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

Nowadays, a lot of applications that we use in our daily life are distributed and subject to load, which every day increases. This kind of applications bring to an expansion of the number of architectures, some microservices oriented, some with other paradigms, in order to deal with the scalability problem [1].

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 [2].

An interesting example of chat application is Signal [3], 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 take for granted.

What we want to do in this thesis is to confirm that the Signal architecture is scalable by using experimental tests. Specifically we take into analysis the Signal architecture, by verifying if the calls to its APIs support this kind of scalability. We can do this because Signal has got an open source architecture, which make easier to design and implement specific load tests for it.

RINGRAZIAMENTI

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

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 [4].

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 ho the application faces it.

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

The scalability dimensions [5] can be divided into:

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

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

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

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 architecture [6, 7] faces the elastic scalability problem. I start from the case study of an outage happened between the 15th and the 17th January 2021 as a reference [8, 9], 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 huge loads in short time periods, but it can be less suited for long term loads. 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[1].

The remaining part of the thesis is organized as follows: the second chapter (see chapter 2 on the facing page) treats about the solution space, where there are points covered from the thesis and uncovered points. The third chapter (see chapter 3 on page 5) 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.

[1] 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

As stated in the previous chapter, in order to test the Signal architecture we use some experiments to create load on it.

Usually this kind of test is 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[1] and from the App Store[2], 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.

[1] https://play.google.com/store

[2] https://www.apple.com/app-store

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 [6, 7], such as the Dropwizard[3] application itself, Redis[4], PostgreSQL[5] and DynamoDB[6] (only on the newer version of the Signal server).

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.

On the experiments how the Signal architecture reacts to a low load constant for a long period of time, which can be compared to a normal use of the Signal application, and to a high load for a short period, which can be compared to an event like the New Year's Eve, when a lot of messages are sent to people who are not near. All the experiments are scaled down to the resources that I could afford.

With base load we mean a load with a long duration in time, but with one user at a time, which simulate the use of the application without relevant events.

With high 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.

2.2 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).

[3] https://www.dropwizard.io/en/latest
[4] https://redis.io
[5] https://www.postgresql.org
[6] https://aws.amazon.com/dynamodb

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 ARCHITECTURE

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

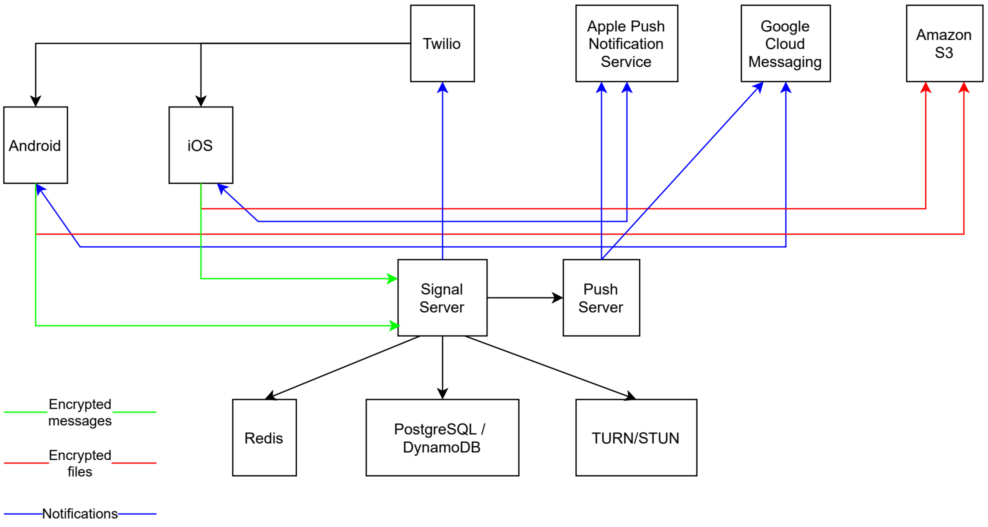


Figure 1: Signal Server general architecture

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.

3.3 EXPERIMENTS

The experiments I made can be divided into:

- base load for a long period of time;
- higher load with ramp-up period.

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 an in order to send a message to 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 version 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.

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

3.4 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.

3.4.1 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 [10].

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[2] and DynamoDB.

I deployed the server to a t3.micro EC2 instance[3], 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[4]
- 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[5]

In order to make it reachable from the outside I used ClouDNS[6], which provides a free dynamic DNS for subdomains. The name I chose is signal.cloudns.cc

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

3.4.2 Client characteristics

In order to perform the test it has been used JMeter [11], 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

3.5 DETAILS ABOUT IMPLEMENTATION

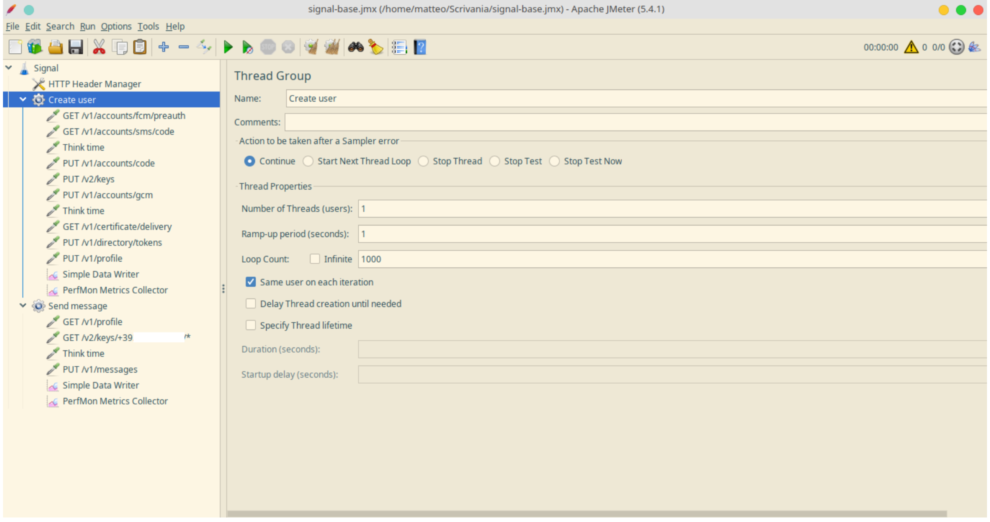
To implement the load experiments I used JMeter, a Java tool made to generate a huge number of requests with lots of parameters.

3.5.1 Base load settings

With base load we mean a low load with a long duration, used to simulate the use of Signal without particular events.

The base load is very low also because of the server capacity. The test is made from one user who executes a sequence of operations for account creations or to send messages. It is executed one sequence at a time, for 1000 consecutive times.

Here I report the settings used to generate a base load for a long period of time on the server using JMeter.



signal-base.jmx (/home/matteo/Scrivania/signal-base.jmx) - Apache JMeter (5.4.1)

- Signal
  - HTTP Header Manager
  - Create user
    - GET /v1/accounts/fcm/preauth
    - GET /v1/accounts/sms/code
    - Think time
    - PUT /v1/accounts/code
    - PUT /v2/keys
    - PUT /v1/accounts/gcm
    - Think time
    - GET /v1/certificate/delivery
    - PUT /v1/directory/tokens
    - PUT /v1/profile
    - Simple Data Writer
    - PerfMon Metrics Collector
  - Send message
    - GET /v1/profile
    - GET /v2/keys/+39[redacted]/*
    - Think time
    - PUT /v1/messages
    - Simple Data Writer
    - PerfMon Metrics Collector

Thread Group

Name: Create user

Comments:

Action to be taken after a Sampler error

● Continue ○ Start Next Thread Loop ○ Stop Thread ○ Stop Test ○ Stop Test Now

Thread Properties

Number of Threads (users): 1

Ramp-up period (seconds): 1

Loop Count: □ Infinite 1000

☑ Same user on each iteration

□ Delay Thread creation until needed

□ Specify Thread lifetime

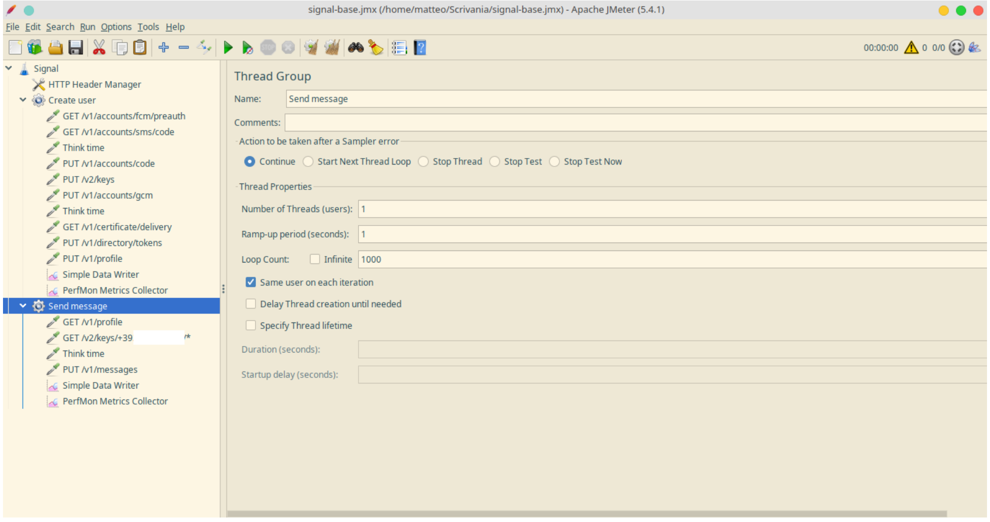
Duration (seconds):

Startup delay (seconds):

Figure 2: JMeter base load for accounts creation

As the figure shows, the test performs 1000 user creation, with 1 thread at a time (so the ramp-up period is not used). So in total JMeter performs 8000 calls to the server.

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



signal-base.jmx (/home/matteo/Scrivania/signal-base.jmx) - Apache JMeter (5.4.1)

- Signal
  - HTTP Header Manager
  - Create user
    - GET /v1/accounts/fcm/preauth
    - GET /v1/accounts/sms/code
    - Think time
    - PUT /v1/accounts/code
    - PUT /v2/keys
    - PUT /v1/accounts/gcm
    - Think time
    - GET /v1/certificate/delivery
    - PUT /v1/directory/tokens
    - PUT /v1/profile
    - Simple Data Writer
    - PerfMon Metrics Collector
  - Send message
    - GET /v1/profile
    - GET /v2/keys/+39[redacted]/*
    - Think time
    - PUT /v1/messages
    - Simple Data Writer
    - PerfMon Metrics Collector

Thread Group

Name: Send message

Comments:

Action to be taken after a Sampler error

● Continue ○ Start Next Thread Loop ○ Stop Thread ○ Stop Test ○ Stop Test Now

Thread Properties

Number of Threads (users): 1

Ramp-up period (seconds): 1

Loop Count: □ Infinite 1000

☑ Same user on each iteration

□ Delay Thread creation until needed

□ Specify Thread lifetime

Duration (seconds):

Startup delay (seconds):

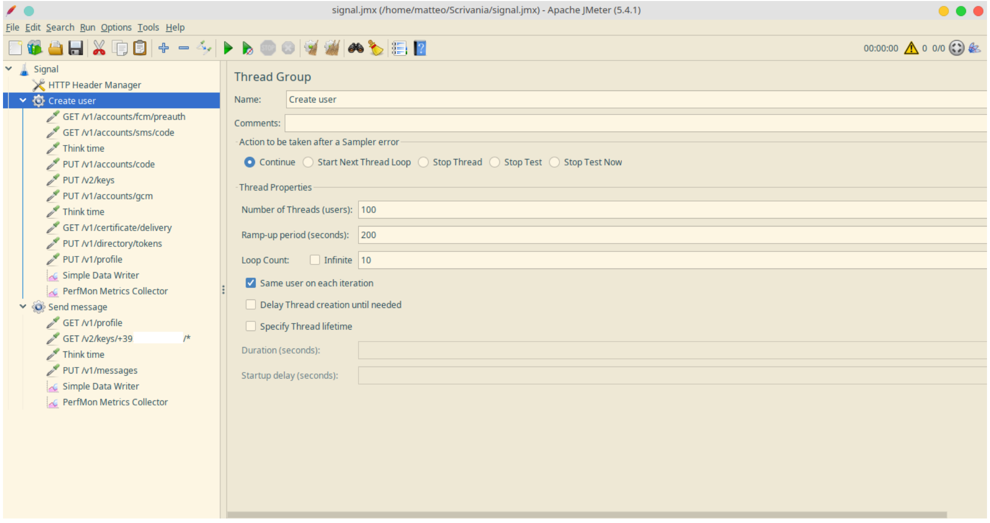
Figure 3: JMeter base load for message sending

3.5.2 High load settings

With high load we mean a load higher than the base one, with a low duration, used to simulate a higher use of the Signal applications. 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 100 parallel users, who execute sequences of requests to the Signal APIs to create accounts or to send messages. Each user is added after 2 seconds (ramp-up of 200 seconds) and the experiment is repeated 10 times.

In order to perform a higher load test, for a shorter period of time, I used the following settings.



signal.jmx (/home/matteo/Scrivania/signal.jmx) - Apache JMeter (5.4.1)

- Signal
  - HTTP Header Manager
  - Create user
    - GET /v1/accounts/fcm/preauth
    - GET /v1/accounts/sms/code
    - Think time
    - PUT /v1/accounts/code
    - PUT /v2/keys
    - PUT /v1/accounts/gcm
    - Think time
    - GET /v1/certificate/delivery
    - PUT /v1/directory/tokens
    - PUT /v1/profile
    - Simple Data Writer
    - PerfMon Metrics Collector
  - Send message
    - GET /v1/profile
    - GET /v2/keys/+39[redacted]/*
    - Think time
    - PUT /v1/messages
    - Simple Data Writer
    - PerfMon Metrics Collector

Thread Group

Name: Create user

Comments:

Action to be taken after a Sampler error

● Continue ○ Start Next Thread Loop ○ Stop Thread ○ Stop Test ○ Stop Test Now

Thread Properties

Number of Threads (users): 100

Ramp-up period (seconds): 200

Loop Count: □ Infinite 10

☑ Same user on each iteration

□ Delay Thread creation until needed

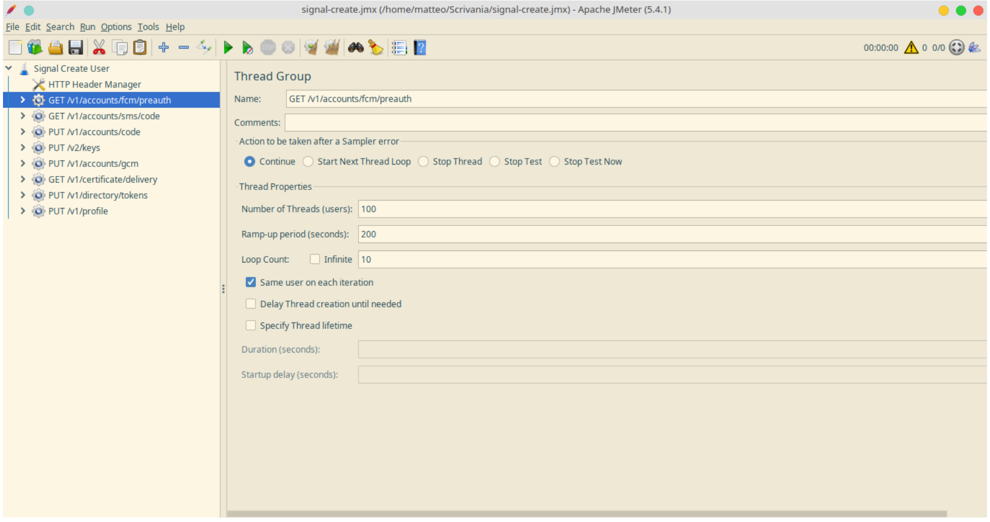
□ Specify Thread lifetime

Duration (seconds):

Startup delay (seconds):

Figure 4: JMeter high load test

This time the maximum concurrent threads is set to 100, with a ramp-up period of 200 seconds (a thread is added every 2 seconds), with 10 cycles. This means that the users created are still 1000.



signal-create.jmx (/home/matteo/Scrivania/signal-create.jmx) - Apache JMeter (5.4.1)

- Signal Create User
  - HTTP Header Manager
  - GET /v1/accounts/fcm/preauth
  - GET /v1/accounts/sms/code
  - PUT /v1/accounts/code
  - PUT /v2/keys
  - PUT /v1/accounts/gcm
  - GET /v1/certificate/delivery
  - PUT /v1/directory/tokens
  - PUT /v1/profile

Thread Group

Name: GET /v1/accounts/fcm/preauth

Comments:

Action to be taken after a Sampler error

● Continue ○ Start Next Thread Loop ○ Stop Thread ○ Stop Test ○ Stop Test Now

Thread Properties

Number of Threads (users): 100

Ramp-up period (seconds): 200

Loop Count: □ Infinite 10

☑ Same user on each iteration

□ Delay Thread creation until needed

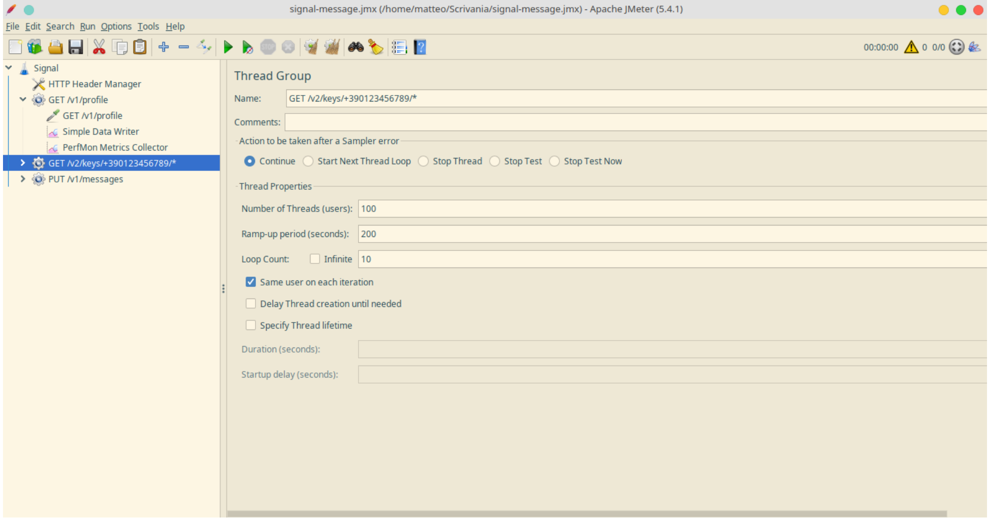
□ Specify Thread lifetime

Duration (seconds):

Startup delay (seconds):

Figure 5: JMeter high load for accounts creation

The same thing has been done for the message sending, and also for the single requests divided by type.


signal-message.jmx (/home/matteo/Scrivania/signal-message.jmx) - Apache JMeter (5.4.1)
File Edit Search Run Options Tools Help
00:00:00 0 0/0
Signal
HTTP Header Manager
GET /v1/profile
GET /v1/profile
Simple Data Writer
PerfMon Metrics Collector
GET /v2/keys/+390123456789/*
PUT /v1/messages

Thread Group

Name: GET /v2/keys/+390123456789/*

Comments:

Action to be taken after a Sampler error

● Continue ○ Start Next Thread Loop ○ Stop Thread ○ Stop Test ○ Stop Test Now

Thread Properties

Number of Threads (users): 100

Ramp-up period (seconds): 200

Loop Count: □ Infinite 10

☑ Same user on each iteration

□ Delay Thread creation until needed

□ Specify Thread lifetime

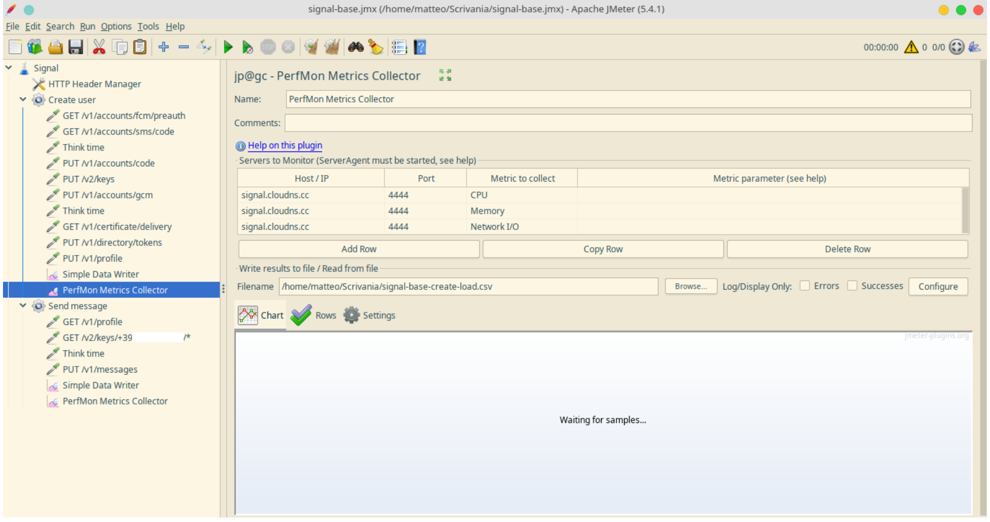
Duration (seconds):

Startup delay (seconds):

Figure 6: JMeter high load for message sending

3.5.3 Perfmon

Perfmon[7] 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.



jp@gc - PerfMon Metrics Collector

Name: PerfMon Metrics Collector

Comments:

Help on this plugin

Servers to Monitor (ServerAgent must be started, see help)

Host / IP	Port	Metric to collect	Metric parameter (see help)
signal.cloudns.cc	4444	CPU	
signal.cloudns.cc	4444	Memory	
signal.cloudns.cc	4444	Network I/O	

Write results to file / Read from file

Filename /home/matteo/Scrivania/signal-base-create-load.csv

Log/Display Only: □ Errors □ Successes

Waiting for samples...

Figure 7: JMeter Perfmon plugin

[7] https://jmeter-plugins.org/wiki/PerfMon

3.6 RESULTS

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

3.6.1 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.

Base load for a long period of time on Signal 4.97

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	1000	37.91	20	744	28.00	36.00	52.95	288.98	0.07	0.07
GET /v1/accounts/sms/code	1000	35.46	19	552	26.00	35.90	51.00	290.92	0.15	0.05
GET /v1/certificate/delivery	1000	44.72	19	2261	27.00	59.90	107.00	312.96	0.14	0.05
PUT /v1/accounts/code	1000	44.64	20	1113	27.00	54.00	123.40	342.90	0.08	0.09
PUT /v1/accounts/gcm	1000	37.39	18	590	27.00	35.00	55.85	296.81	0.07	0.08
PUT /v1/directory/tokens	1000	57.44	41	1236	47.00	54.00	67.00	322.90	0.07	0.49
PUT /v1/profile	1000	43.46	24	551	34.00	48.90	61.95	277.99	0.07	0.10
PUT /v2/keys	1000	70.72	50	660	59.00	70.00	90.85	351.93	0.07	0.66

Table 2: Base load accounts' creation on Signal 4.97

From these data we can see that the calls used to save the keys request an 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.

If we order the requests by the average response time we see that the weak point is always the dimension of the request instead of the component used in the architecture.

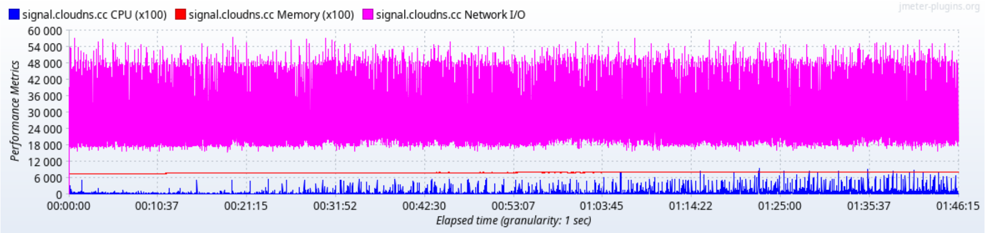


Figure 8: Base load accounts' creation on Signal 4.97

From the diagram it is visible that the CPU use increase over time, also if their number is low and there are pauses between the requests. The diagram shows also that the CPU use is not constant and that it goes from 0% to 100% and vice versa alternatively.

High load with ramp-up on Signal 4.97

Here I show the data obtained from the high load generation with a ramp-up 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	953	177.24	20	57250	37.00	106.00	147.30	455.64	1.46	1.36
GET /v1/accounts/sms/code	951	104.41	18	43321	27.00	66.00	81.00	193.60	3.16	0.92
GET /v1/certificate/delivery	946	87.58	18	5534	26.50	70.30	102.60	2509.52	3.54	1.17
PUT /v1/accounts/code	950	122.05	18	22948	27.00	87.00	119.45	1503.15	1.92	2.02
PUT /v1/accounts/gcm	946	66.39	19	5541	26.00	62.00	89.65	378.28	1.79	2.00
PUT /v1/directory/tokens	946	135.69	41	12532	48.00	94.30	126.95	3836.24	1.77	11.81
PUT /v1/profile	946	1054.17	24	80726	41.00	105.60	178.90	69366.33	1.48	1.93
PUT /v2/keys	950	310.79	51	58985	71.00	109.00	133.00	1727.85	1.38	12.78

Table 3: High load accounts' creation on Signal 4.97

The 99th percentile shows very high response times, because some requests, in order to be included in the curve, raise the maximum level of the percentile. They are next to the break point, which happened near the end of the experiment run.

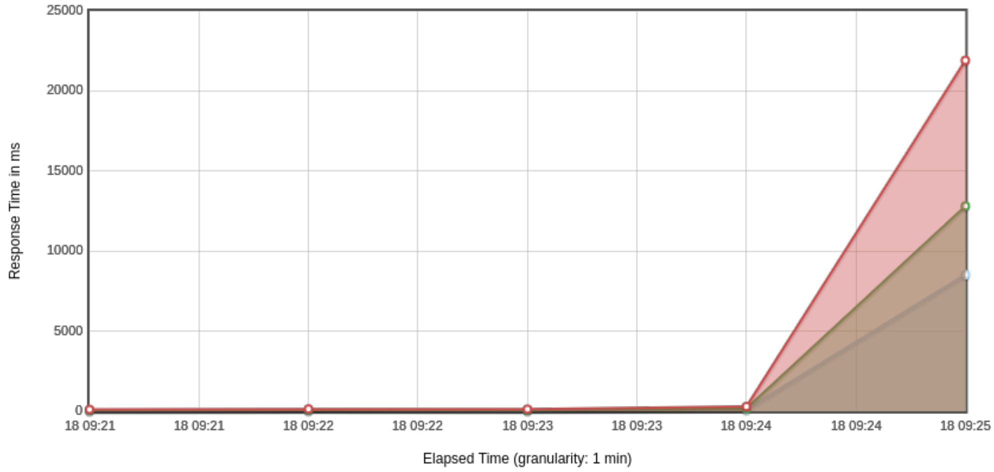


Figure 9: Percentiles of high load accounts' creation on Signal 4.97

Also in this experiment the requests that required more time are the ones with the heavier message body, excluded the ones which are close to the break point.

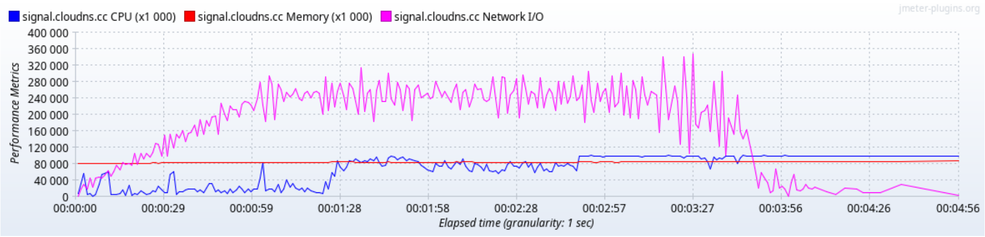


Figure 10: High load accounts' creation on Signal 4.97

The diagram shows that the server started to run at 100% of CPU load after 3 minutes from the start of the experiment run, and it reached the break point at 3 minutes and 40 seconds, when the requests are not taken into account anymore.

To check which requests require more resources I do again the same experiment by request type.

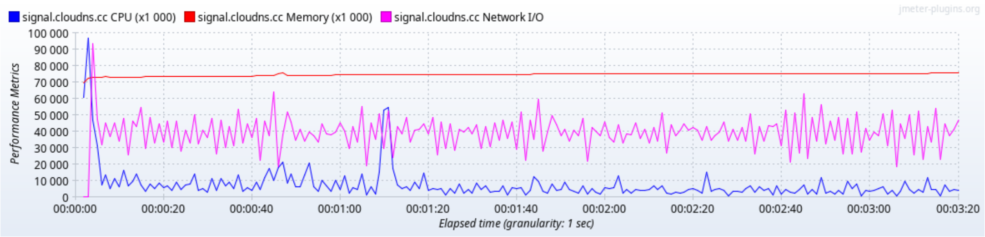


Figure 11: High load GET /v1/accounts/fcm/preauth on Signal 4.97

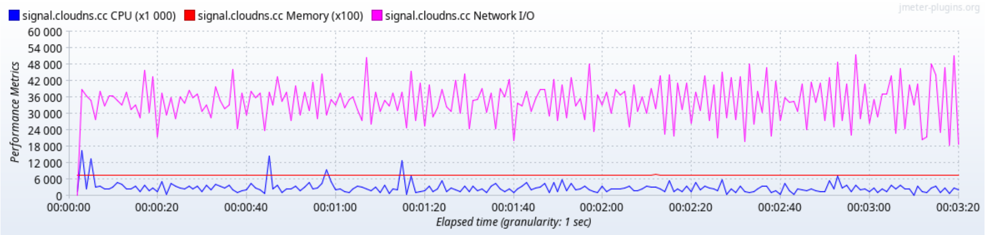


Figure 12: High load GET /v1/accounts/sms/code on Signal 4.97

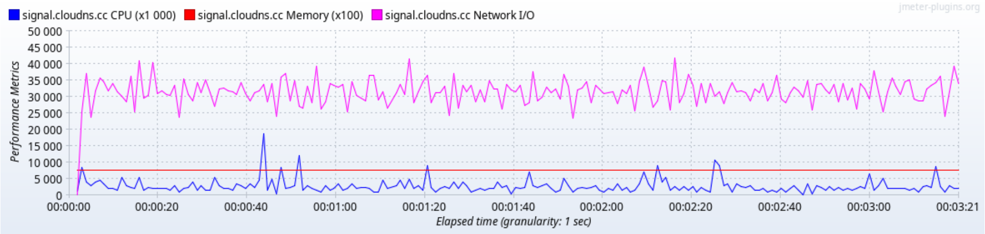


Figure 13: High load PUT /v1/accounts/code on Signal 4.97

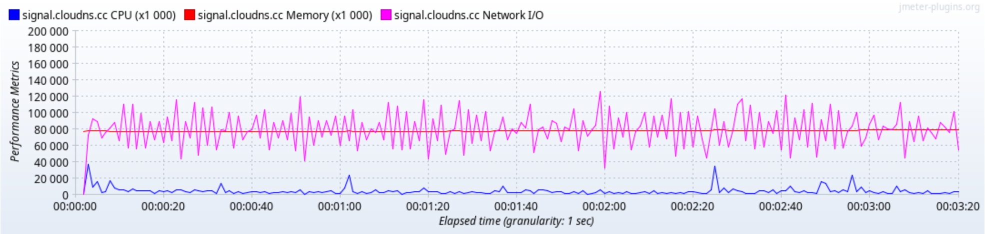


Figure 14: High load PUT /v2/keys on Signal 4.97

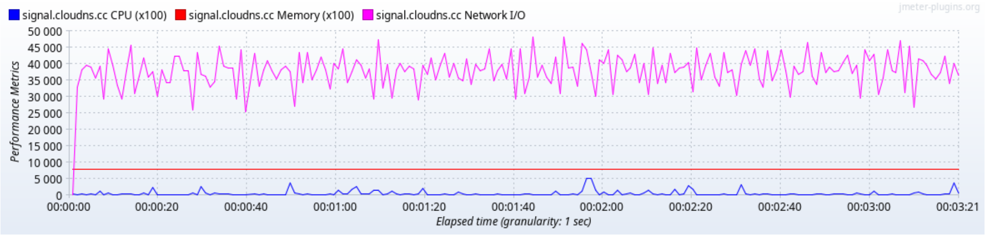


Figure 15: High load PUT /v1/accounts/gcm on Signal 4.97

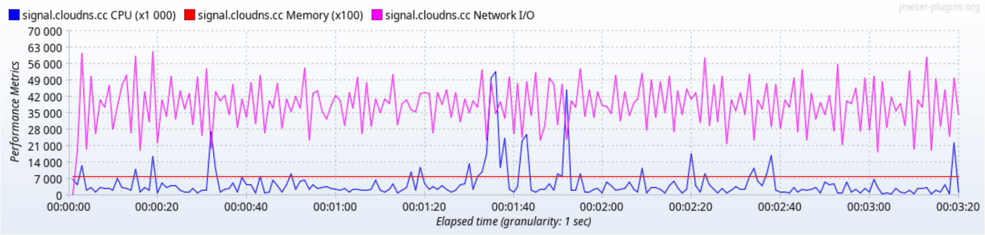


Figure 16: High load GET /v1/certificate/delivery on Signal 4.97

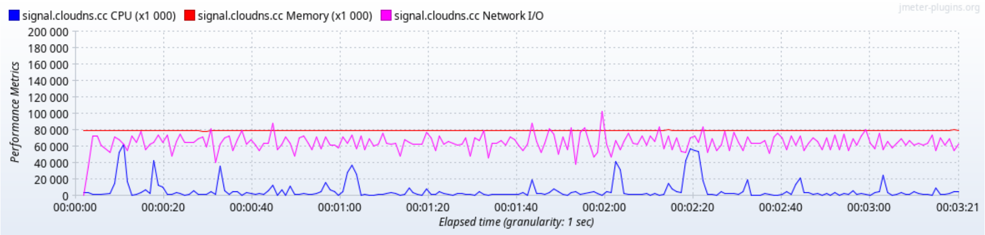


Figure 17: High load PUT /v1/directory/tokens on Signal 4.97

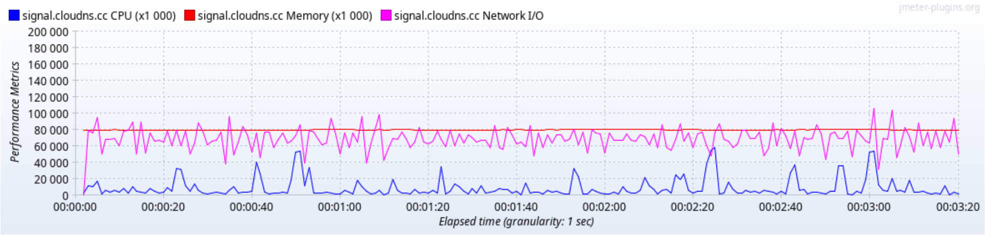


Figure 18: High load PUT /v1/profile on Signal 4.97

The obtained results confirm that all the requests are satisfied and no ramp-up happens.

So the overload of the first experiment is caused by the set of the requests, not from a particular one. None of the architectural components needs an excessive number of resources.

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

Base load for a long period of time on Signal 6.13

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	283	49.58	32	544	42.00	63.60	93.40	179.28	0.06	0.05
GET /v1/accounts/sms/code	283	36.87	20	171	30.00	47.00	89.60	154.16	0.12	0.04
GET /v1/certificate/delivery	282	310.26	21	73408	44.00	91.70	115.00	169.87	0.11	0.04
PUT /v1/accounts/code	282	48.27	21	363	40.00	76.70	100.85	318.17	0.07	0.07
PUT /v1/accounts/gcm	282	32.55	20	851	27.00	33.00	53.40	118.17	0.05	0.06
PUT /v1/directory/tokens	282	1177.65	41	317648	46.00	55.40	67.85	307.51	0.06	0.38
PUT /v1/profile	282	539.26	26	135067	54.50	96.00	119.55	200.68	0.07	0.08
PUT /v2/keys	282	70.45	52	415	65.00	86.00	100.00	164.99	0.05	0.51

Table 4: Base load accounts' creation on Signal 6.13

The requests that show a higher response time are the one used to send the profile data and the one used to send the code, according to the 90th percentile and the 95th percentile.

We can see from the following diagram that the server is overloaded at the 30th minute of execution, and the traffic gets stooped until an interruption.

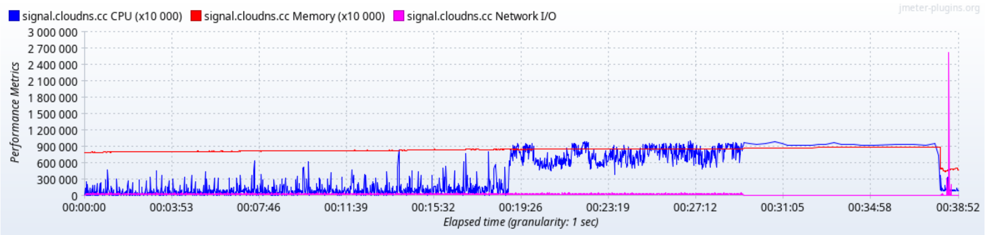


Figure 19: Base load accounts' creation on Signal 6.13

This architecture, compared from the previous one, is more susceptible to a lower load which continue for a long time. The next section shows the high load experiment.

High load with ramp-up on Signal 6.13

Here there are the data related to the high load experiment with ramp-up 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	1000	175.23	25	130222	32.00	85.00	99.95	140.97	1.81	1.70
GET /v1/accounts/sms/code	1000	35.85	23	111	28.00	59.00	70.00	90.99	3.64	1.15
GET /v1/certificate/delivery	1000	30.94	18	182	23.00	54.90	70.00	110.00	3.35	1.10
PUT /v1/accounts/code	1000	32.80	19	180	24.00	56.90	71.95	118.98	2.03	2.14
PUT /v1/accounts/gcm	1000	29.53	20	178	24.00	42.90	55.00	108.97	1.67	1.87
PUT /v1/directory/tokens	1000	51.17	40	141	45.00	72.00	78.00	101.00	1.77	11.76
PUT /v1/profile	1000	36.12	21	120	31.00	57.00	64.00	88.99	2.31	2.40
PUT /v2/keys	1000	64.43	50	147	58.00	82.90	90.00	114.00	1.67	15.83

Table 5: High load accounts' creation on Signal 6.13

The times on the median are comparable to the ones of the experiment executed with the 4.97 version of the server. This time the server is not overloaded, this is visible from the following diagram.

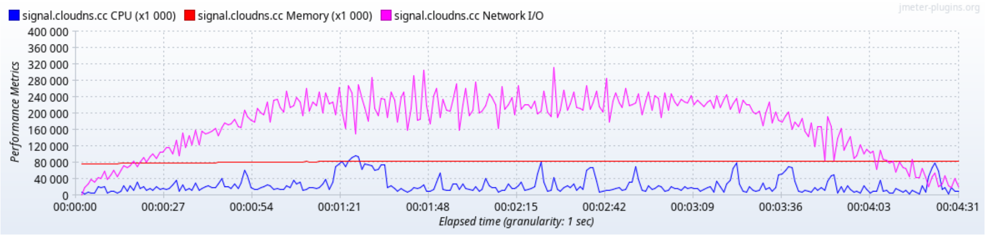


Figure 20: High load accounts' creation on Signal 6.13

Here I report the results for the requests divided by type.

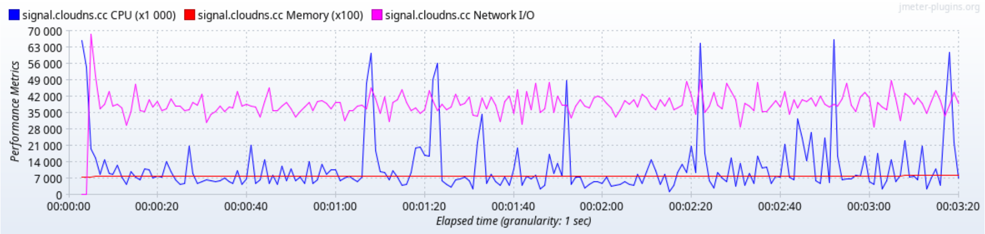


Figure 21: High load GET /v1/accounts/fcm/preauth on Signal 6.13

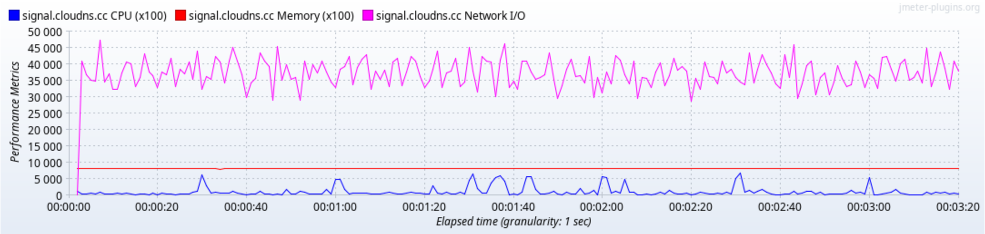


Figure 22: High load GET /v1/accounts/sms/code on Signal 6.13

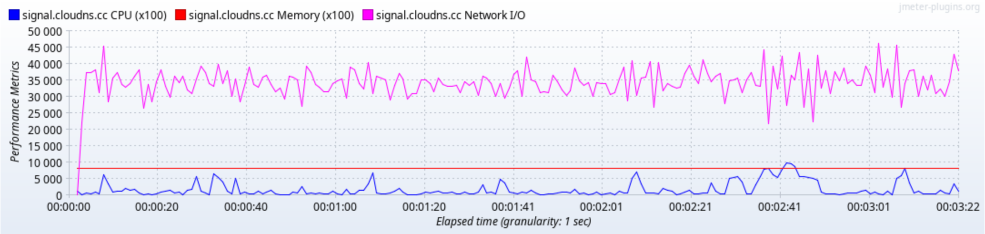


Figure 23: High load PUT /v1/accounts/code on Signal 6.13

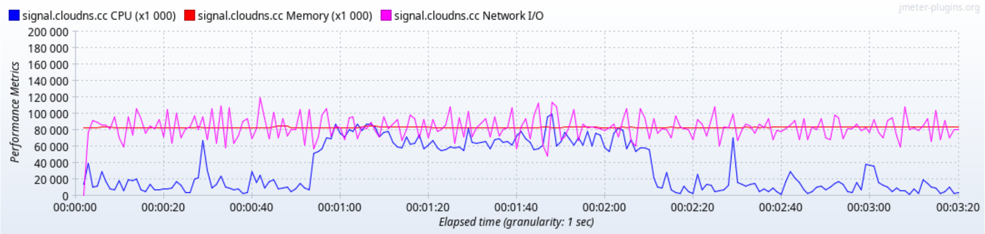


Figure 24: High load PUT /v2/keys on Signal 6.13

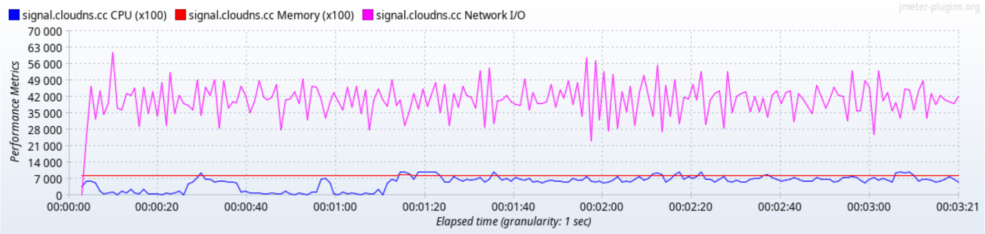


Figure 25: High load PUT /v1/accounts/gcm on Signal 6.13

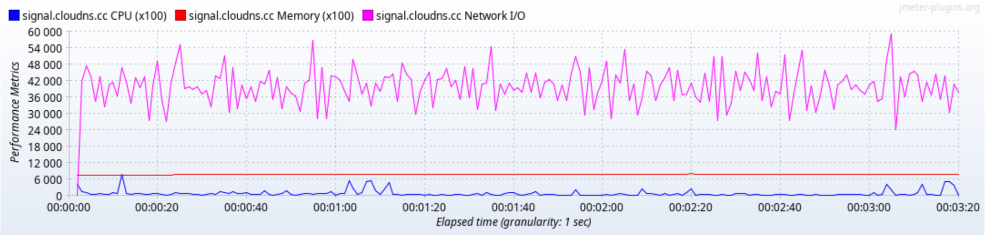


Figure 26: High load GET /v1/certificate/delivery on Signal 6.13

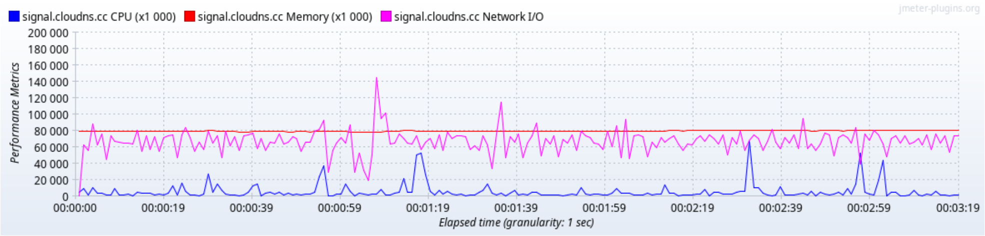


Figure 27: High load PUT /v1/directory/tokens on Signal 6.13

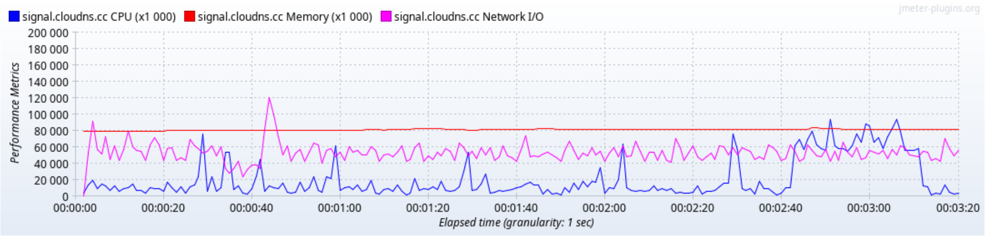


Figure 28: High load PUT /v1/profile on Signal 6.13

As on the 4.97 version, with the 6.13 version there are not overloads.

3.6.2 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.

Base load for a long period of time on Signal 4.97

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	1000	56.68	33	308	53.00	78.00	87.95	112.99	0.48	0.32
GET /v2/keys/+390123456789/*	1000	26.73	19	306	24.00	33.00	42.00	57.00	0.33	0.09
PUT /v1/messages	1000	47.74	23	987	44.00	66.00	87.00	133.00	0.17	0.28

Table 7: Base load message sending on Signal 4.97

The request which requires more time for the response is the one used to obtain the receiver profile data. This request uses more Redis and PostgreSQL if the data are not present on Redis.

Here I show the diagram with the resource use.

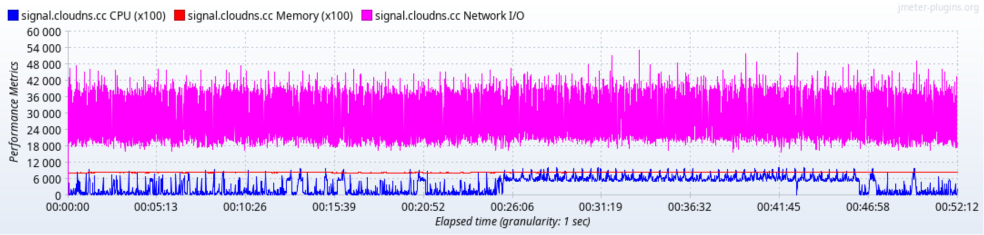


Figure 29: Base load message sending on Signal 4.97

From the diagram it is visible a higher load from the 26th minute to the 46th one, but there is no overload on the server.

High load with ramp-up on Signal 4.97

Here there are the data relative to the experiment with high load and ramp-up 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	1000	174.16	31	130122	37.00	80.00	91.00	114.00	5.68	3.78
GET /v2/keys/+390123456789/*	1000	25.19	19	306	24.00	27.00	33.00	54.99	3.89	1.04
PUT /v1/messages	1000	26.55	22	97	25.00	29.00	35.00	52.98	2.00	3.26

Table 8: High load message sending on Signal 4.97

In this case, as on the previous one, the order of the requests is the same for the response time. The following diagram shows the resources use.

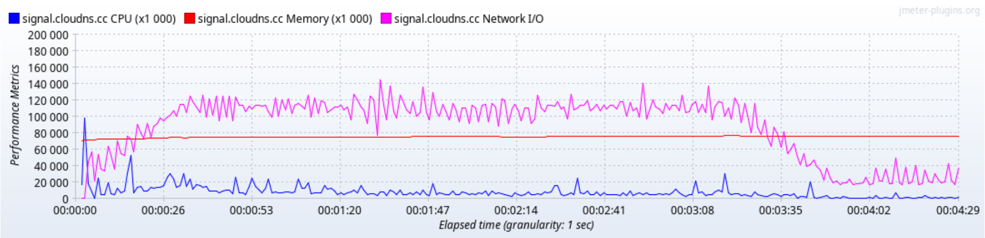


Figure 30: High load message sending on Signal 4.97

The diagram confirms the scale-up, so the requests are queued, but there is no overload on the server for the CPU usage.

To verify which requests require more resources, I did the same experiment as before, looking on the single type requests.

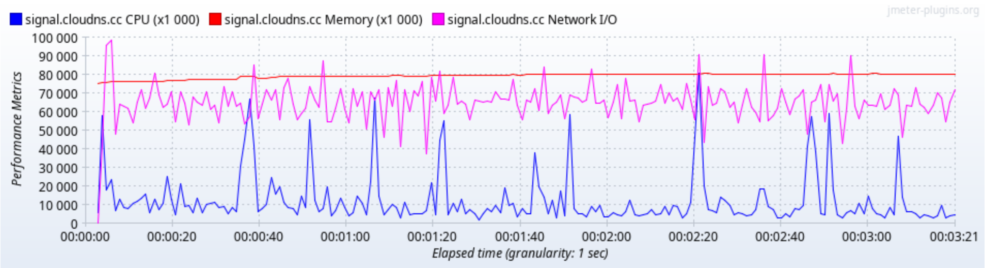


Figure 31: High load GET /v1/profile on Signal 4.97

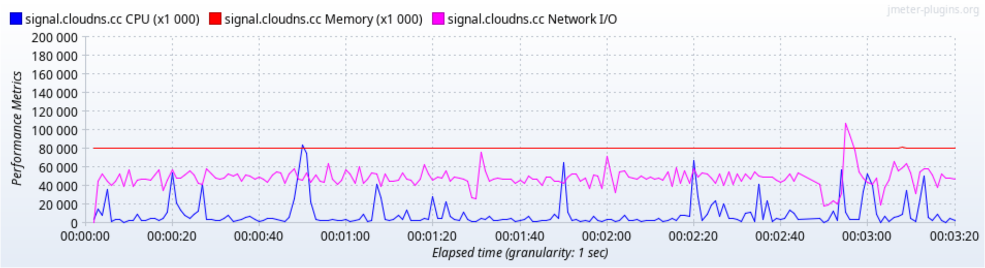


Figure 32: High load GET /v2/keys/+390123456789/* on Signal 4.97

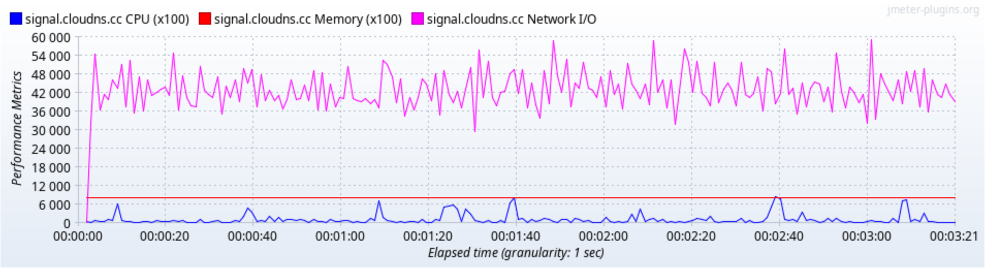


Figure 33: High load PUT /v1/messages on Signal 4.97

In this case all the three kind of request finish in a correct way, without overloads.

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

Base load for a long period of time on Signal 6.13

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	1000	56.68	33	308	53.00	78.00	87.95	112.99	0.48	0.32
GET /v2/keys/+390123456789/*	1000	26.73	19	306	24.00	33.00	42.00	57.00	0.33	0.09
PUT /v1/messages	1000	47.74	23	987	44.00	66.00	87.00	133.00	0.17	0.28

Table 9: Base load message sending on Signal 6.13

As for the 4.97 version of the server, the requests to get the profile data of the receiver and to send the messages are the ones which needs a higher response time, but their distance is higher this time.

The following diagram shows the resource usage.

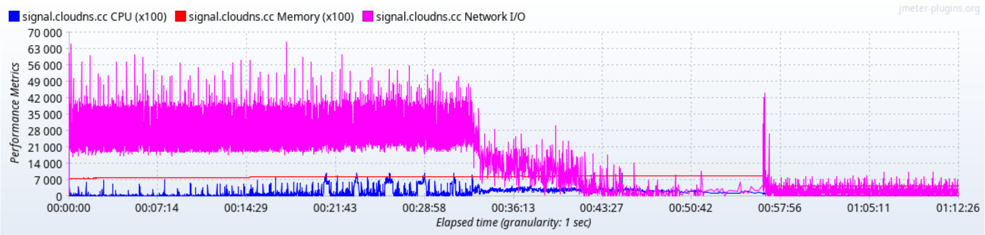


Figure 34: Base load message sending on Signal 6.13

Compared to the 4.97 version of the server, there is a rejection of requests after 30 minutes from the starting point of the experiment, this is the reason of the traffic decrease. As a consequence the CPU use decreases.

On the next section there are the high load results.

High load with ramp-up on Signal 6.13

Here the data relative to the high load with ramp-up period.

Request	#Samples	Response times (ms)							Network (KB/s)	
		Average	Min	Max	Median	90th pct	95th pct	99th pct	Received	Sent
GET /v1/profile	1000	54.95	34	328	43.00	98.00	112.00	152.98	7.02	4.45
GET /v2/keys/+390123456789/*	1000	27.91	20	495	24.00	37.00	46.95	64.97	4.35	1.22
PUT /v1/messages	1000	28.89	22	123	26.00	37.00	43.00	59.99	2.35	3.84

Table 10: High load message sending on Signal 6.13

Here the median response times are comparable to the 4.97 version. The load is higher compared to the other version, but there is no overload. It is clearly visible the ramp-up on the following diagram.

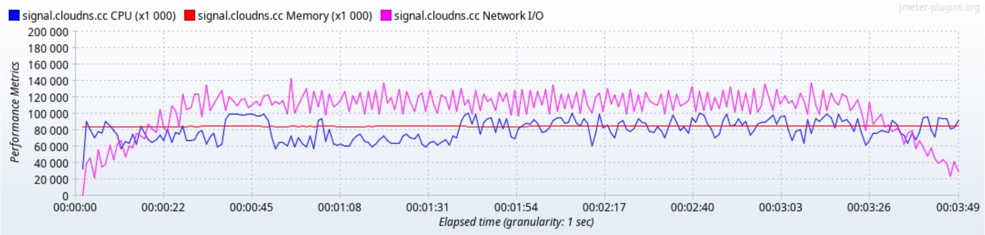


Figure 35: High load message sending on Signal 6.13

The following diagrams report the resource usage for the requests divided by type.

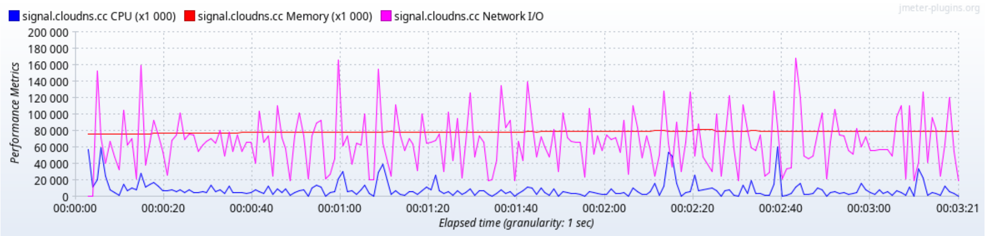


Figure 36: High load GET /v1/profile on Signal 6.13

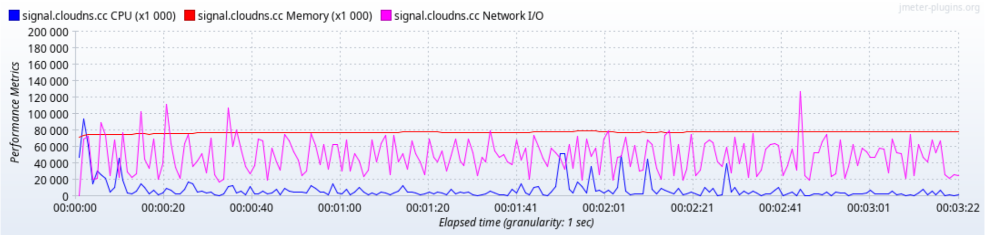


Figure 37: High load GET /v2/keys/+390123456789/* on Signal 6.13

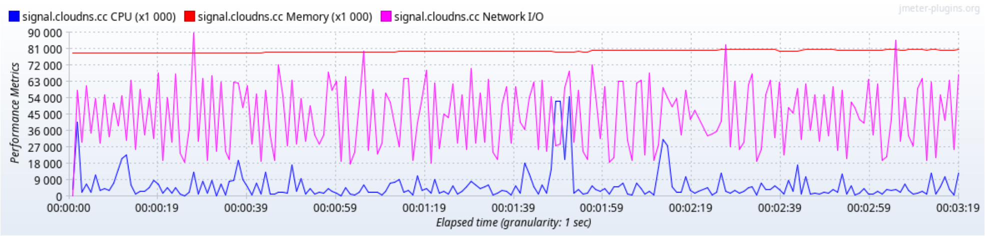


Figure 38: High load PUT /v1/messages on Signal 6.13

As in the previous experiments, there is no overload, only some queued requests between the minute 1 : 40 and the minute 1 : 50.

3.7 OBSERVATIONS

From the performed experiments there are no evidence about a load due to the single API calls. What can be deducted is that while the first architecture model, which uses PostgreSQL and less AWS services, reacts better to less intensive loads for long periods of time, while the second model, which uses DynamoDB and other AWS services not present before, reacts better to more intensive loads and more concentrated in a short period of time.

It is reasonable saying that the outage happened on January 2021 as described on the Signal community blog [12] 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.

4 CONCLUSIONS AND OUTLOOK

The load tests are used to simulate situations which are closed 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.

The lower load experiments with a higher duration refer to a single sequence of APIs calls to Signal, to complete an operation between an account creation and a message sending. This operation gets completed by a single user, for 1000 consecutive executions (see section 3.5.1 on page 8).

Instead, the higher load experiments with a lower duration refer to a maximum of 100 parallel users which can send single sequences of APIs calls, with a ramp-up period of 200 seconds. This means that a thread (which correspond to a user) is launched every 2 seconds. In these experiments the repetitions are 10 instead of 1000 (see section 3.5.2 on page 9).

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 long term low loads (see figures 8 on page 13 and 29 on page 20 compared to figures 19 on page 17 and 34 on page 22), while the 6.13 version supports better higher loads in short times (see figures 20 on page 17 and 35 on page 23 compared to figures 10 on page 14 and 30 on page 21). Moreover, the new version delegate the management of components to AWS, which is good to waste less time into them.

None of the two architecture seem to have weak points directly connected to a specific API call.

What we demonstrate 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 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.

4.1 OUTLOOK

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.

Other possible outlooks are the following:

- experimental analysis with more focus on each single component took by itself, with JMeter and plugins or other tools dedicated to this aim [13];
- analysis of the remaining API calls, as stated before;
- experiments on the AWS components, for example by using Distributed Load Testing [14];
- more realistic application scenarios based on real data, which can be provided by asking Open Whisper Systems [15].

BIBLIOGRAPHY

[1] 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. iii).

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

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

[4] 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. 1).

[5] Chris Richardson of Eventuate. *Introduction to microservices*. Aug. 2021. URL: https://www.nginx.com/blog/introduction-to-microservices/ (cit. on p. 2).

[6] 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 pp. 2, 4).

[7] Sorin Cocorada. *Signal Messanger Architecture*. June 2018. URL: https://sorincocorada.ro/signal-messanger-architecture/ (cit. on pp. 2, 4).

[8] Nick Woodard. *Secure Messenger Service Signal is down*. Jan. 2021. URL: https://screenrant.com/signal-down-secure-messenger-service-outage-explained/ (cit. on p. 2).

[9] 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. 2).

[10] *Technology preview: Private contact discovery for Signal*. URL: https://signal.org/blog/private-contact-discovery/ (cit. on p. 7).

[11] *Apache JMeter™*. URL: http://jmeter.apache.org/ (cit. on p. 8).

[12] 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. 24).

[13] Abhishek Dubey. *Redis load testing*. May 2019. URL: https://iamabhishek-dubey.medium.com/redis-load-testing-d99f81e97842 (cit. on p. 27).

[14] Stephen King and Tullio Dobner. *Distributed Load Testing on AWS*. 2019. URL: https://aws.amazon.com/it/solutions/implementations/distributed-load-testing-on-aws/ (cit. on p. 27).

[15] *Open whisper systems*. Nov. 2021. URL: https://en.wikipedia.org/wiki/Open_Whisper_Systems (cit. on p. 27).