System Evaluation: Testing BFT-SMaRt

Robert Fishel
School of Computing and Information
University of Pittsburgh
Pittsburgh, United States
rmf105@pitt.edu

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

***Abstract*—We tested the performance and security of BFT-SMaRt, one of the most popular implementations of byzantine fault tolerant replication. We deployed BFT-SMaRt on 4 nodes, and ran a simple replicated application on top of it. We assumed that we could upload malicious code to 1 node, so that it may fail arbitrarily. We began by testing the performance of the system under normal conditions. We then attempted to craft exploits to force the system to violate its consistency and availability guarantees. We succeeded in violating consistency by forcing 2 nodes to disagree on the state of the application. In certain cases, we were also able to violate availability by severely degraded system throughput.**

I. Introduction

A byzantine fault tolerant system is a system that is capable of tolerating f of its $3f+1$ nodes failing in arbitrary ways. This includes crash faults, lost messages, and inconsistent messages being sent to different nodes. BFT-SMaRt guarantees that the system will remain consistent and available, even in the face of f byzantine failures. The consistency constraint guarantees that all nodes in the system that are not compromised should have the same state. That is, they all execute the same operations in the same order. The availability constraint guarantees that the system must eventually make progress. An operation submitted by a client must eventually be executed on the system, and may not block for a long period of time.

BFT-SMaRt was one of the first practical systems that implemented byzantine fault tolerant replication. It is written in Java, and has a simple interface developers can use to build replicated applications. The implementation is split into several pieces. At the core of the system is a leader based consensus protocol to orders updates. To execute a new operation, a client sends a message to every node in the system. The leader node collects a batch of client messages, and begins a 3 phase consensus protocol to execute those updates across the system. In the *PROPOSE* phase, the leader signs the batch of operations and forwards it to the other nodes. When a node receives the signed message, it enters the *WRITE* phase. It verifies its signature is correct and sends a copy with its own signature to all other nodes. When a node receives $2f + 1$ signed messages from the other nodes for a single value, it knows the system has reached a consensus on a single value. At this point, the node transitions to the *ACCEPT* phase and sends the batch of signed messages to all other nodes. When a node collects $2f + 1$ *ACCEPT* messages, it commits the operation and sends its result back to the client. See figure 1 for an illustration of this algorithm.

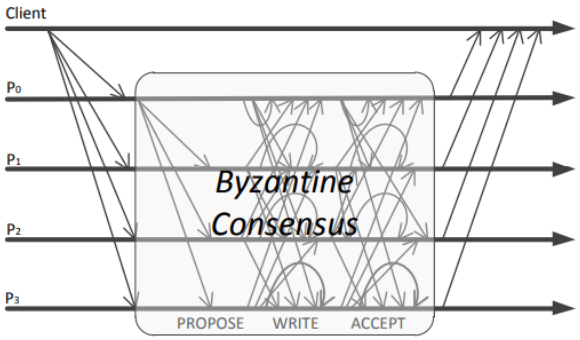

Fig. 1. BFT-SMaRt consensus protocol

When a leader goes down, another node is elected as the leader and everyone updates their state to match that node. A leader may crash, or it may be evicted by the group after it is recognized as faulty. When a node crashes and recovers, it must get the most updated state of the system from a node with a more updated view. This is accomplished through a state transfer protocol. Periodically, each node takes a snapshot of its state and writes it to disc. When requested by another node, that node sends this snapshot, as well as all operations it has executed since that snapshot was taken.

We want to test the security and performance of BFT-SMaRt. We deployed a BFT-SMaRt system containing 4 nodes on a remote GENI cluster. We run a simple application on top of the system. This application stores the value of the counter on the server, and allows clients to submit two types of operations, read operations and increment operations. The read operation returns the value of the counter, and the increment operation increments the counter by an inputted value. To test security, we assume we have control of the leader node, and can upload a modified copy of the BFT-SMaRt code in which we have added a vulnerability. We have three versions of the BFT-SMaRt code, one that is secure, one that contains an exploit to attacks consistency, and one that contains an exploit to attack availability. To test performance, we create a

python script that spawns multiple clients in separate threads and concurrently sends increment operations to the system. We measure the amount of time that each operation takes to execute on the system (latency) and the number of operations the client can execute per second (throughput). We measure these variables under a number of different circumstances.

Our exploit to break the consistency constraint was successful. We were able to produce a situation where 2 nodes disagreed on the final value of the counter. This was an unexpected result, as BFT-SMaRt is supposed to handle this type of attack. Our exploit to break the availability constraint succeeded when only one client was connected to the system. In this case, throughput of the system dropped consistently until the time it took to execute the client's next operation became unreasonably long. However, when there were multiple clients connected to the system, the other nodes recognized that the leader was faulty, ejected it from the group, and elected a non-faulty leader. In this case, availability was maintained.

II. Experimental Setup

Our first step in completing the experiments we desired was establishing the testing environment we would be determining and establishing the specifications of the environment that we would be running the application in. We used 4 virtual machines (which will henceforth be referred to as nodes) in 1 GENI cluster at the same site, where we would run one instance of the BFT-SMaRt server in each node. The standard BFT-SMaRt protocol code (pulled from the zip file here) was loaded onto 3 of the nodes (labeled 1 thru 3). The malicious code, which we will outline in greater detail as this section progresses, was uploaded to node 0.

For the code loaded onto each of these nodes, we needed to make some minor modifications to the configuration variables associated them. The config folder of the BFT-SMaRt source code contained two important configuration files: config/system.config and config/hosts.config.

In system.config, the batch size variable (maxbatchsize) was initially set to 400 - meaning 400 requests were being pooled together and sent to the system at the same time. We wanted to see how the system handled 1 message at a time, so we reduced this configuration variable to 1. We also made sure to set the variable representing the number of servers in the system (n) equal to 4, 1 for each node.

The file hosts.config (Fig. 2) contains server id-host name-port mappings for each of the nodes in the system. For each of the system code blocks we uploaded to each node, we made sure that this file had the correct server host names associated with each node. That is, for server with id 0, node 0's local IP was used as the host name, for id 1, node 1's local IP was used, etc. The port number for each server in the system was selected arbitrarily. We had to ensure that all hosts.config files contained the same hosts and ports across all 4 nodes.

With a modification to these files, for the changes to accurately take effect, before running an application on the protocol again, config/currentView needs to be deleted. It will

```
# This file defines the replicas ids, IPs and ports.
# It is used by the replicas and clients to find connection info
# to the initial replicas.
# The ports defined here are the ports used by clients to communicate
# with the replicas. Additional connections are opened by replicas to
# communicate with each other. This additional connection is opened in the
# next port defined here. For an example, consider the line "0 127.0.0.1 11000".
# That means that clients will open a communication channel to replica 0 in
# IP 127.0.0.1 and port 11000. On startup, replicas with id different than 0
# will open a communication channel to replica 0 in port 11001.
# The same holds for replicas 1, 2, 3 ... N.

#server id, address and  port (the ids from 0 to n-1 are the service replicas)
0 10.10.5.2 11000
1 10.10.3.2 11010
2 10.10.2.1 11020
3 10.10.5.1 11030
```

Fig. 2. Sample hosts.config file

be re-compiled upon a server or client program being run with the new configuration under consideration.

As far as the application built on top of BFT-SMaRt will be using, it is one of the demo programs provided along with the BFT-SMaRt code. The program simply keeps track of an integer counting variable between the n servers, with the ability for clients to send requests customizing increment amounts that the counter should increase by for every increment call, and the number of iterations the client would like to request. For example, if the increment is 2, and the client sends 1,000 requests, we would anticipate the counter variable in the system to end up as 2,000.

The counter program is built in a straightforward manner. It is made up of 2 Java classes: **CounterClient.java** and **CounterServer.java**. (counter application explanation) The client file is ran with three command line inputs in this format:

./runscripts/smartrun.sh bftsmart.demo.counter.CounterClient *id inc iter*

where *id* is the client id number, *inc* is the value of the increment we send to the system, and *iter* is the number of iterations this client should send. The script we are running is built into the BFT-SMaRt code to run a Java file from one of the demo programs. Similarly, the server is ran with one command line input representing the server id from the hosts.config file:

./runscripts/smartrun.sh bftmsart.demo.counter.CounterServer *id*

To accommodate multiple clients sending requests simultaneously, we built a python script which used python's subprocess library to call the client program simultaneously in several different threads. This python library also keeps track of metrics such as latency and throughput, and passes them into csv files so that we could organize and parse the data neatly into excel spreadsheets to visualize the results. The python script, named **script.py**, is run as follows:

python3 script.py *numClients increment iterations*

Here is a description of the relevant code we wrote, both in the faulty version of the BFT-SMaRt protocol, and the multi client python script.

A. Faulty Message Injection Code

After exploration of the BFT-SMaRt code and learning more about how the service worked, we realized that the consensus message sent in between nodes which come to a decision on some primitive that necessitates conesnsus (in this case, the increment value) was broadcasted by each node in the **ServersCommunicationLayer.java** file. Here, we went to the corresponding send() message and added code which established a faulty version of a ConsensusMessage object (Fig. 3).

```
// Faulty Message Injection Code
if (FAULTY_MSG_INJECTION) {
    if (me == 0 && sm instanceof ConsensusMessage) {
        ConsensusMessage cm = (ConsensusMessage) sm;
        if (cm.getPaxosVerboseType().equals(anObject:"PROPOSE") && cm.getValue() != null) {
            byte[] oldBytes = cm.getValue();
            byte[] bytes = new byte[oldBytes.length];
            for (int i = 0; i < oldBytes.length; i++) {
                bytes[i] = oldBytes[i];
            }
            bytes[63] = 2;
            faultyMessage = new ConsensusMessage(
                // int paxosType, int id,int epoch,int from, byte[] value
                MessageFactory.PROPOSE,
                cm.getNumber(),
                cm.getEpoch(),
                cm.getSender(),
                bytes
            );
            try {
                new ObjectOutputStream(bOut).writeObject(faultyMessage);
            } catch (IOException ex) {
                Logger.getLogger(ServerConnection.class.getName()).log(Level.SEVERE, msg:null, ex);
            }
            faultyData = bOut.toByteArray();
        }
    }
}
```

Fig. 3. Faulty ConsensusMessage Creation

This consensus message was only created under several conditions: First, a global boolean variable *FAULTY_MSG_INJECTION* must have been set to true (in order to easily turn the injection on/off). If faulty message injection was allowed, the way we designed it was that the message could only be created if the current node ID is 0, because we only wanted node 0 to send the faulty message to one other node. There were also different types of messages being sent here, so we specified that only consensus messages were to be tampered with so that other parts of the code were not modified. This part of the code, where this newly constructed message was packaged and sent, only occurred if the destination node id was 2. So the message both had to have been created in this chunk (which required the sending node id to be 0), and the destination node had to be 2 (Fig. 4).

```
if (i == 2 && faultyData != null) {
    System.out.println("** SENDING FAULTY INCREMENT TO NODE " + i + " **");
    getConnection(i).send(faultyData, useMAC);
} else {
    if (sm instanceof ConsensusMessage) {
        if (((ConsensusMessage) sm).getPaxosVerboseType().equals(anObject:"PROPOSE")) {
            System.out.println("** SENDING LEGIT INCREMENT TO NODE " + i + " **");
        }
    }
    getConnection(i).send(data, useMAC);
}
```

Fig. 4. Injection code only takes place if destination node is 2

B. Latency Injection Code

Similarly, we have built in latency injection code when a message is sent in textbfServersCommunicationLayer.java. Essentially, whenever the leader node (which will initially be 0 unless it is kicked out) sends a consensus message to all three servers, the time between that message and the next one will take an extra *LEADER_DELAY_INCREMENT* ms (global variable for our purposes was set to 10). Similarly to the faulty message injection failure, in order to easily turn this fault on or off, the *LEADER_DELAY_INJECTION* must be set to true for the injection to occur. The full code associated with this failure can be found in Figure 5.

```
// Incremental Latency Attack Code
if (LEADER_DELAY_INJECTION) {
    try {
        if (me == 0 && sm instanceof ConsensusMessage) {
            System.out.println("**CURRENT DELAY " + LEADER_DELAY + " **");
            Thread.sleep(LEADER_DELAY);
            LEADER_DELAY += LEADER_DELAY_INCREMENT;
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

Fig. 5. Latency Injection Code that occurs before a send

C. Latency

To measure system latency, we started timers before and after an increment request was sent in the counter code, textbfCounterClient.java. That way, we could track the latency of each of the individual increment operations to the system. We then wrote them to an output file named **data*id*.csv**, where id is the unique client id specified at client start time. For instance, client with id 1000 will write to **data1000.csv**. Additionally, the python script aggregates this information and calculates average latencies for each client if the system is running a multi client operation. This data is stored in a file called **summary.csv**.

D. Throughput

We had two different definitions of throughput as follows:

$$\text{System Throughput} = \frac{\text{iterations} * \text{num clients}}{\text{total system time}} \quad (1)$$

$$\text{Client Throughput} = \frac{\text{iterations}}{\text{client run time}} \quad (2)$$

We calculated the system time by taking the start time of the earliest client and subtracting it from the end time of the last remaining client. Client run time was calcuated by fetching the unix timestamp before and after the set of increments occurred. The times are tracked in the **CounterClient.java** file, while the system throughput is calculated in **script.py** as it concerns the potential for multiple clients. Ultimately, the throughput data for each client is saved in a file called **throughput.csv**.

III. Experimental Methods

A. Write Latency Distribution

We want to show the latency distribution on our system.

- We run a BFT-SMaRt system with 4 nodes.
- We run a single client on it that executes 1000 operations. We record the latency that each operation takes.
- We construct a histogram of those values.

B. Write Latency Experiment

We want to show the relationship between the number of clients submitting writes to the system and the average latency that each client experiences. We run the following experiment.

- Run the BFT-SMaRt system with 4 nodes. Execute the CounterServer application.
- Create *n* clients, each of which execute *iterations* write operations to the application.
- For each client, calculate the latency for each operation. Take the average of all of these latencies as *avgLat*.
- *n* = 1, 5, 10, 20, *iterations* = 100
- Construct a graph with *n* on the y axis, *avgLat* on the x axis.

C. Throughput Experiment

We want to see the relationship between the number of clients connected to the server and the throughput.

- We run a BFT-SMaRt system with 4 nodes.
- System throughput is measured by the operations per second that the servers can process. Client throughput is measured by the amount of operations per second a client can execute.
- We measure *iterations* as the number of commands each client executes, *n* as the number of clients running on the system, *totalTime* as the time it takes for all concurrent clients to finish, and *clientTime* as the time it takes for a single client to finish.
- $systemThroughput = (iterations * n)/totalTime$
- $clientThroughput = iterations/clientTime$
- We run this experiment several times, each time with a different *n* number of clients connected to the system. We record the average *clientThroughput* for the *n* clients and the *systemThroughput* for each experiment.
- *n* = 1, 5, 10, 20, *iterations* = 100
- We construct two graphs. In one, we have *n* on the x axis and *systemThroughput* on the y axis. On the other, we have *n* on the x axis, average *clientThroughput* on the other.

D. Faulty Message Injection Attack

We want to determine if a lying leader node, that gives 1 node a different operation than it gives to the other 2 nodes, can fool that node into having a different state from the rest of the system.

- Run BFT-SMaRt with 4 nodes running the CounterServer application. We upload malicious code to the node with id 0. When the system starts up, node 0 is assigned as leader by default.
- On receiving a message, the malicious code creates 2 separate *PROPOSE* messages. One that is correct and forwards the correct operation to nodes 1 and 3, and one that is incorrect and forwards the operation *ADD 2* to node 2.
- A single client connects to the system and submits an *ADD 1* operation to the system.
- We inspect the logs of each node, and see the final value of the counter at each. We see if there is an inconsistency.

E. Latency Attack - 1 Client

We want to show the effect of a leader latency attack. In this attack, the leader node slowly introduces delay in the system by sleeping for some amount of time before forwarding new client requests to other nodes.

- Run BFT-SMaRt with 4 nodes running the CounterServer application. We upload malicious code to the node with id 0. When the system starts up, node 0 is assigned as leader by default. The code inserts a wait before the node sends a propose message for a new value. This delay increases by 3 ms every time a consensus message is sent.
- A single client connects to the system and submits *iterations* write operations to the application. It records the latency that each operation experienced, and the time the operation completed.
- *iterations* = 100
- Construct a graph with the time an operation completed on the x axis, and the latency for that operation on the y axis.

F. Latency Attack - 3 Clients

We want to show that eventually, the system recognizes the latency attack and evicts the faulty leader.

- Follow the same steps as the previous experiment, but run *n* clients each running *iterations* operations.
- *n* = 5, *iterations* = 100
- Construct a graph with time an operation completed on the x axis, and the latency for that operation on the y axis.

IV. Results

A. Write Latency Distribution

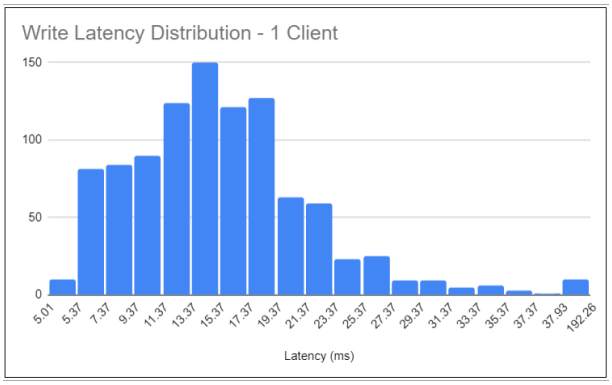

Fig. 6. Distribution of Latencies over 1000 Iterations for 1 Client

In Figure 6, we see the data we obtained for experiment A. Most of the increment operation latencies occurred around 15 ms, in a right-skewed distribution with a few outliers around 100 ms. This is the sort of distribution we expected to see and laid a solid groundwork for our further data collection.

B. Write Latency Experiment

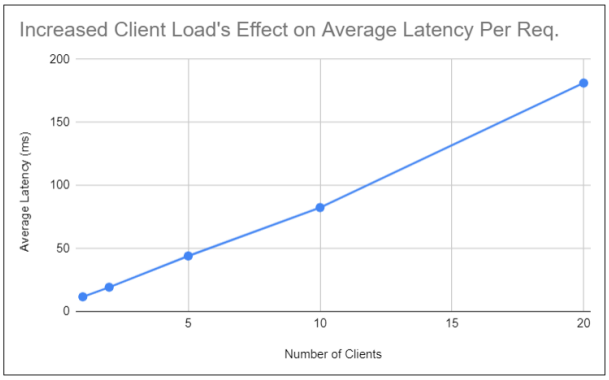

Fig. 7. The effects num of clients has on Latency

Figure 7 showcases the results from experiment B. As the number of clients increases, the average latency experienced by any client on average for an operation also increases in a linear fashion. This is what we expected to see, as with more load (more clients running at the same time), there are more requests for the system to handle.

C. Throughput Experiment

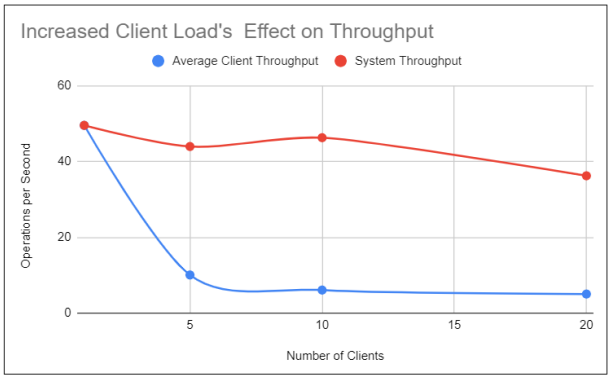

Fig. 8. The effects num of clients has on Throughput

In a very similarly structured graph to figure 7, figure 8 compares the number of clients with the throughput (as opposed to latency) of the system. The whole system throughput remains unchanged with more clients entering the system, however an individual client's throughput will reduce. This makes sense, as the system should stay about consistent in operations served per second even with more clients, but this does mean that the average client will be served less efficiently. We were encouraged to see that system throughput did not decrease substantially, even increasing the number of clients to 20.

D. Faulty Message Injection Attack

With faulty message injection, we enabled our injected protocol on node 0 and observed some interesting results. Nodes 1 and 3 received the correct increment, as intended, however node 2 was registering the faulty value that was being sent to it by node 0 as the consensus value. This was not what we anticipated, as we thought that all servers would agree upon some value before sending it to the client. However, the client did end up receiving the correct increment and ultimately the correct value.

E. Latency Attack - 1 Client

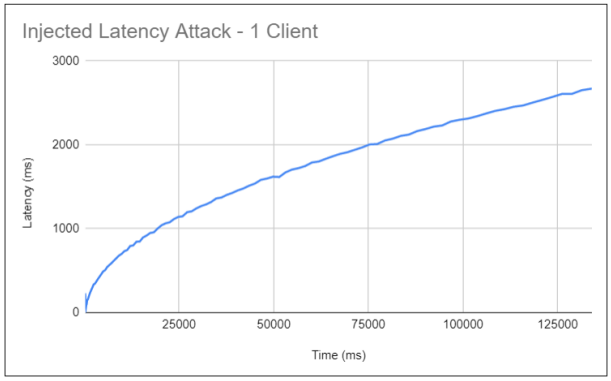

Fig. 9. The effects num of clients has on Latency

As seen in figure 9, the latency attack was successful in increasing latency when only 1 client is connected to the system. The latency increases according to a logarithmic pattern. A single client running a job that executes a mere 100 operations took many minutes to execute. Because the progress was so slow, the availability guarantee is practically violated, even though the system still technically makes progress.

F. Latency Attack - 3 Clients

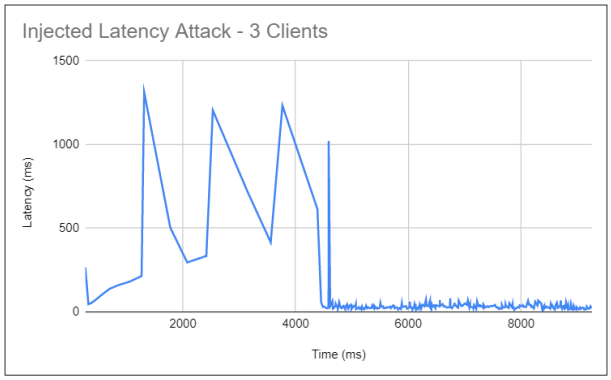

Fig. 10. The effects num of clients has on Latency

When we tested the latency attack with 3 clients, the system started out with high spiking latency as seen in figure 10. However, the system was able to realize that the leader was faulty, at which point it kicked it out of the system and reassigned the leader node. This can be seen at around time = 4500, where the latency spikes and then dips back down to very low values. Ultimately, this shows that the system is robust to these types of attacks under very specific conditions.

V. Limitations & Future Work

We are not completely sure why our message injection attack worked. BFT-SMaRt is supposed to withstand attacks like this, where the leader sends inconsistent messages to different parts of the system. According to the specifications, the system sends the actual value that consensus is being established on only once, in the *PROPOSE* phase. In subsequent phases, only the hash of the value is sent. It is possible that these two messages hashed to the same value, but this is unrealistic. Further investigation is needed to determine the exact cause of this failure.

Further testing is needed to determine the effect of message batching on performance. By default, BFT-SMaRt runs a consensus instance on a batch of 400 messages. In order to make our attack possible, we reduced to 1, so that a consensus instance is run on every message. This likely degraded throughput, as the consensus protocol is the most time consuming part of this application. In reality, it is likely that the true throughput of the system is many times larger (possibly 400 times as large) as the values we calculated. However, this comes at the cost of increased latency, as a client must wait for the whole batch of operations to be processed before it can get its result.

We would also like to test the effect that a larger number of nodes in the system has on performance. Because we only had access to 4 virtual machines, we only ran these experiments on a system with 4 nodes. However, we would have liked to see the relationship between throughput and latency for systems with 7 nodes and 11 nodes. We hypothesize that this would have resulted in higher latency and lower throughput, as more message exchanges would have to happen for consensus to be established, increasing the time it takes to reach consensus on an operation.