzantine behavior: none (no Byzantine), no response (Byzantine replica / primary sends no messages), fake client transactions (when a bad replica is primary, it will manipulate the client transaction before sending PrePrepare messages), bad transaction results (), and bad view change messages ().

Currently, visualize-pbft only supports one type of Byzantine behavior at a time i.e. if b Byzantine replicas are specified, all of them will follow the Byzantine behavior selected in the input.

Bank state

The pBFT transactions in this visualization are being executed on a simple replicated mock database – a bank with three

customers (Ana, Bo, Elisa). Initially, their balances are set to $500, $200, and $100 respectively. During the Inform phase of each transaction, the "consensus bank state" is updated to reflect the execution results.

5 Discussion

In this section, we explore the behavior of the simulation given certain input combinations and verify the validity of the results.

Scenario 1: Normal case (n replicas)
- **Inputs**: 4 total replicas, 0 Byzantine replicas, 1 transaction, "None" Byzantine behavior.
- Primary 0 sends PrePrepare messages to all other replicas.
- Every pair of replicas exchanges Prepare messages (many-to-many broadcast).
- Every pair of replicas exchanges Commit messages (many-to-many broadcast).
- Each replica (including primary) sends an Inform message to the client. The bank state is updated with the transaction results.

Scenario 2: No response ($< n/3$ bad)
- **Inputs**: 4 total replicas, 1 Byzantine replica (Replica 2), 3 transactions, "No response" Byzantine behavior.
- Primary 0 elected – same behavior as normal case, except Replica 2 does not participate in any phase. Because only 2 replicas are needed for consensus, pBFT continues.
- Primary 1 elected – same as previous step.
- Primary 2 elected. It does not send PrePrepare messages.
- After a timeout (count to 5), a replica detects a primary failure and broadcasts a ViewChange request, triggering all other good replicas to agree.
- Replica 3 is elected the next primary, and the transaction is proposed.

Scenario 3: No response ($\geq n/3$ bad)
- **Inputs**: 4 total replicas, 2 Byzantine replicas (Replica 1, Replica 3), 1 transaction, "No response" Byzantine behavior.
- Primary 0 elected, sends PrePrepare messages to all other replicas.
- Replicas 0 and 2 exchange Prepare messages but can't move to the next stage because they have not reached at least $g = 2$ corroborations. The consensus algorithm hangs at the Prepare phase.

Scenario 4: Fake client transactions ($< n/3$ bad)
- **Inputs**: 4 total replicas, 1 Byzantine replica (Replica 0), 1 transaction, "Fake client transactions" Byzantine behavior.
- Primary 0 elected – modifies the transaction it received from the client and broadcasts PrePrepare messages to all other replicas.
- A replica detects that the transaction has been modified because of a digest mismatch. It detects primary failure and broadcasts a ViewChange request, triggering all other good replicas to agree.
- Replica 1 is elected the next primary, and the transaction is correctly proposed.

Scenario 5: Fake client transactions ($\geq n/3$ bad)
- **Inputs**: 4 total replicas, 2 Byzantine replicas (Replica 1, Replica 3), 4 transactions, "Fake client transactions" Byzantine behavior.
- Primary 0 elected – same behavior as normal case.
- Primary 1 elected – modifies the transaction it received from the client and broadcasts PrePrepare messages to all other replicas.
- A replica detects that the transaction has been modified because of a digest mismatch. It detects primary failure and broadcasts a ViewChange request, triggering all other good replicas to agree.
- Replica 2 is elected the next primary, and the transaction is correctly proposed.
- Primary 3 elected – modifies the transaction it received from the client and broadcasts PrePrepare messages to all other replicas.
- A replica detects that the transaction has been modified because of a digest mismatch. It detects primary failure and broadcasts a ViewChange request, triggering all other good replicas to agree.
- Replica 0 is elected the next primary, and the transaction is correctly proposed.

Scenario 6: Bad transaction results ($< n/3$ bad)
- **Inputs**: 4 total replicas, 1 Byzantine replica (Replica 1), 1 transaction, "Bad transaction results" Byzantine behavior.
- Primary 0 elected – processes the transaction it received from the client and broadcasts PrePrepare messages to all other replicas.
- Same behavior as normal case, until Inform phase. Replica 1 sends a result that is inconsistent with the other replicas. The client ignores Replica 1 and takes the other identical results as the consensus.

Scenario 7: Bad transaction results ($\geq n/3$ bad)
- **Inputs**: 4 total replicas, 2 Byzantine replica (Replica 1, Replica 3), 1 transaction, "None" Byzantine behavior.
- Primary 0 elected – processes the transaction it received from the client and broadcasts PrePrepare messages to all other replicas.
- Same behavior as normal case, until Inform phase. Replicas 1 and 3 send a result that is inconsistent with the other replicas. If the client receives Replicas 1 and 3's results before the others, it will reach the threshold of g corroborating Inform messages and accept this result. However, this situation may vary depending on a race condition (unless the majority of replicas is malicious).

Scenario 8: Bad view change requests ($< n/3$ bad)
- **Inputs**: 4 total replicas, 1 Byzantine replica (Replica 1), 1 transaction, "None" Byzantine behavior.
- Primary 0 elected – processes the transaction it received from the client and broadcasts PrePrepare messages to all other replicas.
- Replica 1 spams a ViewChange message but is ignored because all other replicas have successfully received the PrePrepare message. pBFT proceeds normally.

Scenario 9: Bad view change requests ($\geq n/3$ bad)
- **Inputs**: 4 total replicas, 2 Byzantine replicas (Replica 1, Replica 3), 1 transaction, "None" Byzantine behavior.
- Primary 0 elected – processes the transaction it received from the client and broadcasts PrePrepare messages to all other replicas.
- Replicas 1 and 3 spam ViewChange messages but are ignored by the other replicas which have received the PrePrepare message. pBFT proceeds normally. If Replicas 1 and 3 halt the commit algorithm and become nonresponsive, pBFT will hang (but this is not currently implemented in the simulation).

6 Future Work

The current visualization is a rudimentary model of pBFT's structure and its robustness to nondeterminism in message ordering and node / primary Byzantine behavior. While it is a

relatively good approximation of normal case and simple malicious handling / response, there are many identifiable points of improvement:

- Adding support for replica-specific logs. At the moment, the simulation enforces separation between sequential phases and transactions i.e. messages (unless view change / new view) will generally not be interspersed because communication queues are cleared between phases. However, this simplification reduces throughput and isn't representative enough of pBFT's possible multiprocessing.
- More descriptive error messages for incorrect combinations of input options in the frontend form. Currently, all errors return the same message, so it's difficult to fix a validation failure.
- Support more varieties of Byzantine behavior: e.g. convincing good replicas that other good replicas are malicious.
- Support combinations of Byzantine behavior: e.g. one replica is "No response" while another malicious primary is "bad client transactions"
- Fix processing for new view requests (currently hardcoded to one replica max able to send this type of message until primary resolution).
- Link the consensus bank to the SQL database instead of maintaining separate dictionaries for balance data per replica.

The simulation framework created in this project can be modified and streamlined to use cases of greater scope e.g. modeling sharded systems, larger distributed attacks like Sybil, etc [4]. For now, it is a start towards combining data visualization in the frontend with distributed systems concepts in the backend to showcase a protocol's robustness to malicious attacks on message authenticity and integrity.

7 Bibliography

[1] M. Castro and B. Liskov, "Practical byzantine fault tolerance", 1999.

[2] J. Hellings and M. Sadoghi, "Byshard: Sharding in a byzantine environment", *Proc. VLDB Endow.*, vol. 14, no. 11, pp. 2230–2243, Jul. 2021, ISSN: 2150-8097. DOI: 10.14778/3476249.3476275. [Online]. Available: https://doi.org/10.14778/3476249.3476275.

[3] S. Gupta, J. Hellings, and M. Sadoghi. 2021.

[4] T. Rajab, M. H. Manshaei, M. Dakhilalian, M. Jadliwala, and M. A. Rahman, "On the feasibility of sybil attacks in shard-based permissionless blockchains", *CoRR*, vol. abs/2002.06531, 2020. arXiv: 2002.06531. [Online]. Available: https://arxiv.org/abs/2002.06531.