University of Padua
Department of Mathematics "Tullio Levi-Civita"

Master degree in Computer Science

Load testing of distributed applications: the Signal case study

Master candidate:
Matteo Marchiori
matteo.marchiori.4@studenti.unipd.it

Supervisor:
Tullio Vardanega
tullio.vardanega@unipd.it

Academic year 2020–2021

Abstract

This thesis talks about the load testing of distributed applications, and how it can be used to check their scalability. My goal is to obtain some resources which can be used to test cloud based applications with different architectures, but common characteristics.

A category of distributed application that everyday we use is the one of chat applications to send messages and to keep in contact with people. The most known and used around the world are WhatsApp, Telegram and Facebook Messenger [1].

An interesting example of chat application is Signal [2], because it is fully open source. This means that also the source code on the server side is available and deployable, which is not common among distributed applications.

The architecture of Signal needs to be scalable in order to provide a good service to its users, but it cannot be taken for granted.

What I do in the thesis is to design some load tests and use them on Signal. This is done to test its scalability in specific scenarios, which can be compared to real situations, and to have some tests that can be used on similar applications to get details about their scalability.

The applications which I refer to are cloud-based applications, so distributed applications, which communicate to the world using the HTTP protocol. Signal is used because it is open source, so it provides a lot of information about its architecture, but the experiments are designed to be used also in other similar applications.

The thesis is organized as follows: the first chapter (see chapter 1 on page 1) states the problem, so it goes more into the details of scalability, load testing, related works and the faced problem. The second chapter (see chapter 2 on page 5) talks about the solution space, where there is a description of the space explored with the experiments and what has not been covered. The third chapter (see chapter 3 on page 11) talks about the experiments, the metrics and the obtained results. The last chapter (see chapter 4 on page 25) is about a conclusion to the problem and possible outlooks.

ACKNOWLEDGEMENTS

I thank Professor Vardanega for his patience and for the invaluable support provided in the creation of this thesis, despite my black periods.

I thank mum for always supporting me over this long period. Thanks for your strength and hugs.

I thank dad for always being here, for his kindness and for being open to a comparison with my ideas.

I thank Alberto, that will always be here, pandemics or meteors. David Murphy is nothing compared to you.

I thank the old friends rediscovered and the new ones, especially Celeste, Cristi, Gianluca, Giovanni, Mirko and Sarah, without whose support I would have already dropped out of university.

I thank the people who I lost and the ones who abandoned me, for making me understand how important being present for someone can be.

I thank all the people who have welcomed me and taught me something in these two years, in particular true resilience and true consistency with their own ideas. Thanks to Matteo Gracis, to all the members of the NoBufale.it team and to the Students against the Green Pass.

I thank Professors Andrea Camperio Ciani and Maristella Agosti for the life lessons, I will carry them in my luggage.

Padova, 16 december 2021 Matteo Marchiori

CONTENTS

Abstract iii

Acknowledgements v

Contents vii

List of Figures ix

List of Tables xi

1 Problem Statement 1

2 Solution space 5

2.1 Covered points 5

2.2 Architecture 5

2.3 Experiments 6

2.4 Uncovered points 9

3 Experimental results 11

3.1 Metrics 11

3.2 Environment characteristics 11

3.2.1 Server characteristics 12

3.2.2 Client characteristics 13

3.3 Details about implementation 13

3.3.1 Steady load settings 13

3.3.2 Peak load settings 15

3.3.3 Perfmon 16

3.4 Results 17

3.4.1 New user registration 17

3.4.2 New message sending 21

4 Conclusions and outlook 25

4.1 Observations 25

4.2 Outlook 27

Bibliography 29

LIST OF FIGURES

Figure 1 Signal Server general architecture 5
Figure 2 JMeter steady load for accounts creation 14
Figure 3 JMeter steady load for message sending 14
Figure 4 JMeter peak load for accounts creation 15
Figure 5 JMeter peak load for message sending 16
Figure 6 JMeter Perfmon plugin 17
Figure 7 Steady load accounts' creation on Signal 4.97 18
Figure 8 Peak load accounts' creation on Signal 4.97 19
Figure 9 Steady load accounts' creation on Signal 6.13 20
Figure 10 Peak load accounts' creation on Signal 6.13 21
Figure 11 Steady load message sending on Signal 4.97 22
Figure 12 Peak load message sending on Signal 4.97 23
Figure 13 Steady load message sending on Signal 6.13 24
Figure 14 Peak load message sending on Signal 6.13 24

LIST OF TABLES

Table 1 Sequence of calls to register a new account 17
Table 2 Steady load accounts' creation on Signal 4.97 18
Table 3 Peak load accounts' creation on Signal 4.97 19
Table 4 Steady load on Signal 6.13 20
Table 5 Peak load accounts' creation on Signal 6.13 21
Table 6 Sequence of calls to send a message 21
Table 7 Steady load message sending on Signal 4.97 22
Table 8 Peak load message sending on Signal 4.97 22
Table 9 Steady load message sending on Signal 6.13 23
Table 10 Peak load message sending on Signal 6.13 24

1 PROBLEM STATEMENT

Scalability is a challenge that all the applications used from a lot of people need to face. Cloud applications are designed to satisfy a high number of requests, from lots of users.

We can give the following definition of scalability:

> [...]the ability to handle increased workload by repeatedly applying a cost-effective strategy for extending system capacity, without intolerable latency or excessive waste. [3]

We can achieve the scalability with the following dimensions [4]:

FUNCTIONAL DECOMPOSITION : scalability by splitting different things, from a functional point of view. It is obtained by the orchestration of components.

HORIZONTAL REPLICATION : scalability by cloning the same thing. It is obtained by statelessness of components.

DATA PARTITIONING : scalability by splitting similar things. This is used with orchestration and statelessness.

In order to test the scalability of an application it is possible to use load tests. This kind of tests are built starting from real situations, where you verify the behavior of the distributed application with the goal of understand if its architecture is designed to face the scenarios. Moreover, it is possible to look for its breakpoints and see how it reacts to these situations.

The scientific literature provides numerous examples of this kind of tests on distributed architectures with different goals.

A first example is from Avritzer [5], where authors present an approach to compare different configurations of a microservice system to get the best among them.

They use a metric called Domain-based, which is obtained using these steps:

1. collection of operational data of the analyzed system, which are used to obtain an operative profile;
2. analysis of the data (probability of a specific load, groups of these probabilities and analysis of the frequencies);
3. generation of the experiment (determined by the group loads, the configuration of the architecture and from a baseline used to understand if the test is passed or not). From more experiments a test case is produced;
4. computation of the baseline used to understand how many systems have the requisites of scalability based on the test cases;
5. execution of the experiment (fraction of executions of the services which pass the test with the given configuration on the total of the executions);
6. computation of the metric multiplying the previous sum with the probability of the given load.

The design of the metric is useful to highlight how the configuration of the system (which is how many resources are usable) has an influence on the performance obtained on the collected operational profiles.

On another work from Trubiani [6] authors use a method to look for weak points on the source code of the analyzed system which can bring to worse performance. As on the previous case operational profiles are used to generate load tests, which in this work are used to find antipatterns.

The proposed steps are the following:

1. use of the load tests on the system;
2. manual analysis of performance issues;
3. rules development to find the antipattern;
4. fix of the antipattern;
5. repetition of the load tests with the applied fix.

This work is interesting as the results report for the better performances obtained after the antipattern resolution.

Davatz and other authors [7] focus their attention on the cost of IAAS and on their comparison. Okta is the framework that they present and it evaluates the infrastructure on the number of completed requests per second (SRPS) with each configuration, by using different benchmarks. They derive also a metric based on the dollar value. The interesting point is this metric performance/cost, which is used to obtain a ranking of the IAAS providers.

Another work which I have found interesting is the following one [8]. Authors of this work talks about a method to evaluate the scalability of a distributed system, by using the platform Scalar. They use a configuration file to give some feature that are transformed into code and that can be activated or deactivated. In order to execute the evaluation it is selected a set of features, used to run the experiments.

The last example that I report is about the resilience of microservices architectures [9]. No real load is generated on the system, failures are produced on the system by using a proxy system agnostic on the used technologies, which cause configured operations on messages among the microservices.

Gremlin is useful to apply Chaos engineering principles[1], so to evaluate the behavior of the system under stress conditions caused by using experiments.

These and other works inspired this thesis work. I focus on the scalability problem on Signal, a chat application with open source code.

In particular, I want to understand the behavior of the Signal architecture in the face of steady loads and of peak loads, and build a battery of load tests which can be reused as a base to test similar architectures.

I chose Signal because it is open source, so it allows to get all the information to get deep into its internal architecture. This is not a common thing among chat applications.

The final goal of this thesis is to obtain some useful resources to understand the behavior under load tests of distributed applications. Signal is used as a case study, in particular some of its functions, but the tests are designed to be reused with similar cloud applications in order to check their behavior, with some modifications on the configurations of the tests.

The reason is to understand that the theoretical scalability requirements are respected in reality. So the proposal is a set of configurable tests in this sense.

Applications like Signal face a lot of traffic every day, they are used from a huge number of users. Moreover, the generated traffic is not linear and easy to manage.

Some examples about peaks of traffic are the ones generated for work or during particular holidays. The best example is the one of Christmas holiday wishes, when in a short period of time a lot of traffic is generated [10].

1 https://principlesofchaos.org

This means that the architecture should manage different situations of traffic, and scale up or down in the correct way based on the number of requests.

In other cases the traffic generated from Signal if much lower and distributed in time, because a lower number of users chats on it.

Resources that are needed to afford huge traffic cost money. This is the reason why elastic scalability systems are needed to acquire and release resources on the right time, to reduce the waste of money in the best possible manner.

On the other way, it is also necessary to maintain the correct level of resources needed to afford any number of requests. This is done by expanding and retrieving the available resources.

The scalability is that property which allows an application to face a number of requests which is not known in advance. This is the reason why the application should be able to receive them and give a response, also if the number of these requests increase during time.

In theory the scalability can be applied to an unlimited number of requests. This is not possible in reality, because the resources of a system are always limited. For this reason scalability of an application is always to be considered with an upper limit.

For this reason it is useful to evaluate the scalability of an application with load tests, in order to understand if its architecture respect the theoretical requisites also in the real world.

On the load tests we simulate real situations' behavior, in order to understand how the application faces it.

The problem becomes how to design and implement a set of significant load tests, which can cover the real situations that the application can face.

Starting from these characteristics we will implement the load tests which will put under load the Signal architecture.

The final goal of this thesis is to analyze how the Signal server reacts to high load situations. I start from the case study of an outage happened between the 15th and the 17th January 2021 as a reference [13, 14], by comparing the 4.97 architecture, the one in use at the time of the outage, with the 6.13 version, which is the most recent one available when I started the experiments.

What we demonstrate is that the evolution of the architecture with a change on its components is good to avoid some effects of huge loads in short time periods, but it can be less suited for long term loads effects. Moreover, changes apported are useful to delegate the management of the database system and other services used by the server, so their performance must respect the ranges offered by AWS[2].

2 https://aws.amazon.com

2 SOLUTION SPACE

In this chapter we explain the explored solution space, so which points are covered of the Signal architecture and what has been proved by the experiments, and the uncovered space, which are some points excluded in the experiments.

2.1 COVERED POINTS

In this thesis I explore only a part of Signal, not the whole application, but a very important part of it. The focus is on the server side of Signal, the experiments will test only that, excluding the client side.

In particular, I will analyze the components of the architecture which are involved into the operations for the account creation and for the sending of messages, that are the principal aspects of a chat application like Signal. Some parts of the architecture which are not used for these functions but for other services, for example sending attachments, making calls and video calls, are not analyzed.

2.2 ARCHITECTURE

The architecture of the Signal server is represented in the following schema [12]:

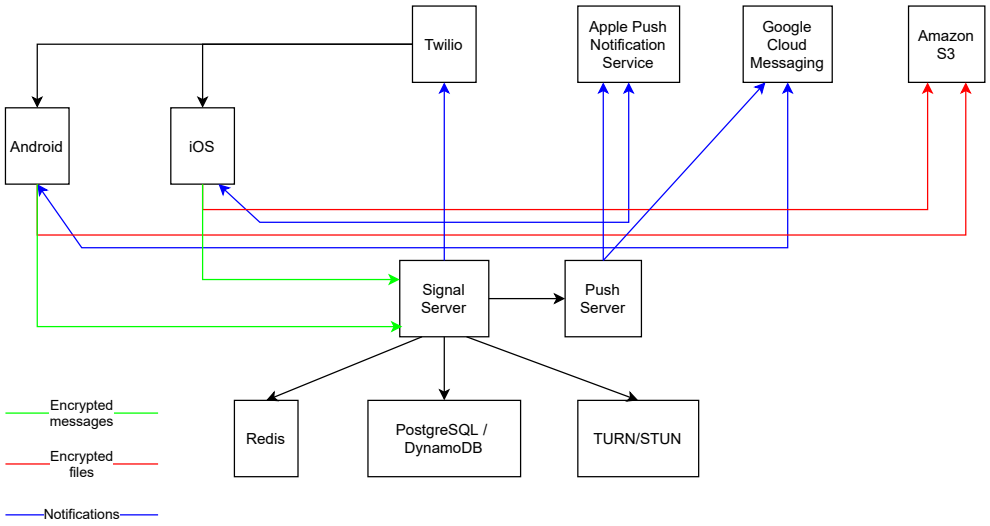

Figure 1: Signal Server general architecture

The main differences between Signal 4.97 and Signal 6.13 are the use of DynamoDB as the main database instead of PostgreSQL and the use of AWS AppConfig[1] to retrieve dynamic configuration data.

2.3 EXPERIMENTS

The experiments I made can be divided into:

STEADY LOAD EXPERIMENTS which use a base load slowly incremented over time in order to understand a measure of resilience of the architecture;

PEAK LOAD EXPERIMENTS which use a high load in a short time to understand the behavior of the architecture in this kind of conditions.

For both of them I used a sequence of APIs call of Signal which reflects the ones made by an Android client in order to register a new account and in order to send a message from an account to another.

All the experiments have been executed both on Signal 4.97 and on Signal 6.13 in order to do a comparison between the results obtained with the two versions of the server. The 4.97 version was used in production when an outage happened on the 15th January 2021, while the other version was the last available one when I started the server deployment.

In order to get the sequences of the calls made to the server from the load generator I used a real Android client, with a modified version of the code to log the calls made to the server in the correct order.

1 https://docs.aws.amazon.com/appconfig/latest/userguide/what-is-appconfig.html

The following code is the modified method in the class `PushServiceSocket.java` to get the sequence of the calls which have been covered.

```
private String makeServiceRequest(String urlFragment, String method,
↪  String jsonBody, Map<String, String> headers, ResponseCodeHandler
↪  responseCodeHandler, Optional<UnidentifiedAccess>
↪  unidentifiedAccessKey)
    throws NonSuccessfulResponseCodeException, PushNetworkException
{
    System.out.println();
    System.out.println();
    System.out.println();
    for (Map.Entry<String, String> header : headers.entrySet()) {
        System.out.println("header-key: "+header.getKey());
        System.out.println("header-value: "+header.getValue());
    }
    if(urlFragment!=null) System.out.println("urlFragment:
    ↪  "+urlFragment);
    if(method!=null) System.out.println("method: "+method);
    if(jsonBody!=null) System.out.println("jsonBody: "+jsonBody);
    if(headers!=null) System.out.println("headers:
    ↪  "+headers.values().toString());
    if(unidentifiedAccessKey!=null)
    ↪  System.out.println("unidentifiedAccessKey:
    ↪  "+unidentifiedAccessKey.toString());
    System.out.println();
    System.out.println();
    System.out.println();
    ResponseBody responseBody = makeServiceBodyRequest(urlFragment,
    ↪  method, jsonRequestBody(jsonBody), headers, responseCodeHandler,
    ↪  unidentifiedAccessKey);
    try {
        return responseBody.string();
    } catch (IOException e) {
        throw new PushNetworkException(e);
    }
}
```

Instead, in order to get the architectural components of Signal which are used from the previous calls I analyzed the source code of the 6.13 version of the Signal server, and I obtained some UML sequence diagrams that made it easier to understand the involved components[2].

More in detail on the design of the experiments, what I do is to create some situations of interests inspired to real scenarios.

Usually these kinds of test are designed by using real data of use, provided by logs or other sources and from configurations of the application itself. In our experiments we have the components of the architecture, but we don't know any data relative to the load of the real system.

One of the possible options in this case is to ask some sample data to the people who manage the real application used in production. I did not do this for time and privacy reasons. Signal is an application well known for the security of data of users who use it. Obviously load data can be aggregated data, but maybe some of this kind of information can help in doing things that the owners of the application would not appreciate, so I decided to not ask them.

Another option is to verify the number of users who downloaded the application from the Play Store[3] and from the App Store[4], and to do an evaluation similar to the real situation on the number and on the behavior of the users. I did not choose this option for time and cost motivations, it results very difficult for a master thesis project to rent on AWS the resources to simulate a real Signal server system.

This is the reason why we used a reduced scale system, which is easier to put under load, also with a limited number of requests, and which can be useful to give some interesting points about the weak points of the used architecture.

To load test the Signal server architecture we treated it as a black box system, in order to apply the possible results to other similar architectures.

Signal offers REST APIs which are used from its clients to exchange information with the server. We took into consideration a subset of those APIs, used for the creation of accounts and for sending messages.

As a consequence only some components of the architectures have been involved in the tests [11, 12], such as the Dropwizard[5] application itself, Redis[6], PostgreSQL[7] and DynamoDB[8] (only on the newer version of the Signal server).

2 https://drive.google.com/drive/folders/1G5RfzsEX8zLyv_yoq2h5FOEC-pLG0TPQ?usp=sharing
3 https://play.google.com/store
4 https://www.apple.com/app-store
5 https://www.dropwizard.io/en/latest
6 https://redis.io
7 https://www.postgresql.org
8 https://aws.amazon.com/dynamodb

What have been done in the thesis project is:

1. extrapolate the components used by each API call from the Signal source code;
2. find the components which are overloading the server;
3. compare the two server versions between them.

All the experiments are scaled down to the resources that I could afford. For the server side I used the ones provided from the AWS free tier[9], with some expenses for the traffic and for DynamoDB resources. For the client side I used my laptop. These resources have a detailed description on the section 3.2 on page 11.

With steady load we mean a load with a long duration in time which simulate the use of the application without relevant events. It gets increased over time.

With peak load we mean a load with a short duration in time, which is used to simulate a particular event. For example, it can be the New Year's Eve when a lot of messages are sent in a limited period of time.

I give a more detailed description of these loads on the experimental section (see chapter 3 on page 11).

2.4 UNCOVERED POINTS

The following points are not covered from the thesis:

- test of the architecture components used to send, receive and store multimedia contents different from simple messages;
- test of the components used for voice calls and video calls;
- test in a real world scale environment (only in a limited version);
- test with repetition from real world clients (only with load systems).

9 https://aws.amazon.com/free

3 EXPERIMENTAL RESULTS

In this chapter I describe the design of the experiments made to load test the Signal server architectures and I report the obtained results.

3.1 METRICS

The metrics considered the response times of the system under load tests.

To make the implemented tests usable in a black box way, avoiding any kind of knowledge regarding the internal operations of the server, I only used the APIs offered by the server itself.

The metrics used for the requests are the following ones:

- average response time per request type;
- median response time per request type;
- 90th, 95th and 99th percentile response time per request type.

The metrics used to measure the server load are the following ones:

- percentage of memory used per second;
- percentage of CPU used per second;
- network I/O (B/s).

With these metrics it is possible to verify which requests require more time to get a response and which ones have a higher impact on the server load.

It is possible to know which are the most used components of the architecture and which requests are a weak point for the system by understanding the interactions among them.

3.2 ENVIRONMENT CHARACTERISTICS

Here there are some details for the reproducibility of the experiments, so a technical description of the server used and of the client used on them.

3.2.1 Server characteristics

The server side used to run the experiments is a fully scaled reproduction of the Signal server from their source code, apart from some components, for example the one used in production from Signal in order to make possible an encrypted contact discovery at hardware level [15].

The deployment of the server side has been done on the AWS infrastructure, because it is used on the most recent version of the Signal server and on less recent version for some provided services, such as AWS SQS[1] and DynamoDB.

I deployed the server to a t3.micro EC2 instance[2], which is included in the free resources plan of AWS. It provides:

- 2 vCPUs of burstable type;
- 1 GB of RAM;
- 30 GB of SSD.

The software level characteristics are listed here:

- OS: Ubuntu 20.04.3 LTS[3]
- Dropwizard 2.0.13 (Signal 4.97) / 2.0.22 (Signal 6.13)
- Redis server v=5.0.7
- PostgreSQL 12.9
- Nginx 1.18.0[4]

In order to make it reachable from the outside I used ClouDNS[5], which provides a free dynamic DNS for subdomains. The name I chose is `signal.cloudns.cc`

Between the Signal server 4.97 version and the 6.13 version there are some performance differences that are highlighted from some results of the experiments.

The older version use less components of the AWS architecture (DynamoDB and AppConfig are not used). This means that the newer version of the server, which use less the PostgreSQL database has less load on its own resources and delegates the load to the IAAS offered by Amazon.

1 https://aws.amazon.com/sqs
2 https://aws.amazon.com/ec2/instance-types
3 https://ubuntu.com
4 https://www.nginx.com
5 https://www.cloudns.net

3.2.2 Client characteristics

In order to perform the test it has been used JMeter [16], a general purpose tool used for load testing analysis. To perform the tests I used my laptop, which was enough for the characteristics of the server side.

The resources of the laptop are the following:

- Intel(R) Core(TM) i3-4005U CPU @ 1.70GHz;
- 4GB of DDR3 RAM;
- 500GB of SSD.

The software level characteristics are listed here:

- OS: Ubuntu 21.04
- Apache JMeter 5.4.1

3.3 DETAILS ABOUT IMPLEMENTATION

To implement the load experiments I used JMeter, a Java tool made to generate a huge number of requests with lots of parameters.

3.3.1 Steady load settings

With steady load we mean a low load with a long duration, used to simulate the use of Signal without particular events. It gets incremented over time in order to test the resilience of the server, until it reaches a breakpoint.

The test is made from a number of users which starts from 1 and gets incremented by 1 every 2 seconds. These users execute a sequence of operations for account creations or to send messages.

Here I report the settings used to generate a steady load for an undefined period of time on the server using JMeter.

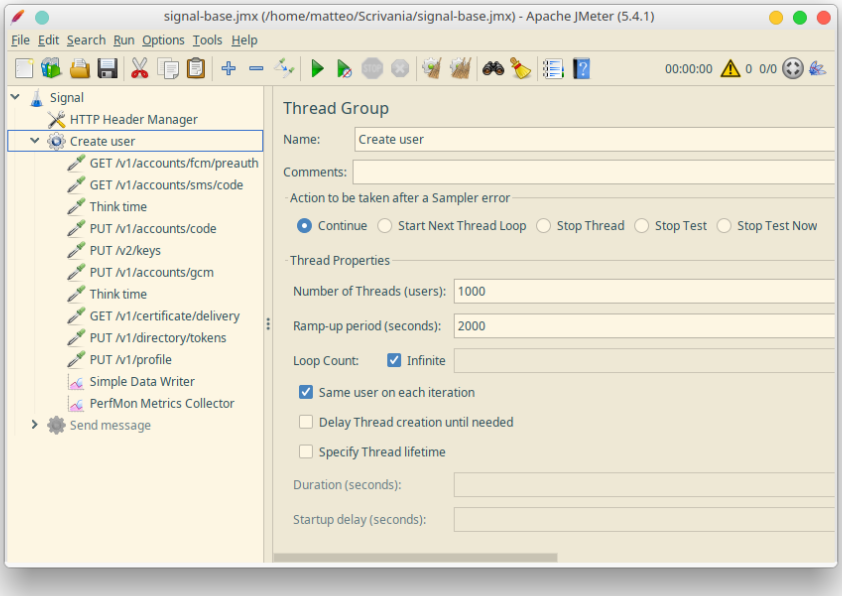

Figure 2: JMeter steady load for accounts creation

As the figure shows, the test performs one account creation for each user, with a number of threads at a time which gets incremented slowly. JMeter proceeds until it gets a manual interruption.

The same thing is done for the message sending steady load, as the following figure shows.

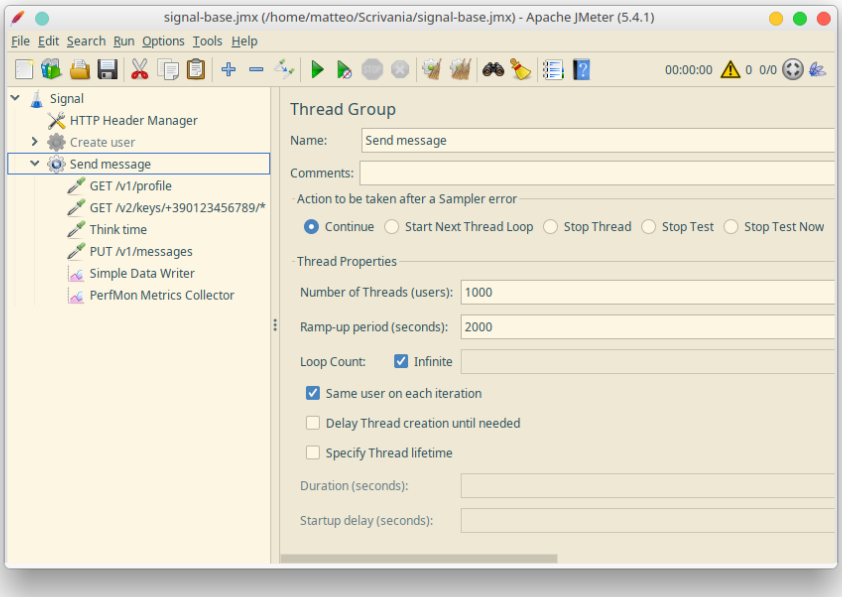

Figure 3: JMeter steady load for message sending

3.3.2 Peak load settings

With peak load we mean a load much higher than the steady one, with a low duration, used to simulate a higher use of the Signal application. An example can be the New Year's Eve, where a lot of messages are sent in a specific period of time.

The load is high enough for the server because of the low capacity. There is a limit of 200 parallel users, who execute sequences of requests to the Signal APIs to create accounts or to send messages. Each user is added after 0.5 seconds (ramp-up of 50 seconds) and the experiment is repeated only once.

In order to perform a higher load test, for a shorter period of time, I used the following settings. This time the maximum concurrent threads is set to 200, with a ramp-up period of 50 seconds (a thread is added every 0.5 seconds), with 1 cycle.

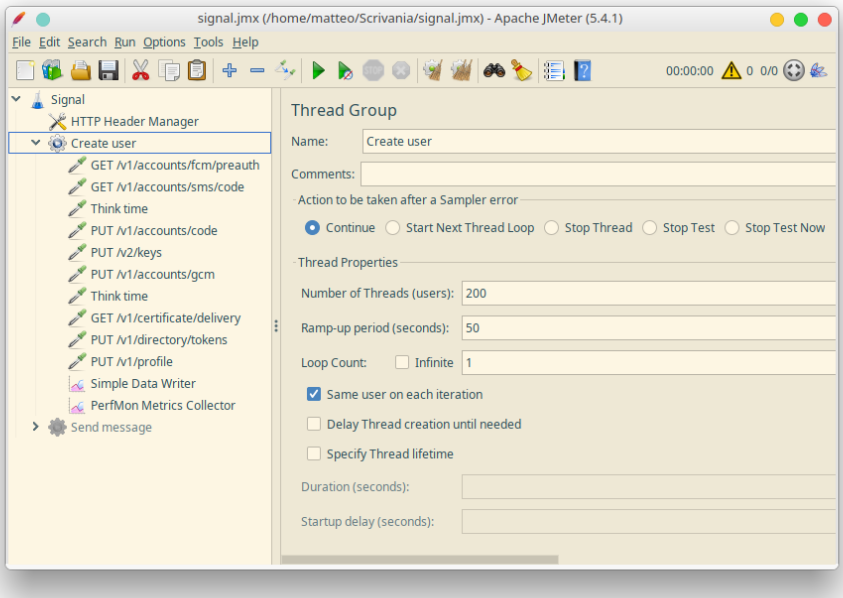

Figure 4: JMeter peak load for accounts creation

The same thing has been done for the message sending.

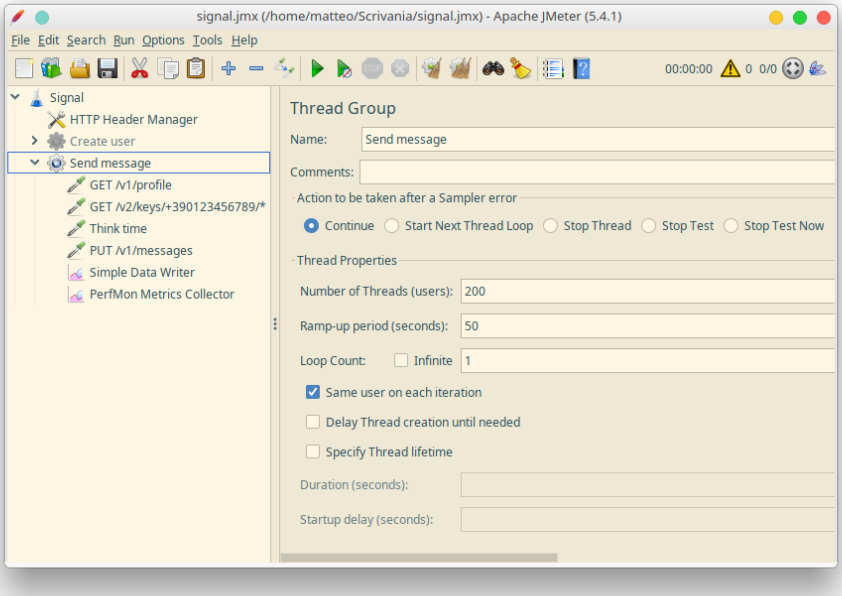

Figure 5: JMeter peak load for message sending

3.3.3 Perfmon

Perfmon[6] is a plugin made for JMeter which is used to collect metrics about the resources of the machine which is tested. I used it to collect data about the CPU usage, the RAM usage and the network performance during the tests.

6 https://jmeter-plugins.org/wiki/PerfMon

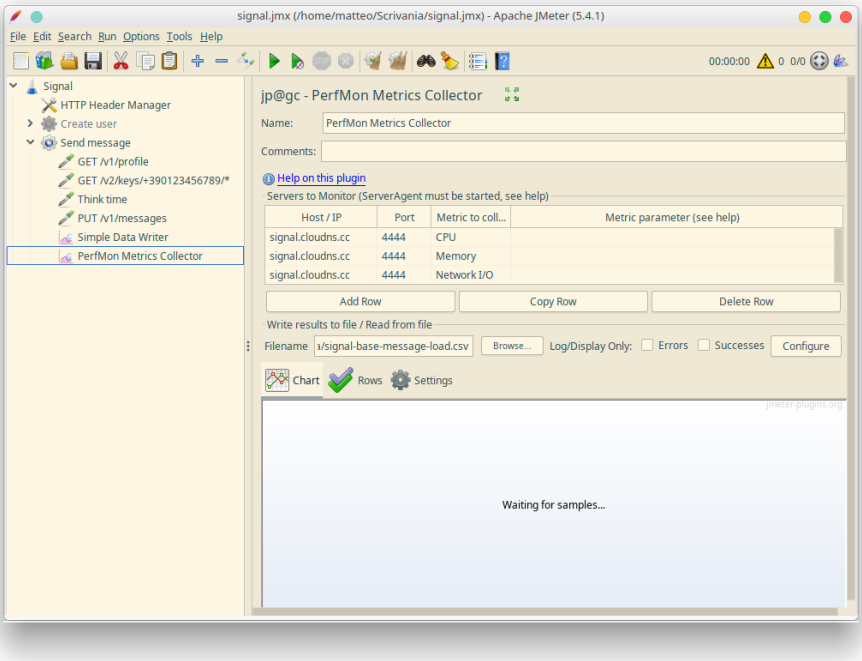

Figure 6: JMeter Perfmon plugin

3.4 RESULTS

In this section I report the measures found by the test executions on both of the server versions, and some observations on them.

3.4.1 New user registration

All the Signal APIs are managed by the Dropwizard controllers, the framework used in order to implement the server orchestrator.

The experiment uses the following calls to the REST APIs of the Signal server:

Type of request	Request path	Description	Involved components
GET	/v1/accounts/fcm/preauth	Request a preauth to Firebase Cloud Message	Dropwizard, Redis, PostgreSQL/DynamoDB, GCM
GET	/v1/accounts/sms/code	Request an OTP code, sent using Twilio	Dropwizard, Redis, PostgreSQL/DynamoDB, GCM, Twilio
PUT	/v1/accounts/code	Send the received code from Twilio	Dropwizard, Redis, PostgreSQL/DynamoDB, AWS SQS
PUT	/v2/keys	Send the keys for encryption protocol	Dropwizard, Redis, PostgreSQL/DynamoDB
PUT	/v1/accounts/gcm	Send details for Google Cloud Message	Dropwizard, Redis, PostgreSQL/DynamoDB, AWS SQS
GET	/v1/certificate/delivery	Get certificate from the Signal server	Dropwizard, Redis, PostgreSQL/DynamoDB
PUT	/v1/directory/tokens	Put tokens of Signal account	Dropwizard
PUT	/v1/profile	Put information of Signal account	Dropwizard,Redis, PostgreSQL/DynamoDB

Table 1: Sequence of calls to register a new account

Between the request for the Twilio OTP code and its insertion and between the insertion of the GCM data and the insertion of the profile data there are two intervals of 3 seconds to simulate the user time to do an action with precompiled fields.

Steady load on Signal 4.97

From this experiment I expect to find some results which confirm that Signal 4.97 is easier to overload compared to the 6.13 version. This is because the 4.97 version was the one used in production at the time of the outage. So the response times should be high, and the experiment duration should be shorter than the one in 6.13 version.

Here there is a table with the measures obtained from the accounts' creation on Signal 4.97.

Request	#Samples	Response times (ms)							Network (KB/s)	
		Average	Min	Max	Median	90th pct	95th pct	99th pct	Received	Sent
GET /v1/accounts/fcm/preauth	3916	1709.69	17	96872	654.50	1959.50	3051.05	38072.90	3.70	3.19
GET /v1/accounts/sms/code	3858	1415.33	17	95211	664.00	2029.10	2308.10	5377.15	7.48	2.15
GET /v1/certificate/delivery	3677	1631.82	16	93170	845.00	2197.00	2787.20	6742.70	6.16	1.99
PUT /v1/accounts/code	3834	2923.34	17	94035	750.00	2059.00	3053.75	90726.65	4.17	3.93
PUT /v1/accounts/gcm	3683	1139.68	17	94560	784.00	2246.60	2693.80	4371.12	3.04	3.37
PUT /v1/directory/tokens	3652	1082.61	38	95104	802.50	2159.40	2514.75	4523.02	3.21	21.19
PUT /v1/profile	3648	949.07	21	94376	732.50	1942.10	2263.55	4689.34	3.09	4.31
PUT /v2/keys	3747	2537.98	47	95779	785.00	2203.80	2839.00	92842.44	3.34	28.59

Table 2: Steady load accounts' creation on Signal 4.97

From these data we can see that the calls used to save the keys request a higher time compared to the other requests. This delay is due to the higher weight of the message, which contains a lot of keys, so it is heavier compared to the other messages.

All the requests which require read/write operations on a database involve Redis, which can trigger requests to PostgreSQL. This is useful, Redis act as a database in RAM, so it is more efficient compared to PostgreSQL.

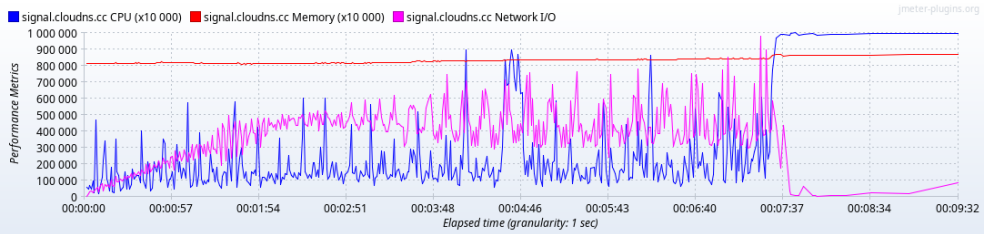

Figure 7: Steady load accounts' creation on Signal 4.97

From the diagram it is visible that the steady load increases gradually over time, with the traffic load. At 7 : 30 the server reaches a breakpoint, the traffic goes to 0 and the usage of CPU is stable at 100%.

Peak load on Signal 4.97

Comparing the situation to the previous experiment, here I do not expect to find an outage on the server. This time the results should highlight how the server reacts to the peak of load, and it is expected that they are worse compared to the 6.13 version.

Here I show the data obtained from the peak load for the accounts' creation.

Request	#Samples	Response times (ms)							Network (KB/s)	
		Average	Min	Max	Median	90th pct	95th pct	99th pct	Received	Sent
GET /v1/accounts/fcm/preauth	200	86.13	65	424	80.50	97.90	114.90	254.35	1.91	1.80
GET /v1/accounts/sms/code	200	23.87	18	90	22.00	26.90	40.00	65.89	4.17	1.22
GET /v1/certificate/delivery	200	23.17	17	146	21.00	25.00	36.95	68.94	3.57	1.18
PUT /v1/accounts/code	200	23.87	19	165	22.00	24.00	41.75	72.96	2.17	2.28
PUT /v1/accounts/gcm	200	21.96	18	75	21.00	24.00	28.90	51.98	1.78	1.99
PUT /v1/directory/tokens	200	44.56	38	161	42.00	47.00	62.85	99.91	1.89	12.55
PUT /v1/profile	200	29.61	23	99	27.00	33.00	49.00	79.78	1.83	2.56
PUT /v2/keys	200	59.21	47	113	57.00	69.90	73.95	110.88	1.78	16.85

Table 3: Peak load accounts' creation on Signal 4.97

The table shows that the response times are lower compared to the ones obtained with the steady load. This is because a breakpoint is not reached, and all of the requests receive a response from the server.

The requests with a higher response times are the preauth ones.

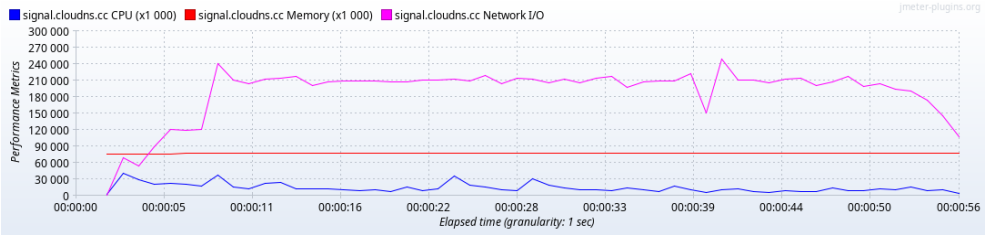

Figure 8: Peak load accounts' creation on Signal 4.97

The diagram shows that there is no overloading on the server, and all the requests receive a response.

Now I compare the obtained results with the 6.13 server version.

Steady load on Signal 6.13

In this experiment I expect that the results are worse, compared to the previous version. The experiment, as for the version 4.97, is designed to create a steady load which slowly increments over time.

Here I report the results for the accounts' creation on Signal 6.13.

Request	#Samples	Response times (ms)							Network (KB/s)	
		Average	Min	Max	Median	90th pct	95th pct	99th pct	Received	Sent
GET /v1/accounts/fcm/preauth	1323	234.66	23	15185	97.00	382.60	487.60	1528.00	3.21	2.89
GET /v1/accounts/sms/code	1312	175.04	19	13576	74.50	315.70	405.35	586.09	6.25	1.95
GET /v1/certificate/delivery	1268	448.49	18	14760	80.50	356.00	480.95	13488.77	5.96	1.83
PUT /v1/accounts/code	1307	334.37	18	14909	85.00	346.20	448.20	12462.56	3.72	3.63
PUT /v1/accounts/gcm	1277	227.33	17	15074	73.00	323.00	405.50	930.90	2.92	3.13
PUT /v1/directory/tokens	1238	233.04	38	15143	100.00	358.20	421.05	1065.75	2.99	19.47
PUT /v1/profile	1233	249.62	20	14900	74.00	332.00	413.60	4343.24	3.97	3.93
PUT /v2/keys	1287	270.94	48	14315	110.00	349.00	457.60	1275.20	2.96	26.71

Table 4: Steady load on Signal 6.13

As the table shows, the response times are much lower than the 4.97 version of the server on average and on median times. This is due to the experiment duration, but also from the more efficient architecture, which uses AWS components. Apart from this, the breakpoint is reached in a shorter time, so the architecture use components which have better performance but are more influenced by load.

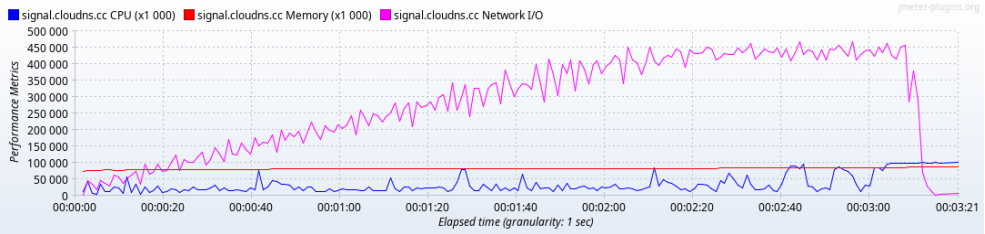

Figure 9: Steady load accounts' creation on Signal 6.13

The diagram shows that the breakpoint is reached in about 3 minutes from the start of the experiment, much less than the previous version.

The next section shows the peak load experiment.

Peak load on Signal 6.13

The results should show the behavior of the new Signal server to high load. I expect that the behavior is better compared to the previous version, as it uses DynamoDB and more AWS services.

Here there are the data related to the peak load experiment for the account creation.

Request	#Samples	Response times (ms)							Network (KB/s)	
		Average	Min	Max	Median	90th pct	95th pct	99th pct	Received	Sent
GET /v1/accounts/fcm/preauth	200	101.37	71	365	94.00	128.90	157.95	252.39	1.91	1.80
GET /v1/accounts/sms/code	200	29.96	22	90	26.00	39.00	57.90	77.99	3.89	1.23
GET /v1/certificate/delivery	200	26.18	17	114	22.00	38.90	57.75	109.88	3.59	1.18
PUT /v1/accounts/code	200	23.81	19	65	22.00	28.90	40.95	54.98	2.18	2.29
PUT /v1/accounts/gcm	200	23.24	17	81	22.00	27.90	39.90	50.00	1.79	2.00
PUT /v1/directory/tokens	200	44.37	39	88	42.00	48.00	62.85	80.95	1.90	12.62
PUT /v1/profile	200	34.56	20	169	31.00	44.90	58.00	130.52	2.49	2.57
PUT /v2/keys	200	60.59	48	110	58.00	72.00	75.95	105.95	1.79	16.94

Table 5: Peak load accounts' creation on Signal 6.13

The response times on the table are comparable between the two server versions, so they are both able to handle the peak loads.

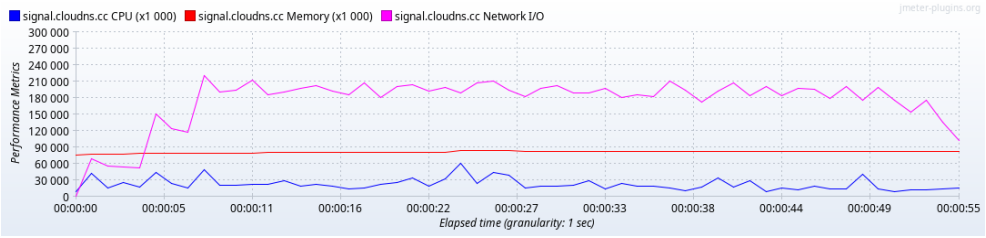

Figure 10: Peak load accounts' creation on Signal 6.13

As on the 4.97 version, with the 6.13 version there are not overloads. We can see that the percentage of CPU is lower on the 6.13 version of the server, which uses more AWS services instead of other services which require an internal management.

3.4.2 New message sending

This experiment uses the following REST APIs:

Type of request	Request path	Description	Involved components
GET	/v1/profile	Request data of the receiver profile	Dropwizard, Redis, PostgreSQL/DynamoDB
GET	/v2/keys/+390123456789/*	Get the keys for message encryption	Dropwizard, Redis, PostgreSQL/DynamoDB
PUT	/v1/messages	Send the message to the receiver	Dropwizard, Redis, PostgreSQL/DynamoDB

Table 6: Sequence of calls to send a message

Between the request of the keys to send the message to the receiver and the request to send the message there is a think time of 3 seconds, to simulate a low think time of the user when he writes the message.

Steady load on Signal 4.97

Results of this experiment should show that sending a message requires less resources than creating an account (for number of requests and involved components of the architecture). Moreover, the results should be worse compared to the 6.13 version of the server.

Here I report the table with the results from the experiment on Signal 4.97.

Request	#Samples	Response times (ms)							Network (KB/s)	
		Average	Min	Max	Median	90th pct	95th pct	99th pct	Received	Sent
GET /v1/profile	48665	2083.32	23	130681	3032.00	3785.00	9200.95	35696.04	63.51	36.36
GET /v2/keys/+391234567890/*	48442	1920.06	17	129352	2989.00	3691.00	6656.70	10982.99	37.58	9.93
PUT /v1/messages	48300	2227.99	20	130654	3063.50	6985.40	9446.95	38666.65	19.67	31.07

Table 7: Steady load message sending on Signal 4.97

The results are not as expected, because the response times are higher compared to the accounts' creations. This is because the requests are heavier.

Here I show the diagram with the resource use.

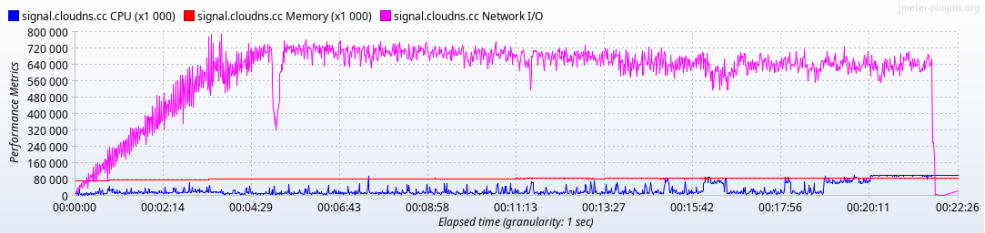

Figure 11: Steady load message sending on Signal 4.97

The diagram shows that the generated traffic is much higher compared to the previous experiment. Apart from this, the 4.97 version of the server needs at least 22 minutes to reach an outage due to overload.

Peak load on Signal 4.97

As before, I expect that in this experiment the peak should be lower than the account creation for the same server version, while the response time should be higher than the 6.13 version.

Here there are the data relative to the experiment with peak load for sending messages.

Request	#Samples	Response times (ms)							Network (KB/s)	
		Average	Min	Max	Median	90th pct	95th pct	99th pct	Received	Sent
GET /v1/profile	200	96.74	78	508	89.00	109.90	123.90	397.77	6.07	4.05
GET /v2/keys/+391234567890/*	200	25.54	19	91	23.00	29.00	43.85	90.91	4.19	1.12
PUT /v1/messages	200	26.80	22	61	26.00	29.00	34.00	57.00	2.16	3.52

Table 8: Peak load message sending on Signal 4.97

In this case the response times are comparable to the ones for the peak load of the accounts' creations. The server is not overloaded, and each request gets a response.

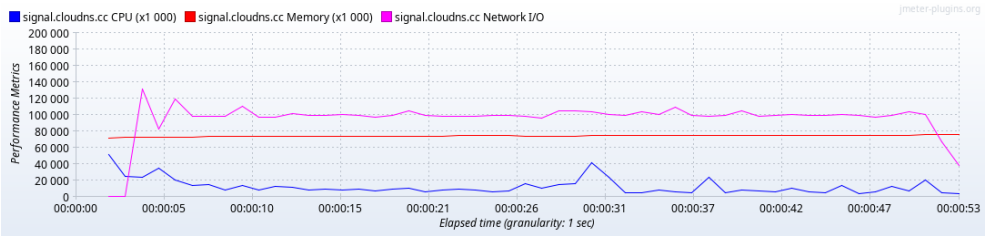

Figure 12: Peak load message sending on Signal 4.97

As the diagram shows, the CPU usage is comparable to the one for the accounts' creations, and the generated network traffic is lower.

Here I compare the previous results with the 6.13 Signal server version.

Steady load on Signal 6.13

Here the results should demonstrate that the 6.13 version of the server is able to handle a steady load for a higher duration compared to the 4.97 version, because the new server version involves more architectural components which use less resources on the server itself. The results should be worse compared to the ones related to the accounts' creation on the same version of the server.

The following table has got the measures for the message sending on Signal 6.13.

Request	#Samples	Response times (ms)							Network (KB/s)	
		Average	Min	Max	Median	90th pct	95th pct	99th pct	Received	Sent
GET /v1/profile	352	57919.86	42	253103	609.00	217865.70	248695.65	252657.40	2.48	0.68
GET /v2/keys/+391234567890/*	209	11001.51	25	251773	52.00	462.00	30623.00	250584.10	0.72	0.18
PUT /v1/messages	199	1730.33	31	246506	55.00	2537.00	3366.00	6867.00	0.36	0.57

Table 9: Steady load message sending on Signal 6.13

The response times in this experiment are very high because it has been manually interrupted after much time from the overload of the server, so the last requests have a very high response time. We can compare the median time for the requests with the 4.97 version of the server, to see that the server itself is more efficient compared to the old version, but it gets overloaded in much less time.

The following diagram shows the resource usage.

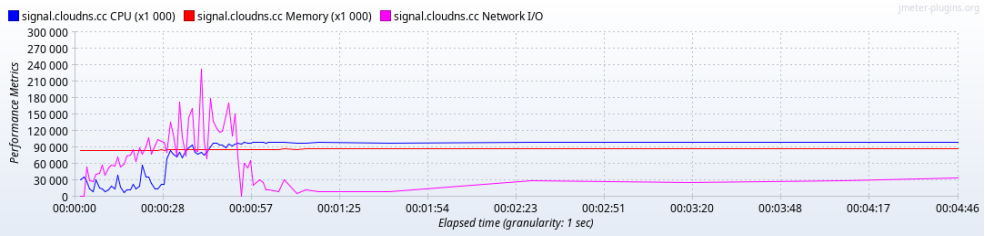

Figure 13: Steady load message sending on Signal 6.13

The diagram clearly shows that the overload of the server happened at 30 seconds from the start of the experiment. Much less network traffic has been generated.

On the next section there are the peak load results.

Peak load on Signal 6.13

Here I expect that the results show that the new version of the server requires less resources to manage a peak of traffic. At the same time the response times should be lower compared to the ones for the peak load related to the accounts' creation.

Here the data relative to the peak load.

Request	#Samples	Response times (ms)							Network (KB/s)	
		Average	Min	Max	Median	90th pct	95th pct	99th pct	Received	Sent
GET /v1/profile	200	86.27	73	199	83.00	98.00	111.95	174.78	6.37	4.04
GET /v2/keys/+391234567890/*	200	22.94	18	53	22.00	25.00	29.00	46.97	3.96	1.11
PUT /v1/messages	200	25.68	21	56	24.00	30.00	42.00	54.97	2.14	3.48

Table 10: Peak load message sending on Signal 6.13

We can see that the obtained results are comparable for average and median times with the 4.97 version of the server. No overload takes place.

The next diagram shows the used resources.

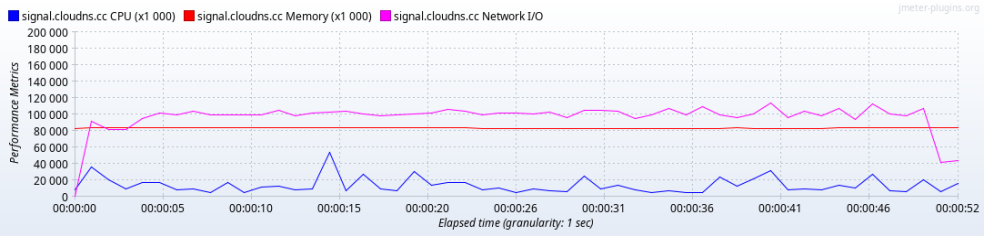

Figure 14: Peak load message sending on Signal 6.13

In this case both the CPU usage and the network traffic are quite similar, there are no significant difference between the two server behaviors.

4 CONCLUSIONS AND OUTLOOK

In this chapter there are some observations about the results of the performed experiments and possible outlooks starting from this work.

4.1 OBSERVATIONS

From the performed experiments there are no evidence about overload vulnerabilities due to the implementation of the APIs. What can be deducted is that while the first architecture model, which uses PostgreSQL and less AWS services, reacts better to steady loads for long periods of time, also if it is less efficient compared to the newer version.

This is confirmed from the steady load experiments, where the 4.97 version of the server is more resilient of the newer version.

However, the two servers have more or less the same behavior with the peak load experiments, and the 6.13 version is more efficient if we take into consideration the CPU usage.

It is reasonable saying that the outage happened on January 2021 as described on the Signal community blog [17] happened because of an accumulation of sessions made from the Android clients in order to retry the failed message sending, without success, and making the situation worse.

The given solution was on the client side, to automatically reset the connection, so they released the resources from the server, which were necessary to manage the new incoming connections.

The load tests are used to simulate situations which are close to the reality, in order to understand how it behaves under an experimental load.

The problem is to understand how to implement a set of significant load tests, which should cover some real situations that the application can face.

To do this we designed the tests to simulate real situations but with scaled down numbers, reduced, because the used resources are a low number. This choice has been done to easily put under load the application.

The different kind of performed experiments can be compared to a normal use of Signal, so a low load during a long period of time, and huge load in a short

period of time, which can be compared to a real situation such as the New Year's Eve wishes.

As the results show both of the architectures are able to face the different loads, with some differences among the experiments.

Resources that are needed to afford huge traffic cost money. This is the reason why elastic scalability systems are needed to acquire and release resources on the right time, to reduce the waste of money in the best possible manner.

The experiments do not investigate on it. To do this, it is necessary to use a IAAS such as AWS is configured in a way similar to the production environment. This is not under public knowledge.

What the experiments performed in the thesis confirmed is that the architectures used for the Signal server behave better in specific conditions, the 4.97 version is more suited for steady loads (see figures 7 on page 18 and 11 on page 22 compared to figures 9 on page 20 and 13 on page 24), while the 6.13 version is generally more efficient (see figures 10 on page 21 and 14 on page 24 compared to figures 8 on page 19 and 12 on page 23). Moreover, the new version delegates the management of components to AWS, which is good to waste less time into them.

What we highlight is that the evolution of the architecture with a change on its components is good to avoid huge loads in short time periods, but it can be less suited for long term loads. Moreover, changes on the new server version are useful to delegate the management of the database system and other services used by the server, so their performance must respect the ranges offered by AWS.

4.2 OUTLOOK

This thesis has been made to provide some resources useful to load different kind of architectures of cloud applications.

One possible example can be WhatsApp. It is possible to use the same experiments, with some changes on the HTTP calls, in order to get some results about it. It would be possible to do a comparison between it and Signal, in order to see the more resilient to steady load.

This would be possible only with access to the WhatsApp architecture, which is not open source.

Other outlooks with the focus of extending this work are the following.

It is possible to cover also the entire architecture of Signal to test if the AWS provided services respect the agreed level of resources.

Two examples are:

- the voice/video call related components;
- the attachment related components.

Another outlook is related to the kind of experiments which can be performed.

For example, it is possible to load test the architecture in a white box way, so repeat the previous tests while Redis is running other operations, or while PostgreSQL is serving another application.

It would be possible to change the characteristics of the environment where the Signal server is hosted, to check how the results change.

BIBLIOGRAPHY

[1] Matthias Mehner. *WhatsApp, WeChat and Facebook Messenger apps - global usage of messaging apps, penetration and statistics*. Nov. 2021. URL: https://www.messengerpeople.com/global-messenger-usage-statistics/ (cit. on p. iii).

[2] *Signal*. URL: https://github.com/signalapp (cit. on p. iii).

[3] Tullio Vardanega. *Runtimes for concurrency and distribution*. 2021 (cit. on p. 1).

[4] Chris Richardson of Eventuate. *Introduction to microservices*. Aug. 2021. URL: https://www.nginx.com/blog/introduction-to-microservices/ (cit. on p. 1).

[5] Alberto Avritzer et al. "Scalability assessment of microservice architecture deployment configurations: A domain-based approach leveraging operational profiles and load tests". In: *Journal of Systems and Software* 165 (2020), p. 110564 (cit. on p. 1).

[6] Catia Trubiani et al. "Exploiting load testing and profiling for performance antipattern detection". In: *Information and Software Technology* 95 (2018), pp. 329–345 (cit. on p. 2).

[7] Christian Davatz et al. "An approach and case study of cloud instance type selection for multi-tier web applications". In: *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE. 2017, pp. 534–543 (cit. on p. 3).

[8] Davy Preuveneers et al. "Systematic scalability assessment for feature oriented multi-tenant services". In: *Journal of Systems and Software* 116 (2016), pp. 162–176 (cit. on p. 3).

[9] Victor Heorhiadi et al. "Gremlin: Systematic resilience testing of microservices". In: *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2016, pp. 57–66 (cit. on p. 3).

[10] Haryadi S Gunawi et al. "Why does the cloud stop computing? lessons from hundreds of service outages". In: *Proceedings of the Seventh ACM Symposium on Cloud Computing*. 2016, pp. 1–16 (cit. on p. 3).

[11] Michał Matłoka. *What I've learned from Signal Server source code*. Apr. 2021. URL: https://softwaremill.com/what-ive-learned-from-signal-server-source-code/ (cit. on p. 8).

[12] Sorin Cocorada. *Signal Messanger Architecture*. June 2018. URL: https://sorincocorada.ro/signal-messanger-architecture/ (cit. on pp. 5, 8).

[13] Nick Woodard. *Secure Messenger Service Signal is down*. Jan. 2021. URL: https://screenrant.com/signal-down-secure-messenger-service-outage-explained/ (cit. on p. 4).

[14] Jason Aten. *How signal became the most popular app in the world overnight, and why it matters*. Jan. 2021. URL: https://www.inc.com/jason-aten/how-signal-became-most-popular-app-in-world-overnight-why-it-matters.html (cit. on p. 4).

[15] *Technology preview: Private contact discovery for Signal*. URL: https://signal.org/blog/private-contact-discovery/ (cit. on p. 12).

[16] *Apache JMeter™*. URL: http://jmeter.apache.org/ (cit. on p. 13).

[17] Svazzoler et al. *Signal Service outage on 2021-01-15*. Jan. 2021. URL: https://community.signalusers.org/t/signal-service-outage-on-2021-01-15/22941/334 (cit. on p. 25).