FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Confiabilidade de Sistemas Distribuídos Work-Assignment #3

José Leal 50623 Rafael Gameiro 50677 Cláudio Bartolomeu 58413

April 2020

Abstract

This summary report is focused on a dependability study (of security, availability and reliability guarantees) as implemented and provided after the WA#2 implementation requirements and the observed operation.

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.

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 the replicas in order to all of them to reach a consensus of a response to return.

2. Security auditing and analysis

2.1 Security properties in End-Client/Service interaction

The communication between the End-Client and the service use TLS with Server Authentication, so the security properties guaranteed are:

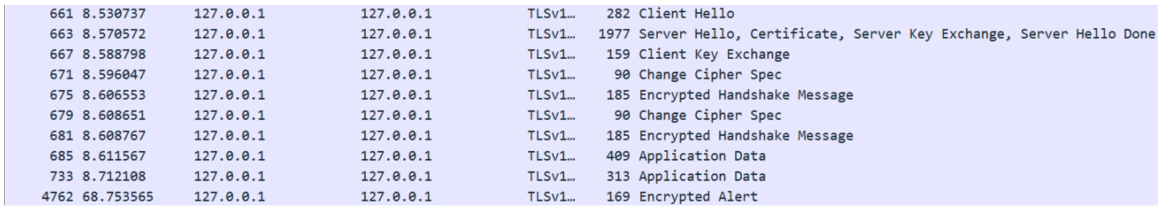
- Message and Data Flow-Integrity.
- Message and Data Confidentiality.
- Peer Authentication and Message Authentication.

2.2 Security settings in TLS for Rest Operations

We configured the server to use TLSv1.2 and use the following cipher suites:

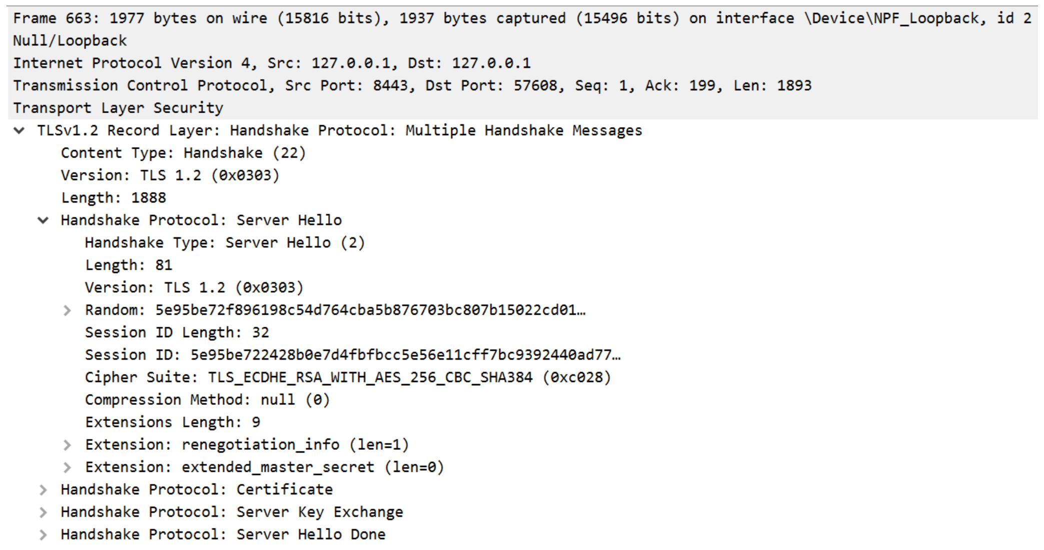
```
TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA384,
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384,
TLS_DHE_DSS_WITH_AES_256_CBC_SHA256,
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA
```

Figure 1, 2 and 3 shows the TLS messages exchanged between the End-Client/Service and that the server imposed the use of TLSv1.2 and the cipher suites mencioned above.



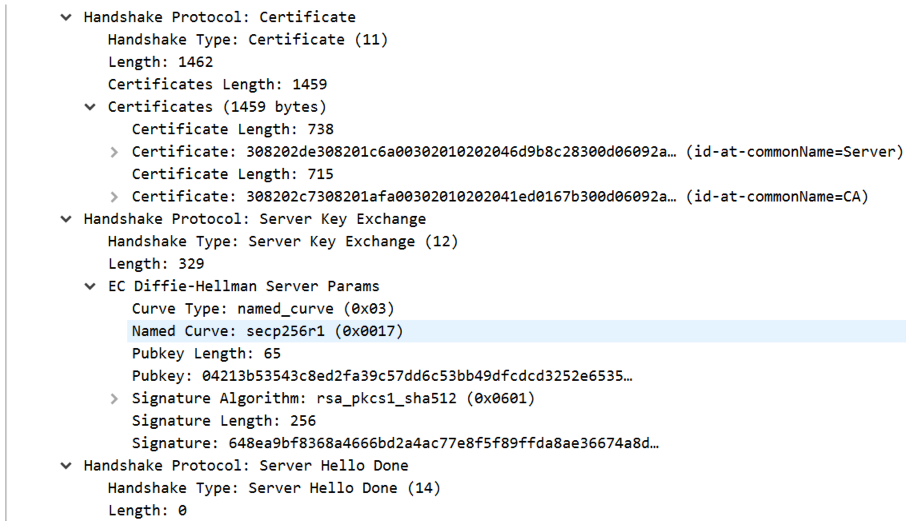
```
  661 8.530737    127.0.0.1     127.0.0.1     TLSv1…   282 Client Hello
  663 8.570572    127.0.0.1     127.0.0.1     TLSv1…  1977 Server Hello, Certificate, Server Key Exchange, Server Hello Done
  667 8.588798    127.0.0.1     127.0.0.1     TLSv1…   159 Client Key Exchange
  671 8.596047    127.0.0.1     127.0.0.1     TLSv1…    90 Change Cipher Spec
  675 8.606553    127.0.0.1     127.0.0.1     TLSv1…   185 Encrypted Handshake Message
  679 8.608651    127.0.0.1     127.0.0.1     TLSv1…    90 Change Cipher Spec
  681 8.608767    127.0.0.1     127.0.0.1     TLSv1…   185 Encrypted Handshake Message
  685 8.611567    127.0.0.1     127.0.0.1     TLSv1…   409 Application Data
  733 8.712108    127.0.0.1     127.0.0.1     TLSv1…   313 Application Data
 4762 68.753565   127.0.0.1     127.0.0.1     TLSv1…   169 Encrypted Alert
```

Figure 1: Wireshark capture of the End-Client/Service interaction



```
Frame 663: 1977 bytes on wire (15816 bits), 1937 bytes captured (15496 bits) on interface \Device\NPF_Loopback, id 2
Null/Loopback
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 8443, Dst Port: 57608, Seq: 1, Ack: 199, Len: 1893
Transport Layer Security
  TLSv1.2 Record Layer: Handshake Protocol: Multiple Handshake Messages
      Content Type: Handshake (22)
      Version: TLS 1.2 (0x0303)
      Length: 1888
    Handshake Protocol: Server Hello
        Handshake Type: Server Hello (2)
        Length: 81
        Version: TLS 1.2 (0x0303)
        Random: 5e95be72f896198c54d764cba5b876703bc807b15022cd01…
        Session ID Length: 32
        Session ID: 5e95be722428b0e7d4fbfbcc5e56e11cff7bc9392440ad77…
        Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384 (0xc028)
        Compression Method: null (0)
        Extensions Length: 9
        Extension: renegotiation_info (len=1)
        Extension: extended_master_secret (len=0)
    Handshake Protocol: Certificate
    Handshake Protocol: Server Key Exchange
    Handshake Protocol: Server Hello Done
```

Figure 2: Wireshark capture of the Server-Hello message



```
    Handshake Protocol: Certificate
        Handshake Type: Certificate (11)
        Length: 1462
        Certificates Length: 1459
      Certificates (1459 bytes)
            Certificate Length: 738
            Certificate: 308202de308201c6a00302010202046d9b8c28300d06092a… (id-at-commonName=Server)
            Certificate Length: 715
            Certificate: 308202c7308201afa00302010202041ed0167b300d06092a… (id-at-commonName=CA)
    Handshake Protocol: Server Key Exchange
        Handshake Type: Server Key Exchange (12)
        Length: 329
      EC Diffie-Hellman Server Params
            Curve Type: named_curve (0x03)
            Named Curve: secp256r1 (0x0017)
            Pubkey Length: 65
            Pubkey: 04213b53543c8ed2fa39c57dd6c53bb49dfcdcd3252e6535…
            Signature Algorithm: rsa_pkcs1_sha512 (0x0601)
            Signature Length: 256
            Signature: 648ea9bf8368a4666bd2a4ac77e8f5f89ffda8ae36674a8d…
    Handshake Protocol: Server Hello Done
        Handshake Type: Server Hello Done (14)
        Length: 0
```

Figure 3: Wireshark capture of the Handshake details

2.3 Replicated Service and Dependability guarantees

2.3.1 Group Communication security (Replica-to-Replicas)

After analysing the BFT-Smart paper and visualizing the communications between replicas with wireshark, we could determine that all replicas communicate through TCP/IP channels using Message Authentication Codes (MAC). The keys used by the MAC to ensure its properties are generated using Diffie-Hellman agreements, signed with RSA keys.

2.3.2 TLS Security settings as provided with BFT-Smart

In the beginning of each session, the client establishes secure channels with each replica using TLS, in order to send the requests to replicate to each replica, using total-order multicast. Each replica also establishes a TLS connection with each other. We could not verify this using wireshark, after many frustrated attempts. The tool was not recognizing the ports at first, so we tried overriding the configuration to recognize the ports as TLS, but without success. Although now we could see the packets marked as SSL in the protocol we could not inspect the contents of TLS layer in those packets. But to verify our claims, we inspected a little bit of the source code (ServerConnection.java, ServersCommunicationLayer.java) and we concluded that they were using SSLSockets.

2.3.3 Security guarantees against data-exposure

With the use of MACs, the message exchange between replicas are provided with message integrity and message authenticity properties. The message integrity gives protection against message tampering attacks, while message authentication property ensures protection against Masquerade attacks. The symmetric keys for the replica-replica channels are generated through the signed Diffie-Helman agreement ensures perfect forward and backward secrecy. Those properties protect against Man-in-the-Middle (MIM) attacks.

Even though the implemented mechanism provide some security insurances, attacks like traffic analysis can't be prevented, since none of them has any type confidentiality properties.

2.4 Correctness observation in case of replica-failure

2.4.1 Experimental observation with no failures/attacks

To test the correctness of the system with no failures we just started 4 replicas normally with one of them being a client benchmark that executed a workload. With that environment we noticed that all operations were executed by all replicas, which was expected since no failures happened.

2.4.2 Experimental observation with fail-stop conditions

For this experiment we started 4 replicas normally with one being a client benchmark that was mentioned above and during the execution of the workload we stopped one of the replicas that was not the client benchmark. Stopping one of the replicas does not affect the correctness of the system, because the quorum 3f+1 supports one replica failure. If we stop 2 replicas the system fails since f>1.

2.4.3 Experimental observation with byzantine failure conditions

To test this last experiment we started 4 replicas with one being a client benchmark that was mentioned above and a byzantine replica that responds always incorrectly. We observed with this environment that the correctness of the system is not affected, but if we have more than one byzantine replica an error occurs since a consensus cannot be achieved.

3. Auditing Conclusions

3.1 Main auditing results

In terms of dependability guarantees we analyzed our implementation assumptions using the reliability, availability and security properties.
In terms of reliability, since the BFT Smart mode used is prepared to tolerate possible faults, our system is only fault tolerant. However, if the BFT Smart client itself is byzantine, the clients wouldn't have any way to ensure that the given answer was the same given by all replicas, so in that case the system wouldn't be able to prevent byzantine faults. During the system execution our write operations are synchronous, therefore some faults can be detected, like omission or timing faults. However, during the read operations, which are asynchronous, the system wouldn't be able to detect this same faults. Our system availability is only measured when all the servers infrastructure is running since the system is on localhost.
The security properties offered by our system are the guarantees given by our communication channels between replicas, between the leader and the replicas, and between the client and the leader. The information storage in the Redis repository is also in plaintext which could compromise private information related to the users.

3.2 Possible improvement of dependability guarantees of the implemented solution

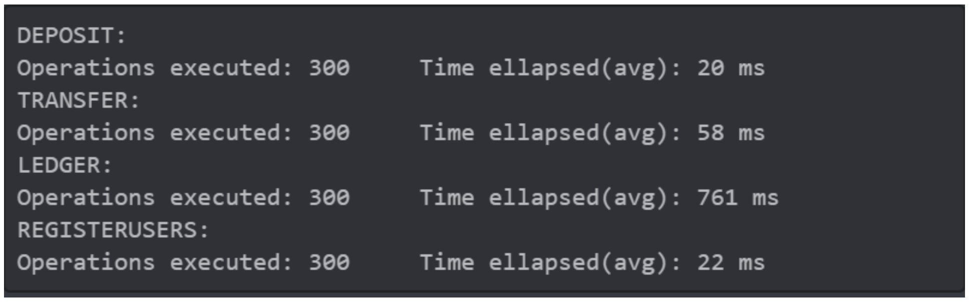
Based on the previous assumptions regarding the dependability properties, we can find some future improvements to take into account in the next implementations.
In order to further improve reliability, a possible solution could be found to solve the previously mentioned problem regarding the request answer authenticity. Also, the system could be made more robust by implementing other fault models. In terms of the system synchronism during write operations

and system asynchronism during read ones, we think there is no need to give much focus, since the throughput might be already closed to ideal.
In terms of availability, a possible improvement would be to run the servers infrastructure in a cloud environment so that the availability wouldn't be limited only to the time frame when the system is being executed in local-host. Also, the servers infrastructure itself could be increased to support more quorums, like a read quorum and a write quorum.
The security properties offered by our system could be improved in matter of confidentiality during the information exchange between servers and the client. Also, when storing the information in a persistent way, the data could be encrypted to prevent future intrusions and access to classified information. In terms of access control, the system could develop a mechanism to limit the operations the clients could execute. That way, the clients would only execute operations they had permission to do, possibly limiting actions from attackers.

4. Performance analysis

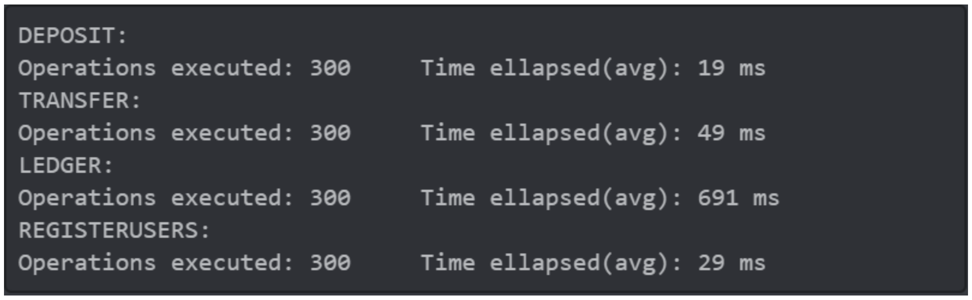
4.1 Throughput and Latency Conditions

Figures {4, 5, 6} show the results of running 300 operations between 1 client and 4 server replicas, during different scenarios.



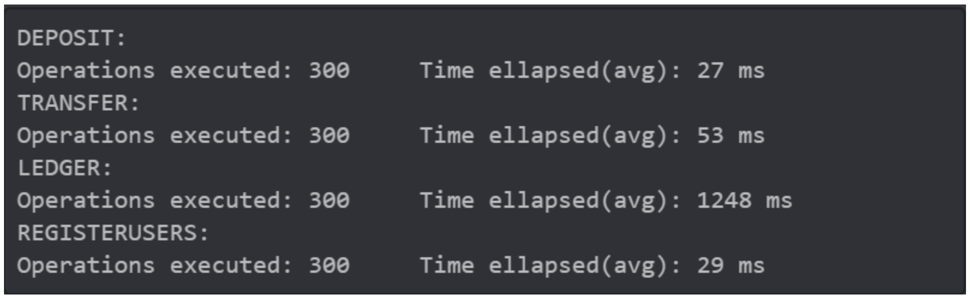
```
DEPOSIT:
Operations executed: 300     Time ellapsed(avg): 20 ms
TRANSFER:
Operations executed: 300     Time ellapsed(avg): 58 ms
LEDGER:
Operations executed: 300     Time ellapsed(avg): 761 ms
REGISTERUSERS:
Operations executed: 300     Time ellapsed(avg): 22 ms
```

Figure 4: Performance test, during normal execution conditions.



```
DEPOSIT:
Operations executed: 300     Time ellapsed(avg): 19 ms
TRANSFER:
Operations executed: 300     Time ellapsed(avg): 49 ms
LEDGER:
Operations executed: 300     Time ellapsed(avg): 691 ms
REGISTERUSERS:
Operations executed: 300     Time ellapsed(avg): 29 ms
```

Figure 5: Performance test, during normal execution, with 1 crash failure.

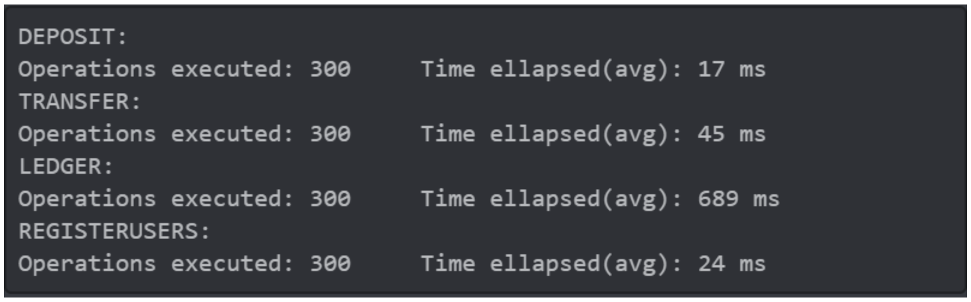


```
DEPOSIT:
Operations executed: 300     Time ellapsed(avg): 27 ms
TRANSFER:
Operations executed: 300     Time ellapsed(avg): 53 ms
LEDGER:
Operations executed: 300     Time ellapsed(avg): 1248 ms
REGISTERUSERS:
Operations executed: 300     Time ellapsed(avg): 29 ms
```

Figure 6: Performance test, with 1 byzantine replica executing.

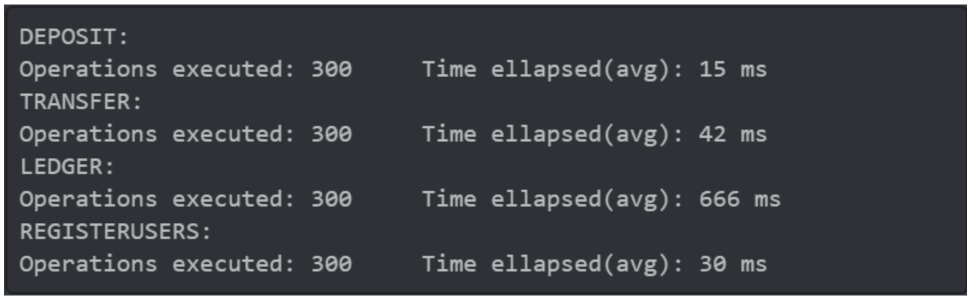
Although we didn't run the tests many times, we can conclude that the ledger operations are expensive, specially when running with a byzantine replica. The remaining operations don't seem to suffer after changing the execution conditions, although in some cases the average elapsed time is higher under failure scenarios.

Figures {7, 8, 9} show the results of running 300 operations between 4 server replicas, during different scenarios with 1 acting as a benchmark server. The purpose of this test scenarios is to display the amount of time consumed to order operations, using the BFT-SMaRt library. As before, we can conclude pretty much the same results. Ledger operations are the most expensive, while the others have small impact. The results are quite similar with the first 3, and this is because the client and replica servers are running on the same machine, which minimizes the latency between the https interactions rendering it almost insignificant in this comparison.



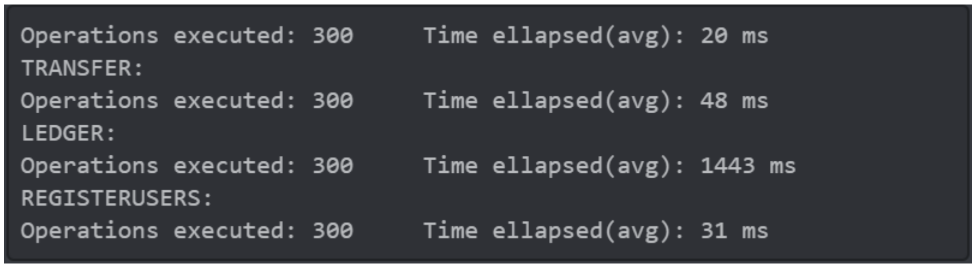
```
DEPOSIT:
Operations executed: 300     Time ellapsed(avg): 17 ms
TRANSFER:
Operations executed: 300     Time ellapsed(avg): 45 ms
LEDGER:
Operations executed: 300     Time ellapsed(avg): 689 ms
REGISTERUSERS:
Operations executed: 300     Time ellapsed(avg): 24 ms
```

Figure 7: Performance test on replicas, during normal execution conditions.



```
DEPOSIT:
Operations executed: 300     Time ellapsed(avg): 15 ms
TRANSFER:
Operations executed: 300     Time ellapsed(avg): 42 ms
LEDGER:
Operations executed: 300     Time ellapsed(avg): 666 ms
REGISTERUSERS:
Operations executed: 300     Time ellapsed(avg): 30 ms
```

Figure 8: Performance test on replicas, during normal execution, with 1 crash failure.



```
Operations executed: 300     Time ellapsed(avg): 20 ms
TRANSFER:
Operations executed: 300     Time ellapsed(avg): 48 ms
LEDGER:
Operations executed: 300     Time ellapsed(avg): 1443 ms
REGISTERUSERS:
Operations executed: 300     Time ellapsed(avg): 31 ms
```

Figure 9: Performance test on replicas, with 1 byzantine replica executing.

4.2 Optimization

In this version the operation that takes more time is the ledger, the times we registered are high because we are getting the ledgers from Redis and sometimes it responds with a list in a different order and then the replicas would fail to reach a consensus since the values were different. To overcome this problem we sort the list before returning that is the reason that operation is so costly. So the main improvement would be to find a way to fix this problem without sorting the lists in every replica.