Cloud-based Verification IDE

Bachelor’s Thesis

Dennis Buitendijk

September 1, 2021

Advisors: Prof. Dr. Peter Müller, Linard Arquint, João Perreira
Department of Computer Science, ETH Zürich
Abstract

Program verifiers typically depend on an SMT solver that complicates their installation. Especially for novices, the installation overhead might stop them from trying out a verifier. For expert users verifying large programs, verification can be perceived as being slow. In this thesis, a verification service running in the cloud is developed that results in zero installation overhead for novice users and can offer faster verification for expert users. Our evaluation simulating a classroom setting shows that the service can handle 34 concurrent users. In addition, we have measured slightly worse verification durations on the server with a single user and locally. These results are promising because the service can be deployed to more performant servers and the verification tools have not yet been optimized to utilize the full power of a distributed deployment.
# Contents

<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2 Background</td>
<td>3</td>
</tr>
<tr>
<td>2.1 Web-based IDE (Web-IDE)</td>
<td>3</td>
</tr>
<tr>
<td>2.2 Verification Tools</td>
<td>4</td>
</tr>
<tr>
<td>2.2.1 Viper</td>
<td>4</td>
</tr>
<tr>
<td>2.2.2 Gobra</td>
<td>4</td>
</tr>
<tr>
<td>2.3 Gobra-IDE</td>
<td>4</td>
</tr>
<tr>
<td>2.3.1 Client</td>
<td>5</td>
</tr>
<tr>
<td>2.3.2 Server</td>
<td>5</td>
</tr>
<tr>
<td>3 Architecture</td>
<td>7</td>
</tr>
<tr>
<td>3.1 Single-user vs. Multi-user IDE</td>
<td>7</td>
</tr>
<tr>
<td>3.1.1 Multi-user IDE</td>
<td>8</td>
</tr>
<tr>
<td>3.1.2 Single-user IDE</td>
<td>8</td>
</tr>
<tr>
<td>3.1.3 Chosen Architecture</td>
<td>9</td>
</tr>
<tr>
<td>3.2 Hub Architecture</td>
<td>10</td>
</tr>
<tr>
<td>4 Implementation</td>
<td>11</td>
</tr>
<tr>
<td>4.1 Web-based IDE implementation</td>
<td>11</td>
</tr>
<tr>
<td>4.1.1 Storage</td>
<td>11</td>
</tr>
<tr>
<td>4.2 Code-Server Hub</td>
<td>14</td>
</tr>
<tr>
<td>4.2.1 Hub</td>
<td>14</td>
</tr>
<tr>
<td>4.2.2 Dockerspawner</td>
<td>15</td>
</tr>
<tr>
<td>4.2.3 Idle-culler</td>
<td>15</td>
</tr>
<tr>
<td>5 Evaluation</td>
<td>17</td>
</tr>
<tr>
<td>5.1 Methodology</td>
<td>17</td>
</tr>
<tr>
<td>5.2 Single-user Performance</td>
<td>18</td>
</tr>
</tbody>
</table>
## Contents

5.2.1 Zune ................................................. 18  
5.2.2 Parallel_Search_replace .......................... 19  
5.2.3 Comparison ........................................ 19  
5.3 Resource usage ..................................... 20  
5.3.1 Zune ............................................... 20  
5.3.2 Parallel_Search_replace .......................... 20  
5.3.3 Takeaways ........................................ 22  

6 Conclusions and Future Work .......................... 25  

A Verification files ...................................... 27  
  A.1 Zune Program ....................................... 27  
  A.2 Parallel_Search_replace .......................... 28  

Bibliography .............................................. 31
Chapter 1

Introduction

We have found that students often struggle with learning formal methods. They find a lot of the concepts challenging. To help learn about verification they are often introduced to verification tools. However, these tools often have complicated installation steps. The additional effort required may lead some people to quit before they try the tools themselves. By providing a ready-to-use verification environment, people could be encouraged to try the tools.

In this thesis, we built such a service. As a first step, the service provides the ability to use Gobra, a deductive verification tool for the Go programming language, developed at the Programming Methodology group. Currently, Gobra can be installed in two ways. It is provided as a command-line interface (CLI) or as a Visual Studio Code (VS Code) extension.

We believe that such a service would be adequate for a classroom environment. Universities offering courses on formal methods could run their own instance of the service for their students. Additionally, running verification jobs on a powerful server has the added benefit of being faster.

The main focus of this project is to develop a service that provides users an integrated development environment (IDE) that is accessible through a browser. The IDE comes preconfigured with Gobra. This service provides an easy way to use Gobra in a browser. As such it overcomes the shortcomings mentioned above.

The report is structured as follows: In Chapter 2 the necessary information on used technologies is provided. The service’s architecture is presented in Chapter 3. This chapter gives a high-level overview of the service and how a user interacts with it. The implementation of this architecture is presented in Chapter 4. In Chapter 5 the performance of the service is presented. In the last chapter, the work is concluded and an outlook over future work is provided.
This chapter aims at giving an understanding of what a Web-based IDE is, what Gobra is, and how it is used in the Gobra-IDE.

2.1 Web-based IDE (Web-IDE)

Web-based IDEs consist of an IDE built on top of Web technologies. Because of this, they are often provided as cloud services and accessible via Web browsers. A wide range of Web-IDEs are available both commercially (e.g., GitHub Codespaces [4] and AWS Cloud9 [1]) and in the open-source community (e.g., JupyterLab [7] and Eclipse Theia [3]).

GitHub Codespaces offers high-performance VMs that run VS Code. A user can access the IDE either from the Desktop or through a Browser. They then have access to all features VS Code has to offer. AWS Cloud9 offers users an IDE running on AWS servers, that allows users to work together on the same code and allows code deployment directly to AWS. Eclipse Theia is an IDE platform that can be used both as a Desktop and as a cloud application.

A more specialized example of a Web-IDE is JupyterLab [7]. This is a Web-IDE created by Project Jupyter specifically for Jupyter Notebooks [7]. Jupyter Notebook is an interactive file format that includes the users’ session history, code, and the output of the code. These are often used in machine learning or data science. JupyterLab is often used together with JupyterHub [7] to provide a pre-configured workspace to members of an organization. JupyterHub uses a centralized managing system that handles the authentication of users and spawns a JupyterLab instance for each user.
2. Background

```
1 requires acc(&x.val) && acc(&y.val)
2 ensures acc(&x.val) && acc(&y.val);
3 ensures x.val == old(y.val) && y.val == old(x.val);
4 func swap1(x, y *cell) {
5 x.val, y.val = y.val, x.val;
6 }
```

Figure 2.1: Gobra swap function

2.2 Verification Tools

2.2.1 Viper

Viper [8] is an intermediate verification language with associated verification tools, developed by the Programming Methodology Grout at ETH Zürich, to prove safety and functional properties on Viper programs. Viper allows users to specify methods with pre- and postconditions, which specify valid states in which the method can be called and the expected state when the method returns, respectively. This enables Viper to modularly verify that (1) every method call is performed in a state satisfying the callee’s precondition, and (2) executing a method in any state satisfying the precondition results in a state satisfying the postcondition.

2.2.2 Gobra

Gobra [10] is a modular, deductive program verifier for Go programs, developed by the Programming Methodology Group at ETH Zürich, based on their Viper toolchain. It takes as input an annotated Go program. The annotations are specifications the program should satisfy. To verify these specifications, Gobra encodes the Go program into a Viper program and verifies it using one of Viper’s backends. If that verification succeeds, Gobra reports verification success for the Gobra program. If it fails, Gobra reports which parts of the Gobra program caused the verification error. The results of verifying a certain method are cached to speed up verification when repeatedly verifying a program that has unchanged methods. As an example of a Gobra program, we look at a program that swaps two values, depicted in Figure 2.1. The annotations define the precondition (requires acc(x.x) && acc(y.val)) and the postconditions ensuresacc(x.val) && acc(y.val) and ensuresx.val == old(y.val) && y.val == old(x.val)). These pre- and postconditions express that there has to be write access to the value field of x and y, and that after execution the values of the fields have swapped.

2.3 Gobra-IDE

Gobra-IDE [9] is a VS Code extension that integrates Gobra into the text editor. Using Gobra-IDE, developers get verification results directly in the text editor.
The implementation of Gobra-IDE follows a client/server architecture. The client corresponds to the actual VS Code plugin and updates the state of VS Code according to the verification results, whenever the server performs the Gobra verification jobs.

### 2.3.1 Client

The client is the part of the Gobra-IDE extension that runs directly in VS Code. It allows the users to implicitly and explicitly interact with Gobra. The explicit interactions include commands to verify the currently opened file, convert it into a Gobra program, or convert it into a Go program. The implicit actions get triggered by opening a file, making changes to a file, and saving a file. These actions trigger events that control the server part.

When verification completes, the client displays the results directly in VS Code. If the verification fails with an error, the piece of code that caused it is underlined by a squiggly line as seen in Figure 2.2. Additionally, in the toolbar of VS Code a message containing the result, the verification time, and what error occurred is displayed. If the verification succeeds, the message displayed in the toolbar will display a success message as seen in Figure 2.3.

### 2.3.2 Server

The server gets started by the client on activation of the extension. It consists of three main parts: a communication module, Gobra, and the Viper backend. The communication module handles all communication between the server and the client. This includes sending the verification request to the server and receiving the result from it.
Figure 2.3: Gobra-IDE on verification success
Chapter 3

Architecture

This chapter gives a high-level overview of the service’s architecture. The service can be split into two parts: The Hub and the IDE servers. Each is implemented as Docker images for ease of deployment. The IDE servers are responsible for executing the IDE. This includes a web interface for VS Code, handles the integration of verification tools, and manages the file storage. The Hub authenticates users, assigns the users to an IDE, and cleans up after the user disconnected. It also controls the IDE containers.

3.1 Single-user vs. Multi-user IDE

Regarding the organization of the IDE environment, we considered two architectures. Firstly, Figure 3.1 depicts an architecture in which multiple users connect to the same IDE container. This means that we have a fixed number of containers running on the Host server, and when a new user connects they get assigned to the IDE container with the least load at the time of connecting. This results in what we will call Multi-user IDE containers as multiple users are active in the same container. Secondly, Figure 3.2 shows an architecture with a dynamic amount of containers. When a new user connects, a new container is created and assigned to the user. Next, the advantages and disadvantages of each architecture are discussed.

![Figure 3.1: Multi-user IDE](image1)

![Figure 3.2: Single-user IDE](image2)
3. Architecture

3.1.1 Multi-user IDE

Advantages. As mentioned previously, multiple users share the same instance of the IDE container in this architecture. As the resources used by the IDE can be reused for multiple users, we estimate that the overhead of the service is lower. Another advantage is that we can start the System with a set amount of containers, depending on the number of simultaneous users that should be supported. Afterwards, containers only need to be restarted in the event of a crash. For the rest of the time, we would not have to make any changes to the IDE containers apart from adding or removing users. As a result of always having a set number of IDE containers, we can easily assign users to their IDE container with a simple load balancer, which makes sure that all requests of a user are routed to the same containers.

Disadvantages. The complexity of each IDE container is much higher. As multiple users connect to the same container, they share the same filesystem. Thus, it becomes much more difficult to isolate users from each other. To prevent users from accessing another user's files, a permission-based system has to be in place. In addition, the permissions have to support dynamic addition and removal of users. With several users connected to the same container, it also gets harder to assess which user is still actively using the IDE. Therefore, it is difficult to decide when to disconnect a user to free up resources for other users. Having a set amount of containers running at all times also has disadvantages. Most notably, it is harder to react to changes in the workload. For example, if we have two containers running with 10 users each. If then all ten users from one container disconnect, we have one container that is idle and one has a high workload. To balance the two containers, five users would need to be disconnected from one container and reconnected to the idle container. For this to work, all verifications from those users would have to be stopped, their files copied over, and their verifications restarted in their new container.

3.1.2 Single-user IDE

Advantages. As mentioned in section 3.1, in this architecture each user connects to their own IDE container. These containers do not have access to any files on the host system and as such users can not access files of another user. By not needing to accommodate multiple users in a single container, these can be kept very simple. They only need to run a single instance of the IDE and expose it through a web interface. They do not need to check for multiple users if they are allowed to have access to the container, they only check if the user attempting to connect is the user assigned to this container. In addition, the filesystem does not need to be partitioned into a section per user. Additionally, choosing where to assign a user is as simple as it gets: a
3.1. Single-user vs. Multi-user IDE

Figure 3.3: The proposed architecture

new user connecting to the service simply results in starting a new container. In particular, no decision based on the current load of each existing container has to be made.

Disadvantages. The disadvantages of this architecture are all related to the overhead. As every user gets their own IDE container, more containers will be simultaneously running. Each container adds overhead as they are lightweight operating systems. By creating a new container for each user and shutting it down when they disconnect adds further performance penalties. In particular, there is a small delay between accessing the service and the IDE container being ready. Furthermore, resource usage tends to be high while starting a container. However, these overheads are very small in comparison with the resource usage during verifications.

3.1.3 Chosen Architecture

As the benefits of having a simpler IDE container with guaranteed isolation between users outweigh the relatively small overhead of the Single-user containers, we have chosen the single-user IDE architecture. This results in an architecture as depicted in Figure 3.3. The Reverse-Proxy handles the routing of users to the correct location. If they already have their IDE container, they get routed to that. If they are newly connecting, they get routed to the Hub to log in.
3. Architecture

Figure 3.4: An overview of the Hub

3.2 Hub Architecture

The Hub is composed of six parts, shown in Figure 3.4. The core part of the Hub is the Hub itself. It handles the interactions between all the other parts. It provides an interface for each of the other parts so that these can be changed based on how the service should work, as shown in Figure 3.4: (1) On startup of the Config tells the Hub which parts are used and in the case of the Proxy and the Database, where they are located. (2) When a new user connects, the Proxy routes them to the Hub. (3) The Hub then uses an Authenticator to let the user log in. There is an Authenticator for every supported authentication method. The Hub then gets either an error to display to the user, or it gets a success message along with the username. (4) When the user is logged in, the Hub tells the Spawner to create a new IDE container for the new user. The spawner starts the container and configures it for the user, makes sure it is running, and then returns the IP Address of the container to the Hub. (5) The Hub then stores all necessary information in the Database. This includes the name of the user, the IP Address, and the Spawner used to start the container. (6) The Hub then configures the Proxy to route the user to their own IDE container.
Chapter 4

Implementation

As mentioned in Section 3.1.3, we use IDE containers and a Hub to control the IDE containers. In this chapter we will discuss the Implementation of both the IDE containers, and the Hub, which we will call Code-Server Hub.

4.1 Web-based IDE implementation

Part of the architecture is a docker image that creates the docker containers that run the IDE. We call these containers IDE containers. The IDE containers are based on Code-Server [2]. Code-Server, is an open-source project based on VS Code, that allows a user to access a VS Code instance through a browser. This allows a company to set up a private server, to which their employees can connect from anywhere, without the need to set up all tools needed for development. Since Code-Server allows unrestricted access to the filesystem, this can only be used for trusted users. The UI of Code-Server (Figure 4.1) is very similar to the default VS Code UI. One difference is that the menu bar is not located at the top, but it is toggled by a button (1). As a second change, we removed the terminal to prevent the execution of arbitrary commands by malicious users. Currently, Code-Server is distributed as a Binary or as a Docker image. In the service, we use a Docker image based on the Code-Server image that includes the Gobra-IDE and the Gobra tools.

4.1.1 Storage

One problem with deploying an IDE service on a server is storage. It needs to be consistent across IDE container restarts and has to be scalable. As storage space on servers is often limited or expensive, we wanted to limit the consumed storage to the minimum required to run the service. Thus, files created by users should not be stored on the server but client-side whenever possible. Therefore, we needed a way to send the files from the server to the user’s browser, where it is stored, and back.
Browser Storage. There are five ways to store data in a browser: Cache, Cookies, SessionStorage, LocalStorage, and IndexedDB. The Cache is designed to store Request/Response object pairs. Cookies are of limited size in the range of a few kB and are restricted in quantity of cookies per domain, which varies between browsers. SessionStorage is a key/value storage that has a limit of a few MB of data and gets cleared each time the browser is closed. LocalStorage is similar to SessionStorage but has a higher storage limit. Although it is limited to a few MB as well, browsers persist the data across browser sessions. The IndexedDB is a transactional database system, which was designed to store significant amounts of structured data, including audio or video files.

For our application, only LocalStorage and IndexedDB offer the necessary persistency. However, as the LocalStorage can only store a few MB of data, we might not be able to store all files for all users with it. Therefore, we decided to use the IndexedDB. The files get stored in the IndexedDB in a JSON format containing the file path and the file content.

Interacting with the IndexedDB requires client-side JavaScript. However, filesystem access is only possible in the IDE container. Therefore, we split the code into two parts as shown in Figure 4.2; A VS Code extension to read and write files on the server, and a small javascript program that interacts with the IndexedDB and runs on the client’s side in the browser.
4.1. Web-based IDE implementation

**Communication.** To communicate between the VS Code extension and the client-side script, we use Socket.io [6]. Socket.io is a library that can be used for event-based communication between the browser and the server. It consists of a Node.js server and a JavaScript client library for the browser. The client library can either be downloaded from the server, or from a Socket.io content distribution network (CDN). Internally, it uses the WebSocket API. In case the WebSocket connection cannot be established, it falls back to HTTP long-polling.

**Server.** The VS Code extension starts the Socket.io server as soon as the extension is activated. When the client connects, the extension requests the files stored in the client, writes these into the server’s filesystem, and starts monitoring the workspace for file changes. When a file is saved or is deleted, the extension updates the database on the client-side. Furthermore, the extension provides two commands to allow users to explicitly store or load files.

**Client.** The client-side script opens the IndexedDB when the page loads. On the first load, the Database (DB) is initialized. It then attempts to connect to the server. When the connection has been established, it reacts to messages
sent from the server signalling changes to the filesystem. This script has to be included in the HTML document through which the user interacts with VS Code. For this reason, the main HTML file provided with Code-Server is overwritten by a version that includes the client-side script along with two other scripts. The first one is the Socket.io client library that will be downloaded from their CDN. The second one is a simple script declaring a constant that specifies the server’s URL to which the client should connect.

**Message.** The server sends three different types of messages. The first one is the `getFiles` message. It has no arguments and is sent to the client to retrieve all files from the DB. The second one is the `setFile` message. It sends a file object as an argument, which the client then stores in the DB. When the file already exists, the contents of the File is updated. The third message is `deleteFile`. This is sent with the File path as an argument to tell the client to remove a particular file from the DB.

The only message the Client sends is used to send all Files in the DB to the server. This message is the `returnFiles` message, which sends an Array of File objects.

### 4.2 Code-Server Hub

As mentioned in Section 3, the Code-Server Hub manages the IDE containers and the authentication of users. The Code-Server Hub is based on Jupyter-Hub [7]. As mentioned in Section 2.1, JupyterHub is used to provide Jupyter Notebook instances to the members of an organization. By default, Jupyter-Hub spawns a container running Jupyter Notebook per connected user. In our use-case, a container per user should be spawned as well but instead of a Jupyter Notebook a Code-Server container, as described in Section 4.1, should be spawned. Thus, we adapted JupyterHub. Three parts of the JupyterHub implementation had to be adapted: The Hub, a Spawner implementation (Dockerspawner) and an addition to shut down idle servers (Idle-culler).

#### 4.2.1 Hub

Adapting JupyterHub to Code-Server required changing the Proxy, responsible for routing users to the correct location, and the authentication mechanism. For the Proxy, additional routes have to be configured per user. The first route is used for the communication between the server and client of the storage extension in Code-Server. The second route handles logout requests such that a user is correctly logged out in Code-Server and the Hub. Additionally, the Hub sets the login cookie used by Code-Server to restrict the use of a Code-Server instance to the user it belongs to.
4.2. Code-Server Hub

To support the use-case of novice users trying out a verifier, having an account should not be a prerequisite. Nevertheless, bots have to be excluded to avoid denial of service attacks. Therefore, we added a guest authentication mechanic to the base `Authenticator` class. This allows us to change the `Authenticator` without needing to reconfigure the guest login. When a guest logs in, they will have to first solve a Captcha, after which they can then use the service like a regular user.

4.2.2 Dockerspawner

The Dockerspawner launches new containers running Code-Server whenever users connect. As the Code-Server container requires a different configuration than the Jupyter Notebooks, the way the container is started needed to be changed. To start a Code-Server container for a specific user, we need to configure several things in the container. To do this, we write configuration files in the container as opposed to using command line arguments as with Jupyter Notebooks.

The first one is to configure Code-Server to use a password to authenticate the connecting user. This password is generated by the Hub and the user, that is allowed to use the container, gets a cookie set so that they are automatically authenticated in the container. This is done with a config file that is overwritten during spawning with the correct password.

The second configuration is related to the storage. The client-side script needs to know where to connect to the server. This is done by writing the address of the server into the HTML specified in 4.1.1. Only after the value is in the IDE container, the spawning of the server is completed.

4.2.3 Idle-culler

When users disconnect or log off, the associated containers should be terminated to minimize resource consumption. JupyterHub offers a service infrastructure that can be configured to be executed along with the Hub. The Idle-culler service is used to shut down idle Jupyter Notebooks. It has been adapted to use the Healthz endpoint provided by Code-Server containers to query the last timestamp when a user interaction took place. The Idle-culler queries this timestamp and if it is older than a specified time, it shuts down the container. This is done to reduce the load on the server.
Chapter 5

Evaluation

To evaluate the system, we look at the resources required to run it per connected user. We measure the resource usage per container during each verification run. We also compare the performance of the service with a single user against running it locally. To do this comparison, we compare the time it takes to verify a certain file locally and on the server.

5.1 Methodology

To simulate real-world usage, we ran a verification on a Gobra program in several stages of completeness. This simulates a user working on a file, gradually adding or completing functions. A user would first verify an initial version of the file, fix possible problems detected by the verification or extend the program, then verify the next version until the complete program was verified. Afterwards, the cache is cleared to prepare for another iteration of the verification run. These user interactions are replayed several times, each separated by flushing the cache to achieve comparable results.

To simulate the user running the verification, we use Selenium [5] to interact with the service. It first logs in, then creates the necessary files on the server, and then writes the files. Afterwards, it starts the verification and reads the result of the verification.

The files used for the evaluation are based on two Gobra files as seen in Appendix A: Zune.gobra and Parallel_Search_Replace.gobra. Zune.gobra is an implementation of the Zune bug, where not considering a leap year temporarily disabled millions of devices. The file includes three functions, two of which stay unchanged across all three files. In the least complete version, the third function has an empty body. In the second file, the function consists of the code that lead to the bug. In the complete version, the function has been fixed to not contain the bug anymore. The complexity
5. Evaluation

and verification duration of this file is rather low and is considered to be representative of verifications that are happening in an interactive classroom setting with a class size of 20 to 30. This would typically happen when introducing new concepts in the verification tool. The lecturer would show how to implement the feature, and the students can replicate it for themselves. The Parallel/Search/Replace file searches an array in parallel for a value to replace it if it is found. It consists of three functions and two predicates. In the least complete version, only the first function is implemented, and the rest consist of empty bodies. The second file also has the two predicates completed. These files will produce an error as the conditions rely on the not implemented functions. The complete version includes all function bodies and as such succeeds verification. This file takes approximately 60 seconds to verify on a modern laptop and can be used to simulate the workload of students completing a project.

The next sections provide the experimental results and discuss them. The verification time is depicted with box plots, while resource usage. File-1 denotes the least complete file that was verified, and File-3 is the complete file. For each file, the box plot shows the median verification time in green, with the Box containing the data points between the upper and lower quartiles. The lines at the top and bottom (Whiskers) cover the data points up to 1.5 times the distance covered by the box. Any data points outside that range are depicted with a circle. The tests were executed with the service running on a Server with 32 Intel Xeon CPUs running at 3.3 GHz with 8 Cores each and 241 GB of RAM.

5.2 Single-user Performance

To measure the server’s maximum performance with the current architecture, we used a single client to perform verifications. As such, we can measure if there is an increase or decrease in performance when running the verification on a server with significantly more RAM and CPU cores in comparison to running on a user’s laptop. The script ran 12 iterations with each version of the program. From the 12 iterations, the data from the quickest and slowest iteration was disregarded. To get the results locally, each file was executed manually, and the verification time was recorded manually. After each iteration of the three files, the cache was cleared. These measurements were taken on a Lenovo X1 Yoga with a 1.9 GHz 4-Core Intel Core i7 CPU and 16 GB of RAM, running Fedora 34.

5.2.1 Zune

Figure 5.1 depicts the verification performance when verifying the Zune files on the Server. It shows that each file took between 2.6 and 3.8 seconds to
verify. Additionally, there is a slight decrease in verification time for the complete file due to much of the program already located in the cache.

The results of verifying the Zune files locally are depicted in Figure 5.2. Verifying the files locally took between 1.1 and 1.9 seconds. This means that the verification is 1.5 to 1.9 seconds slower when executed on the server.

5.2.2 Parallel_Search_Replace

Figure 5.3 shows the verification times when verifying the Parallel Search Replace files on the Server. The first file takes between 9.8 and 19 seconds. The second file takes between 10 and 16 seconds. The complete file takes between 78 and 110 seconds. On the other hand, running the verification locally results in the first file verifying in 4.9 to 6 seconds as depicted in Figure 5.4. The second file took slightly longer at 5 to 6.5 seconds and the complete program was verified in 60 to 110 seconds.

5.2.3 Comparison

This test shows that verifying files on the server took at least a few seconds, and up to 13 seconds longer for each file. These results were however achieved with relatively old server hardware. The hardware used to verify
the files locally was significantly newer. How much influence this had can not be determined with this test alone. An interesting experiment would be to repeat the test with the service deployed on more modern hardware, such as with a cloud service provider.

5.3 Resource usage

In this part of the evaluation, the script was run with a total of 24 users. The users run their verifications in parallel. To be able to compare each iteration, all users wait for each other after verifying the last file. The focus of this experiment was to analyse the resource usage on the server. To analyse that, we sample the resources used by each docker container each second.

5.3.1 Zune

The goal of the Zune files was to simulate a classroom setting. In this setting, all students would verify simple files roughly at the same time. Figures 5.5 and 5.6 show the resources used by the Hub. At its peak, it uses a full CPU Core and about 100 MB of RAM while supporting 24 users. This is an increase of about 10 MB from the idle state as seen at the very start of the plot. Figures 5.7 and 5.8 show the resources used by the Proxy. It uses a maximum of half a Core and 50 MB of RAM, This is an increase of about 35 MB from idle. Figures 5.9 and 5.10 show the resources used by the DB. It uses a maximum of 15% of a Core and a maximum of 50 MB RAM. This is an increase of about 20 MB from idle. Figures 5.11 and 5.12 show the resources used by the Code-Server containers. These are the Components that need the most Resources. At the peak of load, a single container can use up to 20 Cores. During the execution of an iteration, memory usage per container constantly increased. The maximum amount an individual container used was 6.7 GB, although others achieved the same verification outcomes with as little as 4 GB.

5.3.2 Parallel_Search_Replace

Due to the script taking incredibly long to run to completion with these files, and became unreliable with the desired amount of simulated users, we decided to split the load on multiple machines and only ran for eight iterations. As such, the script to run the verifications was run on 3 machines at the same time, each simulating 6 users, resulting in increased parallelism of execution. As a result of running on separate machines, synchronization between the users only occurred with the users controlled from the same machine. Due to the increased load on the server, the Selenium script had problems working reliably. As it simulates button presses and directly interacts with the webpage’s HTML source code, it is susceptible to timing
5.3. Resource usage

**Figure 5.5:** Hub CPU usage for Zune with 24 users connected

**Figure 5.6:** Hub RAM usage for Zune with 24 users connected

**Figure 5.7:** Proxy CPU usage for Zune with 24 users connected

**Figure 5.8:** Proxy RAM usage for Zune with 24 users connected

**Figure 5.9:** DB CPU usage for Zune with 24 users connected

**Figure 5.10:** DB RAM usage for Zune with 24 users connected
issues resulting in the script not making progress anymore. Due to this, the runs had to be restarted after some iterations. As we restarted directly after shutting the script down, the Code-Server containers were still running, resulting in a negligible overhead. The completed iterations of the individual runs were combined so that each simulated user had eight complete iterations. Therefore, the runs did not result in the exact schedule we wanted. Nevertheless, the load on the server should only marginally be affected by the increased number of logins due to restarts of the script.

Figures 5.13 and 5.14 depict the resources used by the Hub during the execution. The Hub utilized a maximum of a quarter Core and 125 MB of RAM. Figures 5.15 and 5.16 depict the resources used by the Proxy during the execution. The Hub utilized a maximum of a third Core and 50 MB of RAM. Figures 5.17 and 5.18 depict the resources used by the DB during the execution. The Hub utilized a maximum of 2.5% of a Core and 75 MB of RAM. Figures 5.19 and 5.20 depict the resources used by the Hub during the execution. The Hub utilized a maximum of a quarter Core and 125 MB of RAM. Once again, these containers produce the real load. During this execution, the overall load on the CPU was higher than in the Zune test, but the peak use by one container was only 12 Cores. On the other hand, the RAM usage was significantly higher. All containers used more between 10 to 15 GB. One container even used more than 50 GB (probably due to a bug) temporarily.

5.3.3 Takeaways

This test leads to the conclusion, that for classroom usage, up to 7 GB of RAM usage per user is to be expected. As such, with the current implementation, we can support approximately 34 users in a classroom setting. However, if we expect the users to verify more complex files, we would need up to 15 GB
5.3. Resource usage

Figure 5.13: Hub CPU usage for Parallel Search Replace with 18 users connected

Figure 5.14: Hub RAM usage for Parallel Search Replace with 18 users connected

Figure 5.15: Proxy CPU usage for Parallel Search Replace with 18 users connected

Figure 5.16: Proxy RAM usage for Parallel Search Replace with 18 users connected

Figure 5.17: DB CPU usage for Parallel Search Replace with 18 users connected

Figure 5.18: DB RAM usage for Parallel Search Replace with 18 users connected
5. Evaluation

Figure 5.19: Code-Server CPU usage for Parallel Search Replace for all 18 users connected. Each colour represents one Code-Server container.

Figure 5.20: Code-Server RAM usage for Parallel Search Replace for all 18 users connected. Each colour represents one Code-Server container.

of RAM per user. The current implementation could therefore handle 16 users working on complex files simultaneously.
Chapter 6

Conclusions and Future Work

The main focus of this project was to develop a cloud-based IDE for verification purposes. The barrier of entry for new users is lowered by the possibility to run Gobra and verify Go programs directly in the browser.

Despite running on server hardware, the current service does not provide lower verification durations even in the single-user scenario. There are multiple possible reasons that could be addressed in future work. Even in the single-user scenario, there is a certain overhead caused by the architecture and by running Gobra in a docker container instead of natively. Furthermore, the server is already dated and does not represent state-of-the-art server hardware. When deploying to a cloud provider, further parallelization could be achieved by separating the IDE containers and the verification backends as depicted in Figure 6.1. Additionally, the SMT solver, which is typically the bottleneck of a verifier, could be separated and run on a different machine. To make this possible, the architecture would need to be adapted to work with Docker swarm, as it currently needs all containers to run on the same Docker host.
6. Conclusions and Future Work

Figure 6.1: Architecture with separated verification backends
A.1 Zune Program

```go
// Any copyright is dedicated to the Public Domain.
// http://creativecommons.org/publicdomain/zero/1.0/
package zune

//
func isLeapYear(year int) bool {
    return year % 4 == 0 && (year % 100 != 0 || year % 400 == 0)
}

const originYear = 1980

func convertDaysFixed(totalDays int) (days, year int) {
    days = totalDays
    year = originYear
    for (isLeapYear(year) && 366 < days) || (!isLeapYear(year) && 365 < days) {
        ghost variant := days
        assert variant >= 0
        if isLeapYear(year) {
            days -= 366
        } else {
            days -= 365
        }
        year += 1
        assert days < variant
    }
    return
}

func convertDaysFixedWithSomeInvariants(totalDays int) {
    (days, year int) {
        days = totalDays
        year = originYear
        day+ (year - originYear) * 365 <= totalDays
        totalDays <= days + (year - originYear) * 366
        days <= 366
    }
}
```

Appendix A

Verification files
A. Verification files

```plaintext
46  invariant  days + (year - originYear) * 365 <= totalDays
47  invariant totalDays <= days + (year - originYear) * 366
48  for (isLeapYear(year) && 366 < days) ||
49    (!isLeapYear(year) && 365 < days) {
50      ghost variant := days
51      assert variant >= 0
52
53      if isLeapYear(year) {
54        days -= 366
55      } else {
56        days -= 365
57      }
58      year += 1
59      assert days < variant
60    }
61    ret
```

A.2 Parallel_Search_Replace

```plaintext
1  // Any copyright is dedicated to the Public Domain.
2  // http://creativecommons.org/publicdomain/zero/1.0/
3  package pkg
4
5  import "sync"
6
7  // Proof Utility: dump a slice into a sequence
8  ghost
9  requires forall j int :: 0 <= j && j < len(s) == > acc(&s[j],_)
10  ensures len(res) == len(s)
11  ensures forall j int :: {s[j]} {res[j]} 0 <= j && j < len(s) ==>
12    s[j] == res[j]
13  pure func toSeq (ghost s [] int ) ( res seq [ int ]){
14    return ( len(s) == 0 ? seq [ int ]{} :
15      toSeq(s[: len (s) -1]) ++ seq [ int ]{s[len (s) - 1]})
16  }
17
18  pred replacedPerm (ghost s0 seq [int], ghost s [] int , ghost x, y int ) {
19    len(s0) == len(s) &&
20      (forall i int :: 0 <= i && i < len(s) == > acc(&s[i])) &&
21      forall i int :: {s[i]} 0 <= i && i < len(s) == >
22        s[i] == (s0[i] == x ? y : s0[i])
23  }
24
25  pred messagePerm (ghost wg *sync.WaitGroup, s [] int , ghost x, y int ) {
26    (forall i int :: 0 <= i && i < len(s) == > acc(&s[i])) &&
27      wg.UnitDebt(replacedPerm!<toSeq(s),s,x,y!>)
28  }
29
30  requires acc(c.RecvChannel(),_)
31  requires c.RecvGivenPerm() == PredTrue!<!>;
32  requires c.RecvGotPerm() == messagePerm!<wg...x,y!>;
33  func worker (c <- chan [] int, wg *sync.WaitGroup, x, y int ) {
34    fold acc(PredTrue!<!>()(),2/1);
35    invariant PredTrue!<!>() && acc(c.RecvChannel(),_)
36    invariant c.RecvGivenPerm() == PredTrue!<!>;
37    invariant c.RecvGotPerm() == messagePerm!<wg...x,y!>;
38    invariant ok == > messagePerm!<wg...x,y!>(s)
39    for s, ok := <- c; ok; s, ok := <-c {
40      unfold messagePerm!<wg...x,y!>(s)
41    }
42  }
43  ghost s0 := toSeq(s)
44  invariant 0 <= i && i <= len(s)
45  invariant forall j int :: 0 <= j && j < len(s) == > acc(&s[j])
46  invariant forall j int :: {s[j]} 0 <= j && j < len(s) ==>
47    s[j] == (s0[j] == x && j < i ? y : s0[j])
```

for i := 0; i != len(s); i++ {
    if(s[i] == x) { s[i] = y }
}

func SearchReplace(s []int, x, y int) {
    fold replacedPerm!<s0, s, x, y!>()
    // Gobra can detect automatically that this call is ghost.
    // In these cases, the 'ghost' annotation is not necessary.
    wg.PayDebt(replacedPerm!<s0, s, x, y!>)
    wg.Done()
    fold PredTrue<!>()
}

requires forall i int :: 0 <= i && i < len(s) == > acc(&s[i])
ensures forall i int :: 0 <= i && i < len(s) == > acc(&s[i])
ensures forall i int :: {s[i]} 0 <= i && i < len(s) == >
s[i] == (old(s[i]) == x ? y : old(s[i]))

for offset := 0; offset != len(s); {
    nextOffset := offset + workRange;
    if(nextOffset > len(s)) { nextOffset = len(s) }
    for i := 0; i != workers; i++ {
        go worker(c, &wg, x, y)
    }
    wg.Join()
    for i := 0; i != len(s); i++ {
        s[i] = s0[i] 
    }
    return
}

workers := 8
workRange := 1000
assert workers > 0
assert workRange > 0
ghost s0 := toSeq(s)
c := make(chan []int, 4)
var wg sync.WaitGroup
ghost pr := messagePerm!<&wg, _, x, y!>

c.Init(pr, PredTrue<!>())
wg.Init()
ghost seqs := seq[seq[int]]{}
ghost pseqs := seq[pred()]{}
invariant acc(c.SendChannel()) && c.SendGivenPerm == pr
forall i int :: {seqs[i]} 0 <= i && i < len(seqs) == >
seqs[i] == s0[i]
invariant len(seqs) == (len(seqs) - 1) * workRange + len(seqs[len(seqs)] - 1)
invariant forall i int :: {seqs[i]} 0 <= i && i < len(seqs) == >
seqs[i] == s0[i] + workRange
invariant len(pseqs) == len(seqs)
invariant forall i int :: {pseqs[i]} 0 <= i && i < len(pseqs) == >
pseqs[i] == replacedPerm!<seqs[i], s[i + workRange], len(seqs), x, y!>
invariant fold replacedPerm!<seqs[i], s[i] + workRange + len(seqs), x, y!>
A. Verification files

```
119       wg.GenerateTokenAndDebt(wpr)
120       fold wg.TokenById(wpr, len(pseqs))
121       seqs = seqs ++ seq[seq[int]]{ s1 }
122       pseqs = pseqs ++ seq[pred()] ( wpr }
123       fold messagePerm!<&wg,...,x,y!>(section)
124       c <= section
125       offset = nextOffset
126   }
127       wg.SetWaitMode(1/2,1/2)
128       wg.Wait(1/2, pseqs)
129   ghost {  
130       invariant 0 <= i && i <= len(seqs)
131       invariant forall j int :: i <= j && j < len(seqs) ==>  
132       sync.InjEval(pseqs[j],j),j)
133       invariant forall j int :: 0 <= j &&  
134       j < (i == len(seqs) ? len(s) : i * workRange) ==>  
135       acc(&[j]) && s[j] == (s0[j] == x ? y : s0[j])
136       for i := 0; i != len(pseqs); i++ {  
137       unfold sync.InjEval(pseqs[i],i)
138       low := i * workRange
139       up := low + len(seqs[i])
140       s1 := s[low:up]
141       unfold replacedPerm!<seqs[i],s1,x,y!>()
142       assert forall j int :: &s[j] low <= j && j < up ==>  
143       &s[j] == &s1[j-low]
144   }
145   }
146  ```
Bibliography


Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor’s thesis, Master’s thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Cloud-based Verification IDE

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):

Buiendijk

First name(s):

Dennis

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the ‘Citation etiquette’ information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date

Zürich, 09.09.2021

Signature(s)

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.