JOHANNES KEPLER
UNIVERSITY LINZ

Submitted by
Christopher Warmbold

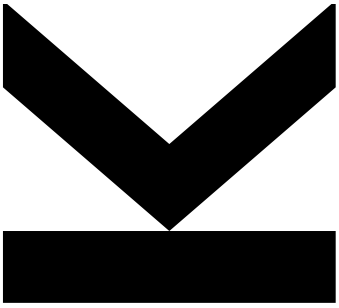
Submitted at
Institut für Systemsoftware

Supervisor
Christian Wirth

Co-Supervisor
Matthias Grimmer

October 2016

Node.js benchmark suite



Bachelor Thesis
to obtain the academic degree of
Bachelor of Science
in the Bachelor's Program
Informatik

JOHANNES KEPLER
UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Abstract

This thesis presents a way to benchmark typical *Node.js* applications. Node.js is a *JavaScript* based runtime environment mostly for server-sided web applications. There are already existing JavaScript benchmark suites which just test the performance of the core runtime. This thesis shows how to test the performance of a typical Node.js application with its typical characteristics, such as request handling, I/O performance and VM-level performance.

Firstly, it evaluates which packages are currently the most popular ones by analyzing the registry of the package manager *npm*. Then some of these popular packages which are considered as relevant for performance testing are used in some benchmark applications. The thesis also describes how these applications are benchmarked with *Apache JMeter*.

Kurzfassung

Diese Bachelorarbeit präsentiert eine Möglichkeit, typische *Node.js* Applikationen zu benchmarken. Node.js ist eine auf *JavaScript* basierende Plattform die hauptsächlich für serverseitige Webapplikationen verwendet wird. Es gibt bereits einige JavaScript Benchmark Suiten, welche aber nur das Kern-Laufzeitverhalten testen. Diese Arbeit zeigt wie man die Performance einer typischen Node.js Applikation und ihrer typischen Charakteristiken, wie zum Beispiel das Request-Handling, die I/O Performance und die VM-Level Performance, testen kann.

Zuerst werden die derzeit populärsten Pakete evaluiert, indem das Verzeichnis des Paket-Managers *npm* analysiert wird. Dann werden einige dieser Pakete, welche für das durchführen eines Benchmarks relevant sind, in verschiedenen Test-Applikationen verwendet. Weiters zeigt diese Bachelorarbeit, wie man diese Applikationen mit *Apache JMeter* testen und auswerten kann.

Contents

Chapter 1

Introduction

Node.js is a JavaScript based runtime environment mostly for server-sided web applications. To test the performance of engines executing Node.js applications some Node.js specific benchmarks are needed.

There are already some Node.js or JavaScript benchmarks but none of them test the typical characteristics of Node.js applications, such as parallel request handling, IO performance and VM-level performance. For this reason some sample applications which behave like typical real world Node.js applications will be developed. These applications should have many dependencies to popular libraries. Therefore the package manager npm will be analyzed in order to find the most popular packages. Some of these packages which are considered as relevant for performance testing will be used within the sample applications.

The performance of the sample applications will then be tested with the help of an existing workload generator called Apache JMeter. The results of these tests will be discussed and the bottlenecks will be identified.

Chapter 2

Node.js Ecosystem

Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js' package ecosystem, npm, is the largest ecosystem of open source libraries in the world.[18]

2.1 Packages

Node.js has a built in package manager called *nmp* (node package manager). In order to keep the Node.js core as lightweight as possible a lot of functionality comes with packages. Npm provides a public package registry where everyone can publish packages which can be used in any Node.js application[24].

2.1.1 Popular Packages

The goal of the benchmark suite it to test the performance of the underlying Node.js runtime. This means that the benchmarks should cover typical use cases of Node.js applications. This could be archived by using the most popular npm packages within the benchmarked applications.

There are different ways of measuring the popularity of npm modules. Each npm module can get a *star* from any developer. This are currently the most starred packages[20]:

Table 2.1: Most starred npm packages

package	stars
express	1719

Table 2.1: Most starred npm packages

package	stars
gulp	1086
request	948
async	922
lodash	857
browserify	729
pm2	703
grunt	616
commander	607
socket.io	594
forever	541
moment	532
mongoose	514
mocha	497
bower	483
chalk	466
underscore	431
gulp-uglify	416
cheerio	404
debug	398
q	367
npm	366
passport	357
react	356
bluebird	352
gulp-concat	352
nodemailer	351
colors	343
redis	332
gulp-sass	330

Note: All bold packages are used by the benchmarked applications (see Chapter 3).

Another method of measuring the popularity of packages is to find the most depended-upon packages. This are the currently the most depended-upon packages[19]:

Table 2.2: Most depended-upon packages

package	stars
lodash	27018
request	16534
async	15407
underscore	12799
express	10953
commander	10816
chalk	10640
debug	9700
bluebird	8754
mkdirp	7147
colors	6683
moment	6335
through2	6200
q	6129
yeoman-generator	5497
react	5343
glob	5172
minimist	4945
gulp-util	4885
coffee-script	4646
fs-extra	4148
body-parser	4075
jquery	4017
cheerio	3792
optimist	3413
yargs	3305
node-uuid	2968
yosay	2922
babel-runtime	2919
gulp	2915

Note: All bold packages are used by the benchmarked applications (see Chapter 3).

Chapter 3

Benchmark Applications

This chapter introduces the applications which are used by the benchmark suite. The benchmarks should test typical Node.js characteristics. Therefore these applications are using many of the popular packages from npm which were discussed Section 2.1.1.

With Node.js it is not only possible to write web applications but also to write console applications. The benchmark suite should not test the same functionality again and again and for this reason three different applications were created:

- I/O intensive application → should produce many I/O-operations
- CPU intensive application → should be computation heavy
- Rich web application → should have many dependencies and should behave like a modern web application

3.1 I/O-Application

This console application copies all files from a given source directory to a given target directory.

3.1.1 Usage

Output of `node app --help`:

Usage: app sourceDir targetDir [options]

I/O-Application: copies all files from the source directory to the target directory.

Options:

-h, --help output usage information
-V, --version output the version number
-c --copyfunction <function> select copyfunction to use. fse (=fs-extra) or ncp. fallback is fse.

I found two popular packages, `fs-extra`[5] and `ncp`[14], which can copy the contents of a directory recursively. I tried both of them and sometimes they were quite different in aspect of the runtime. Because of that I decided to add the option to switch between those packages with the `-c` option.

3.1.2 Used Packages

This application uses following packages:

Table 3.1: I/O-application dependencies

package	version
colors[2]	1.1.2
commander[3]	2.9.0
fs-extra[5]	0.30.0
mkdirp[13]	0.5.1
ncp[14]	2.0.0
verror[23]	1.6.1

3.1.3 Known Issues

When using this application with a huge amount of files the applications gets not only I/O intensive but also the CPU load reaches a high level.

3.2 CPU-Application

This console application compresses and encrypts a given file or directory. It is also able to decrypt and decompress a given file. Furthermore you can choose the encryption algorithm and set the encryption password. In order to increase the CPU load it is also possible to set a count how often the encryption or decryption algorithm should be applied.

3.2.1 Usage

Output of `node app --help`:

Usage: app [options] <source>

CPU-Application: compress and encrypt a file/directory.

Options:

-h, --help output usage information
-V, --version output the version number
-a --algorithm <cipheralgorithm> set cipher algorithm. default: CAST-cbc
-c --count <count> how often the cipher algorithm should be applied
-p --password <password> set encryption password. default password is "password" -d --decrypt decrypt file

3.2.2 Used Packages

This application uses following packages:

Table 3.2: CPU-application dependencies

package	version
colors[2]	1.1.2
commander[3]	2.9.0
crypto[4]	0.0.3
fstream[6]	1.0.10
path[21]	0.12.7
tar[22]	2.2.1

Table 3.2: CPU-application dependencies

package	version
verror[23]	1.6.1

3.2.3 Known Issues

Because Node.js is running single threaded the CPU load of just one core is going to rise. Node.js could support multiple cores with its **child_process.fork() API** or with the **cluster** module but this is not implemented in this application (see Chapter 7)[15][16][17].

3.3 Web Application

The goal of this application is to be as realistic as possible. This means that it should use technologies and frameworks which are used in many real world applications. Furthermore it should have many dependencies to popular third party libraries and follow the programming guidelines of Node.js. The effort to write such an application would exceed the scope of this thesis and therefore the best way to go is to use an existing platform. The currently most starred package in npm is *express* (see Section 2.1.1 which is a framework for building web applications[20]. For this reason this application uses a publishing platform which is based on express called *ghost*[10][9].

3.3.1 Usage

To start ghost following command has to be executed inside the folder of the application: `npm start`. The default port is *2368* and the back-end is accessible via `http://localhost:2368/ghost`. The application has following settings:

Username:	benchmark.suite@jku.at
Password:	password
Public API:	enabled

3.3.2 Used Packages

This application only uses ghost[11]. But ghost itself has many dependencies:

Table 3.3: Ghost dependencies

package	version
amperize	0.3.1
archiver	1.1.0
bcryptjs	2.3.0
bluebird	3.4.6
body-parser	1.15.2
bookshelf	0.10.1
chalk	1.1.3
cheerio	0.22.0
compression	1.6.2
connect-slashes	1.3.1
cookie-session	1.2.0
cors	2.8.1
csv-parser	1.11.0
downsize	0.0.8
express	4.14.0
express-hbs	1.0.3
extract-zip-fork	1.5.1
fs-extra	0.30.0
ghost-gql	0.0.5
glob	5.0.15
gscan	0.0.15
html-to-text	2.1.3
image-size	0.5.0
intl	1.2.5
intl-messageformat	1.3.0
jsonpath	0.2.7
knex	0.12.2
lodash	4.16.0
moment	2.15.1
moment-timezone	0.5.5
morgan	1.7.0
multer	1.2.0
netjet	1.1.3

Table 3.3: Ghost dependencies

package	version
node-uuid	1.4.7
nodemailer	0.7.1
oauth2orize	1.5.0
passport	0.3.2
passport-http-bearer	1.0.1
passport-oauth2-client-password	0.1.2
path-match	1.2.4
rss	1.2.1
sanitize-html	1.13.0
semver	5.3.0
showdown-ghost	0.3.6
sqlite3	3.1.4
superagent	2.3.0
unidecode	0.1.8
validator	5.7.0
xml	1.0.1

3.3.3 Known Issues

As mentioned in Section 3.2.3, Node.js is not supporting multiple cores by default. Because of that ghost can not handle parallel requests. This causes requests to wait till the previous requests are finished. It is possible to scale ghost with the cluster module but this is not implemented in the benchmark suite (see Chapter 7)[8][17].

First I tried to add ghost as npm module[7]. In order to start ghost you have to install all dependencies by calling `npm install`. This command will download ghost and all its dependencies inside the *node_modules* folder. The *content* folder where the configuration and the database are saved are then also located inside the *node_modules* folder. Typically you add *node_modules* to your *.gitignore* file so that these files are not uploaded to the git repository but in this case where you want to keep the stored configuration and database you have to include it. Ghost has many dependencies and this causes very long filenames inside this folder unfortunately the windows git client has a small character limit for filenames as a result you might not be able to push or pull from the git repository. This limit could be bypassed by editing the git configuration with the `git config --system core.longpaths true` command but I decided to switch to the standalone ghost release where the *content* folder is not placed inside the *node_modules* folder and so this folder can be excluded from git.

Chapter 4

Benchmark System

The goal of the benchmark system is to test the applications described in Chapter 3 under realistic workload conditions. For this reason the benchmark system uses an existing workload generation tool called *Apache JMeter*[1]. Furthermore an utility script which wraps all together and starts the benchmarks is provided.

4.1 Apache JMeter

Apache JMeter is an open source load testing tool written in Java. It is a powerful tool to generate workload and measure the performance of any static or dynamic resource (Webservices, Web dynamic languages - PHP, Java, ASP.NET, Files, Java Objects, Data Bases and Queries, FTP Servers and more)[1].

JMeter can send predefined HTTP requests to the Node.js applications and measure the response time. It is possible to set how often these requests should be looped or how many threads should send the predefined requests simultaneously to simulate multiple parallel user requests.

4.2 Web Application Wrapper

As mentioned in Section 4.1 the applications need to be accessible via HTTP but as described in Chapter 3 the benchmark suite contains two console applications. For this reason a web application wrapper was developed. This wrapper application wraps a given console application and starts a web server where the console application can be accessed. The console parameters can be passed in the

query string of a request. The console application is called synchronously in order to measure the execution time and the results are sent to the HTTP client. This allows the benchmarking suite only to measure the response time of the requests.

4.2.1 Usage

Output of `node app --help`:

> Usage: app [options] <source>
>
> Webserver-Wrapper-Application: wraps a console application in a web application.
>
> Options:
>
> -h, --help output usage information
> -V, --version output the version number
> -a --application <application> set the application which should be wrapped
> -p --port <port> set the port of the web server

Example:

The following example starts the I/O application on port 8081.

```
node webserver-wrapper\app.js -a "node io-application\app.js" -p 8081
```

Sample Requests:

The following examples show how to pass parameters to the application.

```
http://localhost:8081?args=--help
http://localhost:8081?args=sourceDir%20targetDir
```

4.2.2 Used Packages

This application uses following packages:

Table 4.1: Web Application Wrapper dependencies

package	version
child_process	1.0.2
colors	1.1.2
commander	2.9.0
express	4.14.0

4.3 Sample Files

Two of the applications described in Chapter 3 need files or a folder as input. Therefore the benchmark suite contains some sample files such as audio files, video files, documents and an empty ghost application. The files should represent real data. The empty ghost application was added because it has a big file structure with more than 400 files (without dependencies).

Source of the media files: `https://kodi.tv/media-samples/`

This sample-data folder ensures that the tests have the same conditions and work with the same data. The contents of this folder can be modified because the benchmarks always use the whole folder for its requests but it should stay the same when you want to compare the results.

4.4 Benchmarks

4.4.1 I/O-Application

Following scenarios are covered by the benchmark:

- copy sample-data to *tmp\io_<threadId>_<loopCounter>*
- copy sample-data to *tmp\io_<threadId>_<loopCounter>* with ncp as copy library

To prevent different threads and loop iterations from copying to the same folder the thread-id and the loop-counter are added to the target directory folder.

The test cases above are executed in different ways:

- looped 10 times in 1 thread → total 10 times
- executed once in 10 threads with a ramp-up period of 1 second → total 10 times

The ramp-up period tells JMeter how long to take till all threads are started. For example if 30 threads are used and the ramp-up period is 10 seconds, JMeter will start all 30 threads within 10 seconds.

4.4.2 CPU-Application

Following scenarios are covered by the benchmark:

- compress sample-data to *tmp\cpu_<threadId>_<loopCounter>*
- compress sample-data to *tmp\cpu_c5_<threadId>_<loopCounter>* with encryption count 5

To prevent different threads and loop iterations from writing to the same folder the thread-id and the loop-counter are added to the target directory folder.

The test cases above are executed in different ways:

- looped 10 times in 1 thread → total 10 times
- executed once in 10 threads with a ramp-up period of 1 second → total 10 times

The ramp-up period tells JMeter how long to take till all threads are started. For example if 30 threads are used and the ramp-up period is 10 seconds, JMeter will start all 30 threads within 10 seconds.

4.4.3 Web Application

Following scenario is covered by the benchmark:

1. go to the sign-in page

2. get an access_token (has to be extracted and stored in a JMeter variable in order to use it for future requests)

3. navigating through the back-end

4. open the creation view of a new article

5. add some text to the new article. Ghost sends some AJAX requests during typing → these requests were recorded and are simulated during the benchmark. The id of the created post has also to be extracted and stored in a JMeter variable.

6. publish the post

7. navigate to the main page where new posts are displayed

The test cases above are executed in different ways:

- looped 30 times in 1 thread → total 30 times

- executed once in 30 threads with a ramp-up period of 10 second → total 30 times

The ramp-up period tells JMeter how long to take till all threads are started. For example if 30 threads are used and the ramp-up period is 10 seconds, JMeter will start all 30 threads within 10 seconds.

The database of ghost has to be cleared to ensure the same conditions for future benchmarks. Ghost offers a "Delete all Content" button in the back-end. Therefore a separate JMeter testplan was created which sends a request to this page. It is executed at the end of a benchmark.

4.5 Utility Script

The utility script wraps all benchmarks together and it also handles the startup of the sample applications described in Chapter 3.

Following steps are included:

1. install the npm packages of all applications

2. start the ghost application
3. start a web server wrapper for the I/O-application on port 8081
4. start a web server wrapper for the CPU-application on port 8082
5. wait 15 seconds for all servers to startup
6. execute the I/O-application benchmarks looped
7. delete the *tmp* folder
8. execute the I/O-application benchmarks threaded
9. delete the *tmp* folder
10. execute the CPU-application benchmarks looped
11. delete the *tmp* folder
12. execute the CPU-application benchmarks threaded
13. delete the *tmp* folder
14. execute the ghost benchmarks looped
15. clear the ghost database
16. execute the ghost benchmarks threaded
17. clear the ghost database

Caution: The JMeter binary folder has to be added to the PATH environment variable.

All the JMeter test are started with following command:

```
$ jmeter -n -t <pathToTestplan> -l results\output_<testPlanName>.jtl
```

4.5.1 Usage

The utility script can be executed in a terminal with:

```
$ startBenchmark
```

The results are saved in the *results* folder.

Chapter 5

Results

As described in Section 4.5, JMeter testplans can be executed on the command line. But for this chapter the JMeter GUI is used because it allows to generate graphs and it can summarize the results of a test. Each benchmark described in Section 4.4 can be opened with the JMeter GUI. The .jmx files are placed under `jmeter\<application-name>`.

5.1 Test Setup

The following benchmarks were executed with the official Node.js runtime.

Following setup was used:

CPU	Intel Core i5-4690K (4 x 3.5 GHz)
RAM	8GB
Storage	Samsung SSD 830 128GB
GPU	Nvidia Geforce GTX 960
OS	Windows 10 Pro
Node	Version 4.2.3
NPM	Version 3.10.8

5.2 I/O-Application

As described in Section 4.4.1 the benchmarks are performed multi-threaded and looped.

5.2.1 Looped

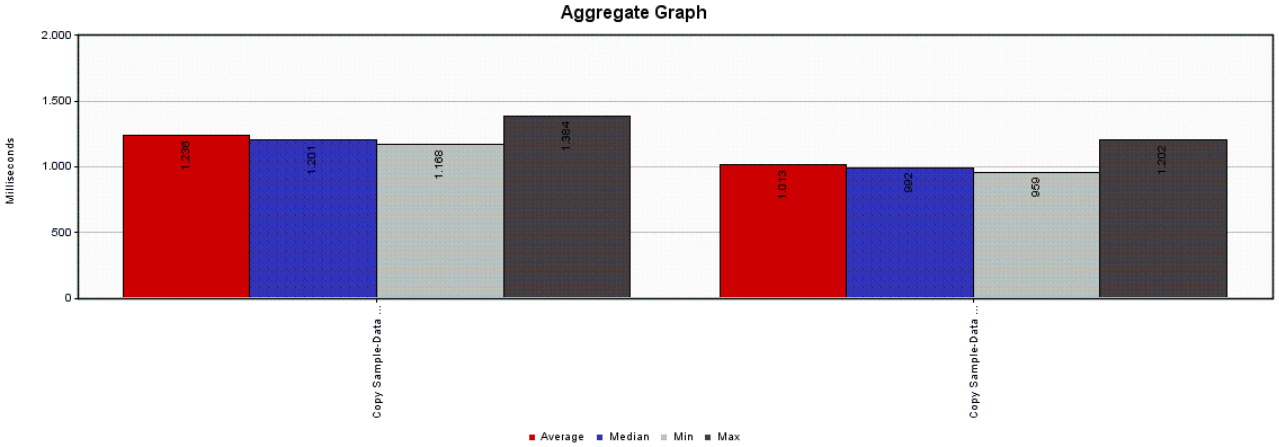
In this test the benchmarks are executed 10 times in 1 thread.

Figure 5.1: Results view of I/O-application when executed synchronously

Sample #	Start Time	Thread Name	Label	Sample Time(ms)	Status	Bytes	Latency	Connect Time(...
1	12:59:46.723	Thread-Gruppe 1-1	Copy Sample-Data to tmp/	1200		287	1200	1
2	12:59:47.924	Thread-Gruppe 1-1	Copy Sample-Data to tmp/ with ncp	992		267	992	0
3	12:59:48.916	Thread-Gruppe 1-1	Copy Sample-Data to tmp/	1201		287	1201	1
4	12:59:50.117	Thread-Gruppe 1-1	Copy Sample-Data to tmp/ with ncp	959		344	959	0
5	12:59:51.079	Thread-Gruppe 1-1	Copy Sample-Data to tmp/	1300		288	1300	0
6	12:59:52.380	Thread-Gruppe 1-1	Copy Sample-Data to tmp/ with ncp	978		267	978	0
7	12:59:53.359	Thread-Gruppe 1-1	Copy Sample-Data to tmp/	1174		287	1174	0
8	12:59:54.534	Thread-Gruppe 1-1	Copy Sample-Data to tmp/ with ncp	1010		307	1010	0
9	12:59:55.544	Thread-Gruppe 1-1	Copy Sample-Data to tmp/	1178		287	1178	0
10	12:59:56.723	Thread-Gruppe 1-1	Copy Sample-Data to tmp/ with ncp	981		267	981	0
11	12:59:57.705	Thread-Gruppe 1-1	Copy Sample-Data to tmp/	1168		287	1168	1
12	12:59:58.874	Thread-Gruppe 1-1	Copy Sample-Data to tmp/ with ncp	978		344	978	0
13	12:59:59.852	Thread-Gruppe 1-1	Copy Sample-Data to tmp/	1259		288	1259	1
14	13:00:01.111	Thread-Gruppe 1-1	Copy Sample-Data to tmp/ with ncp	1013		268	1013	0
15	13:00:02.125	Thread-Gruppe 1-1	Copy Sample-Data to tmp/	1210		287	1210	0
16	13:00:03.336	Thread-Gruppe 1-1	Copy Sample-Data to tmp/ with ncp	1025		307	1025	0
17	13:00:04.362	Thread-Gruppe 1-1	Copy Sample-Data to tmp/	1291		288	1291	0
18	13:00:05.653	Thread-Gruppe 1-1	Copy Sample-Data to tmp/ with ncp	998		306	998	0
19	13:00:06.651	Thread-Gruppe 1-1	Copy Sample-Data to tmp/	1384		288	1384	1
20	13:00:08.036	Thread-Gruppe 1-1	Copy Sample-Data to tmp/ with ncp	1202		307	1202	0

Scroll automatically? Child samples? No of Samples 20 Latest Sample 1202 Average 1125 Deviation 129

Figure 5.2: Aggregate Graph of I/O-application when executed synchronously



In Figure 5.1 you see the requests JMeter has generated and in Figure 5.2 you see the aggregated results. The aggregated results show that copying with the `ncp` library is slightly faster than with the `fs-extra` library.

5.2.2 Multi-threaded

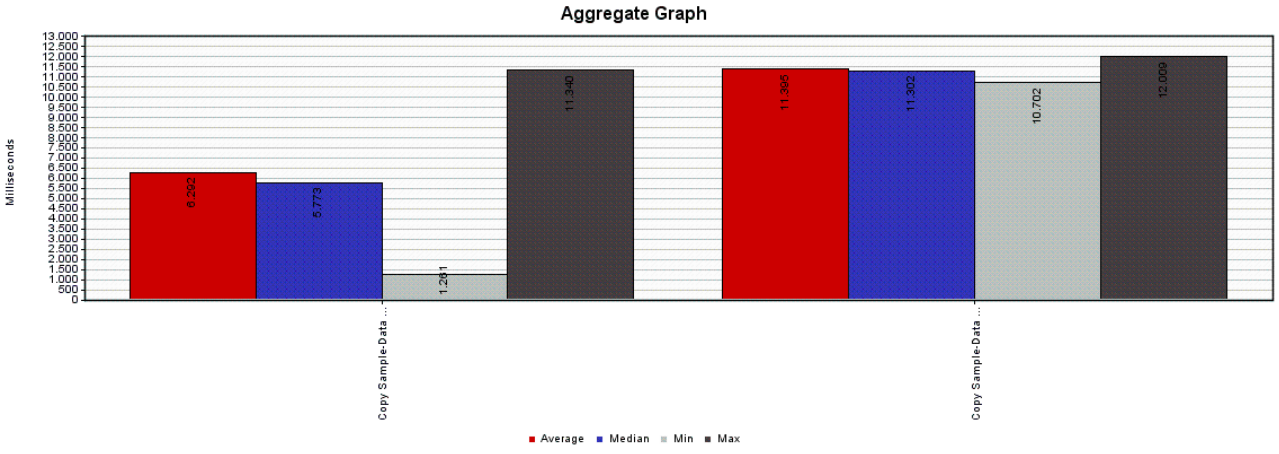
In this test the benchmarks are executed once in 10 threads with a ramp-up time of 1 second.

Figure 5.3: Results view of I/O-application when executed parallel

Sample #	Start Time	Thread Name	Label	Sample Time(ms)	Status	Bytes	Latency	Connect Time(ms)
1	13:01:53.139	Thread-Gruppe 1-1	Copy Sample-Data to...	1261		288	1261	3
2	13:01:53.240	Thread-Gruppe 1-2	Copy Sample-Data to...	2368		287	2368	1
3	13:01:53.343	Thread-Gruppe 1-3	Copy Sample-Data to...	3541		288	3541	1
4	13:01:53.443	Thread-Gruppe 1-4	Copy Sample-Data to...	4672		288	4672	1
5	13:01:53.543	Thread-Gruppe 1-5	Copy Sample-Data to...	5773		287	5773	1
6	13:01:53.643	Thread-Gruppe 1-6	Copy Sample-Data to...	6841		287	6841	0
7	13:01:53.743	Thread-Gruppe 1-7	Copy Sample-Data to...	7917		287	7917	1
8	13:01:53.842	Thread-Gruppe 1-8	Copy Sample-Data to...	8992		287	8992	1
9	13:01:53.942	Thread-Gruppe 1-9	Copy Sample-Data to...	10224		288	10224	1
10	13:01:54.042	Thread-Gruppe 1-10	Copy Sample-Data to...	11340		287	11340	1
11	13:01:54.402	Thread-Gruppe 1-1	Copy Sample-Data to...	12009		307	12009	0
12	13:01:55.609	Thread-Gruppe 1-2	Copy Sample-Data to...	11897		307	11897	1
13	13:01:56.884	Thread-Gruppe 1-3	Copy Sample-Data to...	11812		307	11812	1
14	13:01:58.115	Thread-Gruppe 1-4	Copy Sample-Data to...	11604		307	11604	1
15	13:01:59.317	Thread-Gruppe 1-5	Copy Sample-Data to...	11472		268	11472	1
16	13:02:00.485	Thread-Gruppe 1-6	Copy Sample-Data to...	11302		267	11302	0
17	13:02:01.660	Thread-Gruppe 1-7	Copy Sample-Data to...	11159		307	11159	1
18	13:02:02.834	Thread-Gruppe 1-8	Copy Sample-Data to...	11099		307	11099	1
19	13:02:04.167	Thread-Gruppe 1-9	Copy Sample-Data to...	10894		307	10894	0
20	13:02:05.383	Thread-Gruppe 1-10	Copy Sample-Data to...	10702		268	10702	0

Scroll automatically? Child samples? No of Samples 20 **Latest Sample** 10702 Average 8843 Deviation 3422

Figure 5.4: Aggregate Graph of I/O-application when executed parallel



In Figure 5.3 and in Figure 5.4 you can see that Node.js is single threaded and therefore multiple requests at the same time are queued and handled synchronously. Each request has to wait for the previous requests to finish, this increases the response time. Furthermore you can see that JMeter sent the request where `ncp` is used as copy function later than the request where `fs-extra` is used. Because of this the minimum response time of the `ncp` copy requests is above 10 seconds.

5.3 CPU-Application

As described in Section 4.4.2 the benchmarks are performed multi-threaded and looped.

5.3.1 Looped

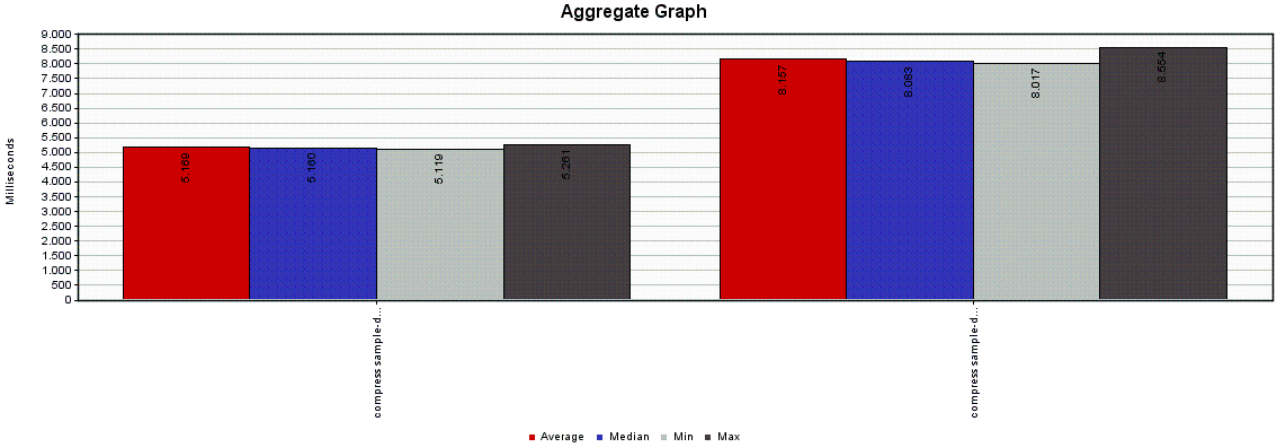
In this test the benchmarks are executed 10 times in 1 thread.

Figure 5.5: Results view of CPU-application when executed synchronously

Sample #	Start Time	Thread Name	Label	Sample Time(ms)	Status	Bytes	Latency	Connect Time(ms)
1	13:09:47.551	Thread-Gruppe 1-1	compress sample-d...	5179		237	5179	2
2	13:09:52.731	Thread-Gruppe 1-1	compress sample-d...	8083		237	8083	0
3	13:10:00.814	Thread-Gruppe 1-1	compress sample-d...	5186		237	5186	1
4	13:10:06.001	Thread-Gruppe 1-1	compress sample-d...	8170		237	8170	0
5	13:10:14.171	Thread-Gruppe 1-1	compress sample-d...	5173		237	5173	1
6	13:10:19.344	Thread-Gruppe 1-1	compress sample-d...	8220		237	8220	1
7	13:10:27.565	Thread-Gruppe 1-1	compress sample-d...	5261		237	5261	1
8	13:10:32.826	Thread-Gruppe 1-1	compress sample-d...	8033		237	8033	1
9	13:10:40.859	Thread-Gruppe 1-1	compress sample-d...	5173		237	5173	1
10	13:10:46.032	Thread-Gruppe 1-1	compress sample-d...	8103		237	8103	1
11	13:10:54.136	Thread-Gruppe 1-1	compress sample-d...	5159		237	5159	0
12	13:10:59.295	Thread-Gruppe 1-1	compress sample-d...	8017		237	8017	1
13	13:11:07.313	Thread-Gruppe 1-1	compress sample-d...	5119		237	5119	0
14	13:11:12.433	Thread-Gruppe 1-1	compress sample-d...	8078		237	8078	0
15	13:11:20.513	Thread-Gruppe 1-1	compress sample-d...	5159		237	5159	0
16	13:11:25.673	Thread-Gruppe 1-1	compress sample-d...	8238		237	8238	0
17	13:11:33.912	Thread-Gruppe 1-1	compress sample-d...	5122		237	5122	0
18	13:11:39.033	Thread-Gruppe 1-1	compress sample-d...	8082		237	8082	1
19	13:11:47.116	Thread-Gruppe 1-1	compress sample-d...	5160		237	5160	1
20	13:11:52.276	Thread-Gruppe 1-1	compress sample-d...	8554		237	8554	1

Scroll automatically? Child samples? No of Samples 20 Latest Sample 8554 Average 6663 Deviation 1498

Figure 5.6: Aggregate Graph of CPU-application when executed synchronously



In Figure 5.5 you see the requests JMeter has generated and in Figure 5.6 you see the aggregated results. The aggregated results show that request where the encryption counter (see Section 3.2.1) is set to 5 have a longer response time and therefore a longer computation time than the requests where the files are only encrypted once.

5.3.2 Multi-threaded

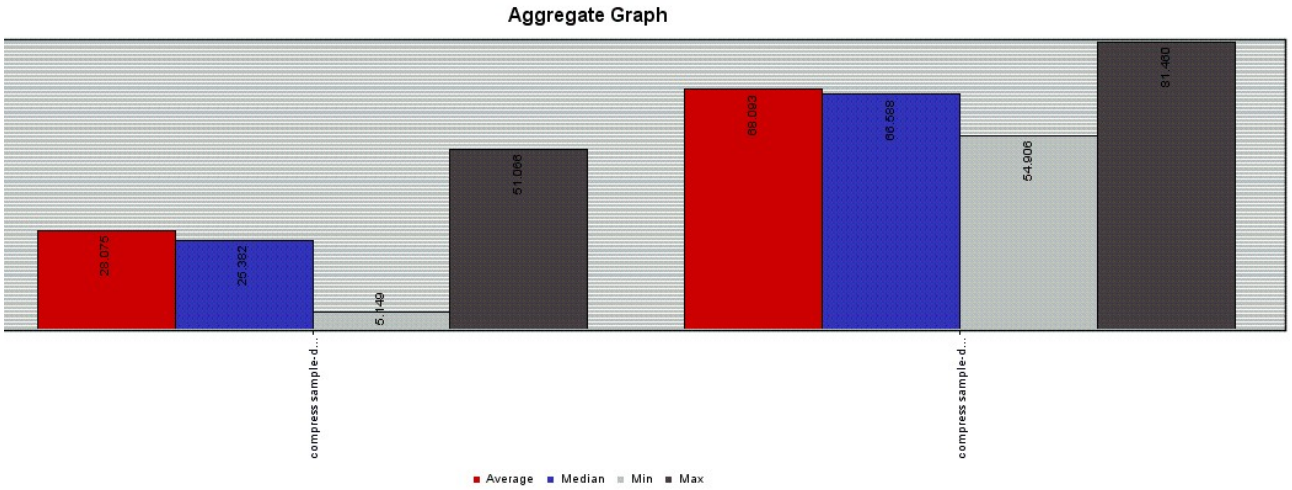
In this test the benchmarks are executed once in 10 threads with a ramp-up time of 1 second.

Figure 5.7: Results view of CPU-application when executed parallel

Sample #	Start Time	Thread Name	Label	Sample Time(ms)	Status	Bytes	Latency	Connect Time(ms)
1	13:13:49.021	Thread-Gruppe 1-1	compress sample-d...	5149		237	5149	3
2	13:13:49.121	Thread-Gruppe 1-2	compress sample-d...	10202		237	10202	1
3	13:13:49.221	Thread-Gruppe 1-3	compress sample-d...	15291		237	15291	1
4	13:13:49.320	Thread-Gruppe 1-4	compress sample-d...	20326		237	20326	1
5	13:13:49.421	Thread-Gruppe 1-5	compress sample-d...	25382		237	25382	0
6	13:13:49.520	Thread-Gruppe 1-6	compress sample-d...	30517		237	30517	2
7	13:13:49.620	Thread-Gruppe 1-7	compress sample-d...	35782		237	35782	0
8	13:13:49.724	Thread-Gruppe 1-8	compress sample-d...	40990		237	40990	1
9	13:13:49.826	Thread-Gruppe 1-9	compress sample-d...	46048		237	46048	1
10	13:13:49.923	Thread-Gruppe 1-10	compress sample-d...	51066		237	51066	0
11	13:13:54.172	Thread-Gruppe 1-1	compress sample-d...	54906		237	54906	0
12	13:13:59.326	Thread-Gruppe 1-2	compress sample-d...	57897		237	57897	1
13	13:14:04.512	Thread-Gruppe 1-3	compress sample-d...	60795		237	60795	1
14	13:14:09.646	Thread-Gruppe 1-4	compress sample-d...	63653		237	63653	1
15	13:14:14.804	Thread-Gruppe 1-5	compress sample-d...	66588		237	66588	0
16	13:14:20.038	Thread-Gruppe 1-6	compress sample-d...	69456		237	69456	1
17	13:14:25.403	Thread-Gruppe 1-7	compress sample-d...	72265		237	72265	1
18	13:14:30.714	Thread-Gruppe 1-8	compress sample-d...	75467		237	75467	1
19	13:14:35.875	Thread-Gruppe 1-9	compress sample-d...	78443		237	78443	0
20	13:14:40.989	Thread-Gruppe 1-10	compress sample-d...	81460		237	81460	1

Scroll automatically? Child samples? No of Samples 20 Latest Sample 81460 Average 48084 Deviation 23322

Figure 5.8: Aggregate Graph of CPU-application when executed parallel



As described in Section 5.2.2 you see that the application is not optimized for multiple requests at the same time. All requests are queued and handled synchronously.

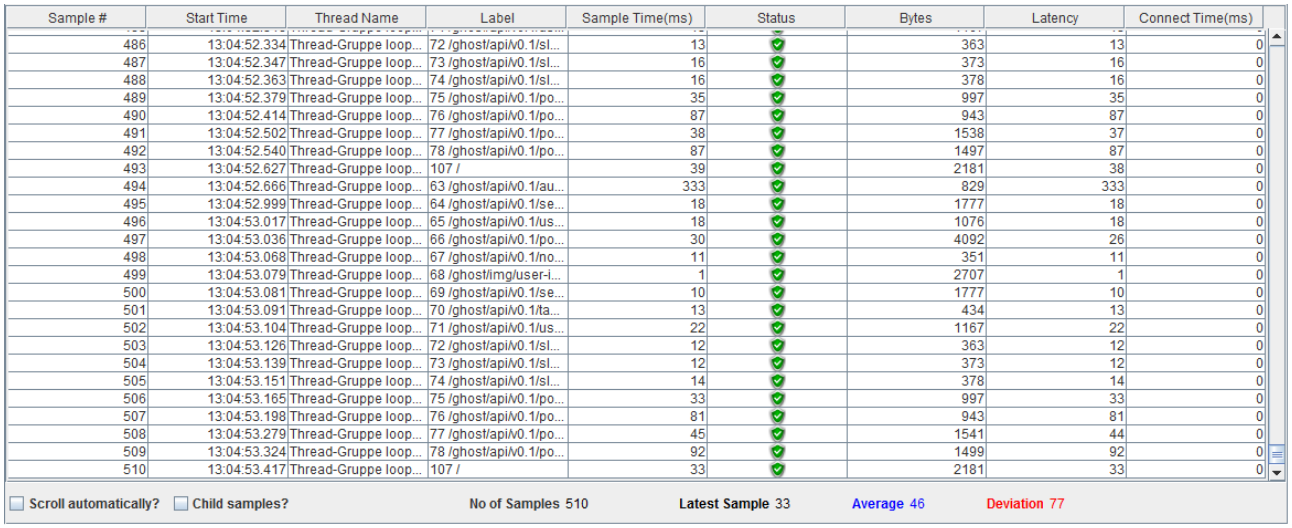
5.4 Web Application

As described in Section 4.4.3 the benchmarks are performed multi-threaded and looped.

5.4.1 Looped

In this test the benchmarks are executed 30 times in 1 thread.

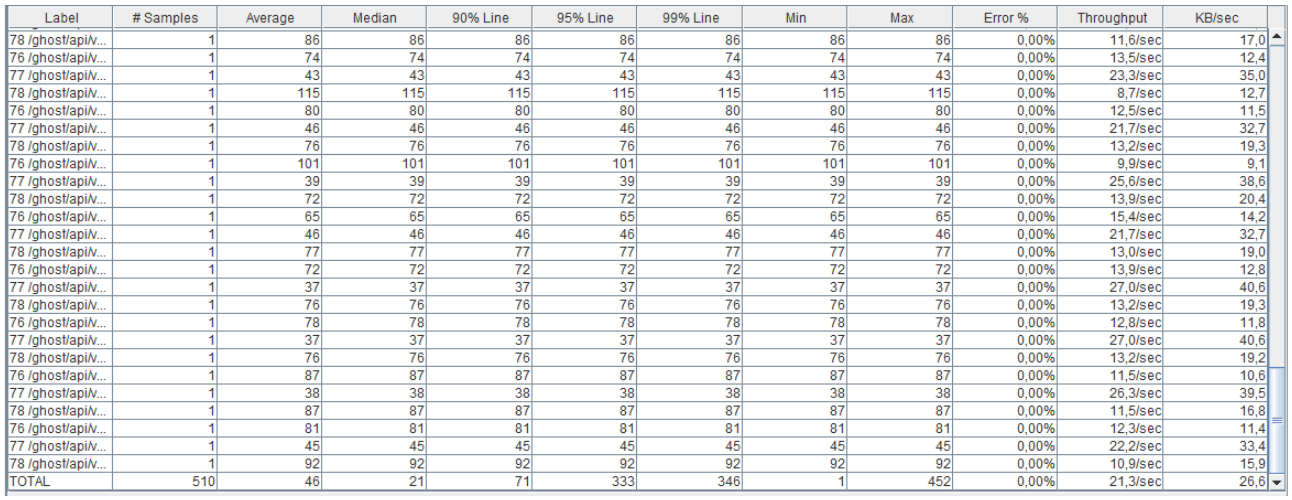
Figure 5.9: Results view of ghost when executed synchronously



Sample #	Start Time	Thread Name	Label	Sample Time(ms)	Status	Bytes	Latency	Connect Time(ms)
486	13:04:52.334	Thread-Gruppe loop...	72 /ghost/api/v0.1/sl...	13		363	13	0
487	13:04:52.347	Thread-Gruppe loop...	73 /ghost/api/v0.1/sl...	16		373	16	0
488	13:04:52.363	Thread-Gruppe loop...	74 /ghost/api/v0.1/sl...	16		378	16	0
489	13:04:52.379	Thread-Gruppe loop...	75 /ghost/api/v0.1/po...	35		997	35	0
490	13:04:52.414	Thread-Gruppe loop...	76 /ghost/api/v0.1/po...	87		943	87	0
491	13:04:52.502	Thread-Gruppe loop...	77 /ghost/api/v0.1/po...	38		1538	37	0
492	13:04:52.540	Thread-Gruppe loop...	78 /ghost/api/v0.1/po...	87		1497	87	0
493	13:04:52.627	Thread-Gruppe loop...	107 /	39		2181	38	0
494	13:04:52.666	Thread-Gruppe loop...	63 /ghost/api/v0.1/au...	333		829	333	0
495	13:04:52.999	Thread-Gruppe loop...	64 /ghost/api/v0.1/se...	18		1777	18	0
496	13:04:53.017	Thread-Gruppe loop...	65 /ghost/api/v0.1/us...	18		1076	18	0
497	13:04:53.036	Thread-Gruppe loop...	66 /ghost/api/v0.1/po...	30		4092	26	0
498	13:04:53.068	Thread-Gruppe loop...	67 /ghost/api/v0.1/no...	11		351	11	0
499	13:04:53.079	Thread-Gruppe loop...	68 /ghost/img/user-i...	1		2707	1	0
500	13:04:53.081	Thread-Gruppe loop...	69 /ghost/api/v0.1/se...	10		1777	10	0
501	13:04:53.091	Thread-Gruppe loop...	70 /ghost/api/v0.1/ta...	13		434	13	0
502	13:04:53.104	Thread-Gruppe loop...	71 /ghost/api/v0.1/us...	22		1167	22	0
503	13:04:53.126	Thread-Gruppe loop...	72 /ghost/api/v0.1/sl...	12		363	12	0
504	13:04:53.139	Thread-Gruppe loop...	73 /ghost/api/v0.1/sl...	12		373	12	0
505	13:04:53.151	Thread-Gruppe loop...	74 /ghost/api/v0.1/sl...	14		378	14	0
506	13:04:53.165	Thread-Gruppe loop...	75 /ghost/api/v0.1/po...	33		997	33	0
507	13:04:53.198	Thread-Gruppe loop...	76 /ghost/api/v0.1/po...	81		943	81	0
508	13:04:53.279	Thread-Gruppe loop...	77 /ghost/api/v0.1/po...	45		1541	44	0
509	13:04:53.324	Thread-Gruppe loop...	78 /ghost/api/v0.1/po...	92		1499	92	0
510	13:04:53.417	Thread-Gruppe loop...	107 /	33		2181	33	0

Scroll automatically? Child samples? No of Samples 510 Latest Sample 33 Average 46 Deviation 77

Figure 5.10: Report of ghost when executed synchronously



Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Max	Error %	Throughput	KB/sec
78 /ghost/api/v...	1	86	86	86	86	86	86	86	0,00%	11,6/sec	17,0
76 /ghost/api/v...	1	74	74	74	74	74	74	74	0,00%	13,5/sec	12,4
77 /ghost/api/v...	1	43	43	43	43	43	43	43	0,00%	23,3/sec	35,0
78 /ghost/api/v...	1	115	115	115	115	115	115	115	0,00%	8,7/sec	12,7
76 /ghost/api/v...	1	80	80	80	80	80	80	80	0,00%	12,5/sec	11,5
77 /ghost/api/v...	1	46	46	46	46	46	46	46	0,00%	21,7/sec	32,7
78 /ghost/api/v...	1	76	76	76	76	76	76	76	0,00%	13,2/sec	19,3
76 /ghost/api/v...	1	101	101	101	101	101	101	101	0,00%	9,9/sec	9,1
77 /ghost/api/v...	1	39	39	39	39	39	39	39	0,00%	25,6/sec	38,6
78 /ghost/api/v...	1	72	72	72	72	72	72	72	0,00%	13,9/sec	20,4
76 /ghost/api/v...	1	65	65	65	65	65	65	65	0,00%	15,4/sec	14,2
77 /ghost/api/v...	1	46	46	46	46	46	46	46	0,00%	21,7/sec	32,7
78 /ghost/api/v...	1	77	77	77	77	77	77	77	0,00%	13,0/sec	19,0
76 /ghost/api/v...	1	72	72	72	72	72	72	72	0,00%	13,9/sec	12,8
77 /ghost/api/v...	1	37	37	37	37	37	37	37	0,00%	27,0/sec	40,6
78 /ghost/api/v...	1	76	76	76	76	76	76	76	0,00%	13,2/sec	19,3
76 /ghost/api/v...	1	78	78	78	78	78	78	78	0,00%	12,8/sec	11,8
77 /ghost/api/v...	1	37	37	37	37	37	37	37	0,00%	27,0/sec	40,6
78 /ghost/api/v...	1	76	76	76	76	76	76	76	0,00%	13,2/sec	19,2
76 /ghost/api/v...	1	87	87	87	87	87	87	87	0,00%	11,5/sec	10,6
77 /ghost/api/v...	1	38	38	38	38	38	38	38	0,00%	26,3/sec	39,5
78 /ghost/api/v...	1	87	87	87	87	87	87	87	0,00%	11,5/sec	16,8
76 /ghost/api/v...	1	81	81	81	81	81	81	81	0,00%	12,3/sec	11,4
77 /ghost/api/v...	1	45	45	45	45	45	45	45	0,00%	22,2/sec	33,4
78 /ghost/api/v...	1	92	92	92	92	92	92	92	0,00%	10,9/sec	15,9
TOTAL	510	46	21	71	333	346	1	452	0,00%	21,3/sec	26,6

In Figure 5.9 you can see the last 25 of 510 samples. In Figure 5.10 you see the report of the different requests. The last line of Figure 5.10 shows the summary of all requests.

Following metrics might be interesting:

Table 5.1: Ghost results looped

metric	value
Throughput	21.3/sec
Average	46ms
Median	21ms
Min	1ms
Max	452ms
Error	0%

5.4.2 Multi-threaded

In this test the benchmarks are executed once in 30 threads with a ramp-up time of 10 second.

Figure 5.11: Results view of ghost when executed parallel

Sample #	Start Time	Thread Name	Label	Sample Time(ms)	Status	Bytes	Latency	Connect Time(ms)
486	13:08:00.457	Thread-Gruppe threa...	77 /ghost/api/v0.1/po...	540	✓	1538	540	1
487	13:07:59.867	Thread-Gruppe threa...	76 /ghost/api/v0.1/po...	1149	✓	943	1149	0
488	13:07:59.930	Thread-Gruppe threa...	77 /ghost/api/v0.1/po...	1106	✓	1542	1105	0
489	13:08:00.964	Thread-Gruppe threa...	107 /	128	✓	2175	128	0
490	13:08:00.104	Thread-Gruppe threa...	77 /ghost/api/v0.1/po...	1026	✓	1542	1026	0
491	13:08:00.545	Thread-Gruppe threa...	78 /ghost/api/v0.1/po...	662	✓	1499	662	0
492	13:08:01.208	Thread-Gruppe threa...	107 /	119	✓	2176	119	0
493	13:08:00.997	Thread-Gruppe threa...	78 /ghost/api/v0.1/po...	379	✗	1589	379	0
494	13:08:00.723	Thread-Gruppe threa...	78 /ghost/api/v0.1/po...	665	✓	1500	665	0
495	13:08:00.990	Thread-Gruppe threa...	77 /ghost/api/v0.1/po...	460	✓	1542	460	1
496	13:08:00.788	Thread-Gruppe threa...	78 /ghost/api/v0.1/po...	704	✓	1500	703	0
497	13:08:00.886	Thread-Gruppe threa...	78 /ghost/api/v0.1/po...	652	✓	1499	652	0
498	13:08:01.376	Thread-Gruppe threa...	107 /	172	✓	2177	172	0
499	13:08:01.388	Thread-Gruppe threa...	107 /	171	✓	2177	171	0
500	13:08:01.036	Thread-Gruppe threa...	78 /ghost/api/v0.1/po...	595	✗	1593	595	0
501	13:08:01.492	Thread-Gruppe threa...	107 /	145	✓	2176	145	0
502	13:08:01.016	Thread-Gruppe threa...	77 /ghost/api/v0.1/po...	662	✓	1541	661	1
503	13:08:01.539	Thread-Gruppe threa...	107 /	149	✓	2176	149	0
504	13:08:01.450	Thread-Gruppe threa...	78 /ghost/api/v0.1/po...	278	✓	1498	278	0
505	13:08:01.631	Thread-Gruppe threa...	107 /	130	✓	2176	130	0
506	13:08:01.130	Thread-Gruppe threa...	78 /ghost/api/v0.1/po...	653	✓	1499	653	0
507	13:08:01.728	Thread-Gruppe threa...	107 /	145	✓	2181	145	0
508	13:08:01.783	Thread-Gruppe threa...	107 /	111	✓	2181	111	0
509	13:08:01.678	Thread-Gruppe threa...	78 /ghost/api/v0.1/po...	226	✓	1499	226	0
510	13:08:01.904	Thread-Gruppe threa...	107 /	32	✓	2182	32	0

Scroll automatically? Child samples? No of Samples 510 Latest Sample 32 Average 786 Deviation 747

Figure 5.12: Report of ghost when executed parallel

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Max	Error %	Throughput	KB/sec
78 /ghost/api/v...	1	961	961	961	961	961	961	961	0,00%	1,0/sec	1,5
78 /ghost/api/v...	1	571	571	571	571	571	571	571	0,00%	1,8/sec	2,6
77 /ghost/api/v...	1	1173	1173	1173	1173	1173	1173	1173	0,00%	51,2/min	1,3
78 /ghost/api/v...	1	879	879	879	879	879	879	879	0,00%	1,1/sec	1,7
77 /ghost/api/v...	1	725	725	725	725	725	725	725	0,00%	1,4/sec	2,1
78 /ghost/api/v...	1	469	469	469	469	469	469	469	0,00%	2,1/sec	3,1
77 /ghost/api/v...	1	399	399	399	399	399	399	399	100,00%	2,5/sec	3,9
77 /ghost/api/v...	1	512	512	512	512	512	512	512	0,00%	2,0/sec	2,9
78 /ghost/api/v...	1	1082	1082	1082	1082	1082	1082	1082	100,00%	55,5/min	1,4
76 /ghost/api/v...	1	940	940	940	940	940	940	940	0,00%	1,1/sec	1,0
77 /ghost/api/v...	1	540	540	540	540	540	540	540	0,00%	1,9/sec	2,8
76 /ghost/api/v...	1	1149	1149	1149	1149	1149	1149	1149	0,00%	52,2/min	,8
77 /ghost/api/v...	1	1106	1106	1106	1106	1106	1106	1106	0,00%	54,2/min	1,4
77 /ghost/api/v...	1	1026	1026	1026	1026	1026	1026	1026	0,00%	58,5/min	1,5
78 /ghost/api/v...	1	662	662	662	662	662	662	662	0,00%	1,5/sec	2,2
78 /ghost/api/v...	1	379	379	379	379	379	379	379	100,00%	2,6/sec	4,1
78 /ghost/api/v...	1	665	665	665	665	665	665	665	0,00%	1,5/sec	2,2
77 /ghost/api/v...	1	460	460	460	460	460	460	460	0,00%	2,2/sec	3,3
78 /ghost/api/v...	1	704	704	704	704	704	704	704	0,00%	1,4/sec	2,1
78 /ghost/api/v...	1	652	652	652	652	652	652	652	0,00%	1,5/sec	2,2
78 /ghost/api/v...	1	595	595	595	595	595	595	595	100,00%	1,7/sec	2,6
77 /ghost/api/v...	1	662	662	662	662	662	662	662	0,00%	1,5/sec	2,3
78 /ghost/api/v...	1	278	278	278	278	278	278	278	0,00%	3,6/sec	5,3
78 /ghost/api/v...	1	653	653	653	653	653	653	653	0,00%	1,5/sec	2,2
78 /ghost/api/v...	1	226	226	226	226	226	226	226	0,00%	4,4/sec	6,5
TOTAL	510	786	480	1894	2435	3121	2	3782	11,18%	25,6/sec	26,5

In Figure 5.11 you can see the last 25 of 510 samples. In Figure 5.12 you see the report of the different requests. The last line of Figure 5.10 shows the summary of all requests.

Following metrics might be interesting:

Table 5.2: Ghost results multi-threaded

metric	value
Throughput	22.6/sec
Average	786ms
Median	480ms
Min	2ms
Max	3782ms
Error	11.18%

As shown in Table 5.2 ghost is not able to handle many requests at the same time and this causes some requests to fail.

Chapter 6

Related Work

There are already some other Node.js benchmarks available. Following benchmarks are related to the benchmarks of this thesis:

1. Acmeair (`https://github.com/acmeair/acmeair-nodejs`) is an existing Node.js benchmark. It tests the performance of a big Node.js application. The applications in this thesis are smaller but they cover more use cases such as I/O intensive and CPU intensive applications.

2. Node.js Core benchmarks → these micro benchmarks cover only some core runtime performance tests, such as testing the performance of event emitters. The benchmarks in this thesis cover more than only testing some core runtime characteristics because the benchmarked applications use the most popular packages and behave like real world applications.

Chapter 7

Future Work

This thesis showed how to benchmark Node.js applications but there might be some improvements possible.

Firstly, the sample applications could be extended to support the Node.js cluster module. This module enables parallel request handling. Of course the multi-threaded benchmarks would then be faster but the benchmarks would then also cover the cluster module which is essential for many productive Node.js applications.

The results in Chapter 5 cover only the official Node.js runtime. In order to compare these results the benchmarks have to be executed in different JavaScript engines with the same test setup. A possible runtime would be *Graal.js* which is part of the *Graal* framework[12].

Chapter 8

Conclusion

This thesis showed how to benchmark typical Node.js applications. First of all, the currently most popular packages were evaluated by analyzing the npm registry. Then some of these packages which are considered as relevant for performance testing were used in three different sample applications. These applications should represent typical use cases such as I/O intensive applications, CPU intensive applications and modern rich web applications. Then the previous developed applications were benchmarked with the help of an existing workload generator called Apache JMeter. The results were discussed in detail and the bottlenecks were identified by analyzing the results. Finally, possible future improvements were presented.

List of Figures

List of Tables

Bibliography

[1] Apache jmeter. http://jmeter.apache.org/. Accessed: 2016-10-12.

[2] colors package. https://www.npmjs.com/package/colors/. Accessed: 2016-10-12.

[3] commander package. https://www.npmjs.com/package/commander/. Accessed: 2016-10-12.

[4] crypto package. https://www.npmjs.com/package/crypto/. Accessed: 2016-10-12.

[5] fs-extra package. https://www.npmjs.com/package/fs-extra/. Accessed: 2016-10-12.

[6] fstream package. https://www.npmjs.com/package/fstream/. Accessed: 2016-10-12.

[7] Ghost as module. https://github.com/TryGhost/Ghost/wiki/Using-Ghost-as-an-npm-module/. Accessed: 2016-10-12.

[8] Ghost cluster example. http://blog.fabianbecker.eu/scale-ghost-to-the-next-level/. Accessed: 2016-10-12.

[9] Ghost github repository. https://github.com/TryGhost/Ghost/. Accessed: 2016-10-12.

[10] Ghost homepage. https://ghost.org/. Accessed: 2016-10-12.

[11] ghost package. https://www.npmjs.com/package/ghost/. Accessed: 2016-10-12.

[12] Graal overview. http://www.oracle.com/technetwork/oracle-labs/program-languages/overview/index-2301583.html. Accessed: 2016-10-15.

[13] mkdirp package. https://www.npmjs.com/package/mkdirp/. Accessed: 2016-10-12.

[14] ncp package. https://www.npmjs.com/package/ncp/. Accessed: 2016-10-12.

[15] Node.js about. https://nodejs.org/en/about/. Accessed: 2016-10-12.

[16] Node.js child process api. https://nodejs.org/api/child_process.html#child_process_child_process_fork_modulepath_args_options/. Accessed: 2016-10-12.

[17] Node.js cluster module. https://nodejs.org/api/cluster.html. Accessed: 2016-10-12.

[18] Node.js homepage. https://nodejs.org/en/. Accessed: 2016-10-12.

[19] Npm most depended-upon packages. https://www.npmjs.com/browse/depended/. Accessed: 2016-10-12.

[20] Npm most starred packages. https://www.npmjs.com/browse/star/. Accessed: 2016-10-12.

[21] path package. https://www.npmjs.com/package/path/. Accessed: 2016-10-12.

[22] tar package. https://www.npmjs.com/package/tar/. Accessed: 2016-10-12.

[23] verror package. https://www.npmjs.com/package/verror/. Accessed: 2016-10-12.

[24] What is npm. https://docs.npmjs.com/getting-started/what-is-npm/. Accessed: 2016-10-12.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Bachelorarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Linz, am 15. Oktober 2016

Christopher Warmbold