FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Confiabilidade de Sistemas Distribuídos Final Project

José Leal
50623

Rafael Gameiro
50677

Cláudio Bartolomeu
58413

June 2020

Abstract

This report is focused on the final project that will integrate, consolidate, optimize and extend the successive requirements, initially conducted as Work-Assignments in Labs.

Contents

1. Introduction

This project is based on a wallet service that supports operations like creating an account, deposit money, transfers from a user to another and consult a client ledger or the global ledger.

The project was conducted in two parts, the first one aimed at improving the solution developed in the work assignments and the second one was focused on the experimental study of the dependability properties offered by Redis, to enhance the project implementation and the overall dependability guarantees, using redundant configurations, like primary-backup or replication cluster.

Since the proposed work was to develop a REST-based service we decided to use the framework Spring, which provides functionality such as RestTemplates that makes the implementation of the client much easier. It also offers abstractions that manage the endpoints behavior which also makes the implementation of the server much easier. We also chose Spring because it is easy to configure and setup.

We store the data persistently in a Redis database. To ensure dependability we used the BFT-Smart library. When one replica receives a client HTTP request before executing it, the request is shared with all replicas in order for them to reach consensus of a response to return.

2. Architecture

In this section will be discussed the architecture of the components that make up the system. Figure 1 illustrates how the different components interact, in a very general way.

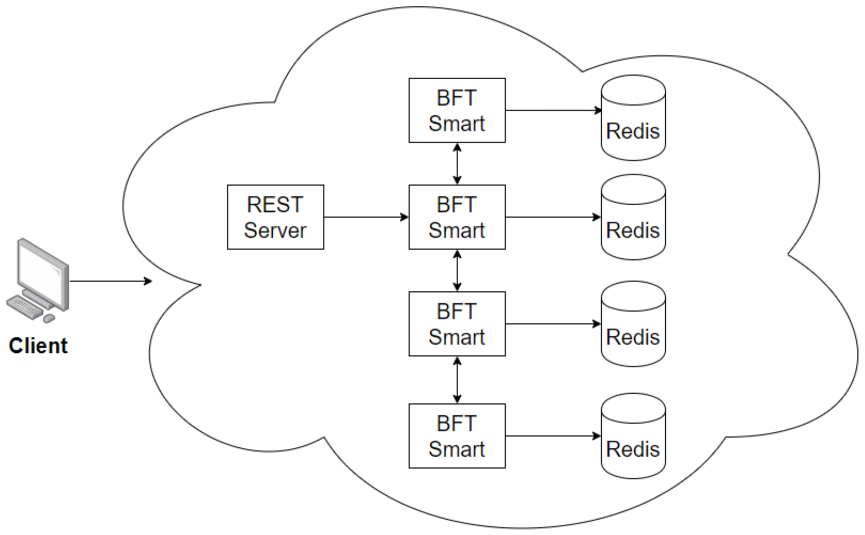
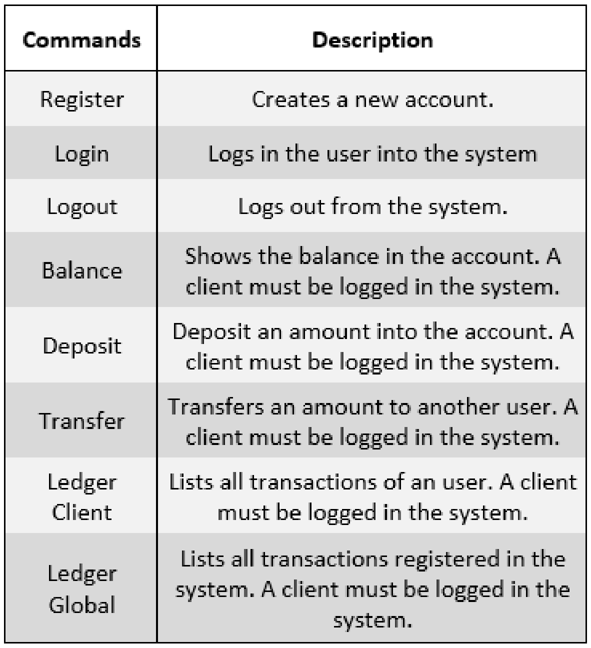


Figure 1: Interaction between the components

2.1. Client

Users will interact through this component. The client component will support the operations specified in the figure 2.



Commands	Description
Register	Creates a new account.
Login	Logs in the user into the system
Logout	Logs out from the system.
Balance	Shows the balance in the account. A client must be logged in the system.
Deposit	Deposit an amount into the account. A client must be logged in the system.
Transfer	Transfers an amount to another user. A client must be logged in the system.
Ledger Client	Lists all transactions of an user. A client must be logged in the system.
Ledger Global	Lists all transactions registered in the system. A client must be logged in the system.

Figure 2: Client's command description

2.2. Server + BFT Smart

Clients will connect to the server component to request the supported operations. As said before the solution is REST-based and allows interactions between clients and servers supported by TLS, with server-only authentication handshake guarantees and appropriate configurable TLS endpoints, via configuration files (in both clients and server).

In order to respond to a request, the server component will interact with the BFT Smart module to get the result of the operations executed in the replicas.

2.3. Redis

The standalone configuration is used when each replica is assigned a single redis instance.

2.3.1. Redis Cluster

The cluster configuration organizes many redis instances in order to balance the operations requested and the stored information. The cluster is organized in masters and slaves, where the masters store the information, while the slaves only store a pointer to the information in its master. Since the cluster uses sharding, when an operation is performed, the information regarding the operation is stored in only one master. Below, in the figure 3 we can see the cluster structure and how its components interact.

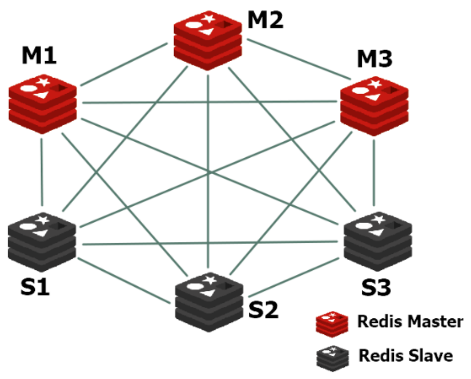


Figure 3: Redis cluster overview

2.3.2. Redis Replication

The replication configuration consists in assigning a group of redis instances organized in a primary-backup manner where the primary instance communicates directly with the assigned replica, while the backup instances just replicate the information stored in the primary. To provide high availability this instances are monitored by Sentinel nodes that provide capabilities such as notification, automatic failover and configuration provider. Underneath, in figure 4 we can see how the instances are organized, which reinforces what was mentioned (data flows from the master to the replicas).

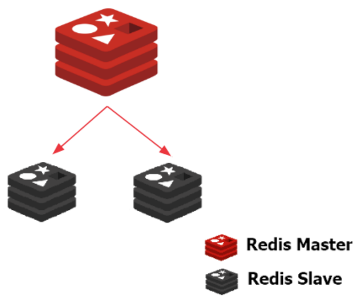


Figure 4: Redis replication overview

3. Threat Model

In this project we considered the typology of communication attacks against the underlying communication channels used between our components. The communication channels are between the client and the server infrastructure and the server infrastructure between different components, specified as the replication component and repository component

3.1. Client and Server Communication

The TLS channels provide between the client and server, message and data-flow integrity (including message ordering control), message and data confidentiality, and peer authentication as well as message authentication. All these services are only guaranteed during the TLS session and therefore, connection oriented. The peer authentication protection given by the TLS might be only one-sided since during the establishment of the TLS connection, mutual authentication might not be required, so only the server or the client provide proofs of authenticity.

3.2. Replication Component

The server replicas are managed using the BFT Smart library which can support a Crash-fault model or a Byzantine-fault model. In this project we considered our main implementation to use the Byzantine-fault model. In this model, in order to maintain the normal execution the system must support at least 3f+1 replicas, where f is the number of faulty replicas. If the number of faulty replicas in the system is greater than f, the system won't be able to reach a consensus, and therefore perform as if were under normal circumstances, which will disrupt the system and compromise its liveness properties.

3.3. Repository Component

The repository, as previously mentioned in the architecture section, was built using Redis repositories with different configurations. Despite the different configurations, interactions between the server and the Redis instances are modeled as Crash-fault. In this model, if one instance suddenly suffers a shutdown, its information cannot be longer accessed by other instances, and this is where redundancy comes into play.

In case this model were used in a real-world application the replica and repository would be in the same system hardware structure and because of that, possible attacks to the communication channels between them is not taken into account.

4. Implementation

4.1. Application Layer

Spring was used to implement both the server and the client.

4.1.1. Client

The client is a basic Command-line Application, where we used RestTemplates which allowed us to easily make HTTPS requests to the server. The TLS configurations (e.g., protocol versions, ciphersuites, keystores, truststores, etc.) are specified in the application.properties file which is then managed by Spring to auto-configure the server container context and the client command line application context.

For the application itself we use another keystore for the client, which yields another private key plus its public key certificate, that can be used to encrypt/decrypt and verify sent/received messages. The truststore of the client for use with the application is the same used in the TLS session.

4.1.2. Server

For the server we took advantage of the Controller-Service-Repository pattern and also the annotations provided, which make it very easy to build endpoints.

To provide server-side authentication our TLS approach maintains for the server a keystore which contains the server's private key and a certificate of the corresponding public key and also has a truststore which includes the certificates of all authorized users.

Similarly for the client we maintain a keystore which contains the private key to be used in the TLS sessions and the corresponding public key certificate. In addition to that we also maintain a truststore which contains the public-key certificate of the server.

To limit the actions of new clients and provide a form of incentive, the server implements a permission system, through the use of smart contracts. By using a smart contract, the system can process the client stored information and based on that, execute the operation the respective client is allowed to perform. If the client wishes to execute more high level commands, he must execute more low level operations. That way, the client needs to show commitment to the system and its way of operation.

Since smart contracts are simply code to run, clients can create a smart contract that can harm the system. To protect the system against that problem, the smart contracts are executed inside a container with a predefined limit of memory, CPU time to execute the contract and also with blocked I/O operations. That way a smart contract cannot run indefinitely, monopolise the system memory or compromise the database.

We also employ an access-control mechanism, based on an authentication header which functions as the primary line of defense. That is, before the server even attends the request, there is a handler that checks if the username and password on the header match any of the users registered in the system, if not the request will be discarded immediately.

4.2. Consensus Layer

Regarding the server infrastructure, in order to increase availability and prevent possible wrong responses from the server side, a state machine replication library named BFT Smart library was used. The library, as previously mentioned supports two modes, although we decided to use the Byzantine-fault model. With this model, the server infrastructure can be replicated to prevent byzantine faults. However, the library only prevent those faults if the number of fault servers are lower or equal than *f*, which is the max number of fault servers a byzantine fault system can support, in order to preserve the rule that states that a system to be a byzantine-fault tolerant must have at least *3f+1* server nodes.

4.3. Storage Layer

Our storage component is based on Redis instances that are configured to function as a database. Note that, each time the system is booted new redis instances are created, meaning each instance is empty at the start of the system. Spring offers an abstraction based on CrudRepositories that is very flexible, since it does not depend on the storage implementation. That is, it can be used with Redis or any other database that is supported by Spring. This modularity allows the programmer to swap the storage layer without having to rewrite the operations provided to the upper layers of the system. Further, CrudRepositories allow us to easily dictate which type of data is going to be stored in a repository and also implements most of the basic operations that are possible to perform on data stored in a persistent layer (retrieving records by id, storing records by id ...). To configure where the data in a repository is going to be saved, we also made use of RedisTemplates that are created from the properties specified in the application.properties file. These templates, support a variety of configurations:

- RedisStandaloneConfiguration: used to configure only 1 instance of Redis that is identified by a hostname and a port;

- RedisClusterConfiguration: used to configure the cluster. This configuration accepts the list of nodes that form the cluster and additional properties that define how the redirect mechanism will perform, because as mentioned when one is using Redis cluster the data will be sharded among the nodes and RedisTemplate will choose one at random when performing a request. In the case where the chosen instance is not responsible for the region where an operation should be performed it reefers to this information to redirect the request properly.

- RedisSentinelConfiguration: used to configure Redis instances to function with primary-backup replication. In this configuration, one needs to specify the master name, which instances are running in sentinel mode and several other properties such as the timeout used to identify when a master has failed. This configuration, enables the primary and backup replicas to function without the need of human intervention to resist certain kinds of failures.

5. Analysis

This section will be focused on the performance analysis depending on the configuration of the BFT module and different configurations of Redis.

5.1. Setup

To test the system performance, it was developed a client benchmark that creates a specified number of threads, which will request a given number of each operation to the server. When all the operations were executed, it will output the throughput of each one. The redis performance was measured using a benchmark provided by the redis official installation package. The benchmark will execute a set of operations, where each operation is done a number of times. At the end of each operation test, the throughput and time elapsed is printed out. Both performance tests were executed in a machine with an Intel i5 CPU with 4 cores, 8 threads and 16GB of memory.

5.2. BFT Smart benchmark

The client benchmark consists of 4 threads, each performing the operations 100 times. The number of threads and operations can be modified.

These tests had the purpose to compare the performance of BFT Smart module using the different models. Here we are taking into account the use of cryptography methods during the operations execution, which hinders performance affecting the overall throughput of the system.

The first test consisted in measuring the performance using the Crash-fault option.

Secondly, we tested the module using the Byzantine-fault option, where here we performed two evaluations: one with a clean execution and another where the system had a faulty replica.

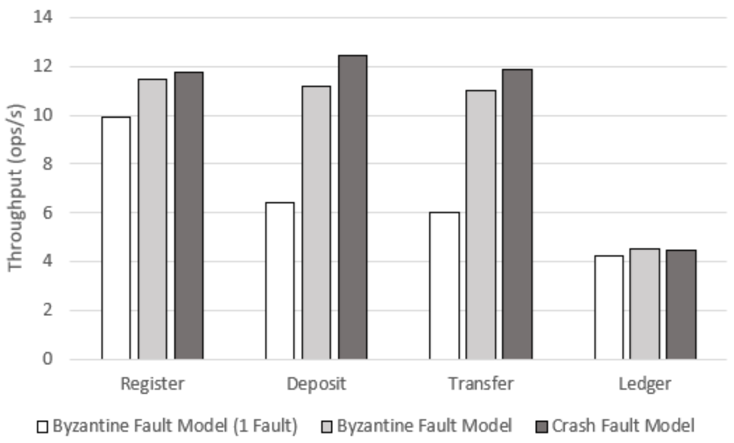


Figure 5: System performance with different fault models

As we can observe in fig. 5 there was no huge difference between the two models in terms of performance, although as expected the Crash-fault model performed slightly better than the Byzantine-fault.

Ledger operations are expensive by default, because it involves heavy read operations. So most of the time, is spent gathering this information and verifying its correctness between replica responses. Therefore, even when one replica is faulty this operations doesn't seem to suffer from its behavior, since the cost is amortized by the latter. However we can't state the same about the remaining operations when its results are compared between Byzantine-fault model tests, with and without faulty replicas.

Deposit consists in verifying if the user exists, retrieve its wallet, perform the addition and rewrite the wallet information. The transfer operation is basically two executions of the deposit, where the first one retrieves the amount from one wallet, so then it can sum it to the second wallet.

In all the operations the outcome is the same, basically the performance of the crash fault model is always better than the byzantine fault model. Also, byzantine fault model without faults outperforms the one with faults, because it is able to reach consensus much faster.

5.3. Redis Benchmark

Two set of tests served to measure the performance of redis. The first set focused in measuring in a direct way the repository performance, while the second set was used to evaluate its performance when incorporated as a component of the system.

5.3.1. Redis performance

This test included 50 parallel client connections, with payloads of 512 bytes and with keep alive set to false, which means the client will open a new connection every time it performs a request. We chose to set keep alive to false because it is similar to our model of interaction between our main server and the Redis instances. The number of requests issued per operations were 100000.

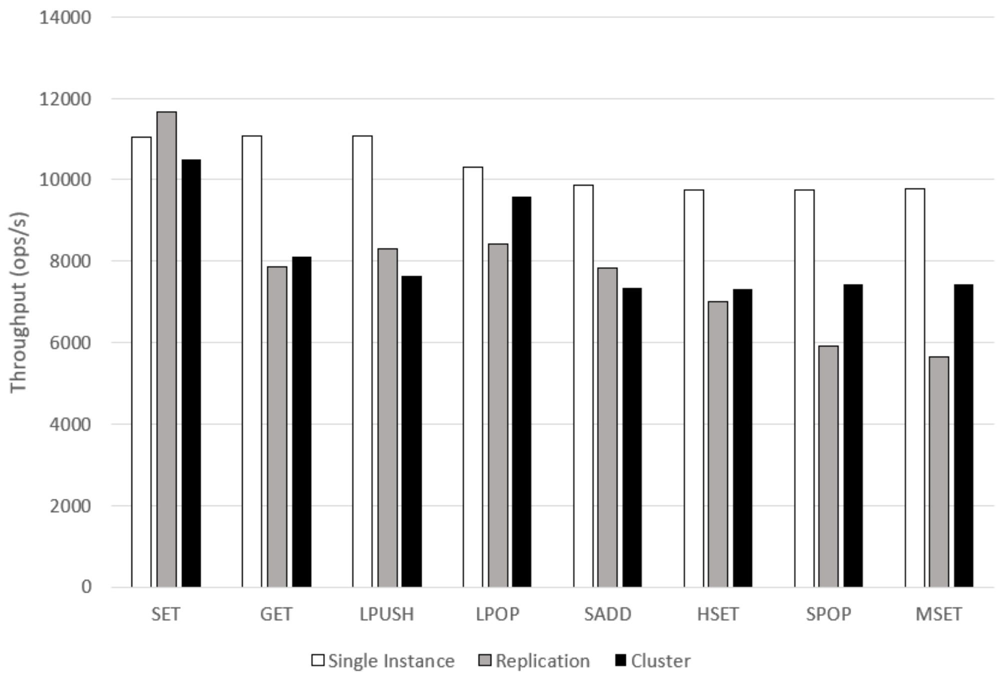


Figure 6: Redis performance with different configurations

The single instance was included as reference point and serves to illustrate how much the performance degrades from transitioning from a single Redis instance to a replicated environment.

Replication only allows the master to perform the write operations, and this instance is the only one allowed to handle this type of requests. The replication process is asynchronous and therefore the writes are always faster than the cluster, as we can see in fig. 6. The cluster setup involves 6 instances, from which 3 are masters. As mentioned before, the cluster shards the information it stores, therefore a master can receive a request for a hash fragment that it does not own and has to redirect this request to the appropriate master, wasting execution time which reflects directly in the performance of the writes.

In the read operations we can see that the cluster outperforms the replication configuration, because not only its structure allows to attend more requests per second, but also the fact that all requests won't be send to a single instance and therefore the workload will be more evenly distributed.

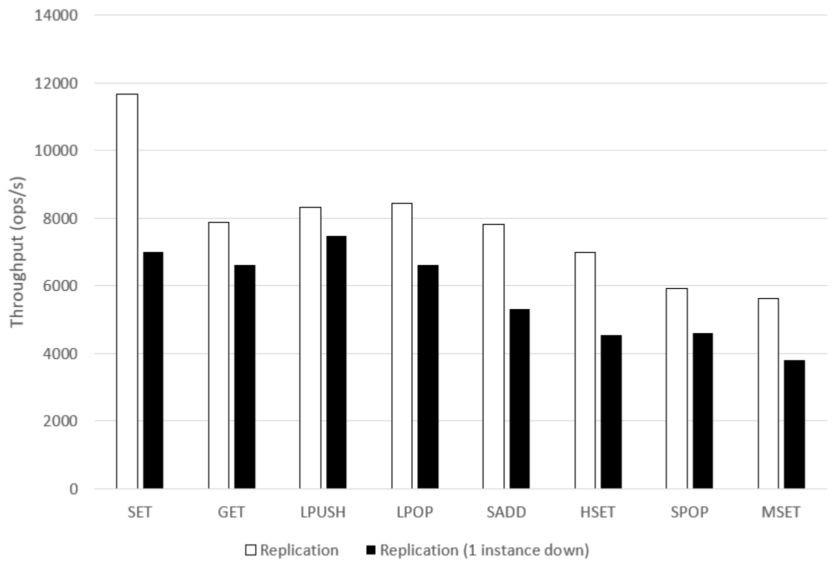


Figure 7: Redis performance with replication mode with 0 and 1 instance failing

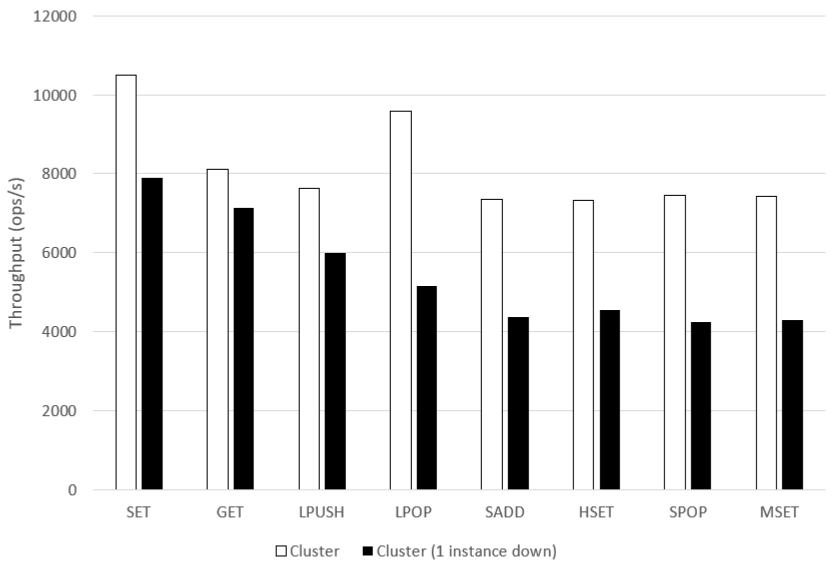


Figure 8: Redis performance with cluster mode with 0 and 1 instance failing

Figs. 7 and 8 compare the two configurations when one of its instances suffers a crash. As we can see, when a configuration performs without any crashes, its results are better, since the system does not need to deal with leader reelection and restore the system integrity. Because the leader is only suspected to have failed after a period of time, some operations might need to be requested again in order to be correctly executed.

Although the benchmark didn't allow us to test further failure scenarios we can conjure that cluster structure can tolerate better faults than the replication, because when one master fails the cluster can still respond to requests if the hash does not belong to the failing master, that does not apply with replication since a leader reelection must occur.

5.3.2. System performance

The client benchmark consists of 4 threads, each performing the operations 100 times. The number of threads and operations can be modified.

This tests included the full system, where 1 of the replicas used either a cluster or primary-backup replication, all others used a standalone Redis instance. We couldn't have a cluster or primary-backup configuration for each replica due to the constraints on the physical machine where the tests were performed. This would require the creation and management of a lot of containers and the results extracted wouldn't be accurate.

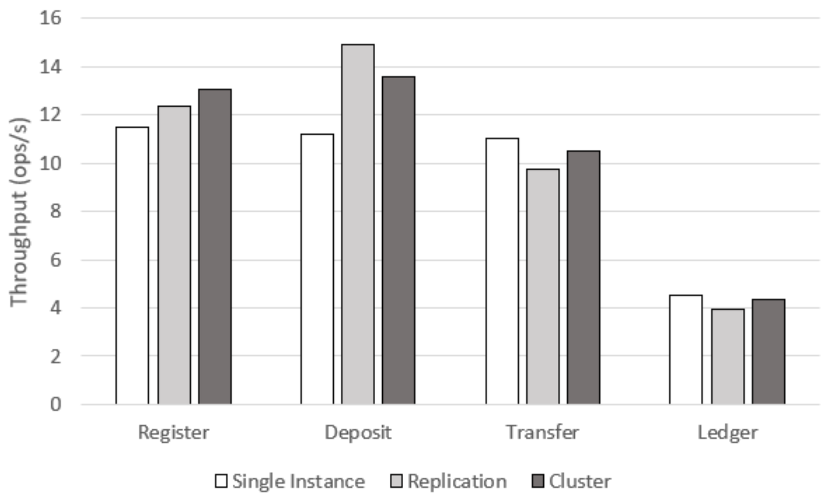


Figure 9: System performance with different Redis configurations

In fig. 9 we observe the test results after executing the client benchmark using different redis configurations.

As we can see, depending on the operation, each configuration seems to perform slightly better than the other ones. However, the cluster configuration yields the most interesting results, and seems to adapt very well to the bulk of operations that are performed in our system. This is possibly due to the sharding that allows the balancing of operations between the master nodes.

Our system was designed to be wallet repository where a client could deposit, consult and transfer money between other clients. This system can very much resemble a banking system where the principal objective is currency circulation, therefore in this scenario we would expect to get more write operations than read ones.

Based on this and the presented test results we can conclude that for the designed system, overall the best configuration would be a cluster. With it, the system would be able to perform more operations per seconds, mainly write ones, which are the desired operations in this environment.

5.4 Discussion

The replication solution is based on master-slave replication. The other scenario, cluster solution, is based on sharding the data and can also replicate each partition to other nodes, functioning like a primary-backup for each partition. Depending on the requirements of the system one solution might be more fitting than the other. From our results, we would advice read intensive systems to use cluster configuration, since it can serve more requests per unit of time and balances the storage evenly. Redis benchmarks showed that most write operations happened to be faster in replication mode, although the cluster solution didn't lose by much, so if the performance of the system is really important and the workload is mainly write operations, we advice the use of replication. Otherwise, cluster is always the better choice, since it scales better and has a higher degree of availability.

6. Conclusion

The project allowed the group to develop a dependable system. The features developed in the work assignments were refined to fit into the requirements of the project. For the second part of the project, the group accomplished the main objective that was to discuss and compare different Redis configurations. The two types of structures tested, primary-backup replication and cluster, did enhance the dependability property, more precisely the availability, but had counter effects on the performance of the system.

Depending on the requirements of the system one solution might be more fitting than the other. And in the end, it is up to the developers to chose the most fitting configuration depending on which operations they consider more critical.

As future work the group would like to improve the client authentication and further test the different solutions inside a trusted execution environment (TEE) like Intel SGX. Also since our solution is dockerized, we couldn't implemented the smart contracts using a "sealed" docker container because that would demand the creation of a docker container inside a docker container. Based on a search[1,2], we found some ways to mitigate the problem but they are not advisable since they could lead to system corruption. Therefore a future solution would be to restructure our system and allow the execution of smart contracts in the "sealed" container.

[1]https://jpetazzo.github.io/2015/09/03/do-not-use-docker-in-docker-for-ci/

[2]https://www.docker.com/blog/docker-can-now-run-within-docker/