Eötvös Loránd University

Faculty of Informatics

Dept. of Software Technology and Methodology

Automated testing of networked applications

Supervisor:
Cserép Máté
Assistant Lecturer

Author:
Jakab Levente
Computer Science MSc

Budapest, 2023

Contents

Chapter 1

Introduction

As of 2023 it seems to be the case that automated testing is considered an extremely important practice in software development [1, 2, 3]. The main reason for this is that by automating tests, developers are able to avoid having to manually write out sizeable, difficult and repetitive test cases, thereby reducing time and resources spent on testing software. This method is also critically important for the principles of continuous testing and continuous delivery.

As an example, instructors in universities already have many difficulties keeping track of all of the submissions for each of their subjects, let alone having to manually check and test each submission one-by-one [4, 5]. This problem gets even more complicated when the to-be-tested submissions require additional setup steps like in the case of networked applications, especially in multi-agent networked application systems where ensuring communication between agents is of paramount importance.

This exact case was the main motivation behind the thesis, as one of the courses running in the Faculty of Informatics, Eötvös Loránd University deals with such networked applications. The instructors of the Telecommunication Networks course have expressed prior demand for automatic testing capabilities, as manually testing multi-agent networked applications has often proven to be tedious.

The aim of this thesis paper is to analyze and implement an automated networked application testing framework into an existing system. The *TMS* assignment-management and plagiarism-detection autograding system developed at the Faculty of Informatics, Eötvös Loránd University was used for this prototype implementation. *TMS* already possesses automated testing capabilities, just not for networked applications. The goal would be to perform adequate research, collect information

from the research, then - using this information - implement the prototype and draw the appropriate conclusions based on the results.

The primary goal of this paper is to examine the way instructors are able to specify network connections and components, as well as how to most effectively test student submissions for general tasks involving networked applications. To this end, there are five various chapters progressing through the topic at hand. I will start by introducing my research of the topic in related scientific papers and other pieces of research, then describe the existing functions of *TMS* in detail, followed by a detailed look at the methodology behind my proposed model. This is followed by a chapter describing some of the specifics of the implementation, ending with taking a look at the results of the thesis, and the conclusion.

Chapter 2

Related Work

When researching related material the most important condition was to look at papers where the system under test is structured in a way where a network of independent components communicate with each other and/or possibly also with a central node in the network. This is important because this type of structuring within an application or network of applications can be equated to student submission programs and instructor programs in our case and be used by the method introduced in my thesis paper. Examples of this structuring can include containerised systems, microservice architectures, portlet architectures for web pages or even just a network of standalone computers (distributed system). The following section will deal with related topics and papers according to this delineation.

2.1 Distributed Systems

A distributed system is a system of components that are located on independent, networked computers. A paper by Torens and Ebrecht [6] introduces a testing framework for distributed systems called RemoteTest. The idea behind the framework was to address multiple problems with testing distributed systems including decoupling individual modules, abstraction of test logic and interface details as well as the questions of programming language variability and platform independence. The structure decided for the framework follows a master-slave architecture where the master is responsible for executing test cases and the slaves' job is to emulate only the relevant components on individual computers for a given SUT. Communication between the SUT and other components' interfaces is abstracted away by the slaves

also.

The testing process is broken up into three separate phases which are the setup, test execution and cleanup phases. The conclusion of the paper states that using RemoteTest enormously reduces the complexity of the system-under-test, abstracts away details of the interface to ensure easier testing, and facilitates a platform, programming language and application independent framework. As Maicus et al. notes in a different paper [7] this approach "separates system components from the environment" and thus tests the application component by component.

2.2 Microservice Architectures

Microservice architecture can be described as an architectural pattern wherein an application consists of loosely coupled, specific goal-oriented services, that communicate through lightweight protocols. This style of structure has gained steam in the world of software engineering compared to the monolith architectural style which used to be the standard structure of applications. There are many reasons for this, but the most important ones are the massively reduced deployment time and complexity, and the ease to understand and develop microservices compared to monoliths.

Sotomayor et al. compare 16 different tools for testing microservices [8] where they compare each tool based on multiple conditions including platform, test case language, and the type of testing objectives the tool has. As for the last point our aim is to examine functional testing and end-to-end testing tools because these are the objectives represented when testing student submissions. Their methodology starts by examining how each of the testing tools view the components of a microservice architecture application in terms of testing strategy (component testing, contract testing, integration testing, system testing). This analysis is followed by introducing a polyglot testbed microservices system called Rideshare which will serve as the SUT for some of the discussed tools. The case study of the paper aims to provide information about the setup and running of testing tools for Rideshare wherein they use four different testing tools to perform component and integration testing.

System test case execution times are compared and it is concluded that test harness tools performed the worst due to the communication setup with the services.

It is also noted that there are many challenges when utilizing these tools such as setup steps, resource management and platform dependencies.

2.2.1 Resilience Testing

Out of the four testing tools mentioned Gremlin yielded the best results therefore, it would make sense to examine it in detail. Gremlin is a resilience testing tool introduced in a paper [9] by Heorhiadi et al. The methodology behind Gremlin's functionalities has to do with two base principles. Firstly, all communication within the application happens over the network. This means not only that common types of failures can be emulated easily but also that recovery of these failures can be observed from the network. The second principle is that the interaction between microservices can be described using simple patterns such as a request-response pattern. What this means is that simulating failure of a single microservice can be done just by manipulating its network interactions (its HTTP responses for example).

Gremlin's structure consists of two main components, the control plane and the data plane. The control plane is the central management and orchestration layer of the platform. It is responsible for allowing the user to create error scenarios, communicating with the data plane and describing the status of the test experiments in real time. The data plane is the other main layer, it consists of Gremlin agents. Gremlin agents function as network proxies for the API calls by the control plane, whose instructions they are also responsible for executing. These agents run in each microservice component to be tested.

In the Evaluation section of the paper the authors describe case studies wherein Gremlin is used to test an application and a plugin as well. It was then concluded that Gremlin requires a minimal learning curve to start using but once in use it is very helpful in determining bugs for microservice developers.

2.2.2 Acceptance Testing

A different approach to testing microservices is introduced by Rahman and Gao [10]. The focus here is on testing microservices in a behavior-driven development environment while addressing multiple challenges with maintainability, systems integration complexity during runtime as well as with black-box testing using production-like execution environment.

The methodology behind the testing framework (Reusable Automated Acceptance Testing Architecture) lies in its architecture, specifically the way repositories are structured within the tool. Instead of separate repositories for each microservice the authors describe a central repository with three additional subdirectories. The first contains subdirectories for each running microservice with the feature files that describe BDD scenarios for that service. This type of organisation allows easy auditing of capabilities within the application. The second directory deals with the issue of reusability by storing the step implementations in a way that common implementations (that multiple services use) go in a mutual folder. The third subdirectory contains the source code for the used BDD framework.

The testing architecture is then concluded to successfully address all mentioned challenges and provide a reusable, automated acceptance testing environment for developers.

2.2.3 Regression Testing

Kargar and Hanifizade discuss automation of microservice regression testing using continuous delivery in their paper [11]. Their approach also utilizes containers to run the entire CD process. Containers are pieces of software whose purpose is operating system virtualisation. They provide a way to package and deploy applications reliably, regardless of environment. Docker is the most popular containerisation tool. The testing process starts out by setting up the pipeline, taking a Java Docker image as base and setting environmental variables. Source code is then compiled, tests are run and the resulting artifact is deployed in a Maven repository. For the individual microservices Docker images are built based on dockerfiles written inside of the microservices which are then deployed in a private docker registry.

Kubernetes is a container orchestration system for automating the software deployment process. Kubernetes is used in the next step to create a cloud-native to run containers which are separated into different environments using the namespace concept within Kubernetes. The authors describe three different environments: the production environment, where the latest stable version of the application microservices are available, the secondary environment, wherein qualitative test are performed on the microservices, and the development environment, which holds the latest developed version of the application.

Regression tests are run by a tool called Diffy which compares deterministic response outputs from the development environment to the production environment after which the tester reviews the test outputs and makes the decision as to deploy the application to production or not using rolling deployment technique.

The authors conclude that the use of their proposed model has sucessfully reduces testing costs for the National Iranian Oil Products Distribution Company and that the automation of the testing process has prevented unnecessary work by developers not having to write more test units. It is mentioned as a limitation however, that this framework only processes deterministic responses from the microservices.

2.3 Portlet Architectures

In a portlet architecture self-contained applications called portlets run inside of a web page. In-container testing of such applications is dealt with in a paper by Xiong, Bajwa, and Maurer [12] wherein they describe the benefit of using container-based technologies, but also the challenges with automating the testing process for such applications.

Their suggested framework consists of multiple, separate components with various assigned tasks. The testing client invoker initiates the testing process by telling the testing UI controller to send a request to portlets under test and write test control instructions to the common repository. After successfully executing portlet code the test results are also written in the repository and sent back to the controller.

The authors also address three main problems encountered when deploying an application in the production environment, outlined at the beginning of the paper: deployment-related problems are addressed with the use of a configuration file where testers are able to configure individual portlet deployment parameters, the issue of role-based resource access is dealt with using authentication and specific naming conventions withing tests, and the problems arising from the interaction between the container and the application code are solved by separating specific test cases into before- and after test functions to cull unexpected outcomes.

The paper concludes by stating that with the use of the framework some manual testing can be replaced by automated tests, as well as that the need for container-based testing successfully addressed problems found at deployment time.

2.4 Autograders

2.4.1 APAC

Other existing autograders were also examined as these systems are able to serve us with the most relevant information about automated, in-container testing of networked student submissions. It is general consensus that in order to perform testing on networked applications that communicate with each other over a private network one would have to somehow virtualise running environments for each agent in the network in order to simulate a physical distributed system. There are various ways to achieve this such as virtual machines (VMs), sandbox environments and of course containers.

A paper by Špaček et al. [13] investigates using Docker as platform for assignment evaluation where the authors state early on that their primary motivation for using Docker and the concept of kernel namespace isolation were security and realising a safe runtime testing environment. Kernel namespace isolation in this case means that six different namespaces are implemented from the Linux kernel version 3.8 that provide complete separation for standalone container creation. Control groups help configuring resource constraints between containers, while MAC mechanisms such as SELinux or AppArmor help improve security. Interaction with Docker containers is possible vie CLI or REST API.

A MySQL database provides storage for assignment configurations, test vectors and information about student submissions. These submissions are processed from an input queue and are executed by a batch job which is defined in an XML file that contains all cases of the process flow. Data flow between the application and container is realised through an SSH connection with the SSH daemons running in each container instance. All communication commences on a private LAN network. A sandbox pool holds all sandbox SSH sessions, which are established upon creation of a container. Uploaded files by the student are compiled on host, then sent to containers through SCP protocol, after which the execution command is submitted via exec channel of the SSH connection. STDIN and STDOUT are read from every container, based on which test case validations occur and points are calculated.

The conclusion of the paper states that the issues of security and isolation are successfully addressed by the suggested system but that there are also limitations with the plugin subsystem of the implementation.

2.4.2 Submitty - Comparing Jailed Sandboxes and Containers

A paper by Peveler, Maicus, and Cutler [14] compares jailed sandboxes with containers in the context of an autograder system. The paper also states early on that virtual machines could technically be used for this purpose however, they are found to be extremely expensive in terms of resources for running student application instances. This is due to the fact that virtual machines have to emulate an entire operating system using considerable resources from the host system such as CPU, memory, storage and network resources. Additionally, virtual machines often take more than a minute to boot which is considered too long in this context.

The authors describe their autograder system called Submitty in which, through the autograding pipeline the instructor is able to specify whether he/she wishes to test student submissions using the jailed sandbox approach or the Docker container approach. Jailed sandboxes run on the host system and use a security model as to separate processes to access only specific directories. This separation is implemented in Submitty by defining untrusted users that each have access to files in a single, specific workspace directory.

The autograding process begins by a higher permission level user copying the student submission into an untrusted user workspace wherein the code is compiled and the test cases run as the specific untrusted user. The limitations over actions and resources for this user are implemented by defining a whitelist of allowed system calls using the Linux seccomp library. To protect against CPU and RAM hogging and other malicious attacks RLimits are placed on the execution of the process that essentially immediately kill the process once they sense violation of defined limits. The test running phase is followed by validation of the test cases and by outputting the test results from this directory.

The drawbacks of jailed sandboxes are mentioned again which mainly have to do with difficulties with isolation of student code from other resources of the host machine. This could be dealt with by more complex virtualisation of the filesystem, this approach however, is deemed mostly too complicated and is also exacerbated by the potential need for student code to utilise external modules.

After introducing their jailed sandbox methodology the authors make the case for using Docker containers pointing out that they occupy a solid middle-ground

between virtual machines and jailed sandboxes due to their ability to isolate more than sandboxes, but provide a more lightweight virtualisation option than VM-s. Containers also provide virtualisation on the operating system level, allowing them to use resources of the host and its kernel. Programming language independence is secured by utilising specific images for specific languages while networked and distributed programs are also possible to test and grade. Instructors are able to use their own specific Dockerfile, mix and match Dockerfile components using Submitty's command-line interface or use a provided existing image.

The process of autograding using Docker in Submitty uses the same untrusted user workspace concept as the jailed sandbox method with additional steps such as creating a container, mounting it to said workspace directory then destroying it after code is compiled and test cases are executed and validated. Implementation of the jailed sandboxes are still used inside of containers because even though Docker is able to limit CPU and memory use of a container, RLimits are still needed to contain file sizes created by runaway processes. The authors go into detail about the security advantages of using Docker containers for running student submissions wherein they also note that the untrusted user workspace concept is still required to be maintained within containers due to the default container user being root.

In-depth performance analysis is performed at the end of the paper where the authors compare containers to jailed sandboxes in terms of resource usage and time elapsed for various parts of the autograding process. It is concluded that for normal workloads the two methods do not differ that much in terms of these conditions, although jailed sandboxes tend to use less resources and take less time to perform the tasks. The small wait time produced by the Docker approach is concluded to exist due to time taken to create and destroy individual containers. The same test is then run with a massively increased load whereafter it is seen that Docker performs visibly worse in terms of resource usage as well as runtime. It is however, important to note for the Docker approach that a somewhat catch-all, general image was used for the tests, meaning that with more specific, smaller images performance is expected to increase.

The conclusion of the paper states that although there were some differences in performance between the two methods, these are deemed mostly negligible in a regular workload. The authors also emphasise the advantages of utilising Docker images which mainly lie within security and isolation contexts. Improving start up

times for containers is suggested by the authors by defining an existing container pool from where the application is able to pull a container and immediately use it instead of having to create it from scratch. The possibility of decreasing destruction times is also discussed by implementing a container garbage collector that could periodically clean up unnecessary containers.

2.4.3 Submitty - Networked Applications

Especially relevant to my thesis is the idea of autograding student submissions in networked containers discussed in a paper by Maicus et al. [7] also demonstrating their methods with Submitty as SUT.

Their methodology involves inserting an additional node into the private network called the router. The existence of such a node on the network means that not only are instructors able to limit network communication to specified protocols and evaluate student code in a simpler fashion but also that their ability to easily customize network rules is greatly increased. Test cases can also be run in routerless mode in case the additional node proves to be too much of an intrusion. The router itself is invisible as far as the other agents are concerned however, all communication within the network passes through it.

In order to ensure that student code does not interfere with other containers in a malicious way and that the communication flow is not disturbed a concept called Docker network aliasing is taken advantage of. This concept allows developers to define aliases for containers in a Docker network which then serve as the point at which other containers can communicate with the aliased container. The authors explain that they were able to create as many networks as there are containers and have only the agent and the router in any given network, giving the router the alias of the container that the agent wishes to communicate with and the agent an alias that confirms its identity to the router.

To ensure the safety of communication through the router necessary network information such as the addresses and ports of hosts on the system is provided by an automatically generated file that the student code will have to parse in order to establish necessary connections. This adds a small constraint to the system, as port assignments have to be unique within the system.

Upon initialization, the router creates a switchboard dictionary which maps a

port to a sender, recipient, and connection type. It is through this switchboard that instructors are able to modify the router. In order to make configuration of test cases as simple as possible, default values are defined for a JSON configuration file which hold only five parameters responsible for the existence of the router node, the names of the containers, the images for the containers, the connections between the containers as well as the Unix commands to be executed sequentially within the containers. For each test case, new Docker containers and networks are created. As a result, each new test case can test different types of containers, network sizes, and network conditions after which, the output of the run is stored in an individual folder which is then given to the existing validation pipeline.

In Submitty, there are multiple ways of evaluating student submissions. In the simplest test cases, instructors do not even have to interact with student code, as it is often enough to initialize a student node with some known state, either through command line arguments or an initialization file after which the node is capable of achieving the goal of a test case by itself. There are three main types of instructor interactions when looking to evaluate student code. We can imagine the network as ground for a service-client architecture, where there might be instructor modules functioning as servers which the student's code has to interface with. In this case the standard output of the instructor nodes can directly be used for validation and grading.

The opposite of this dynamic can be a different way for instructors to grade student solutions, as students might have to implement a complete system functioning as a service that the deployed instructor client nodes interface with periodically. In a more complicated test case there might be a multithreaded execution of commands from an instructor node to test concurrency of the system.

A more direct approach of instructor interaction foregoes the use of instructor nodes and instead allows for instructors to dispatch strings to the standard input of containers. Submitty allows for four types of this dispatch, including delay, standard input, start and stop. These actions are performed sequentially and are able to target one or more containers. Start and stop are used to disconnect nodes from the network, thus simulating system faults, delay helps in ensuring correct timing, and standard input allows strings to be piped in to given containers.

The authors describe performing a case study of their system capabilities and conclude that it "capable of successfully evaluating the challenging testcases from

RPI's Distributed Systems and Algorithms course". It is also noted that the complete test suite took less than two minutes for all student solutions, most of which consisted of initialization time because student nodes were given five seconds to fully initialize per test case so they do not receive inputs before they are fully operational.

Finally, limitations are discussed, including the fact that the current router implementation expects every network connection to use a unique port, as well as the possibility of reducing redundancy in defining configuration files and developing a web GUI for autograding configuration.

Chapter 3

TMS

TMS (*Task Management System*) is the assignment-management and plagiarism-detection autograding system of the Faculty of Informatics, Eötvös Loránd University. It is an open-source project written in PHP using the Yii 2 framework with a TypeScript-React frontend that interacts with the backend following the REST architecture. As an atuograding system, the main job of TMS is to automatically test and grade student assignments based on predetermined criteria. Using such an autograding system is essential in IT educational environments as to improve efficiency, consistency and accuracy of grading assignments, as well as to alleviate some pressure and workload from the instructors. The project can be found at `https://tms-elte.gitlab.io/`.

There are currently two types of applications that students are able to submit to TMS as submissions, console applications and web applications. TMS supports automatic testing and evaluation of both of types [15, 16].

The main focus of my thesis is the examination and implementation of automated testing and grading functionalities for networked applications, for which it can be advantageous to take a closer look and analyze the process of automated evaluation of console applications. The workflow of the current autograding capabilities of the system is implemented by a module of TMS called the AutoTester.

The process begins by an instructor navigating through the TMS interface to enter a group of students he or she is responsible for and creating a task for the given group. When creating a task, the instructor is able to set a variety of parameters for the assignment, such as the name, category, description and availability of the task, soft and hard deadlines as well as a password to access the task with. All of these

options are easy to edit later on, if needed. After the task is saved to the database it can then be seen and interacted with on the page of the corresponding group. Clicking on it gives the instructor additional options to monitor student submissions and configure the automatic evaluator, as well as extra possibilities such as static code analysis and a code comprehension and visualisation tool.

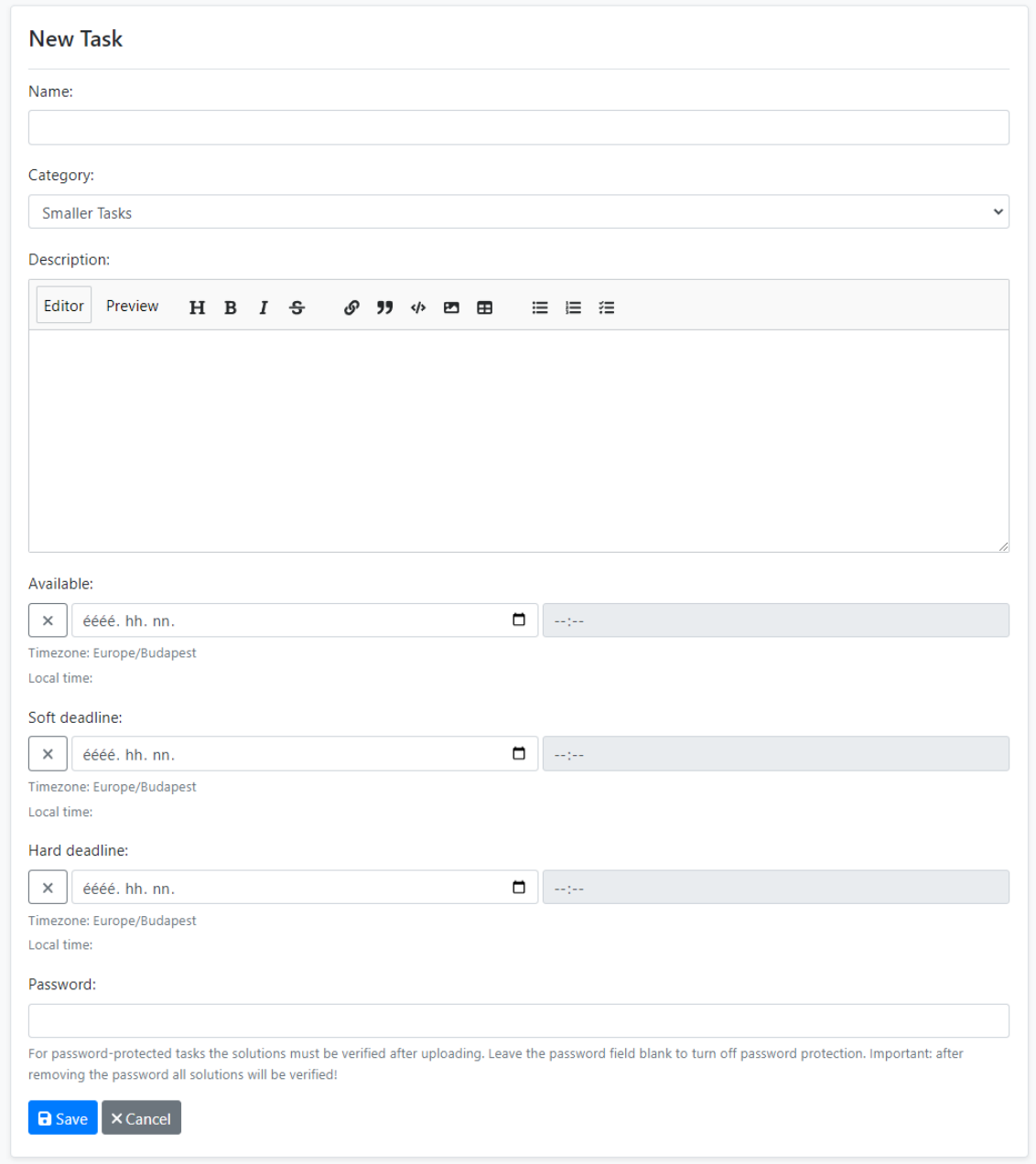

Figure 3.1: Creating a new task

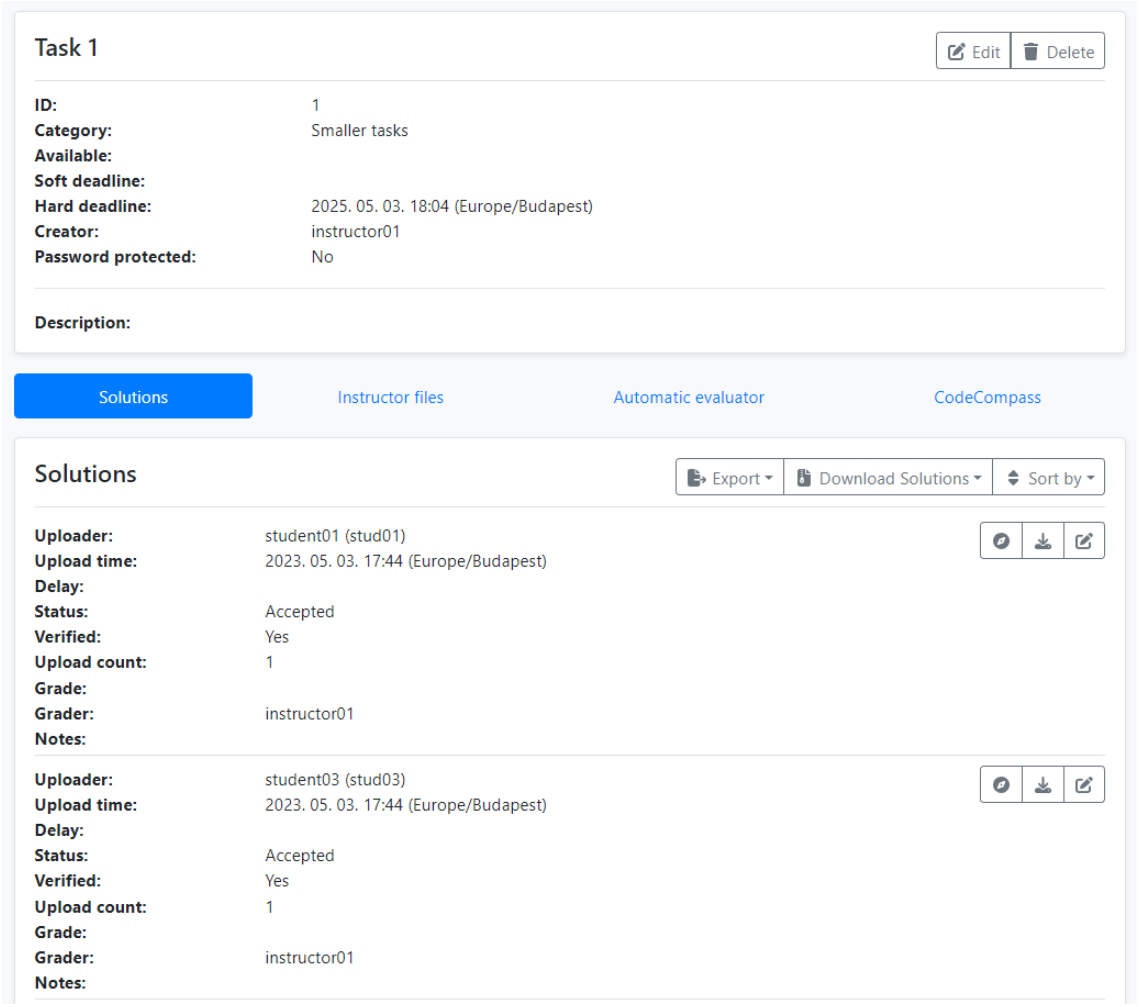

Figure 3.2: Interface of a task with solutions

Navigating to the Automatic evaluator tab shows us the main configuration options for the environment and automatic tester. The environment is used to set up the Docker container, as evaluation of even console applications occurs by using containers. The options on this form include the operating system the submission would have to run on, the Docker image, as well as a possibility for the instructor to upload his or her own Dockerfile. The automatic tester section on the other hand is responsible for the specific instructions the submission is run by, as well as the test cases that serve as input for the student's code. In order for these functions to operate, the instructor has to configure the compile and run instructions for the code in addition to each test case that contains the arguments, input and output fields. TMS additionally provides templates for and easier and quicker way to configure the evaluator and automatic tester.

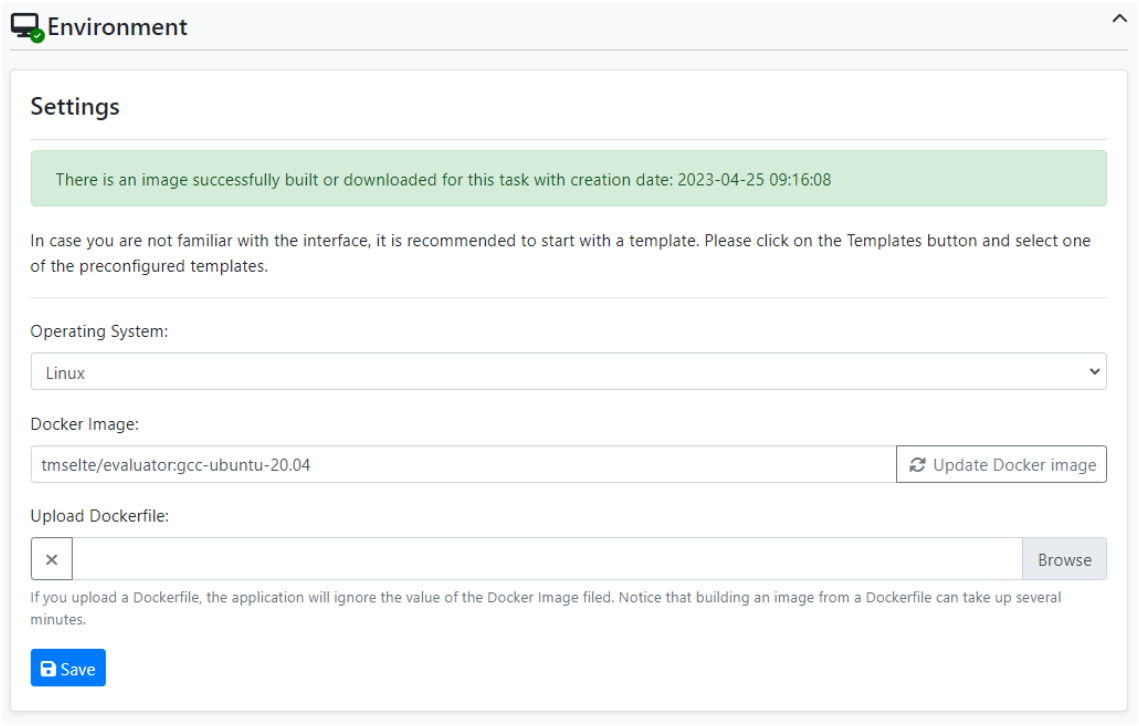

Figure 3.3: Environment settings of the automatic evaluator

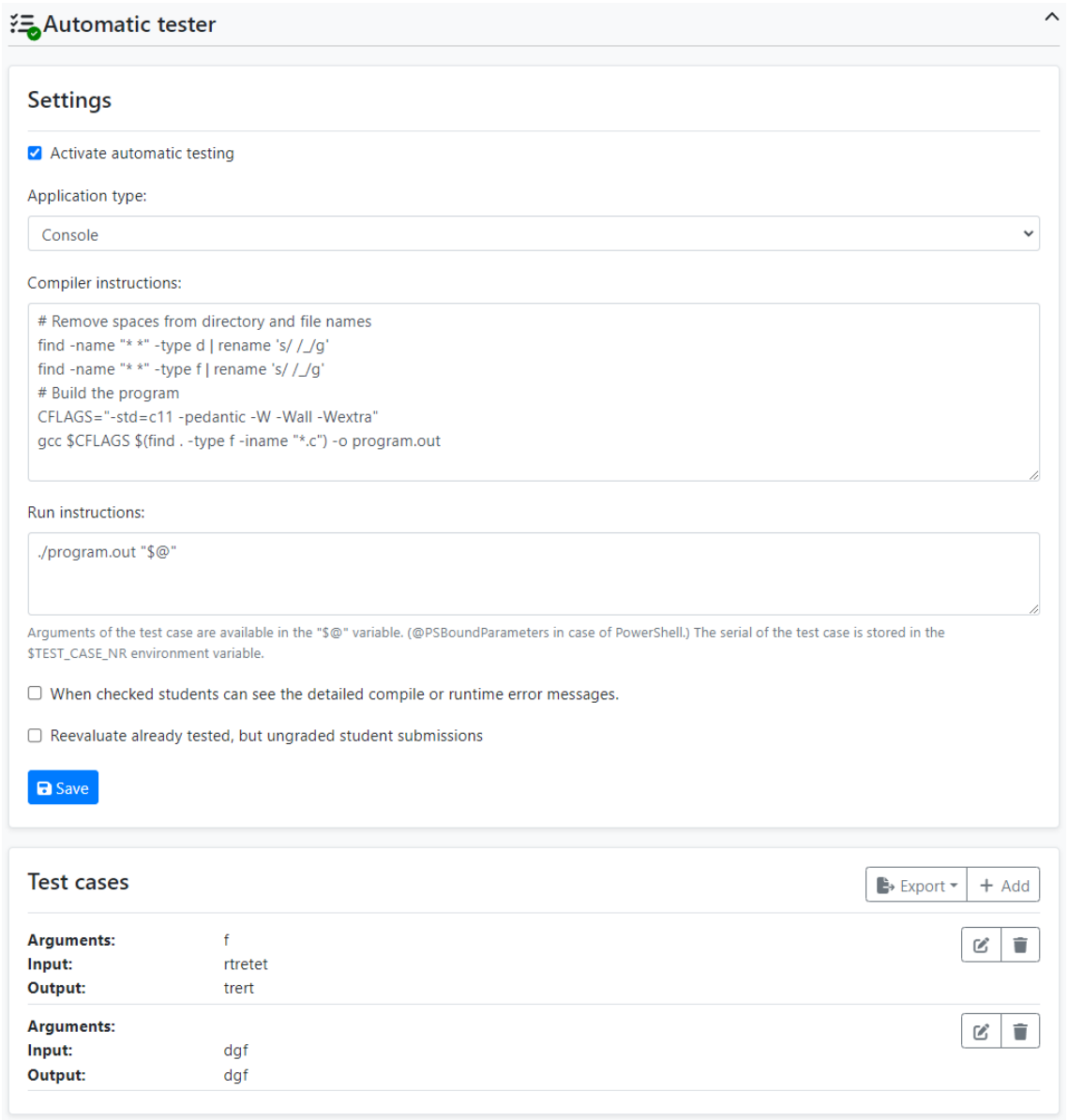

Figure 3.4: Tester settings of the automatic evaluator

For a single test case, the arguments field specifies the console arguments to the process, while the input field is passed to it through the standard input. The output field describes the expected content of the output stream. There is also the possibility for instructors to export and import these test cases for convenient and quick testing purposes. In the case that the instructor deems that some specific miscellaneous files need to be given to the students or copied into the container in order to help out testing, he or she has the opportunity to upload multiple of these files. All of these options are editable so long as the date is not past the task's hard deadline. Whenever the student submission is uploaded, the test environment and the automatic tester are configured and saved and the instructor's miscellaneous

files if necessary are also uploaded, the testing process may begin.

The backend handles the problem of testing submissions using what is know on Unix systems as cron, which is a utility that allows for scheduling jobs. The job for automatic testing is scheduled to run periodically, checking if there are new untested solutions. If there are, the testing process is initiated using all of the parameters set by the instructor. If the instructor chose to upload his or her own Dockerfile, it is compiled into a Docker image, if not, the given Docker image is used to create and run the container. The uploaded student submission - along with the InstructorFiles if there are any - is copied into the file system of the container, then compiled and ran in there with the given arguments and inputs that were specified in the current running test case. The output of the run process is then compared as a string with the expected output value, and the appropriate result messages are set.

The testing process finishes by parsing the results and setting a passed or failed value to the student submission that the student is able to see wherever he or she uploaded the submission. There are also various ways the testing can fail, and so, multiple types of failure options can be set including initiation failure, compile failure, run failure and just simply test case failure. This, in addition to the option for the instructor to set it so that students are able to see detailed error messages makes it so that the students have the option to monitor exactly at what point their submission has failed the tests, if it did indeed fail.

Chapter 4

Methodology

In this chapter I intend to go over the methodology of describing, analysing, comprehending and constructing an architecture for an automated tester that is able to run networked applications and evaluate them based on predefined criteria. The source of the basic problem is the following: in order for networked applications to run and be evaluated, a different environment to non-networked applications is necessary with additional configuration. The reason this is the case is because a networked application by definition has to perform some sort of communication over a communication channel (the network) with another networked application.

Indeed, both sending and receiving data over a network makes an application networked implicitly. For such applications to operate in a correct manner, it is imperative to ensure the order, manner, security and precision of communication, since an error in any of these categories results in erroneous operation and most likely an undesired result. This chapter goes into detail as to what is required for networked applications to communicate properly, thus ensuring the possibility to run and automatically test them.

4.1 Research Process

The research process for the topic at hand primarily followed a qualitative research methodology approach, wherein the following steps were employed:

1. Determine initial problem.

2. Lay out the requirements for any given solution to the problem.

3. Perform research along the lines of the requirements by analysing scientific research papers dealing with the topic.
4. Collect conclusions from research sources.
5. Compare conclusions from research sources in terms of being able to fulfill requirements.
6. Work out the details of the solution and document them.
7. Make sure all requirements are dealt with and the solution makes sense.
8. Implement solution to verify correct behaviour.

4.2 Problem Description

The initial problem was determined as motivation for the thesis paper due to demand within the Faculty of Informatics, Eötvös Loránd University for running and testing networked applications for the Telecommunication Networks subject. TMS, the assignment management and autograding system in place currently has no option to run and test these kinds of applications, as discussed in the above chapter, and so, the topic for my thesis was offered.

4.2.1 Networks and Agents

In the context of networked applications, a network refers to a system that allows multiple devices or entities to communicate and share information with each other. These entities are called agents, all of which possess their own unique identities and can interact with other agents to exchange information or collaborate on tasks. It is important to note that the identity of an agent is always in relation to the network or networks it currently inhabits, meaning that the only way for an outside actor to reach this agent is by having access to the given network, as well as knowing the identifier of the agent on that network. Knowing this, it becomes apparent that a large part of the problem lies in making sure the identities of agents are unique and accessible to the right entities by making sure that the network itself is configured properly.

4.3 Requirements

There is much room to discuss general requirements for autograders, however, this section is going to focus exclusively on requirements for the specific function of our autograder that my thesis intends to introduce: automated testing of networked applications.

Scalable network size : When testing networked applications it is of course very important to discuss the actual network that the components will run and communicate on. The network has to be reasonably sized, being able to accommodate all student and instructor code instances, called agents. The minimum number of agents is of course two, otherwise there would be no network communication to speak of. The maximum size for student submissions plus instructor nodes should never really reach over seven or eight agents, even for the most complex assignments, at the Faculty of Informatics, ELTE for example, the Telecommunications Networks course exclusively deals with network agents numbering two to three, while RPI's Distributed Systems and Algorithms course handles as many as eight student nodes [7].

Easy specification options : Network specification should be as simple as possible with sensible default values so as to ensure that instructors are able to construct simple and more complex examples as well.

Data flow : Facilitation of input and output channels is also required to be the input for the validation process.

Language support : TMS currently supports multiple programming languages, these are of course expected to be available in this functionality, as well as some popular communication protocols such as UDP and TCP.

Feedback system : The system also needs to provide students with meaningful and as quick as possible feedback to debug and fix errors, even through multiple submissions.

Security and isolation : One of the most important things to look out for when testing networked student submissions is security and isolation of components, as there might be malevolent or just mistakenly harmful submissions that

would access and execute processes outside of their designated range. As discussed at length in the Related Work chapter, the best way to achieve such levels of isolation seems to be by using Docker containers.

4.4 System Design

In chapter 2, I have analysed multiple different scientific papers discussing the possibilities of automatically testing some variation of a distributed system. Comparing these conclusions drawn from the scientific papers and the requirements specified above, I have selected to construct an architectural overview.

4.4.1 Choice of Virtualisation

In order for a networked application to be ran, it is required that the communication between the components of the application (agents) is simulated in some way. Of course one could technically run all agents on the same host computer, but that would not realistically emulate the communication architecture of a real, distributed, networked system.

To this end, I have taken an analytic approach to examine ways to virtualise an environment wherein these agents could run and communicate amongst each other, identically to an actual distributed system. Most of these options are discussed in chapter 2, based on which I have chosen containerisation, specifically Docker as my choice of virtualisation. Containers in general, but especially Docker are a great option for this type of problem for multiple reasons:

Portability : A container is a lightweight and standalone executable unit that packages software, along with its dependencies and configurations, into a single self-contained unit. It provides an isolated environment where an application can run consistently across different computing environments. In Docker, these containers encapsulate an application and its dependencies, including libraries, frameworks, and system tools, into a single package called an image. The image contains everything needed to run the application, ensuring that it behaves consistently regardless of the host environment. This is required in order to facilitate platform-independence, and make the solution more widely usable.

Isolation : Containers provide a level of isolation between applications and the host system. Each container runs as an independent process, isolated from other containers and the underlying host. This isolation prevents conflicts between applications and enables them to run reliably and securely. This feature is especially important in the context of my prototype implementation, since autograders also need to make sure potentially malicious or unintentionally harmful student code does not receive the chance to interact with components outside of its designated scope.

Efficiency : Containers also leverage a shared operating system kernel, which makes them lightweight compared to traditional virtual machines. Containers use the host system's kernel, eliminating the need for a separate guest operating system for each container. This efficient resource utilization allows for running multiple containers on the same host without significant overhead.

Network Options : Imperative to the topic of this thesis, containers also provide several network options that enable communication between agents, as well as connectivity to external networks and services. In Docker for instance, developers are able to construct a network of one of three different types, connect a number of containers to the network then monitor communication, as the agents, now represented by isolated containers send and receive data amongst each other within the network.

Support : Containers have received significant support and popularity within the software development and IT communities as they have gained steam over the traditionally used virtual machines. This means that not only does the community of developers actively contribute to and expand the ecosystem of containers, but also that various industries have and are adopted containers as a way to develop, test and deploy code leading to an even larger gain of popularity. For Docker in particular, Docker Inc. has been actively driving the evolution of container technology and introducing new features and improvements, ensuring future support for the technology.

TMS : In addition to widespread support and an active developer base for containers, TMS also already supports Docker containerisation of various types of applications, along with helper libraries.

For these reasons that I have elected to use containers and specifically Docker containers for my solution to the networked application problem. In the following part I will examine in what way exactly does the choice of containers, network and agents address the requirements listed at the beginning of chapter 4.

To start with the scale and scope of the network we must note that there is no actual hard limit on the amount of containers on could theoretically connect to a Docker network, which essentially means the only limiting factor in scale is the infrastructure itself, which are the resources of the host system and network bandwidth. In the context of automatic grading of student submissions, this means that the estimated maximum number of agents for a given task - which is around eight agents - is easily accommodated for on the network.

For Docker containers specifically, language support is ensured by utilizing Docker images, plenty of which can be found and pulled from public repositories, or even built from a Dockerfile, giving the user essentially almost universal access for whichever programming language he or she wishes to write applications in.

One of the main features of containers is isolation, which allows for extremely secure running of applications, ensuring no malicious or unintentionally harmful code interacts with entities outside of the container.

4.4.2 Input / Output Specification

In the following section I intend to discuss the necessary specification fields for running and testing networked applications. These parameters provide sufficient options for the end user for testing such applications, and are easy enough to understand, thus fulfilling another requirement of the solution. In the proposed model, the user will have to make specifications at two different points in the application.

Container Configuration

On the one hand, the agents themselves require specification for them to be able to run within the containers. The user must be able to dynamically add, delete and edit agent configurations on demand. Depending on the programming language of choice, the code needs to be probably compiled and definitely to be ran. By default this means that the system needs to account for the compilation instructions which can be empty or specified and the running instructions which have to be specified.

However, this will of course only work, if the necessary environment in the container is in place to be able to perform compilation and execution. Specifying a Docker image for each container solves this issue. Finally, the name of each container also needs to be given in advance by the user, because this name will be identical to the host name of each agent within the network, allowing for code within the containers to communicate to each other using this host name.

Networked Test Cases

On the other hand, for the testing process, additional parameters are needed by the end user, so that these test cases can be run, and their output evaluated. One test case is specified for each container, and there can be multiple amount of test case configurations for a given application. This means, that if there are an N number of agents in a network of the application, for each test configuration there will be an N number of test cases in the given configuration, which is also the first parameter the end user is obligated to specify. As for the second parameter for the test cases, in case any single application agent requires console arguments, these can be specified for the given test case, being concatenated to the running instructions of the container code. There is also an option for providing the input field parameter, which is going to be put onto the standard input stream of the application, as well as the non-optional parameter of the output field, which describes the expected standard output for the given agent. Finally, the identification number of the container also needs to be provided so as for the tester to recognise which container the given test case belongs to. These test cases must also be free to be added, edited and deleted by the end user.

Order of Execution

There is one additional parameter, specified for both the agents and the test cases. This is the order number of the container, which refers to a priority list in whose order the containers will run. The "identification number of the container" phrase in the previous paragraph refers to this field, it is also required for each container to be specified. The existence of this parameter is required for a complex reason, which I will examine and describe in detail during the following section.

4.5 Workflow

4.5.1 Occurrence of Events

This section showcases the structure of the proposed model by providing a list of events that will occur as the data propagates throughout the system.

1. Parameters are configured separately for the networked agents and test cases, then they are saved into the database.
2. Files containing the actual code that will run inside various containers are uploaded and extracted into a temporary directory.
3. The running process is initiated.
4. A network is created with the appropriate configurations.
5. For each agent, a container is created.
6. For each container, the corresponding files are copied into a directory within the container's operating system.
7. For each container, the same mount is mounted to a specified directory of the container.
8. For each container, using the appropriate compilation instructions, running instructions, command line arguments and standard inputs the applications are executed.
9. The standard output of these applications is directed to specific text files inside of the mounted directory.
10. After the last execution command of the last container in order is finished, the containers are shut down.
11. The network is shut down.
12. The testing process is initiated.
13. The text files containing the output from the containers are parsed and compared as strings against the expected output from the given test configuration test cases.
14. The results of the comparison are displayed to the end user.
15. The evaluation process is complete.

4.5.2 Challenge - Network Configuration

There are multiple types of networks one is able to create as a Docker network using network drivers. The basic driver for docker networks is 'bridge'. Containers connected to this network can communicate with each other using IP addresses. The bridge network provides basic network connectivity but doesn't expose containers' ports to the host system or external networks by default. This type of network can also be created by the user.

Another option for creating a docker network is to use the 'host' driver. This makes it possible for containers to run on the host network directly. When a container uses this type of network, it shares the network namespace with the host, which means the container uses the host's network stack and has access to the host's network interfaces.

A third possible driver option is the 'overlay' network driver, which connect multiple Docker daemons together and enable swarm services to communicate with each other.

There are also other choices for this task such as the 'ipvlan' and 'macvlan' network drivers which grant the user control over IPv4/IPv6 and MAC addressing respectively.

My choice for the task at hand is the bridge network driver, for multiple reasons, but the main one being that the functionalities of other options are either irrelevant or overkill for the particularly simple task of creating an isolated private network, attaching containers to it and having them run and communicate. Bridge networks also massively help with the issue of isolation and security, well emulating a real distributed system by having the network be private and forbidding containers access to the host machine's interface.

After selecting the appropriate network driver, the rest of the network configuration phase is relatively straightforward. The driver is set, a random name for the network is generated for identification, the network is created, and the containers are attached. The only thing to pay attention to at this point in terms of the network, is to wait until all containers on the network are shut down before the network is deleted.

4.5.3 Challenge - Output Gathering

This section will go into detail and discuss the process of acquiring the output of each agent container for evaluation. In order to get the standard output or even potential errors during compilation, execution or run time of an agent application, a mount is used. In Docker, a mount is a way to make a file or directory from the host machine available within a container or between containers. It establishes a connection between a specific location on the host machine's filesystem and a specific location within the container's filesystem. Docker allows the option to specify three different types of mounts: bind mounts, volumes and tmpfs mounts.

Bind mounts are created by directly specifying a directory on the host machine to be mounted into the container. The host directory and container directory are explicitly linked, and changes made in either location are immediately visible in the other.

Volumes are created using Docker's volume management system. They provide an abstracted way to manage storage for containers. Docker takes care of creating and managing the volume, which has a specific location on the host filesystem. Containers can then reference the named volume by name and access its content.

Tmpfs mounts are created in-memory and do not involve the host filesystem. They allow for the creation of a temporary filesystem within the container, residing in the host's memory.

My choice of mount for the task is the bind mount, although a volume could certainly be used as well. The mount is created and configured in advance of the running process, just after the network is created. Besides setting the type of mount to be 'bind', there is also a need to set the source and destination directories for the mount. The source is set to the already created temporary directory on the host machine, while the mount is set to be a specific directory on the container's filesystem, that will come into existence once the appropriate files are copied onto the given agent's container.

As additional preparation before running the networked application, there are four text files created for each agent's container on the host machine, within the bind mount's directory. These text files are empty at this point, but will possibly contain the following information after the execution of the networked system:

- The code compilation's error code.

- The code compilation's error message.
- The execution's error code.
- The execution's standard output and error message.

Standard output and standard error stream are both directed to the appropriate files in the bind mount on compilation and execution of the application. It is certain that all output will be visible inside of the text files in the bind mount, because of the way the execution of commands happens inside of the containers.

As mentioned above, the end user of the evaluator is obligated to specify the order, in which the code inside of each container runs. This means, that - unless specified incorrectly - it is always known, which container is created and ran last. There is a flag for this, which is flipped to be true, whenever the last command execution of the very last container happens. After this, it is safe to delete the containers and the network, because the outputs and possible errors of the execution processes have already been written inside of the bind mount at this point.

Chapter 5

Implementation

As mentioned in chapter 3, the backend of TMS is developed in PHP using the Yii 2 framework, while the frontend is developed in Typescript-React. Interaction between these two layers happens through a REST architecture, with the application structure conforming to the MVC (Model-View-Controller) software design pattern.

5.1 Backend

The process of implementing the backend for the automated testing of networked applications is the following:

1. Necessary tables and foreign keys are added or modified in the database.
2. Main logic written for running networked applications in containers.
3. Logic written for testing networked applications and setting correct test results.
4. Necessary API endpoints for configuration of automatic evaluator added.
5. Evaluation process integrated into existing cron job.

5.1.1 Database

TMS uses migrations to interact with the database. For simplicity's sake, there is only one new migration created which contains all the necessary changes to the database for the automatic evaluator to run on networked applications. The following tables are created or modified to represent the new entities:

Figure 5.1: Newly added tables and their connection to the tasks table

As for the table representing instructor agent files, we can note that there was no need to add the same for student files, since that feature is already part of TMS. These uploaded files by the instructor and the students operate in the exact same way, which is to say that both instructors and students have to upload one single .zip compressed folder file, which contains all of their agents in subdirectories differentiated by numbers corresponding to the order numbers of the containers.

For all three tables, model classes were written to represent them, along with resource classes for interactions inside of the API endpoints.

5.1.2 Runner and Tester

With the necessary database modifications in place, development of the main business logic for the networked application runner and tester is initiated. The implementation plan for this part of the code was the following:

1. Implement a runner class responsible for constructing the environment in which a networked application is able to be ran based on the given configurations and running the application.

2. Implement a tester class responsible for receiving one test configuration and running the system with the parameters configured inside of the test cases of this test configuration.

3. Integrate the call to the tester class into the existing command controller class responsible for checking for untested student submissions and running and evaluating them when called by the cron.

Runner

As for running networked applications, the runner class is inserted into TMS. This class is responsible for constructing the environment where the student and instructor agents are able to run and communicate with each other. The runner class receives two important points of data, namely one test case configuration, and the model representation of the student submission. The class is structured in a way that after instantiation, only one public method needs to be called to initiate the running process. This command is called 'run', and it uses numerous other private methods within the class to perform the delegated task. Due to this dynamic, the easiest way to understand the workings of this class is to explore the workflow of the run method:

1. Prepare a temporary directory for the agent files to be extracted into.
2. Create and initiate the Docker client.
3. Create the Docker network with the configurations specified above.
4. Create the bind mount with the configurations specified above.
5. For each agent:
   (a) From the temporary directory, copy the appropriate agent's files into an agent directory which will be copied onto the respective agent container's filesystem.
   (b) Do the same for the miscellaneous instructor files.
   (c) Prepare, construct and save the compilation instructions for the agent's code.
   (d) Set the running parameters (console arguments, standard input) for the agent using the respective test case.
   (e) Prepare, construct and save the running instructions for the agent's code.
   (f) Build the container for the agent.

   (g) Connect the container to the Docker network.

   (h) Copy the required files onto the container's filesystem, compile and execute the code inside the container.

   (i) After execution, delete the agent directory for the agent.

6. After the last command execution of the last container, shut down all containers, the network and delete the temporary directory.

Error handling is performed throughout the class, checking for errors in the initiation process as well as the build and execute phases of the containers.

Dockerisation in TMS is performed using the Docker PHP library functionalities, many of which are wrapped into helper classes already implemented in TMS. These helper classes are used by the runner class not just to build containers, but also to run them, interact with them and shut them down. Implementing the running process of networked applications involved expanding these classes as well.

After the execution of the containers commences, the results of the execution process can be found inside of a temporary directory inside of the student submission's folder on the host machine. This is where the tester class will pick up and continoue the process of evaluating networked applications.

Tester

The tester class - similarly to the runner class - has only few public methods, one of which contains the main steps of the testing business logic. Instantiating the tester class also passes one test case configuration and the model representation of the student submission to the object. The test method is then called, which performs the following steps in order to test a run result.

1. Instantiate the runner class with the test configuration.

2. Call the run method of the runner object, thus beginning the execution process.

3. After the run finishes, copy the results from the temporary directory of the runner to the student submission's directory.

4. Check the results against the test case configuration and collect them into an array.

The result checking process starts out by iterating over each agent and getting the file contents for it. There are four result files for each agent participating in the task, these contain the compilation error code, compilation error message, execution error code and finally, the execution standard output and error message written into one file. The contents are then cleaned by removing whitespaces and break lines, after which the string comparison against the expected results can commence. The result of the comparison is put inside of the results array, for which there is a getter method written in the class.

Scheduling

As stated above, there already exists a scheduled command class for checking if there are any uploaded, untested student solutions and running the automatic evaluator on these submissions. Part of my implementation was to expand this command controller class so that it would account for networked applications and call a separate function that could then call the tester class for each test case configuration and perform the automatic evaluation on the networked student submission. The expanded command now performs the following steps when called:

1. Check if automatic evaluation is enabled.
2. Check if task, student submission and running instructions for the submission exist.
3. Determine application type, if 'Networked', collect all networked test case configurations.
4. For each networked test case configuration:
   (a) Instantiate the tester class and call its test method.
   (b) Get the results from the test class and write them inside of an array.
   (c) Parse the array and set the appropriate test result values and messages for the student submission.
   (d) Save these values.

In the case that there is any kind of failure in the test process, all of the execution results are concatenated onto the same error message that is visible for the student.

Knowing the output of each agent makes it easy to recognize where the error is inside of the solution.

5.1.3 API Controller

There have been a total of 12 new API endpoints added to TMS in the form of two new controllers and modification of an existing controller. In this section, I'll be listing the new API endpoints of TMS along with the type of the request and functionality within the automatic evaluator. The two new controllers are the `NetworkedAppController`, and the `InstructorAgentsController` responsible for managing networked containers and instructor agent files respectively, while the `TestCasesController` has been modified to accommodate networked test cases as well. This is required, because unlike for regular console applications, for networked applications, there is an option to specify multiple test case configurations consisting of test cases for each individual container agent. All of these controllers extend the `BaseInstructorRestController`, which in TMS allows for endpoints to access instructor-specific functions in the autograder.

The following is a description of the newly added API endpoint calls represented as CRUD models:

NetworkedAppController: This controller's objective is management of networked container instances which are equivalent to networked agent configurations.

- **Index** is a GET HTTP request method responsible for displaying all networked container configurations for a given task.
- **Create** is a POST HTTP request method responsible for creating a new row of networked container configurations.
- **Update** is a PUT/PATCH HTTP request method responsible for editing certain fields of an existing networked container configuration row.
- **Delete** is a DELETE HTTP request method responsible for deleting a networked container configuration row.

InstructorAgentsController: This controller's objective is management of instructor agent files.

Index is a GET HTTP request method responsible for displaying all instructor agent files for a given task.

Download is a GET HTTP request method responsible for downloading an instructor agent file to local disk.

Create is a POST HTTP request method responsible for creating and uploading a new instructor agent file.

Delete is a DELETE HTTP request method responsible for deleting an instructor agent file.

TestCasesController: This controller's objective is management of networked test cases.

Index-Networked is a GET HTTP request method responsible for displaying all networked test cases for a given task.

Create-Networked is a POST HTTP request method responsible for creating a new networked test case for a given task.

Update-Networked is a PUT/PATCH HTTP request method responsible for editing certain fields of an existing networked test case.

Delete-Networked is a DELETE HTTP request method responsible for deleting a networked test case.

Some of these endpoints such as the create and update method calls of the `NetworkedAppController` and `TestCasesController` classes utilize scenarios of their respective model classes. Scenarios offer a way to specify different sets of validation rules for different scenarios or use cases within the application. These are used inside of the controllers to limit which fields can be updated and also to increase clarity within the code.

5.2 Frontend

The frontend implementation follows much of the same line of the backend implementation in a sense that the various networked container configurations and networked test cases all need to be input through the frontend of TMS. These are set by the instructor on the automatic evaluator tab of the specific task, along with

the field where instructors are able to upload the code for their agents. This uploading process is identical in structure to the students uploading their solution, because both uploaded directories contain the agents separated by subdirectories.

There have been changes to four main parts of the automatic evaluator interface of a task:

- Firstly, the option to set the application type to networked is the catalyst to display the rest of the functionalities added.
- Once this is set, a list of the networked container configurations becomes visible with the instructor being able to add, edit and delete individual agent configurations.
- Below this is an upload field to upload the instructor agents. The uploaded file is displayed below this option.
- Finally, there is an additional list, identical in structure to the previous one, where instructors are able to add, edit and delete networked test cases for the application.

The lists mentioned above utilize modal windows when adding or editing an element for clarity and easier use. The components of the frontend call on the API endpoints using mutation hooks.

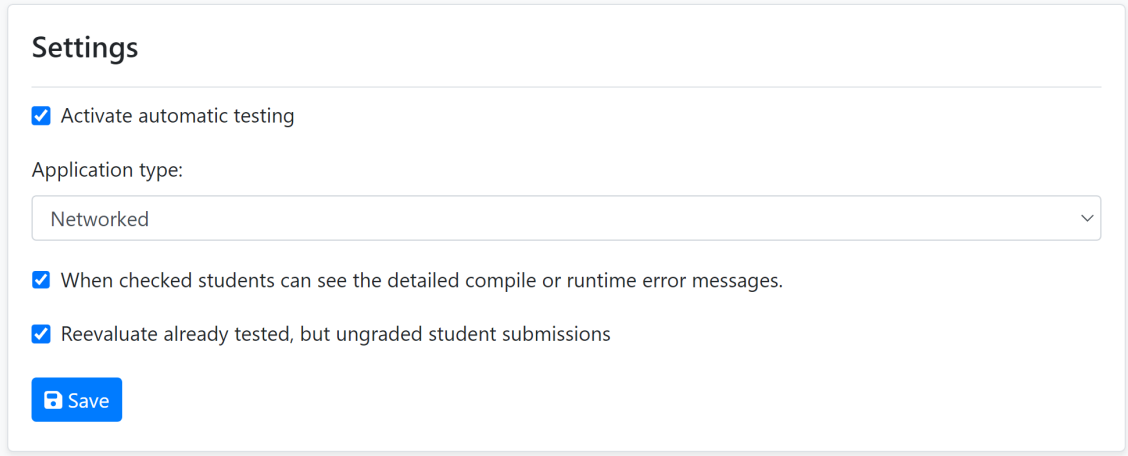

Figure 5.2: Application type is set to networked in the settings

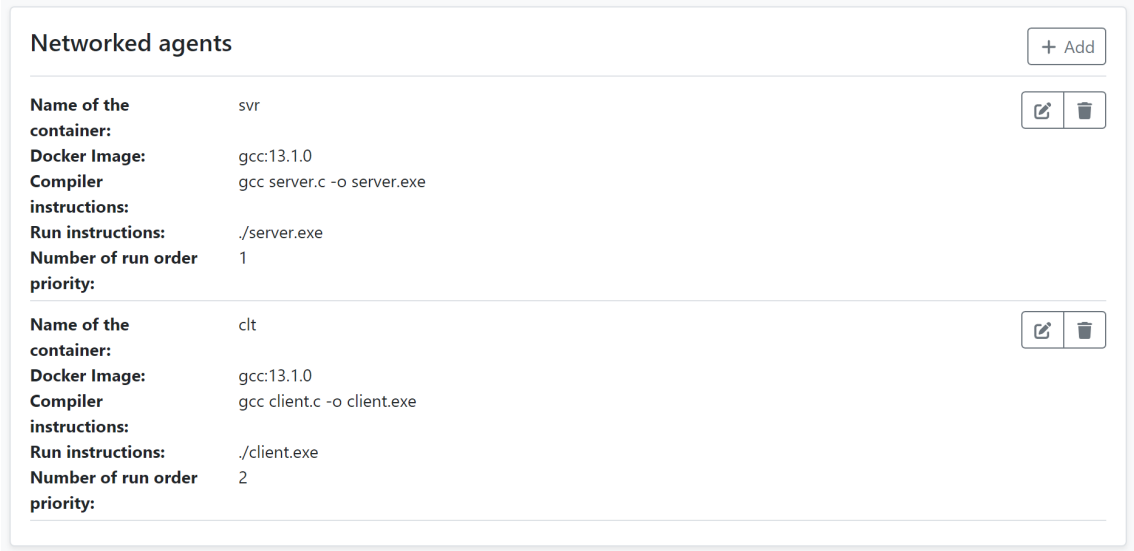

Figure 5.3: List of networked container configurations

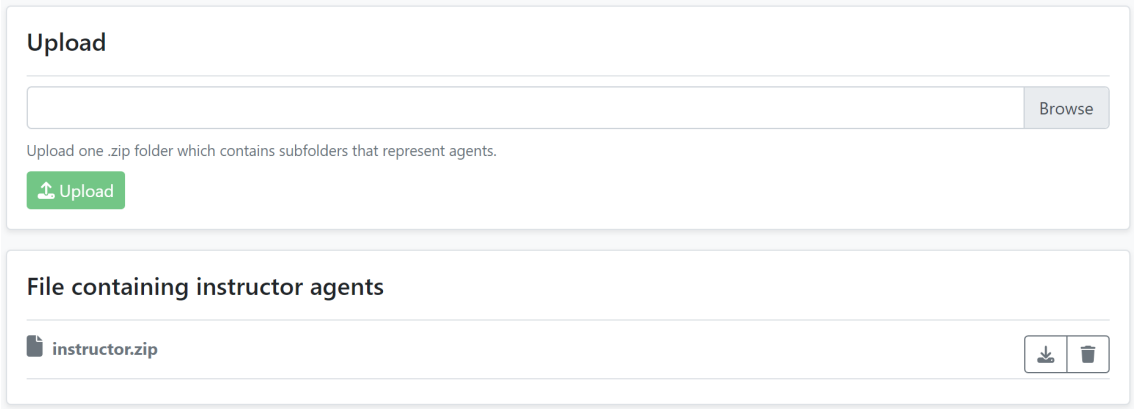

Figure 5.4: Upload field for instructor agents

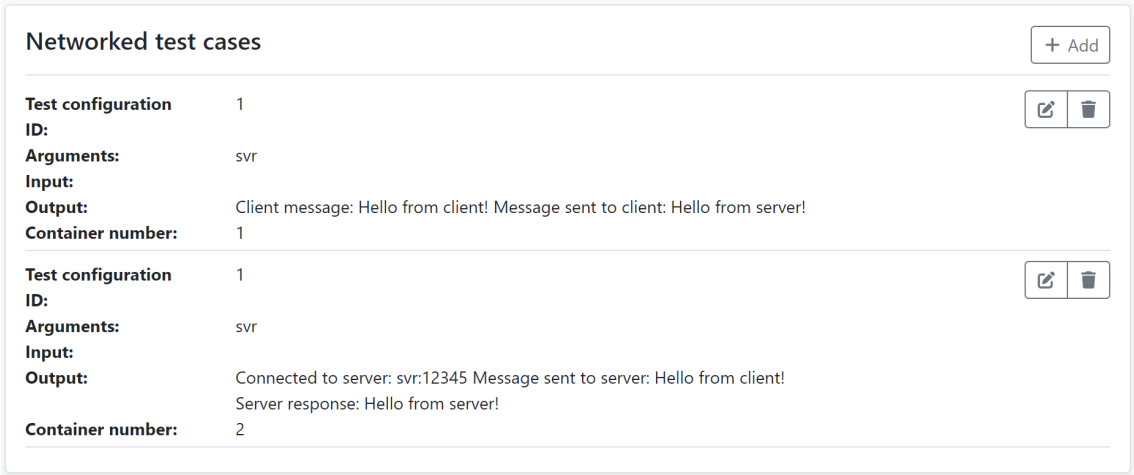

Figure 5.5: List of networked test cases

Chapter 6

Results

As for the results of the prototype implementation, I examine multiple experiments within two base scenarios. The first one is a scenario with four active agents within the network, the code for all four of them written in Python while the second scenario contains just two agents written in C. The experiments inside the scenarios aim to showcase different test results based on the submitted code and configurations. In both scenarios, the host names of the relevant containers are passed to each appropriate piece of code so that connection is able to be established.

6.1 Scenario 1 - Python, four agents

In this scenario there are four agents on the Docker network, two of them written by the instructor, two written by the student. Let's suppose that the following flow of communication needs to be implemented:

1. The first agent implements a server which awaits connection indefinitely, prints a confirmation message string after successful connection, then sends back a message to the client containing the received data from the client concatenated to a string.

2. The second agent implements a client that connects to the first agent and sends it data containing a string that it received on the standard input stream, then receives the confirmation message and prints it out.

3. The third agent is the exact same as the first agent, it just listens on a different port.

4. The fourth agent is the exact same as the second agent, except it connects to both the first and third server agents, sends them a message and prints out both confirmation messages when received.

To parse the following experiments, some information is needed. The instructor implements the code inside the first and fourth agents, while the student is responsible for the second and third ones. This means that both the instructor and the student have to implement one server and one client in this exchange. The first server will be listening on port 8082, while the second server will be listening on port 8083. Since the code for the agents is written in Python, the compilation instruction field of the networked container configuration list is left empty.

The file name of the server code for both agent one and three is server.py, for the clients it is client.py. The running instruction for the server agents contains the -u flag for the python command in order to flush the standard output immediately after printing. This is required, because for this particular implementation, the server code runs in an infinite loop and only ends when the container is destroyed. Since the containers are destroyed only after the last execution of the last container occurs, this means that for this type of implementation, output gathering can only happen with the flush option enabled.

6.1.1 Experiment 1 - Execution Failed

In this experiment, a mistake is made on purpose while configuring the networked containers. The running instructions of the third agent are mistyped from 'python server.py' to 'python servers.py'. The agents are ran, after which the results are checked and – as per expected – the tester sets the student submission to 'Execution Failed' status. The detailed error messages provides the correct information about the error.

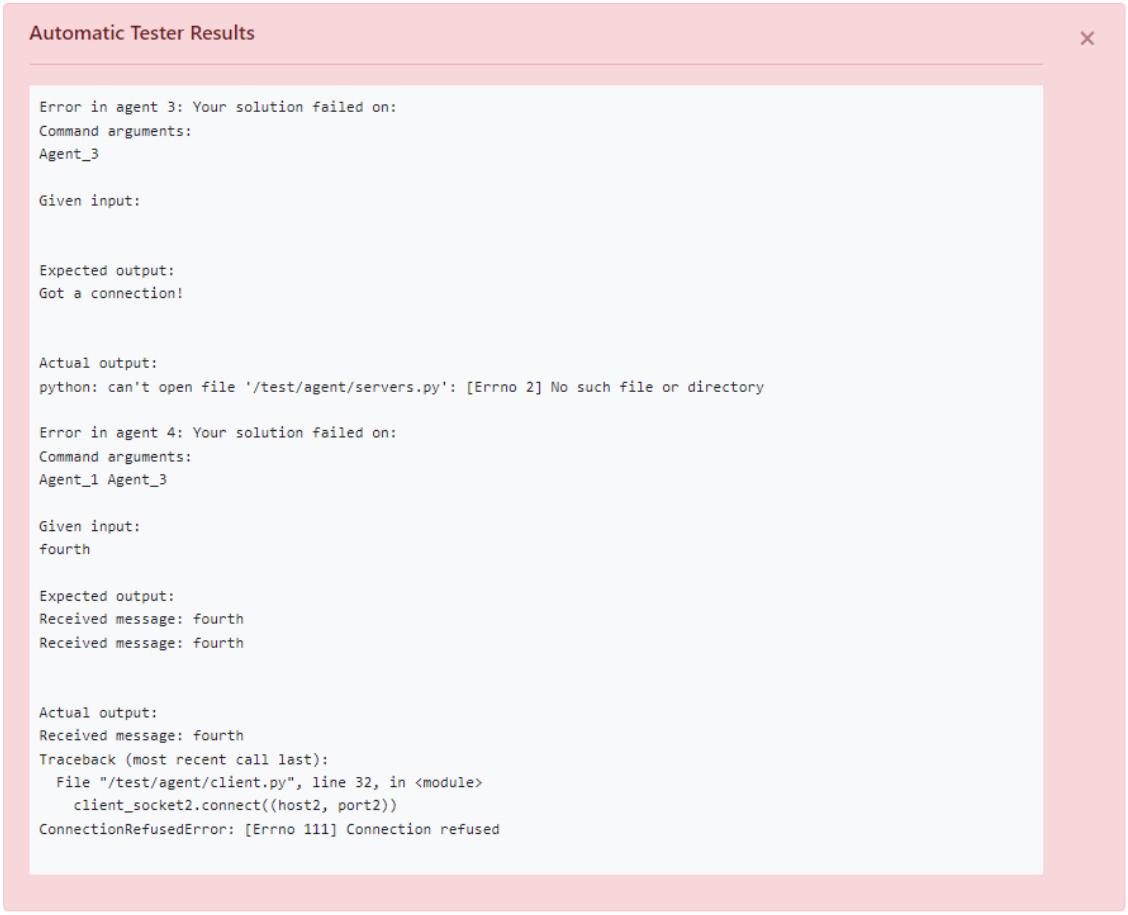

Figure 6.1: Results of experiment 1.1

6.1.2 Experiment 2 - Tests Failed

In experiment 2, the execution of the code will commence without errors however, another intentional mistake is made, this time inside of the code. A simple error is made in the client code of agent two, mistyping the 'socket.connect' method call to 'socket.connects' which does not exist. After the run process is completed a lengthy message can be seen informing the student and the instructor that the output of the container was an error message saying that the socket object has no attribute named 'connects' instead of the expected output.

```
Error in agent 2: Your solution failed on:
Command arguments:
Agent_1

Given input:
second

Expected output:
Received message: second


Actual output:
Traceback (most recent call last):
  File "/test/agent/client.py", line 16, in <module>
    client_socket.connects((host1, port1))
    ^^^^^^^^^^^^^^^^^^^^^^
AttributeError: 'socket' object has no attribute 'connects'. Did you mean: 'connect'?
```

Figure 6.2: Results of experiment 1.2

6.1.3 Experiment 3 - Tests Passed

In the last experiment for this scenario there will be no errors made in the configuration of the networked containers, or inside of the code. The run process finishes, the results are checked against the expected output and everything is found to be in order.

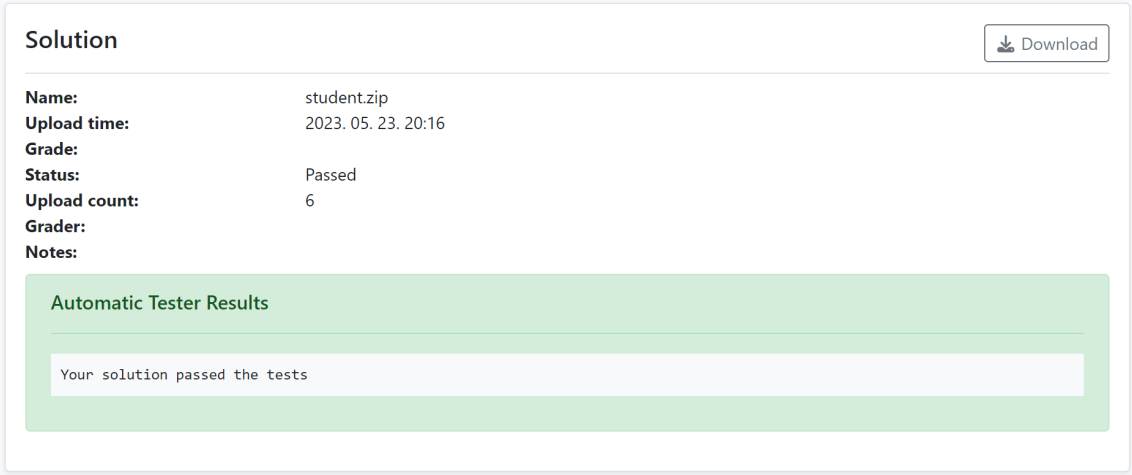

Figure 6.3: Results of experiment 1.3

6.2 Scenario 2 - C, two agents

In scenario 2, a single experiment is conducted to demonstrate a specific type of error. The flow of communication can be imagined the same as the previous scenario's first and second agents, the first being the server, the second being the client.

6.2.1 Experiment 1 - Compilation Failed

Since the code for these agents is written in C, there is a compilation step in the run process. This experiment aims to demonstrate that the compilation error is found by the tester and displayed accordingly. The error is enacted by mistyping the compilation instructions for the second agent from 'gcc client.c -o client.exe' to 'gcc clients.c -o client.exe' inside the networked container configuration list.

```
Error in agent 2: cc1: fatal error: clients.c: No such file or directory
compilation terminated.
```

Figure 6.4: Results of experiment 2.1

6.3 Performance

As for the performance of the evaluation process we can make a couple of observations. Serious performance analysis is not possible at this point, since the evaluator is not yet in production as of writing this thesis paper. For this exact reason, performance may vary as the prototype is deployed to production. The most apparent observations are the following:

- Taking by far the longest time in the entire evaluation process is the creation and destruction of containers with both operations taking approximately one second long. This means that for a four agent networked system, the run process takes approximately eight seconds long.
- The rest of the evaluation process of creating the network and runner configurations as well as the tester method calls take only a negligible amount of time, less than one second all combined.

Chapter 7

Summary and Future Work

Automated testing of networked applications is an important topic to research for developers and organisations working on such applications, as well as for instructors whose college courses deal with the topic. I have performed research with the aim of finding the most appropriate parameters to construct an automatic tester for such applications. Specifically, the information from the research needed to provide a good framework to ensure the robustness, agility and safety of my prototype implementation.

After the qualitative research, the process of of analysing the steps of running and evaluating networked applications began. The approach was to list the requirements for the tester, then assemble the system design and workflow based on the structural makeup of a networked application, making sure the requirements were always addressed.

The working prototype was then developed with the adequate functionalities. The automatic tester module of *TMS* was expanded to account for networked applications. Using the solution, instructors are able to configure agents for the networked task, add test cases, and upload their own code as well. Testing of student submissions happens after an untested solution is detected by the cron command. The information gathered during the reseach phase of the thesis was used to ensure adequate results for the evaluator.

7.0.1 Limitations and Future Work

As for limitations, there are two that are immediately apparent. The first one has to do with the human component of the testing process, as instructors will have

to configure a much larger amount of data as compared to the automatic console application tester of *TMS*. This might require a sort of introductory period, as instructors learn to use the tester well and provide feedback about its usability.

The second significant limitation is that in the current prototype implementation, there is no option to run code inside of non-Linux based containers. For most of the time, this shouldn't prove to be that much of a drawback, but it definetly needs to be noted, as there is room for improvement in this category.

Another possible point of upgrade might be the optimisation of the code, as it is currently deemed capable of improvement.

In summary, the basic objective of researching the topic of automated testing of networked applications and implementing a prototype that showcases the results of the research has been successfully reached. The implementation is robust although open to further advancement, and the technical research methodology stands on solid ground.

Acknowledgements

I would like to express special thanks to multiple individuals in this section. Most importantly, I would like to thank Máté Cserép, my supervisor for his patience and insight he provided me with despite his busy work schedule. I would additionally like to thank two other fellow students, Péter Kaszab and Marcell Hajdu who have actively been developing TMS and also helping me with miscellaneous questions.

Bibliography

[1] Marcantonio Catelani et al. "Software automated testing: A solution to maximize the test plan coverage and to increase software reliability and quality in use". In: *Computer Standards & Interfaces* 33.2 (2011), pp. 152–158. ISSN: 0920-5489. DOI: https://doi.org/10.1016/j.csi.2010.06.006.

[2] Joseph Bosas. "Automated Testing Importance and Impact". In: *2018 IEEE AUTOTESTCON*. 2018, pp. 1–4. DOI: 10.1109/AUTEST.2018.8532522.

[3] Karuturi Sneha and Gowda M Malle. "Research on software testing techniques and software automation testing tools". In: *2017 International Conference on Energy, Communication, Data Analytics and Soft Computing (ICECDS)*. 2017, pp. 77–81. DOI: 10.1109/ICECDS.2017.8389562.

[4] Chris Wilcox. "The Role of Automation in Undergraduate Computer Science Education". In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. ACM, Feb. 2015. DOI: 10.1145/2676723.2677226.

[5] Chris Wilcox. "Testing Strategies for the Automated Grading of Student Programs". In: *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. ACM, Feb. 2016. DOI: 10.1145/2839509.2844616.

[6] Christoph Torens and Lars Ebrecht. "RemoteTest: A Framework for Testing Distributed Systems". In: *2010 Fifth International Conference on Software Engineering Advances*. IEEE, Aug. 2010. DOI: 10.1109/icsea.2010.75.

[7] Evan Maicus et al. "Autograding Distributed Algorithms in Networked Containers". In: *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. ACM, Feb. 2019. DOI: 10.1145/3287324.3287505.

[8] Juan P. Sotomayor et al. "Comparison of Runtime Testing Tools for Microservices". In: *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*. IEEE, July 2019. DOI: 10.1109/compsac.2019.10232.

[9] Victor Heorhiadi et al. "Gremlin: Systematic Resilience Testing of Microservices". In: *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, June 2016. DOI: 10.1109/icdcs.2016.11.

[10] Mazedur Rahman and Jerry Gao. "A Reusable Automated Acceptance Testing Architecture for Microservices in Behavior-Driven Development". In: *2015 IEEE Symposium on Service-Oriented System Engineering*. IEEE, Mar. 2015. DOI: 10.1109/sose.2015.55.

[11] Mohammad Javad Kargar and Alireza Hanifizade. "Automation of regression test in microservice architecture". In: *2018 4th International Conference on Web Research (ICWR)*. IEEE, Apr. 2018. DOI: 10.1109/icwr.2018.8387249.

[12] Wenliang Xiong, Harpreet Bajwa, and Frank Maurer. "WIT: A Framework for In-container Testing of Web-Portal Applications". In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2005, pp. 87–97. DOI: 10.1007/11531371_14.

[13] František Špaček, Radomír Sohlich, and Tomáš Dulík. "Docker as Platform for Assignments Evaluation". In: *Procedia Engineering* 100 (2015), pp. 1665–1671. DOI: 10.1016/j.proeng.2015.01.541.

[14] Matthew Peveler, Evan Maicus, and Barbara Cutler. "Comparing Jailed Sandboxes vs Containers Within an Autograding System". In: *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. ACM, Feb. 2019.

[15] Zoltán Zele. "Automatizált beadandó tesztelő webalkalmazás fejlesztése". http://hdl.handle.net/10831/38848. BSc thesis. ELTE Eötvös Loránd University, Faculty of Informatics, 2018.

[16] Kálmán Kostenszky. "Webes programozási beadandók automatizált tesztelése és távoli végrehajtása". MSc thesis. ELTE Eötvös Loránd University, Faculty of Informatics, 2022.