

Bachelor Project



**Czech
Technical
University
in Prague**

F3

**Faculty of Electrical Engineering
Department of Computer Science**

Distributed computations on RaspberryPi cluster

Roman Janků

**Supervisor: doc. ing. Jiří Vokřínek, Ph.D.
May 2022**

I. Personal and study details

Student's name: **Jank Roman**

Personal ID number: **495658**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Science**

Study program: **Software Engineering and Technology**

II. Bachelor's thesis details

Bachelor's thesis title in English:

Distributed computations on RaspberryPi cluster

Bachelor's thesis title in Czech:

Distribuované výpočty na RaspberryPi klastru

Guidelines:

Implement an experimental platform using RaspberryPi computational cluster for evaluation/validation of distributed algorithms.

1. Study the RaspberryPi platform and applicable technologies for cluster implementation
2. Analyze and propose SW pipeline to implement and deploy distributed algorithms to RaspberryPi cluster
3. Implement server and client applications for managing and executing distributed computational tasks
4. Implement selected distributed algorithms to evaluate the functionality of the SW pipeline
5. Evaluate the implemented system using experiments on implemented algorithms, provide a scientific-like evaluation of the selected distributed computation algorithms
6. Use proper SW engineering methodology and SW testing, i.e. unit tests and integration tests of the SW pipeline
7. Provide an quick-user guide to enable easy reuse and experimentation with the platform

Bibliography / sources:

- [1] Ansible documentation. [online]. Ansible project contributors. Available at: <https://docs.ansible.com/ansible/latest/>
[2] Docker documentation. [online]. Docker Inc. Available at: <https://docs.docker.com/>
[3] Java SE 11 & JDK 11 documentation. [online]. Oracle. Available at <https://docs.oracle.com/en/java/javase/11/docs/api/index.html>

Name and workplace of bachelor's thesis supervisor:

doc. Ing. Jiří Vokřínek, Ph.D. Department of Computer Science FEE

Name and workplace of second bachelor's thesis supervisor or consultant:

Date of bachelor's thesis assignment: **02.02.2022** Deadline for bachelor thesis submission: **20.05.2022**

Assignment valid until: **30.09.2023**

doc. Ing. Jiří Vokřínek, Ph.D.
Supervisor's signature

Head of department's signature

prof. Mgr. Petr Páta, Ph.D.
Dean's signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

Date of assignment receipt

Student's signature

Acknowledgements

Firstly, I would like to thank Mr. Jiří Vokřínek for giving me this topic of bachelor thesis and supervising me while I was working on it. His critique and advices were irreplaceable.

Secondly, I would like to thank my classmates for giving me advice and criticising my work. Their comments allowed me to improve the thesis.

Finally, I would like to thank my family and friends for supporting me and having patience with me working on the thesis most of the time.

Declaration

I declare that this work is all my own work and I have cited all sources I have used in the bibliography.

Prague, May 14, 2022

Abstract

This bachelor thesis deals with the development of a platform for implementing and measuring distributed algorithms on RaspberryPi cluster. Client-server application was developed in Java programming language. This application enables users to measure performance of distributed algorithms that were already prepared or were custom made on RaspberryPi cluster. This application is distributed and computers in cluster are being prepared by Ansible. Client-side of application runs in virtual environment of Docker platform. Subsequent measurements showed good results the platform reached when running distributed algorithms. The result of the work is finished platform that will enable users to implement and measure distributed algorithms.

Keywords: cluster, PicoCluster, distributed algorithms, distributed computations, RaspberryPi, Ansible, Docker, Java

Supervisor:

doc. ing. Jiří Vokřínek, Ph.D.
Department of Computer Science,
Karlovo náměstí 13,
120 00 Praha 2

Abstrakt

Tato bakalářská práce se zabývá vývojem platformy pro vývoj a měření distribuovaných algoritmů na clusteru RaspberryPi. V programovacím jazyce Java byla vyvíjena client-server aplikace, která umožňuje měření jak již naprogramovaných tak i vlastních distribuovaných algoritmů na clusteru RaspberryPi. Tato aplikace byla distribuována a jednotlivé počítače v clusteru byly připraveny pomocí nástroje Ansible. Klientská část aplikace běží ve virtuálním prostředí platformy Docker. Následným měřením bylo zjištěno, že platforma dosahuje velmi dobrých výsledků při běhu distribuovaných algoritmů. Výsledkem práce je hotová platforma, která umožní uživatelům vývoj a měření distribuovaných algoritmů.


Klíčová slova: cluster, PicoCluster, distribuované algoritmy, distribuované výpočty, RaspberryPi, Ansible, Docker, Java

Contents

1 Introduction	1	Bibliography	47
1.1 Assignment	2	A How to use platform	49
1.2 Requirements	2	B Attachments	55
2 RaspberryPi platform	5		
2.1 Hardware	6		
2.2 Software	6		
3 Proposal of solution	9		
4 Software pipeline for development and deployment	11		
4.1 Comparison of build tools	11		
4.2 Maven configuration	12		
4.3 Comparison of deployment tools	13		
4.4 Ansible configuration	14		
4.5 Comparison of runtime environments	15		
4.6 Docker configuration	16		
5 Client and server applications	19		
5.1 External libraries	19		
5.2 Shared library	20		
5.3 Server application	21		
5.3.1 Clients	22		
5.3.2 Message handling	22		
5.3.3 Tasks	22		
5.3.4 Distibuted data	23		
5.4 Client application	23		
5.4.1 Blinkt! module	24		
5.5 Communication	25		
5.5.1 Main client-server communication	25		
5.5.2 Client-server data download .	27		
5.5.3 Communication between clients	28		
6 Distributed algorithms	29		
6.1 Implementing algorithms	29		
6.2 Implemented algorithms	31		
7 Experiments with distributed algorithms	35		
8 Software testing	41		
9 Conclusion	45		
9.1 Future improvements	45		

Figures

2.1 Pico 5 cluster	5
3.1 Scheme of network	9
5.1 Class diagram of the server	21
5.2 Picture of Blinkt! module	24
7.1 Comparison of reliability with and without notifications depending on delay between messages	36
7.2 Comparison of response time with and without notifications depending on delay between messages	37
7.3 Comparison of two response curves depending on delay between messages	38
7.4 Comparison of two delivery rates depending on the delay between messages	38
8.1 Pico 48 cluster running Blinker task	43
A.1 Graphical user interface of server application	52
A.2 Dialog for adding new task	53



Chapter 1

Introduction

The objective of this bachelor thesis is to develop a platform that will enable users to implement their custom distributed algorithms and evaluate their properties. The platform should be easy to use even for people without much experience in programming.

At first, a proper study of the platform should be carried out to find all possibilities and limitations it has. After that, a solution will be proposed that will be then developed into a full platform that will cover all the functionality given by the requirements.

All applications should be developed in such a way that they would be easy to modify. This would make the platform versatile and easy to adapt for use in many environments. The implementation of user algorithms should also be very straightforward without unnecessary complications and repeated code.

The platform should use automation as much as possible. It is expected that the platform will use much more nodes than the five RaspberryPis that were used for development. In addition, these nodes can be placed at different locations with no physical access to them making their orchestration more difficult. Also, it should be able to run on variety of target machines without extensive modifications.

Proper testing of the platform should also be done to find and fix as many bugs as possible. Variety of testing methods should be used to test all the aspects of the platform ranging from unit test to full user testing of the platform.

Measurements should also be done to characterise the performance of important parts of the platform. These measurements should focus on the network as it will probably be the bottleneck of the whole platform.

Finally, as the platform will probably be modified and extended by future users it must be well documented. The documentation should cover everything from source code thru theory of operation and development and deployment pipeline to graphical user interface.

Bellow is the assignment of the bachelor thesis and requirements gathered throughout the development of this platform.

1.1 Assignment

Implement an experimental platform using RaspberryPi computational cluster for evaluation/validation of distributed algorithms.

1. Study the RaspberryPi platform and applicable technologies for cluster implementation
2. Analyze and propose SW pipeline to implement and deploy distributed algorithms to RaspberryPi cluster
3. Implement server and client applications for managing and executing distributed computational tasks
4. Implement selected distributed algorithms to evaluate the functionality of the SW pipeline
5. Evaluate the implemented system using experiments on implemented algorithms, provide a scientific-like evaluation of the selected distributed computation algorithms
6. Use proper SW engineering methodology and SW testing, i.e. unit tests and integration tests of the SW pipeline
7. Provide an quick-user guide to enable easy reuse and experimentation with the platform

1.2 Requirements

Several requirements were specified at the beginning of work on this project. However, most of them were added during development after consultations with supervisor.

Functional requirements of the system are:

- User can run a task with parameters and on selected number of nodes.
- Platform can run user-created algorithms.
- Tasks are placed in queue and started when possible.
- Tasks results can be exported in a standardized format.
- User can put nodes into suspended state in which the node will not run tasks.
- Algorithms can download additional data from server.
- Algorithms can communicate between nodes running the same task using UDP.
- Nodes can notify server that message had been sent.

- User can specify maximum size of UDP packet.
- Algorithms can send status information to the server.
- State of node is displayed on the Blinkt! module.
- Algorithms can use Blinkt! module for its own purposes.
- Deployment process should be automated and easily scalable.

Non-functional requirements are:

- The platform should be well documented.
- User guide should be created. It should cover the implementation of custom algorithms as well.
- Platform should support distributed algorithms written in Java.

Chapter 2

RaspberryPi platform

Pico 5 cluster¹ with five RaspberryPi 4B+ boards² was used. Apart from five RaspberryPi boards it contains power supply and gigabit ethernet switch. All this is enclosed in an acrylic cube with selected connections wired to sides of the cube. On the figure Figure 2.1 can be seen the completed cluster.

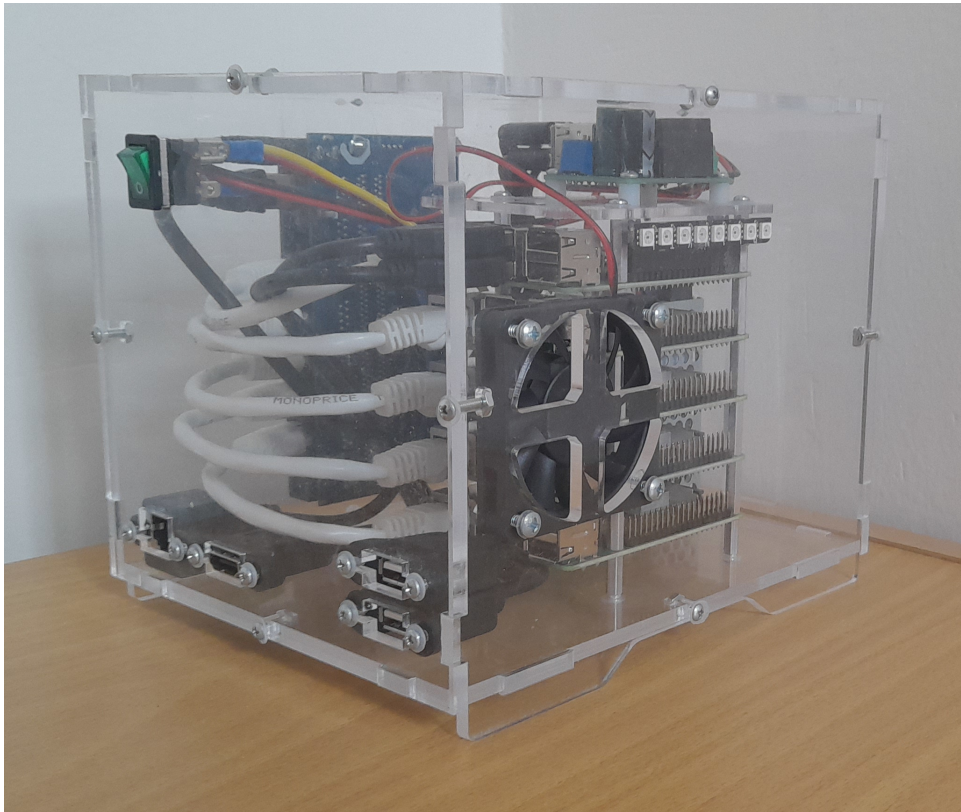


Figure 2.1: Pico 5 cluster

Apart from PicoCluster with five RaspberryPis a PicoCluster with forty-eight RaspberryPis is available. The cluster with forty-eight RaspberryPis

¹<https://www.picocluster.com/collections/pico-5>

²<https://www.raspberrypi.com/products/raspberry-pi-4-model-b/>

will be used later to test how the platform scales for bigger number of nodes. For development the version with five RaspberryPis was used. These two versions are exactly the same from hardware as well as software perspective. The only difference is the number of RaspberryPis.

Let's describe the hardware and software of the Pico 5 cluster in a detail.

2.1 Hardware

RaspberryPi 4B+ boards are used in our cluster. These boards have quad-core 64-bit ARM v7 processor with 8GB RAM. As a permanent storage a microSD card is used with capacity of 32GB.

Each RaspberryPi also has four USB 2.0 ports, one HDMI port, one gigabit Ethernet RJ-45 port, 3.5 mm jack for audio and composite video and USB-C for power. However, only two USB ports and HDMI port of the top most RaspberryPi are wired to the side of the cube. The Ethernet ports are connected to the internal switch and one connection from this switch is wired to the side of the cube.

Power to the cube is supplied by a standard IEC connector for 240 V. It is then converted by internal power supplies to voltages that the RaspberryPis and switch use. The power can be switched on and off by switch placed on the side of the cube.

Each RaspberryPi has a Blinkt!³ module with eight super-bright RGB LEDs connected to its GPIO header. It can be used for many purposes but in this work it was mostly used for state information of the node. However, some tasks utilize the Blinkt! module for displaying different colors and patterns on the LEDs.

Although it can be controlled directly by driving the pins 23 and 24 on the GPIO header that work as an I²C bus with data on the pin 23 and clock on the pin 24, library was used to make the controlling more convenient. It takes care of the timing, communication protocol and addressing of the LEDs making it easy to use the module. The used libraries and functionality of the module are described in detail in the section 5.1.

2.2 Software

RaspberryPi boards come with Raspbian 9⁴ preinstalled on their microSD cards. Raspbian is a modified version of Debian GNU/Linux operating system⁵ created specially for RaspberryPi boards. The main features of this operating systems are small size, low system requirements and drivers for RaspberryPi GPIO header and other specific hardware.

Also, basic tools come preinstalled as on standard Debian. Most important are `ssh` used for connecting to the boards, `python3` used for running

³<https://shop.pimoroni.com/products/blinkt?variant=22408658695>

⁴<https://raspbian.org/>

⁵<https://www.debian.org/>

scripts on the boards and `sudo` for running programs with elevated privileges. Default user is named `picocluster` and has the same password. This user is added in the `sudoers` group eliminating any need to use `root` user. The `picocluster` user can also login remotely using the `ssh`.

Finally, RaspberryPis come preconfigured for network `10.1.10.0/24`. The topmost RaspberryPi has IP address `10.1.10.240`, the second from the top has `10.1.10.241` all the way to the most bottom one that has `10.1.10.244`. Default gateway is also configured on all RaspberryPis with the value `10.1.10.1`.

Backup of microSD cards from all RaspberryPis was made to ease restoration of RaspberryPis after the experiment is done or in case anything goes wrong. To make backup each microSD card was connected to a machine running Debian operating system and `dd`⁶ program was used to copy the microSD card as whole to the `*.img` file using the following command

```
dd if=/dev/sdj of=~/card0.img bs=10M
```

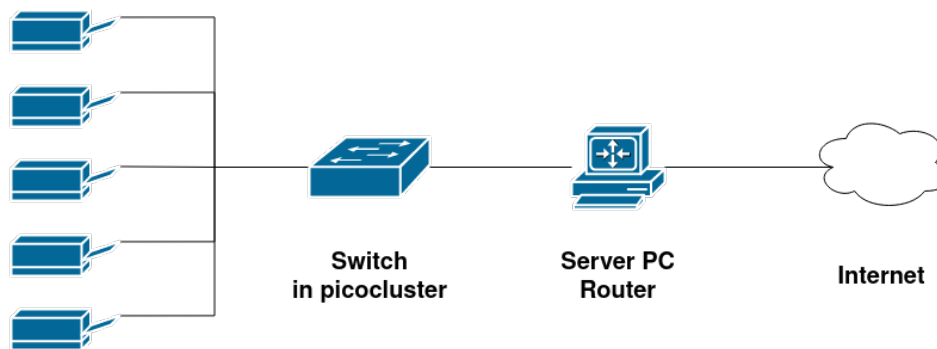
where parameter `if` specifies the input file. In our case it's device that the microSD card is connected to. The `of` parameter specifies the output file to which the microSD card was copied. The last parameter `bs` specifies size of data that is copied each time and causes the `dd` to work faster. If needed, the microSD cards can be restored by running the `dd` command with switched `if` and `of` parameters to copy the image to the microSD card.

⁶<https://manpages.debian.org/stretch/coreutils/dd.1.en.html>

Chapter 3

Proposal of solution

After considering several possibilities on how to use the RaspberryPi cluster the following option shown on Figure 3.1 was chosen.



RaspberryPis

Figure 3.1: Scheme of network

The application will use the server-client architecture with server being outside the PicoCluster. The PicoCluster will be connected using Ethernet to a PC with two Ethernet cards. This PC will act as a server for the cluster as well as an router to provide an internet connection for nodes.

This architecture is easily scalable for larger amount of nodes. Also, reconfiguration of the RaspberryPis will not be needed as they are configured to use static IP addresses and have a gateway configured. Also, thanks to this, no DHCP server will be required.

The server and client applications should be developed in Java. Firstly, it has a good functionality for networking and parallel computing. In addition, a large number of available libraries significantly speed up the development by eliminating boilerplate code. Moreover, the applications developed in Java will make implementation of user algorithms easier. Finally, it can run on variety of operating systems making the applications usable on variety of machines.

For deployment at least a partially automated pipeline would be needed. This will make the build and deploy process easily scalable and not that labour

intensive. At the best, the tool should be able to build the application, prepare target machines and deploy the application to the target machines fully automatically. Alternatively, the build part can be separated without significantly complicating the deployment procedure.

Testing should be also automated or at least its part that can be done automatically. The tests should be done during build or even earlier to find bugs as soon as possible to minimise the costs and time it takes to fix the bugs. Also, preventive measures should be taken to eliminate bugs in the first place.

Finally, the node application should be running in a virtual environment. This would hide the differences between different machines the application will run on and provide unified runtime environment. Also, it will eliminate collisions with other software running on the machine. Lastly, it would make it easier to start and stop the application. Virtualization probably would not be needed for server application.

Chapter 4

Software pipeline for development and deployment

At least partially automated pipeline for development and deployment was needed for this project because of its size. Now, we will go thru different technologies for development, deployment and runtime. We will compare them, select the most suitable one and we will describe how it is used in this project.

For versioning a git¹ was used. Czech Technical University has its own GitLab² which is used for all school projects and it was used for this one as well. All the source code is available at public repository³ at this GitLab. There is a main branch named `master`. All feature development is done in separate branches that are then merged into the main branch after they are fully working. After merging the branch is deleted.

4.1 Comparison of build tools

- *Maven*⁴ is an open-source tool for project management. It is mainly used for managing project dependencies and running tests and builds. Although Maven can be used for variety of programming languages it was made primarily for Java.

Configuration is done in `pom.xml` files in XML format. Being designed primarily for Java a little setting is required to make the Maven generate `jar` archives. In addition, a lot of libraries and plugins are available from official Maven repository and many more can be loaded from other repositories including GitHub⁵. Finally, it can handle multi-project builds.

¹<https://git-scm.com/>

²<https://gitlab.fel.cvut.cz/>

³<https://gitlab.fel.cvut.cz/jankurom/semestralni-projekt-a-bakalarska-prace>

⁴<https://maven.apache.org/>

⁵<https://github.com/>

- *Gradle*⁶ is very similar to the Maven. It is also an open source tool available under Apache License 2.0. It primarily focuses on Java, Groovy and Scala development. However, there are some differences.

The main difference is the use of its own domain-specific language for configuration files. Other than that it supports multi-project builds and can use the same repositories as Maven. Finally, it has a different system for handling tasks that are building the final application than Maven.

Maven and Gradle are very similar and both would be a good choice for this project. In the end, Maven was selected because it is the first choice for use in Java projects and it was used previously.

4.2 Maven configuration

The online Maven documentation^[7] was used when writing the scripts for build. Project has four Maven modules `library`, `node`, `server` and `root`. The `root` module⁷ only defines Java version, encoding and three child modules. Each child module has its own directory named the same as the module.

The child modules add dependencies to the libraries used in source codes. These libraries are listed in section 5.1. The `node` modules also add additional repositories for libraries that are not available in the standard Maven repository. These include JitPack⁸ and Sonatype⁹. Libraries that enable Java programs to use GPIO header of RaspberryPi are hosted in these repositories.

The `node` and `server` modules use `maven-assembly-plugin`¹⁰ that is used to build fat `jar` that includes all dependencies and compiled source code in one `.jar` archive. This makes it easy to distribute the executable and eliminates the need to distribute the dependencies separately and include them in class path. In addition, main classes with `main()` methods are added to manifest using this plugin to make the `.jar` archives runnable.

The `library` module is not being build as an executable `jar` and is included in the `node` and `server` `.jar` archives for simpler distribution.

Maven scripts are run using the `package` target which will recompile of the source code, run the JUnit tests and package the compiled `.class` files with all dependencies in `.jar` archive. This can be done from IDE or by running command

```
mvn package
```

in the `root` module. The finished `.jar` archives are placed in `target` directories in `library`, `node` and `server` modules.

⁶<https://gradle.org/>

⁷<https://gitlab.fel.cvut.cz/jankurom/semestralni-projekt-a-bakalarska-prace/-/blob/master/picocluster-distributed-algorithm/pom.xml>

⁸<https://jitpack.io/>

⁹<https://oss.sonatype.org/>

¹⁰<https://maven.apache.org/plugins/maven-assembly-plugin/>

4.3 Comparison of deployment tools

- *Manual deployment* means that the developer will copy the required files to the target machines, build them, prepare the environment and run them.

This method is very laborious and does not scale easily as twice the number of nodes to manage means twice as work. It is also very prone to mistakes. However, it has no requirements for the target machines that can be very different and it also does not require installation of any additional tools.

- *Ansible*¹¹ is a tool for managing, configuring and deploying software on target machines.

It is easily configurable and can perform a variety of tasks on the target machines. It does not require installation of any additional software apart from Python 2.4 or later on Linux target machines and PowerShell 3.0 or later on Windows target machines. Of course, Ansible itself must be present on the controlling machine.

Number of deployment scripts can be created and called separately by changing parameter in `ansible-playbook` command. The deployment runs in parallel on the target machines so it is easily scalable on large number of nodes.

However, it requires an UNIX operating system to run. On Windows it can be used in Windows Subsystem for Linux¹². In addition, it requires the target machines to have the same operating system installed as it will configure the `apt` tool and this configuration is dependent on the operating system. Also, all the machines must have the same user with same password and `sudo` privileges as the Ansible uses `ssh` to connect to the target machine and run commands on it.

- *GitLab CI/CD*¹³ is tool for continuous integration and continuous deployment that is integrated in GitLab. Action can be hooked to events happening in the repository. For example, push will trigger tests, merge will trigger build and deploy to test environment, etc.

All this would shorten the deployment time considerably and is available at the GitHub repository used for this project. However, it would require the RaspberryPis to be accessible from the internet as the GitLab server needs access to them to deploy the application to them.

The manual deployment was rejected right at the beginning as it would be too laborious even with five nodes and would scale terribly for more nodes. It is also very prone to mistakes because of the human element.

¹¹<https://www.ansible.com/>

¹²<https://docs.microsoft.com/en-us/windows/wsl/>

¹³<https://docs.gitlab.com/ee/ci/>

The GitLab CI/CD would make the deployment fully automated as it would be hooked on actions in git repository. These actions are done as part of the development pipeline and would be done even without the Gitlab CI/CD. However, opening the RaspberryPis to the internet would not only require extensive network configuration but also would be a huge security risk. Therefore it was rejected.

In the end, Ansible was selected. Unlike GitLab CI/CD it can run from local machine and controll machines in local network. Because of this it eliminates all the work and risks associated with opening the RaspberryPis to the internet. Moreover, several scripts for different purposes can be created. This method would require some user input when launching the script but after that it would be fully automated. Finally, it would easily scale to a larger number of nodes.

4.4 Ansible configuration

The online Ansible documentation[1] and DigitalOcean tutorial[2] was used for configuring the Ansible. The Ansible handles the whole process of deployment of node application including preparing the runtime, installing dependencies and creating the Docker container.

The target machines are listed in `hosts` file¹⁴. It is in `ini` format with three groups. The `localhost` group was used for testing, the `cluster` group contains list of IP addresses of the RaspberryPis in Pico 5 cluster and the `big` group contains list of IP addresses of nodes in the Pico 48 cluster.

There are eight tasks that cover the whole deployment process. The first task *Install dependencies* installs all dependencies that will be needed thru the deployment process. The `apt` tool is used for installation and it loops over a list of packages that will be needed later.

Second task *Add Docker GPG key* adds key for the official Docker repository. Again, `apt` tool is used to add the key to the trusted ones. Third task *Add Docker repository* adds URL of the official Docker repository. `apt` is also used for this task. Fourth task *Install Docker* finally installs Docker using the `apt` tool. It loops over three packages that need to be installed.

Fifth task *Install Docker for Python* install python libraries that are needed for Ansible to be able to control Docker.

Sixth task *Delete old content & directory* deletes all old files in `/srv/` directory. Seventh task *Copy Docker and program files* copies source files to the target machine. They are copied to the directory `/srv/`. Eighth task *Build Docker image* creates a Docker image from the files that have been copied. The ninth and last task *Run Docker image* starts the previously created Docker image. It is started with elevated privilege to have access to GPIO header and with `pass thru` on UDP port 12347.

The Ansible script is run using the following command

¹⁴<https://gitlab.fel.cvut.cz/jankurom/semestralni-projekt-a-bakalarska-prace/-/blob/master/picocluster-distributed-algorithm/hosts>

```
ansible-playbook -i ./hosts -kK --ssh-extra-args='-o "
  ↪ PubkeyAuthentication=no"' -u picocluster deploy.yml
```

`hosts` file contains list of target machines, `-u` specifies the name of the user to be used for `ssh` connection and parameter `-ssh-extra-args` tells `ssh` to use password and not the key. The Ansible will then ask for password that will be used to connect to the target machines.

Several scripts were derived from the main Ansible script. These scripts are used when some tasks can be omitted. The `fast-deployed.yml` script skips the preparation of target machine and starts with deleting the content of `/srv/` directory and copying source files. It is used when the target machine was prepared earlier and new version of software needs to be installed.

The `restart-docker.yml` script only restarts the Docker container running on the target machine. Last script `stop-docker.yml` is used to stop the currently running Docker container on the target machines.

All these scripts are run the same way as the main script.

4.5 Comparison of runtime environments

- *Running directly* does not require any additional software on the target machine and running application has access to all hardware on the machine. However, there can be conflicts with other programs and it can cause security problems. Also, it is not easy to start, stop and restart the application.
- *Docker*¹⁵ is an open-source tool that provides operating system-level virtualization. It creates an isolated container for the application containing only the application and required files. It does not fully virtualize the operating system and uses the host operating system instead. Therefore it consumes less resources of the host system.

It provides a unified environment for the application that is not dependent on the host machine. Therefore specific programs, libraries and versions of them can be installed without causing any collisions. This also provides a security layer. If the application in the container fails, only the container fails and not the host operating system. The container can be restarted after that and it does not affect any other programs or containers on the host system.

In addition, the application can access the host hardware directly from container. This includes the GPIO header of the RaspberryPi, TCP or UDP ports and others. This must be specified in the Docker file and drivers for the hardware must be installed.

However, Docker requires Linux operating system to run. On Windows machines a Windows Subsystem for Linux can be used to run Docker.

¹⁵<https://www.docker.com/>

- *VirtualBox*¹⁶ is a tool for virtualization supporting variety of operating systems. It offers a full virtualization and isolation of the application running in the virtual machine. Of course, due to the full virtualization VirtualBox puts a huge load on the system which can be a problem on device with limited power like RaspberryPi.

However, due to the complete virtualization the access to the hardware is very limited and the applications running in the virtual machines would not have access to the GPIO header of RaspberryPis. Moreover, VirtualBox does not offer a good command-line tool for remote configuration of virtual machines and use of GUI would be required.

- *QEMU*¹⁷ is an emulator and virtualizer that uses a hypervisor to emulate either CPU or the whole system. It would isolate the application in a virtual environment similarly to the VirtualBox. However, the virtualization is simpler. It would provide isolation from other applications and would avoid collision between programs and libraries but would also cut off access to the RaspberryPi GPIO header. Finally, it would require installation of full operating system to run the application.

Running the application directly on the RaspberryPi was rejected because it would make the application hard to control. Either services or another method would have to be used to start, stop and restart the application. This would make it hard to use.

VirtualBox was also rejected because it cannot be easily controlled by the Ansible that was selected as a deployment tool. Moreover, the application would not have access to the GPIO header to use it with Blinkt! module and the virtualization used by VirtualBox is resource-intensive that could cause problems on RaspberryPi. Qemu was rejected for similar reasons as VirtualBox although the virtualization is not that resource-intensive.

Docker was selected as the runtime environment for the node application for several reasons. It can provide access to the host hardware making the Blinkt! module connected to the RaspberryPi GPIO header usable for status information. Docker is also low resource-intensive making it ideal for use on RaspberryPi that has limited resources and computational power.

Moreover, running the application in Docker container makes it easier to start, stop and restart, unlike when running directly on the target machine. Finally, it provides security meaning faulty application will not cause crash of the host as well as collisions with other applications are avoided.

4.6 Docker configuration

The online Docker documentation[3] was used for configuring the Docker. All the docker configuration is in the `Dockerfile` file¹⁸.

¹⁶<https://www.virtualbox.org/>

¹⁷<https://www.qemu.org/>

¹⁸<https://gitlab.fel.cvut.cz/jankurom/semestralni-projekt-a-bakalarska-prace/-/blob/master/picocluster-distributed-algorithm/Dockerfile>

The Docker image is created from Debian image Buster version¹⁹. The *slim* variant is used as it has removed unnecessary software. Then the path is switched to the `/app/` directory, which will be used for compilation of the application.

After that, dependencies that will be needed are installed. This includes `maven`, `openjdk-11-jdk`, `sudo`, `make`, `gcc` and `git`. Then a WiringPi²⁰ library from `git` is downloaded and built. This library is needed for the Java application to be able to work with RaspberryPi GPIO header.

After that, all the needed source files are copied to the image and then are built using `mvn package` command. Environment variables with address of the server and maximum UDP packet size are created and after that the entry point is set.

The docker image is started by the Ansible as a part of the deployment process. This is described above. To access logs of the application a following command can be used

```
docker logs -f <CONTAINER_ID>
```

where the `container_id` is unique identification of the running container. It can be obtained using the command

```
docker container ls
```

¹⁹<https://hub.docker.com/layers/debian/library/debian/buster-slim/images/sha256-79c373b3a8b1543cb91ff909eec8c5b6b2751d8ed1806063aba4bea77b83799e?context=explore>

²⁰<https://github.com/WiringPi/WiringPi.git>

Chapter 5

Client and server applications

Several applications were developed during the work on this assignment. Two applications were developed very early and were used to test and create the pipeline for development. They are available at the repository and are named *Simple server*¹ and *Simple server tester*².

Then the main platform for distributed algorithms was developed. The source code is written in Java programming language and *JavaDoc*[4] was used as documentation for writing the applications apart from a few instances where other sources were used. These sources are mentioned where they were used.

Whole source code was commented using JavaDoc comments³ and therefore a `javadoc` program from java binaries can be used to generate documentation of source code in `html` format.

5.1 External libraries

Several external libraries were used for specific tasks. Thanks to these libraries focus could have been shifted from writing boilerplate code to the actual problem that was being solved.

- `reflections`⁴ is library available at the central Maven repository⁵. It is used to encapsulate reflections in Java that are being used for loading algorithms and message handlers. It is available under Apache 2.0 license.
- `org.junit`⁶ is a group of libraries used for unit testing of Java code. It is available under Eclipse Public License v1.0 at the central Maven repository.

¹<https://gitlab.fel.cvut.cz/jankurom/semestralni-projekt-a-bakalarska-prace/-/tree/master/simple-server>

²<https://gitlab.fel.cvut.cz/jankurom/semestralni-projekt-a-bakalarska-prace/-/tree/master/simple-server-tester>

³<https://www.oracle.com/cz/technical-resources/articles/java/javadoc-tool.html>

⁴<https://mvnrepository.com/artifact/org.reflections/reflections>

⁵<https://mvnrepository.com/>

⁶<https://junit.org/junit5/>

of structured data for start messages, information about nodes and also classes defining messages used for downloading data from server by the algorithm.

■ exceptions

Exceptions that are in this package are used directly by the code that is in the library. Any other exceptions that are used only on server or node are in packages in the server or node application respectively.

■ 5.3 Server application

The server application is the center of the platform for distributed calculations. All the nodes connect to it and it controls them and computations running on them.

The class structure of the server application is shown in the figure Figure 5.1. Exceptions, GUI classes, selected utility classes and less important methods were omitted to improve readability of this diagram.

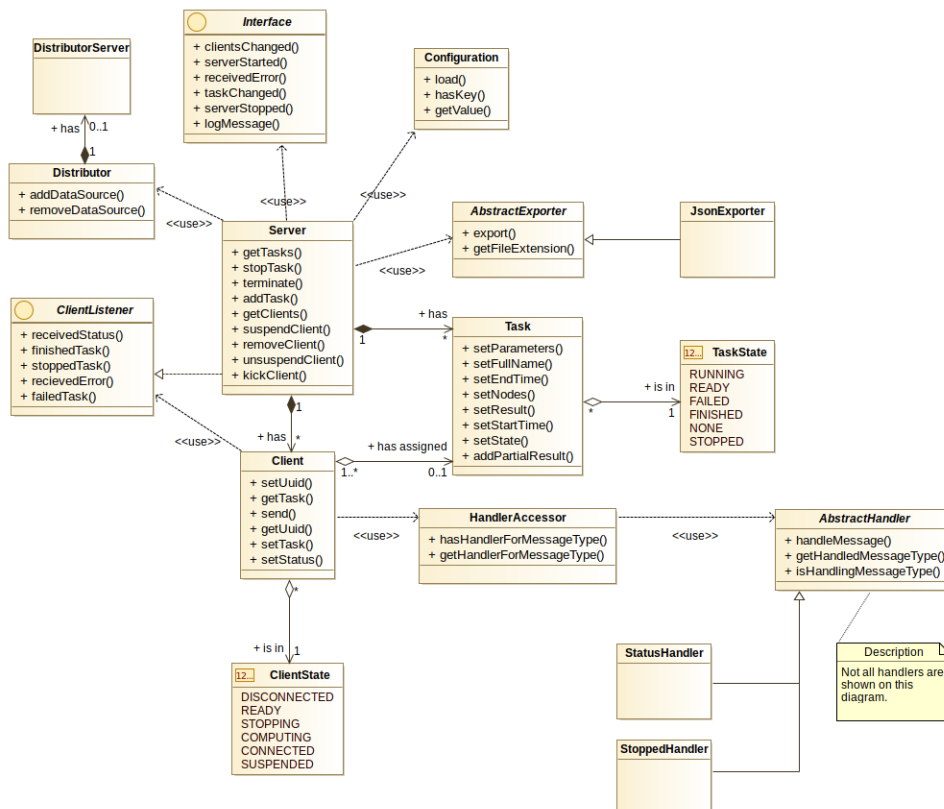


Figure 5.1: Class diagram of the server

■ 5.3.1 Clients

The center of the server application is **Server** class. When started, this class listens on specified TCP port for incoming connections from clients. It also contains a list of tasks that are being run on the cluster with all their information. Listening for incoming connections is done in a separate thread in order not to block the main program.

When a new client connects all information is passed to a new instance of **Client** class. This class handles connection to the client by opening sockets in both directions. As in the **Server** class, listening on incoming socket is done in separate thread in order not to block rest of the application.

Client can have several states. When it has just connected it is in **CONNECTED** state. In this state it cannot do any distributed computations. After it has its **UUID** assigned and all other nodes are notified of its existence it changes its state to **READY**. In this state it is prepared to do distributed calculations.

When it has task received it changes its state to **COMPUTING**. After it finishes either successfully or not it changes its state back to the **READY** state and can receive another task. If it is being stopped it changes its state to **STOPPING** and after the node confirms it had stopped it removes itself from server.

User can put any ready client in **SUSPENDED** state. Node in this state cannot receive task to compute and can be changed back to **READY** state by the user.

The Baeldung tutorial[5] was used as a starting point when writing this part of server application.

■ 5.3.2 Message handling

When a message from client is received it is parsed as an **Message** object. This message is then passed to appropriate handler. Handlers are being found by the **HandlersAccessor** class that uses reflection to load all available handlers.

When the appropriate handler is not found an exception is thrown. This exception is caught and error message is sent back. On server this message is ignored and nothing happens. All handlers implement the abstract class **AbstractHandler**.

■ 5.3.3 Tasks

When a new task is added it is put in the end of the queue. Server then checks at certain events whether it can start the next task. These events include new client connected, node finishing or failing task, node stopping or task starting. If there are enough nodes available to run the task it is started otherwise it waits.

When a new task is being added to the queue the server checks if it has enough nodes connected to run the task. If there are fewer nodes than required the task fails without being started.

When the task is in the queue waiting to be started it is in **READY** state. When it is in the front of the queue and there are enough ready nodes to run it starts and changes its state to **RUNNING**. After finishing it changes its state to **FINISHED**.

If even one node fails the whole task fails and task changes its state to **FAILED**. If task failed the result of the task contains description of the reason it failed. Tasks can also be stopped by user which changes their state to **STOPPED**.

5.3.4 Distibuted data

Port for distributing data is opened on the server. Again, it is handled by a sepearate thread in order not to block the rest of the program. It listens for incomming messages and when message is received it searches for a given data identification.

If data with the given identification is found, it is sent back to the node. If there are no such data an error message is sent back. Connection is closed after replying in both cases.

5.4 Client application

Client application does all the computations. After started it connects to the server and awaits its commands.

Connection to the server is handled by the `Client` class in a separate thread in order not to block the application. After connecting to the server it generates an `UUID` and is waiting for server commands.

Also, in another thread it starts listening on specified `UDP` port for communication between nodes. Any data received on this port are passed to the running algorithm.

When a start command is received from server it starts computing the task in a separate thread. It supplies all the functionality to the node it requires. This includes sending and receiving messages that are sent between nodes, downloading distributed data from server and list of coworkers on the task.

When the task is finished it sends the result to the server and when it fails it sends to the server the reason it failed. On stopping it attempts to stop the algorithm using `stop()` method. This is highly dependent on the task implementation. After the task is stopped it informs the server.

Handling of received messages is done in a same way as on the server. However, different handlers are implemented as each application does not have to handle all messages there are. As on the server, Baeldung[5] tutorial was used for writing this part of client.

5.4.1 Blinkt! module

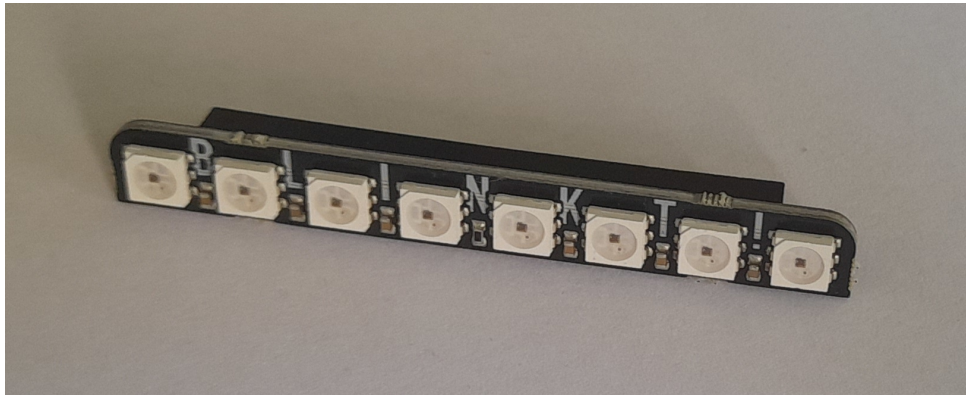


Figure 5.2: Picture of Blinkt! module

Blinkt! module¹¹ (shown in the Figure 5.2) is connected to the RaspberryPi GPIO header and can be used by the client application to show information. Interaction with the module is done exclusively using the `BlinkIt` class¹². This class hides the details of communication with module and publishes a set of methods that can be used either by the `Client` class to display state of the client application or by the task running in the client application.

If not overridden by the algorithm running on the node the Blinkt! module shows current state of node application. The meaning of each LED is described below. The LEDs are numbered from left to right beginning with zero and ending with seven if placed the same way as can be seen on Figure 5.2.

- *LED 0* lights up green if the node application has started successfully.
- *LED 1* lights up green when connection to the server was established or yellow if the connection was interrupted.
- *LED 2* lights up green if node has assigned ID and is ready to participate in distributed algorithms.
- *LED 3* and *LED 4* light up red when the node application has terminated.
- *LED 5* lights up red if the computation failed on the node.
- *LED 6* lights up red if the computation was stopped by server.
- *LED 7* lights up yellow, when the node is doing computations in distributed algorithms and green if it has finished the distributed algorithm.

¹¹<https://shop.pimoroni.com/products/blinkt>

¹²<https://gitlab.fel.cvut.cz/jankurom/semestralni-projekt-a-bakalarska-prace/-/blob/master/picocluster-distributed-algorithm/node/src/main/java/cz/cvut/fel/jankurom/picoclusterDistributedAlgorithm/node/client/BlinkIt.java>

These states can be set using methods from the `BlinkIt` class. Every task can access the `Blink!` module thru the `TaskRunner` interface defined in library. It has two methods

```
void setLed(int index, java.awt.Color)
void setLed(int index, int red, int green, int blue)
```

that allow programmer of the algorithm to control the LEDs. When these methods are called, methods with same name in the `TaskListener` interface are called on the `Client` class which then calls appropriate method on the `BlinkIt` class.

Before setting the LEDs from the task a backup of current state is made and after the task stops using the LEDs (typically after finishing, stopping or failing the task) the state is restored. This is done automatically and does not require any action from the programmer.

5.5 Communication

There are three ways of communication used by the applications. These ways of communications enable server to orchestrate nodes, distribute large data from server to the nodes and to enable algorithms to send messages between nodes. Let's describe each way of communication in detail.

5.5.1 Main client-server communication

This is the main communication that happens between client and server. It is used to control the nodes, give them tasks and collect results from them. The server listens for incoming TCP connections from clients on specified port (default is 12345). Messages are then sent through the socket encoded as an JSON object with the following scheme

```
{
  "type": "<TYPE>",
  "from": "<SENDER_UUID>",
  "to": "<RECIPIENT_UUID>",
  "content": "<CONTENT>",
}
```

`from` specifies sender of the messages. Server fills it out with `server` value and nodes use their UUID in a String form. `to` specifies the recipient of the message and is filled out the same way as `from`.

`content` is the content of the message and this can contain various data encoded as a string. The way the data should be interpreted is dependent on the message type. `type` specifies the type of the message and what information it carries. The type of the message should be written in capital letters.

Let's describe all the implemented messages, their use and content.

passed to it on its start. This array can be empty if the task has no parameters. `network` is an array of strings that contains UUIDs of all nodes collaborating on this task including the UUID of the node this message is sent to.

- **STATUS** is sent by a client to server to inform about the progress made in the task. This is implemented by the task itself. In the content a string is included describing the status of the node.
- **STOP** is sent by server to client to stop computation of the current task. Task should stop as soon as possible but this is implemented by the task and therefore is fully dependent on it.
- **STOPPED** is sent by a node to the server to inform that the task had been stopped and the client is ready to start a new task.
- **UUID_CONFIRM** is sent by a server to client and it confirms the UUID sent by the node. The UUID is included in the content in its string form. Client must use this UUID from now on.
- **UUID_REQUEST** is used by a client to request UUID on the server. Client generates an UUID and includes it in a String form in the content of the message. Then awaits the reply from the server.

It is also a first message that the client must send as he needs to have an UUID assigned. Because client has no UUID assigned yet it leaves the `from` empty.

■ 5.5.2 Client-server data download

This communication happens on port one higher than the port for main client-server communication (by default 12346). It enables the algorithms to download additional data from the server that have unique identification. Data are loaded from disk by the `DataDistributor` on server. All communication is in JSON format.

Client initiates the communication by opening the socket and sending request message

```
{
  "key": "<IDENTIFICATION>"
}
```

where the `key` is the unique identification of the data on the server. The server retrieves the requested data and sends them back to the client using response message

```
{
  "error": "FLAG",
  "data": "<DATA>"
}
```


Chapter 6

Distributed algorithms

Several algorithms for testing and measurements were implemented during development. However, it is meant for users to implement their own algorithms to run on the platform.

6.1 Implementing algorithms

All algorithms must extend the `AbstractAlgorithm` class that defines the interface used by the platform to run the distributed algorithms. It defines methods that the algorithm should implement in order to work correctly. Below is a list of all methods with description.

Algorithm can have other methods and classes defined. However, these classes and methods should be in the package in which the main algorithm class is placed or in its subpackages to avoid collisions.

- `String run(String[], TaskRunner)` is the main method containing the algorithm itself. This method is run on the client after receiving `START` command. Parameters given by user are passed to the algorithm using the `String` array and `TaskRunner` is interface thru which the algorithm can use functionality provided by the platform. It is described in more detail below. Partial result is returned using `String` return value. In case the algorithm fails it should throw an exception that will be caught by the client application and an entire algorithm will be stopped.
- `String composeResult(String[])` is method used by server to combine partial results from each client given in the parameter. These partial results are returned by the `run()` method. This method is only called when the whole algorithm finishes and not when one client fails or the task is topped by the user. Return value of this method is the final result of this algorithm.
- `String name()` returns human-readable name of the algorithm that appears on the server in drop down menu with list of algorithms.
- `String description()` returns short description of the algorithm.

- `String[] parameters()` returns array of strings that are describing each parameter of the algorithm. These descriptions are displayed to the user when starting the algorithm. An empty array must be returned when algorithm has no parameters.
- `void stop()` this method is called when client wants to stop the algorithm. The algorithm should stop at earliest possibility and should call `stopped()` method on the `TaskRunner` class after finishing.
- `void receivedMessage(String, String)` is called by client when message on UDP is received from another node. The content of the message and sender UUID is given as parameters. This method can be empty if the algorithm does not send messages through the UDP connection.
- `Map<String, File> getDistributedFiles()` returns `HashTable` containing files that should be loaded by data distributor and algorithms can download them. The `String` will be used to identify the given file on the server. An empty hash map must be returned if the algorithm has no distributed files. Use sufficiently unique identifying string to avoid collisions with other algorithms.

`TaskRunner` is an interface defining methods available to the algorithm. It is passed through the `run()` method's parameters to the algorithm. Algorithm can use this interface to perform selected tasks on the platform. All available methods are listed below. All methods are blocking.

- `void sendStatusUpdate(String)` sends a status message to server containing string given as a method parameter.
- `void setLed(int, Color)` and `void setLed(int, int, int, int)` set specified LED on the Blink! module to a specified color. Two ways of defining the color can be used, either `Color` from `java.awt` or by supplying values for red, green and blue colors.
- `void taskStopped()` should be called by algorithm when it is stopped after the client called the `stop()` method.
- `String[] getCoworkers()` can be used by algorithm to get list of UUIDs of the coworkers that are working on the same task.
- `void sendMessageToNode(String, String)` is used by algorithm to send message with given content to a coworker with specified UUID. UDP is used and therefore the delivery is not guaranteed.
- `String fetchData(String)` can be used to download data from data distributor on server. Identifier of the data is given as parameter. `null` is returned if error occurs when downloading the data or the data is not available.

Implemented algorithm should be placed in its own subpackage in the

`cz.cvut.fel.jankurom.picoclusterDistributedAlgorithm.algorithms`

package in order for the platform to be able to find it. If it is placed in a different package the `getAllAlgorithms()` method in `Algorithms` class must be changed so it searches the package in which the algorithm is placed. Otherwise, the server and the client will not be able to find the algorithm.

6.2 Implemented algorithms

Nine algorithms were implemented in total. Two of these algorithms were used to measure the performance of the platform. Results of these measurements are summarized in chapter 7. The rest of the algorithms are testing algorithms that were used to verify the functionality of the platform. All algorithms are described below with their functionality. All algorithms are available in repository¹.

- *Blinker task* makes use of the `Blink!` module and displays various patterns on it. The patterns are implemented in the `Patterns` class and are being called thru the

```
static void runPattern(int pattern, TaskRunner taskRunner)
```

method. This method calls the appropriate method depending on the pattern ID passed as an argument. Implemented patterns are

- *RGB snake* displays eight colors on the LEDs (red, orange, yellow, light green, green, cyan, blue and purple) from right to left and shifts them one position left every 250 ms.
 - *RGB fade* slowly changes colors starting from red, going thru green and blue and returning to red.
 - *RGB disco* flashes random colors.
 - *RGB runner* flashes red, green and blue color from right to left.
- *Calculate Pi* is a testing task that calculates π using randomly selected method. Server side of the algorithm then does basic statistics using values received from clients as these are being calculated with certain precisions.

Methods used to calculate π are:

- using the constant² from Java standard library

```
java.lang.Math.PI
```

¹<https://gitlab.fel.cvut.cz/jankurom/semestralni-projekt-a-bakalarska-prace/-/tree/master/picocluster-distributed-algorithm/library/src/main/java/cz/cvut/fel/jankurom/picoclusterDistributedAlgorithm/library/algorithms>

²<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Math.html#PI>

- approximation by summing first thousand members of *Gregory-Leibnitz series*³

$$\pi = 4 \cdot \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1}$$

- approximation by summing first twenty five members of *Nilakantha series*⁴

$$\pi = 3 + \sum_{n=0}^{\infty} \frac{4 \cdot (-1)^n}{(2n+2) \cdot (2n+3) \cdot (2n+4)}$$

- approximation by using limit⁵

$$\pi = \lim_{x \rightarrow \infty} x \cdot \sin\left(\frac{180}{x}\right)$$

where x is `Integer.MAX_VALUE`.

- calculation using `arcsin()`⁶

$$\pi = 2 \cdot \left(\arcsin\left(\sqrt{1-x^2}\right) + |\arcsin(x)| \right)$$

where $x \in \langle -1; 1 \rangle$.

- *Distributor tester* is a task for testing the distributor functionality of the server used by algorithms to download data from server. When the task starts it downloads a text from the server and sends it back as a result of the task.
- *Fail task* was designed to test the functionality of handling errors on client's as well as server's side. It takes one parameter that is float from interval $\langle 0; 1 \rangle$ that sets probability that node running this algorithm will fail. 0 means no chance to fail and 1 is 100% chance the node will fail. The node generates number from interval $\langle 0; 1 \rangle$ from continuous uniform distribution.
- *Interval message* is the first task used for measuring the performance of the network and nodes. After the task is started it waits for two seconds to let coworkers initialize as well. Then it starts sending UDP messages with specified size and in specified intervals to all coworkers. The number of messages is also specified as parameter of the task.

After it is done sending messages it waits for additional ten seconds for any delayed messages and then calculates how many messages it received and what the success rate is. This is sent to the server that sums the statistics.

³<https://www.mathscareers.org.uk/calculating-pi/>

⁴<https://www.mathscareers.org.uk/calculating-pi/>

⁵<http://www.ams.org/publicoutreach/feature-column/fcarc-pi-calc>

⁶<https://www.wikihow.com/Calculate-Pi#Using-Arcsine-Function.2FInverse-Sine-Function>

- *Response message* is the second task used for measuring the performance of the network and nodes. After it is started it waits for two seconds to let coworkers initialize and then it starts to send numbered messages to all coworkers with specified size and in specified intervals. Then it waits for responses and measures the time elapsed from sending. The number of messages sent is specified as an parameter.

After it sends all messages it waits for ten seconds and then calculates statistics mainly the success rate and average response time. These statistics are then sent to the server that combines them.

- *Sleep* is the first implemented task that was used to test the functionality of client and server. It takes two parameters of minimum and maximum sleep time in milliseconds. Node then sleeps for a random duration within the maximum and minimum and then sends server message that he had finished.
- *Status updater* is designed to test functionality of status updates sent from client to server. It takes a positive integer as an argument that defines number of messages that will be sent to the server. Algorithm waits for 5 seconds between sending subsequent status messages.
- *UDP tester* was made for testing the UDP communication between nodes. When it is started it sends messages to all coworkers and awaits messages from them. When it receives all messages it finishes. The algorithm waits for messages indefinitely and needs to be manually stopped.

Chapter 7

Experiments with distributed algorithms

After the platform was finished it was subject to a several measurements. The purpose of these measurements was to characterise performance of the platform. The main focus was on the network as it will be the bottleneck for most of distributed algorithms. Two specially developed algorithms were used for the measurements. These algorithms use the UDP communication between nodes and measure performance.

- *Interval Message* sends UDP packets to all its coworkers. This packet has a specific size and is sent repeatedly with waiting time between repeated sendings. Number of repeated sending, size and waiting time are parameters of the algorithm. All nodes have the same parameters. The success rate is measured as each node knows how many messages it should receive and can compare it to how many messages it has received.
- *Response Message* sends UDP packets to all its coworkers and awaits response from them. Again, the messages have certain size, are repeatedly sent in specified intervals and in specified number as in the *Interval message*. The success rate of delivering messages is measured as in the *Interval Message* however, in addition a response time is measured.

Both algorithms were measured with various parameters that gradually increased the load on the network. In addition, both algorithms were measured when nodes were sending notifications to the server and when they were not. This resulted in four sets of data that can be found either on enclosed CD or at gitlab repository.

When sending notifications to server node is sending **SENT** message to the server after sending message on UDP connection between nodes. As the **SENT** message is sent with each UDP message it increases the load on the network.

To ensure equal conditions Docker image was restarted before each measurement. In addition, a limit for reliability was set. After the reliability of delivery went under 75 % when increasing load on the cluster, the measurements were stopped.

In order not to copy all tables here, several interesting phenomena were selected and described below.

- Contrary to what was expected the reliability does not depend that heavily on the notifications being sent or not as can be seen in Figure 7.1. Moreover, with bigger load the algorithms probably become more reliable, as can be seen on the graph.

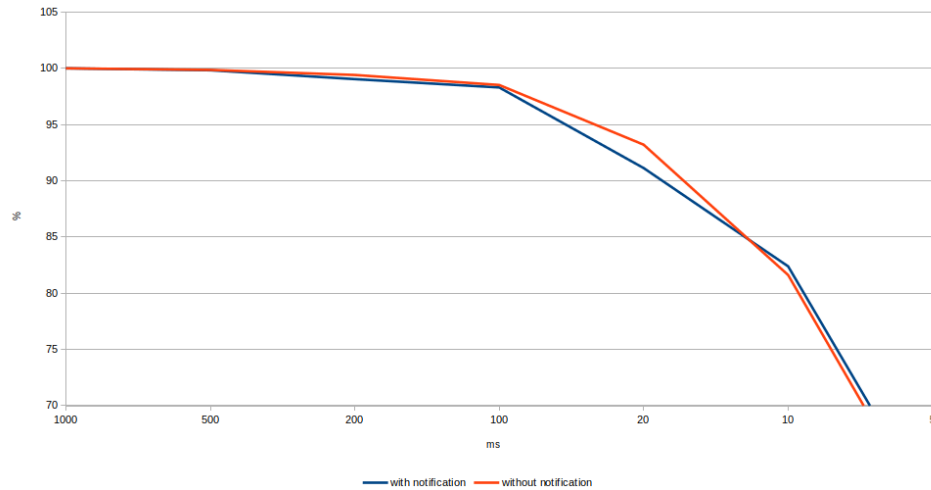


Figure 7.1: Comparison of reliability with and without notifications depending on delay between messages

This can be caused by the fact that the network card on RaspberryPi is connected thru USB which limits the transfer speed. The switch is then able to switch the data more reliably even though more data is being sent. The data for the graph were taken from *Interval measurement* algorithm with packet size of 10 kB and with varying delay between messages and number of message.

The same can be said about the time measured as shown in Figure 7.2. The difference in response time is not that large. The data were taken from *Response measurement* task with 10 kB size of packet.

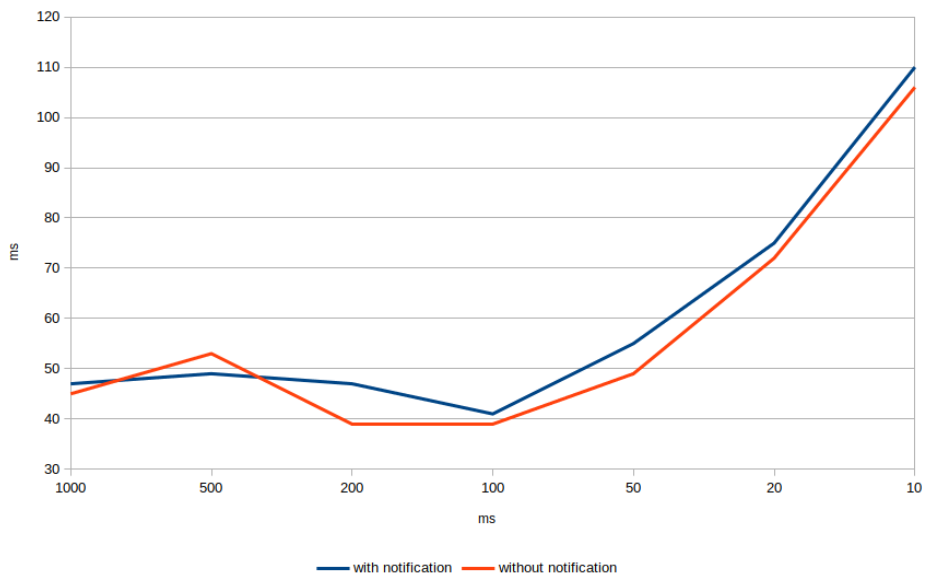


Figure 7.2: Comparison of response time with and without notifications depending on delay between messages

However, the rise in response time is not solely caused by the network itself. The biggest factor is the way the messages received on UDP are being handled. They are being handled one after another causing them to be in a queue for a long time. This significantly increases the response time.

- Response time has a dip or even double dip that was unexpected. Generally, the response time rises with more data being sent. However, at the beginning it takes a small dip or even a double dip in case of bigger data being sent. This can be seen on Figure 7.3. It was not discovered what causes this phenomenon.

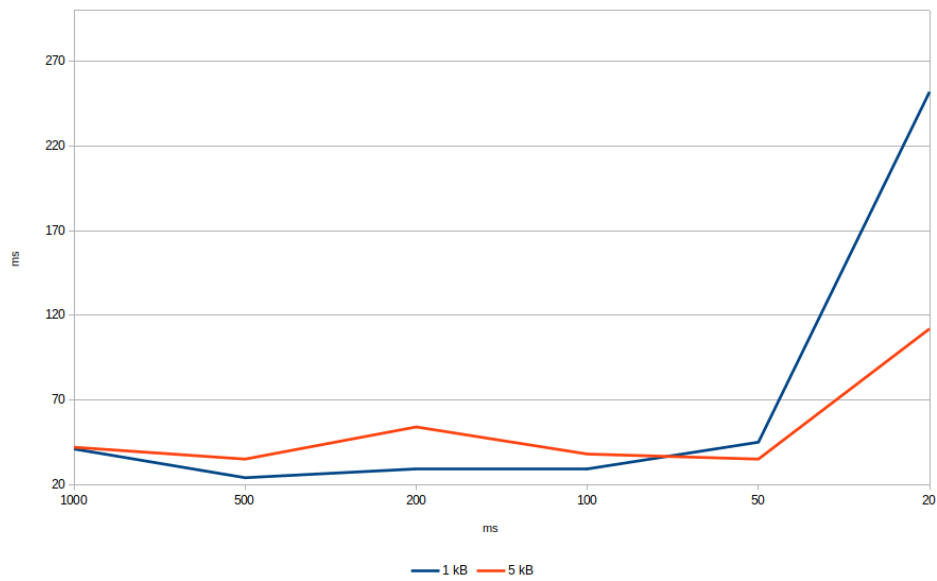


Figure 7.3: Comparison of two response curves depending on delay between messages

- 10 kB data are being delivered more reliably than 5 kB data. This happened in *Interval measurement* algorithm and something very similar happened in *Response measurement* algorithm as well. The percentage of delivered 5 kB packages drops slower than for 10 kB but after some time it starts to drop much faster and the delivery of 10 kB packages becomes more reliable. This can be seen on Figure 7.4.

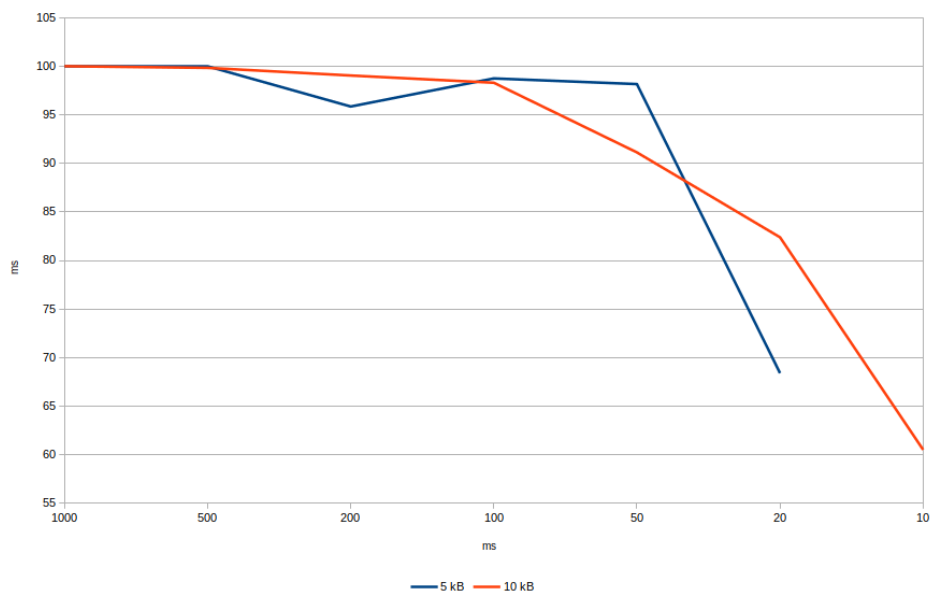


Figure 7.4: Comparison of two delivery rates depending on the delay between messages

This is probably caused by the switch. It has a limited number of packets that can be switched in a given amount of time. When the 5 kB data are being sent, the RaspberryPi sends more smaller packets which means the switch is gonna be overloaded sooner. On the other hand, when the 10 kB data are being sent, the RaspberryPi sends fewer but larger packets. This means the switch becomes overloaded later because it has fewer packets to switch although the packets are larger in size.

As can be seen on versitron[11], there are two main characteristics of an Ethernet switch. The first one is *switching capacity* written down in bytes per second or its multiples. This specifies how much data the switch can handle in one second. The second one is *forwarding performance* written down in packets per second or its multiples. This defines how many packets can the switch handle in one second and probably this limit is being exceeded above resulting in packets being dropped. More about this can be found in Cisco¹ and Huawei² forums.

¹<https://community.cisco.com/t5/switching/quot-switching-bandwidth-quot-and-quot-forwarding-rate-quot/td-p/2116399>

²<https://forum.huawei.com/enterprise/en/forwarding-performance-and-switching-capacity/thread/570609-861>

Chapter 8

Software testing

Several methods of testing were used to minimise the number of bugs. However, in project of this size and complexity it is hard to find and eliminate all the bugs. Several methods of testing were selected for this project to ensure the best testing that was possible and to find most of the bugs.

- *Static testing* was used during the whole process of writing source code. It was used to catch typos and obvious bugs early in the development where they are easy to fix. *QAPlug*¹ for IntelliJIdea was used to automate these tests.

Many potential problems were eliminated by this method. For example usage of concrete class instead of interface from `java.util`, naming conventions, making classes final, hiding constructors and other issues. These issues were not bugs as such but will improve the readability of the code and will make future modifications easier thanks to cleaner code.

However, discretion is needed when fixing issues found out by the tool. Not all issues are real issues and do not need to be fixed. Example of this can be found in `Blinker`² on line 41. Analysis tool points out the empty `while()` loop as a potential problem but it is being used for waiting until the pattern finishes. It might not be the best way to do this but is simple and works as intended without causing any problems.

In addition, *Google Java Style Guide*³ was used for formatting the source code. This improves readability of the source code and prevents mistakes. Again, automated tool was used for this in a form of *google-java-format*⁴ plugin that automatically reformats the code.

Several interesting statistics were also found out during static analysis of the code using *Statistic*⁵ plugin. These statistics are:

¹<https://plugins.jetbrains.com/plugin/4594-qaplug>

²<https://gitlab.fel.cvut.cz/jankurom/semestralni-projekt-a-bakalarska-prace/-/blob/master/picocluster-distributed-algorithm/library/src/main/java/cz/cvut/fel/jankurom/picoclusterDistributedAlgorithm/library/algorithms/blinker/Blinker.java>

³<https://google.github.io/styleguide/javaguide.html>

⁴<https://plugins.jetbrains.com/plugin/8527-google-java-format>

⁵<https://plugins.jetbrains.com/plugin/4509-statistic>

- Module *library* has 1431 lines of source code, *node* has 1240 lines of code and *server* has 2576 lines of code. This adds up to 5247 lines of code in total.
 - Module *library* has 23 Java classes, *node* has 14 Java classes and *server* has 31 Java classes. This is 68 Java classes in total. Interfaces and enums are being counted as classes for this purpose.
 - Algorithms are in 11 Java classes with 939 lines of code.
 - There is one unit test class with 27 lines of code.
- *Unit testing* was used to test tasks and especially the computation of π using different methods. *jUnit*⁶ was used for testing and the tests were launched on each build as a part of Maven build script. All the unit tests are in the Git repository.

Using the unit tests it was uncovered that at least the first thousand members of Gregory-Leibnitz series must be summed to have sufficient precision however only first twenty-five members of Nilakantha series are sufficient to have better precision.

It would be too complicated to test other parts of the application using unit testing. It would require extensive mocking of important parts of application. Manual testing and testing tasks were used instead.

- *Manual testing* was the most important part of testing. Due to the complexity of developed platform it was the easiest way to test the functionality and catch bugs. Several approaches to testing were used.

In the early stages of development when only the connection between server and clients was established a logging of communication was used. Since the communication is in JSON format it is easily readable by humans and the communication can be debugged in this way.

Reading the communication also proved to be very helpful when developing the functionality for starting, ending and stopping tasks. However, in this case, it was also used in conjunction with very simple testing tasks that were created specifically for this purpose. Thanks to them it was easy to get this part of platform working.

More testing tasks were developed later to test other functionalities of the platform. For example downloading additional data from server, communication between clients, sending status updates to server, controlling Blinkt! module and others. These tasks proved to be irreplaceable as testing the functionality other ways would be much more laborious.

All these testing tasks are included in the GitLab repository and can be used as a start point in custom algorithms development for the platform.

- *Deploy to Pico 48 cluster* was used to test how well the platform can scale on a bigger cluster. It is exactly the same from hardware as well

⁶<https://junit.org/junit5/>

as software perspective as the cluster with five RaspberryPis. The same pipeline, tools and scripts were used as for the cluster with five RaspberryPis. The cluster running *Blinker* task can be seen on Figure 8.1.



Figure 8.1: Pico 48 cluster running Blinker task

The main difference was a considerably longer deploy time and this was expected. The deploy and runtime pipeline is not optimal and can be vastly improved as mentioned in section 9.1.

However, the longer time of deployment was the only noticeable difference between working with cluster with five and forty-eight RaspberryPis. Granted, no in-depth testing of performance was done, only a couple of testing tasks were run to verify the functionality.

■ *User testing*

It was hard to do user testing on this project. This project is aimed at a very small group of people with special needs. Therefore, finding a user that will test the project was extremely difficult. However, one user that would test the project was found. In fact, the user is using the project to run his distributed algorithms and perform his own tests and measurements. Full feedback of the user can be found in attachments and summary of the most important ideas is included below.

- The documentation is very well written, it contains most of the information about the platform and is easy to navigate.
- Setup was easy. All scripts for the pipeline were prepared and only problem was bad indentation of three lines in deploy script and wrongly specified user.
- The first deployment took a lot of time and probably is inefficient. It copies unnecessary files to the RaspberryPis.

- The server has hardcoded path to the configuration file which resulted in problems.
- The number of nodes that run the task is badly marked in the dialog making it hard to spot.
- There was no problem implementing custom pattern for Blinkt! module and it took less than 20 minutes.
- The code is very well written and documented which makes it very easy to read and understand. It also follows the correct Java principles.

Overall, the platform was rated very positively, was easy to setup and work with and user enjoyed working with it.

Chapter 9

Conclusion

The objective of this work was successfully met, all the points from the assignment were accomplished and functionality for all requirements was done. The result is a finished platform for running distributed algorithms on the RaspberryPi cluster.

At first, the PicoCluster was studied from hardware as well as software perspective. All the collected information was used to propose a solution. Based on this proposal software pipeline for development and deployment of the client-server application that runs the distributed algorithms was created. Also, runtime environment for the client application on the PicoCluster was created. After that, a client-server application for running the distributed algorithms was developed alongside with selected distributed algorithms.

These algorithms were then used to test the platform and measure performance of the platform. The acquired measurements were then studied and interesting phenomena were discovered. The testing was not done only by testing tasks but also using static analysis of the source code and unit testing as well. In addition, one target user was found to do the user testing and shared his feedback.

Finally, a user guide for setup as well as for implementing custom algorithms was created to make it easy for future users to use the platform. Potentially, the platform can be expanded to include more functionality.

9.1 Future improvements

During development improvements and features were proposed and many of them were implemented. However, not all improvements and features were implemented due to the lack of time or because they were not that important. Below is the list of these improvements and features that were not implemented.

- *Pipeline improvements* are required. As of now, Docker image is being built on each RaspberryPi separately. This is a huge performance issue that has to be solved. The docker image should be built once, uploaded to official Docker repository or repository on machine used

to orchestrate the RaspberryPi and from there downloaded by each RaspberryPi and run. This will shorten the deploy time considerably.

- *Prebuilt Docker image* to skip repeated actions during deployment. Each image starts by installing Java and drivers for RaspberryPi GPIO headers. This can be done in a separate image that will be then used as a starting point for image with client application. It will shorten the deploy time.
- *Loading tasks from JSON* to make it easier to run tasks on the platform. JSON describing tasks that should run on the network will be created by user and then loaded in the program. Then it will be parsed by the server program and tasks will be created in queue. These tasks will be then automatically run.
- *Server part of algorithm* that will be launched only once on the server machine. Now, the platform can only run fully decentralized algorithms and server part only does result agregation. After adding the server part algorithms could use a custom server part. That will mean that algorithms that require strong server part running as 1:N could be run on the platform.

These improvements are worth a future development as they would make the platform faster and easier to use.



Bibliography

- [1] Ansible documentation. [online]. Ansible project contributors. [October 19th 2021]. Available at: <https://docs.ansible.com/ansible/latest/>
- [2] How to Use Ansible to Install and Set Up Docker on Ubuntu 20.04 | DigitalOcean. [online]. DigitalOcean. [October 19th 2021]. Available at: <https://www.digitalocean.com/community/tutorials/how-to-use-ansible-to-install-and-set-up-docker-on-ubuntu-20-04>
- [3] Docker documentation. [online]. Docker Inc. [October 19th 2021]. Available at: <https://docs.docker.com/>
- [4] Java SE 11 & JDK 11 documentation. [online]. Oracle. [November 19th 2021]. Available at <https://docs.oracle.com/en/java/javase/11/docs/api/index.html>
- [5] A Guide to Java Sockets | Baeldung. [online]. Baeldung. [April 30th 2022]. Available at: <https://www.baeldung.com/a-guide-to-java-sockets>
- [6] A Guide to UDP In Java | Baeldung. [online]. Baeldung. [April 30th 2022]. Available at: <https://www.baeldung.com/udp-in-java>
- [7] Maven - Maven Documentation. [online]. The Apache Software Foundation. [November 28th 2021]. Available at: <https://maven.apache.org/guides/>
- [8] Tutorial: Debian Router/Gateway in 10 Minuten einrichten - gridscale. [online] gridscale [April 30th 2022]. Available at: <https://gridscale.io/community/tutorials/tutorial-debian-routergateway-10minuten/>
- [9] NetworkConfiguration - Debian Wiki. [online] Debian Wiki [April 30th 2022]. Available at: <https://wiki.debian.org/NetworkConfiguration>
- [10] iptables(8) - Linux man page. [online] linux.die.net [April 30th 2022]. Available at: <https://linux.die.net/man/8/iptables>
- [11] Three important Network Switching Parameters | Versitron. [online] Versitron [May 1st 2022]. Available at: <https://www.versitron.com/whitepaper/network-switching-parameters>

Appendix A

How to use platform

This is a complete guide describing how to deploy and use the platform for running distributed algorithms. At least basic knowledge of UNIX system including file system, networking and software installation is required.

This tutorial was written for Debian operating system. It might require some changes to work with other operating systems.

1. Preparation of server machine

Server machine should be running Linux, as the Ansible requires UNIX to run. Windows with Windows Subsystem for Linux can be used but this solution was not tested.

Java at least version 11 is required to run the server application. The server application uses settings in file `options` and this file is required to start the server application. It must be placed at the same place as `.jar` archive.

Any files used by distributed algorithms must be placed at the location defined by the algorithm.

Ansible is required to run the deployment scripts. It only needs to be installed on the machine that will be running the script and not the target machines. On Debian, it can be installed using command

```
apt install ansible
```

2. Internet access

Client machines require internet access for their functionality. If they are connected to the internet thru the server machine that will act as a router IP forwarding and firewall must be set. This procedure was taken from online tutorial[8] and modified using Debian Wiki[9] and iptables manual page[10] for this purpose.

Firstly, network interfaces must be set up correctly. The easiest way to do this is using the `/etc/network/interfaces` configuration file that should look as follows

```
source /etc/network/interfaces.d/*
```

```
auto lo
iface lo inet loopback

allow-hotplug eno1
iface eno1 inet dhcp
iface eno1 inet6 dhcp

allow-hotplug enp2s0
iface enp2s0 inet static
    address 10.1.10.1
    netmask 255.255.255.0
```

`eno1` is interface going to the internet and `enp2s0` is interface going to the network containing the RaspberryPi cluster. You may need to change interface names and configuration of the interface going to the internet according to your local network settings.

Secondly, a line

```
net.ipv4.ip_forward=1
```

must be uncommented in `/etc/sysctl.conf` to allow Linux to forward IP packets and act as a router.

Lastly, a tool `iptables-persistent` must be installed. This will backup iptables settings and restore them after powering the server machine off and on. During installation, let the tool create backups of current rules in the `/etc/iptables/rules.v4` file. Then, modify this file to have the following content

```
*nat
-A POSTROUTING -o eno1 -j MASQUERADE
COMMIT

*filter
-A INPUT -i lo -j ACCEPT
-A INPUT -m state --state RELATED,ESTABLISHED -j ACCEPT
-A INPUT -i eno1 -j DROP
COMMIT
```

These settings will enable NAT from the inner network to the internet and forbid any incoming connections apart from those that were opened from the inner network. You may need to use different device names. `eno1` is network interface to the internet. You may combine these rules with your rules you have configured before.

To reload iptables settings use command

```
iptables-restore < /etc/iptables/rules.v4
```

to load settings from the file we have just modified.

3. Preparation of client machines

Ansible requires all clients to have the same user with same password. In addition, this user must be added to the sudoers file. If you are using PicoCluster this is already done.

After that, IPv4 addresses of all client machines must be added to the `hosts` file. You can use the existing `cluster` group and replace the already existing IP addresses or define your own group. If you define your own group, you need to replace the name in all Ansible scripts.

Finally, server address, port, maximum packet size and notification to server must be set in the `Dockerfile`. These three parameters are set using environmental variables.

The variable `PICOCLUSTER_SERVER` sets the address and port of the server. IPv4 address and port number separated by a colon should be supplied. Alternatively, hostname of the server can be supplied.

The variable `UDP_PACKET_SIZE` sets the maximum packet size that can be sent between nodes using the UDP connection. The size is set in bytes and should be as low as possible.

The variable `END_SENT_MESSAGE_INFO` sets whether client will be sending notifications to server every time he sends packet over the UDP connection to another client. Allowed values are `false` and `true`. When it is set to `false` client will not send notifications to the server, when set to `true` it will.

4. Deploying client application

Deployment of the client is done automatically using Ansible scripts. These scripts install all software and dependencies required for the client application to run on the client machine.

All the Ansible scripts are written specifically for use with RaspberryPi running Raspbian operating system. For use with other platforms and operating system they might need to be modified.

For first deploy to new machines use the `full-deploy.yml` script which will install Docker and all dependencies as well as deploy the client application. Use this command to run the script

```
ansible-playbook -i ./hosts -k --ssh-extra-args='-o "
  ↪ PubkeyAuthentication=no"' -u picocluster full-deploy.
  ↪ yml
```

If you are using different user then replace `picocluster` with the name of the user you are using.

To deploy to machines that already have Docker and all dependencies installed, you can use `fast-deploy.yml` script. It has a shorter execution time thanks to omitting installation of Docker and all dependencies.

To only restart the application running in Docker container use `restart-docker.yml` script which only restarts the running Docker container and application running in it.

The server application must be running when the client applications are being started. If the client application cannot connect to server, it will fail.

To stop Docker containers and applications in them use `stop-docker.yml` script.

5. Graphical user interface of the server

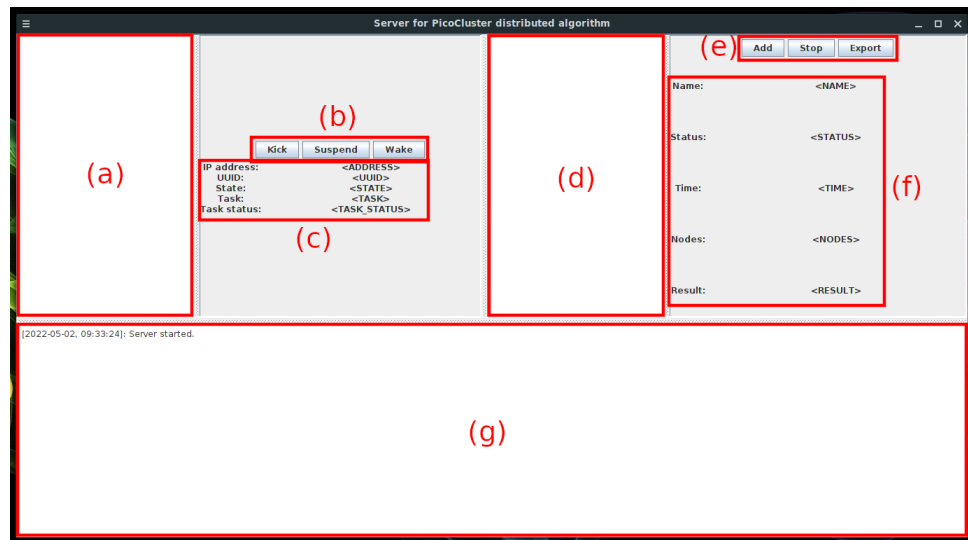


Figure A.1: Graphical user interface of server application

Graphical user interface of server is show on Figure A.1 and bellow is description of its components.

- a. List of connected nodes
- b. Buttons to control nodes (*Kick* kicks selected node from server, *Suspend* changes selected node to suspended state and *Wake* changes selected node to ready state)
- c. Table with information about selected node
- d. List of all tasks on the server
- e. Buttons to control tasks (*Add* opens a dialog for adding a new task, *Stop* stops selected running task or removes ready task and *Export* exports all tasks to JSON file)
- f. Table with details about selected task
- g. List of logs

6. Running tasks

Click the *Add* button in the main server window. A dialog for adding new task will appear as shown on the Figure A.2.

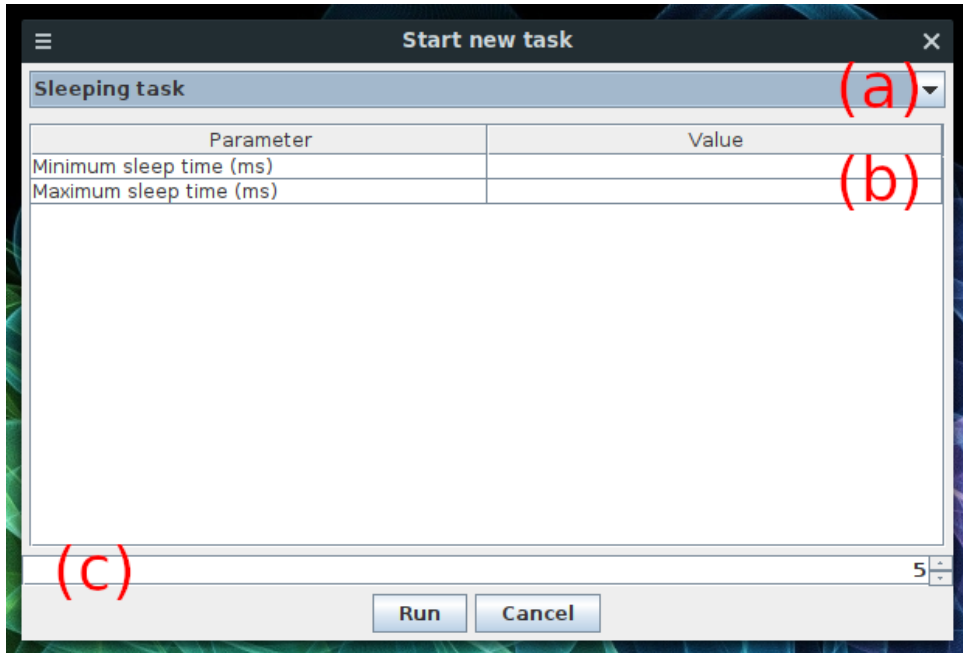


Figure A.2: Dialog for adding new task

Fill the dialog as follows

- a. Use the combobox to selected algorithm you want to run.
- b. Fill the table with parameters for the algorithm. All inputs need to be confirmed by pressing **Enter** key. Otherwise, the input will not be saved.
- c. Select the number of nodes you want the algorithm to run on.

After filling the dialog click *Run* to add the task to queue. The added task will be started as soon as it will be first in the queue and sufficient number of nodes in ready state will be available. The task will fail if the number of connected nodes is smaller than number of nodes on which the task should run. To close the dialog without adding new task click *Cancel* button.


7. Exporting results

Data gathered from distributed algorithms can be exported from the program using the *Export* button. Program asks for location and name of the file where the data will be exported. After dialog confirmation all results are written in JSON format to the file.

All tasks are exported using this method including **READY** and **RUNNING** tasks.

The file has following format

```
[
  {
    "partialResults": [
      "result1",
      "result2"
    ],
    "name": "<NAME_OF_TASK>",
    "nodes": <NODE_COUNT>,
    "result": "<TASK_RESULT>",
    "state": "<TASK_STATE>",
    "fullName": "<FULL_CLASS_NAME>",
    "startTime": "<START_TIME>",
    "endTime": "<END_TIME>",
    "parameters": [
      "param1",
      "param2"
    ]
  }
]
```



Appendix B

Attachments

All attachments are included on the CD enclosed with printed thesis or are available at project's GitLab¹ repository.

- Document *measurements.pdf* contains all data measured when *Interval measurement* and **Response measurement** were run on the platform. The graphs in chapter 7 are created from data in this document.
- File *feedback.md* contains the raw feedback given by tester. It was shortened in chapter 8 in *User testing* part.
- Archive *source.zip* contains source code of all applications developed in this thesis. The source code was taken from *Bachelor thesis* release from the GitLab.

¹<https://gitlab.fel.cvut.cz/jankurom/semestralni-projekt-a-bakalarska-prace>