CS2910 Project Report: Hotstuff Under Attack

Birju Patel
School of Computing and Information
University of Pittsburgh
Pittsburgh, United States
bsp22@pitt.edu

***Abstract*—I evaluated the performance of a byzantine fault tolerant replication system that implemented the Hotstuff consensus protocol. I deployed the system under real world conditions and subjected it to various attack scenarios. I then measured the impact of those attacks on the latency that clients connected to the system experienced. I found that the system was vulnerable to performance based attacks, and that an adversary could create wild swings in latency.**

I. Introduction

This project was completed for the Resilient Systems and Societies Lab at the University of Pittsburgh under the supervision of Professor Amy Babay. The lab conducts research on distributed systems and focuses on designing infrastructure capable of tolerating failures and attacks. The lab is working on a project to make the supervisory control and data acquisition (SCADA) systems used to control power grid infrastructure more resilient. To accomplish this, the lab uses byzantine fault tolerant (BFT) replication.

To handle f byzantine failures and k crash failures, a BFT replication algorithm requires a total of $3f + 2k + 1$ total replicas in a committee. These n replicas are placed across S sites, and the sites are connected by a network. An adversary can attack the network and isolate a single site, preventing it from communicating with the other sites. They can cause a total of k replicas to either crash or become unreachable. They also have control of f byzantine replicas that may fail in arbitrary ways.

Since SCADA systems are critical infrastructure, they have a tight latency constraint. The amount of latency any client connected to the system experiences cannot exceed an upper bound. Traditional BFT algorithms are vulnerable to leader based latency attacks. In such an attack, a byzantine leader does not immediately forward client requests to the rest of the system, but instead waits for some time and then forwards them. Because the algorithm is driven by the leader's actions, it is possible for the faulty leader to significantly slow the system. To solve this problem, the lab uses Prime, a BFT algorithm that uses leader monitoring. In Prime, each replica monitors network conditions and the performance of the leader. If they detect malicious behavior, they forcibly change the leader.

Since the release of Prime in 2010, many advances have been made in BFT replication. I selected a modern BFT system that uses two new technologies. It uses a variant of the Hotstuff consensus protocol for ordering. It also uses a mempool to disseminate blocks of transactions across the network. These improvements allow the system to achieve a much higher throughput than previous BFT systems. For my project, I sought to determine if a this Hotstuff based system was suitable for the lab's SCADA application.

II. System Design

A. *Hotstuff*

The system uses Hotstuff for the consensus layer. Hotstuff has four distinguishing features; three phase commit, linear view change, leader rotation after every round, and blockchain based.

Each Hotstuff replica begins with a chain consisting of a genesis block. When a replica becomes a leader, it gets to create a block that extends its current longest chain. They then broadcast this block to the rest of the network. When a replica receives this block, they determine if it extends their current longest chain. If so, they send a signed vote back to the leader. Once the leader receives a quorum of $2f+1$ votes, they create a quorum certificate for that block and broadcast the confirmed block to the rest of the network. After a replica receives the confirmed block, it adds it to its chain. Now that this round is done, it increments its view number. In Hotstuff, the leader changes with each view, and the leader is determined by the view number. The replica takes its newest confirmed block and sends it to the next leader. The new leader then waits for $2f + 1$ blocks and chooses one of them to extend.

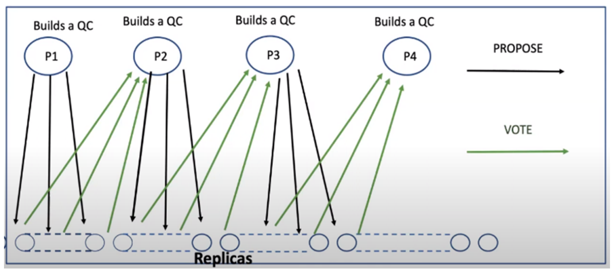


Fig. 1. Hotstuff View Change

Notice that the view change is integrated into the normal operation of the protocol. In traditional BFT algorithms like

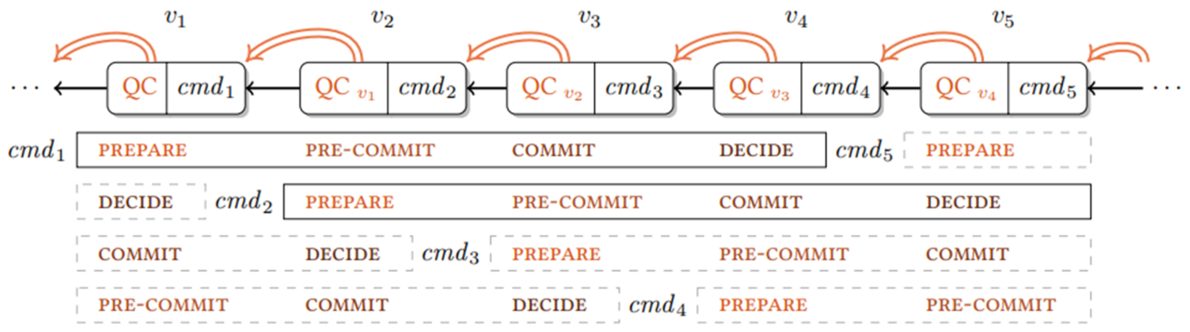


Fig. 2. Pipelined 3 Chain Hotstuff

PBFT, a view change is triggered when a quorum of replicas time out and send view change votes to the new leader. The new leader then batches those votes and sends a new view certificate. This results in a total of $O(n^2)$ total votes per view change. In Hotstuff, since a view change is integrated into the first part of the round, only $O(n)$ messages are sent, creating a linear view change. This structure of the round allows each round to be pipelined. We commit a block once it has been extended by 3 other blocks in the chain. The addition of a particular block to the chain serves as the propose phase for itself, the prepare phase for the block below it, the pre-commit phase for the block 2 blocks under it, and the commit phase for the block 3 blocks below it. This allows a new block of transactions to be added to the chain at every round.

B. *Mempool*

In standard BFT systems, when a leader sends a proposal, it also sends the data that it wants to order with it. Normally, this data is a batch of transactions that the leader wants the system to execute next. However, as the batches become large, this does not scale. The mempool takes over the job of disseminating the batches of transactions so the consensus algorithm can just deal with the ordering. The mempool creates a block of transactions and then creates a hash of that block. That hash is a unique identifier for that block. It then sends the block to the rest of the network and sends the hash to the consensus module. The consensus module now sees an unordered series of hashes. The module then runs the consensus protocol using these hashes as the payload of their proposals, rather than the actual block of transactions. When the system reaches a consensus on a particular hash, it asks the mempool module for the block corresponding with that hash. The mempool then retrieves that block, so the system can now execute all of the transactions contained within it. This optimization allows consensus messages to remain small, which improves the performance of the consensus algorithm.

Since the payload on each consensus proposal is only a small hash, a leader can order multiple blocks in a single round of the protocol by putting multiple block's hashes in its payload. For instance, if there is a buildup of unconfirmed transactions because the leader of the previous round was down, the new leader could pick up the slack by increasing the number of hashes it puts in its proposal message. This keeps the total throughput of the system relatively constant.

III. Experimental Setup

The system's original code can be found here.

A. *Simulating the SCADA Application*

I designed the experiment to mirror the structure of the SCADA application. In a particular configuration of the SCADA application, 3 Prime replicas were deployed at each site, with 4 sites in total. In two of the sites, there are clients which send transactions to the replicas at that site. To simulate this, I used 4 machines in the Pitt aster cluster. Each machine represented a site, so I deployed 3 nodes on each machine, and deployed 3 clients on two of those machines. I added 2.5ms of latency between machines to simulate the real delay between geographically separated sites. In the real SCADA application, there is 2.5ms of latency between clients and the rest of the system, but I did not simulate this.

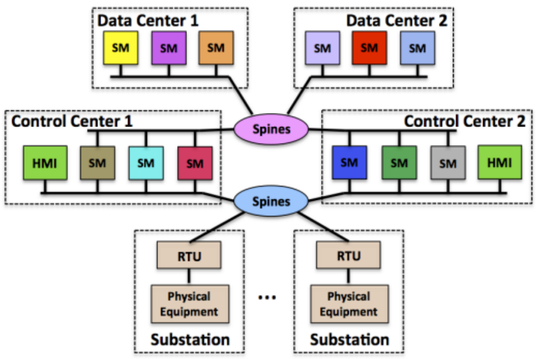


Fig. 3. Simulated 3+3+3+3 Configuration

The committee must support a single site going down (equivalent to 3 failed replicas), 1 byzantine failure ($f = 1$), and 1 additional crash failure (a total of $k = 4$). Using the formula $3f + 2k + 1$, I find I need 12 total replicas, which is what I have. Generally, the required quorum size to commit a block can be found using $q = \left\lfloor \frac{n+f}{2} \right\rfloor + 1$, where n is the total number of replicas and f is the total number of byzantine replicas. In this configuration, a replica needs 7 votes to move forward. This is true because the formula makes a distinction between crash faults and byzantine faults. This is configurable in Prime, but standard BFT systems do not make such a distinction. The system assumes that all faults are byzantine faults, so it uses the formulas $n = 3f + 1$ and $q = 2f + 1$ to calculate the minimum quorum size, resulting in $q = 9$ for this configuration. This quorum size is too large, because it cannot be fulfilled if 4 crash faults occur, yet this scenario needs to be supported.

To account for this, I modified the committee file to include a field named ”byz_faults”. This field specified the total number of byzantine faulty replicas the user wants to tolerate in this committee. I then modified the code to calculate the quorum sizes using the generalized formula and the value given for ”byz_faults”.

B. Deployment

The following command is used to start the node. Each node requires a key file, a parameter file, a committee file, and a storage directory. The key file is the private and public key for this node. The parameter file specifies the configurable settings like the block size and view timeout. The committee file contains the public keys, IP addresses, and ports of the other nodes in the committee. The storage directory is the name of the folder where this node will save its state. The node is run in verbose mode (-vvv) in order to get all the information needed for data analysis. The printed output is then sent to a log file.

```
./node -vvv run –keys [] –parameters [] –committee [] –store []
```

The following command is used to start the client. Each client requires a timeout, a size, a rate, and an IP address. The timeout is the amount of time a client will wait before giving up on a particular transaction. The size is the size in bytes of each transaction. The rate is the number of transactions per second the client will send. The IP address is the place where the transactions are sent. This value is found in the committee file. The output of this program is also sent to a log file.

```
./client –timeout [] –size [] –rate [] []
```

To automate the experiments, I used Ansible, a tool that uses ssh to run commands automatically on target machines. I created 5 Ansible playbooks, start_experiment, kill_experiment, add_latency, remove_latency, and disconnect_site. start_experiment starts all of the nodes and clients on all the machines and pipes their output to a series of log files. kill_experiment kills all running nodes and clients. add_latency and remove_latency use the Linux netem tool to add and remove 2.5ms of simulated delay between machines. disconnect_site uses netem to create a filter that drops all packets being sent to and from a single machine.

C. Data Collection

The original code contained a set of Python scripts for data analysis. The scripts parsed the logs generated by the nodes and clients during their run and created summary statistics for the run showing the average throughput and average latency. Two types of latency data was measured. First, the difference between the time a particular block was proposed by a replica and the time it was accepted by a replica was measured as consensus latency. This is the latency of the protocol itself. Then, the difference between the time a particular transaction was proposed and the time that a block containing that transaction was committed was measured as end to end latency. This is the latency as it was experienced by the client. I modified the original scripts so that these values would be saved as a time series. The script now created an array of tuples, with each tuple containing the time the transaction or block was sent, the latency experienced by that transaction or block, and the id of the block or transaction. Once an experiment was over, I ran the following commands inside of a Python shell to generate this data. I saved the summary statistics, the 2 csv files, and the logs for each run.

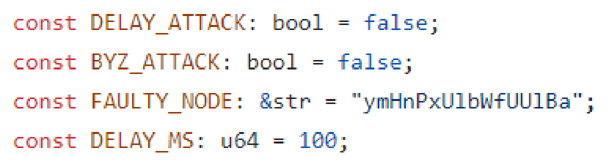
- from benchmark.logs import LogParser
- l = LogParser.process(path_to_log_dir, faults=[])
- print(l.result())
- x = l.consensus_latency_time_series()
- y = l.end_to_end_latency_time_series()
- LogParser.write_csv(fname=”consensus.csv”, data=x)
- LogParser.write_csv(fname=”end_to_end.csv”, data=y)

D. Test Scenarios

I assume that the adversary is capable of disconnecting 1 site from the rest of the network, has control over 1 byzantine faulty replica, and can cause 1 other replica to crash and restart. I tested the performance of the system under 5 different scenarios. The first scenario was a control, and the system would run normally with no interference. In the second scenario, a single replica would be killed and then restarted during the run. In the third scenario, the network would be attacked such that a single site would temporarily lose the ability to communicate with the rest of the system. In the fourth scenario, a single site would be disconnected and reconnected, and two additional replicas located in another site would be killed and then restarted. In the fifth scenario, a single byzantine replica would attempt to execute a delay attack whenever it became the leader. In the sixth scenario, a single byzantine replica would send conflicting proposals to different parts of the system, sending a legitimate block to half of the nodes and an illegitimate block to the other half of the nodes whenever it became the leader of a round. I conducted three experiments for each scenario.

E. Byzantine Attacks

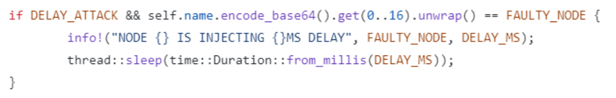
To simulate a delay attack and a fork attack I modified the code. The file consensus/src/proposer.rs contains the logic for proposing a new block when the node becomes the leader in a particular view. At the top of this file there are flags that I used to specify if an attack was running. To run an attack, I switched the corresponding flags to true and recompiled the code.



```
const DELAY_ATTACK: bool = false;
const BYZ_ATTACK: bool = false;
const FAULTY_NODE: &str = "ymHnPxUlbWfUUlBa";
const DELAY_MS: u64 = 100;
```

Fig. 4. Attack Flags

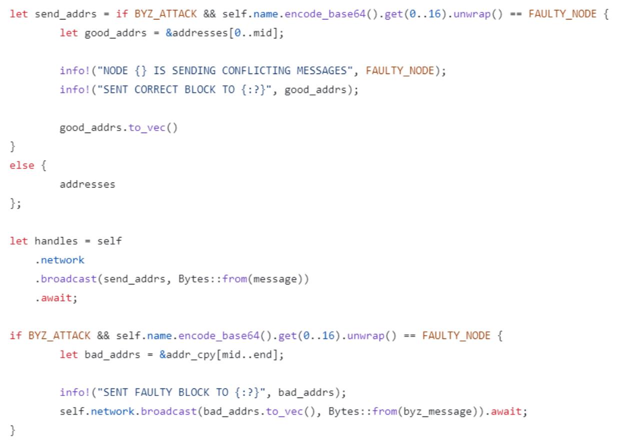
The function make_block is called each time it is a particular node's turn to propose a block. To simulate the delay attack, I added the following segment of code. When the DELAY_ATTACK flag is set to true and the node id is FAULTY_NODE, the node waits for DELAY_MS milliseconds before continuing the proposal.



```
if DELAY_ATTACK && self.name.encode_base64().get(0..16).unwrap() == FAULTY_NODE {
        info!("NODE {} IS INJECTING {}MS DELAY", FAULTY_NODE, DELAY_MS);
        thread::sleep(time::Duration::from_millis(DELAY_MS));
}
```

Fig. 5. Delay Attack

To simulate a fork attack, I added the following segment of code. When the BYZ_ATTACK flag is set to true and the node id is FAULTY_NODE, the node splits the committee in two. The first half of the committee is sent a legitimate proposal, while the second half of the committee is sent an corrupted proposal.



```
let send_addrs = if BYZ_ATTACK && self.name.encode_base64().get(0..16).unwrap() == FAULTY_NODE {
        let good_addrs = &addresses[0..mid];

        info!("NODE {} IS SENDING CONFLICTING MESSAGES", FAULTY_NODE);
        info!("SENT CORRECT BLOCK TO {:?}", good_addrs);

        good_addrs.to_vec()
}
else {
        addresses
};

let handles = self
    .network
    .broadcast(send_addrs, Bytes::from(message))
    .await;

if BYZ_ATTACK && self.name.encode_base64().get(0..16).unwrap() == FAULTY_NODE {
        let bad_addrs = &addr_cpy[mid..end];

        info!("SENT FAULTY BLOCK TO {:?}", bad_addrs);
        self.network.broadcast(bad_addrs.to_vec(), Bytes::from(byz_message)).await;
}
```

Fig. 6. Fork Attack

IV. Data Analysis

A. Control

In the control experiment, I let the nodes and clients run without any interference for approximately 1 minute. Normally, the end to end latency remained low and had little variance. However, there was an anomaly where the latency spiked.

Upon further investigation, I discovered that these anomalies were caused by a particular node timing out. When a node becomes a leader, it must propose a block before its view timer

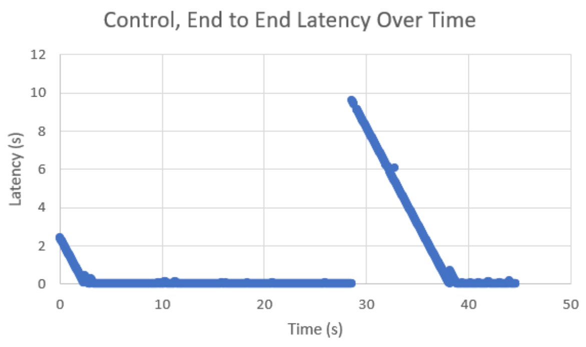


Fig. 7. Control Experiment

expires, or else the other nodes will initiate a view change and nominate another leader. By examining the logs, I discovered that a timeout occurred at the same time that the anomaly was measured. This timeout seems to have been caused by an error with the mempool. As discussed previously, the mempool disseminates blocks of transactions across the network and provides hashes of those blocks to the consensus layer. The consensus layer then votes on proposals which only contain those hashes. It is possible for the consensus layer to move faster than the mempool, and for a node to commit a block that it does not have the transactions for. This causes an error and the node is forced to wait until its mempool syncs with the others. This then causes a timeout. This is what created the anomaly at the 30 second mark.

During the anomalous period, the latency falls in a linear manner. This is due to the effect of batching. When the next leader takes over after the timeout, it aggregates all of the unordered transactions into a large batch. This large batch is committed all at once, after the end of the anomalous period. Transactions are entered into the system at a constant rate, so some were entered at the beginning of the anomaly, while some were entered at the very end. However, they are all committed at once. This creates the linear pattern. During the normal period, end to end latency varies little. A graph of the performance during the stable period shows that latency is normally bounded between 20ms and 40ms.

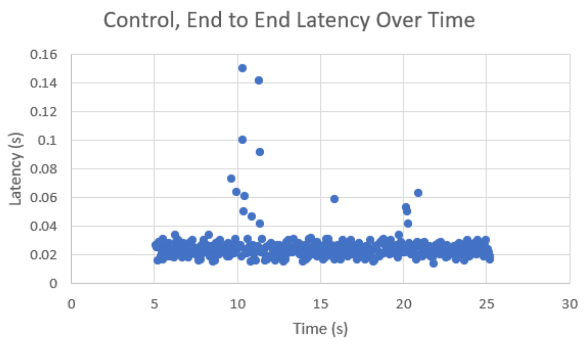


Fig. 8. Control Experiment Between seconds 5 and 25

B. Kill 1

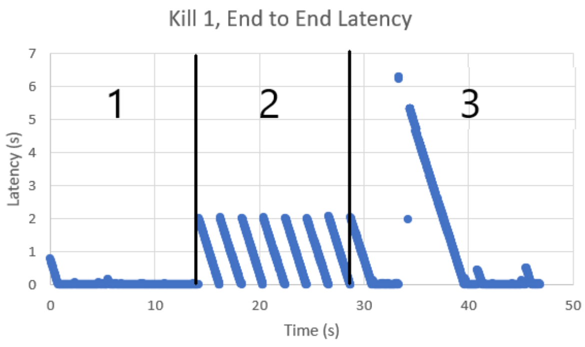


Fig. 9. Kill 1 Node

To conduct this experiment, I first ran all clients and nodes, and then I manually killed a node. I then waited and restarted that node. This experiment has 3 phases, which is depicted in the graph. In the first phase, the client experiences low and stable latency.

In the second phase, the latency follows a consistent pattern of spiking, then falling, and then spiking again. This corresponds to the portion of the experiment when the killed replica is down. Hotstuff rotates leaders in a round robin manner. This means that at the beginning of each round, a new leader is elected. Even though a node is down, it will still be appointed as leader periodically. However, since that node is down, the other replicas will hear nothing, and eventually will vote for a new leader when the view timeout expires. This time is wasted, and in this time, transactions are allowed to build up. This creates the saw tooth pattern of latency. There is currently no method implemented for determining if a node is down and excluding it from the committee. Such a mechanism would provide a considerable performance boost.

In the third phase, the downed node is brought back up again, and it recovers its state from neighboring nodes. While the nodes are getting synchronized, there is a temporary spike in latency. When synchronization is complete, latency returns to normal levels.

C. Site Isolation

The site isolation experiment is similar to the previous scenario. In this experiment, I create a filter to drop all traffic going to and from a single machine in the cluster. I then remove the filter after some period of time. This is meant to simulate an attack on the network connecting different sites.

The latency graph has a similar three phase pattern as the previous experiment. However, since 3 nodes are being taken offline instead of 1, the effect on latency is greater. It also takes more time for the system to recover.

D. Site Isolation and Kill 2

In this experiment, I started up all nodes and clients. Then I added the filter to isolate a single machine from the rest of the cluster. I then killed 2 nodes that were running on another

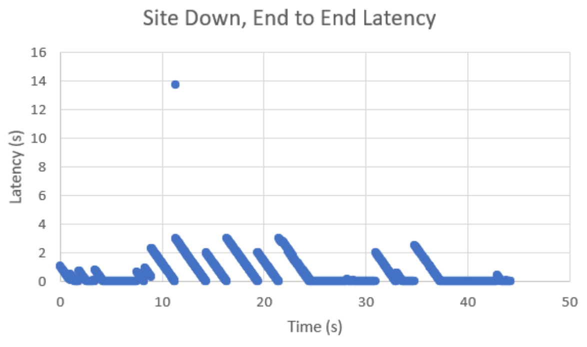


Fig. 10. Site Isolated

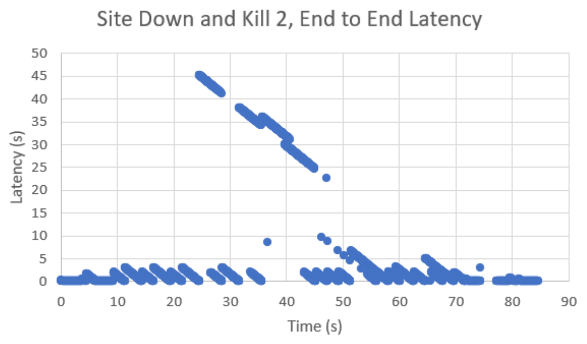


Fig. 11. Site Isolated and 2 Killed

machine. I then turned those 2 nodes back on and reconnected the isolated site. Again, the graph exhibited the same 3 phase pattern, but this time with even higher latency and a longer time to recover.

E. Delay Attack

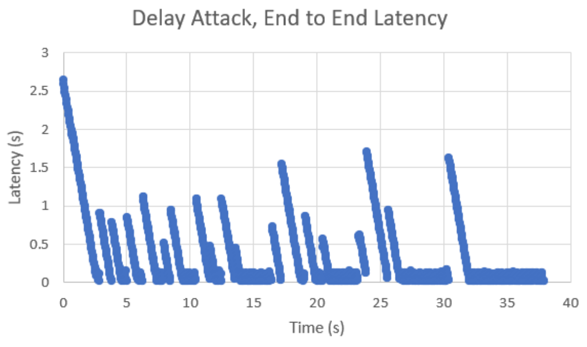


Fig. 12. Delay Attack

In this experiment, I set the DELAY_ATTACK flag to true and recompiled the code. As it ran, a single node was programmed to inject 500ms of delay each time it became the leader. The latency graph shows a temporary spike in latency, followed by a sharp drop and a period of low latency. The spikes correspond to times when the faulty node is appointed leader.

Hotstuff is a chained protocol. The current leader's proposal serves as the accept phase for the previous leader's proposal and the commit phase for block proposed before that. A delay in proposing a block does not just delay the commit of that particular block, but also the commit of the previous two blocks. In the graph, this can be seen as a large spike in latency is followed by a smaller spike in latency directly afterwards.

F. Fork Attack

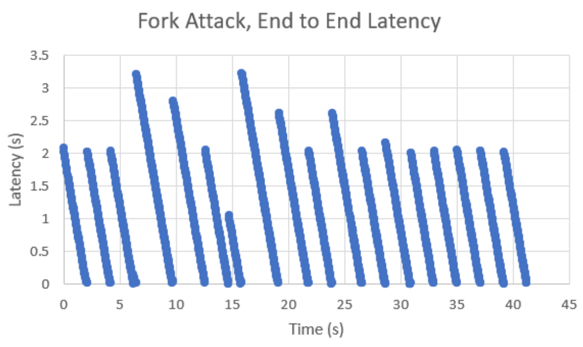


Fig. 13. Fork Attack

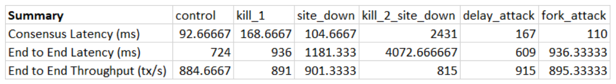
In this experiment, I set the BYZ_ATTACK flag to true and recompiled the code. As it ran, a single node was programmed to send a correct proposal to half of the nodes and a bogus proposal to the other half when it became leader. A consensus would never be reached, and eventually the node would time out and the next would be elected leader.

From a performance standpoint, this is equivalent to that node having done nothing at all. This is exactly what we observe. Whenever the faulty node becomes leader, it wastes time and allows transactions to build up until it times out. This creates a temporary spikes latency.

V. Conclusion

The average latency of the system fluctuated wildly under the various test scenarios. Under the normal case, clients experienced an average of 724ms from the time they submitted transactions to the time those transactions were committed. Under the worst case, when a total of 5 nodes were either down or isolated, average end to end latency increased to 2.43s. Across all scenarios, the throughput stayed relatively consistent. This is because of the mempool, which shares transactions across the network so that each node could see the transactions sent to every other node. Using the technique discussed previously, the live nodes pick up the slack by including the transactions that would have been proposed by the down nodes in their proposals.

In conclusion, Hotstuff based systems are not suitable for low latency SCADA applications. The round robin leadership structure of the protocol requires that down or compromised replicas periodically become the leader. This negatively impacts performance because it allows transactions to build up



Summary	control	kill_1	site_down	kill_2_site_down	delay_attack	fork_attack
Consensus Latency (ms)	92.66667	168.6667	104.6667	2431	167	110
End to End Latency (ms)	724	936	1181.333	4072.666667	609	936.33333
End to End Throughput (tx/s)	884.6667	891	901.3333	815	915	895.33333

Fig. 14. Average Throughput and Latency

and creates periodical spike in latency. The introduction of a mempool is useful for scaling up throughput. However, the mempool layer implemented in the evaluated system was buggy and caused unpredictable spikes in latency even under ideal conditions.

In ideal conditions, the system does provide an improvement over existing BFT systems in both latency and throughput, but performance quickly degrades under real world conditions. The system could be improved by adding a leader reputation mechanism that monitors each node's performance as leader and evicts them from the group if they have crashed or if they are suspected of being malicious.

References

[1] Babay, A., Tantillo, T., Aron, T., Platania, M., Amir, Y. (2018). Network-attack-resilient intrusion-tolerant SCADA for the Power Grid. 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). https://doi.org/10.1109/dsn.2018.00036

[2] Yin, M., Malkhi, D., Reiter, M. K., Gueta, G. G., Abraham, I. (2019). Hotstuff. Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing. https://doi.org/10.1145/3293611.3331591

[3] Gelashvili, R., Kokoris-Kogias, L., Sonnino, A., Spiegelman, A., Xiang, Z. (2022). Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback. Financial Cryptography and Data Security, 296–315. https://doi.org/10.1007/978-3-031-18283-9_14