NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

WEB-BASED CNSS

SIMPLE NETWORK SIMULATOR IMPLEMENTED IN THE BROWSER

JOÃO PEDRO MATEUS VALE

BSc in Computer Science

NOVA University Lisbon
09, 2022

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

WEB-BASED CNSS

SIMPLE NETWORK SIMULATOR IMPLEMENTED IN THE BROWSER

JOÃO PEDRO MATEUS VALE

BSc in Computer Science

Adviser: Sérgio Marco Duarte
Assistant Professor, NOVA University Lisbon

NOVA University Lisbon
09, 2022

Web-Based CNSS

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v6.7.4) [1].

Abstract

In this thesis, our objective is to convert and develop CNSS, a simulator created by Professor José Legatheaux Martins, into a web application. Usually, these network simulators are implemented in a desktop application since some of them are complex and need some computational power. For this reason, the existence of a simple network simulator implemented in a web application is almost null. The existence of this network web simulator will be highly useful in a teaching and learning context since professors can quickly prepare some sketches and demonstrate them, but students can also use and try the simulator in the same class without the need for any previous installation or setup.

In short, to implement this idea, since CNSS is implemented in Java, we need to choose a tool that allows us to convert Java bytecode to Javascript so it can run entirely on a browser without the need for an external server. Our main idea is to implement a simple but robust notebook interface where anyone can change the CNSS source code and try it, since it is the best development environment for an integrated solution. Also, to upgrade CNSS even more, we are going to enhance the textual output of the simulations by using graphs or statistics, but we are also going to develop a collaborative environment where every user can share and see real-time modifications made by a professor or colleague.

Keywords: CNSS, Simple Web Network Simulator, Simulation

Resumo

Nesta dissertação de mestrado, o nosso objetivo é converter e desenvolver o CNSS, um simulador criado pelo professor José Legatheaux Martins, em uma aplicação web. Normalmente, estes simuladores de rede são implementados em uma aplicação desktop, visto que a maior parte são complexos e necessitam de algum poder computacional. Por esta razão, a existência de um simulador implementado em uma aplicação web é quase nula. Deste modo, este simulador web será útil no contexto de ensino e aprendizagem, uma vez que os professores podem preparar e demonstrár aos alunos alguns esboços, e por se tratar de uma aplicação web, os alunos podem também experimentar o simulador em tempo de aula sem a necessidade de qualquer instalação ou configuração prévia.

Em suma, para implementar esta ideia, uma vez que o CNSS é implementado em Java, precisamos de escolher uma ferramenta que nos permita converter Java bytecode em Javascript para que se possa executar inteiramente em um browser sem a necessidade da interação com um servidor externo. A nossa ideia principal é implementar um notebook simples mas robusto onde qualquer pessoa possa alterar o código fonte do CNSS e experimentá-lo, visto que é o melhor ambiente de desenvolvimento para uma solução integrada. Para além disso, para melhorar ainda mais o CNSS, vamos reconstruir a saída textual das simulações usando gráficos ou estatísticas, como também vamos desenvolver um ambiente colaborativo onde cada usuário pode partilhar e ver em tempo real as modificações feitas por um professor ou colega de trabalho.

Palavras-chave: CNSS, Simulador de rede simples implementado na web, Simulação

Contents

List of Figures

List of Tables

Glossary

BBCode "BBCode is a lightweight markup language used to format messages" 34

CNSS "CNSS is a network simulator, written in Java, developed for teaching purposes, to make it possible to simulate computer networks and simple networking protocols." iii, iv, 2

WebAssembly "WebAssembly (Wasm) is a binary code that you can run in a browser." 6

XAML "XAML is a declarative markup language. As applied to the .NET Core programming model, XAML simplifies creating a UI for a .NET Core app." 14

1

Introduction

1.1 Context

As time goes by, the number and complexity of networks and devices continues to grow, so conducting new experiments to test these networks can be very inefficient if tested on an already existent one, since we want to test it in various scenarios instead of testing it on a particular network. Also, a real time experience takes time, but on the other hand, a simulation runs a lot faster since it runs on a single machine, depending on computational resources. Therefore, the need for a network simulator comes in handy when testing and creating new protocols or routing algorithms.

Currently, prototyping new network algorithms and routing protocols is hard to test and try in a real environment due to the extreme number of external factors that can influence their realization, like errors, latency, the characteristics of specific nodes and the pure complexity of the network. For this purpose, simulators are very useful when handling pre-testing since they can be used to improve visualization and test new hypothesis and ideas without the existence of unpredictable factors that would complicate the experiment or demonstration. Therefore, the use of simulators before a real implementation has a lot of advantages, the main characteristic being the deterministic results of a simulation under the same circumstances. In other words, if we run a simulation multiple times with the same inputs and conditions, then the output will always be the same. Simulators also have other advantages, such as being way faster than a real-life experimentation depending on the available computational power, but they also allow for a faster and easier way to find any undesirable condition or input. However, one must keep in mind that the results obtained from this simulator may not be the most accurate due to the already mentioned external factors that exist in a real environment but do not exist in a simulated one.

Although there are various network simulators already implemented, most of them are very complete and complex, with their main purpose being the testing of protocols in almost realistic scenarios. However, these simulators cannot be used in a pedagogic

environment since they have so many features that a student does not need to know about. Therefore, these kinds of simulators are not ideal for teaching purposes, so the need to implement a smaller and simpler one is very useful. These simple simulators allow students to mainly focus on the principal topics studied without wasting time on the details and complexity that are present in a more realistic scenario.

With this in mind, Professor José Legatheaux Martins developed a simple network simulator in order for any student or professor to be able to explore and experiment with some network scenarios, CNSS. This simulator can be used for various things, like implementing a new routing algorithm, testing protocols and even testing a network under high values of jitter. Using a simulator to test these ideas is an obvious solution since it is very easy to observe, analyse and interpret the results obtained instead of trying them in a real network.

Despite the fact that the results might not be the most realistic, CNSS was made for teaching purposes mainly, so professors could simulate a simpler network to show and highlight basic concepts without giving attention to unimportant details that exist in a real network. The demonstration of simple networks with the use of this simulator allows students to better understand, try and study for themselves the related course concepts.

1.2 Motivation

At the present time, most of us use different applications every day for different purposes. Any person with an internet connection and a respective desktop or web app can get informed about current events in another part of the world, have some entertainment or learn something new. Consequently, when starting the creation of a new application, one needs to choose between the development of a desktop application or a web application.

Although desktop applications were very popular some years ago, nowadays, we observe a migration from desktop applications to web ones. This shift probably happened because web applications do not need any installation and are way easier to access. Although this change occurred, most of the web applications are simple and introductive, but often have a more complex and complete application developed for the desktop. This behavior is normally very good for an application since it invites the user to try and experiment with it, but if there is a need for some advanced features, the user can install the desktop version. Since one of our goals in this thesis is the improvement and extension of the network simulator, CNSS, we need to choose in which environment we will be implementing it.

Although this simulator has already proved useful for teaching two courses of Computer Networks classes, in this thesis we have the objective of making it even better by implementing a web version of it since it has a lot of advantages when compared to a desktop application. However, as we will also see in chapter 2.2, if we opt to develop CNSS in a new language, we can have a multiplatform application that runs both on the browser and on the desktop. Despite the existence of frameworks that support the development of multiplatform applications with a single codebase, the principal objective of this thesis is still its implementation to run mainly on browsers.

The first advantage we can point out in a browser application is the fact that one does not need to install any software or allocate space to be able to run the application. With a single URL, anyone can access the application and run it, which is the main focus of the implementation of this simulator, since a professor can quickly run and create an experiment to demonstrate to the students and invite them to try and experiment with the simulator themselves and tinker with the code and the network parameters.

Therefore, this removes the barrier and loss of time in a class that is needed to install or setup a desktop application. Furthermore, these applications sometimes have some incompatibilities with Java versions, operative systems, or types of processors. The browser fixes all these problems and saves precious class time that can be used to stimulate and enrich students knowledge. Also, implementing CNSS on a browser allows room for future improvements and new features, which we can't do on an IDE since we are limited to its features.

Still, the browser is far from perfect since the architecture itself requires the execution of additional services that are not needed while running our simulator. This means that the performance of a web application is slower than a similar application on a desktop device. However, we believe that, despite this disadvantage, our application will run fast enough in a browser since it does not need extraordinary processing power. Despite this, there is another issue with browsers that we are attempting to solve, and that is the inability to compile and run Java code directly in the browser. Consequently, an application that wishes to do this normally needs to use an external server to provide these compilation services, which is what we are trying to avoid.

1.3 Proposal

Since our main target users are students or professors with at least some knowledge in Computer Networks area, the main functionality that our application will have will be the ability to customize the CNSS Java source code directly in the browser like it was a simple text editor. Also, in order to enrich the simulator, our web application shall be

able to transform the output of the simulation into something more readable through the use of graphs and statistics.

However, we start with source code that is written in Java, which is not directly supported to run in a browser, and so, we need to choose a tool that allows us to transpile Java code into anything that the browser can interpret and run. Choosing a tool might be tricky since most of them may have some undesirable characteristics that does not allow you to compile Java code directly in the browser. In other words, and the same problem might exist if we tried to develop in a new language, we would need the use of an external server to do the interpretation and compilation of the code that we would write in the browser.

Another feature that is of the most interest in a didactic environment is the possibility of a collaborative environment where a professor can make real-time simulations to show the students in a class. Also, in combination with this collaborative environment, a student can copy or import the professor's simulation and run it themselves, making it easier for the student to follow the class while avoiding any mistakes.

Concluding, since we want to maintain the application to run entirely on the client browser without the need for an auxiliary external server (excluding an external server for collaborative mode), this document is focused on finding the best way to do so and also finding the most promising visualization tools or APIs for a better demonstration of the simulator results.

1.4 Contributions

Our first contribution consists of the design of an online IDE for the CNSS simulator. This benefits professors, students and other members that might be interested in testing an online simple simulator. This idea also supports all the motivation and context already presented. With that said, our second contribution is the implementation of this integrated solution using one of the specified tools or frameworks analyzed in this document.

A third contribution is the development of a tool to enhance a simulation output with the use of graphs or statistics, making it trivial for a student to understand. Also, a fourth contribution is the design and implementation of a collaborative mode that allows students and professors to share the same development environment, thus creating an easy way to share and highlight code, classes, and whole simulations.

Consequently, our fifth and final contribution will be the experimentation and validation of our solution by picking already existing protocols and preparing some demonstrations with our IDE.

1.5 Document Structure

This document, besides this introduction, will be structured into four different parts, the 'Tools and Related Work', the 'Solution', the 'Implementation', the 'Validation' and the 'Conclusion'.

In this second chapter, we will be analyzing and presenting some of the possible tools or frameworks that we can use in our final solution. The approaches that we will study include the conversion from Java to Javascript and the development of CNSS in a new language. Also, in this chapter, we will analyze the best way to implement an integrated solution as well as a good way to build charts, graphs and statistics.

In the Solution chapter, we will demonstrate our thought process in the designing of our application. This chapter is also the skeleton that the implementation chapter will follow. With this said, in the fourth chapter, we will present our choices to follow our solution design. Also, in this chapter, we will see all the obstacles and barriers that we needed to overcome and how we did it while maintaining our initial design. In the fifth chapter, we will test the limits of our application and measure the average time of compilation.

In the final chapter, we will present our thoughts and conclusions about what we achieved with our application and also comment on good improvements and possible future work.

2

Tools and Related Work

CNSS is a Java implemented project. Therefore, in order to pursue our goal of developing its Web version, we need to find a way of converting it into runnable Javascript or WebAssembly. The first section of the this chapter will be focused on the analysis of the various tools that can achieve this. Tools like CheerpJ[2] and Dragome[3] could possible be the best way to approach this problem since they can possible convert and compile Java in the browser and therefore support an integrated solution. The further study of this tools will be focused on this first section while comparing them so that we can choose what can fit the best into our solution.

In the second section, we will try to analyse another path to approaching this problem. This path consists of developing our simulator in another more friendly browser development language, like Kotlin[4] with Kotlin/JS[5], C#[6] with the Uno platform[7], or Dart[8] with Flutter[9]. Since in this solution we develop in a different language than Java, it is reasonable to also convert the CNSS codebase to the respective language. Along with the analysis and comparison of these tools and languages, we will also see why this might not be a good idea due to the high odds of the need for an external server.

In addition, we will also analyze the best tools or APIs to enhance the output of the CNSS simulator. These tools shall be used to provide graphs or statistics for a better and easier understanding of this simulator's output.

Before starting, we want to make it clear that when we mention that we will implement an integrated solution, we are referring to our main goal of implementing a simple IDE or text editor for one to customize CNSS source code, compile and then run it in real time on the browser.

2.1 Converting Java to Javascript or WebAssembly

Since CNSS is a Java implemented project, we need to find a way of converting it into runnable Javascript to run it in a browser. There are different ways of achieving this since we can grab the Java bytecode of the compiled classes, *.jar* or *.class* files, or use the source code directly.

This first idea of **Converting the Java bytecode into Javascript** equivalent source code might be the one we are going to use to implement our project since it is the way that seems to offer the most reliability and scalability to medium-large projects, since it includes ported and pre-compiled libraries. These pre-compiled libraries can include the Java compiler, which is perfect for what we want to achieve since it seems like the only way to keep our simulator running entirely in the browser without the need for an external server. The only disadvantage that this solution may present is that it may require the load of a lot of files.

Since this is the path that seems to fit our needs the best, this chapter will be devoted to exploring all these different tools that can achieve this purpose:

- CheerpJ[2]
- TeaVM[10]
- Dragome[3]
- Bck2Brwsr[11]

Despite this, another way that can be used to implement our idea, includes **Converting the Java source code to Javascript**. This method transpiles Java source code into Javascript source code, potentially producing small code files, making it is easier for the developer to read the converted output. However, this way is not ideal since it does not support some Java libraries and potentially not support our integrated solution. The different tools offers this functionality are:

- JSweet[12]
- j2s[13]

There is a last option to approach this problem that is **Running Java code directly in the browser**. DoppioJVM[14] is the only tool that we found to pursue this path. This tool "emulates a lot of elements of the operating system, including filesystems, TTY consoles and threads". Unfortunately, we found out that this solution may result in a slower application and that there are better alternatives.

2.1.1 CheerpJ

This tool[2], as mentioned before, translates Java bytecode into WebAssembly or JavaScript. CheerpJ can be used both for converting Java small applications, but also for converting Java applets or Swing clients, being highly optimised and garbage-collectible. This array of choices makes CheerpJ a complete tool that can be used to convert any desired project. However, we are more interested in the pre-compiled libraries that can fit into our integrated solution. Consequently, CheerpJ fits our needs since it does not require any server-side support since all the application components that were converted are static including the pre-compiled libraries.

Another good aspect of CheerpJ is its ability to convert any type of Java application with Swing and AWT libraries. This feature can be used to build our IDE UI. Despite this, if we prefer to build our design outside of Java, CheerpJ also offers bidirectional Java to JavaScript interoperability, which will be of great use since we can invoke Java modules from Javascript. Although we are probably going to build our application UI with native HTML, this tool also supports *DOM* manipulation from Java.

Also, in the case of larger Java applications with reflection or where new classes are generated (called proxy class creation), CheerpJ also handles this by converting them on the fly. CheerpJ also supports the creation of concurrent applications recurring to Javascript WebWorkers, if our code needs to use threads.

Converting a Java Application With CheerpJ

We conducted a cursory analysis of how to convert a Java application with this tool because it appears to be a promising approach to our solution.The following is a demonstration of how to convert and deploy a small application.

After downloading CheerpJ, the extracted folder has a python file, *CheerpJfy.py*, which converts a *.jar* or *.class* file into a *.js* file.

Then, we need to create an HTML file to run the converted jar's of classes. This HTML is mainly composed of 3 calls, as shown below:

```
<script>
      CheerpJInit();
      CheerpJCreateDisplay(800,600);
      CheerpJRunJar("/app/app.jar", arg1, arg2);
      //CheerpJRunMain("simulator","/app/app.jar:/app/my_dependency_archive.
    jar");
      //CheerpJRunJarWithClasspath("/app/app.jar", "/app/my_dependency_archive
    .jar", arg1, arg2);
</script>
```

As previously stated, there are three ways to run the converted files, *'CheerpJRunJar'*, *'CheerpJRunMain'* and *'CheerpJRunJarWithClasspath'* functions.

The function *'CheerpJRunMain'* is used when our Java project is supposed to run the main method, while both of the *'CheerpJRunJar'* run the converted application as if we were using *'java -jar my_application_archive.jar'* to run it. It is worth mentioning that CheerpJ Runtime is allocated in a CDN controlled by the company.

The runtime API of CheerpJ can be found in this link: Runtime API[1].

2.1.2 TeaVM

Just like CheerpJ, TeaVM[10] is an ahead-of-time compiler for Java bytecode that emits JavaScript and WebAssembly that runs in a browser. Since the source code is not needed, TeaVM can also compile Kotlin and Scala classes, which might be useful in the future. Also, TeaVM is way more efficient in compiling the classes compared to CheerpJ, besides the smaller Javascript code generated.

As mentioned in the TeaVM documentation, TeaVM specializes in efficiently compiling and generating fast runnable Javascript code. However, this is not always possible since there are Java APIs that are hard to implement without generating inefficient Javascript, like swing, reflection and class loaders. Therefore, TeaVM restricts the usage of these APIs in order to maintain its main focus. To simulate these APIs, one needs to manually rewrite the code.

TeaVM also has a framework called Flavour[15], which helps to create applications with the restricted APIs defined above. Flavour supports a lot of functionalities and it is very similar to react or vue.js, besides the help that it supports for building a UI, which can be useful for designing our IDE.

In summary, TeaVM seems to be a reliable and efficient tool, but it does not seem as scalable as CheerpJ, since it does not support some Java APIs and might become inefficient for large projects. In any case, TeaVM would be a perfect tool to use for our main goal if we could convert the Java class compiler efficiently. However, we must raise the question of whether this is possible. Unfortunately, the answer to this question is probably no, since TeaVM only compiles maven projects, and therefore, we do not know the whereabouts of this Java compiler in order to develop our integrated version.

[1] https://docs.leaningtech.com/CheerpJ/Runtime-API

Converting a Java Application With TeaVM

In any way, since we are not totally sure about whether TeaVM has or not Java compilator pre-compiled, we analysided as we did for CheerpJ how we would run a small Java application with this tool. Running the following command, followed by a "mvn clean package"a generated war file is created. After this, we can unzip the file and open "index.html"to see the project.

```
mvn -DarchetypeCatalog=local \
  -DarchetypeGroupId=org.teavm \
  -DarchetypeArtifactId=teavm-maven-webapp \
  -DarchetypeVersion=0.6.1 archetype:generate
```

To use the Flavour framework in a TeaVM project we can run this command:

```
mvn archetype:generate \
  -DarchetypeGroupId=org.teavm.flavour \
  -DarchetypeArtifactId=teavm-flavour-application \
  -DarchetypeVersion=0.2.1
```

2.1.3 Dragome

Dragome[3] is very similar to CheerpJ in terms of features, one of them being the possibility to use Dynamic Proxies and Reflection APIs, which were restricted by TeaVM.

In general, the features that stand out in Dragome are:

- Integration of JUnit tests in a browser
- Callback Evictor Tool, to make async calls with no callbacks
- Use of Dynamic Proxies and Java Reflection API
- Very fast and incremental compilation

Within the compilation process made by the Dragome compiler, it is possible to add custom manipulations. One of these configurations was already mentioned before, the CallbackEvictor, that removes callbacks from async calls, and MethodLogger plugins, "Simple method interceptor for automatic model changes detection, it also make use of bytecode instrumentations".

Dragome, as well as TeaVM, only transpile maven projects, so it raises the same question as well. However, unlike TeaVM, Dragome seems to have some mechanism to customize its compiler, as we have seen in the previous paragraph. This might be a way to support our integrated solution. However, we do not know precisely if this is the correct approach to pursue since the documentation is not very clear about this aspect.

Besides this, one of the differences that CheerpJ has over this tool is that it supports Swing and AWT. Dragome, on the other hand, has a GUI Toolkit that contains a set of HTML visual components like Button, Checkbox, List, etc. This toolkit is well complemented with the template engine, capable of locating these components in HTML logicless templates.

2.1.3.1 Converting a Java Application With Dragome

As already mentioned, to use Dragome, we need a Java maven project with the correct dependency. After that, we can run this command:

```
mvn archetype:generate -DarchetypeGroupId=com.dragome -DarchetypeArtifactId=
    simple-webapp-archetype -DarchetypeVersion=1.0 -DgroupId={your-package-name
    } -DartifactId={your-app-name}
```

Dragome also seems easier to debug than CheerpJ or TeaVM since it has two different execution modes. One is the normal execution mode, which executes everything client-side, and the other is the debug mode, which executes everything in a Java Virtual Machine without compiling to Javascript.

2.1.4 Bck2Brwsr

Bck2Brwsr[11] is a Java virtual machine that compiles ahead-of-time the Java bytecode to JavaScript, producing a .js for each .jar file.

Unfortunately, this tool does not have any features that we could not find in the other mentioned tools. Bck2Brwsr was designed only for small Java projects, and its goal was to demonstrate that Java has benefits when creating larger HTML5 applications. Also, as well as TeaVM and Dragome, this tool needs to process the bytecode via its Maven plugin. As well as TeaVM, it does not support reflection, class loaders, network APIs, etc. Bck2Brwsr also does not support threads, which were supported by all of the above tools.

2.1.5 Which tool to choose

We can start by excluding Bck2Brwsr from our list of options since it is the option that lacks the most features and scalability. As shown in the table 2.1.5 below, all the remaining tools have their own strengths and weaknesses.

Pros	Cons
CheerpJ	**CheerpJ**
• Supports almost all Java APIs such as reflection and Swing	• Slow compilation compared to other tools
• Supports Threads	• Not the most efficient tool
• Usable for any size of the Java project	**TeaVM**
• Only needs the .jar file	• Does not support reflection, class loaders, Swing, and other APIs
TeaVM	• Lacks in scalability since it was designed for smaller Java projects
• Supports Threads	• Only compiles maven projects
• Very efficient and smaller code	• Probably unable to approach an integrated solution
• Faster compilation	**Dragome**
Dragome	• Only compiles maven projects
• Supports almost all Java APIs such as reflection and class loaders	• Probably hard to approach an integrated solution
• Supports Threads	
• Perfect for medium codebases	
• Balanced performance between CheerpJ and TeaVM	

Table 2.1: Pros and cons of all studied tools

If we dont consider an integrated solution, the decision would definitely be TeaVM since it was designed for smaller projects like CNSS. However, since our main goal is to have an IDE where we can change Java code and compile it on the browser, we might need to choose CheerpJ.

2.2 Developing in a new language

In order to develop the CNSS project in a new language like Kotlin[4], C#[6] and others, we need to convert the Java source code. This approach has some problems because maybe not all code can be converted automatically.

As a result, in this chapter, we'll talk about which languages or platforms we'll use to re-implement CNSS, how useful they are for web development, and how we'll convert our Java codebase.

2.2.1 Developing in Kotlin

Nowadays, Kotlin[4] is one of the most popular languages in android and web development since it is not only interoperable and similar to Java but also addresses some Java problems, such as nullability issues.

Interoperable means that both Java and Kotlin compile into the same bytecode since both of the compiled versions run on the JVM, which also means that Kotlin can call and execute Java code. Kotlin interoperability with Java is also perfect for this purpose since it is compatible with existing JVM libraries, Android and browsers.

Converting Java to Kotlin is easier than one might expect. Since JetBrains created this language, their IDE, IntelliJ[16], also contains the functionality to convert from Java to Kotlin very easily. Similar to IntelliJ, Android Studio[17] also offers a conversion between the two languages.

Kotlin/JS in Depth

If we plan to use Kotlin/JS[5] in order to make our IDE, we should do it using kotlin.js and kotlin.multiplatform Gradle plugins. This is the recommended way mentioned in the Kotlin/JS documentation since it is easier to control all the projects which target are JavaScript, for example, by adding dependencies from npm[18], in a single place.

Kotlin/JS and JetBrains IDE also provide the ability to build React[19] applications compatible with adjacent technologies like react-redux, react-router and others. With that said, it is possible that after converting the Java source code to Kotlin, we build our integrated solution with React.

2.2.2 Developing in C#

Currently, C#[6] is not a well-known language for web development since, in former times, it was only used to build .NET Framework applications.

However, in 2018, a platform called Uno Platform[7] was created, allowing the easy development of pixel-perfect and multi-platform applications with C# and WinUI[20], delivering consistent visuals on every platform. Moreover, applications built on Uno not only run on the Web (via WebAssembly) but also run on Windows, Linux, macOS, iOS, Android and Tizen.

Taking a deep look at how to convert Java to C#, we will find that there are some tools, like the Java to C# Converter by Tangible Software Solutions[21], that help us achieve this goal. However, although it converts our Java project guaranteeing very high accuracy, it also admits that there will be significant adjustments made to the outputted code. Therefore, the main objective of this tool is to reduce the amount of manual work that we would need to do if we plan on using C#.

Uno Platform in Depth

One of the main features of Uno, is that its API is compatible with Microsoft's UWP[22] application API, meaning that when an application runs on Windows, it is just a UWP normal application. Since our main target is the browser, this feature is not of great importance.

Although we convert our Java code to C# and run it in almost any platform with .NET, in order to build a UI with the Uno platform, things are done differently. Since Uno applications need to be able to run on many platforms, we also need to write a XAML file for this purpose. For example, on the web, each XAML element is converted into an HTML element, "Panels, controls, and other 'intermediate' elements in the visual tree are converted to elements...".

2.2.3 Developing in Dart

Dart[8] is a recent programming language developed by Google, with the initial objective of replacing Javascript. Nowadays, this language is mainly used to develop single codebase applications using Flutter[9].

Despite C# and Dart being two completely different languages, Uno Platform and Flutter are very similar in what goals they want to achieve. Both of the frameworks aim at the development of single codebase applications that run on all platforms without compromising quality, speed or performance. In the next section, we will spot the main differences between the two frameworks and try to choose the best one for our purpose.

The only way that we found to convert Java to Dart, is by using a "little" dart program, 'https://gist.github.com/sma/8180927'[23], which seems to be kind of limited. This Java-to-Dart converter, besides not being able to convert multiple files at once, the developer

of this converter also mentions that the program "... only translates syntax and does not attempt to translate Java library classes and methods to Dart equivalents". With this said, if we tried to convert a huge Java codebase to Dart, we would need to convert all classes one by one and possibly also need to change some code.

However, since the CNSS codebase is small and consists of a few classes, this Java-to-Dart converter might fit into our goals.

2.2.3.1 Uno or Flutter - Strenghts and Weaknesses

Both Uno and Flutter are considered good tools generally because any developed application can run on multiple platforms, most notably the UI, since the application's UI renders exactly the same on all targeted platforms, to the pixel. Therefore, in order to compare these two tools, we should highlight their pros and cons and also study each one's performance, scalability and documentation quality.

Before analysing the above criteria, it is worth mentioning that both of these frameworks, besides being open source, are well supported and maintained, making them highly unlikely to be discontinued or deprecated.

Starting by analysing Flutter's strengths, we found in some programming threads[24], that it is easy to learn and has exceptional documentation, making it very popular among developers. This framework is also well known for the development of high quality and expressive UIs due to its very well-documented and rich APIs.

Despite this, Flutter is not suited for smaller apps like the one we are trying to achieve. If we get ahead of ourselves, we can already compare and highlight that Flutter underperforms significantly compared to the Uno Platform, which is a deciding factor when building an application.

Uno on the other hand, besides having great performance marks compared to Flutter, its code is all reusable. In other words, Flutter and Uno Platform allow developers to write common code for multiple platforms, which is why it's called a "single codebase". However, Flutter has some platform-specific differences while coding, making code reuse limited in this framework since it is needed to add a lot of specific code for each platform.

The only negative aspect of the Uno Platform that we discovered was that its documentation and user base are very poor, which can lead to a more difficult learning curve.

2.2.4 Which language to choose

When it comes to choosing between Uno, Flutter or Kotlin/JS, we need to note that Uno and Flutter are way superior to Kotlin/JS due to the fact that Kotlin/JS is not a framework itself and does not run on multiple platforms.

In any case, the initial objective was only to care about the desktop version of our CNSS converted application since the main purpose of the web version is to test different protocols while changing some source code, and therefore, the mobile version would be somewhat useless. With this said, the argument of running on multiple platforms can be removed from the comparison.

Despite this, it seems that the Uno Platform and Flutter make it easier to build a UI, which can be a great deciding factor. However, it seems that both of them are harder to debug than Kotlin/JS.

However, in order to adopt our integrated solution, some pertinent questions arise: "How will our IDE compile the selected language code?"and "Can it run fully in the browser?". The first question is very simple to answer since Kotlin, C# and Dart offer ways to compile code online. Still, if we solve our first question with the compiler that any of the languages offers, we will be breaking our second question since this compilers makes requests to an external server so it can compile the input code. For example, it is mentioned in the Kotlin documentation, "It compiles code on our backend server and then runs either in your browser (if the target platform is JS) or on a server (if the target is set to JVM)".

2.3 Integrated Solution

As explained in the introduction, the integrated solution of CNSS into a browser is the main goal of this master's thesis since it allows students and professors to casually use this simulator anytime, anywhere with any device. This browser versatility, combined with this simple network simulator, makes teaching and learning much easier.

Therefore, we plan on having a small but user-friendly text editor or IDE, where anyone can change and modify the CNSS source code and then run it to see the results. Most programming languages now have online playgrounds, fig 2.1, where anyone can code and run in real time. With this said, our idea is to also make our IDE behavior like a playground as well.

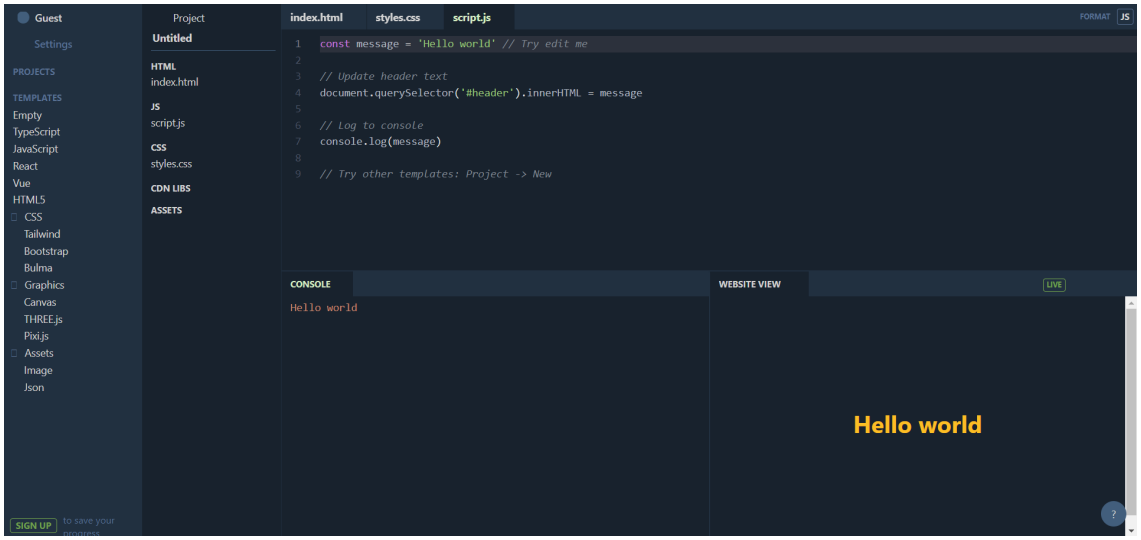

Figure 2.1: playcode.io: Javascript online playground

Along side with the analysis of all these approaches and tools, we mentioned the ability or difficulty of developing an integrated solution for each tool. However, in this section, we will take a more in-depth look at how we can approach this problem.

To start, we need to make sure we answer all the relevant questions about what we do want to avoid and what we do want to guarantee. Until now, we already partially covered some important aspects in each tool, like the ability or difficulty to implement an integrated solution, as well as if it would need an external server to work.

As mentioned in the section 2.1.1, this tool offers the most promising way to approach and integrated solution, since it already has most of the pre-compiled libraries that Java has, including its compiler. Therefore, running Java code without the need for an external server is possible. Despite this, we want to believe that the other tools, like TeaVM and

Dragome, also have Java pre-compiled libraries. However, we've not found a way to find the pre-compiled Java compiler so that we could use it in our solution. Also, TeaVM might not even have it pre-compiled, since we've seen that it lacks a lot of features.

On the other hand, we have another approach that converts CNSS to a new language, as we've seen in section 2.2. This approach seemed to be the hardest one since all of the playgrounds of this languages uses an external server to compile our code, and as mentioned in the introduction, this is not what we want to achieve.

If we sum up what we researched on these topics, we can draw some conclusions, as shown in table 2.3:

Most Promising	Least Promising
CheerpJ	**TeaVM and Dragome**
• Pre-compiled Java compiler allowes to run Java code in the browser	• Since we do not know if the pre-compiled Java libraries exist, we can try to pre-compile them on our own. However this might inefficient or take too long.
• CheerpJ Dom manipulation can be useful to report errors and build a UI	• Possible need of an external server to provide compilation services
	Developing in a new language
	• Need to change the source code to any other language than Java
	• No way of accessing the selected language compiler in the browser
	• Need of an external server to provide compilation services

Table 2.2: Integrated Solution: Reliable and Not Reliable Approaches

2.4 Visualization Enhancement

The CNSS output is a group of hard to read strings, and consequently, it is not very easy to analyze and study the results that the simulator outputs, as we can see in figure 2.2. Therefore, we need to integrate some visualization into our integrated solution with the use of graphs and statistics.

```
simulation starts - first processing step with clock = 0

log: sender time 0 node 1 starting pings
log: receiver time 0 node 2 started listening
log: sender time 1000 node 1 sent ping packet n. 1 - src 1 dst 2 type DATA ttl 32 seq 1 size 26
log: receiver time 1100 node 2 receiver received "ping 1"
log: sender time 1200 node 1  received reply "receiver received "ping 1""
log: sender time 2000 node 1 sent ping packet n. 2 - src 1 dst 2 type DATA ttl 32 seq 2 size 26
log: receiver time 2100 node 2 receiver received "ping 2"
log: sender time 2200 node 1  received reply "receiver received "ping 2""
log: sender time 3000 node 1 sent ping packet n. 3 - src 1 dst 2 type DATA ttl 32 seq 3 size 26
log: receiver time 3100 node 2 receiver received "ping 3"
log: sender time 3200 node 1  received reply "receiver received "ping 3""
log: sender time 4000 node 1 sent ping packet n. 4 - src 1 dst 2 type DATA ttl 32 seq 4 size 26
log: receiver time 4100 node 2 receiver received "ping 4"
log: sender time 4200 node 1  received reply "receiver received "ping 4""
log: sender time 5000 node 1 sent ping packet n. 5 - src 1 dst 2 type DATA ttl 32 seq 5 size 26
log: receiver time 5100 node 2 receiver received "ping 5"
log: sender time 5200 node 1  received reply "receiver received "ping 5""
log: sender time 6000 node 1 sent ping packet n. 6 - src 1 dst 2 type DATA ttl 32 seq 6 size 26
log: receiver time 6100 node 2 receiver received "ping 6"
log: sender time 6200 node 1  received reply "receiver received "ping 6""
log: sender time 7000 node 1 sent ping packet n. 7 - src 1 dst 2 type DATA ttl 32 seq 7 size 26
log: receiver time 7100 node 2 receiver received "ping 7"
log: sender time 7200 node 1  received reply "receiver received "ping 7""
sender received 7 replies to pings
receiver replyed to 7 ping messages
Pkt stats for node 1 :  s 7 r 7 d 0 f 0
(Node1:0 I1:0) s 7 r 7<-->(Node2:1 I2:0) s 7 r 7
Pkt stats for node 2 :  s 7 r 7 d 0 f 0
(Node1:0 I1:1) s 7 r 7<-->(Node2:2 I2:0) s 7 r 7
log: sender time 8000 node 1 sent ping packet n. 8 - src 1 dst 2 type DATA ttl 32 seq 8 size 26

simulation ended - last processing step with clock = 8000
```

Figure 2.2: CNSS simulation output

There are various ways to approach this visualization feature, one of them using the provided APIs by the tool we would use. For example, if we plan to use TeaVM, we can use the Flavour framework library to implement some visualization, or with Flutter, by using the XAML file. Most of the other studied tools also provide some sort of UI or visualization API that probably allows us to build what we intend.

The other similar way of approaching this is building the graphs with a Java library like JFreeChart[25] or EasyChart[26]. This approach can only be supported if we plan to use any of the tools mentioned in the first sections (CheerpJ, TeaVM or Dragome). This libraries offers a lot of different charts and has very rich documentation. However, the

conversion of these graphs using any of the mentioned tools is not certain and might not work.

This leads us to the final approach, which uses a Javascript library to produce our desired visualization. This approach allows us to export any result, besides probably being more efficient since we do not rely on the conversion of the charts. For example, we can use SciChart.js[27], which claims extraordinary flexibility and a number of features, besides the possibility to create dynamic graphs. However, despite the fact that this tool appears to be the best on the market, it is very expensive, and we will not use it because we can use something free and reliable, such as Chart.js[28]. We also need to take into consideration that these approaches need to be deployed to use them. For example, Chart.js needs to be installed with npm, which is not a problem if we plan to use CheerpJ since this tool supports the deployment of Node.JS[29] projects, but it is uncertain if we plan to use TeaVM or Dragome. We can also opt to use any library that we can install dynamically with '<script>' in our HTML file, like canvasjs[30].

2.5 Notebooks

A notebook interface is a development environment where a user creates code cells that can be executed individually. Today, notebooks are most commonly used for statistics, data science, and machine learning, using Python as its main language. For educational purposes, notebooks provide a digital learning environment since teachers can create a notebook page and document it with markdown cells for a better understanding of what the code is trying to achieve. These notebook pages can be exported to a file that anyone can import and test by themselves.

The most well-known notebook is Jupyter Notebook[31] which was built over a previous notebook, IPython. Other various notebooks were built over Jupyter like Google Colab[32], Amazon SageMaker and VSCode notebook. Also, since these notebooks are widely used for machine learning and data science, they can also provide computational power for harder compilations or executions.

Our idea could also be implementing our notebook over Jupyter. However, Java is not a "script"language which can run short code blocks and get a result, and although we did find a Jupyter notebook for Java, IJava[33], it does not seem the best approach. We need to make a notebook that compiles and executes Java similarly to an IDE but also provides all the didactic features a normal notebook does.

2.5.1 Collaborative Environment

Another feature that most of the cloud notebooks have is the ability to collaboratively code. This makes it easier for all of the participants to understand and follow each other, as well as easier to merge code and solve possible conflicts since all the participants are coding in a real-time environment. For learning purposes, this collaborative environment is also very useful since students can program with their group all sharing and watching each other's modifications.

We did find two real-time collaborative tools that might help us implement a good and robust environment without having to code everything from scratch. In table 2.5.1 we will analyse these two tools:

Convergence [34]	**TogetherJS** [35]
• Way more custumizable than TogetherJS • Easy to learn the basics but hard to master the whole documentation • Flexible identity, authentication, and permissioning for managing the users • Shared cursors, pointers, and viewports	• All in one tool for collaborating since the whole webpage is shared and not only what the developer defines • TogetherJS provides hosting • Shared cursors, pointers, and viewports • No authentication or permissioning

Table 2.3: Differences between Convergence and TogetherJS

As seen in this table, Convergence is a tool that offers us everything that we need, but with the drawback of being harder to include in our project since it might need way more code to implement. However, this tool, unlike TogetherJS, offers the possibility to manage users and permissioning which might be useful in a classroom environment where the professor is the host and students are in a read-only mode.

3

Solution

Besides CNSS being a simple network simulator with the intent of helping students in their learning phase, we still think that CNSS as it is, is not the best tool. Starting with the fact that it requires a lot of software installation, the IDE, Java Software Development Kit and CNSS source code. In order to fix this problem, we automatically thought of implementing CNSS on a webpage since it is more accessible.

With this in mind, we must devise a solution that allows us to do everything we could do on an IDE such as Eclipse while avoiding major drawbacks that would render this online tool useless. With this in mind, this chapter will be focused on the design process of the application.

3.1 Initial Thoughts

Because CNSS was originally just a Java source code project, it had to be downloaded from git in order to be used on its own IDE. Mirroring this behaviour, the first idea that came to mind was to build a somewhat similar Java IDE on the web.

One of the main problems of the original CNSS is that it is not an integrated solution, meaning that one student needed to make a lot of changes in different files and folders to execute a simulation, and also, the output of the simulation would be textual and would appear in the console. So, we need to create a development environment that is integrated and still has similarities with an IDE. With this said, and since notebooks are getting more trendy, the implementation of one that is available online is what best fits our needs since we can encapsulate every Java class, file and output in the same environment.

A simulation on a notebook is similar to a simulation on an IDE, the only difference is that each class is now one cell. This helps in our implementation since *config.txt* should also be a cell independent of being a text file. In addition, the implementation of a notebook should be simpler since it requires less control of what the user codes. Also,

notebooks add an important feature to our pedagogic tool, the markdown cells, which can be of great help in a student's studies.

3.2 Notebook

As mentioned, the use of a notebook is the most appropriate decision for our needs. As in all notebooks, some core features are obligatory to implement:

- Code cells and Markdown cells
- Ability to add a new empty cell
- Ability to delete a cell
- Ability to move cells

Besides these features, there are some that are not obligatory in a notebook environment but are very user-friendly:

- Upload multiple files and folders
- Delete the whole notebook cells and files stored
- Import and export notebook

We also need to think of what a cell should look like and how many different types of cells we are going to have. For this reason, four types of cells are crucial to our tool:

- Config cell type, to support the config text file of a simulation
- Java cell type, supporting Java classes
- Markdown cell type
- Console cell type, that supports the output of the simulation, similar to an IDE console

With this said, between these four types of cells, only two require extra implementation details. A Java cell, similarly to any IDE, requires some features that help the user to code, like a colour palette, gutter/line count, highlighting, auto-completion and others. The Markdown cell also needs special attention since we don't want a simple text input but a more robust text editor where the user can highlight text, change color, change font size, among other features.

3.3 Outputs of a simulation

A simulation puts its textual output on the console cell mentioned above. However, as initially proposed, we need to transform this output into a graphical output that is easier to read and understand.

Our idea is to build a graph/stack with all the nodes and messages traded between them, as shown in fig 3.1. This allows any user to better understand what is happening in a simulation. This type of graph is also very common in the student classes, making it the perfect choice to represent our textual output.

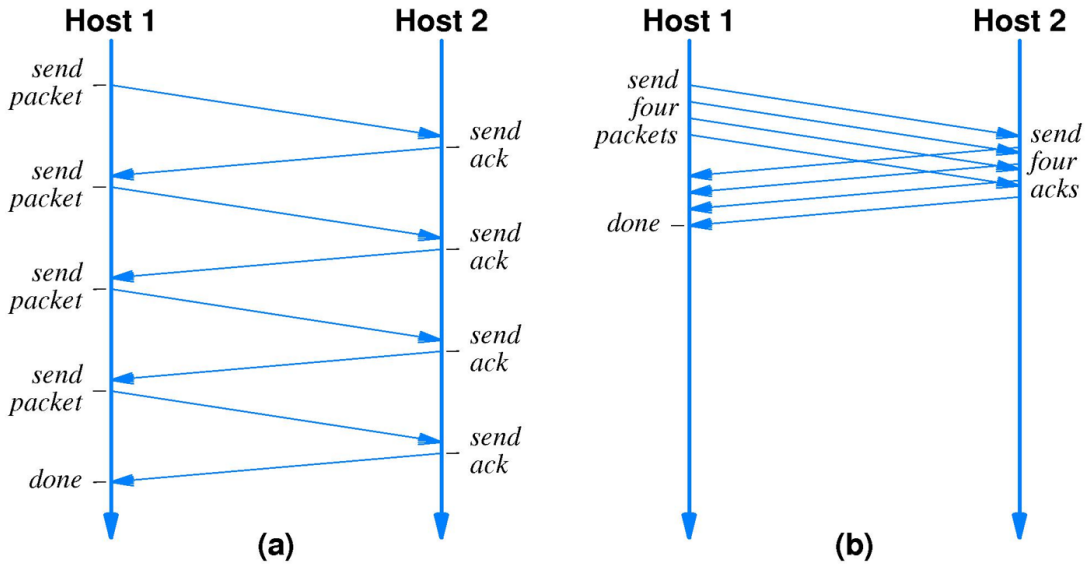

Figure 3.1: Graphical Output Idea

To represent our graphical output, we need to parse the text that is in the console and build it with the information that is displayed. The information that needs to be logged into the console in order to draw a simulation consists of each cell with its respective name/number and all the messages that were traded between the nodes. Also, each message needs to have some additional information:

- Origin node
- Destiny node
- Send time
- Delivery time
- Flag to verify if the message was dropped

3.4 Collaborative mode

Since the main target of this application are the students, it is a great idea to make it so that two different users can interact with the same project in real time. This allows an easier and funnier way of group programming that otherwise the students wouldn't get using Eclipse or any other Java IDE. With this in mind, two main ways of using this collaborative environment arise.

Classroom Mode

This first collaborative way is intended to be used in a classroom context, where the teacher is the host of the notebook and all the students can read and see what the teacher is coding or executing. In this environment, the teacher would get write/read permission while any other student would get read-only permission.

Project Mode

In this second collaborative way, a student can create a project and share it with their group, allowing for group programming. The user that would create the project would be the host and could give any other user permission to write or read.

It is also expected that this feature will be the most complex to implement since it requires some server-side code to maintain the structure mentioned above. Also, the collaborative mode needs to support in real-time at least all of the below-mentioned features in the notebook:

- Real-time cell editing
- Real-time highlighting
- Share of new files, like images
- (Optional) Real-time cursor

4

Implementation

4.1 Choice of Environment

Installed with npm, we are also going to use Bootstrap[36], an open-source front-end development framework, which allows me to build a prettier and more responsive UI without having to code all the website with native CSS. Besides this, I had already used bootstrap as well, so it was an asset to me.To start our journey to implementing our web application, we first need to choose where to implement it. Our main objective is to build a website with a UI that fully runs client-side, and for this purpose, there are various tools and frameworks that we could use since most of them offer what we are looking for. However, we decided to use React[19] since it is a framework that I have already used before, allowing me to capitalise on my previous knowledge. React also uses npm[18] to manage its packages, which is a package manager that is widely known and has a lot of useful packages that we might use. To complement, the runtime environment used by our framework is Node.JS[29].

After setting up React, we now have a localhost environment where we can implement all of our web application.

4.2 Editor choice

Before starting to use and try to run Java code with CheerpJ, we need to choose an editor where we can at least code with all the specified features of a cell mentioned in chapter 3 and some more:

- Syntax highlight for Java code
- Syntax checking for Java code
- Gutter/line count

- Text Highlighting
- (Optional) Auto-completion
- (Optional) Some other useful features ...

There are many text editors that can be used, however, this complex text editor will be the basis for our cell, so it is important to choose something that is reliable and UI editable.

The two most promising options are Ace Editor[37] and CodeMirror[38] Editor. These two editors overlap in all of our important features, so it is irrelevant which of the two we choose. We decided to use Ace since it seemed to us that it was the most popular tool between the two, possibly offering more support for future problems or bugs. We then proceeded to install it with npm, a react package that includes Ace[39].

4.3 CheerpJ Integration

4.3.1 Including CheerpJ Runtime

In order to use CheerpJ on our webpage, we first need to include the CheerpJ loader from their cloud (https://cjrtnc.leaningtech.com/2.3/loader.js). CheerpJ documentation mentions that this file is the only script that needs to be loaded to use CheerpJ, after that, this loader will automatically load all other files. However, this only works if we want to execute a Java application. In other words, we don't have access to a Java environment that we could use to compile and execute code in real time.

To see which files/packages we need to import to our project, we checked a demo made by CheerpJ[40]. In this demo, we can verify that there are two files, tools.jar and tools.jar.js, that are loaded into the scene. These files contain the pre-compiled Java runtime that we need to use to compile and execute Java code in real time, for example the compiler javac and the tool javadoc.

4.3.2 Adding CNSS to CheerpJ Environment

As mentioned in the last subsection 4.3.1, in order to compile and execute any Java code in real time, we need to include in our project tools.jar and tools.jar.js, the last one being the unpacking and compiling of the whole jar file into a Javascript equivalent. So, if all the tools needed to develop in Java are included in this tools.jar file, in order to further develop a simulation with CNSS, we also need to generate a jar file and a jar.js file for the CNSS code base.

So, after exporting CNSS to a jar file, we need to convert it into a jar.js file. This is where CheerpJ python script helps us do this, as shown below, *"using cheerpjfy.py is the recommended way of compiling applications and libraries using CheerpJ."*.

```
python /cheerpj2.2/cheerpjfy.py /CNSS/cnss.jar
```

However, simply generating the jar file of CNSS and then converting it to Javascript won't work since the Java version that needs to be used to generate the cnss.jar file needs to be the same as the version that was used to generate the tools.jar. Despite this, discovering the version was easy by generating a cnss.jar file with any version and then compiling any code in our web application with it while referencing any class in CNSS. This generates an error in the console where we can see the version that should be used. After this, it is just a matter of generating the right cnss.jar and including it in our React project, fig 4.1.

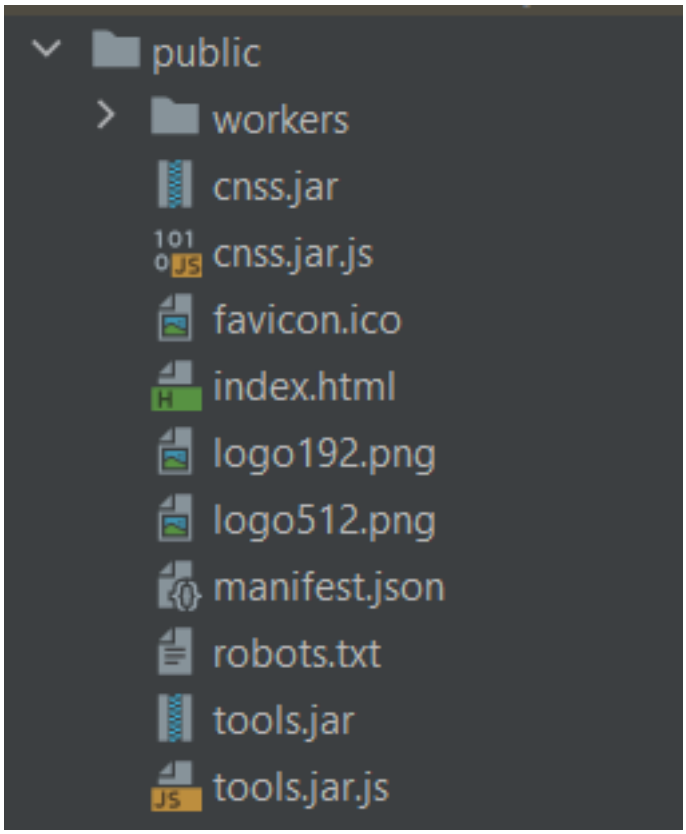

Figure 4.1: React Public Folder With The Presence Of tools.jar and cnss.jar

4.3.3 Running multiple classes with CheerpJ

In the source code from the demo displayed by CheerpJ, it is very simple to compile and execute a Java class.

First it is needed to map the written code to a location in the CheerpJ filesystem, as we see below.

```
    const classPath = "/str/packageName/className.java";
cheerpjAddStringFile(classPath, cell.code);
```

After adding the new class to the filesystem, we need to compile it by using cheerpJRunMain. This method, as we see below, uses the imported tools.jar to compile our class with the path "/str/packageName/className.java".

```
    cheerpjRunMain("com.sun.tools.javac.Main", "/app/tools.jar:/files/", "/str
    /packageName/className.java", "-d", "/files/")
```

However, this doesn't allow multiple files to be compiled at the same time since it requires every single class source code file to be passed by parameter to cheerpJRunMain. To solve this problem, we can use the Javascript built-in function *apply*, which allows us to call the function with arguments provided as an array.

```
    let newArgs = ["com.sun.tools.javac.Main", "/app/tools.jar:/files/:/app/
    cnss.jar", "-d", "/files/"];
for (let i in classFilesPathArray)
    newArgs.push(classFilesPathArray[i]);
cheerpjRunMain.apply(null, newArgs);
```

After compiling all the classes, we just need to run our main class. In the case of a simple Java project, like the demo provided by CheerpJ, it is easily done with another cheerpJRunMain like shown below.

```
    cheerpjRunMain("packageName.className", "/app/tools.jar:/files/");
```

Still, in our integrated CNSS version, we do not want the user to write a main file since it is already included in the CNSS source code, being the main class that is going to be executed by our application *cnss.simulator.Simulator*. We also need to pass another argument in our case, the *config.txt* file that contains the configuration of the simulation. This file also needs to be added with cheerpjAddStringFile to the CheerpJ filesystem but does not need to be compiled.

```
    self.cheerpjRunMain("cnss.simulator.Simulator", "/app/tools.jar:/files/",
    "/str/config.txt");
```

4.3.4 Displaying CheerpJ Textual Output

As mentioned in Chapter 3, we will need to construct a stack/graph out of a simulation textual output. This can be achieved by parsing the output and creating an object to build the graph, as we will see in section **??**. With this said, it might be easier to parse the output if we control the behaviour of the CheerpJ output function.

In Javascript, if we want to modify a CheerpJ function's behaviour, we just need to overwrite the original function with the modified one. So, we tried doing this to the function that displays the output of a simulation. However, this could not be achieved without modifying a lot of other CheerpJ functions, making this option not viable. Originally, this function displayed the output on the browser console or in an HTML element with the ID "console". So another approach, and the one we took, consists of creating a div with this ID and then creating a *setInterval* to check the content of it periodically and update it on our output cell.

4.3.5 Why do we need to use Web Workers?

After implementing the features mentioned before, we proceeded to test a small simulation, originating two bugs/problems that are crucial in any developing environment. We concluded that while the first simulation worked correctly, after updating any part of the code of the same class in a cell, it results in the same output as the first simulation. Also, another problem that arose was the fact that if we refreshed the page and then tried to compile the same code again, it would give an error saying that the class didn't exist.

The way that the CheerpJ demo got around these problems was by changing the package name each time the user decided to compile the code. This works in the demo because it is only possible to write one main class without needing to reference it anywhere. We could also try to fix our problems with the same fix, but it is incredibly ugly and complex to manage all the package names for each compilation, not to mention that in the config file the package needs to be referenced. With this said, we need to find a more practical way of compiling a simulation code, and so trying to comprehend the origin of the problems is mandatory.

Trying to understand all of CheerpJ's codebase is way too time-consuming. Still, we studied some of the source code from the most important files and we found out that it stores its filesystem on an indexedDB[41] as shown in fig 4.2.

This might suggest that the first problem, succeeding simulations resulting in the same output despite the differences in the code, is caused by the existence of this database. Afterwards, we also realised that what the indexedDB stores is already the compiled class files, confirming that this is the source of the first problem, possibly because CheerpJ does not re-compile files that already exist. So, one attempt that we made was to delete this indexedDB before each compilation in order to possibly force CheerpJ to compile each simulation individually. However, after applying this fix, any simulation after the first one resulted in the second problem that was mentioned before. This suggests that CheerpJ keeps some sort of object relatively to the compiled files on memory.

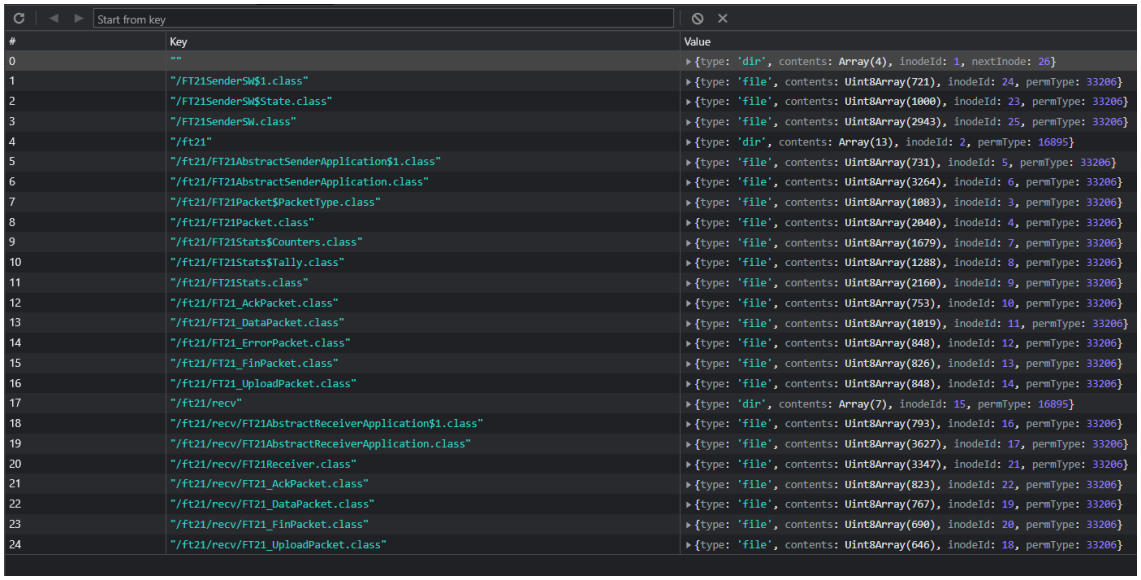

Figure 4.2: CheerpJ FileSystem IndexedDB

The only way we've found to solve this problem is to find a way to run any CheerpJ simulation in a different context/environment from each other. This can be achieved by using Javascript Web Workers[42], which are *"simple means for web content to run scripts in background threads"*. Web workers documentation also explains that a worker runs in a global context that is different from the current window context. With this said, it is just a matter of implementing all of CheerpJ's runtime to run independently in a web worker, and then exchanging data messages between the worker and our main thread to get the status of the simulation.

Summing up, in order to fix all of our problems, we just need to combine the two solutions that we gathered for each problem:

- Running each simulation independently in a Web Worker
- Before the compilation of a simulation delete the CheerpJ FileSystem IndexedDB

4.4 Notebook

As planned in section 3.2, we are going to build a notebook as the basis of our development environment. As designed, the main feature of a notebook is the ability to code in cells and run each cell individually. However, in our case, since this is a Java project, we do not want to run each class individually, but instead want to run the whole project with the defined config.txt. With this in mind, we need to build a UI that is intuitive for the user in the context of a Java project with CNSS already integrated.

4.4.1 UI Organization

To begin, there are various notebook UI types, but they all share some important similarities. The notebook, Google Colab [32] in fig 4.3, demonstrates a template UI for any notebook. With this said, we tried to follow the main characteristics of this notebook but modified them for our case.

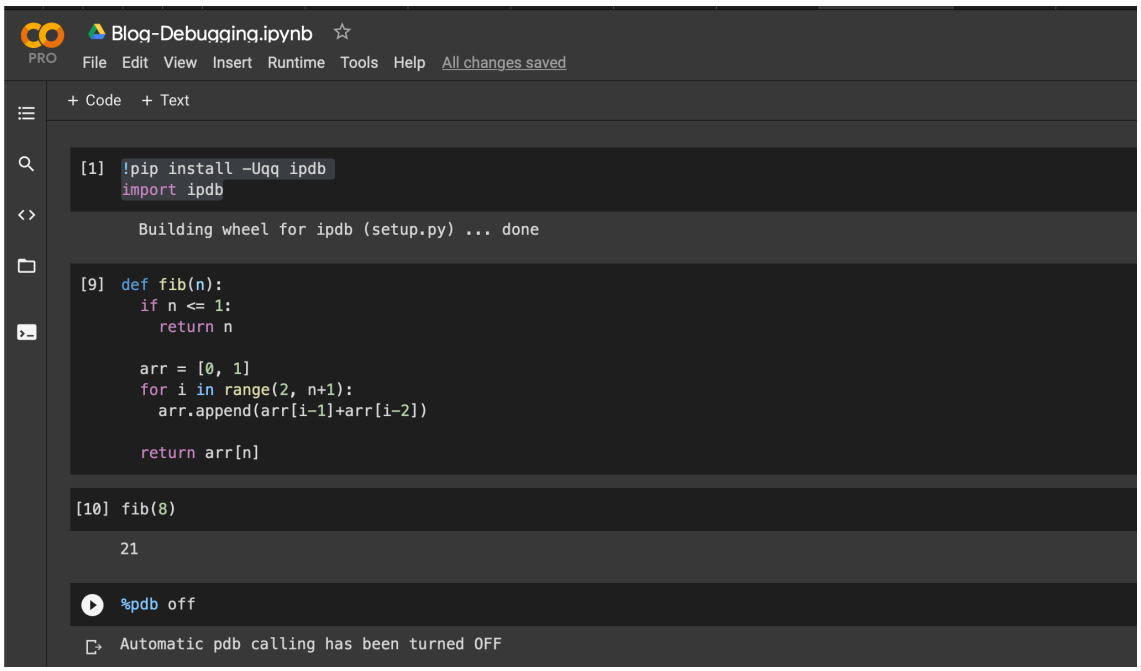

Figure 4.3: Google Colab

Looking from a global perspective, our UI is split into two big div's or spaces, the *main div* and the emph sidebar. This main div's purpose is to hold all of the development environment, or in other words, where the user codes and sees the results of a simulation. The main div contains:

- Config.txt cell at the beginning of the document
- Java and Markdown cells

- Textual Output (Console) cell
- Graphical Output

The sidebar, while not being as important as the main div, holds all of the secondary features for a more organised view:

- Add a new Java or Markdown cell
- Import or Export Project
- Upload files or folders
- Reset/Delete All cells and files
- A list of all files that are loaded
- Compile button

The last important part of our UI is the cell itself. Since there are mainly two different types of cells, Java and Mardown, these need to be different from each other, besides having different features. In any case, each cell in our notebook has its own delete button and two other buttons for reordering the cells as the user likes.

4.4.2 Cells

We already have chosen the text editor for our cells, it is just a matter of customising it as we want. The first thing to customise is the theme of the Ace editor. As we aim to have a dark website, since dark mode is the most popular among developers and programmers, the best theme that Ace offers is the theme called *Monokai*. Besides this, we tried to enable all the features that we already mentioned that are optional but important for a good development environment. Below are the settings that we used on Ace. It is worth mentioning that on the console cell, the readOnly and showGutter modes are different from the classic Java cell as the user cannot code on it.

```
showPrintMargin={false}
showGutter={props.cellInfo.style.gutter}
highlightActiveLine={false}
readOnly={props.cellInfo.style.readOnly}
enableBasicAutocompletion={true},
enableLiveAutocompletion={true},
enableSnippets={true},
showLineNumbers={true},
tabSize={4},
```

Markdown Cells

We did want to include markdown cells in our notebook, but we had no time to implement them from scratch since we wanted a markdown editor that was reliable and robust with some important features such as bold and italic text, changing font size, and most importantly, a feature to preview the markdown without the bbcode BBCode. With this said, we found a good markdown editor, SimpleMde [43], that offers everything that we were looking for. With this said, in order to adapt the original light mode UI from SimpleMde, we changed all of its source CSS files to match the Ace editor colors. We also tried different ways of managing onFocus and onBlur events from the markdown in order to show and hide the preview mode, but the attempts failed, resulting in a known issue documented on its github.

Overall, our UI, as shown in fig 4.4, seems well balanced with some room for improvement, but it still has all of the important features of a notebook.

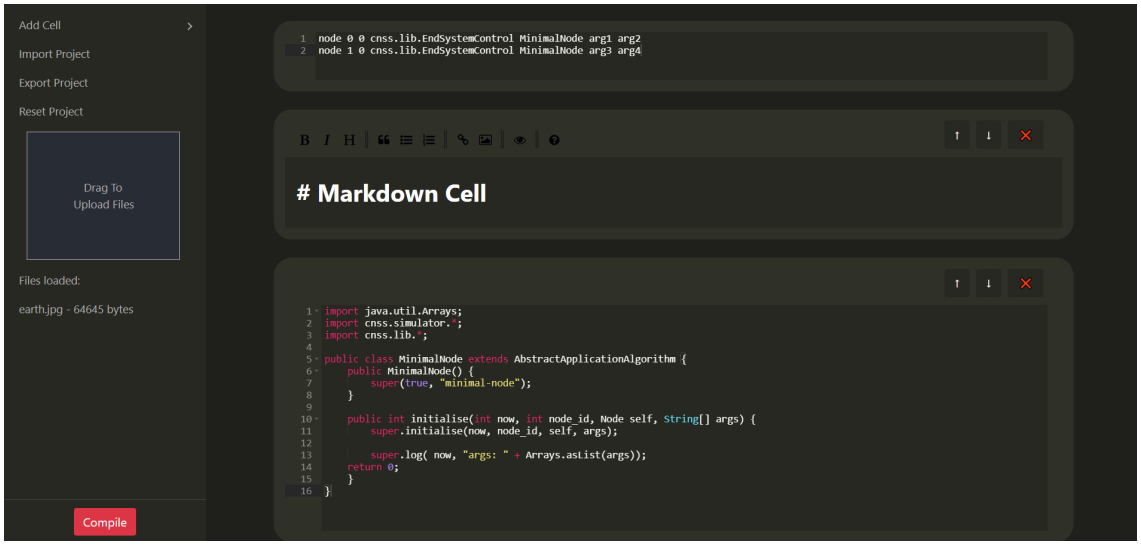

Figure 4.4: UI of Web-Based CNSS

4.4.3 Importing And Exporting

Uploading Files And Folders

As we were testing more complex projects, with more Java classes, we rapidily understood that copying and pasting each single class provided by the teacher or another user into a new cell is way to frustrating. So, we needed a way of at least uploading multiple files from the local computer.

Uploading files is as easy as it can get, the code just needs to accept the files in an HTML input and then read each of them with a Javascript FileReader object. However, uploading a folder is a little trickier due to some restrictions in HTML input element. To solve this

problem we used a react package named react-dropzone[44]. Using this package made uploading folders and multiple files as easy as only uploading files since this package reads all the files inside folders and so on. It is worth mentioning that it is possible to upload three different types of files:

- Java file - Creates a Java cell
- Txt file - Creates a Markdown cell
- Image file - Stores the file in CheerpJ FileSystem

Sharing The Notebook

Besides the ability to upload multiple files, it is hard for a user to share it's notebook with other users. With this in mind, we built a system where the user can simply export a string and send it to the other user so that he can import it. It is worth mentioning that images files will not be exported and therefore the user needs to manually upload them if needed.

4.5 Graphical Output

4.5.1 Parser

To build a stack, as mentioned in section 3.3, we must first parse the textual output of a simulation and create an object with all the necessary information, as also mentioned in the same section.

However, the textual output of CNSS doesn't provide any information about the messages that were traded between nodes. The only information that CNSS currently provides is which nodes were created and the links between them. The only way to get the needed information is to modify the CNSS source code to provide each message. The transformations that were made were simply logging any message that passed through a link. In CNSS source code, the link contains all the information that we need about the messages that pass through it. After modifying the CNSS source code to our needs, we just need to repackage it into a new jar file, *Cheerpjfy it* and then switch the.jar file and .jar.js file in the public folder for these new ones.

The parsing of the textual output is straightforward, we just need to find certain words of tokens that identify both the nodes and the messages in the textual output and then parse the information that comes next to these tokens.

4.5.2 Drawing The Stack

In order to build a message stack like in fig 3.1, we tried to find a package/library that drew the graph for us. However, these stacks are not very common and we could not find anything that could help us. The only option that remained was to draw it with some tool.

To draw lines and shapes on our website, we decided to use the HTML canvas element, which can be used to draw graphics via Javascript. An example of how to draw a single line is shown below:

```
1    let canvas = document.getElementById("myCanvas");
2    let context = canvas.getContext("2d");
3    context.moveTo(0, 0);
4    context.lineTo(200, 100);
5    context.stroke();
```

With this example, we can now build the nodes, which basically consist of a line vertically aligned. Also, besides drawing the name of the node on top of the line, in order for the graph to be scalable, we need to make some calculations on its position to have a symmetrical distance between each node regardless of its number. The formula below shows how the position of the node i is calculated.

```
    let space = this.canvasWidth / nodesObj.length;
    let pos = space / 2 + space * i;
```

To draw the messages, we need to draw an arrow between the two points of two nodes. To do this, we need to know at which time the message was sent and at which time it was received. In the code below, we draw the message between the two x axis values and two y axis values (fromX, toX; fromY, toY). We also need to apply and calculate the scale of the simulation time.

```
    let fromX = this.nodes[node].pos;
  let toX = dropped ? (this.nodes[node].pos + this.nodes[destNode].pos) / 2 :
    this.nodes[destNode].pos;
  let fromY = 75 + clock * scale;
  let toY = 75 + destClock * scale;
  let headLength = 10;
  let dx = toX - fromX;
  let dy = toY - fromY;
  let angle = Math.atan2(dy, dx);
```

In other words, if a simulation duration is long, a message can almost look flat in the stack, however, if the simulation duration is short, the arrow is way steeper. Long simulations with many messages might be hard to see and evaluate due to this behaviour since arrows seem to collide with each other as shown in fig 4.5.

This image also shows the existence of red arrows. These arrows represent a message that was dropped and therefore could not reach the destination. Despite the long simulation, a short simulation is way more understandable, as shown in fig 4.6.

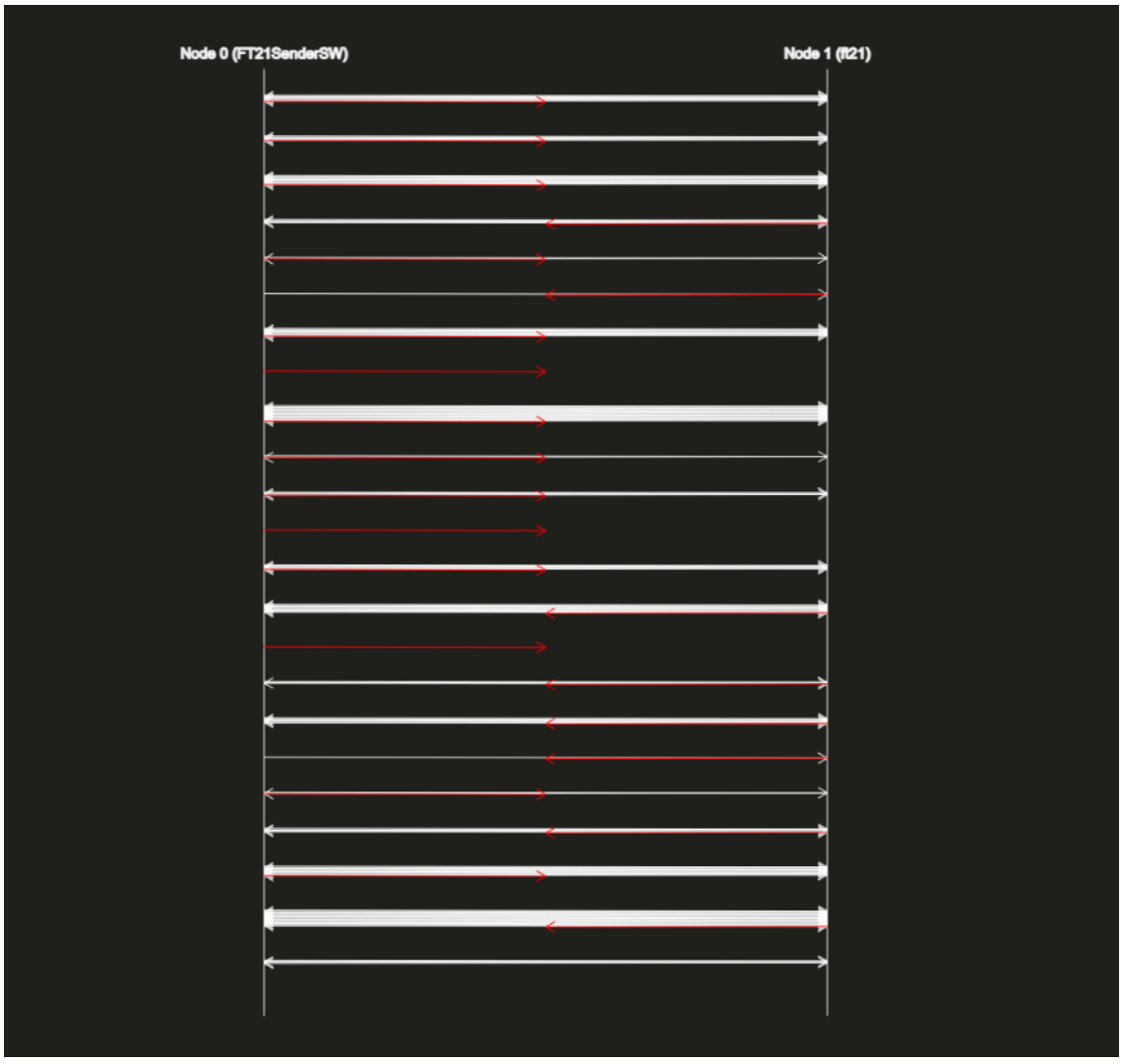

Figure 4.5: Stack of a Long Simulation

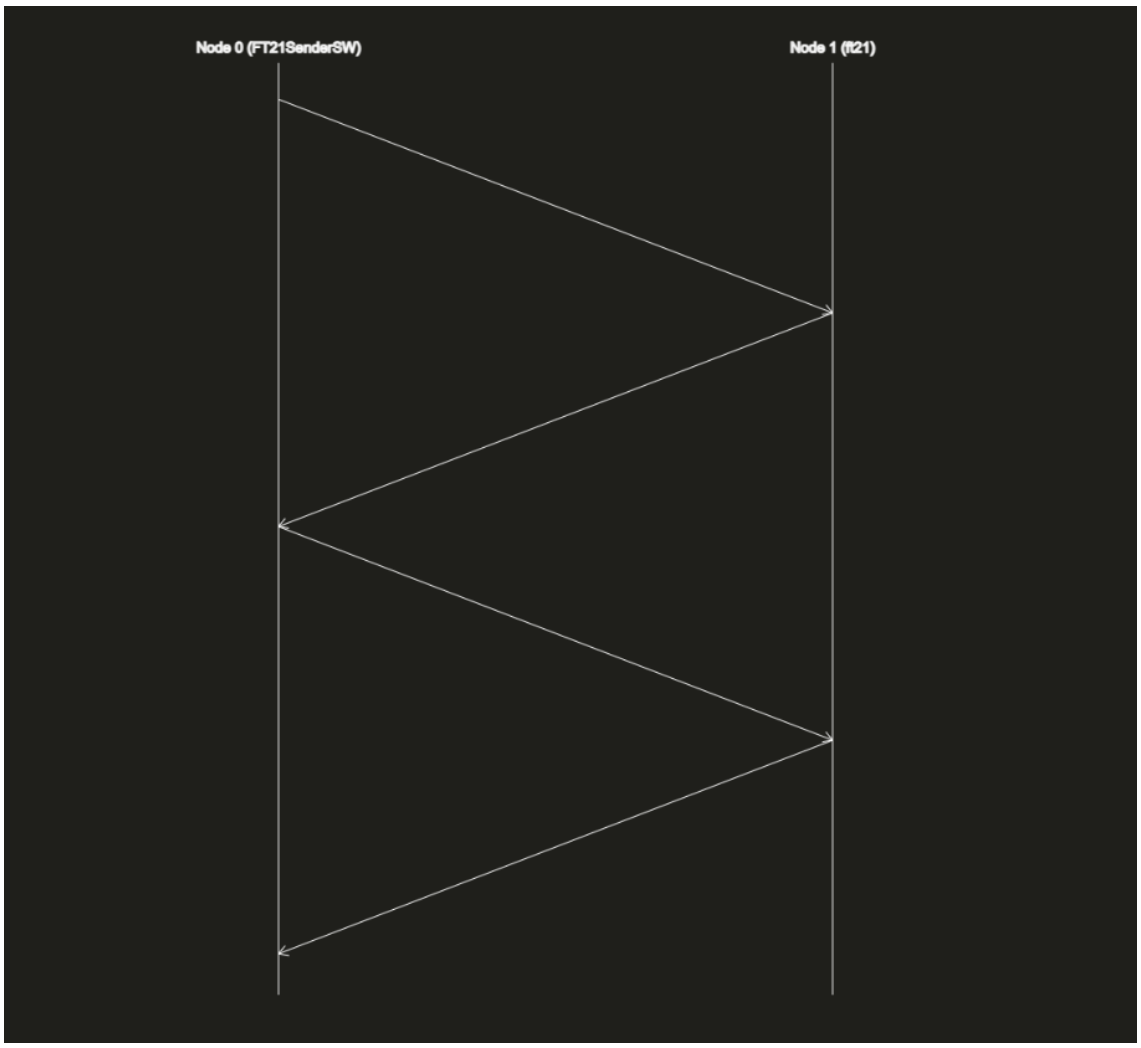

Figure 4.6: Stack of a Short Simulation

4.6 Collaborative Mode

Real-time collaboration is hard and complex to implement and maintain in every system that is known. There are a lot of areas like concurrency control and distributed systems that have been studying and researching effective ways of maintaining a consistent system between different peers. With this said, implementing from scratch a reliable and consistent real-time collaborative environment is not possible for this project.

However, we did find a tool, Convergence[34], which provides us with an API for real-time collaboration. This API, as stated in the official documentation, offers:

- Fine-grained, live editing of shared data
- See who is online and what they are doing
- Shared cursors, pointers, and viewports
- Flexible identity, authentication, and permissioning
- Coordinate through embedded messaging
- History and playback of editing sessions
- Robust offline support
- Commercial-friendly licensing

Although it looks easy, this API is very massive and complex with different features for what the user might need. This means that it would require some study of the documentation to fully understand the extent of Convergence, for which we do not have the time. Despite this, we still implemented a basic collaborative version with no control over what the user does in the shared notebook.

To start, we need to create a Docker[45] container to host the Convergence server. Convergence provides the Docker image, so we just need to pull it and run:

```
1    docker run -p "8000:80" --name convergence convergencelabs/convergence-
     omnibus
```

To connect to the Convergence collaborative environment, we just need to connect to the server that is hosted in the container. After that, we connect to a domain that will allow us to use the various services provided by the Convergence Server and only after that do we have our workable environment. We could proceed by using some sort of authentication, which Convergence supports, but it is not the point in a simple collaborative environment.

```
Convergence.connectAnonymously(CONVERGENCE_URL)
.then((d) => {
  let domain = d;
  return domain.models().openAutoCreate({
    collection: "WEB-BASED-CNSS-COLLECTION",
    id: "WEB-BASED-CNSS",
    ephemeral: true,
  });
})
.then((model) => {
  this.handleOpen(model, thisState);
})
.catch(error => {
  console.error("Could not open model ", error);
});
```

We are now connected to a real-time environment. Now we just need to have a way of telling Convergence to maintain and propagate the changes in all of our cells.

Convergence provides a website[46] that contains the source code of various examples that the developer might need. Two of the examples consisted of using the Ace editor and Simple-Mde text inputs. With this said, we just need to apply these examples to our project. However, this is easier said than done since in our notebook we have multiple text inputs and we can add new ones in real-time.

If we had only one ace text editor, we would need to just find the ID of the input element, get its ace session and document, and then add the necessary listeners (insert, remove, value, onChange) to the Convergence text model. Besides the onChange listener that merges the changes into the user document, the other listeners are what propagate the changes to the Convergence server.

After implementing this, we have a notebook that two simultaneous users can share and collaborate on in real-time. However, adding, removing or changing the order of new cells is not implemented. We did not have the time to implement this, but it should not deviate much from an input element. The only feature that possibly complicates this collaborative environment is the real-time sharing of other files like images. This feature would probably need another outside server to support BLOBs download and upload.

Summing up, a collaborative environment, even with the use of an external tool, is extremely complex since there are a lot of features that would need to be implemented to be usable in student classes, for example, the features that were mentioned in section 3.4. To fully finish the collaborative environment, access control should be implemented with Convergence, as well as the implementation of the usage of multiple domains at the

same time. Features like adding new cells, resetting the notebook, cursor sharing and highlighting in real-time also needed to be implemented.

4.7 Some Other Features

There are some features that could not fit into any of the above sections, so this section will focus on the extra features that were added.

4.7.1 Hosting Web-CNSS

One feature that we have tried to implement is hosting our web page online. However, since this is not to be used professionally, we should not use any paid cloud resources like Google Cloud or Microsoft Azure services.

One free hosting service that has been gaining popularity among developers in recent years is GitHub Pages[47], which can host a project directly from a GitHub repository. Hosting a React project on GitHub Pages is as easy as it can get, we just need to install *gh-pages* with npm, change the homepage on our package.json and deploy.

```
1    npm install gh-pages --save-dev
2    npm run deploy
```

4.7.2 Ability To Store Files Persistently

We've seen in section 4.4.3 how we can import files and folders into our project. With this feature, we are able to run simulations that might require large files like images. However, we cannot store these files on the browser LocalStorage, unlike we did with notebook cells, since their size is too big. With this said, we need to use the browser's IndexedDB to store any big file that is loaded and can't be stored on LocalStorage.

At each added file/image, we need to open the IndexedDB and store any file that is not stored:

```
1    let db = open.result;
2  let tx = db.transaction(indexedDbStoreName, "readwrite");
3  for (let i in context.state.otherFiles)
4      tx.objectStore(indexedDbStoreName).put({id: i, file: context.state.
     otherFiles[i]});
5  tx.oncomplete = function () {
6    db.close();
7  };
```

Loading the files from the DB at each document load is also trivial:

```
    let db = open.result;
  let tx = db.transaction(indexedDbStoreName, "readwrite");
  let store = tx.objectStore(indexedDbStoreName);
  let otherFiles = store.getAll();
  otherFiles.onsuccess = function () {
    let files = otherFiles.result;
    let finalObj = [];
    for (let i in files)
      finalObj.push(files[i].file);
    context.setState({otherFiles: finalObj});
  }
  tx.oncomplete = function () {
    db.close();
  };
```

5

Validation

Since the implementation is finished, it now remains to evaluate our application from a more practical approach. We need to test if the application runs as intended but also how long it takes to run a simulation. With it said, this chapter will be divided into two sections, analysing the aspects mentioned above.

5.1 Qualitative Validation

In order to test if our application runs as intended, we need to not only test if there are any bugs, but also test if the application outputs the same results of a simulation as the original CNSS. Accordingly, we proceeded to test our application with the StopWait protocol source code that was provided in the Network Systems 2021 course[48].

The first and most important thing that we verified was that the output of a simulation was identical both in the original CNSS and in our app. However, this is not entirely the case since we modified the original CNSS code to include in the output information relative to each transmitted packet. Despite this difference, everything else is maintained the same, therefore concluding that the output of our application is even richer and more informative.

We then proceeded to test every other functionality of our application. The easiest, safest and correct way of approaching the testing of our React application is by automating tests using any library that helps us mock and simulate interactions, like jest[49]. However, we did not have the time to do so, making the testing of our application manual. To at least improve this manual testing a little, we asked some outside users to run a simulation and try to find anything unusual. The results were positive and we did not find any undesired bugs or features in our application besides the missing features from collaborative mode. The users also mentioned some quality of life features that could enhance the project and therefore most of the features are mentioned in the subsection 6.0.1.

5.2 Quantitative Validation

In the original CNSS, compiled and executed with Eclipse, a simulation was executed and compiled between 0.5 and 1 second since all that was to be done was the compilation and execution, no new downloads or installations. However, this is not the case in our web application, but besides this, we consider that it is relatively acceptable if the whole simulation takes at most around 10 seconds. So, unlike a local IDE, in our web application there are various variables when it comes to time spent. Below, we will analyse how much time our application takes to run a simulation and identify each phase of it.

In order to calculate how long each part of a simulation takes, we just created an array that would store a time-frame for each segment. Below is the configuration file of a simulation that only creates two nodes and the time array that results from it.

```
node 0 0 cnss.lib.EndSystemControl MinimalNode arg1 arg2
node 1 0 cnss.lib.EndSystemControl MinimalNode arg3 arg4
```

```
[
    {
        "action": "Start",
        "time": 1664335184605
    },
    {
        "action": "Create Worker And Send Data To Worker",
        "time": 0
    },
    {
        "action": "Deleting database",
        "time": 32
    },
    {
        "action": "Compiling Simulation",
        "time": 1294
    },
    {
        "action": "Executing Simulation",
        "time": 4419
    },
    {
        "action": "Output of the Simulation",
        "time": 4574
    },
    {
        "action": "Output of the Simulation",
        "time": 5047
    },
    {
        "action": "Output of the Simulation",
        "time": 5559
```

```
33      }
34  ]
```

In our time array, we can see all the different stages of an execution. The first thing that our code does is to create the Web Worker, which is instantaneous. The next stage, inside the Web Worker, is to delete the IndexedDB database mentioned in Chapter 4, which took 32ms. The next two stages are compiling and executing the simulation code with CheerpJ, which took 1262 ms and 3125 ms, respectively. There are three more stages named *Output of the Simulation*, each one being a time where CheerpJ outputs any progress on the execution of the simulation. This stage can be considered part of the execution stage, which after summing up with the 3125ms noted before, took 4265ms in total.

With this said, we can point out that the stage that is holding almost all of the simulation time is its execution done with CheerpJ. Normally, a simulation taking this long, would only happen once since CheerpJ needs to load all of the files from the cache, like tools.jar. However, since our solution, to solve the problems mentioned in section 4.3.5, consists of isolating each simulation in a Web Worker, all the simulations will take approximately the same time as the one shown before.

Analysing from a global perspective, we can see that the whole simulation took 5.5 seconds to run, which compared with an original CNSS simulation on eclipse is 5 to 6 times slower, but despite this difference, we think that is an acceptable time.

Execution time on the GitHub hosted page

All the quantitative validation that we did before was made in the *LocalHost* environment. With this said, the times calculated above might not be realistic since there are some files, Tools.jar and CNSS.jar, that are already downloaded in a local environment but not downloaded in a hosted scenario. So, we proceeded to test the times of our application that is hosted on GitHub, with the same config file as before.

```
1  [
2      {
3          "action": "Start",
4          "time": 1664342939648
5      },
6      {
7          "action": "Create Worker And Send Data To Worker",
8          "time": 0
9      },
10     {
11         "action": "Deleting database",
```

```
        "time": 23
    },
    {
        "action": "Compiling Simulation",
        "time": 1347
    },
    {
        "action": "Executing Simulation",
        "time": 13338
    },
    {
        "action": "Output of the Simulation",
        "time": 13875
    },
    {
        "action": "Output of the Simulation",
        "time": 14372
    },
    {
        "action": "Output of the Simulation",
        "time": 14869
    }
]
```


This time in our time array, we can check as predicted that every stage remains the same except for executing the simulation with CheerpJ. This additional time comes from the download of Tools.jar and Cnss.jar that are hosted on GitHub. Despite the large execution times, the download of the .jar files are stored in the browser cache, which makes every next simulation approximately the same time as if it was made locally.

6

Conclusions

In a didactic environment, it has always been a challenge to find the best and most efficient way of teaching students. Therefore, the creation of auxiliary tools that help both teachers and students in this environment has been of the utmost importance. For our context, Professor José Legatheaux Martins developed a simple network simulator in order for any student or professor analyse and simulate network systems scenarios. However, this simulator comes with some drawbacks in a didactic environment. The fact that the students need to install Eclipse and download CNSS source code before even trying to test and experiment with the simulator is one of them. With this in mind, our thesis project consists of migrating CNSS to a more accessible place, the web.

However, migrating a Java project to the Web is not as simple as one might think since browsers do not compile or run Java code. Therefore, the need to convert Java to Javascript or WebAssembly is mandatory in order to run a Java application fully client-side. With this in mind, we searched for various alternatives on how to achieve this, the most promising being CheerpJ and Dragome. In section 2.1.5 we analysed every pro and con of the studied tools while making the conclusion that CheerpJ is the right tool to proceed with. We also studied other ways of compiling Java in the browser, section 2.2, however, these solutions were unclear if we would need an external server to compile the Java code.

Having a way of compiling and executing Java code in the browser, we needed to find the best development environment for our needs. There aren't many alternatives besides an IDE or a notebook interface, but despite the lack of options, a notebook seems the perfect choice for us since notebooks are most commonly used for educational purposes. Despite this fact, a notebook is also the best development environment for an integrated solution where every file, both Java classes, markdown cells, and the configuration file, are always shown to the user, making it easier for editing, analyzing, and documenting code. A collaborative environment in this notebook is also a great feature to add since it allows for a more real-time approach in the classroom context.

To develop our web application, we have chosen React, a well-known framework with great documentation and a large user base. After choosing React, we proceeded with the choice of text editor for our notebook cells. Ace editor and CodeMirror were the two most promising options, and altough the two being almost the same in terms of features, we decided to choose Ace editor since it seemed the most popular tool.

Having created a base for our web application, including CheerpJ runtime and CNSS in it, was the next step. However, since CheerpJ was not meant to be used as a compiler in real-time, various problems and inconveniences arose. One of the biggest problems was the fact that all the simulations output were the same as the first simulation. This problem, as mentioned in section 4.3.5, was solved using Web Workers, which allowed us to run each simulation individually.

A lot of user-friendly features were also implemented, like uploading files and folders or exporting the whole notebook to another user. These user-friendly features improve our web application in a didactic environment since a student does not need to waste time copying and pasting code to the simulator and instead can upload a folder or import the professor's notebook to start experimenting themselves. Also, the existence of markdown cells, allows professors to document the code in a more understandable way for the students.

Another feature that was implemented is the parsing of the original textual output from CNSS to a more graphical approach. As shown in fig 3.1, our idea was to create a graph/stack that was familiar to students and could better show what was going on in a simulation. Although we did not find any tool to help us draw these graphs, the Canvas element from HTML was all we needed to draw them simply, as shown in fig 4.6.

After implementing all of the base features for an offline version of CNSS, we then proceeded with the implementation of a simple collaborative environment. With this said, we created a Docker container with a Convergence server, a collaborative API that manages the concurrency and consistency of the system between connected peers. After having our container deployed, we transformed our Ace editor cells into real-time collaborative text inputs, creating the first step into a collaborative environment. However, the time is short and other important features like adding or removing cells in real-time were not implemented.

In order to test and share our project, we also hosted it on GitHub pages by simply installing a package and deploying it on a GitHub repository. With our project hosted, we could test our application in a different environment compared to our localHost. Still, this hosted version does not support the collaborative mode since we also needed to host the docker container somewhere for users to connect to it.

6.0.1 Future Work

The first and most important future objective for this project is the implementation of the basic missing features in our collaborative mode: Collaborative Markdown cells; Adding, removing or changing the order of new cells. Also, it would be wise if the collaborative mode supported at least the Classroom mode mentioned in section 3.4 so it could be used in a classroom by a teacher.

Some quality of life improvements would also be beneficial to further enhance our notebook. First, a way of minimising cells would be great for better organisation when the project has too many classes. Also, another feature that could be added is the auto-scroll to the output cell after a compilation. Despite these enhancements, another useful feature is the ability to manage multiple projects within the same notebook. In other words, a way of quickly changing between projects would be useful since students would have a project for each practical class.

As mentioned in section 4.5 and seen in fig 4.5, long simulations tend to have a graph/stack that is not easy to read or understand. A way of fixing this issue is to implement a feature which could zoom in the graph, so the user could clearly analyse the details in the simulation.

A last improvement, but not too important, is to find a way of reducing compilation and execution time. It is worth noting that this might not be possible without rewriting some of the CheerpJ source code.

Bibliography

[1] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/master/template.pdf (cit. on p. ii).

[2] https://leaningtech.com/cheerpj/. 10/11/2021 (cit. on pp. 6–8).

[3] http://www.dragome.com/. 2/12/2021 (cit. on pp. 6, 7, 10).

[4] https://kotlinlang.org/. 17/02/2022 (cit. on pp. 6, 13).

[5] https://kotlinlang.org/docs/js-overview.html. 20/01/2022 (cit. on pp. 6, 13).

[6] https://docs.microsoft.com/en-us/dotnet/csharp/. 17/02/2022 (cit. on pp. 6, 13).

[7] https://platform.uno/. 23/01/2022 (cit. on pp. 6, 14).

[8] https://dart.dev/. 17/02/2022 (cit. on pp. 6, 14).

[9] https://flutter.dev/. 26/01/2022 (cit. on pp. 6, 14).

[10] https://www.teavm.org/. 2/12/2021 (cit. on pp. 7, 9).

[11] http://wiki.apidesign.org/wiki/Bck2Brwsr. 3/12/2021 (cit. on pp. 7, 11).

[12] http://www.jsweet.org/. 10/11/2021 (cit. on p. 7).

[13] http://j2s.sourceforge.net/. 10/11/2021 (cit. on p. 7).

[14] https://plasma-umass.org/doppio-demo/. 10/11/2021 (cit. on p. 7).

[15] https://teavm.org/docs/flavour/templates.html. 2/12/2021 (cit. on p. 9).

[16] https://www.jetbrains.com/idea/. 17/02/2022 (cit. on p. 13).

[17] https://developer.android.com/studio. 17/02/2022 (cit. on p. 13).

[18] https://www.npmjs.com/. 17/02/2022 (cit. on pp. 13, 26).

[19] https://reactjs.org/. 17/02/2022 (cit. on pp. 13, 26).

[20] https://microsoft.github.io/microsoft-ui-xaml/. 23/01/2022 (cit. on p. 14).

[21] https://www.tangiblesoftwaresolutions.com/converters.html. 17/02/2022 (cit. on p. 14).

[22] https://docs.microsoft.com/en-us/windows/uwp/. 17/02/2022 (cit. on p. 14).

[23] https://gist.github.com/sma/8180927. 26/01/2022 (cit. on p. 14).

[24] https://www.c-sharpcorner.com/article/flutter-vs-uno-platform-which-is-the-better-cross-platform-development-platfor/. 17/02/2022 (cit. on p. 15).

[25] https://www.jfree.org/jfreechart/. 17/02/2022 (cit. on p. 19).

[26] https://www.drupal.org/project/easychart. 17/02/2022 (cit. on p. 19).

[27] https://www.scichart.com/javascript-chart/. 17/02/2022 (cit. on p. 20).

[28] https://www.chartjs.org/. 17/02/2022 (cit. on p. 20).

[29] https://nodejs.dev/. 17/02/2022 (cit. on pp. 20, 26).

[30] https://canvasjs.com/. 17/02/2022 (cit. on p. 20).

[31] https://jupyter.org/. 21/09/2022 (cit. on p. 20).

[32] https://colab.research.google.com/. 11/09/2022 (cit. on pp. 20, 32).

[33] https://github.com/SpencerPark/IJava. 21/09/2022 (cit. on p. 20).

[34] https://convergencelabs.com/. 16/09/2022 (cit. on pp. 21, 40).

[35] https://togetherjs.com/. 22/09/2022 (cit. on p. 21).

[36] https://getbootstrap.com/docs/5.2/getting-started/introduction/. 08/09/2022 (cit. on p. 26).

[37] https://ace.c9.io/. 08/09/2022 (cit. on p. 27).

[38] https://codemirror.net/. 08/09/2022 (cit. on p. 27).

[39] https://www.npmjs.com/package/react-ace. 08/09/2022 (cit. on p. 27).

[40] https://javafiddle.leaningtech.com/. 09/09/2022 (cit. on p. 27).

[41] https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API. 09/09/2022 (cit. on p. 30).

[42] https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers. 10/09/2022 (cit. on p. 31).

[43] https://simplemde.com/. 12/09/2022 (cit. on p. 34).

[44] https://react-dropzone.js.org/. 13/09/2022 (cit. on p. 35).

[45] https://www.docker.com/. 17/09/2022 (cit. on p. 40).

[46] https://examples.convergence.io/examples/ace/?id=181d712a-e444-47a5-8c8f-b0d462fbaacc. 17/09/2022 (cit. on p. 41).

[47] https://pages.github.com/. 18/09/2022 (cit. on p. 42).

[48] https://github.com/smduarte/RC2021-labs/tree/main/tp2/RC2021-tp2. 01/10/2022 (cit. on p. 44).

[49] https://jestjs.io/. 01/10/2022 (cit. on p. 44).