Reliable Distributed Systems

School year: 2016/2017

Lab project part I and II

Implementation of DDS – A Dependable Storage System

Group: CSD-10

Nº 41689 – Sérgio Pimentel
Nº 41710 – David Gago

Abstract

In the scope of Reliable Computer Systems subject which is part of the Master's Degree in Science and Computer Engineering at FCT-UNL, this report is intended to specify the implementation of a Dependable Storage System and the results obtained. The main focus of the project is to build a reliable storage system over a secure SSL/TLS communication between the client and the server. Using the Attiya, Bar-noy, Dolev algorithm, ABD for short, our system achieves reliability with all replicas, or a quorum, replying to a Proxy call which serves as an intermediate between the replicas and the client. In this project, the system is considered to be a real target and a set of evaluations will be performed to test the reliability.

1. Introduction and Project Context

With the constant evolution of digital information and technology, reliable systems are crucial when it comes to providing the clients safe, consistent and private functionalities. Medical and Bank systems are only two examples of how important distributed system's procedures must be studied and developed, with millions of private and personal data entries being digitally registered, daily. This data is stored in servers, but having only one server is risky since we never know when it might fail, for some reason. That's why most systems have a set of servers, so when one fails, we can always rely on the other ones. In order to rely on multiple set of replicas, we must ensure that all of them, or at least most of them, contain the true and wanted values. What if, in a Bank transfer, two clients perform two distinct operations in the same account? The client A adds 10% of the current account's balance in one replica, while the client B adds €200 in another replica. Both replicas may receive both operations by communicating between them, but execute them at different

times, giving an inconsistent balance between replicas in the end. This is the kind of problem that, in this project, we try to solve by using distributed systems techniques. The service will implement a storage distributed system of big-table kind where the entries correspond to key-value sets. To ensure the system properties are achieved, it'll use consistent replication on top of the data. This storage is distributed across several organized replicas that rely on a byzantine quorum system.

2. Model and system's architecture

The system's architecture is composed by the client and server side where the server provides an API and uses a Proxy as an intermediate between the client and the available replicas. The model is shown in the image below.

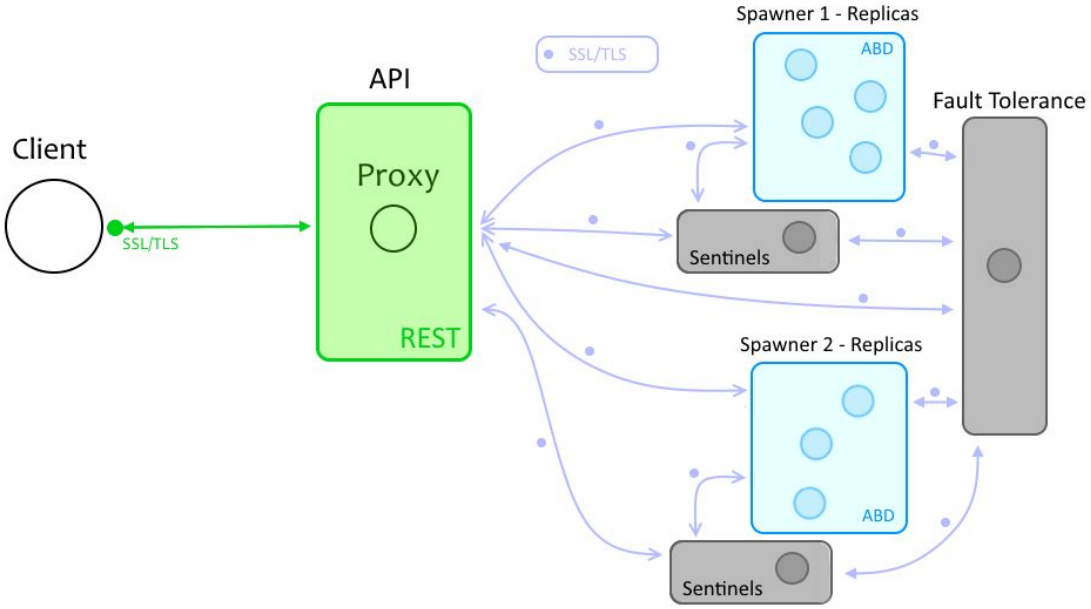

Figure 2.1 - Architecture Model

The client will perform operations over the server's API using a secure SSL/TLS communication. Each call is handled by a Proxy and will also be in charge of communicating with a set of available replicas. The reliable component of our system's replicas will be taken care of with the ABD algorithm. Each spawner will have a sentinel waiting to be assigned as replicas when any of the replicas might fail or behave in a strange manner. For now, we'll describe each component role in a very brief way, topic 4 will discuss them in more detail.

1. **Client** - Will simply connect to the server and make use of the API.
2. **Server** – Has five main components.
   2.1. **API** – A set of read and write operations that allows the client to use the system's functionalities.
   2.2. **Proxy** – Serves as an intermediate between the client and the set of

replicas, evaluates the replicas replies.

2.3. **Replicas** – Used to store the client's the data and that we wish to maintain constant reliability.

2.4. **Fault Tolerance** - Constantly checking for weird-behaving replicas which may be replaced by sentinels.

2.5. **Spawner** - Spawns new replicas, active or sentinel, by command of the Fault Tolerance component.

3. API specification

The API is composed of two parts, primary and extended API. In the following section we'll list and describe each operation according to our point of view (implementation). All the operations may be executed on either encrypted or unencrypted mode. The allowed/valid entries that our storage system can handle is found in a *conf.txt* file and is processed by the server and sent to the client, better explanation at section 5.1, but for now it's only important to mention that the client will know what types of Entries are allowed to insert and which operations are allowed in each entry fields. For example, an allowed entry type may be (int string int string int string) and for each and one of those fields, only the corresponding operations can be executed (< % + = & .), which means that the first field will have order preserving encryption, the third field searchable encryption and so on. (check section 5.1 for detailed explanation).

Primary:

1. PutSet: The client is allowed to insert any value by choice for each of the entry's allowed fields and also inserts a key associated with it.
2. GetSet: The client inserts the key associated with the entry which wishes to retrieve.
3. AddElem: Simply appends a null value to the list of values of an existing Entry, the key is the only argument.
4. RemoveSet: Given a key associated with an existing entry, the entry is removed.
5. WriteElem: Overwrites a specific field of an existing entry, given the position and value.
6. ReadElem: Reads a specific field of an existing entry, given the position and key associated to an existing entry
7. IsElem: Checks if the element given as an argument exists in an entry associated with a key(also given as an argument).

Extended:

1. Sum: given two keys and a field position where the sum operation is allowed, returns the sum of the fields of both entries
2. SumAll: sums all values of all entries on a specific position given as parameter where the sum is allowed.

3. Mult: given two keys and a field position where the multiplication is allowed, returns the product of the fields of both entries.
4. MultAll: multiplies all values of all entries on a specific position given as parameter where the sum is allowed.
5. SearchEq: given a position and a value where deterministic operations are allowed, this method will return all keys associated with the entries that encountered an equal match.
6. SearchNEq: given a position and a value where deterministic operations are allowed this method will return all keys associated with the entries that encountered an equal match.
7. SearchEntry: given a value to search, this value will be searched in all "searchable" (%) fields of each entry, returning the keys associated with the entries that match.
8. SearchEntryOR: given a set of values to search, this set will be searched in all "searchable" (%) fields of each entry, returning the keys associated with the entries that match at least one of the values present in the set.
9. SearchEntryAND: given a set of values to search, this set will be searched in all "searchable" (%) fields of each entry, returning the keys associated with the entries that match all the values present in the set.
10. OrderLS: given a position by the client where ordering preserving encryption is allowed, returns all keys of entries in ascending order.
11. OrderSL: given a position by the client where ordering preserving encryption is allowed, returns all keys of entries in descending order.
12. SearchEqInt: given a position and a value by the client where ordering preserving encryption is allowed, returns all keys of entries that equal the value.
13. SearchGt: returns all keys of entries that have a greater value than the value given as argument by the client, in a specific ordering preserving position.
14. SearchGtEq: returns all keys of entries that have a greater or equal value than the value given as argument by the client, in a specific ordering preserving position.
15. SearchLt: returns all keys entries that have a smaller value than the value given as argument by the client, in a specific ordering preserving position.
16. SearchLtEq: returns all keys of entries that have a smaller or equal value than the value given as argument by the client, in a specific ordering preserving position.

4. Reliable and Adversary Model

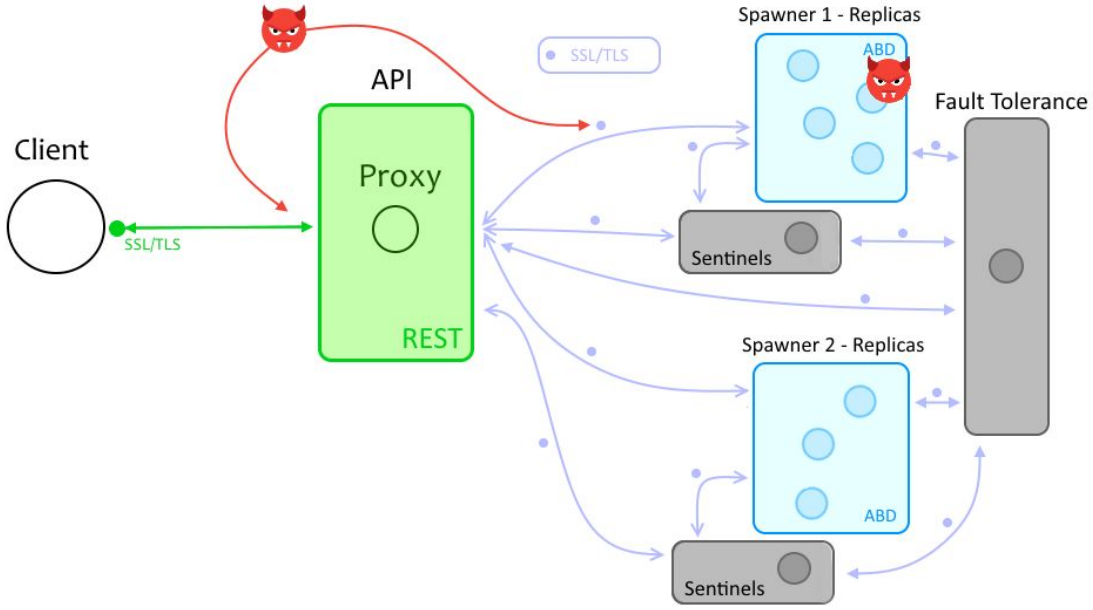

Figure 3.1 - Attack Model

The outside attacker tries to make a client - server communication or break its properties, the solution for this is using TLS and REST server with the authentication of the server. The trusted computing base is a set of all hardware, firmware and software components that are critical to the security of our system. In our case, these components are keystores, truststores, internal replica/proxy memory, and the TLS security protocol.
The Proxy - Replica communication also uses TLSv1.
Several intrusions from both type of attackers (outside and inside) may be sent such as malicious actions against the execution environment of the distributed storage service. DoS attacks against one replica causing a crash failure. Byzantine attacks inducing byzantine failures in the DSS execution environment. For our system we also considered the attack model referenced in the OSIX.800 and RFC2828 recommendations for a proper security service.

5. System's architecture detail and its components

In this topic we'll describe each component in more detail.

1) **Client**: Is provided with a set of operations exported from the server's API. The client can then manage the storage and search for available data in the DDS. Secure TLS communication over a REST interface.
2) **Server**: The server is built using three main components, the API, proxy and a set of replicas.

a) **API:** The primary API has the following operations: (the extensive API is in the annexes).
   i) Key PutSet(String key, Entry set)
   ii) Entry GetSet(String key)
   iii) Status AddElement(String key, Object val)
   iv) Status RemoveSet(String key)
   v) Status WriteElem(String key, type element, int pos)
   vi) Boolean IsElement(String key, String element)

b) **Proxy:** The proxy, will store the requests and process them iteratively, waiting for a quorum of replies from the replicas and redirecting those responses back to the client. The following list is the set of operations and data structure that match with what we wish to achieve with the ABD algorithm.
   i) HashMap[Long,Request]
   ii) Read(nonce:Long,key:String)
   iii) ReadResult(tag: Tag, v: Entry, sig: String, nonce: Long, key: String)
   iv) ReadTagResult(tag: Tag, sig: String, nonce: Long)
   v) APIWrite(nonce: Long, key: String, id: String, v: Entry)
   vi) Ack(nonce: Long)

c) **Replicas**: The set of replicas that will store the client's data and that we wish to ensure reliability in all replicas or a quorum. The following list is the set of operations and data structure that match with what we want to achieve with the ABD algorithm.
   i) HashMap[String,(Entry,Tag,String)]
   ii) ReadTag(nonce: Long, key: String)
   iii) Write(new_tag: Tag, v: Any, sig: String, nonce: Long, key: String)
   iv) Read(nonce: Long, key: String)
   v) CrashReplica
   vi) SetByzantine(chance: Int)
   vii) sendMessage(target:ActorRef, message: Any)

d) **Fault Tolerance:** Constantly checking for weird-behaving replicas which may be replaced by sentinels.
   i) SyncReplica
   ii) Set of replicas
   iii) Set of sentinents
   iv) Set of proxys
   v) Votes : HashMap
   vi) Synced Sentinents : HashMap
   vii) RegisterReplica()

viii) RegisterSentinent()
ix) RegisterProxy()
x) Vote()

e) **Spawner**: Spawns new replicas, active or sentinel, by command of the Fault Tolerance component.
f) **ABD**: This algorithm assumes the system is asynchronous with reliable channels and consists in waiting for a quorum of responses. As described in [1], the fault tolerance idea behind the ABD algorithm is to send a timestamp request to all replicas, but only to wait until one hears from all members of some Read Quorum. Similarly, one sends writes to all replicas, but considers the task completed after receiving Acks from any Write Quorum. That way, as long as there is at least one full Read Quorum and Write Quorum of correct processors, all other processors may fail.

The **ABD** Algorithm:

1. State
   a. Val_i: Value of the variable, initially v0
   b. Tag_i: Pair <Number of sequence, ID> initially <0,0>
   c. Sig_i: signature of <Val_i,Tag_i>
2. Client c: Write(v)
   a. Generate nonce
   b. Step1
      i. Send(<read-tag(nonce)>) to all processes(or to quorum)
      ii. Wait for a quorum Q of valid replies (with nonce and authenticated)
      iii. Let seqmax = max {sn:<sn,id,sig> belonging to Q}
   c. Step 2
      i. Send(<write(<seqmax+1,c>,v,sig,nonce)>) to all processes (or to a quorum) with sig = sign(<<seqmax+1,c>,v>)
      ii. Wait for a quorum of valid acks with the given nonce
3. Client c : Read()
   a. Generate nonce
   b. Step 1:
      i. Send(<read(nonce)>) to all processes (or to a quorum)
      ii. Wait for a quorum Q of valid replies (with nonce and authenticated)
      iii. Let <tagmax,valmax,sigmax> belonging to Q be the reply

with the largest tagmax

c. Step 2:
   i. Send(<write(Tagmax,Valmax,Sigmax,Nonce)>) to all processes (or to a quorum)
   ii. Wait for Quorum of valid acks
   iii. Return valmax

5.1 Support for encrypted operations and implementation

In order to support encrypted homomorphic operations we started by adding a configuration file that tells the server which type of entries are valid/allowed and what operations may be executed in each and one of the entries fields upon server initialization. So for example, the conf.txt file has the following lines:
Line 1: int string int string int string
Line 2: < % + = & .
Line 1 explicitly informs the application the valid entry. As for Line 2, we approached our implementation based on the homomorphic library documentation [2] provided by our teachers.

Property	Homomorphic Operations	Class	Input Data Types
Random	None (strong cryptanalisys resistance)	HomoRand	Strings, Byte Arrays
Deterministic	Equality an inequality comparisons	HomoDet	Strings, Byte Arrays
Searchable	Keyword search in text	HomoSearch	Strings
Order preserving	Less, greater, equality comparisons	HomoOpeInt	32 bit Integers
Sum	Add encrypted values	HomoAdd	BigInteger, String
Multiplication	Multiply encrypted values	HomoMult	BigInteger, String

=, <>	– determinist encryption
>, >=, <, <=	– order preserving encryption
%	- searchable encryption
+	– Paillier
&	– RSA.
.	– random encryption
other value	– no encryption

Therefore, Line 2 (< % + = & .) informs the server that the first field has order preserving encryption, the second field searchable encryption and so on. The

moment the Client connects to the Proxy, the server will respond with the configuration file back to the Client. From the operations mentioned above, only one is allowed for each entry field. The library used to encrypt our data was the library supplied by our teachers, SJ-HomoLibrary [2]. and the component in our model, responsible for encrypting and decrypting some of our operations is the Proxy component.

5.2 Fault tolerance and dynamic replica support implementation

In the following sections we'll describe how we proceed in implementing the fault detection component of our project and all techniques associated with it.

5.2.1 Fault Detection

To implement detection of either crashed replicas or byzantine replicas, we decided to make a verification asynchronously in the proxy after having sent the request to every replica. That verification consists in filtering, from the list of active replicas at the time of the request, the ones that did not send an answer or sent an invalid answer.

That verification is different for the different types of answers that the request is waiting for, as in the case of SumAll/MultAll we must compare the values received determine the right one and detect the replicas that sent a different result. Similar process for the methods that return a list of keys as we must determine the right list and filter the replicas that did not send it.

Whenever the proxy detects an anomaly, it votes for the corresponding replica in the FaultTolerance actor.

5.2.2 Sentinent Replicas

When the replicas are spawned, it is not only defined the number of replicas but also the amount of sentinent replicas. Those replicas do not belong to the quorum so they don't receive requests, but each of them is controlled by an asynchronous thread of the FaultTolerance actor that every few seconds asks that sentinent replica to synchronize with another random replica from the quorum.

5.2.3 Recovery & Installation

When the number of votes of a certain replica reaches the threshold value

(parameterized in the FaultTolerance actor initialization), that actor is removed from the quorum and a fresh sentinent replica is added to the quorum, if available, and set an active replica and forced to do one last synchronization with a random replica. It can receive requests between being active and having the last updated state which makes it possible to give wrong answers. The threshold value must be selected having this in mind so that the number of wrong answers the replica gives in that interval is not superior to it.

The FaultTolerance actor also sends the new replica list to all the proxies registered in that actor so that the requests can be sent to the correct quorum.

5.2.4 Availability Guarantees

Every time a faulty replica is detected, the system tries to put a fresh replica available, so that would characterize the recovery mechanism as pro-active. Since the number of replicas needed for quorum is configurable as is the number of replicas in the system, the resilience factor (K) is the difference between these two. So, for instance, in the system configuration suggested (3 replicas + 2 replicas + 2 replicas, with a quorum of 5), the resilience factor would be 2.

When a sentinent replica is set to active, the FaultTolerance actor also signals one of the spawners to create a new sentinent replica, starting also the thread to synchronize it with the others.

6. Implementation aspects

In our implementation we used an hybrid approach. Using Akka's programming environment. The Proxy and Replicas were implemented using the available Actor System provided by Akka which makes use of the SCALA programming language. While the Client, Server initialization and API were developed in JAVA.

For experimental evaluation we decided to run our system in a virtual local area network using tunngle, for that we had to make small changes to our project. One of them was adding hostname to the akka configuration so it had the IP address of each of our adapters in that network. The other change was fixing a bug that would only manifest in a high request intensity environment, breaking the system. To fix it, we had only to make sure that any vote would only be valid if the replica was still in the list of active replicas.

7. Analysis and experimental results

In this section we'll show the experimental results for each type of experiment in our system. The first experiment evaluates the throughput without any attacks, the second with one or two crash attacks and the third with one or two byzantine attacks. For each experiment, the following benchmarks are executed:

Primary Benchmarks:

Benchmark 1: 100 Putsets
Benchmark 2: 100 Getsets
Benchmark 3: 50 Putsets and 50 Getsets, alternating
Benchmark 4: 50 AddElements and 50 ReadElements, alternating
Benchmark 5: Mixture of all available operations

Extended API Benchmarks:

Benchmark E1: 10 operations based on the search operations on top of unencrypted data.
Benchmark E2: Mixture of 100 operations on top of unencrypted data.
Benchmark E3: 10 operations based on the search operations on top of encrypted data.
Benchmark E4: Mixture of 100 operations on top of encrypted data.

How we structured our experiments

The results were obtained by first implementing a Benchmarks class that includes all of the benchmarks mentioned above. Every time one of the benchmarks is executed through the client's interface a set of features are registered in a CSV called "resultsEncrypted.csv" or "resultsUncrypted.csv" depending on which type of server currently running (unencrypted or encrypted). The CSV contains the following columns: benchmark , operation, status, time, encrypted. The first column (benchmark) goes from 1 to 5. The operation column can be PUT, GET, ADD, READ, WRITE, REM, ISEL that correspond to the primary API operations Putset, Getset, AddElem, ReadElem, WriteElem, RemoveSet, IsElem, respectively. In regards to the Extensive API the operations are SUM, SUMALL, MULT, MULTALL, SEQ, SNEQ, SE, SEOR, SEAND, SEQINT, SGT, SGTEQ, SLT and SLTEP which correspond to Sum, Sumall, Mult, Multall, SearchEq, SearchNEq, SearchEntry, SearchEntryOR, SearchEntryAND, SearchEqInt, SearchGt, SearchGtEq, SearchLt and SearchLtEq. The status show us the response's

status from the server. Finally the time column shows us the execution time it took to perform the operation. When the application first starts, a brand new "results[unencrypted/encrypted].csv" file is created and by performing any benchmark, the results will be appended to the end of the file. This file will then be executed by a Python script that reads all this data and displays it in bar charts.

How we proceeded to execute the experiments

The way we configured our system in order to do the experiments was to first implement the project in a way that could be parameterizable and therefore allow the application's settings to be changed. We can execute each server with the following arguments:

```
-bz,--byzantine <arg>   number of replicas that are byzantine
 -ch,--chance <arg>      probability of crashing/byzantine error
 -cr,--crash <arg>       number of replicas to crash
 -f,--fault <arg>        fault detection server's address
 -k,--keystore <arg>     keystore path
 -n,--number <arg>       number of replicas to spawn
 -q,--quorum <arg>       quorum size
 -s,--sentinent <arg>    number of sentinent replicas to spawn
 -t,--type <arg>         type of server (spawner1/spawner2/proxy/fault)
 -th,--threshold <arg>   number of votes to kick a replica
```

With these arguments we could create three types of servers (proxys, replicas, fault tolerance), and also define, in the creation of a proxy, the number of replicas we wish to crash or to set as byzantine, with a specified chance of that happening. For each operation there would then be a chance (given as parameter) of a number of replicas (also given as parameter) to be set as byzantine or to simply crash. As replicas, we can either set a server as type spawner1 and spawner2, this was created in order to distribute replicas across different hosts.

The client in its turn can be initialized in the following way:

```
-k,--keystore <arg>     truststore path
-h,--hostname <arg>     rest server address
```

We also used a program called Tunngle that allows us to create a virtual

private network and therefore create remote replicas across hosts.

First Experiment - No Attacks

In this first experiment, we simply create all three types of servers using the arguments described previously (spawner1, spawner2 and a proxy) and execute the benchmarks. Spawner1 was set to have four replicas, Spawner2 three replicas and a Proxy receiving the listing of addresses of those replicas. The following throughput was obtained:

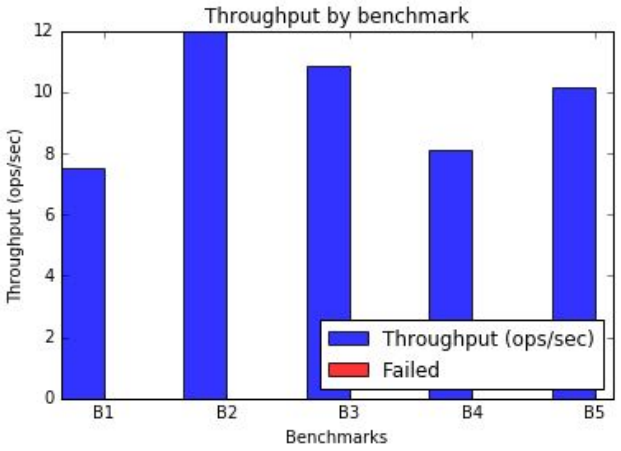

Figure 6.1 - No Attacks

Second Experiment - Two Replica Crash

This experiment follows the same line of settings as in the first experiment, but this time the proxy will have two more arguments (-cr and -ch) to define the number of replicas we wish to crash (-cr) with a given chance (-ch), for each operation.

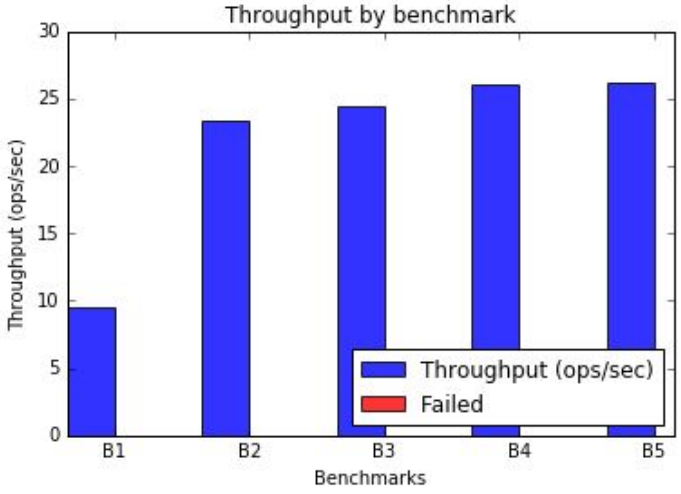

Figure 6.2 - Two Replica Crash

Third Experiment - Two Byzantine Replicas

The third and final experiment uses the same line of settings as the previous experiments, but again, the proxy will now have the two corresponding arguments for byzantine configuration (-bz and -ch) to define the number of replicas we wish to set as byzantine(-bz) with a given chance(-ch), for each operation.

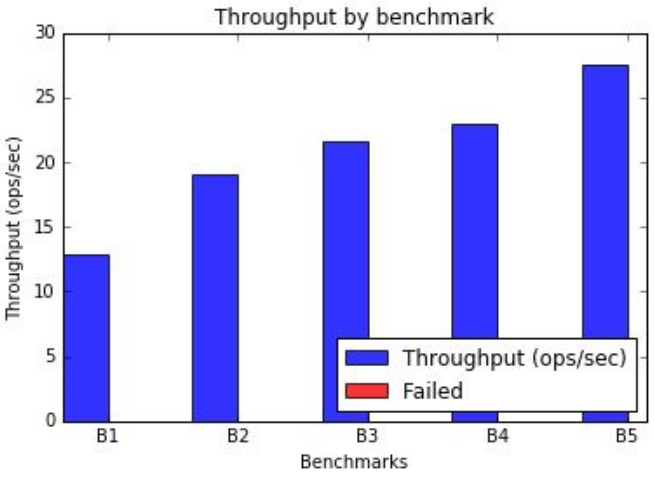

Figure 6.3 - Two Byzantine Attacks

As we can see in the charts above, our system behaved properly in all three required experiments. Overall, the benchmark with highest throughput was Benchmark 5, and the lowest Benchmark 1.

Fourth Experiment - Extensive API benchmarks

For the extensive API there were, as mentioned above, four benchmarks (BE1,BE2,BE3,BE4) where BE1 and BE2 are executed in unencrypted data while BE3 and BE4 are executed on top of encrypted data. The results obtained, as requested, were the following:

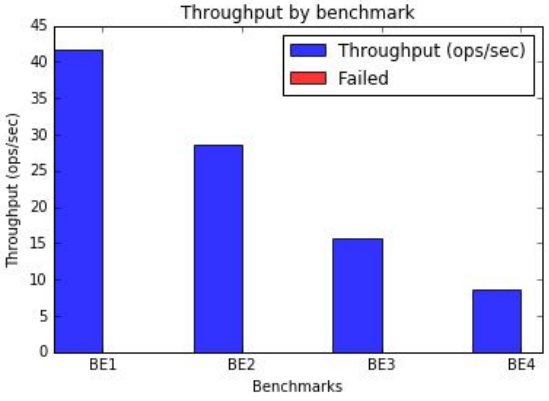

Figure 6.4 - Extensive API benchmarks

As we can see from the chart, there's a major difference between the throughput in unencrypted data (BE1, BE2) and the throughput of encrypted data (BE3,BE4). Despite the more secure system when using homomorphic operations, it comes with the cost of having a higher computational execution time. Although the initialization of the homomorphic operations such as key generation for each client may increase some of the computational time the latency of the network won't directly impact the time that it takes for the encryption itself to be executed.

Referências

[1] Soma Chaudhuri. 2003. Lec18.pdf. [ONLINE] Available at: http://web.cs.iastate.edu/~chaudhur/cs611/Sp13/notes/lec18.pdf .
[2] Topics on Homomorphic Encryption and Support included in the SJ-HomoLibrary. Henrique Domingos.