Imperial College London

AUTHOR // **MAURICE YAP**
SUPERVISOR // **DR ANANDHA GOPALAN**
MEng COMPUTING INDIVIDUAL PROJECT

Diorama

A portable web-based platform for testing distributed algorithms

Submitted in partial fulfilment of the requirements for the degree of
Master of Engineering in Computing of Imperial College London

Abstract

The study of fault-tolerating distributed algorithms — designed to run on networked computers (nodes) which communicate with each other only by sending messages — is common on university computer science courses. Students of distributed algorithms often implement such distributed algorithms in order to simulate them in a network of nodes in order to better understand them and to analyse their behaviour and performance. Many tools exist which allow students to do this, and there are also more involved other ways to test distributed algorithms such as implementing and running nodes from scratch in some programming language or setting up networks of *Docker* containers. However, such methods can be less than ideal since they may require users to use one particular language to implement algorithms, require root access to computers which may not be possible in a classroom or they may require users to spend lots of time learn to use other technologies which takes time away from learning about the algorithms themselves.

We present the design of *Diorama*, a browser-based application which facilitates: the writing of distributed algorithms in the user's choice of a range of available languages; the defining of a network topology which can include automatically-generated nodes; and the simulation of this distributed network. We create a proof-of-concept of *Diorama* to demonstrate its value and technical feasibility and publish a deployment tool which can be used to install and run *Diorama* from a cloud- or hypervisor-based virtual machine.

Acknowledgements

I would like to thank:

- my supervisor for this project, **Dr Anandha Gopalan**, for his excellent advice, guidance and support over the past academic year;

- **my parents** for enabling me to come to Imperial College London to study and to develop, and for their support during this time;

- those working both on the front line and behind the scenes to continually improve the teaching, facilities, academic support and student welfare provision of the Department of Computing;

- and finally, **God**, who is exclusively responsible for my academic gifts and abilities.

I also want to express gratitude to the vital financial support I received during my four years of study from the university's *Imperial Bursary* scheme. This funding has been vital for me and many of my peers to ensure that we minimise our financial burden on our already-squeezed families. It will continue to be essential in making sure future students from more economically-deprived backgrounds are not discouraged, for financial reasons, from coming to this institution to both study and contribute to student life. The *Imperial Bursary* scheme is something of which this university should be immensely proud.

Contents

List of Figures

List of Tables

1 Introduction

Distributed computing is an incredibly important and influential area of computer science. It is the study of systems of networked computers which communicate by passing messages between each other. Such systems include the World Wide Web, wireless sensor networks [46] and distributed database systems [20]. They commonly utilise distributed algorithms to perform tasks particular to distributed systems, such as reliably broadcasting data, performing atomic commit operations and achieving consensus across all the processes operating in a network [45].

Since distributed algorithms are an important concept in computer science, and commonly come up in software engineering, they are a popular topic for students to study. Students studying distributed algorithms will often need to implement and experiment with specific algorithms to study and observe their behaviour in a real-world context. Whilst setting up a real network of interconnected physical computers is a naive way to achieve this, it is impractical in many ways. Instead, a virtual environment simulating computers in a network, which can be run on a single computer, is often the most convenient way to test distributed algorithms.

Several network simulator software packages for studying distributed algorithms exist; we explore one example — *TETCOS NetSim* — in Section 3.5. Their focus, however, is running simulations of networks, and as such, they allow for a high degree of configurability for the networks themselves and the simulation of them; network simulators usually prove to be too bulky to set up and use for investigating distributed algorithms. This means that students would need to spend a lot of time setting these up, before they can focus on testing algorithms.

There are several advantages that virtual simulated networks have over physical test networks in the study of distributed algorithms:

- No extra physical resources are required, and thus has no extra financial cost for hardware will be incurred.

- Much less time is needed to deploy a simulated virtual network, both initially, and when modifications are made.

- Scalability is much better. It is much easier, cheaper and quicker to add extra computers to a virtual network than it is to a physical network.

- Making changes to the topology of the network (which computers are directly connected to which others - the shape of the network) is also much easier and quicker.

Another good way for students to test distributed algorithms is to create bespoke networks of containers or virtual machines, for example, *Docker* (Section 3.4). However, this has a high setup time cost and it is difficult to configure more complex network topologies. *Docker* is also a very powerful technology, and therefore has a steep learning curve for beginners. Again, this has the practical pitfall of being time-consuming and so is an inefficient method for students to learn about algorithms. Another practical problem with using *Docker* is that it requires root permissions on the host machine - something that may not be possible for students using shared laboratory machines.

1.1 Objectives

The primary aim of the project is to create a software package for testing out distributed algorithms, optimised for use by students. This means that a minimum solution should:

- Allow the user to configure their own network topology to simulate, that is, to program nodes and define how they are connected to each other.

- Simultaneously run all the nodes in their network to simulate its operation, and provide an interface to observe the output of all running nodes in real-time.

- Be simple and quick to set up and learn to use.

We want our solution to be portable - either easy to install on a wide range of platforms, or, avoiding installation difficulties altogether, hosted remotely on an external web server. We can draw inspiration from online IDEs, such as Repl.it [55], CodeSandbox [13] and Codeanywhere [19], which allow users to write and run code in a browser environment, and thus requiring no software installation whatsoever. A key advantage with their approach is that these services can be run on any machine with a modern web browser.

There exist several common network topologies, such as fully-connected, line and star. The solution should support defining topologies based on, or including such topologies.

1.2 Contributions

This project contributes *Diorama*, a software application which can be used by students and educators, like lecturers and laboratory demonstrators to study distributed algorithms through simulating them. We present in Chapter 4 the design of powerful and usable programming interfaces for (1) coding distributed algorithms in the form of node programs and (2) defining network topologies, which includes the capacity to automatically generate nodes and connections for common network topology shapes. We also present the design of a usability-focused web user interface for *Diorama*. We then create a working implementation of this design, which we describe in detail in Chapter 5.

2 Distributed Algorithms

Distributed algorithms are algorithms which are designed to simultaneously run on multiple independent processes, or *nodes*, each with their own local memory and clock. Nodes communicate and share information with other nodes in the system only by passing messages to each other. These algorithms do not assume the existence of a central coordinator [37], and typically are designed to tolerate failure of communication channels and other nodes (*faults*).

A simple example of a distributed algorithm is Eager Reliable Broadcast, which re-broadcasts every message which the node delivers. It ensures that every message which is delivered by a correct node, that is one which is not crashed, is also delivered by every correct node [21, slide 25]. We provide pseudo-code to illustrate this in Figure 2.1 [53, slide 18].

```
On start:
    set delivered to {}

On broadcast message:
    send message to all connected nodes except self
    deliver message
    set delivered to (delivered union {message})

On receive message from sender:
    if delivered doesn't contain message:
        send message to all connected nodes except self and sender
        deliver message
        set delivered to (delivered union {message})
```

Figure 2.1.: A pseudo-code representation of Eager Reliable Broadcast.

2.1 Network topology

The logical topology, or shape, of a distributed network, on which distributed algorithms would be run, can be modelled by a graph. Nodes each represent an independent process (*node*), and edges each represent a communication channel through which two nodes can pass messages to communicate [48]. For example, Figure 2.2 shows a very simple network topology, involving two nodes which are connected to each other.

There exist several common shapes of network topology in the study of distributed networks, which we illustrate in Figures 2.3, 2.4, 2.5, 2.6 and 2.7.

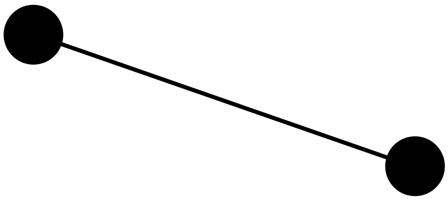


Figure 2.2.: A network of two connected nodes.

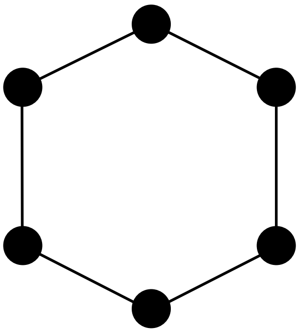


Figure 2.3.: A **ring** network topology with six nodes.

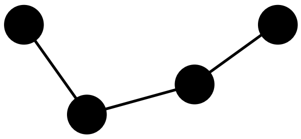


Figure 2.4.: A **line** network topology with four nodes.

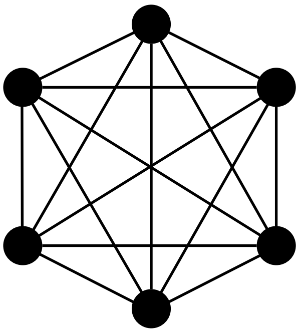


Figure 2.5.: A **fully-connected** network topology with six nodes. All nodes are connected to each other. Logically, this is equivalent to a bus network in physical networking.

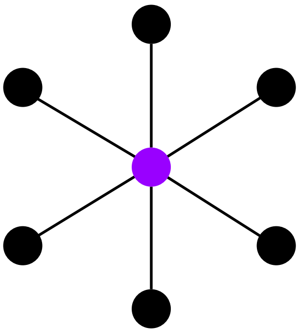


Figure 2.6.: A **star** topology with a hub (purple) and six hosts (black).

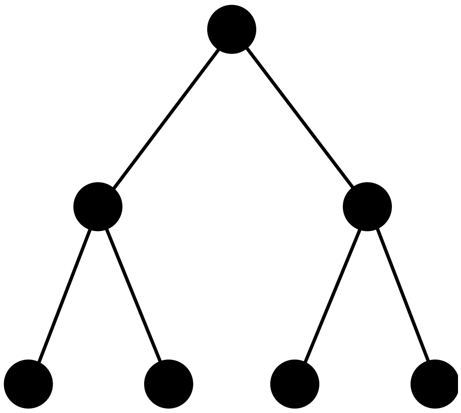


Figure 2.7.: A **tree**, or *hierarchical*, topology with three layers, and where non-leaf nodes each have two child nodes.

3 Related Work

We consider in this chapter examples of existing possible solutions for testing distributed algorithms, along with other relevant software packages and services. We analyse their suitability for this testing distributed algorithms and identify features they have which could be applied to our design.

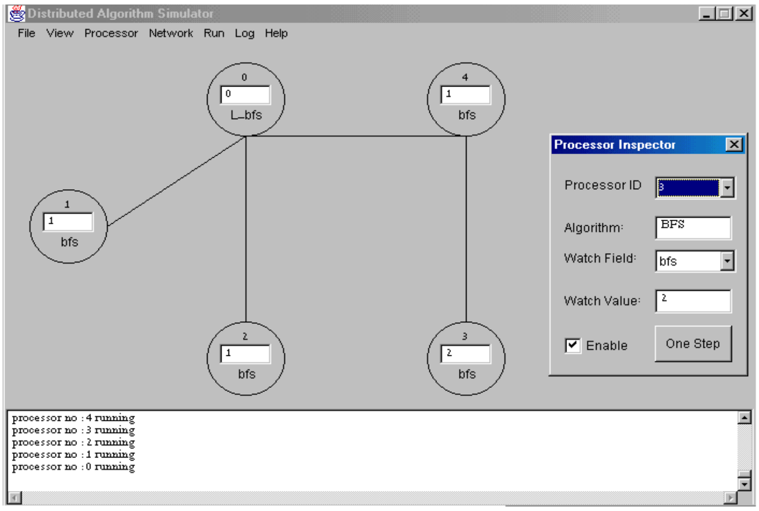
3.1 *A Simulator for Self-Stabilizing Distributed Algorithms*

Self-stabilising distributed algorithms work with nodes in a network, which each have one or more values. From any starting state (a configuration of these values in the network), running such an algorithm will cause the network to reach an allowed, or *correct*, state after a finite number of steps [31, p.97].

A Simulator for Self-Stabilizing Distributed Algorithms is a distributed algorithm simulator, created for the final BSc degree project of Oded Har-Tal [38]. It can be used to run user-defined algorithms written in Java (specifically, Java 1.1), focusing particularly on experimenting with self-stabilising distributed algorithms to understand their behaviour. Har-Tal's design enables users to:

- Simulate and observe the impact of different types of failures, including failed nodes and imperfect (failed or delayed) connections between nodes.
- Prove correctness of an algorithm by showing that it converges to a correct state.
- Observe the behaviour of algorithms across networks with different topologies.

The simulator displays a network in a visual graph representation (see Section 2.1). Each node's values can be inspected while an algorithm is running, and as well as this, a debug log is provided at the bottom of the window, where users can print messages.


Distributed Algorithm Simulator
File View Processor Network Run Log Help
0
0
L_bfs
4
1
bfs
1
1
bfs
2
1
bfs
3
2
bfs
Processor Inspector
One Step

Processor ID: 3
Algorithm: BFS
Watch Field: bfs
Watch Value: 2
☑ Enable

```
processor no : 4 running
processor no : 3 running
processor no : 2 running
processor no : 1 running
processor no : 0 running
```

Figure 3.1.: Screenshot from *A Simulator for Self-Stabilizing Distributed Algorithms* [39, slide 12].

Programming model

Each node uses a do-forever loop and is run as a separate thread, taking advantage of Java's thread management capabilities to ensure complete isolation between each node's computation — an important principle in distributed algorithms. To implement an algorithm, the user must define the behaviour of the algorithm in a single iteration of this loop, as well as any initialisation tasks by extending the provided `Processor` abstract class and implementing the abstract methods:

- `void initialize()`
- `void singleStep()`

Property values can also be added to each node, by creating objects which implement the given `Property` interface, then adding these to the `properties` field of the node (`properties.addProperty(Property property, String propertyName)`). The `Property` interface requires two methods to be implemented:

- `void setValue(String value)`
- `String getValue()`

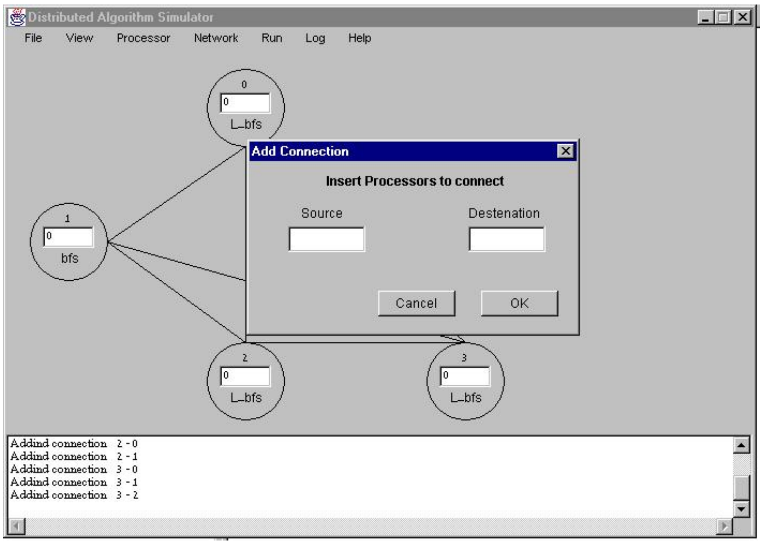
Four concrete API methods in the `Processor` class are provided to the user to communicate between other nodes and to output information to the debug log:

- `void send(int i, Object data)` - sends `data`, an instance of any Java object, to the node's `i`th neighbour.
- `void sendAll(Object data)` - sends `data` to all the node's neighbours.
- `Object receive (int i)` - receives data from the node's `i`th neighbour.

- `void PrintMessege(String msg)` - prints `msg` to the debug log.

Defining the network topology

The user must first create nodes, known as *Processors*, loading each with a compiled algorithm which they have implemented. Each node appears in the graphical user interface as a circle, inside which are its ID number (assigned dynamically by the simulator), the name of the node's first property and its initial value. The property displayed in the circle can be changed by using the *Processor Inspector* window, which can be seen on the right of Figure 3.1. The *Processor Inspector* window is opened through the *View* menu in the menu bar.



```
Addind connection  2 - 0
Addind connection  2 - 1
Addind connection  3 - 0
Addind connection  3 - 1
Addind connection  3 - 2
```

Figure 3.2.: The *Add Connection* dialog in *A Simulator for Self-Stabilizing Distributed Algorithms* [38].

The user can create connections between nodes by opening the *Add Connection* dialog window from the *Network* menu in the menu bar and then entering the id numbers of the two nodes to connect. We show this in Figure 3.2.

Useful features

Using this simulator, the user is able to send any Java object from one node to another. This makes it convenient for the user to use this simulator for algorithms where nodes communicate with each other using more complex data types, as opposed to simply strings or integers, for example. This is because the user does not need to encode complex data types before sending and then decode it when receiving. This feature does however mean that it is the responsibility of the user's node implementations to perform type-casting when receiving data.

This simulator also provides the ability to easily change things at runtime, when the simulation is running. This allows the user more freedom when experimenting with a distributed network and distributed algorithms. While a simulation is running and without needing to restart the simulation, the user is able to:

- load an existing node with a different algorithm.
- change the network topology by adding new and removing existing connections between nodes.
- add and remove nodes.

Drawbacks

One significant limitation of *A Simulator for Self-Stabilizing Distributed Algorithms* is that it only supports up to eleven simulated nodes. This makes it unsuitable for use with larger networks.

This software package is written entirely in Java, and users also implement algorithms using Java. Whilst this presents many advantages, such as the fact that users can take advantage of features from the entire Java language, a key disadvantage is that if someone wants to use this software, they must use Java. Users who do not already know Java must spend time learning the language, and some algorithms may be more difficult to write in Java, instead being more suited to languages with other features.

Takeaways

We should include a visual representation of the user's network topology as a graph, showing all nodes and connections as they are added.

We should provide a simple programming interface for the user, which is intuitive to use with as little time required to learn as possible. This should include API methods for communicating with other nodes and outputting strings to the log in our simulation viewer for debugging and other purposes as desired by the user.

There should be a high, or no, limit to the number of simulated nodes that our solution can support, so that distributed algorithms can be investigated over large network topologies.

Diorama should allow the user to perform modifications to their simulated network and nodes without having to restart the entire simulation. This could include adding and removing nodes, changing the program running on a node and modifying connections between nodes in the network.

Diorama should support the use of different types of programming languages for writing distributed algorithms to be run on nodes, for example, functional, imperative and object-oriented. Ultimately, it should be easy to generalise our solution to be theoretically completely language-agnostic.

The flexibility with sending different kinds of data structures between nodes is something we should strive to replicate, however, a solution which is language-agnostic will inevitably require some form of decoding and encoding of data in users' programs. Nonetheless, *Diorama* should use a message format that is as generic as possible and can easily be processed in most languages.

3.2 *JBotSim*

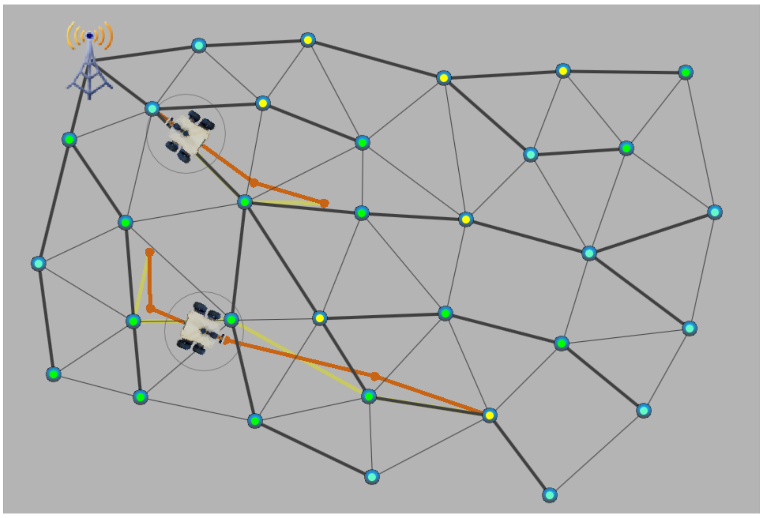


Figure 3.3.: Screenshot of *JBotSim* in action [16].

JBotSim is a Java library for simulating distributed algorithms in dynamic networks [16, 18]. It provides an interactive graphical interface and allows users to use create and simulate distributed networks using their own topology and node programs, which we show in Figure 3.3. *JBotSim* allows users to cause changes and actions in their network while nodes are running simulations by triggering the execution of some code on a node when it is clicked. *JBotSim* uses a network model which is better at modelling a real-life situation than many other distributed algorithm simulators, incorporating factors such as two- or three-directional positioning of nodes, wireless and wired connections, and directional wireless broadcasting of messages.

Running a simulation

```
1  import io.jbotsim.core.Topology;
2  import io.jbotsim.ui.JViewer;
3
4  public class Main{
5      public static void main(String[] args){
6          Topology tp = new Topology();
7          tp.setDefaultNodeModel(EmptyNode.class);
8          tp.setTimeUnit(500);
9          new JViewer(tp);
10         tp.start();
11     }
12 }
```

Figure 3.4.: Example code to run *JBotSim* [18].

As we show in Figure 3.4, *JBotSim* is run by instantiating the `JViewer` Java object (line 9), passing it a `Topology` object (lines 6 and 9), and then calling its `start()` method (line 10). Line 8 uses the `Topology.setTimeUnit()` method to make each time slice last 500ms. This can be done so that the graphical viewer displays changes at a pace which a user can easily follow.

Node Programming model

JBotSim uses an event-driven asynchronous programming model for node algorithms. All nodes are run on a single thread, coordinated by a central *clock*. Users implement distributed algorithms by extending the provided `Node` class and overriding the event handler methods:

- `void onStart()` - executed on initialisation.
- `void onSelection()` - executed when the node is selected (clicked on) in the graphical user interface.
- `void onClock()` - executed on each clock tick.
- `void onMessage(Message message)` - executed when a message is received.

The `Node` class provides the user with a large list of library methods, all well-documented in the *Javadoc* for *JBotSim* [17]. These include getters for many pieces of data, such as its neighbours and its colour, as well as methods to send messages, enable or disable wireless connections and move around the environment's virtual space.

Defining the network topology

Nodes can communicate using two types of connections, called "links": wired, which pass messages between two nodes no matter how far apart they are; and wireless, which pass messages between two nodes so long as the receiver is within the communication range of the sender. Links are implemented as directed, meaning if node *alice* can send messages to node *bob*, *bob* cannot necessarily send messages back to *alice*, given the distance between them is not a limiting factor. However, users have the option of making links either directed or undirected (which is simply an abstraction of two directed links going between two nodes in opposite directions).

```
Topology tp = new Topology();
Node alice = new Node();
Node bob = new Node();
Node charlie = new Node();
Node dorothy = new Node();
Node eric = new Node();
tp.addNode(100, 100, alice);
tp.addNode(200, 100, bob);
tp.addNode(300, 100, charlie);
tp.addNode(400, 100, dorothy);
tp.addNode(500, 100, eric);
Link ab = new Link(alice, bob);
Link bc = new Link(bob, charlie, Link.Mode.WIRELESS);
Link cd = new Link(charlie, doris, Link.Type.UNDIRECTED);
Link de = new Link(doris, eric, Link.Type.DIRECTED, Link.Mode.WIRED);
tp.addLink(link);
```

Figure 3.5.: Creating links in *JBotSim*.

Users create links in the network by instantiating `Link` objects, and then adding them to the `Topology`. The `Link` constructor takes two required arguments — `from` and `to`, both of the type, `Node`, followed by, optionally, the type (directed or undirected) of the link, and then, also optionally, the mode of the link (wired or wireless). In Figure 3.5, we show an example of this we have four nodes, *alice*, *bob*, *chalie*, *doris* and *eric*, arranged in a horizontal line 100 units apart from each other, connected in a line topology.

Useful features

The ability to interact with nodes live, as they are running, is a useful feature which gives users a very simple way to trigger specific events, namely the execution of the code in the `onSelection()` handler method. One of the examples of its use which the documentation gives is sending a message to trigger a broadcast throughout the network. The handler could be used to trigger various different situations, such as a network partition.

JBotSim also provides a very useful visualisation feature, in that users can make nodes change colour in the visualiser. This is great for visualising when certain things have happened in the network, for example, receiving a full set of messages.

The (currently incomplete) documentation on the project website [18] is quite extensive and provides full well-explained concrete examples, which are very helpful to readers and users trying to understand how to use the software package.

Drawbacks

JBotSim does not provide any way for nodes to print text onto the user interface, but users can use Java's `System.out.println` method to print to the console. Whilst this does work in that users are able to access this data, it is much less than ideal, since in order to obtain metadata into output messages, such as the node which printed it, users must manually put this into the content of their debug message.

Takeaways

Since one of our objectives is to create something which is easy to use for students of various levels of skill, knowledge and experience, good documentation is a key priority for *Diorama*. *JBotSim* demonstrates the effectiveness of including concrete code examples, accompanied by good, clear and concise explanations of it. We will accordingly make sure to include well-annotated concrete examples of code in our user documentation.

We will also display standard output from nodes in our interface in a way that users can view metadata about each line of output, including which node it was output from, and the time when it was outputted.

3.3 *DSLabs*

DSLabs is a framework for writing, simulating and testing distributed algorithms, developed at the University of Washington by Michael et al. [50, 49]. It is accompanied by an interactive visual debugger called *Oddity* [72], a graphical interface which users can use to inspect and manipulate their simulations of distributed networks with nodes running programs they have written.

The primary focus of *DSLabs'* functionality is very specific: to help users easily identify and fix bugs in their own distributed algorithm implementations. This makes sure that they meet their intended specifications under all possible faults and variations that could occur in a distributed network: delayed, dropped, duplicated and reordered messages. It does this through optimised execution-based and model checking tests.

Another stated goal for *DSLabs* was to be an accessible framework which is useful to novice programmers, so that users "can spend more of their time focusing on the subject material", as opposed to using the framework itself. We share this goal for *Diorama*.

Programming model

Users are presented with an asynchronous Java programming interface for nodes. Writing a node program involves extending the provided `Node` abstract class, and implementing three event handlers in the form of abstract methods:

- `void init()` - executed on initialisation.
- `void handleMessage(Message message, Address sender)` - executed when a message is received.
- `void onTimer(Timer timer)` - executed when a timer is received.

The interface also provides users with three API methods which they can use in their code:

- `void send(Message message, Address to)` - sends `message` to `to`.
- `void set(Timer timer, int duration)` - schedules `timer` to be re-delivered to itself after `duration` milliseconds.
- `void set(Timer timer, int minDuration, int maxDuration)` - schedules `timer` to be re-delivered to itself after some time between `minDuration` and `minDuration` milliseconds, chosen uniformly at random.

Nodes each run as a single-threaded event loop, so that the *DSLabs* model checker can explore the possible executions of its code using the coarsest granularity possible.

Takeaways

The creators of *DSLabs* used the application for teaching a class, by setting students a series of assignments using it to create a dynamically-sharded key-value store, which is correct, fault-tolerant and linearisable. Through this, they gained valuable feedback about the effectiveness and usability of *DSLabs*. Although due to timescales, it would almost certainly not be possible to replicate this in our project, part of our evaluation of *Diorama*, and of obtaining feedback to drive improvements, should be to have people, who are representative of its intended user-base, test it out with real tasks.

We learn from Michael et al. [50] that the *Oddity* network visualiser proved to be very helpful to students in helping to eliminate bugs in their code. While this is not directly applicable to *Diorama*, it does indicate that visualisations are helpful to users. Providing a graphical viewer to *Diorama's* users as they define their network topology in code would be useful.

3.4 Container orchestration: *Docker*

Containers, such as those provided by *Docker* [28], are lightweight and self-contained applications which provide the functionality of an operating system in which user programs run. *Docker* uses container images — executable files which become containers when they are run on *Docker Engine*, *Docker's* container engine. Images can be reused, allowing for multiple containers with the same program to be run simultaneously [29]. *Docker Engine* runs on Windows and Linux, and it is also included within *Docker Desktop*, which can be run on Windows and macOS [12]. Images can also be used across different platforms. Users interact with *Docker Engine* through its command-line interface and a RESTful API served over HTTP is available, with official SDKs in Go and Python and unofficial community libraries in many other popular languages [24].

Docker Compose is a tool for managing the running of such containers, which includes defining which images to use, how many containers to spin up and how containers are networked together. Users can configure an orchestration by declaring it in a `docker-compose.yml` or `docker-compose.json` file [30, 36].

Drawbacks

Docker Compose provides a very powerful and rich API for orchestrating containers. Its complexity, however, means that users must spend a significant amount of time and effort learning how to use it in order to write JSON or YAML configuration files to model networks through orchestrating *Docker* containers. These configuration files can also be long and complex to write for larger networks.

There are four network drivers provided out-of-the-box for networking Docker containers [26]:

- *Bridge* - all containers running on the same *Docker* host and connected to the same bridge network can communicate with each other; however a container cannot communicate with (and are therefore isolated from) containers which are not connected to its bridge network.

- *Host* - containers use their hosts' networking directly. The network isolation between containers and their host is removed.

- *Overlay* - multiple *Docker* hosts are connected together, allowing containers running on different hosts to communicate with each other.

- *Macvlan* - containers are each given a MAC address and connect directly to their host's physical network.

Since these drivers all create connections between every container in the same network, they do not, by default, support the creation of complex network topologies, such as a ring network. In fact, they all facilitate a fully-connected network topology, which is illustrated in Figure 2.5.

Additionally, whilst *Docker* can create networks for containers to communicate with each other over TCP or UDP, it is the responsibility of the application running in each container to implement the sending of messages to other containers on the network via these channels. In a context where *Docker* is used by students for testing distributed algorithms, this means that the student must implement this themselves. Whilst in higher-level languages, such as Python and Elixir, this is relatively straightforward [34], this is nonetheless additional work which does not directly benefit the investigation and understanding of algorithms.

Useful features

Being able to view the outputs of nodes in a network is important for analysing distributed algorithms, and having fine control over this, such as filtering and simultaneously viewing outputs from a selection of nodes, is very useful. *Docker* provides a very simple way to view the output of each container (logging) through a command-line interface. To view to outputs of multiple containers simultaneously, a simple command can be used if they were started using *Docker Compose*. Otherwise, this can be done using command-line utilities, or simply by opening multiple command-line windows [27]. Though this is powerful, it is partly reliant on the user having a good command of their operating system's command line. However for students who do not have a strong computing background, this may not be something they have.

Since *Docker* provides clear and extensive online documentation on how to do so [23], it is relatively straightforward for a user to create *Docker* images around their own programs to simulate nodes. Creating and running containers from these images ("spinning-up" containers) is very quick, and a very large number of containers can be run on a single *Docker* host running on an ordinary laptop [47]. This is useful to experiment with large networks to see how distributed algorithms behave when run at a large scale. Because *Docker* containers are built using layers of images, a user making a change to their program does not require the entire container image to be rebuilt, as only the top image layer will have been modified [22]. This makes it very quick for a user to make small changes to the code of their program and quickly re-run it without re-configuring any underlying support, such as networking.

Because *Docker* containers provide the functionality of a fully-fledged operating system, it is possible to use practically any programming language and platform to write programs to be run in one. The absence of a constraint on programming language is beneficial to students writing programs to simulate nodes, since they can choose, for example, the language they are most comfortable with, or the language most suited to the nature of their program. It is also possible to use different languages for different nodes.

Docker containers can each be stopped or paused (which allows them to later be resumed) [23]. This is useful for studying distributed algorithms, as it is a simple way to simulate a faulty or crashed node.

Takeaways

Diorama should provide a way to easily define common and more complex network topologies through a streamlined configuration API, or perhaps through a graphical interface.

We should give the user an interface to view the outputs of nodes, including the ability to simultaneously view a selection of nodes, filtering of output messages, and perhaps basic data analytics.

3.5 Network simulators: TETCOS *NetSim*

NetSim is a network simulation Windows software application designed for research and development of networks, and designing and testing new protocols. The product is aimed at both industry and education. In industry, *NetSim* is used by firms in sectors including defence, aerospace and public transport. It is also used for teaching in the curricula of over 100 universities, mainly in India, but also across 14 other countries [66]. *NetSim* is used in areas of research which include Internet of Things, mobile phone networks and wireless sensor networks [67].

NetSim supports designing network topologies through its GUI or XML config files and provides an animated visualised simulation, showing the flow of packets. Users are able to develop their own algorithms for nodes using *NetSim's* C API, although several existing algorithms are also given and can be modified. There are also extensive facilities for debugging. *NetSim* includes features for statistical and objective analysis at all levels of the network, such as built-in production of charts, graphs and tables, and integration with *MATLAB*, *Wirehark* and *SUMO* [66].

Drawbacks

Whilst as a network simulator, the purpose of *NetSim* is very close to ours of providing a service to test and simulate distributed algorithms, it has a few significant drawbacks to being used for this purpose.

Firstly, its focus towards simulating real-world networking means that using the low-level language, C, for users to develop their own algorithms is entirely appropriate. However, this is generally not a good choice of programming language for quickly prototyping distributed algorithms to see how they run. Performance is less of an issue, and benefits of using higher-level languages like Python, Ruby or Elixir, namely requiring fewer lines of code to express the same algorithm, far outweighs any performance benefits of using C.

As well as this, the focus towards real-world networking means that *NetSim* allows for and requires lots of configuration irrelevant to and unnecessary for testing distributed algorithms. We are not concerned with the network below the application layer when testing distributed algorithms, but *NetSim* provides configurability for all layers below this. This means that the set up process is long and bulky, again taking up time which could otherwise be spent on the algorithms themselves.

Away from the technical side, *NetSim* also has two minor practical downsides. It is only available as a Windows desktop application, which means Linux and macOS users must use workarounds. *NetSim* is also large in size, since it includes a vast array of features which are very useful for network research and development, but not so for distributed algorithms. For users, this is a redundant use of disk space and processing power.

Useful features

NetSim is a fully self-contained software application which has a single straightforward installation process. This makes it easy and quick for users to install.

A rich graphical user interface, as opposed to a command-line interface reduces the barrier to entry for users, especially those with less experience or confidence using the command line. However, *NetSim* mitigates the potential drawback of a GUI being too limiting, by also allowing users to write or generate their own network topologies in the form of declarative XML files.

The API for *NetSim's* algorithm development environment is very usable and well-documented.

Takeaways

We should strive to make *Diorama* support one or more higher-level and popular programming languages, and abstract away (or hide by default) non-essential configuration variables as much as possible to minimise setup time. Another good but less important aim is to make our solution cross-platform.

Diorama should have a straightforward or easy-to-follow installation process to minimise setup time and effort. We should also create an easily-navigable and self-explanatory interface, but provide more advanced users with deep functionality and configurability.

Our API should be well-documented and usable, built with strong software engineering best-practices.

3.6 Online integrated development environments (IDEs)

We explore two web-based programming environments, which although not designed nor usually used, for simulating networks or distributed algorithms, provide us with some useful ideas for our approach to designing *Diorama*. In recent years, with the continued increase in popularity of mobile working, many such services have appeared and become more developed. We select two popular online IDEs to look at: *Codeanywhere* and *Repl.it*.

Online IDEs are web services, hosted on the internet ("in the cloud") and accessed through a web browser, which seek to provide developers with all the tools they need to work on developing applications. They typically incorporate a code editor, a compiler or interpreter, and sometimes, a debugger, all of which are accessed through a web user interface. The main overarching benefit of online IDEs over desktop-based environment is their accessibility. Online IDEs typically only require a modern web browser and a (reliable) internet connection, and can therefore be used on practically any device, on any operating system and anywhere in the world. There are, of course, many other benefits which we will discuss.

3.6.1 *Codeanywhere*

Codeanywhere provides users with CentOS and Ubuntu Linux containers to use for developing their own applications [19]. These containers are provisioned, hosted and managed by *Codeanywhere*, and can be accessed through its web interface, which includes a terminal console, which connects to the container through SSH, as well as a feature-rich browser-based code editor for writing code on these containers.

As well as interaction through the web interface, *Codeanywhere* allows users to access their containers directly through SSH and SFTP, without going through their browser. This is useful for developers who want a more lightweight setup, or who simply want to use their own IDE or text editor program.

Codeanywhere provides users with several predefined common development environments with relevant software pre-installed to compile and run code, and to manage and install dependencies. These include PHP (LAMP stack), NodeJS (including yarn and npm) and Python. This helps users to start coding and developing their applications quickly with minimal setup.

Another useful feature of *Codeanywhere* is that its containers can import code directly from file hosting services, such as *Dropbox* and *Google Drive*, and code repository services, including *GitHub* and *BitBucket*. This gives users more freedom and means that they can deploy code without writing it on, or manually copying it over to their container.

In the same way as many online document editors, such as *Google Docs*, users of *Codeanywhere* are able to share their code, as well as collaborate with others in real-time over the internet. *Codeanywhere* allows users to generate links to a file, folder or even an entire project, which can be shared with anyone.

3.6.2 *Repl.it*

Repl.it provides users with environments to run projects in a range of runtimes. Like *Codeanywhere*, *Repl.it* provides a rich code editor, collaboration (which they call “Multiplayer”) and sharing projects using links. It is however much more restrictive in that it does not allow users to access the underlying operating system in which their code is running.

In addition to these environments, *Repl.it* has a feature called *Classrooms* — a platform for educators to create “classroom” environments, which they can invite students to. It allows for collaboration between teachers and students and also provides the ability to track progress and automatically grade students’ code using automated tests.

Another feature of node is that *Repl.it* enables users to embed projects into web pages, so that visitors to a web page can view, edit and run the live code.

Evaluation of online IDEs

One of the most prominent drawbacks to using cloud-hosted IDEs, as with any cloud-hosted web service, is that it requires a constant internet connection. Whilst in a classroom environment, where there is, for example, good wired connections and wireless internet coverage, it can be a problem when travelling, or in places where internet access is not reliable.

Another consideration which users must make when using online IDEs is performance. The speed of code compilation and execution is entirely dependent on the IDE provider, since these tasks are all run on the provider's machines and environments. Whether or not performance is a positive or negative thing for an online IDE user depends entirely on whether or not the alternative available to them is more powerful. A key benefit of using online IDEs is that they can be used on devices regardless of performance, since the only processing required is to run and display the web application for the user interface. It means that mobile devices and low-cost machines are no less suitable than high-performance ones. Another benefit of this is that users can easily switch physical machines and seamlessly continue developing with the same code and environment as before.

Related to this is that project dependencies like external and third-party libraries are taken care of and installed onto cloud environments and not the user's local machine. This not only means that they save disk space on their machine by not reeding to install these, it also means it is much quicker and easier to get started with developing, since they do not need to install any tools or runtime environments to compile or run their code. Users can literally log in and start developing straight away.

3.7 Serverless/FaaS: *AWS Lambda*

We take a brief look at *AWS Lambda* [8], an example of an online function-as-a-service (FaaS), or so-called serverless, offering, since we can learn and apply principles from it when designing *Diorama*'s interface for users to code programs for their nodes.

AWS Lambda is an event-driven platform provided by *Amazon Web Services* (*AWS*). Essentially, it runs, or *invokes*, a user-programmed handler when triggered by some event. These triggers, determined by the user on the screen shown in Figure 3.6, can include HTTP requests, timers and events on other *AWS* services. A *Lambda function* is a combination of one or multiple triggers, and a handler.

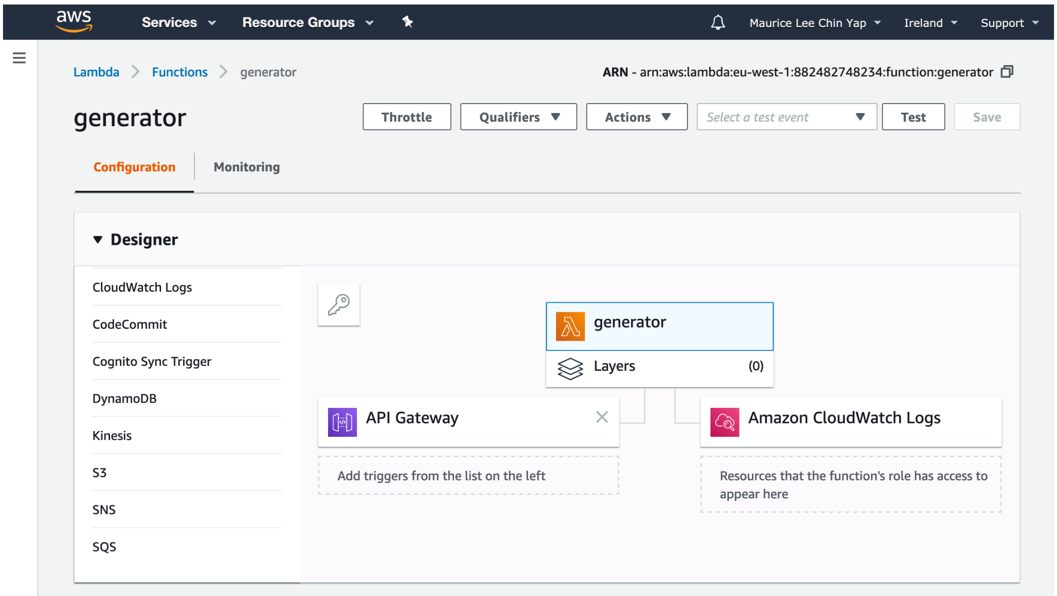
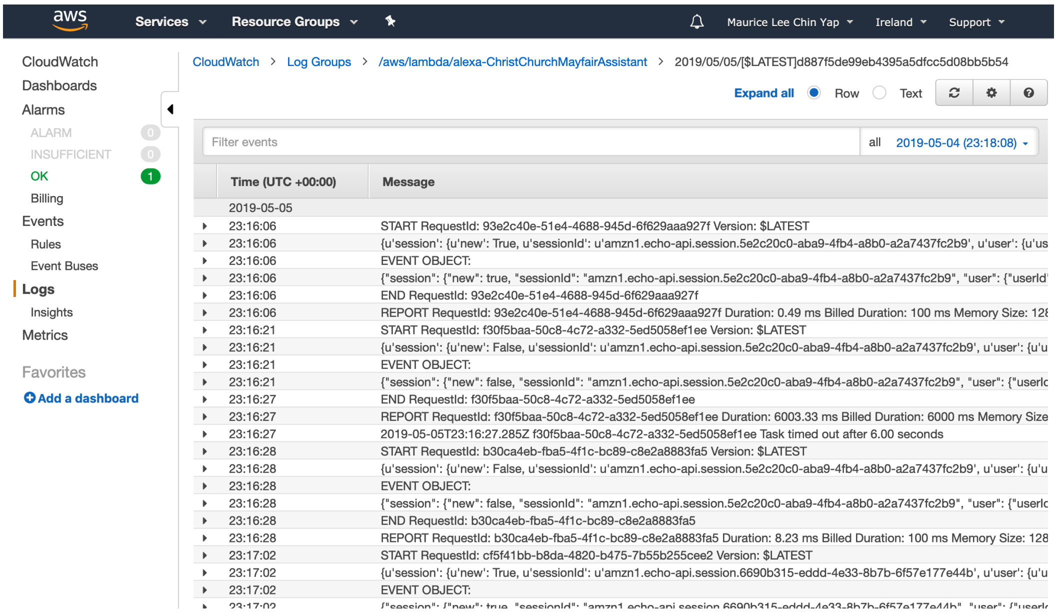


Figure 3.6.: Screenshot of the function designer for *AWS Lambda* [6].

Users create a Lambda function handler by writing a method which takes two arguments: `event`, which contains data about the event which triggered the invocation of the function; and `context`, which contains data about the runtime of the function. The return value is passed back to the trigger where this is relevant, for example, if a Lambda function is invoked due to a HTTP request, the value it returns is its response. *AWS* provides very extensive and detailed documentation for creating Lambda functions and their handlers [9].

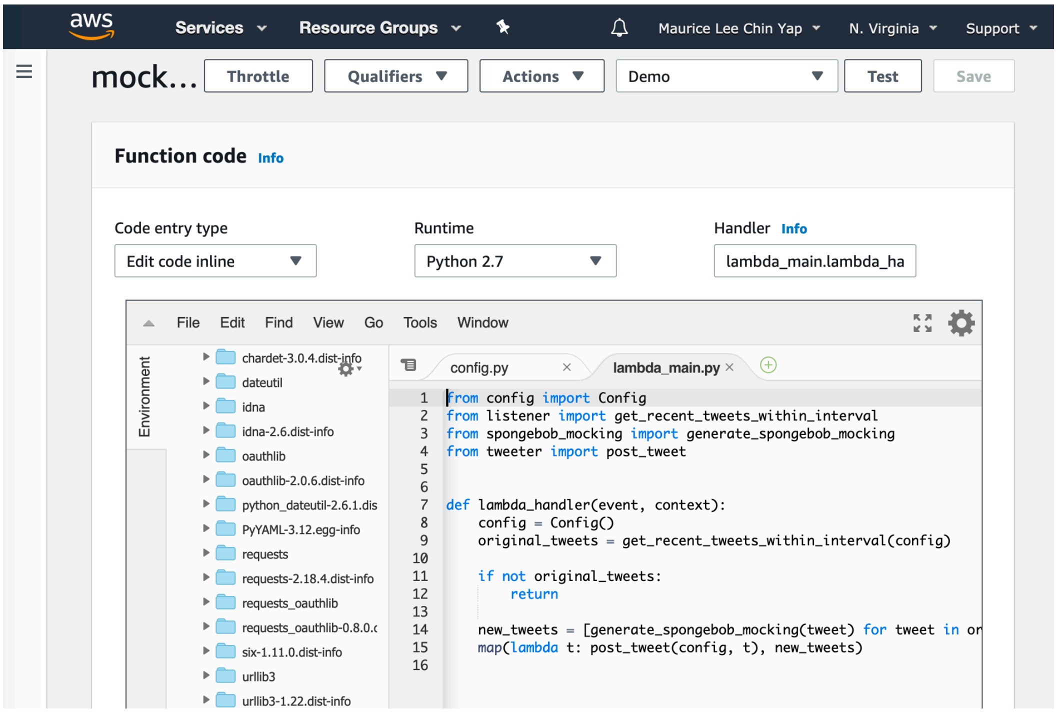


```
Time (UTC +00:00)   Message
2019-05-05
23:16:06   START RequestId: 93e2c40e-51e4-4688-945d-6f629aaa927f Version: $LATEST
23:16:06   {u'session': {u'new': True, u'sessionId': u'amzn1.echo-api.session.5e2c20c0-aba9-4fb4-a8b0-a2a7437fc2b9', u'user': {u'us
23:16:06   EVENT OBJECT:
23:16:06   {"session": {"new": true, "sessionId": "amzn1.echo-api.session.5e2c20c0-aba9-4fb4-a8b0-a2a7437fc2b9", "user": {"userId'
23:16:06   END RequestId: 93e2c40e-51e4-4688-945d-6f629aaa927f
23:16:06   REPORT RequestId: 93e2c40e-51e4-4688-945d-6f629aaa927f Duration: 0.49 ms Billed Duration: 100 ms Memory Size: 128
23:16:21   START RequestId: f30f5baa-50c8-4c72-a332-5ed5058ef1ee Version: $LATEST
23:16:21   {u'session': {u'new': False, u'sessionId': u'amzn1.echo-api.session.5e2c20c0-aba9-4fb4-a8b0-a2a7437fc2b9', u'user': {u'u
23:16:21   EVENT OBJECT:
23:16:21   {"session": {"new": false, "sessionId": "amzn1.echo-api.session.5e2c20c0-aba9-4fb4-a8b0-a2a7437fc2b9", "user": {"userIc
23:16:27   END RequestId: f30f5baa-50c8-4c72-a332-5ed5058ef1ee
23:16:27   REPORT RequestId: f30f5baa-50c8-4c72-a332-5ed5058ef1ee Duration: 6003.33 ms Billed Duration: 6000 ms Memory Size
23:16:27   2019-05-05T23:16:27.285Z f30f5baa-50c8-4c72-a332-5ed5058ef1ee Task timed out after 6.00 seconds
23:16:28   START RequestId: b30ca4eb-fba5-4f1c-bc89-c8e2a8883fa5 Version: $LATEST
23:16:28   {u'session': {u'new': False, u'sessionId': u'amzn1.echo-api.session.5e2c20c0-aba9-4fb4-a8b0-a2a7437fc2b9', u'user': {u'u
23:16:28   EVENT OBJECT:
23:16:28   {"session": {"new": false, "sessionId": "amzn1.echo-api.session.5e2c20c0-aba9-4fb4-a8b0-a2a7437fc2b9", "user": {"userIc
23:16:28   END RequestId: b30ca4eb-fba5-4f1c-bc89-c8e2a8883fa5
23:16:28   REPORT RequestId: b30ca4eb-fba5-4f1c-bc89-c8e2a8883fa5 Duration: 8.23 ms Billed Duration: 100 ms Memory Size: 128
23:17:02   START RequestId: cf5f41bb-b8da-4820-b475-7b55b255cee2 Version: $LATEST
23:17:02   {u'session': {u'new': True, u'sessionId': u'amzn1.echo-api.session.6690b315-eddd-4e33-8b7b-6f57e177e44b', u'user': {u'u
23:17:02   EVENT OBJECT:
23:17:02   {"session": {"new": true, "sessionId": "amzn1.echo-api.session.6690b315-eddd-4e33-8b7b-6f57e177e44b", "user": {"userIc
```

Figure 3.7.: Screenshot of the *Amazon Cloudwatch* logs for a Lambda function [7].

Data written to the standard output by Lambda function handlers is made available to users in logs in *Amazon CloudWatch*, the cloud resource monitoring service provided by

AWS. We show this in Figure 3.7, where we can also see filtering capabilities provided to the user. The user can search for log messages containing a certain string, or filter by the date and time at which a message was generated.



Function code

Code entry type: Edit code inline

Runtime: Python 2.7

Handler: lambda_main.lambda_ha

```
from config import Config
from listener import get_recent_tweets_within_interval
from spongebob_mocking import generate_spongebob_mocking
from tweeter import post_tweet


def lambda_handler(event, context):
    config = Config()
    original_tweets = get_recent_tweets_within_interval(config)

    if not original_tweets:
        return

    new_tweets = [generate_spongebob_mocking(tweet) for tweet in or
    map(lambda t: post_tweet(config, t), new_tweets)

```

Figure 3.8.: Screenshot of *AWS Lambda's* function code editor [6].

We show the Lambda function code editor in Figure 3.8. Here, users provide three parameters for their Lambda function handler. The first, *Code entry type*, is the source from which the code for the handler is obtained from, and has three options: the contents of the in-browser editor (as shown), an uploaded zip file or a file on *Amazon S3* — *AWS's* file storage service. The other two options are the runtime for the handler, and the path to the handler method (whose format depends on the runtime chosen).

The environment upon which Lambda functions are run does not include any third-party libraries not part of the core runtime, and so any such dependencies in users' handlers must be included its code. A common, and in fact, *AWS's* suggested way to achieve this is to simply download these libraries into the code base, and then package them into a zip file to upload [6]. In *Diorama*, we should provide users with an easy way to use external dependencies in their code.

Currently, *AWS Lambda* functions can be written in runtimes for six languages - Python, Ruby, Java, C#, Node.js and Go [10]. This allows a developers to write handlers in one of these six languages, which is useful, since they have the freedom to choose the language out of those, which best for them, be that the one most well-suited to the task, or one most widely used in their organisation. However, an even wider range or runtimes can be used — *AWS* allows developers to create and use their own custom runtimes for the execution environment (Amazon Linux), meaning that theoretically, developers can use any language which can run in this environment, given a custom runtime has been created for it [11]. Such freedom to choose a programming language would be useful to students using *Diorama* to writing node programs, because it would mean they are less likely to need to learn a new language to do so.

3.8 Summary of related work

Table 3.1 summarises the features that each existing product we looked have have. In the rightmost column, we set out which of these features we want *Diorama* to have.

[0] In theory, any language can be used with a custom runtime implementation. In practice, *AWS* only officially offers runtimes for six languages.

	A Simulator for Self-Stabilizing Distributed Algorithms	JBotSim	DSLabs	Docker	TETCOS NetSim	Codeanywhere and Repl.it	AWS Lambda	***Diorama***
Node isolation	Threads	Single-thread	Single-thread	Processes	Single-thread	n/a	n/a	?
Visual network topology	✓	✓	✓	n/a	✓	n/a	n/a	✓
Live network and node modification	✓	✓	X	X	X	n/a	n/a	✓
Limit on number of simulated nodes	11	None	None	None	None	None	None	**None**
User language(s)	Java	Java	Java	All	C	Several popular	All[1]	**All**
Output message filtering	X	X	X	X	X	n/a	✓	✓
Output message metadata	X	X	X	*Docker Compose* only	X	n/a	✓	✓
Provided messaging API	✓	✓	✓	X	✓	n/a	n/a	✓

Table 3.1.: Feature comparison of selected related works.

4 Software Design

This chapter presents the overall design of our software package, *Diorama*. We also discuss the rationale behind how design decisions were made in order to satisfy our high-level objectives as set out in Section 1.1, as well as features we outlined in Section 3.8.

4.1 Overview

We have created a distributed algorithm simulator in the form of a web service. It can be installed onto and served from a single virtual machine (VM). This would allow for *Diorama* to be developed for one particular operating system, while at the same time, being cross-platform for users (users can set up their own VM using hypervisor software like *VMware Workstation Player* or *VirtualBox*, or use a cloud VM service).

We have published *Diorama* in two forms:

- VM images compatible with popular cloud platforms and hypervisors;
- a deployment tool which the user can run to install *Diorama* onto an existing VM.

A key advantage of this approach is that the web service and all underlying system dependencies would run on a VM and therefore separately and isolated from the user's operating system. This means that other than those required to run the deployment tool, we do not need to consider system and software requirements on the user's machine.

4.2 Goals of the software package

We intend for *Diorama* to be used by people in the context of classroom teaching. It is therefore appropriate to design it with the objective of facilitating defined user journeys.

We use the envisioned high-level user journeys for two personas:

- **Leroy** — a computer science lecturer who delivers a course on distributed computing. He uses live coding in one of his lectures to demonstrate the behaviour of a distributed algorithm running on nodes in a network.

- **Stacey** — a student who is following Leroy's distributed computing course.

4.2.1 Use by an educator: Leroy's user journey

Leroy wants to demonstrate to his class four distributed algorithms for broadcasting messages:

- Best-effort Broadcast

- Eager Reliable Broadcast

- Uniform Reliable Broadcast

- Eager Probabilistic Broadcast

Installation

- Preparing for his lecture at home on his personal computer, Leroy creates a cloud virtual machine on the cloud services platform provided by his university through a subscription.

- Leroy installs *Diorama* onto his cloud VM, following simple instructions provided in our installation guide.

- Leroy opens the web user interface by connecting to public IP address of the VM through his web browser.

Creating programs

- On the web user interface, Leroy reads the documentation for creating node programs.

- Leroy uses his preferred text editor to code the programs for the four algorithms he wants to demonstrate and publishes them to a public git repository. He does this as he wants the code to be viewable by his students.

- Leroy creates four node programs on *Diorama* by providing the URL of each git repository in the web user interface.

Defining the network topology

- Leroy defines in code a network of seven fully-connected nodes, named `node_1` through to `node_7`, which each run the Best-effort Broadcast program. He uses the provided documentation for defining network topologies.

Running the simulation

Later on, Leroy delivers a lecture on reliable broadcast algorithms. He uses the desktop computer in the lecture theatre and will demonstrate each of the four reliable broadcast algorithms, beginning with Best-effort Broadcast.

- Leroy opens a web browser on the lecture theatre computer and navigates to the URL of his cloud virtual machine, which opens *Diorama's* web user interface.

- Leroy runs the seven nodes on the network he created at home and shows the real-time output for these nodes. This shows his class how the Best-effort Broadcast algorithm works and behaves.

- Leroy removes two of the connections from one of the nodes in the network and shows the output so his class can observe the effect this has had.

- Leroy replaces these two connections.

- Leroy schedules two of the nodes to stop in six seconds to simulate them crashing.

- Leroy continues showing the output from each of the seven nodes as two of them crash.

- Leroy changes the network so that the seven nodes all run the program for his next algorithm — Eager Reliable Broadcast.

- Leroy resets the simulation and in the same way, runs these seven nodes and schedules two to crash. The output logs now show how Eager Reliable Broadcast works and behaves.

- Leroy repeats this for the remaining two algorithms — Uniform Reliable Broadcast and Eager Probabilistic Broadcast.

4.2.2 Use by a learner: Stacey's user journey

Leroy has set Stacey an exercise involving implementing a broadcast algorithm and observing how it behaves with different program parameters, different network topologies and connection delays and failures.

Installation

- Stacey creates a new virtual machine using a hypervisor she already has installed on her laptop.

- Stacey installs *Diorama* onto her VM, following simple instructions provided in the installation guide.

- Stacey opens the web user interface using her web browser by connecting to the local IP address of her VM.

Creating the program

- Using the web interface, Stacey creates a node program and implements her algorithm using the editor in *Diorama's* web user interface. She is guided by the documentation provided for creating node programs.

Defining the network topology

- Stacey creates a network of five nodes arranged in a line topology by defining this in code. She uses the provided documentation for defining network topologies.

Running the simulation

- Stacey runs the five nodes in her network.
- Stacey schedules all her nodes to stop running after 10 seconds.

Exporting output

- Stacey downloads the output from the five nodes as a CSV file. She will use this later to analyse the success of the network in a spreadsheet software package, and present her analysis using graphs in her report.

Modifying program parameters

- Stacey changes some program parameters by editing her code for her node program.
- Stacey then runs her simulation and exports the output as before.

Adding connection delays and failures

- Stacey navigates adds a random delay to one of the connections in her network. She defines this delay to be normally-distributed, with a mean of 300ms and a standard deviation of 70ms.
- Stacey adds an 85% message-sending success rate on three other connections in the network.
- Stacey resets and runs her simulation and exports the output as she did before.

Modifying the network topology

- Stacey changes the topology to her own custom topology by defining it in code.
- Stacey resets and runs her simulation and exports the output as before.

4.3 Technical representation of concepts in distributed algorithms

We use *Docker* as the underlying service on which we will run simulations. As explored in Section 3.4, *Docker* has several features which make it well-suited for this in light of our aims, for example, language-agnosticism, quick setup of containers and the ability to run a large number of containers simultaneously.

A *Docker* image will be created for each of the user's programs. Each image will share the name of the program and will be built on top of image layers for the user's chosen runtime for each program. We will create Docker images to implement the underlying behaviour of nodes in each of our supported runtimes.

Each node in the user's network will be simulated with a *Docker* container running the *Docker* image for the node's program. After our software creates each Docker container, the user will be able to start, stop, pause and resume (unpause) a node's container depending on its state in its lifecycle. We illustrate the possible states and the actions available for each state in Figure 4.1.

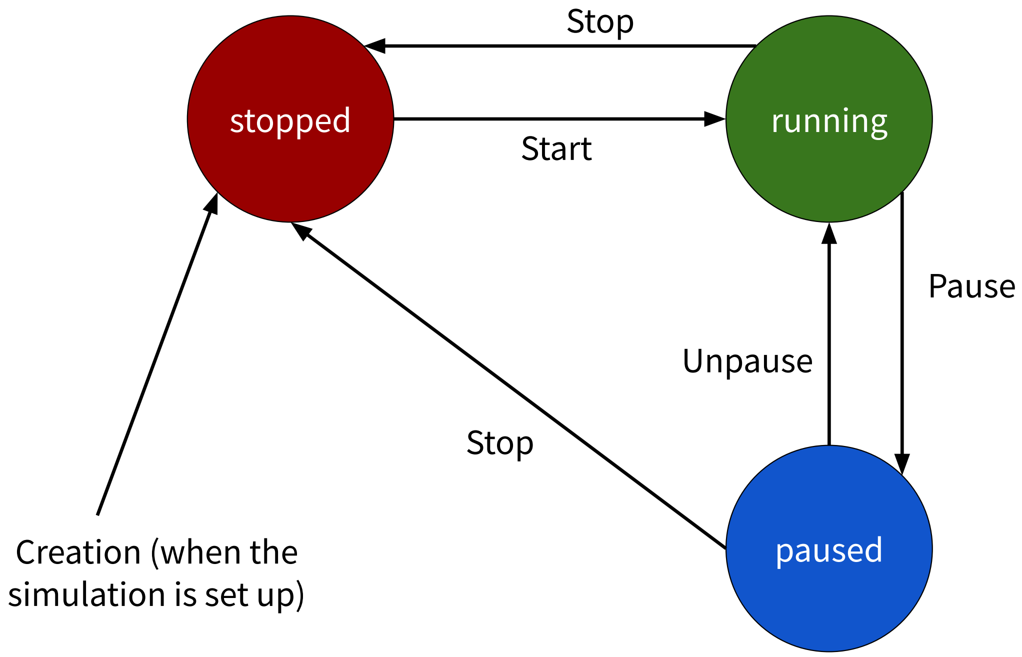


Figure 4.1.: State diagram for a simulated node.

Communications between nodes will be implemented by sending UDP packets between containers. Containers will all be connected to the same network, using the provided *Bridge* network driver, but two containers will only be able to send and receive packets to each other if their respective nodes are connected together in the user-defined network topology. We enforce this restriction in our underlying node implementation for each runtime.

4.4 Programming interface

We provide the user with documented interfaces to write their own node programs and create network topologies.

4.4.1 Node program API

The user can create a node program by implementing a method which the node effectively runs as its *main* method. It takes five arguments:

- `peerNids` (list of strings) - the node IDs (or *nids*) of all nodes that this node is connected to.
- `myNid` (string) - the *nid* of this node.
- `send` (method) - sends a message to a connected node. It takes two arguments - `message` (bytes), a bytes object to be sent, and `recipientNid` (string), the *nid* of the node to whom the message is being sent.
- `receive` (method) - receives the next message and returns a tuple of the message (bytes) and the sender's nid (string). This is a blocking method.
- `storage` (`Storage` object) - a persistent static key-value pair data store. It can be used, for example, to recover data after a node failure.

The `Storage` object is modelled as a simple key-value pair store. Keys must be strings, but values can take any type. It has seven public methods:

- `get(key)` (return value can be of any type; `key` is a string) - returns the value associated with the given `key`, or null if there isn't one.
- `getAll()` (returns an associative array, where all keys are strings and values can be of any type) - returns all key-value pairs stored.
- `put(key, value)` (returns null; `key` is a string; `value` can be of any type) - stores the given key-value pair. If a pair with the same `key` exists, this replaces it.

- `remove(key)` (returns null; `key` is a string) - removes the key-value pair with the given `key` if one exists.

- `containsKey(key)` (returns a boolean; `key` is a string) - returns whether or not a pair with the given `key` is present in the store.

- `clear()` (returns null) - removes all key-value pairs in the store.

- `size()` (returns an integer) - returns the number of key-value pairs in the store.

Anything which the user writes to standard output will be displayed in the node logs in our web user interface.

Example node program

Let us consider a node program which stores the last message received from each of its connected nodes. After every 20th message received, it prints this for each node, retrieving the message from the store. We present the pseudo-code for this in Figure 4.2.

```
On receive message from sender:
    put <sender, message> into storage
    send message to sender
    on every 20th message received:
        output "Here are the last messages I received from each of my
            connected nodes:"
        for node in connected nodes:
            output node: node-message
            where node-message is get node from storage
        reset storage
```

Figure 4.2.: The pseudo-code for our example node program.

To demonstrate the node program API we have presented, we use it to write the program in Python. We show this in Figure 4.3 and observe that our implementation does not add any unnecessary complexity.

Note that names of provided API functions and arguments are in snake case, while our presentation of the API uses lower camel case. This is because the standard naming convention for Python uses snake case [64].

```
number_of_messages_to_receive_before_broadcasting = 20

def main(peer_nids, my_nid, send, receive, storage):
    while True:
        for i in range(0,
            number_of_messages_to_receive_before_broadcasting):
            bytes_message, sender_nid = receive()
            message = bytes_message.decode("utf8")
            storage.put(sender_nid, message)
            send(bytes_message, sender_nid)  # Echo the message back to
                sender
        print("Here are the last messages I received from each of my
            connected nodes:")
        for peer_nid in peer_nids:
            if storage.contains_key(peer_nid):
                print(f"{peer_nid}: {storage.get(peer_nid)}")
        storage.clear()
```

Figure 4.3.: The Python code for our example node program, using our interface.

4.4.2 Network topology schema

The user defines their network topology by declaring its nodes in a serialisation language, such as JSON or YAML. All connections between nodes are two-way, meaning that if `nodeA` can successfully send messages to `nodeB`, then `nodeB` can successfully send messages to `nodeA`. We provide support for automatically generating nodes connected together in common topologies, which can be included as part of the user's overall topology.

At the top level, the topology definition is a dictionary with at least one of the keys, `single_nodes` and `node_groups`. The values for each of these is a list of dictionaries.

Single nodes

Each dictionary in the `single_nodes` list represents a single node and has two required and one optional key-value pairs:

- `nid` (string) - the *nid* of the node, determined by the user.
- `program` (string) - the name of the program that the node should run.
- `connections` (list of strings; optional) - *nids* of nodes that this node is connected to. Since connections are two-way, each connection only needs to be defined

once: if `alice` is in the `connections` list of `bob`, then `bob` is effectively in the `connections` list of `alice`.

Node groups

Each dictionary in the `node_groups` list represents a group of nodes in a common topology. We support five such network topologies — ring, line, fully-connected, star and tree (we illustrated each of these in Section 2.1).

The generated *nids* for topologies take the pattern:

```
[PREFIX][COUNTER][SUFFIX]
```

where `[COUNTER]` is an incrementing integer and `[PREFIX]` and `[SUFFIX]` are determined by the user. For example, a fully-connected network of three nodes would have the *nids*, `myFC-0a`, `myFC-1a` and `myFC-2a` if the `[PREFIX]` were `myFC-` and the `[SUFFIX]` were `a`.

A dictionary representing **ring, line or fully-connected topologies** has four required and four optional key-value pairs:

- `type` (string) - the type of standard network topology. It must be one of `ring`, `line` or `fully_connected`.

- `number_nodes` (integer) - the number of nodes in this topology.

- `program` (string) - the name of the program that the nodes in this topology should run.

- `nid_prefix` (string) - the prefix for the generated *nids*.

- `nid_suffix` (string; optional) - the suffix for the generated *nids*. The default suffix is the empty string.

- `nid_starting_number` (integer; optional) - the first number of the counter for the generated *nids*. The default starting number is 0.

- `nid_number_increment` (integer; optional) - the increment step size of the counter for the generated *nids*. The default step size is 1.

- `connections` (list of dictionaries; optional) - extra connections from a node in this group to another node in the overall topology. Each dictionary in this list has two keys: `from` and `to`, which take the *nids* of the two nodes to connect together.

A dictionary representing a **star** topology of a hub and one or more hosts has six required and four optional key-value pairs:

- `type` (string) - `star`.

- `hub_program` (string) - the name of the program that the hub node should run.

- `hub_nid` (string) - the *nid* of the hub node.

- `number_hosts` (integer) - the number of hosts in this star topology.

- `host_program` (string) - the name of the program that the host nodes should run.

- `host_nid_prefix` (string) - the prefix for the generated *nids* of the host nodes.

- `host_nid_suffix` (string; optional) - the suffix for the generated *nids* of the host nodes. The default suffix is the empty string.

- `host_nid_starting_number` (integer; optional) - the first number of the counter for the generated *nids* of the host nodes. The default starting number is 0.

- `host_nid_number_increment` (integer; optional) - the increment step size of the counter for the generated *nids* of the host nodes. The default step size is 1.

- `connections` (list of dictionaries; optional) - extra connections from a node in this star topology to another node in the overall topology. Each dictionary in this list has two keys: `from` and `to`, which take the *nids* of the two nodes to connect together.

We model a **tree** topology has several levels of nodes. At the first level is a single node - the root of the tree. Each level thereafter contains the children of the nodes in the level above. A dictionary representing a tree topology has five required and four optional key-value pairs:

- `type` (string) - `tree`.

- `number_levels` (integer) - the number of node levels.

- `number_children` (integer) - the number of children (in the next level) that each node has.

- `programs` (ordered list of strings) - the names of the programs that nodes on each level should run, starting from the first level (root).

- `nid_prefixes` (ordered list of strings) - the prefixes for the generated *nids* of the nodes on each level.

- `nid_suffixes` (ordered list of strings; optional) - the suffixes for the generated *nids* of the nodes on each level. The default suffix is the empty string.

- `nid_starting_numbers` (ordered list of integers; optional) - the first numbers of the counters for the generated *nids* of the nodes on each level. The default starting number is 0.

- `nid_number_increments` (ordered list of integers; optional) - the increment step sizes of the counters for the generated *nids* of the nodes on each level. The default step size is 1.

- `connections` (list of dictionaries; optional) - extra connections from a node in this tree topology to another node in the overall topology. Each dictionary in this list has two keys: `from` and `to`, which take the *nids* of the two nodes to connect together.

Example network topology definition

To demonstrate our interface, we use it to define the network visually illustrated in Figure 4.4. We show the *nid* of each node in bold type, with the italicised name of the program it runs underneath in brackets. This network consists of `alice`, a standalone node coloured orange; a star network with three hosts coloured red, a ring network of four nodes coloured blue and a tree network with three layers coloured green. Using our network topology interface, we show the code needed to generate this network, written in YAML, in Figure 4.5.

4.4.3 User events interface

During the simulation phase, we provide the user the ability to schedule events - actions that will be performed on nodes at some point in the future. Our interface requires a list of events, with each event consisting of three of three values:

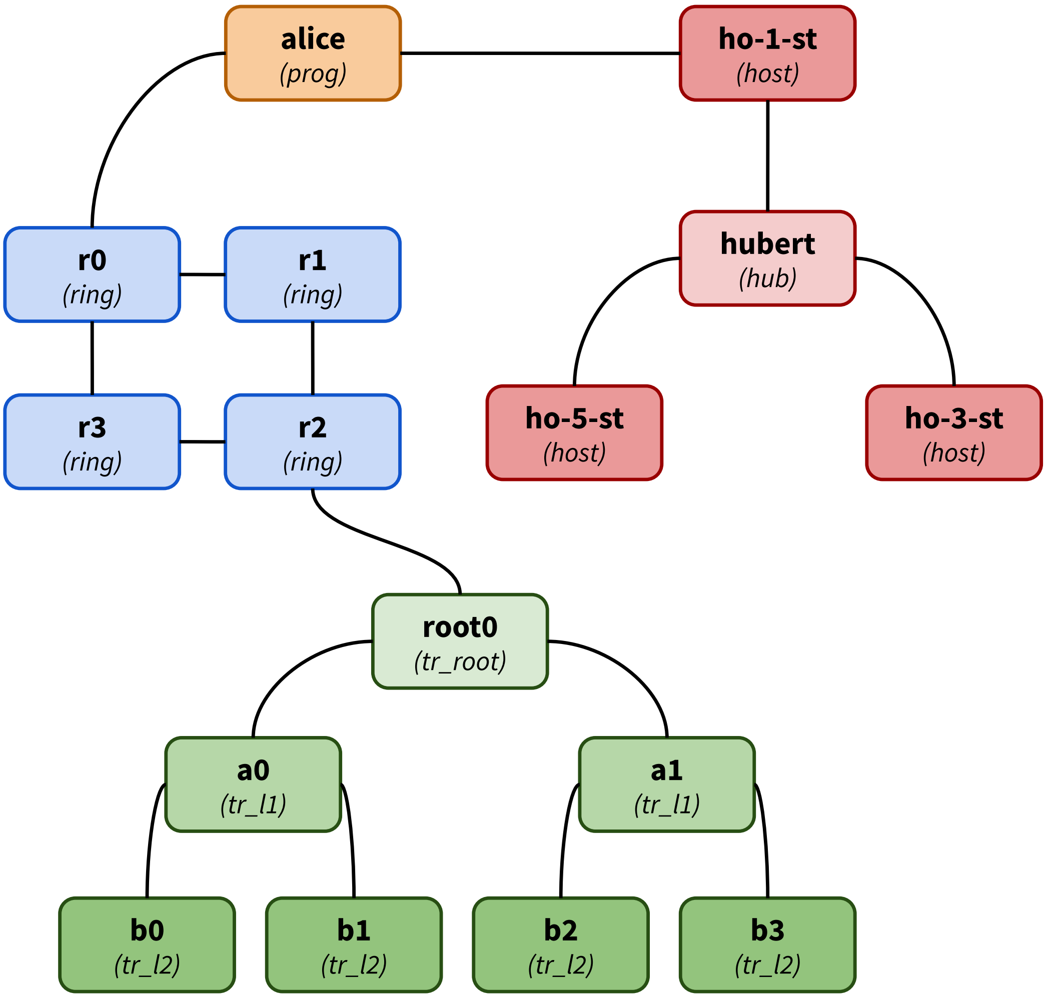


Figure 4.4.: An example network topology.

- The time when the action should be performed, defined in length of time after the user triggers the event schedule.

- The *nid* of the node on which the action is to be performed.

- The action to be performed - one of *start*, *stop*, *pause* and *unpause*.

Example user events

Given that a user has three nodes — `a`, `b` and `c`, which are in the stopped, running and paused states respectively, the following user events result in the timeline illustrated in Figure 4.6:

- Time: 1,000 ms; *nid*: `a`; action: *start*.
- Time: 1,000 ms; *nid*: `c`; action: *unpause*.

```
single_nodes:
- nid: alice
  program: prog
node_groups:
- type: ring
  number_nodes: 4
  program: ring
  nid_prefix: r
  connections:
  - from: r0
    to: alice
- type: star
  hub_program: hub
  hub_nid: hubert
  number_hosts: 3
  host_program: host
  host_nid_prefix: ho-
  host_nid_suffix: -st
  host_nid_starting_number: 1
  host_nid_number_increment: 2
  connections:
  - from: ho-1-st
    to: alice
- type: tree
  number_levels: 3
  number_children: 2
  programs:
  - tr_root
  - tr_l1
  - tr_l2
  nid_prefixes:
  - root
  - a
  - b
  connections:
  - from: root0
    to: r2
```

Figure 4.5.: The YAML example network topology. We also provide a JSON version of this code in Appendix A.

- Time: 2,500 ms; *nid*: `a`; action: *stop*.
- Time: 2,500 ms; *nid*: `b`; action: *stop*.
- Time: 2,500 ms; *nid*: `c`; action: *stop*.

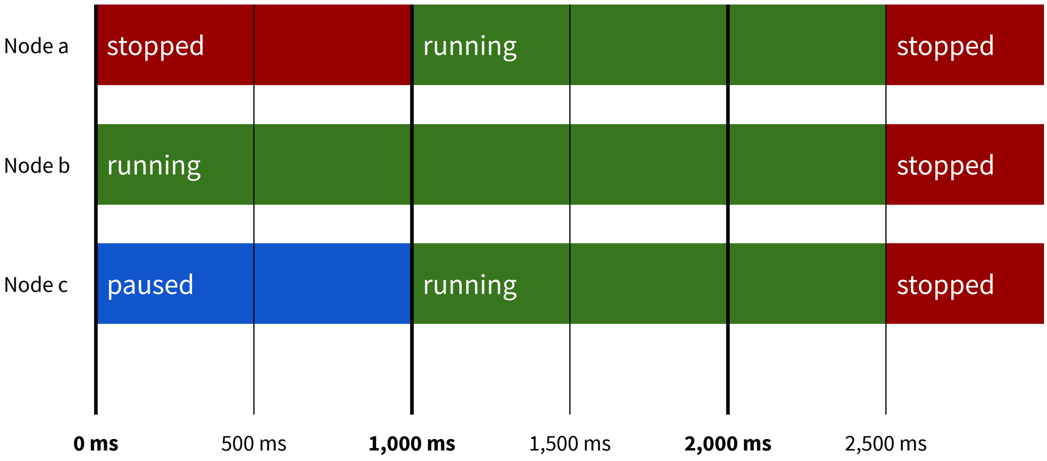
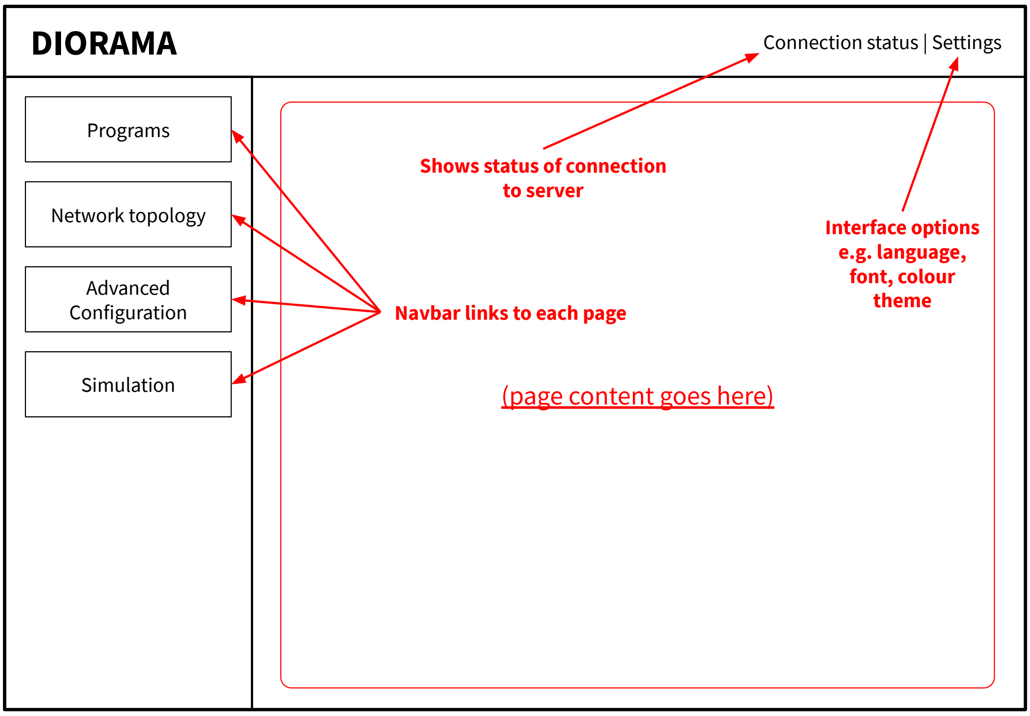


Figure 4.6.: The timeline of node states generated by our example user events.

4.5 Web user interface



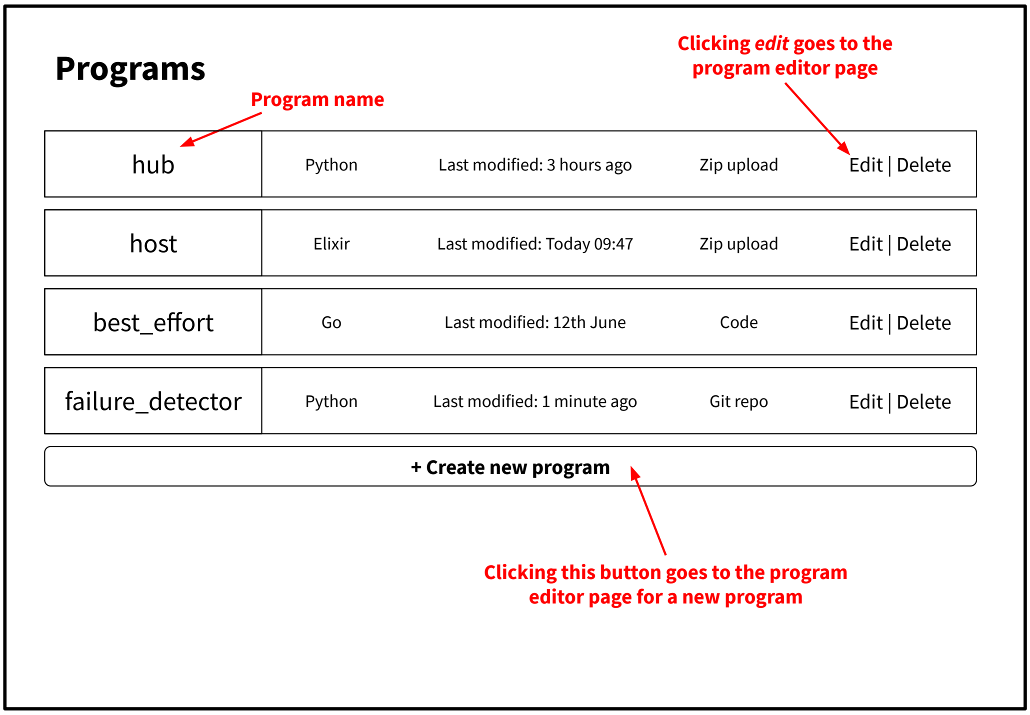
DIORAMA
Connection status | Settings
Programs
Network topology
Advanced Configuration
Simulation
Shows status of connection to server
Interface options e.g. language, font, colour theme
Navbar links to each page
(page content goes here)

Figure 4.7.: Wireframe of the main navigational and other fixed elements of the web application.

Based on our user journeys outlined in Section 4.2, we present the design of the web user interface for *Diorama* through a series of wireframes.

4.5.1 Main elements

We show in Figure 4.7 the elements of our interface which remain fixed across all its pages — the navigation bar and the top bar. Our interface will contain four main pages which can be accessed directly through the navigation bar on the left side of the interface. In our fixed top bar, we include a visual indicator of the status of the connection to the server, as well as a button which brings up a modal in which the user can change user interface options, such as visual preferences and language. The contents of other wireframes will be included in the interface in the red box.



Programs

hub	Python	Last modified: 3 hours ago	Zip upload	Edit | Delete
host	Elixir	Last modified: Today 09:47	Zip upload	Edit | Delete
best_effort	Go	Last modified: 12th June	Code	Edit | Delete
failure_detector	Python	Last modified: 1 minute ago	Git repo	Edit | Delete

Program name

Clicking *edit* goes to the program editor page

Clicking this button goes to the program editor page for a new program

Figure 4.8.: Wireframe of the programs viewer.

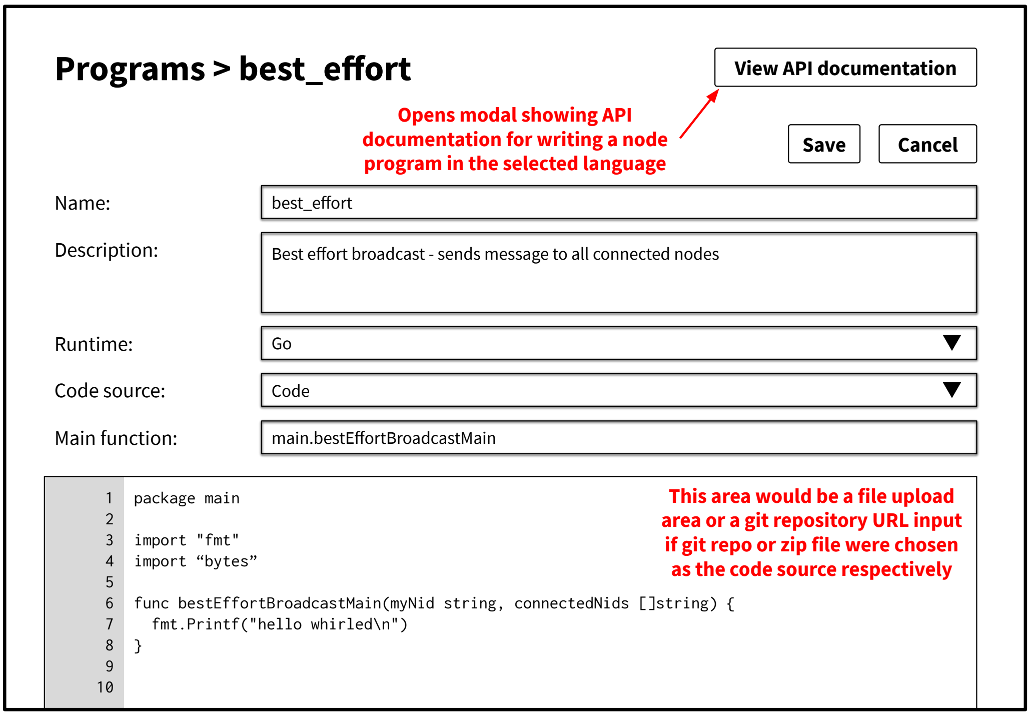
4.5.2 Programs explorer

Figure 4.8 shows the first main page, the programs explorer. It lists the programs which the user has written, along with details (runtime, last edit time, code source), a button to delete each program and a link to each program's editor page, illustrated in Figure 4.9. The user can also create a new program using the button as illustrated.

4.5.3 Program editor

On the program editor page in Figure 4.9, the user can view documentation for the node program interface which they will implement. They can provide the code for their program in three ways (the *code source*):

- Coding directly in the web user interface using the graphical text editor. This is suited to smaller programs and is particularly advantageous in that it is a very quick way to write programs and provides a quick turnaround for making changes to



Opens modal showing API documentation for writing a node program in the selected language

Name: best_effort

Description: Best effort broadcast - sends message to all connected nodes

Runtime: Go

Code source: Code

Main function: main.bestEffortBroadcastMain

```
package main

import "fmt"
import “bytes”

func bestEffortBroadcastMain(myNid string, connectedNids []string) {
  fmt.Printf("hello whirled\n")
}
```

This area would be a file upload area or a git repository URL input if git repo or zip file were chosen as the code source respectively

Figure 4.9.: Wireframe of the program editor.

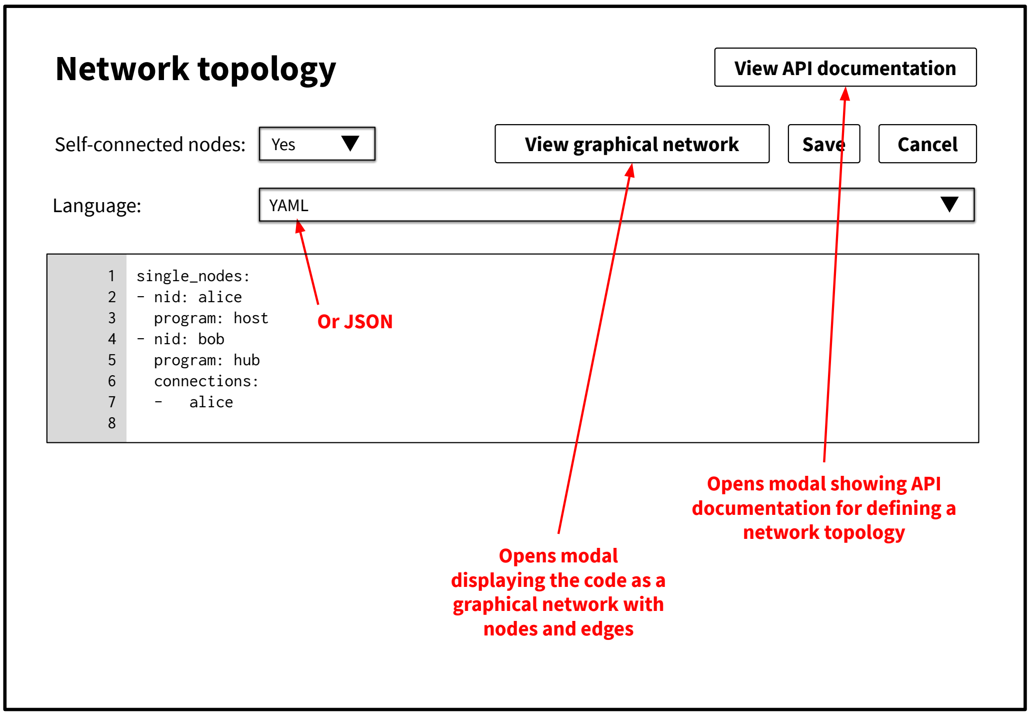
them. For runtimes where it is straightforward and appropriate to do so, we will also allow the user to list required external dependencies from the runtime’s official online software repository, such as *PyPI* for Python, *npm* for Node.js and *RubyGems* for Ruby.

- Providing the URL to a public git repository and optionally, the desired checkout tag for a specific commit. This allows users to share their code (as in the case of our persona, Leroy, in Section 4.2.1), for example, on *GitHub*, and is good for larger projects where multiple files and modules are needed.

- Uploading a zip file. If a user’s program requires custom third-party libraries, these can be packaged together with the program code. It also allows for multiple files and modules to be used - useful in larger projects.

The *main* method which implements our node program API (Section 4.4.1), needs to be provided. For programs where the code source is raw code (coded directly in the interface editor), this is simply the name, for example, `nodeMain`. Where the code source is a git repository or a zip file, the path to the handler must be provided, for example, `myModule.fileName.nodeMain`. This allows the user to program using helper methods and to use a more complex file structure for larger projects.

4.5.4 Network topology editor



Self-connected nodes: Yes

Language: YAML

```
single_nodes:
- nid: alice
  program: host
- nid: bob
  program: hub
  connections:
  -   alice
```

Or JSON

Opens modal showing API documentation for defining a network topology

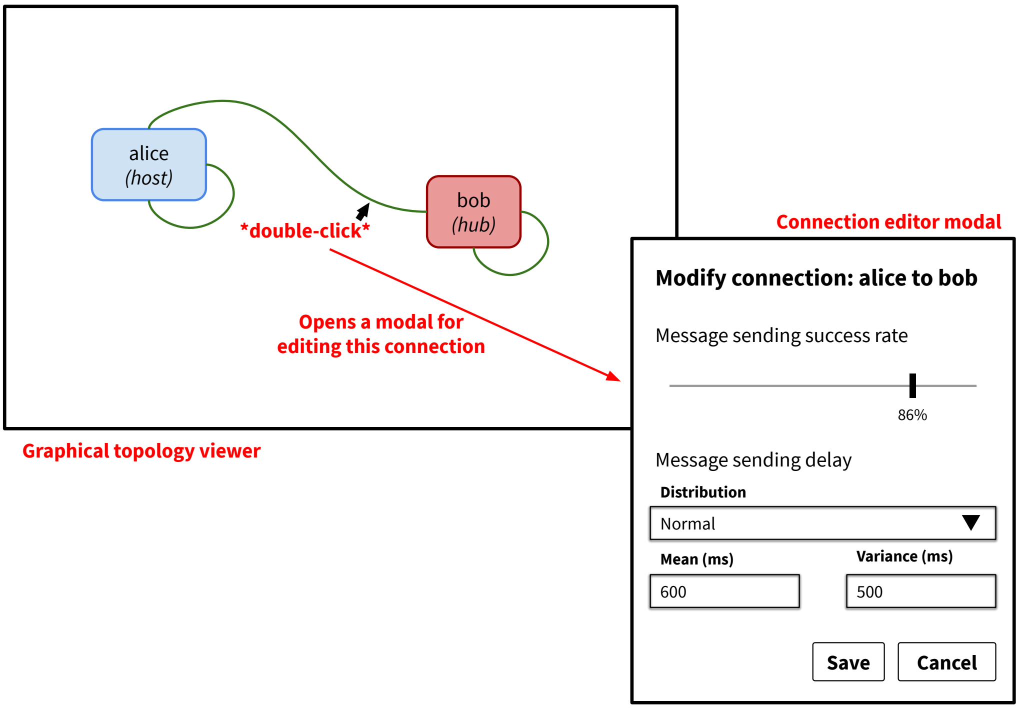
Opens modal displaying the code as a graphical network with nodes and edges

Figure 4.10.: Wireframe of the network topology editor.

The design for the network topology editor page is outlined in Figure 4.10. Here, we provide a text editor for the user to write their topology, following our interface presented in Section 4.4.2. We will initially support the serialisation languages, JSON and YAML, since these are two of the most popular and human-readable serialisation languages. Like the program editor page, the user is able to bring up documentation for our network topology interface through the button in the top right corner.

We give the user the option to make nodes self-connected. This is useful, since in many distributed algorithms, a node's ability to send messages to itself is assumed, and so the option saves the user from needing to add these connections explicitly in their topology definition code.

As well as this, when the user clicks the button labelled *View graphical network*, a visual graph, showing their topology will be displayed in a modal, as we show in Figure 4.11. This will allow them to check that the graph they have coded is as they intended. We also allow the user to add a random or fixed delay and set success rates for message-passing to a connection by double-clicking on the graph edge representing it.



Modify connection: alice to bob

Message sending success rate

86%

Message sending delay

Distribution

Normal ▼

Mean (ms): 600

Variance (ms): 500

Figure 4.11.: Wireframes of the graphical topology viewer and the connection parameters editor.

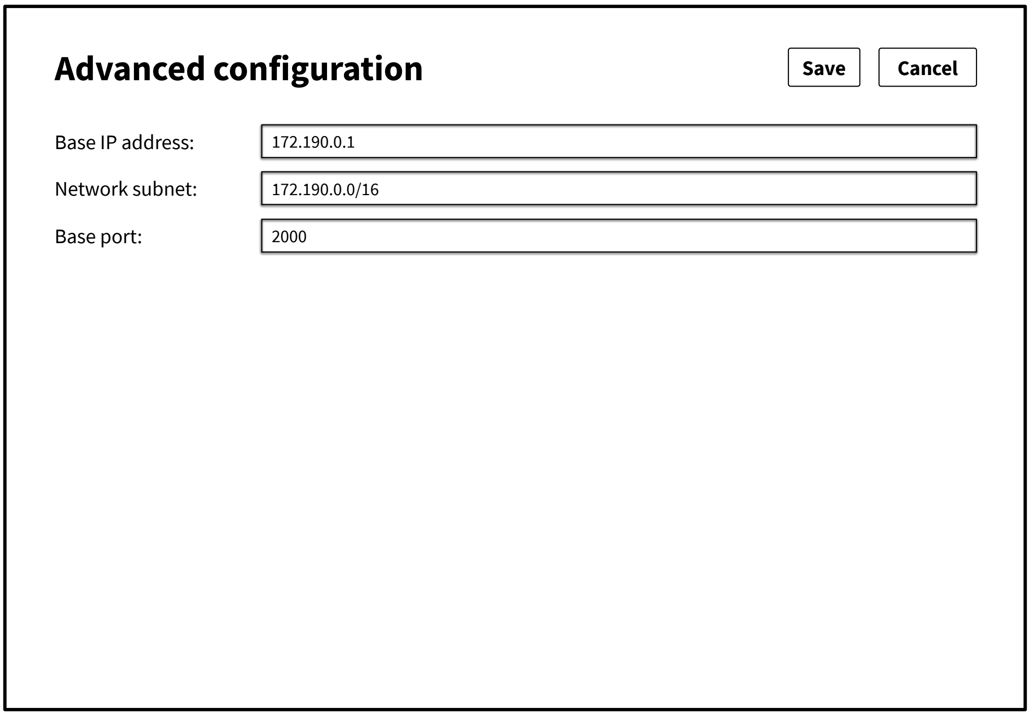
4.5.5 Advanced configuration

The *Advanced configuration* page, shown in Figure 4.12, allows the user to change more advanced settings in their simulation. Our intention is that the settings here need not be changed in the vast majority of uses, however, there may be some occasions where this is needed, for example, when the user is using other local IP addresses in other applications.

4.5.6 Simulation

The user will run and manage their simulation using the simulation page. This has two tabs: the node manager view, shown in Figure 4.13, and the node output logs view, shown in Figure 4.14. At the top of the page, there are three buttons: one which brings up a graphical network modal (Figure 4.11) to allow the user to edit node connection parameters[1]; one for the user to bring up the modal to schedule events; and a reset button, which contains an indicator that is visible when the user has made changes to their programs or their network topology since their simulation was set up (in other words, their simulation is out-of-date).

[1] The user journey for Leroy sees him removing node connections (Section 4.2.1). Our solution for this would be to set the message sending success rates for such connections to 0 %. Doing this achieves the same effect.



Advanced configuration

Base IP address: 172.190.0.1

Network subnet: 172.190.0.0/16

Base port: 2000

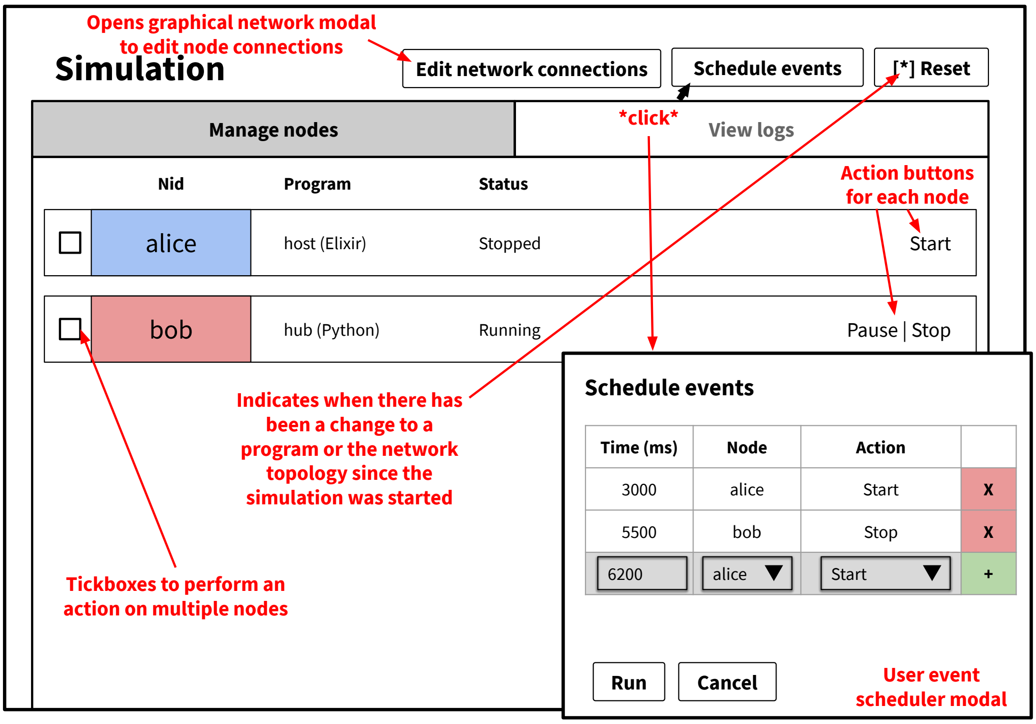
Figure 4.12.: Wireframe of the advanced configuration page.

Node manager view

The node manager view (Figure 4.13) lists the nodes that have been generated for the simulation. The status of the *Docker* container for each node is shown, and the user can use buttons to perform the possible actions on the container, given its state. We also provide checkboxes so that multiple nodes can be selected, and then an action can be performed on all selected nodes. The user event scheduler modal allows users to add events to a list, and then run all of these. The columns of the table shown correspond to the three values needed to define an event using our interface (Section 4.4.3).

Node output logs view

The node output logs view (Figure 4.14) displays, in real-time, data written to standard output (*stdout*) for each node in chronological order. We use coloured rows to differentiate between the output from different nodes, and these colours are accordingly used in the node manager view for each node. We also provide the user with a filter which they can use to select which lines of output to show and hide. They can filter by the contents of the message, optionally using regular expressions; the program which the node is running; and choose to just show output from their selection of one or more nodes. Users may need to use the logs from their simulation elsewhere; for example, our persona, Stacey,


Opens graphical network modal to edit node connections
Simulation
Edit network connections
Schedule events
[*] Reset
Manage nodes
click
View logs
Action buttons for each node
Indicates when there has been a change to a program or the network topology since the simulation was started
Tickboxes to perform an action on multiple nodes
User event scheduler modal
Run
Cancel

	Nid	Program	Status	
□	alice	host (Elixir)	Stopped	Start
□	bob	hub (Python)	Running	Pause | Stop

Schedule events

Time (ms)	Node	Action	
3000	alice	Start	X
5500	bob	Stop	X
6200	alice ▼	Start ▼	+

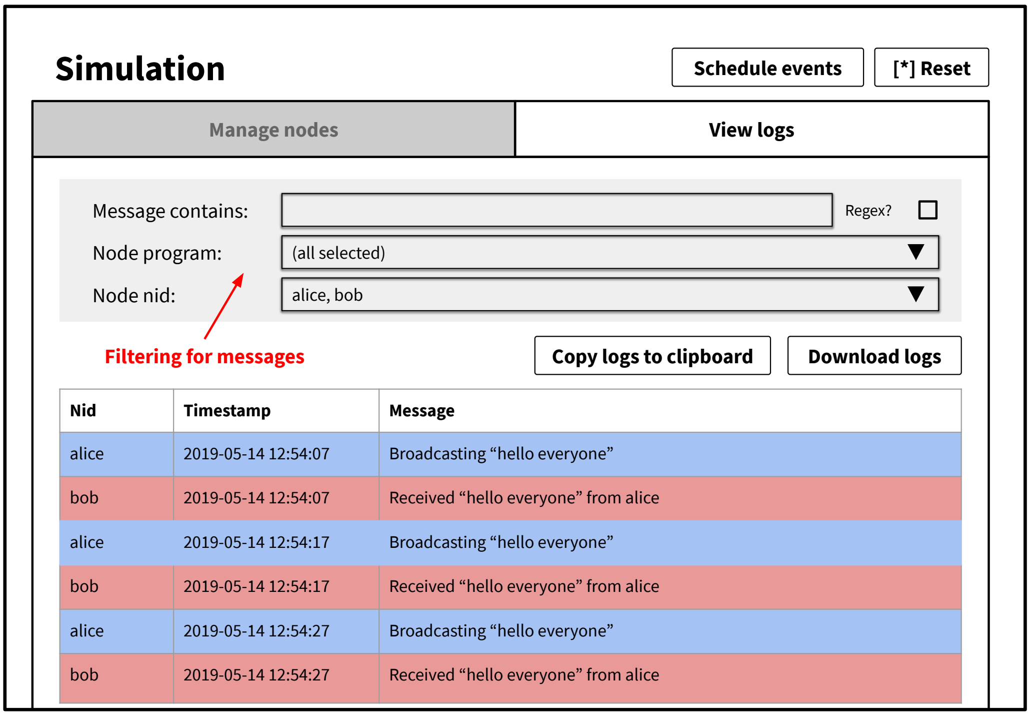
Figure 4.13.: Wireframe of the node manager view within the simulation page and the user event scheduler modal.

uses spreadsheet software to statistically analyse her data and draw graphs. We therefore provide the ability to download these logs as a CSV or JSON file and to copy the data in the logs table to the clipboard, ready to be pasted straight into a spreadsheet software package.

4.6 Concluding remarks

We have presented a design for a distributed algorithm simulator web service which is cross-platform and straightforward for users to set up. We have outlined, at a high level, how a user's programs, nodes and network topology would be represented using *Docker*, which allows users to write programs in a wide variety of languages and to run simulations with large numbers of nodes running simultaneously.

We have designed an intuitive programming interface for users to write node programs, which can easily send and receive messages to and from other connected nodes, and incorporates a persistent static key-value data store. We provide a powerful schema for users to declare their network topology — the nodes in it and how they are connected — using the common serialisation languages, YAML and JSON. It also allows users to include in their topology groups of automatically-generated nodes connected together



Message contains: Regex? □

Node program: (all selected) ▼

Node nid: alice, bob ▼

Filtering for messages

Nid	Timestamp	Message
alice	2019-05-14 12:54:07	Broadcasting “hello everyone”
bob	2019-05-14 12:54:07	Received “hello everyone” from alice
alice	2019-05-14 12:54:17	Broadcasting “hello everyone”
bob	2019-05-14 12:54:17	Received “hello everyone” from alice
alice	2019-05-14 12:54:27	Broadcasting “hello everyone”
bob	2019-05-14 12:54:27	Received “hello everyone” from alice

Figure 4.14.: Wireframe of the node output logs view within the simulation page.

in common topology shapes. Our system for creating future node-action events allows users to create a simple timeline of future desired events.

Guided by two clearly-defined user journeys, we have proposed an ergonomic web user interface on which a user can run a simulation of their defined network topology with nodes running programs they have written using our well-documented API. During simulations, it displays logs from all nodes in real-time, which can be filtered and exported.

Proof-of-concept: *Diorama* 5

> ❞ DAI ·UH ·**RAA** ·MUH. *noun - a model that shows a situation . . . in a way that looks real.*
>
> — **Cambridge English Dictionary**
> (Definition of Diorama [14])

In this chapter, we present our proof-of-concept implementation of *Diorama*, discuss key parts of the technical implementation and explore some of the challenges we faced.

5.1 Aims for the proof-of-concept

We set out to create a practical working implementation of *Diorama*, following the design we presented in Chapter 4 as much as possible in order to entirely support the two user journeys presented in Section 4.2. We wanted to finish with a product which was ready to be deployed by users and used for its purpose of simulating distributed algorithms. Given the limited time frame available to achieve this, we would prioritise implementing key features and user functionality, over striving for production-quality standards in engineering or user experience design.

Furthermore to this, we wanted to engineer a proof-of-concept which uses, to as great of an extent as possible, best-practices, and modern widely-used technologies. This is towards making our product something which could, in future, be easily built upon by myself or others in the open-source community, hopefully ending up with something which can be used in the *real world* — by *real* teachers and lecturers, by *real* students, in *real* classrooms and lecture theatres.

5.2 Overview of product

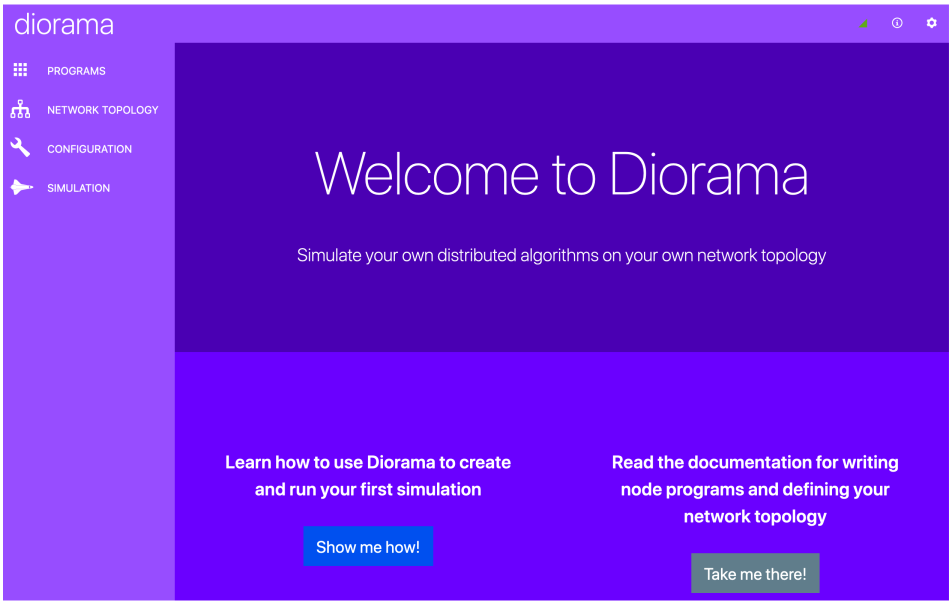
Our proof-of-concept *Diorama* application successfully meets the vast majority of our primary aim — to implement our design and support the user journeys of Leroy and Stacey, our two personas. It does, however, miss one piece of desired functionality: editing node connection parameters live, that is, while a simulation is running. We explore reasons for this, as well as possible workarounds in Section 6.1.

We have produced a web application which can be deployed by users in a straightforward manner. The application is self-contained on a single Ubuntu Linux virtual machine, and we publish this in two forms:

- full virtual machine disk images for each of the cloud platforms, *Microsoft Azure*, *Google Cloud Platform* (Compute Engine) and *AWS* (EC2), as well as for the *VirtualBox* hypervisor software.
- a deployment tool which installs Diorama on an existing virtual machine using SSH.

5.2.1 Gallery

In Figures 5.1, 5.2, 5.3, 5.4, 5.5, 5.6, 5.7, 5.8, 5.9, 5.10, 5.11, 5.12 and 5.13 we show the different parts of our web user interface.



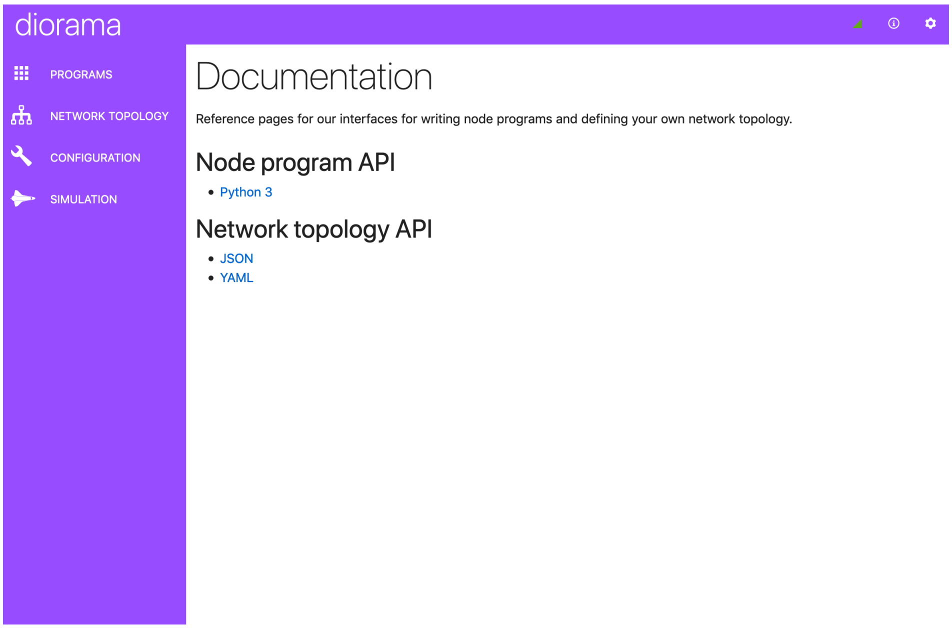
Welcome to Diorama

Simulate your own distributed algorithms on your own network topology

Learn how to use Diorama to create and run your first simulation

Read the documentation for writing node programs and defining your network topology

Figure 5.1.: The home page.



Documentation

Reference pages for our interfaces for writing node programs and defining your own network topology.

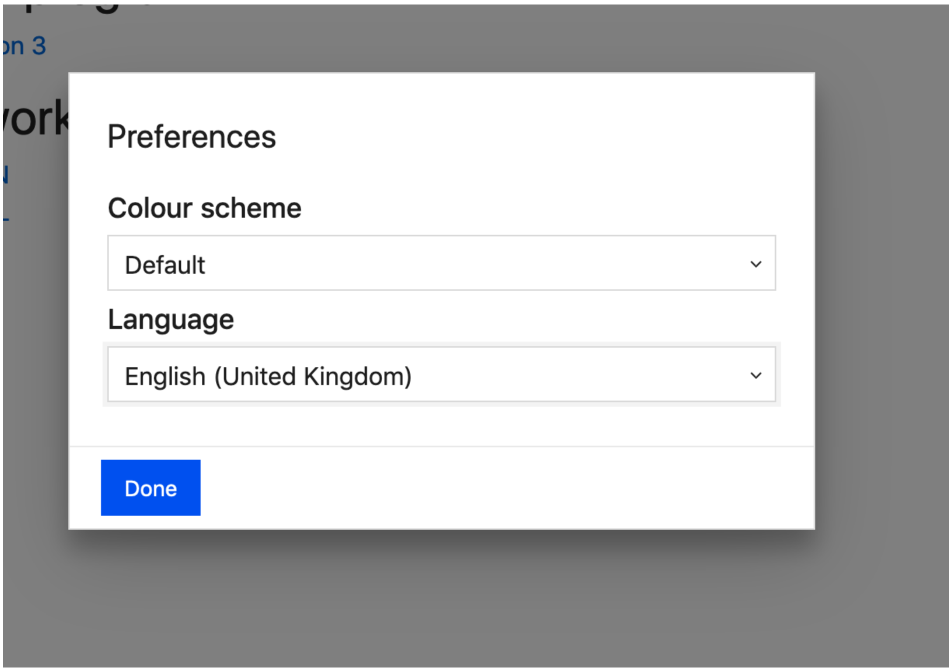
Node program API

- Python 3

Network topology API

- JSON
- YAML

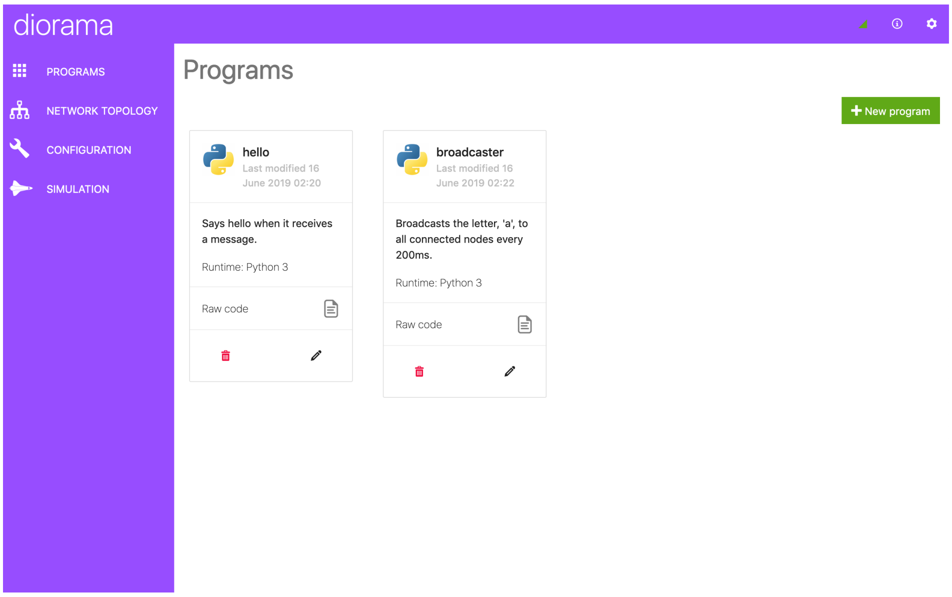
Figure 5.2.: The documentation page.


Preferences

Colour scheme: Default

Language: English (United Kingdom)

Figure 5.3.: The user interface preferences modal, where users can select their language and colour scheme.



Programs

hello
Last modified 16 June 2019 02:20

Says hello when it receives a message.

Runtime: Python 3

Raw code

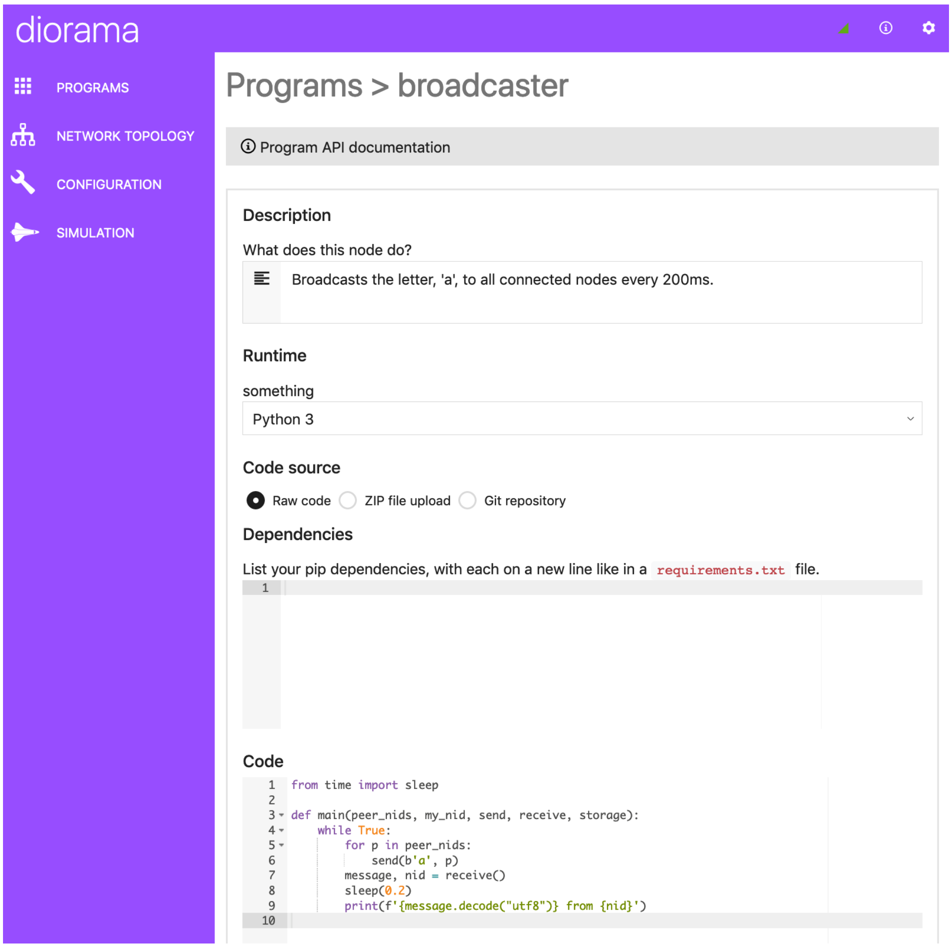
broadcaster
Last modified 16 June 2019 02:22

Broadcasts the letter, 'a', to all connected nodes every 200ms.

Runtime: Python 3

Raw code

Figure 5.4.: The node programs viewer.



Programs > broadcaster

ⓘ Program API documentation

Description

What does this node do?

Broadcasts the letter, 'a', to all connected nodes every 200ms.

Runtime

something

Python 3

Code source

● Raw code ○ ZIP file upload ○ Git repository

Dependencies

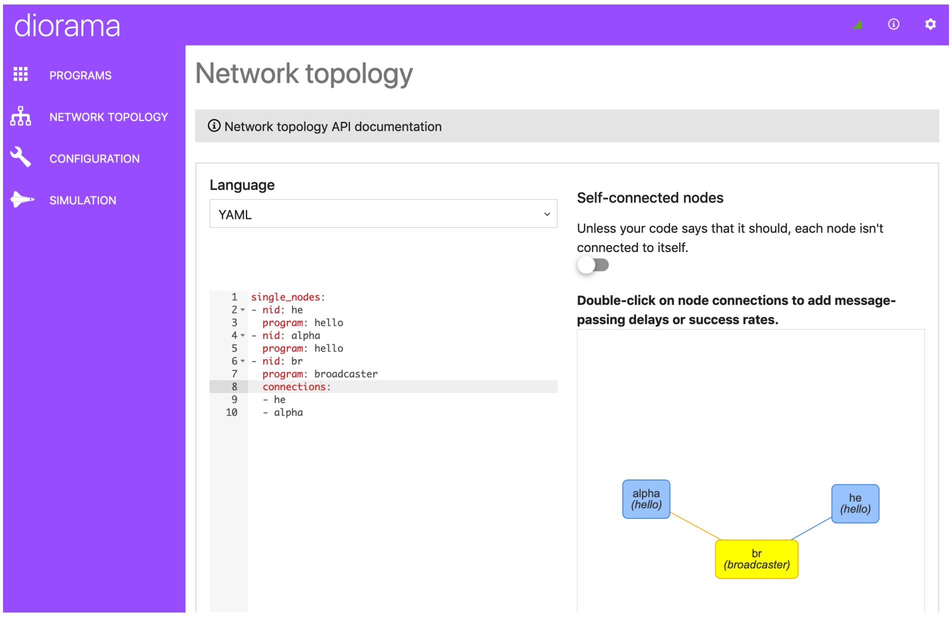
List your pip dependencies, with each on a new line like in a `requirements.txt` file.

Code

```
from time import sleep

def main(peer_nids, my_nid, send, receive, storage):
    while True:
        for p in peer_nids:
            send(b'a', p)
        message, nid = receive()
        sleep(0.2)
        print(f'{message.decode("utf8")} from {nid}')
```

Figure 5.5.: The node program editor, showing raw code input.



Network topology

ⓘ Network topology API documentation

Language

YAML

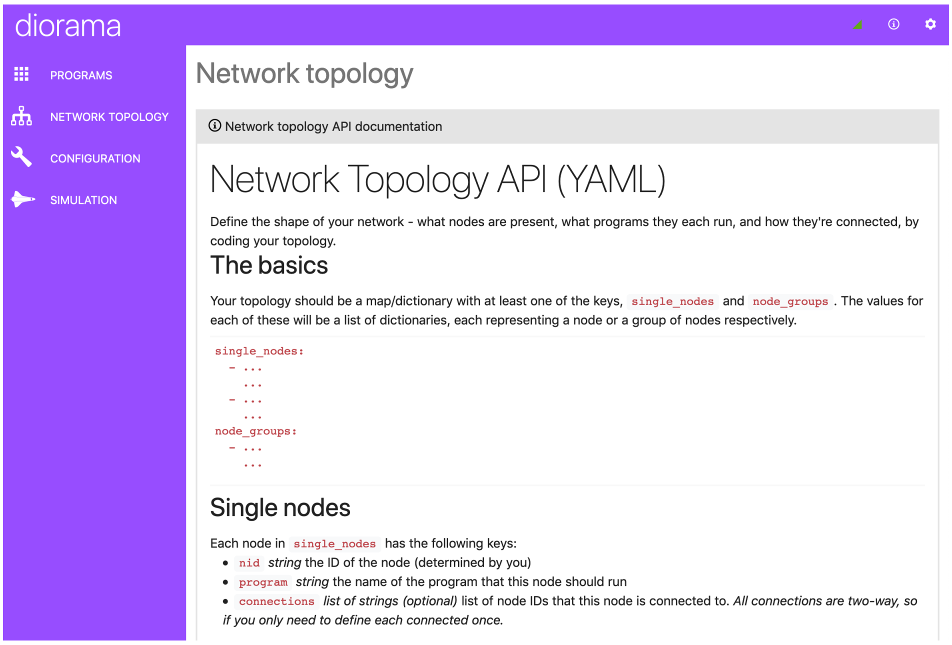
```yaml
single_nodes:
- nid: he
  program: hello
- nid: alpha
  program: hello
- nid: br
  program: broadcaster
  connections:
  - he
  - alpha
```

Self-connected nodes

Unless your code says that it should, each node isn't connected to itself.

Double-click on node connections to add message-passing delays or success rates.

Figure 5.6.: The network topology editor.



Network topology

ⓘ Network topology API documentation

Network Topology API (YAML)

Define the shape of your network - what nodes are present, what programs they each run, and how they're connected, by coding your topology.

The basics

Your topology should be a map/dictionary with at least one of the keys, `single_nodes` and `node_groups` . The values for each of these will be a list of dictionaries, each representing a node or a group of nodes respectively.

```yaml
single_nodes:
  - ...
    ...
  - ...
    ...
node_groups:
  - ...
    ...
```

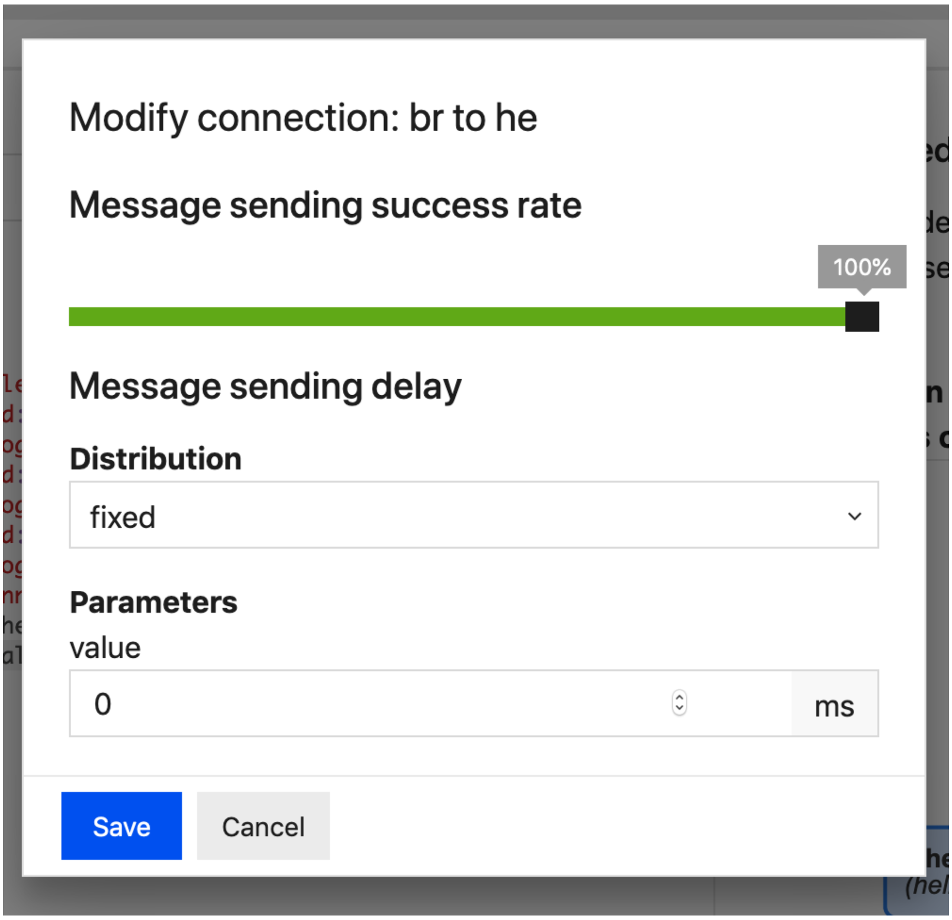
Single nodes

Each node in `single_nodes` has the following keys:

- `nid` *string* the ID of the node (determined by you)
- `program` *string* the name of the program that this node should run
- `connections` *list of strings (optional)* list of node IDs that this node is connected to. *All connections are two-way, so if you only need to define each connected once.*

Figure 5.7.: The network topology editor with the API Documentation accordion expanded

.



Modify connection: br to he

Message sending success rate

100%

Message sending delay

Distribution

fixed

Parameters

value

0 ms

Figure 5.8.: The network connection editor modal.

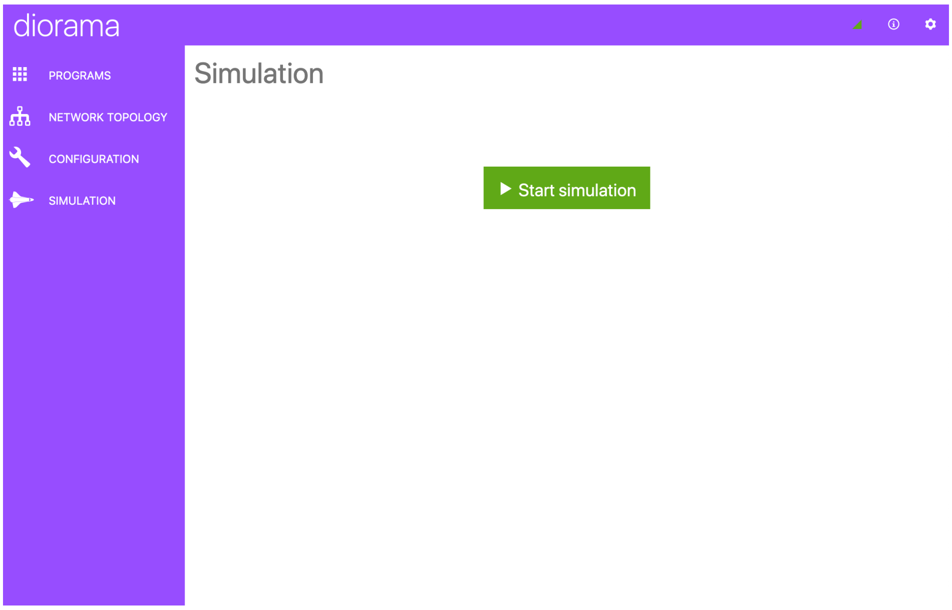
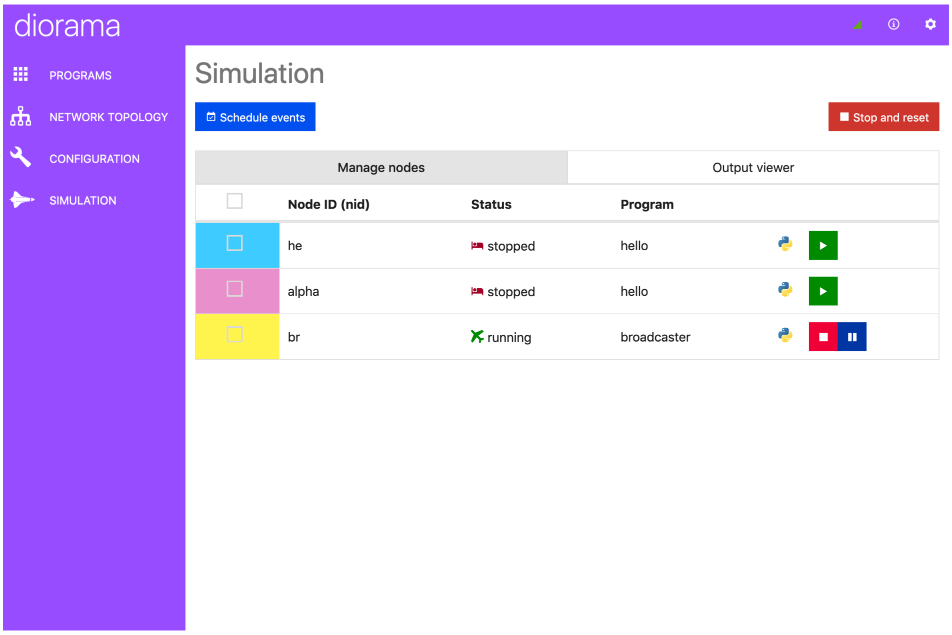
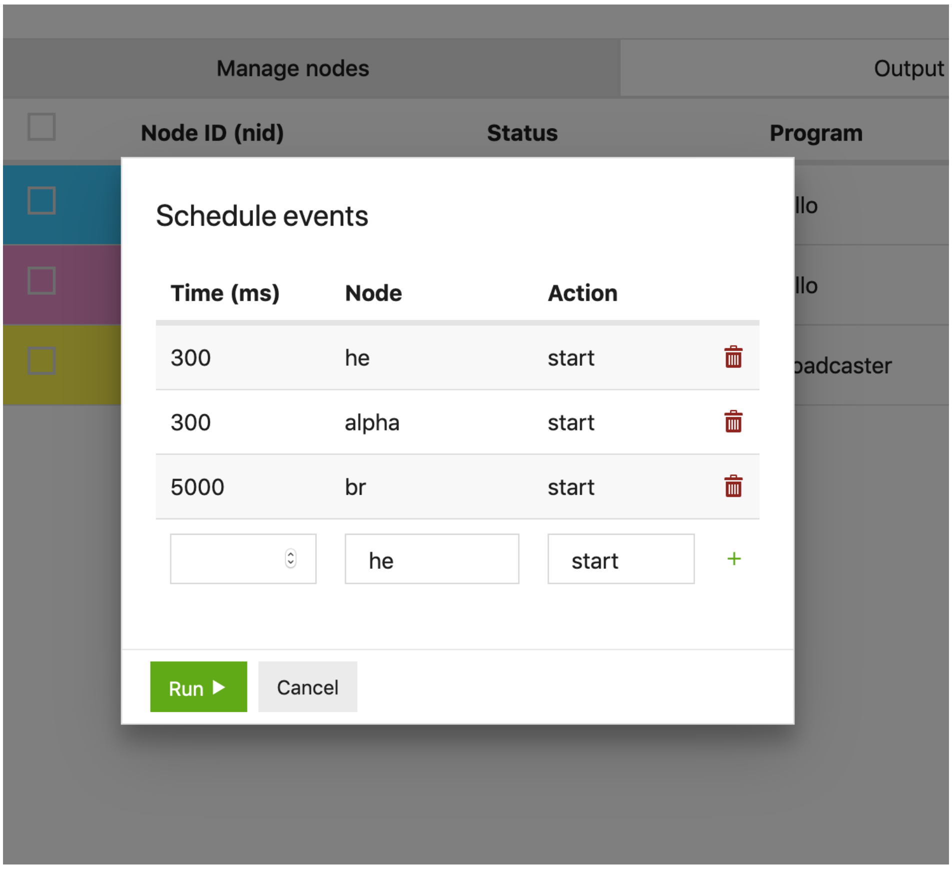


Figure 5.9.: The simulation page, before a simulation has been started.



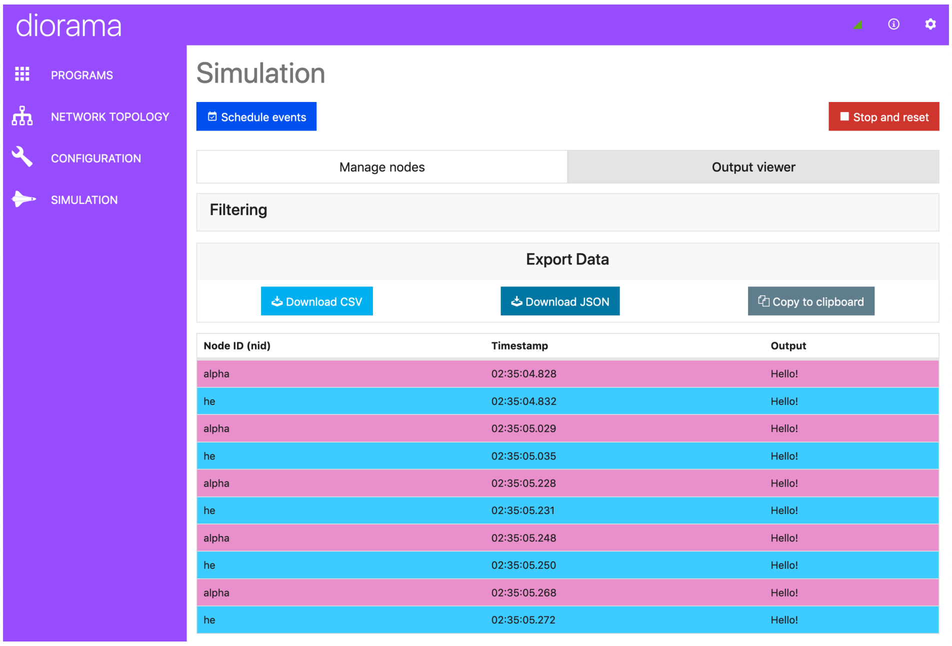
□	Node ID (nid)	Status	Program
□	he	stopped	hello
□	alpha	stopped	hello
□	br	running	broadcaster

Figure 5.10.: The node manager view on the simulation page.



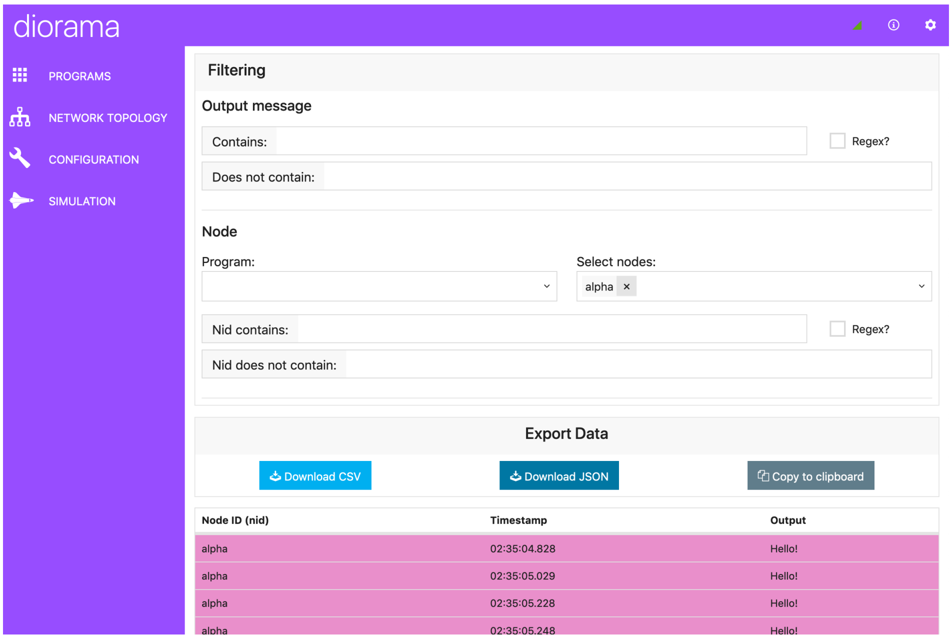
Time (ms)	Node	Action
300	he	start
300	alpha	start
5000	br	start
	he	start

Figure 5.11.: The user events scheduler modal.



Node ID (nid)	Timestamp	Output
alpha	02:35:04.828	Hello!
he	02:35:04.832	Hello!
alpha	02:35:05.029	Hello!
he	02:35:05.035	Hello!
alpha	02:35:05.228	Hello!
he	02:35:05.231	Hello!
alpha	02:35:05.248	Hello!
he	02:35:05.250	Hello!
alpha	02:35:05.268	Hello!
he	02:35:05.272	Hello!

Figure 5.12.: The output logs view on the simulation page.



Node ID (nid)	Timestamp	Output
alpha	02:35:04.828	Hello!
alpha	02:35:05.029	Hello!
alpha	02:35:05.228	Hello!
alpha	02:35:05.248	Hello!

Figure 5.13.: The output logs view on the simulation page with an active filter.

5.3 Web service architecture

The *Diorama* software package consists of several applications running on a single self-contained virtual machine running *Ubuntu Linux*, which requires connections to the external web service, *Docker Hub*, external software repositories for different runtimes as well as, optionally, public git repositories (if a user chooses this as their code source for a node program). We summarise this visually in Figure 5.14.

Our web user interface takes the form of a web application, written in JavaScript using the React framework. We compile our React source code into static HTML, CSS and JavaScript files which we serve to the user's web client (browser) from the filesystem of our virtual machine using an Nginx server. We server this on port 80, the default HTTP port, of the virtual machine, so that it can be accessed by simply typing the IP address or DNS name of the virtual machine into the web browser.

The web application, running on the user's web client, establishes a two-way connection to the main server on the virtual machine through the WebSocket protocol. This WebSocket connection is the primary way that the client communicates with the virtual machine. This main server is written in Python and uses the *Tornado Web Server* framework. In addition to this, some particular communications between the client and the main server are performed through traditional HTTP requests, namely uploading ZIP files for user node programs and uploading network topology definition code. The role of the main server is to perform all the logic required in the *Diorama* application, store user data, such as programs and the network topology, and to provide an interface for the web application to interact with underlying *Docker* resources.

We run *Docker Engine* on our virtual machine, using *Docker* networks, images and containers as outlined by our design in Section 4.3. These *Docker* resources are managed by the main server through the *Docker Engine* API using the official Python SDK.

We use another *Tornado Web Server*-based Python application — the node logger. Its role is to monitor *Docker* containers' output messages, and to relay these back to the main server though HTTP requests.

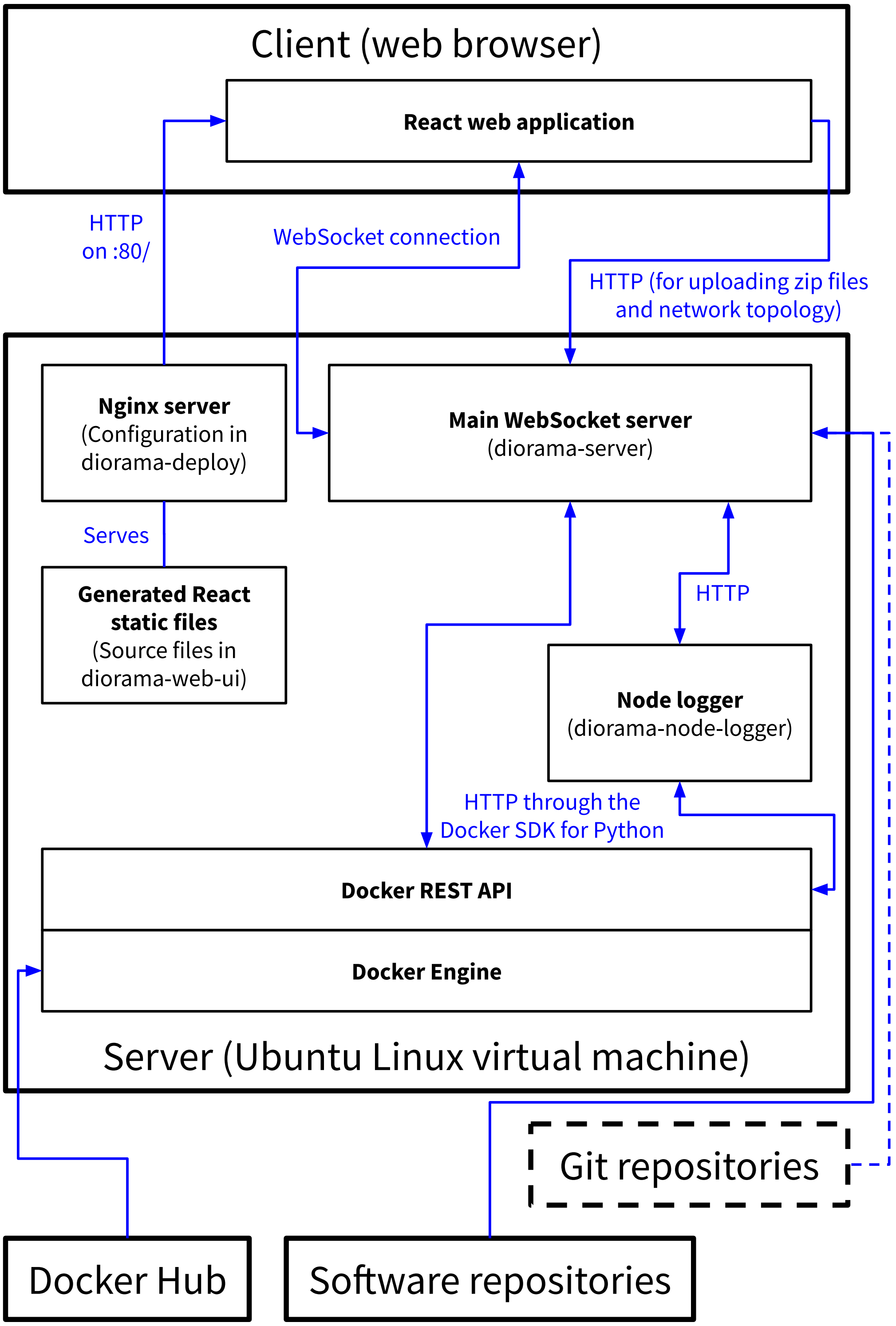


Figure 5.14.: A high-level view of the architecture for our web service.

5.4 Back-end: implementing programs, nodes and networks with *Docker*

Following our design (specifically Section 4.3), we used *Docker* images and containers to represent the node programs written by the user and their simulation nodes respectively.

5.4.1 Node program images and containers

We set out to create a *Docker* image for each of our supported runtimes, which when packaged with the user's node program code, would perform the correct functionality when run on a container, namely:

- install the user's specified dependencies.
- run the user's specified *main* method when initialised, performing the functionality expected by the user and defined in Section 4.4.1.
- through UDP, send messages to and receive messages from other nodes **if and only if** they are connected in the user's network topology.
- observe and satisfy the user's chosen message-passing success rates and (random) delays when sending messages.

We have implemented such a *Docker* image in the Python 3 runtime. Due to time constraints and because we judged it to be of relatively low priority, we have not created implementations in other runtimes. We have however demonstrated with our Python 3 runtime implementation, that building a satisfactory *Docker* image in other runtimes is certainly technically feasible. We believe that replicating this in other high-level and imperative runtimes, such as Ruby and Node.js (JavaScript), would be straightforward, building upon and using the ideas in the code for the Python 3 image we have built.

We refer to the program code (in this case, Python 3 code) we have written for the *Docker* image as our base node program.

Setup of the *Docker* image

Our base node program requires several extra files, generated based on the user's setup, to be packaged with it to set up and run:

- the user's node program file(s).

- the user's program's dependencies.
- a list of the IP addresses of all nodes in the network.
- a list of the node connection parameters (success rates and delays) for all node connections in the network.

```
1  for program in programs:
2      with tempfile.TemporaryDirectory() as _program_temp_dir:
3          program_temp_dir = os.path.join(str(_program_temp_dir), 'tmp')
4          shutil.copytree(os.path.join(constants.BASE_NODE_FILES_DIRECTORY,
                program['runtime']), program_temp_dir)
5          shutil.copy2(node_addresses_file_path, program_temp_dir)
6          shutil.copy2(connection_parameters_by_node_file_path,
               program_temp_dir)
7          get_code_for_program(program, program_temp_dir)  # copies code
               into <program_temp_dir>/user_node_files
8          inject_user_dependencies(program, program_temp_dir)
9          docker_interface.create_image(str(program_temp_dir), program['
               name'])
```

Figure 5.15.: A modified code snippet outlining the creation of node program *Docker* images.

We present a code snippet from the main *Diorama* back-end server in Figure 5.15 which outlines the steps taken when images are created. For each node program the user has created, we create a temporary directory (line 2) which contains the files required to create the *Docker* image (line 9). On line 4, we begin by copying across the base node program code into our temporary directory. On lines 5 and 6, we copy across files listing the IP addresses of all nodes and the node connection parameters for all node connections in the network respectively. We encode these lists as YAML files, but any data serialisation language, such as JSON, could be used here. On line 7, we obtain the user's node program files and place it into the temporary directory. This could be from any one of our three supported code sources, and so could involve cloning a public git repository and checking out a tag, unpacking an uploaded zip file, or simply creating a file from raw code that a user has submitted. On line 8, we generate our dependency list file by combining the dependency lists of our base node program and the user's program. In our case, since we are using Python and *pip* as our dependency manager and repository, the dependency list takes the form of a requirements file (`requirements.txt`).

```
1  FROM python:3
2
3  WORKDIR /usr/src/app
4
5  COPY requirements.txt ./
6  RUN pip install --no-cache-dir -r requirements.txt
7
8  COPY . .
```

Figure 5.16.: The Dockerfile for our Python 3 node program *Docker* image.

We present our Dockerfile for our Python 3 runtime node program image in Figure 5.16. It simply uses *pip* to install the listed requirements (lines 5 and 6) and copies across all the files in the aforementioned temporary directory (line 8). We build this on top of the official *Docker* image for Python 3 (line 1).

Structure of a Python 3 node program and running containers

We create *Docker* containers for each node, and for each, set its run command to:

```
python -u main.py
```

```
for node in nodes:
    peer_nid_list = ’,’.join(node[dict_keys.NODE_CONNECTIONS])
    run_args = [peer_nid_list, node[’nid’], str(node[’port’]), program[’main_method’]]
    docker_interface.create_container_and_connect(..., run_args, ...)
```

Figure 5.17.: A modified code snippet outlining the creation of node *Docker* containers’ command-line argument lists.

We also pass to each container four command-line arguments, as can be seen in Figure 5.17:

1. a comma-separated list of *nids* of all nodes connected to it
2. its *nid*
3. its UDP port
4. the Python path to the user’s *main* method.

```
src/
|-- user_node_files/
|   |-- my_node.py  # contains my_main() - the user’s main method
|   \-- utils.py
|
|-- NetworkAdapter.py
|-- Node.py
|-- Storage.py
|-- connection_parameters.yml
|-- main.py
\-- node_addresses.yml
```

Figure 5.18.: The simplified post-setup file structure of a Python 3 node program. We assume that the user has two files for their implementation — `my_node.py` and `utils.py`.

The post-setup file structure of a Python 3 node program is shown in Figure 5.18. The entry point is `main.py` and in this file, we parse the command-line arguments passed to it, as well as the `connection_parameters.yml` and `node_addresses.yml` files. We then create a `NetworkAdapter` object, which handles communication for the node, as well as a `Storage` object for the static persistent storage available to the user’s program. Finally, we create a Node object, which takes the created NetworkAdapter and Storage objects, and runs the main method of the user's program implementation (in this case, user_node_files.my_node.my_main).

The NetworkAdapter object uses Python's built-in socket library to send and receive UDP messages to and from other *Docker* containers, simulating nodes. It enforces message-sending success rates and delays for each connection as the user has defined in their setup of the network topology. It also implements the shape of the network topology by preventing the sending of messages to other nodes not connected to it according to the user's network topology definition (we explain in Section 5.4.2 that the user's network topology is actually implemented as a single fully-connected network).

```
1  delay_variables = {
2      "fixed": lambda params: params[’value’],
3      "uniform": lambda params: uniform.rvs(loc=params[’a’], scale=(params[
           ’b’] - params[’a’])),
4      "normal": lambda params: norm.rvs(loc=params[’mean’], scale=math.sqrt
           (params[’variance’])),
5      ...
6  }
7
8  class NetworkAdapter:
9      ...
10     def send(self, message, nid):
11         if nid not in self.peer_nids:
12             print(f"I tried to send a message to {nid}, but that nid
                   doesn’t belong to a node I’m connected to")
13             return
14
15         is_failed_send = random.random() > self.send_success_rates[nid]
16         if is_failed_send:
17             return
18
19         delay = max(0, delay_variables[self.send_delays[nid][’
               distribution’]](self.send_delays[nid][’params’]))
20
21         def send_message():
22             self.socket.sendto(message, self.address_port_from_nid(nid))
23
24         Timer(delay / 1000, send_message).start()
```

Figure 5.19.: A modified code snippet illustrating the behaviour of the send() function of the NetworkAdapter.

We illustrate this in the modified code snippet in Figure 5.19, which shows the behaviour of the send() function of the NetworkAdapter. Lines 11-13 prevent the sending of messages to unconnected nodes. Lines 15 to 17 implement the message sending success rate parameter of the appropriate connection in the network using Python's built-in random library. Line 19 determines the correct message delay for the sending of the message using the *SciPy* library. We show how this is done for some distributions in the delay_variables variable on line 1. On lines 21 to 24, we set the message to be sent using the `socket.sendto` function after this determined delay by creating and starting a `threading.Timer` object.

5.4.2 Networking nodes

When we start a simulation, we connect every node container in the entire network topology to a single *Docker* network, which the *bridge* driver. We set the subnet as the IPv4 address:

`172.190.0.0/16`

which allows for up to 65,534 containers to be connected to it (IPv4 addresses `172.190.0.1` through to `172.190.255.254` are available). *Please note that this subnet was arbitrarily chosen an bears no major significance.*

```
def get_ip_address_for_node_index(index: int, base_ip_address: str) ->
    str:
    return str(IPv4Address(int(IPv4Address(base_ip_address)) + index))
```

Figure 5.20.: The method which assigns IPv4 addresses to node *Docker* containers.

We see in Figure 5.20 the method which assigns IPv4 addresses to each node. The each node is given a unique and consecutive index ascending from 0 when passed into this method — this is the `index` argument. From this, we use the built-in `ipaddress` Python library to generate its IPv4 address: The node with index 0 is assigned `base_ip_address`, which is `172.190.0.1`, the the node with index 1 is assigned `172.190.0.2` and so on. All nodes use port 2000 to send and receive messages.

5.5 Back-end: Main WebSocket server

As mentioned in Section 5.3 our main server uses the *Tornado Web Server* framework, written in Python [68]. It has one WebSocket connection handler, as well as three HTTP endpoints: two for the front-end web user interface, and one with the node logger server, which we later describe in Section 5.5.5. This can be seen in lines 4 to 7 of the code snippet we present in Figure 5.21. We further explore the `WSHandler` object, seen on line 6, in Section 5.5.1.

```
def make_server() -> tornado.web.Application:
    return tornado.web.Application([
        (r"/", BaseHandler),
        (r"/uploadZipFile/(.*)", ZipFileUploadHandler),
        (r"/saveNetworkTopology", SaveNetworkTopologyHandler),
        (r'/ws', WSHandler),
        (r'/loggingMessage', LoggingMessageHandler)
    ])


if __name__ == "__main__":
    server = make_server()
    server.listen(2697)
    tornado.ioloop.IOLoop.current().start()
```

Figure 5.21.: A modified code snippet from the main method of our main server application.

```
1  {
2    "event": "deleteProgram",
3    "data": "{\"name\":\"hub_prog\"}"
4  }
5
6  {
7    "event": "getSimulationState"
8  }
```

Figure 5.22.: Two example WebSocket messages using our format protocol.

5.5.1 Using WebSocket messages

We establish a protocol for the format of the WebSocket messages send between the front-end (client) and the back-end application (server), which is a simplification of the Channels Protocol used by the technology company, *Pusher* [60]. As we exemplify in Figure 5.22, messages are encoded in JSON and are objects consisting of:

- an `event` field - a lower camel case string (we maintain a set list of possible values);
- and optionally, a `data` field - a string, an encoded JSON object if relevant to the `event`.

```
def parse_message(message):
        message_dict = json.loads(message)
        return message_dict['event'], json.loads('data') if 'data' in
            message_dict else None)
```

Figure 5.23.: The WebSocket message parser method.

As we show on line 3 of Figure 5.22, the `data` field is a JSON object which we double-encode, in order to enforce consistency in how it is encoded. We show in Figure 5.22

the method within `WSHandler` which parses the messages into a Python tuple. (The equivalent action is performed on the front-end, using JavaScript.)

```
 1  class WSHandler(tornado.websocket.WebSocketHandler):
 2  ...
 3      def on_message(self, message):
 4          event, data = self.parse_message(message)
 5          handle(event, data)
 6          handlers[event](data, self.send_message)
 7
 8      def send_message(self, event, data):
 9          self.write_message(json.dumps({'event': event, 'data': json.dumps
                (data)}))
10
11
12  handlers = {
13      ...
14      'addProgram': (lambda data, _: programs.add_program(data)),
15      'getPrograms': (lambda _, send_func: send_func('programs', programs.
            get_programs())),
16      ...
17  }
```

Figure 5.24.: A code snippet showing how WebSocket messages are handled.

In Figure 5.24, we show how we handle received WebSocket messages. Upon receiving a WebSocket message, we perform one of two actions, determined by the `event`:

- Update the of the state to reflect a change made. For example, we see on line 14 that in response to an `addProgram` `event`, we add the program encoded in the `data` field.

- Send a WebSocket message in response to a request for data. For example, on line 15, we reply with a the list of stored programs in response to a `getPrograms` event.

5.5.2 Storing data persistently

The main server handles the storage of user-generated data, including node programs, the network topology and custom configuration (the main server does not store dynamic state, such as unsaved edits — these are maintained in the front-end web user interface, as explained in Section 5.6.1). We use the *TinyDB* library, which is a simple document oriented database that stores data on disk as JSON files [65]. We chose this library since several of its features were advantageous to us:

- It does not use an external server. An external server would have added to the overall complexity of our overall web architecture.

- Its API allows direct storage and retrieval of documents of Python types, including booleans, dictionaries, strings and integers. This meant we did not need to handle encoding or decoding of stored and to-be-stored data.

- It stores documents in JSON files, which allowed us to easily read databases when performing debugging.

```
1  programs_db = \textit{TinyDB}(’out/programs_db.json’)
2
3
4  def add_program(data):
5      programs_db.upsert(data, Query().name == ’name’)
6
7
8  def get_programs():
9      return programs_db.all()
```

Figure 5.25.: A code snippet showing how a *TinyDB* is used for storing and retrieving.

We demonstrate how it is used in the modified code snippet in Figure 5.25. On lines 4 and 5, we show how a user-defined program is stored. *TinyDB* databases are unindexed, so we implement our own alternative: we effectively use the value of the `name` in a program dictionary as its index. The code on line 9 simply returns a list of all program dictionaries stored in the database.

5.5.3 Interacting with *Docker Engine*

We interact with *Docker Engine* by using the *Docker SDK for Python* [25]. This is a library which interacts with *Docker Engine's* HTTP API, and we chose to use it since was easier and quicker to program with, compared to the alternative of directly making HTTP requests.

We structured the code so that all interactions with *Docker Engine* reside in one file — `docker_interface.py`. We show an outline and an example use of the library in Figure 5.26.

```
import docker

DOCKER_CLIENT = docker.from_env()

...

def create_image(path, tag: str):
    DOCKER_CLIENT.images.build(path=path, tag=tag, rm=True)

...
```

Figure 5.26.: An outline of our `docker_interface.py` *Docker SDK for Python*.

5.5.4 Fetching node program code

Users can provide their code for a node program in three ways: directly typing or pasting it into the code editor; uploading a zip file; or providing a URL and branch or checkout tag to a public git repository. In cases where uses use the first way and directly input their code, we simply store it using *TinyDB*, as illustrated in Section 5.5.2.

```
def write_zip_file(program_name, file_data):
    with open(f"out/program_zip_files/{program_name}.zip", "wb") as fh:
        fh.write(file_data)


class ZipFileUploadHandler(GeneralHTTPHandler):
    def post(self, program_name):
        write_zip_file(program_name, self.request.body)
        self.write('Upload successful')
```

Figure 5.27.: A code snippet showing how we handle received zip files for user node program code.

We enable users to upload their zip files by sending them to our server in the body of an HTTP POST request from the front-end. On line 4 of Figure 5.21, we show that the endpoint path for this is `/uploadZipFile/[PROGRAM_NAME]`. We save the received file in the directory `out/program_zip_files/[PROGRAM_NAME].zip` relative to the root directory of the server's source code and show this in Figure 5.27.

For code programs where the code source is a git repository, we use the *GitPython* library [71] to clone the repository from the URL provided by the user, and then checkout the given branch or revision tag. We show this in lines 17 and 18 of Figure 5.28.

We show in Figure 5.28 the process of retrieving code for a node program in the process of creating its *Docker* image when preparing for a simulation. In it, the `temp_dir` argument is a temporary directory we create for compiling together all the code required for creating the *Docker* image, including the base node program file for the program's runtime and the Dockerfile. We place the code retrieved into this temporary directory.

5.5.5 Handling *Docker* container messages: node logger server

One of the most difficult challenges we faced when building the back-end was obtaining output messages from running *Docker* containers, and sending them to the web client through the existing WebSocket connection live, as these messages were being generated. The *Docker SDK for Python* provides us with a *blocking* generator for streaming container output messages [25]. We would usually simply use multithreading to solve this problem, and have the blocking generator run in a separate thread to not prevent the rest of the server from running. However, the *Tornado Web Server Framework* uses Python's built-in

```
def get_code_for_program(program, temp_dir):
    code_source = program[’code_source’]
    assert code_source in [’zip’, ’git’, ’raw’]

    dir_to_write_to = os.path.join(temp_dir, ’user_node_files’)
    os.mkdir(dir_to_write_to)

    if code_source == ’raw’:
        file_name = ’’.join(
            [’node’, constants.FILE_EXTENSIONS_FOR_RUNTIME[program[’
                runtime’]]])
        with open(os.path.join(dir_to_write_to, file_name), ’w’) as file:
            file.write(program[’code_data’][’raw_code’])
    elif code_source == ’zip’:
        with ZipFile(f"out/program_zip_files/{program[’name’]}.zip", ’r’)
             as zip_file_obj:
            zip_file_obj.extractall(dir_to_write_to)
    elif code_source == ’git’:
        git_repo = Repo.clone_from(program[’code_data’][’repo_url’],
            dir_to_write_to)
        git_repo.git.checkout(program[’code_data’][’
            checkout_branch_or_tag’])
```

Figure 5.28.: A code snippet showing how we retrieve a user’s code for a node program.

asyncio module. Unfortunately, using concurrency in such a situation was not part of *Tornado’s* documentation, and despite much experimentation and searching, we were not able to implement this by taking this approach.

We instead opted to use a separate process to stream container logs to the front-end web client in the form of another web server. We outline its operation in Figure 5.29. This web server, which we label as the “Node logger” in Figure 5.14 uses the *Docker* API to monitor and stream output messages for running *Docker* containers, and sends these to our main back-end server using HTTP calls in a JSON object. This JSON object also contains two pieces of metadata — the name of the container from which the message was output, and the time at which the message was output (according to the container itself, to avoid delays in receiving the message from the *Docker* API from affecting this). We show this in lines 15 to 19.

A challenge we faced when using this approach was preventing the front-end from receiving duplicated messages, when its WebSocket connection had re-connected from after a disconnection (after a disconnection, the main server would have requested *all* output messages from containers meaning that it would have again received the messages it had already received before the disconnection). We solve this problem by having the front-end send to the back-end the timestamp of the latest message it had previously received, and then our node logger server simply doesn’t send all output messages with a timestamp *before* this. The `since_raw` argument on line 5 is the timestamp of the latest

```
1  DOCKER_CLIENT = docker.from_env()
2  container = DOCKER_CLIENT.containers.get(name)
3  generator = container.logs(stream=True, timestamps=True, since=since)
4
5  def stream_logs_from_generator(name, since_raw):
6      asyncio.set_event_loop(asyncio.new_event_loop())
7      http_client = HTTPClient()
8      while True:
9          try:
10             line: bytes = generator.__next__()
11             split_line: List[bytes] = line.split(b’ ’, maxsplit=1)
12             timestamp = split_line[0].decode(’utf-8’)
13             if since_raw and timestamp <= since_raw:
14                 continue
15             to_send: Dict[str, str] = {
16                 ’timestamp’: timestamp,
17                 ’message’: split_line[1].decode(’utf-8’),
18                 ’nid’: name
19             }
20             http_client.fetch(
21                 HTTPRequest(url=constants.MAIN_SERVER_LOGGING_MESSAGE_URL
                       , method=’POST’, body=json.dumps(to_send)))
22         except StopIteration:
23             break
```

Figure 5.29.: An outline of the logic in our node logger server.

message which the front-end client has, and we enforce the requirement of all messages to be sent to be after this time on lines 13 and 14.

5.6 Front-end:React web application

Our front-end web user interface is a web application built using the React JavaScript framework [33]. We chose to use this because its component-based programming model and state management capabilities make it well-suited to our web user interface’s requirements. It is also extremely popular in the open-source community, meaning that many useful and well-built open-source libraries are available for it, it is very well-documented and many community-written guides and articles on using it are available. These factors made using it to build *Diorama’s* front-end much easier and more efficient.

For the visual design of our user interface, we use the *MetroUI* CSS library [59] to help us. It provided us with styling for HTML elements, as well as some useful React components, such as the *Wizard*, which we use in the Programs viewer when the user creates a new program.

We connect the web interface to our main back-end server application through a WebSocket connection. We expose this on port 2697 of our server and make the connection during the initialisation process of our React application. In the case where this Web-

```
function connectWebSocket() {
  websocket = new WebSocket(`ws://${window.location.hostname}:2697/ws`);
  websocket.onopen = onSocketConnected;
  websocket.onclose = () => {
    onSocketDisconnected();
    const reconnect = setInterval(() => {
      clearInterval(reconnect);
      connectWebSocket();
    }, 2000);
  };
  websocket.onerror = onSocketError;
  websocket.onmessage = onMessage;
}
```

Figure 5.30.: The function which creates the WebSocket connection from the web interface to the main back-end server application, which is executed on initialisation.

Socket connection is broken for some reason, for example, a temporary loss of internet connection on the user's device, we attempt to re-connect every two seconds. We show these things in Figure 5.30.

5.6.1 State management

We engineer the front-end such that React components are as small as is practical, adhering as much as possible to the *single responsibility* principle [58]. This is good practice since it makes it easier to reuse components where needed, and simplifies and separates logic, making it easier to maintain code. It does however mean that components can easily become nested many levels down in the overall component tree. If we store data in the state of React components, we may need to pass it down the component tree through many components in order to get it to the component where it is needed (parameter passing). This technique is called "props-drilling" and is generally regarded as an anti-pattern and thus avoided by React developers; this is because it can cause many problems when developing, such as fragility and inconsistent naming of data.

For state data which we know will be used across multiple components, we use the *Redux* open-source JavaScript library for maintaining and managing the global state of our web application [1]. *Redux* stores a global state (a plain JavaScript object) separate of React components, which along with *reducer* functions, are called the *store*. Reducers are functions we define which take the state and an *action*, then returns the next state. Components can access the *Redux* store by being connected to it using the `connect()` function. This allows them to use data from the state, as well as to dispatch actions to reducers to modify the state.We show an example of this in Figure 5.31, where the `connect()` can be seen on line 35. The connect function takes four optional arguments, the first of which is `mapStateToProps` - a function which selects parts of the global state object and passes them to the connected React component as properties, or *props*.

```
import React, { Component } from "react";
import PropTypes from "prop-types";
import { connect } from "react-redux";

...

class PreferencesDialog extends Component {

  ...

  render() {
    const { colourScheme } = this.props;
    return (
      <div ... >
        ...
        <div ... >
          ...
          <select defaultValue={colourScheme} ... >
            ...
          </select>
          ...
        </div>
        ...
      </div>
    );
  }
}

function mapStateToProps(state) {
  return {
    colourScheme: selectCustomisation(state).colourScheme
  };
}

export default connect(mapStateToProps)(PreferencesDialog);
```

Figure 5.31.: An example of our use of *Redux* with a React component.

We show the outline the global state of our *Redux* store in Figure 5.32, and explain some parts of this in in-line comments.

5.6.2 Page routing

Other than the main elements of our user interface — the top bar and the navigational side bar on the left of the screen — our web application consists of several different pages and so we need a way of routing our web application. We use the open-source library *React Router* for this [61]. In Figure 5.33, we show the top-level, or *root* React component, called `Diorama`. We see on lines 19 and 21 to 27 the routes used in our web application, as well as the React components for the pages for each of these routes. These are enclosed within the `BrowserRouter` (which we have aliased as `Router`) component provided by *React Router*. On lines 13 and 16, we have our fixed top bar (`AppBar` component) and navigational side bar (`SideNav` component) respectively.

```
 1  {
 2    customisation: { colourScheme: " ... " }, // user interface
        customisations
 3    programs: [ ... ], // a list objects containing data for each node
        program, such as runtime and code data
 4    socket: { // WebSocket status
 5      connected: false
 6    },
 7    networkTopology: { // the user's network topology
 8      rawNetworkTopology: " ... ",
 9      unpackedNetworkTopology: [ ... ],
10      language: 'YAML'
11    },
12    customConfig: { ... }, // advanced custom configurations
13    polyglot: { ... }, // Polyglot.js phrases for each supported language
14    simulationState: " ... ", // the stage of the simulation, including
        initialised, setting up program images and running
15    simulationNodes: [ ... ], // a list of objects containing data about
        the Docker container representing each node in the current
        simulation
16    simulationLogs: [ ... ], // the output logs for each node container
17    simulationLogsFilter: { ... }, // user inputs for the output logs
        filter
18    editingConnectionParameters: { ... }, // user inputs for the connection
         editor modal for editing success rate and delays
19    connectionParameters: { ... }, // delays and success rates for each
        connection in the user's network topology
20    userEvents: { ... }, // user's planned scheduled node action events
21    currentSimulationHash: " ... " // hash of the parameters used for the
        current running simulation, in order to detect if changes have been
         made since it started
22  }
```

Figure 5.32.: An outline of the state of our *Redux* store.

5.6.3 User event scheduling

We implement user-scheduled node action events, whose interface we described in Section 4.4.3, using JavaScript timers. In the user event scheduler modal we show in Figure 5.11, when each user adds an event, it is added to the `userEvents` object in our *Redux* state, shown on line 20 of Figure 5.32. When the user clicks the *Run* button in the modal, we create timers for each scheduled event using the built-in `setTimeout()` method, as we show in Figure 5.34. These timers are set to send a message to the main back-end server to execute the action on the chose node container (line 3) and display a *toast* to the user (line 4) after the scheduled delay for the user event.

```
import { BrowserRouter as Router, Route } from "react-router-dom";
import ...

class Diorama extends Component {

  ...

  render() {
    ...
    return (
      <Router>
        <Fragment>
          <AppBar />
          <div ... >
            <div ... >
              <SideNav />
            </div>
            <div ... >
              <Route exact path={"/"} component={ProjectHome} />
              <div ... >
                <Route exact path={"/docs"} component={Documentation} />
                <Route path={"/docs/:interfaceLanguage"} component={Viewer} />
                <Route exact path={"/programs"} component={Programs} />
                <Route path={"/programs/:programName"} component={ProgramEditor} />
                <Route exact path={"/network-topology"} component={NetworkTopology} />
                <Route exact path={"/custom-config"} component={CustomConfig} />
                <Route exact path={"/simulation"} component={Simulation} />
              </div>
            </div>
          </div>
        </Fragment>
      </Router>
    );
  }
}

function mapStateToProps(state) { ... }

export default connect(mapStateToProps)(Diorama);
```

Figure 5.33.: A code snippet showing how we use *React Router*.

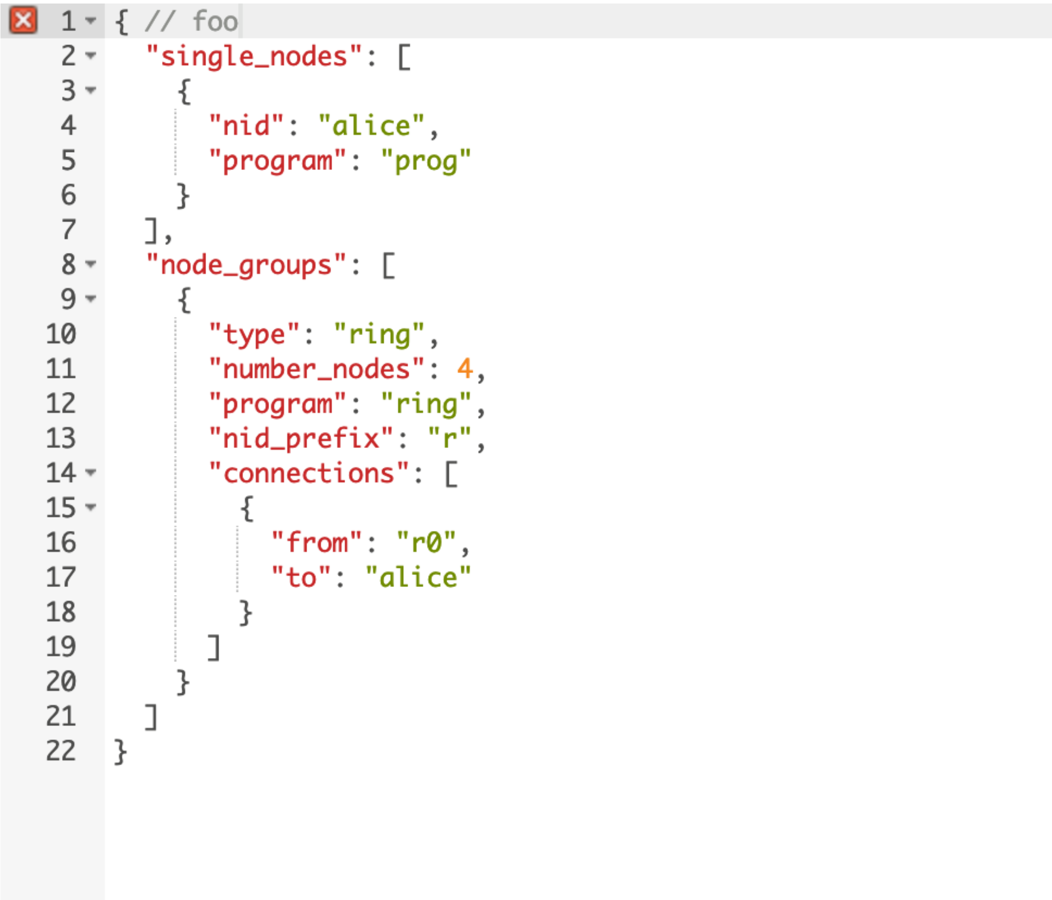
```
orderedUserEvents.forEach(({ time, node: nid, action }) =>
  setTimeout(() => {
    Socket.send("performNodeAction", { nid, action });
    Metro.toast.create(`${action} ${nid}`, noop, toastTimeout, "info");
  }, time);
);
```

Figure 5.34.: A code snippet showing how we use JavaScript timers to implement scheduled user events.

5.6.4 Use of selected third-party libraries

We select and describe three of the other three third-party libraries we have used in our web application.

***Ace* code editor**



```
{ // foo
  "single_nodes": [
    {
      "nid": "alice",
      "program": "prog"
    }
  ],
  "node_groups": [
    {
      "type": "ring",
      "number_nodes": 4,
      "program": "ring",
      "nid_prefix": "r",
      "connections": [
        {
          "from": "r0",
          "to": "alice"
        }
      ]
    }
  ]
}
```

Figure 5.35.: The *Ace* code editor for users to edit their network topology.

For the user to edit their code for node programs or their network topology, we use the *Ace* code editor [54], which we show in Figure 5.35. We chose to use this since it provides several benefits to the user:

- syntax highlighting, which is extensible for custom languages.
- line number indicators.
- included colour and font themes, as well as the ability to extend these with our own custom ones.
- automatic indentation.
- the ability to drag and drop text.

- live syntax checker for some languages (this can be seen working on line 1 in Figure 5.35, where there is a red cross).

Another advantage of using *Ace* was that it was straightforward to embed into our React application using the *React-Ace* library [42], which gives us a React component, with a simple interface, to work with. We tried to use several alternatives before settling on *Ace*. *CodeMirror* [40] and *Monaco Editor* [52] were two particularly strong candidates, which also offered the features we listed. However, their pitfalls were that they both required more complex work to embed them into our web application — particularly the need for using the *webpack* module bundler [32] directly.

Visualising network graphs with *vis.js*

We use the *vis.js* JavaScript visualisation library [4] to display graphical network graphs to the user, as shown in Figure 5.6. This is a very powerful open-source library, whose very wide range of customisation options allowed us to craft a viewer in the way that is most well-suited to its purpose within *Diorama*. We especially make use of several particular features:

- formatted text inside nodes.
- mouse event callback functions (for when user double-clicks on an edge).
- ability for user to move nodes around.
- node groups so the same colour could be used for nodes running the same program.

Preparing for internationalisation with *Polyglot.js* and *Moment.js*

Since we knew that it would have been much easier than to do so from the start of writing code than retrospectively, we used two internationalisation libraries in our web application. This makes it much easier to in future, translate the application into languages and locales other than British English.

We use *Polyglot.js* [3] to provide translations of all text displayed in our user interface. Instead of writing such text directly in our React components (this is effectively hard-coding), we dynamically load the strings to be displayed to the user, based on which language in which they have selected to view the web interface. We use an additional library, *redux-polyglot* [69], which is a toolset that allows us to store translations for each language and the currently selected language and locale in the *Redux* state. In the code base, we store *non-capitalised* translations of phrases in separate JavaScript files for each language. We show how this is used in Figure 5.36, where we translate the phrase represented by the string, `networkTopology`.

To display dates and times in the user interface, we use the *Moment.js* library [44]. One of the capabilities of this library is to format dates and times in localised formats; for example, the short date format for 17TH June 2019 is *17/06/2019* in British English, but 06/17/2019 in United States English. Combining this with *Polyglot.js* allows us to have our interface in different languages with appropriate date and time formats for the selected language (and in some cases, locale). We create a `DateTime` React component, which uses *Moment.js* and the `Polyglot.js` locale from the *Redux* state to display dates and times in the appropriate localised format. This component is used throughout the web application where dates or times are displayed to the user and is shown in Figure 5.37.

The *NetworkTopology* React component. The translation is dynamically loaded from the *Redux* state on line 8:

```
 1  import { getP } from "redux-polyglot";
 2  ...
 3
 4  class NetworkTopology extends Component {
 5    ...
 6
 7    render() {
 8      const { p } = this.props;
 9      ...
10      return (
11        <Fragment>
12          <span ... >{p.tc("networkTopology")}</span>
13          ...
14        </Fragment>
15      );
16    }
17
18  function mapStateToProps(state) {
19    return {
20      p: getP(state),
21      ...
22    };
23  }
24
25  export default connect(mapStateToProps)(NetworkTopology);
```

The British English phrases file, `translations/en_gb.js`:

```
export default {
  networkTopology: "network topology",
  ...
}
```

The German phrases file, `translations/de.js`:

```
export default {
  networkTopology: "Netzwerktopologie",  // nouns in German must be capitalised
  ...
}
```

Figure 5.36.: Code snippets showing how we use *Polyglot.js* and *redux-polyglot* for translating user interface text.

```
import moment from "moment";
import "moment/min/locales";
import { getLocale } from "redux-polyglot";
import ...

class DateTime extends Component {
  render() {
    const { locale, dateTime, format } = this.props;
    return (
      <Fragment>
        {moment(dateTime)
          .locale(locale)
          .format(format)}
      </Fragment>
    );
  }
}

function mapStateToProps(state) {
  return {
    locale: getLocale(state)
  };
}

export default connect(mapStateToProps)(DateTime);
```

Figure 5.37.: The `DateTime` React component.

5.7 Installing and running *Diorama*

The source code for *Diorama* is stored across five public *GitHub* git repositories:

- *diorama-web-ui* - source files for our React web application (JavaScript). `https://github.com/mauriceyap/diorama-web-ui`.
- *diorama-docs* - user documentation, served as part of the React web application (Markdown). `https://github.com/mauriceyap/diorama-docs`.
- *diorama-server* - main back-end server, which the React web application connects to (Python). `https://github.com/mauriceyap/diorama-server`.
- *diorama-node-logger* - application which monitors output from generated *Docker* containers and relays this to the main back-end server (Python). `https://github.com/mauriceyap/diorama-node-logger`.
- *diorama-node-base-python3* - base node program from which node program *Docker* images for the Python 3 runtime are created (Python). `https://github.com/mauriceyap/diorama-node-base-python3`.

We have created a deployment tool for prospective users to deploy *Diorama* to their existing virtual machine using the *Ansible* deployment tool [62], which is stored on another public *GitHub* repository — *diorama-deploy* at `https://github.com/mauriceyap/diorama-deploy`. This repository includes a README file which contains instructions to use it.

Ansible deployment works by running a series of *tasks*. A task involves establishing an SSH connection to the target machine, which is the VM on which the prospective user wants to run *Diorama*, and then running some command to perform some action (or to establish that there is no need to do so). We define a (ordered) list of tasks, called an *Ansible playbook*, which when successfully completed, installs and runs *Diorama*. At a high level, these tasks are (for the sake of simplicity, not in order):

- installing *Docker*,
- installing and configuring Nginx to serve generated static files for the front-end React web application on port 80,
- installing and configuring *Supervisor* [2] to run the main WebSocket server and node logger server,

- configuring *systemd* to run *Supervisor* when the VM is switched on.

```
language: node_js
node_js:
  - "stable"

notifications:
  email: false

cache:
  directories:
  - node_modules

script:
  - yarn build

after_success:
  - zip -r build.zip build/

deploy:
  provider: releases
  api_key: $github_token
  file: build.zip
  skip_cleanup: true
  overwrite: true
  on:
    tags: true
```

Figure 5.38.: The *Travis CI* configuration file for the *diorama-web-ui* project.

We generate static HTML, CSS and JavaScript files for the front-end React application using *Travis CI* [70], a hosted continuous integration service. We add the `stable` git tag to the latest commit in *diorama-web-ui* which is ready to be deployed. We configure *Travis CI* so if a pushed git commit if it has this `stable tag`, it builds the static files for the given source code, compresses them into a zip file and uploads them to *GitHub Releases* (`https://github.com/mauriceyap/diorama-web-ui/releases`). We show `.travis.yml`, the *Travis CI* configuration file for the project in Figure 5.38, which shows all of these steps.

6 Evaluation

In this chapter, we evaluate the extent to which the *Diorama* proof-of-context meets the goals which we designed for it to meet in Section 4.2, those we presented in Section 5.1, as well as how our project as a whole meets the high-level objectives we set out in Section 1.1.

Our aim for *Diorama* was, in essence, to be easy to set up and highly usable by students and teachers alike, while providing them the functionality they need to create, test, demonstrate and analyse their own distributed algorithms on their own networks. We wanted the proof-of-concept to be a fully-working implementation which was immediately usable with lots of core functionality, but was also in a state which is easily extensible by collaborators in the open source community.

6.1 Supported functionality

In Section 1.1, we set out to create an application where users are able to test distributed algorithms by programming nodes, define how they are connected to each other, run them simultaneously and observer their output in real-time.

Diorama provides users a way to program nodes in their network by letting them add to a library of node programs which they have created. In the same way that *AWS Lambda* requires users to create a *handler* function to create a serverless function, users of *Diorama* create programs by implementing the *main* method of the node it is running on. Our API is defined by the arguments passed into it; this API provides simple methods to send messages to and receive messages from other nodes which are connected to it, a feature of several pieces of related work we explored and which we desired for *Diorama* in Section 3.8. One major desired feature which *Diorama*, at least in its current state, fails to achieve is language agnosticism. Currently, users must use the Python 3 runtime to implement their node programs. This failure came about because support for other languages was given a very low priority relative to other features, and due to the limited time we had to complete the project, it was not completed. We took this decision because building *Diorama* to support one runtime in the way that we did demonstrates that we can easily support for implementing nodes in other runtimes in future (we evaluate and support this particular claim further in Section 6.4.1.

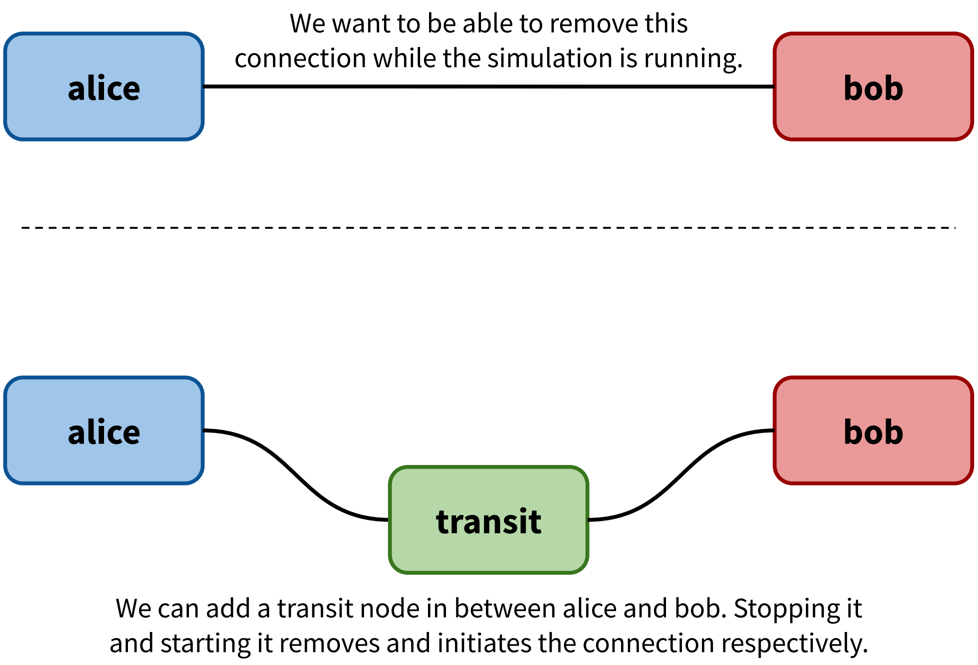

We want to be able to remove this connection while the simulation is running.
alice
bob
alice
transit
bob
We can add a transit node in between alice and bob. Stopping it and starting it removes and initiates the connection respectively.

Figure 6.1.: Suggested workaround for adding and removing connections during the running of a simulation.

Through the network topology editor, users can define which nodes are in their network and how these nodes are connected to each other using our network topology API. We also provide users with a visualisation of the network they have defined in code and allow them to give random or fixed dealys as well as message-passing failure rates to any connection between any two nodes in their network. Whilst users can edit their network topology as much as they want before simulating it, we fail provide a way to edit their topology while a topology is running so that they can see the effect of changes as they are made. This was a feature required by Leroy's user journey, as well as one we identified in related applications in Section 3 which we wanted *Diorama* to have. We did not implement this feature, again, due to time constraints, however, we do believe that this is technically feasible and present an outline of how we could achieve this in future in Section 7.1. Using *Diorama* in its current state, we suggest a workaround for adding or removing a connection while the simulation is running: if nodes *alice* and *bob* are connected, we can indirectly connect them through an extra *transit* node, whose behaviour is to simply pass messages through, and then stop and start this *transit* node to effectively remove and initiate the connection between *alice* and *bob*. We illustrate this in Figure 6.1.

Diorama achieves the aim of enabling users to simulate their network by running nodes simultaneously and to view their outputs in real time. On our simulator page, in the node manager tab, users can start, and stop individual or groups of selected nodes, while in the log viewer tab, lines of output from each node are displayed, coloured according to the node. As required by Stacey's user journey in Section 4.2.2, users can export this data by downloading a CSV file, which can be opened in a spreadsheet software package, or copy the data to their clipboard, which can be directly pasted into a new spreadsheet document. In addition to the requirements of the user journeys, we enable users to filter these output messages, so that only desired messages are displayed. This particular feature would be useful to a lecturer demonstrating a simulation, particularly in a network where there are a lot of output messages, in that it would allow them to focus on specific messages and thus parts of the behaviour of the network.

6.2 Usability

One of the stated key objectives of this project was to create a solution which is "simple and quick to set up and use" (Section 1.1). To evaluate the extent to which *Diorama* achieves this, we conducted user-testing with people who have never seen or used *Diorama*. (Since we created, worked on and have used the interface for several months, it would have been unreliable to us to exclusively assess this ourselves given our unusually high level of familiarity and set opinions.) Our five volunteer testers were people from a range of relevant backgrounds who had knowledge and experience in programming. In this way, we could simulate how *Diorama* would be used and received by its intended audience. Our five testers were[1]:

- **Matt** - a full-stack software engineer working at a large multinational financial services company. He graduated from a top-20 UK university one year ago, having studied Computer Science for four years. Matt took a course on Distributed Systems as part of his degree and has a good knowledge of distributed algorithms, having studied and implemented some himself. Matt also has very extensive programming experience as well as experience of using and writing technical documentation for programmers.

- **Gordon** - a third-year Aeronautical Engineering student at Imperial College. He is a proficient programmer, having had a summer internship working as a front-end engineer for a large global technology company. He had never heard of, nor come across any concepts in distributed computing, including distributed algorithms.

- **Nick** - also a third-year Aeronautical Engineering student at Imperial College. He has some programming experience and also didn't know about distributed algorithms.

- **Alan** - a fourth-year Computing student at Imperial College. He is an experienced programmer and took a Distributed Algorithms course last year. As part of this

[1] The names are changed for sake of anonymity.

course, he studied, implemented, tested and analysed distributed algorithms by creating networks using Docker containers. Like Matt, both Alan and Michael have had lots of experience using documentation like API guides.

- **Amy** - a second-year Computing student at Imperial College. She is an experienced programmer, but has not studied or come across distributed algorithms.

6.2.1 User testing

We set each of our testers a series of tasks which would cover as much of the functionality of *Diorama* as possible. We would observe whether or not they could perform these tasks using our proof-of-concept application, how easy it was for them to do so and whether or not they had any problems, and also take any verbal feedback. We wanted to avoid giving too much guidance to our volunteers during their attempts to perform the set tasks, in order to keep the test as accurate of a representation as possible of what real users would experience when first using it independently. However, for the sake of saving volunteers' time, we allowed ourselves to give pointers about aspects of the test not directly related to the usability of our implementation, for example, using the Microsoft Azure interface to create a VM instance, using the Python language, how to implement the given node program (high level algorithm principles, not how to code it using our API) and YAML syntax. After the test, we asked a series of questions to gather feedback.

All testers other than Gordon and Matt tested *Diorama* using laboratory desktop computers, each running Ubuntu Linux 18.04.2 LTS without sudo access, using Google Chrome as the web browser. Gordon used a laboratory computer running OSX 10.14.2, also without sudo access, and using the Safari web browser. Matt used his own laptop, running OSX 10.15.5 with sudo access and using Firefox as his web browser.

We provide the guide document given to testers in Appendix C. The tasks we set were:

1. to create a virtual machine and use the installation instructions (provided in the *README* file of the deployment tool repository) to install *Diorama* onto it.

2. to create three node programs: two already provided in publicly-accessible git repositories (we provide these in Appendix D) and one to code using the in-browser code editor.

3. using our API, to create a provided network topology (using just the *single nodes* part of our API).

4. to run the nodes in the network they have created.

5. to export the node output data to a spreadsheet.
6. to introduce failure rates for particular connections in the network.
7. to introduce delays for particular connections in the network.
8. to create a network topology using our *API* which incorporates an automatically-generated node group (namely a star topology).

We include the notes we took during user testing in Appendix E.

Setup and installation

Of our five testers, we only asked Matt, Amy and Nick to attempt to create cloud VMs and to install the software onto them. Given the high level of computing experience and technical setups of the other testers, it was safe to assume that should these three selected users be successful in doing this, the others would be able to as well. Matt and Amy were both able to install *Diorama* with no problems at all. The tool worked as intended from a technical perspective, and none of the installation process required any prompts from us, other than some help using Azure and selecting parameters (not related to Diorama). Both Amy and Matt found the installation instructions clear and easy to follow. Amy expressed that she liked the fact that after creating a VM, installation only required running one single command. Nick struggled to use Microsoft Azure to create a VM instance, perhaps because he had never set up a cloud virtual machine, nor in fact used cloud services at all. I had to guide him on which parameters to select for each part of the new instance setup. This was not part of the testing for our proof-of-concept, but it was still useful to have observed.

After setup had been completed, we invited the testers to each implement the network (programs as well as topology) described in the testing guide. We deliberately did not tell them to navigate to the programs page. Every one of the five testers used the walkthrough of the application on the home page of the web application. Other than Matt, they all spent some time reading the documentation to familiarise themselves with the programming model before starting to implement the network.

Creating programs

All testers were able to create the first two programs, *sender* and *receiver* — whose code we had written in advance and provided in a public git repository — with little trouble.

Both Nick and Gordon made the same mistake which had to be corrected, in that the URL they entered for the git repository was that of the repository's project homepage on GitLab's web interface, as opposed to the address of the repository itself. Amy copied the code from the repository and pasted into *Diorama's* in-browser code editor for the *sender* program, which would have worked, but was not the user journey we had intended for. We hinted that *Diorama* could import code directly from a public git repository, and this prompted her to use the git repository code source for her *receiver* node program, as we intended. Matt commented that this feature (directly importing a git repository) could be very useful for students working in groups, since it allowed easy collaboration.

Gordon, Alan and Matt were all able to implement the *relay* program using our API with the behaviour which our testing guide had specified. Both Gordon and Matt remarked that the API documentation provided was very clear and they found it easy to read and use. Amy and Nick were both able to independently write code to implement the core of the node behaviour using our API (though Nick required some assistance with Python syntax). Both their implementations had however not included the given constraint that only messages not previously received should be re-broadcast to neighbouring nodes. Amy also did not exclude the sender of a message when listing nodes to whom to rebroadcast the message. Amy's and Nick's problems were not related to our web interface, API or documentation, so we thought it appropriate to suggests ways to correct these in their algorithms. Alan had trouble finding the API documentation accordion element on the program editor page, so he opened the documentation to the side in a separate window. Nick, though he found it eventually, also said that this was difficult to find as it was not immediately obvious that it was there.

Defining and modifying the network topology

After creating programs, testers attempted to use our network topology API to define the network we had illustrated graphically, made up of eight single nodes (i.e. no groups), each running one of the three programs. All five volunteers were able to understand the documentation for this API and were all eventually able to produce the desired topology correctly. Comments about the documentation itself were overwhelmingly positive. Matt said that concrete code examples and accompanying ASCII art visualisations were very useful to help understand the interface. One common struggle was using the YAML language's syntax. Nick, Gordon and Amy all required assistance with this, but this was an aspect of our test. They all had the option to use JSON and documentation was provided for the use of this language, but they all elected not to due to its undesirable verbose nature.

Later on in the test, testers were asked to add delays and failure rates to selected connections within the network. There was mixed success here — those who did not immediately see the bold text telling users to double click on connections in the network graph visualiser in order to add such parameters naturally looked to the API documentation to find out how to do this (they would ultimately have been unsuccessful, so we pointed this out to not waste time).

Matt, Alan, Gordon and Amy attempted to create another different network topology of which we had provided an illustration (*Step 3* of the user testing guide in Appendix C). This topology included a star topology node group in addition to a single node. All testers took a longer time to read and understand the documentation for creating node groups, but with the exception of Amy, all were able to independently code what was required. Amy required some prompting about the structure of the network topology definition code when she was initially unsuccessful, but was able to achieve success afterwards (we told her to re-read the first part of the documentation and did not explicitly tell her the changes she needed to make).

Two common criticisms were made about the network topology editor and interface as a whole: that delays and failure rates could not be defined in the code; and likewise, that the option to make nodes self-connected was also not able to be defined in the code. At the time of testing, these two aspects of the network were controlled through graphical elements in the user interface — by double-clicking on connections in the graphical topology representation and with a graphical switch respectively.

Nick, Gordon and Matt all said that the topology visualiser, which shows the user's network as a graph, was very useful for checking that what they had coded was what they had intended, although Matt said that every time changes are made to the topology or any connection parameters, the nodes in the graph all move to somewhere else, and this was a minor annoyance.

Simulating the network and exporting output data

All testers were able to simulate their network by running generated nodes from the simulation node manager tab, although there were a variety of ways that users tried to start the nodes. Amy initially tried to schedule events to start them, before realising that there were checkboxes next to each node which could be used to start them all together. Both Alan and Gordon did not realise that these checkboxes could be used and started all nodes individually using the start button of each one.

One issue that came up when setting up simulations was that on the first time a simulation was set up, the generation of program images took a long time. This was because *Diorama* needs to download the *Docker* image for the Python runtime (over 900MB) to create images for Python node programs, but it only needs to to this once, so all subsequent simulations would take significantly less time to set up.

They also didn't have any difficulties in exporting the output data from nodes to a spreadsheet - some elected to download a CSV file and open it in a spreadsheet software package, while others copied the data to their clipboard and pasted it into a new spreadsheet document. Matt remarked that this copy-to-clipboard feature was advantageous to him personally because it meant he could avoid unnecessarily downloading a single-use file which might have ended up staying in his *Downloads* folder for a long time, as many other files do.

Evaluation of testing methodology

Since the proof-of-concept only came about to be in a usable state quite late into the project, we unfortunately had much less time than desired to perform user testing. Had more time have been available, we would have wanted to test its use by having a group of similar students complete a real coursework exercise from our department's Distributed Algorithms course, with some using *Diorama* and the other others using some of the alternative applications and methods available, which we explored in Section 3. We would have then wanted to metrics such as time taken to complete certain tasks. We acknowledge that such an experiment would demanded significantly more of each volunteer's time, and would have required a much larger test group.

Though the task we set our test users was designed to cover as much as the envisioned user journeys and as many of the available features of *Diorama* as possible, there were some small parts which it missed, for example, independently setting up a *local* VM (we did not ask users to test this because restrictions on the university network add lots of complexity); using our pre-made virtual machine images; scheduling node action events and observing their effect on the network; and creating and uploading zip files.

We had a very small group of test users — five students or, in the case of Matt, recent students. Though the sample size was small, there was good diversity in these five volunteer participants in terms of skill and experience levels, from Nick, with relatively little programming experience, to Matt, by far the most experienced (he had about a year of industry experience, had completed a four-year Computer Science course at a high-quality institution and also programmed for many years before university). This meant that the test was somewhat representative of a range of students who would use *Diorama*.

Our test group did not include any individuals who represented lecturers who would use *Diorama*, like our persona, Leroy. However, we can reasonably assume that since most lecturers likely to teach about distributed algorithms would have more programming experience and skill, they would probably not have found *Diorama* any *more* difficult to use than our test group did.

Results and changes made and planned in light of testing feedback

Although our sample size of five people is small, albeit diverse, the user testing we carried out with them has provided us with useful feedback about the usability of *Diorama* in its current state and gave us a clear list of potential modifications to improve this aspect of the application. Ultimately, all users were generally able to perform the tasks we had set using *Diorama's* web user interface, and there were no major problems with using the provided functionality of the application.

We learned from Nick's testing experience that setting up a cloud virtual machine can be a challenging step for some students — particularly those who don't have extensive experience with cloud services. One of the key objectives of this project was to be simple and accessible to students studying distributed algorithms from different technical backgrounds, by mitigating the need to learn other technologies so they can focus on the algorithms themselves. We can mitigate this problem by creating step-by-step guides for users on how to create cloud virtual machines with different providers. Furthermore, using virtual machine images which we have created — both for cloud providers and hypervisors — is a way to install *Diorama* which is significantly more straightforward.

In the program editor, if the user selects a public git repository as their code source, we have seen that there can be ambiguity as to which URL to provide. This could be especially true in a situation where a lecturer has provided the code for a node program to their students by giving a link to a git repository's project homepage (this is effectively what we did in our testing instructions). We have now removed this ambiguity by adding some guide text which explains exactly what URL the user should enter.

User testing has shown that on the program editor page, the button which displays API documentation does not appear prominently enough, since users had trouble seeing it was there. We have therefore moved this button closer to the code input part of the page, and made it more visible by increasing the size and weight of the label font.

Positively, the API documentation we have provided for both writing node programs and defining the network topology was found to have been clear and detailed enough to be used for their purpose.

Our testers found our method of defining a network topology through code powerful and each eventually got to grips with it after spending time reading the documentation — they were able to use it fluently to define a network topology presented to them visually. Although this is satisfactory, we could make the creation of network topologies even easier by giving users a visual editor to define nodes and the connections between them.

```
single_nodes:
- ...
- ...

node_groups:
- ...
- ...

# Addition of self-connected nodes switch
all_nodes_self_connected: YES  # or NO (boolean value)

# Addition of success rates - if the user doesn’t include a connection here, it is given the default value of 100%
connection_success_rates:
- from: alice
  to: bob
  percentage_rate: 67
- ...
- ...

# Addition of random or fixed delays in ms - if the user doesn’t include a delay here, it is given the default of a fixed 0ms delay (that is, no delay)
connection_delays:
- from: alice
  to: bob
  type: fixed
  params:
    value: 1200 # a 1200ms delay for all messages between alice and bob
- from: bob
   to: charlie
   type: normal
   params:
     mean: 3000
     variance: 90000 # a normally-distributed delay, with mean 3000ms and variance 90000ms (or standard deviation 300ms)
- ...
```

Figure 6.2.: Suggested change to our network topology API.

Reflecting upon the constructive criticism that some testers raised, about connection delay and success rate parameters not being part of the code which defines the topology, we do accept that this seems somewhat illogical. These parameters are, in effect, part of the effective substance of a network topology. Likewise, we have a graphical switch which determines whether or not all nodes are connected to themselves. For both these aspects, it does seem to make more sense to have them set as part of the topology definition code. A further disadvantage of *not* having these in the code, is that it is not possible for users to simply share these parameters with other users, by sending the code. A user receiving this network topology definition code (for example, a student from their lecturer, through a coursework specification) would have to perform further actions on the graphical interface, in order to obtain the same network topology setup. Currently, since this is not the case, two users with the exact same node program code and the same network topology definition code could have simulated networks with vastly different behaviours, and thus vastly different results. We will therefore change our network topology API in future to include these parameters. We suggest a way of making this change in Figure 6.2. We may keep the ability to change these parameters graphically, since users may find it useful; however, should we do this, we will automatically make relevant changes to the code in the editor as changes are made graphically.

In response to observations made my testers, we will make changes to the graphic network topology visualiser. In exploratory testing, though not part of the set task, Alan discovered that the visualiser finds it difficult to display large networks with many connections. He found that a large fully-connected network with 30 nodes would cause the nodes in the visualiser to uncontrollably bounce around. Furthermore, Matt pointed out that when he changed connection delays and success rates, the nodes in the visualiser would change position, despite the shape of the topology having not changed. We will attempt to fix these two issues by changing the parameters we pass to the vis.js library, or otherwise, if this is unsuccessful, we will look into using a different library for visualising graphs. We could also allow users to selectively hide nodes from the visual topology viewer.

In the node manager tab of the simulation page, many test users did not independently and immediately see that it was possible to select all nodes by ticking a single checkbox. We will try to make this feature more obvious, perhaps by making small changes to the visual design, or by displaying a dismissible hint in a non-intrusive way.

6.2.2 Comparison to directly using *Docker*

For several assessed and unassessed exercises on a Distributed Algorithms course taken as part of our degree, Alan and I had previously used *Docker* for implementing and testing distributed algorithms, then analysing the results they produced when run. It involved downloading images, creating containers to run our own code and handling networking for these containers. The process involved spending a significant amount of time reading and understanding *Docker* documentation. We both found that using *Diorama* instead to perform similar albeit slightly simpler tasks, much simpler and easier than using *Docker*. It took much less time to go from no code or setup, to obtaining output from a simulated self-defined. This is one key advantage that *Diorama* presents.

As we mentioned in Section 3.4, another big advantage which *Diorama* has over using *Docker* directly, is its ability to define more complex network topology shapes. In a *Docker* *bridge* network, everything is, in effect, fully connected, since all containers (nodes) on a network can send and receive messages to and from every other container in it. If a user wanted to equivalently simulate complex (or even non-fully-connected) networks, they would have to, for example, implement this in their node programs by ignoring and selectively sending messages, or modify iptables rules for different containers.

Since *Diorama* is an application which users access through a graphical web user interface, they do not need to use the command line. The our graphical interface is designed to be (and has been proven by user testing to be) self-explanatory; it requires no documentation to use. In contrast, users using *Docker* directly must use the command line to generate images and containers, which requires learning commands through reading documentation, and using the command line. Though experienced programmers would likely have little problem with this, less experienced programmers, particularly those who have never used Docker before, would find this more challenging.

The main disadvantages of using *Diorama* over *Docker* directly stem from the fact that in simplifying things for users, users have much less technical freedom in aspects of creating their network. One example is in how we have defined the node program API. Our programming model is a synchronous one, designed around a blocking message receiver. If a user preferred, or needed to program in a more asynchronous way using event handlers, such as the one used by *JBotSim* (Section 3.2), they would need to adapt the interface we have provided, for example by using multithreading. Another limitation for users is that they are not able to use runtimes and languages which we have not implemented, whereas using *Docker* directly would allow them to easily use any runtime for which an image has been created on *Docker Hub*.

6.2.3 Conclusion

Notwithstanding some of the issues described, we conclude that *Diorama* is to a large extent, usable *and* user-friendly — users were generally able to use its functionality to perform desired tasks, and the web user interface appeared to be self-explanatory for the most part. We believe that based on this, our hypothetical personas would be able to use *Diorama* to follow their envisioned user journeys; however, we have identified lots of areas for improvement to enhance usability. This was certainly to be expected, as what we have created and tested is a proof-of-concept application, whose implementation prioritised a large amount of functionality over production-quality usability.

6.3 Portability

We desired to create a solution which easy to install and use on a wide range of platforms and so assess how well we have achieved this.

6.3.1 The web interface

Our approach to making *Diorama* compatible and accessible across different platforms and environments has been to rely on the fact that its interface exists as a web application, and the assumption that modern web browsers are available on all popular operating systems. *Diorama's* single strict system requirement for accessing and using its interface (other than, of course, an internet connection) is a web browser which supports common and extensively-used technologies, including Javascript 5 (ECMAScript 2009), HTML5, CSS and the WebSocket protocol. We conclude that we have successfully made this aspect of *Diorama* very portable; it can be used on any desktop, laptop or even mobile or tablet device[2] which has a modern web browser installed.

6.3.2 The server

This interface and all underlying back-end services, are served from a single virtual machine running *Ubuntu Linux* with sudo access. Currently, all users wishing to use *Diorama* must create one themselves. The two methods we have set out for users to access this is to create and run one locally using hypervisor software, or to create a cloud virtual machine instance with a cloud services provider.

The first of these methods requires a machine with an operating system which can run a hypervisor, as well as sufficient system resources (namely processing power, memory and disk space) to run a virtual machine on top of the host operating system. Since *Ubuntu Linux* is a free and open-source distribution of Linux, it is free and readily available to download from the web [15]. We consider *VirtualBox*, which exemplifies a free and open-source hypervisor. *VirtualBox* is available on *Microsoft Windows*, *Apple macOS* and many Linux distributions. Its stated requirements are "reasonably powerful x86 hardware" and as much memory as is required by the virtual machine's operating system plus that by the host operating system. Users also require the hard drive space required by the virtual machine, plus around 30MB for *VirtualBox* itself [57]. Our test installations of *Diorama* on *Ubuntu Server 18.04 LTS* have used around 3.5GB of disk space, and 8GB of memory has been sufficient. From these considerations, we conclude that this method would be

[2] We have tested to ensure that *Diorama* works on smartphones and tablets running recent versions of *Android* and *Apple iOS*.

accessible to practically all users using a reasonably powerful personal computers[3] (*Stack Overflow's* 2019 Developer Survey found that 99.9% of developers use one of *Microsoft Windows*, *Apple macOS* or a Linux distribution [56]).

The other method usually requires users to use a cloud service provider's web interface to create a virtual machine instance, which can be accessed through the internet. In this regard, the only requirement on a user's device is that it has a web browser installed, which can display and use this web interface. Some educational establishments may provide students with in-house cloud services through which virtual machines can be created[4]. However, we must also consider the fact that for public cloud service providers, there is a financial cost to creating and running a cloud virtual machine instance. Although the cost of this is relatively low, in the realms of £0.07 per hour[5], it could still be an obstacle to some students.

6.3.3 The installation tool

If a user is using one of the cloud providers or hypervisors we provide VM images for, they are able to set up and use *Diorama* without our installation tool. Our installation tool uses *Ansible*, which requires a Unix operating system and Python (Python comes pre-installed on most Linux distributions and *Apple macOS*). *Ansible* can also be run on *Windows 10* using the *Windows Subsystem for Linux*[6] [63]. The installation tool for *Diorama* can therefore be used on personal computers running Linux, *Apple macOS* and *Microsoft Windows 10*.

6.3.4 Conclusion

The way in which we have built and published *Diorama* has ensured that it can be used on personal computers running all widely-used operating systems. A user's operating system is therefore unlikely to prevent them from using *Diorama*.

However, notwithstanding our best efforts to mitigate potential issues for users through publishing virtual machine images and a cross-platform deployment tool, the fact that *Diorama* requires a virtual machine to install and run is less than ideal from the standpoint of portability. Ultimately, the bottom line is that users must either be able (and

[3] By "personal computer", we mean any laptop or desktop device.

[4] For example, the Department of Computing at Imperial College London provides its students with an IaaS private cloud service [43].

[5] We consider the on-demand pricing of three cloud virtual machine offerings, each with 2 vCPUs and 8GB memory and running Ubuntu Linux: *Microsoft Azure's* `B2MS` costs £0.0621 per hour [51]; *AWS EC2's* `t3a.large` costs £0.0752 per hour [5]; and *Google Compute Engine's* `n1-standard-2` costs US$0.095 per hour [35].

[6] Ansible does not officially support the *Windows Subsystem for Linux*, but it does work. We have also tested our deployment tool on the *Windows Subsystem for Linux*.

willing to pay) to create and run a cloud VM instance, or be using a machine which is capable of running a virtual machine. It is very much possible that there are a minority of prospective users who would not be able to do either of these, for example, if they have a low-specification laptop and do not have a debit or credit card to pay online for cloud services.

One particular way to eradicate this potential issue would have been to make *Diorama* a hosted service which can be set up and installed on one machine, and that multiple users can use remotely, like the online IDEs we mentioned in Section 3.6. In our context, a lecturer (who is much less likely to be unable to meet these constraints) could set this up and students could simply connect to and use *Diorama* on their web browsers through a web connection.

6.4 Extensibility

We identify several areas in which the proof-of-concept could be extended in future, in order to be made more useful for students and teachers, and evaluate how easy it would be to extend these.

6.4.1 Node program runtimes

We have structured our *diorama-server* repository so that all base node program files reside in the `base_node_files` directory as git submodules. In order to add support for a new runtime, we would need to write the base node program files for it, publish it as a public git repository, then simply include this repository in *diorama-server*`/base_node_files` as a submodule.

It is also quite simple to add a new runtime to our front-end. We need to add the runtime to the list of runtimes available when creating and modifying programs. We have structured our code so that this information, as well as all data associated with runtimes, for example icons, display names and default code is defined in one single place which acts as the source of truth for this - `src/components/Programs/constants.js`, which we outline in Figure 6.3.

We must also add documentation for new runtimes to show users how to write code in its language. All user documentation which can be accessed from the front-end is stored in the *diorama-docs* git repository, and each runtime's documentation is a Markdown file, with the runtime as the filename. Adding a new runtime's documentation simply involves creating and writing Markdown file in this repository and renaming it to the name of the runtime.

```
import pythonIcon from "./runtimeIconImages/python.png";
import ...

export const runtimes = ["python3"];

export const runtimeIcons = {
  python3: pythonIcon
};

export const runtimeLabels = {
  python3: "Python 3"
};

export const braceEditorModes = {
  python3: "python"
};

export const defaultCodeDataForRuntime = {
  python3: {
    code:
      "def main(peer_nids, my_nid, send, receive, storage):\n" +
      "    while True:\n" +
      "        message, sender_nid = receive()\n" +
      "        print(f’{message.decode(\"utf8\")} from {sender_nid}’)\n",
    dependencies: ""
  }
};
```

Figure 6.3.: An outline of program runtime constants in `src/components/Programs/constants.js` of *diorama-web-ui*.

Given these things, we conclude that it would be very straightforward to add a new node program runtime, and we should put what we have described in a contributing guide in future.

6.4.2 Network topology definition mechanisms

```
import yaml
import json

parsers: Dict[str, Callable] = {
  ’YAML’: lambda raw: yaml.load(raw, Loader=yaml.FullLoader),
  ’JSON’: json.loads
}

topology = parsers[language](raw)
```

Figure 6.4.: A code snippet showing how we parse raw network topology definitions.

We currently allow users to define the shape of their network topology in the JSON or YAML languages. We show in Figure 6.4 how we process the user’s raw code for the main back-end server. In order to add support for a new serialisation language, such as *property list* or *XML*, we need to add a parser for the language to the `parsers` dictionary shown on line 4. This parser needs to take in the raw code and return a Python dictionary object in the same schema as our current design. On the front-end, the changes needed to add support for a new language would be trivial, since it just sends to the back-end server the raw string which the user has inputted.

A limitation of our current design for processing raw network topology code is that to change our schema would be a more complex task. We would need to make changes to the methods which turn a parsed topology definition Python dictionary into a list of nodes, which are spread over around 20 helper methods, albeit all adjacent to each other in the same file (`network_topology.py` in the *diorama-server* repository).

Like for node programs, user documentation for defining network topologies is stored in the *diorama-docs* git repository as separate Markdown files for each serialisation language.

6.4.3 Front end localisation

As we explained in Section 5.6.4, our front-end is in already a state where it can be localised for different languages and locales. With our current design is very easy to add translations for new languages. All the existing files in the `translations/` directory of the *diorama-server* repository each default export an object which contains the translations of all phrases for their language, as we show in Figure 5.36. To translate Diorama into a new language, we need to duplicate one of these files, rename it to the *ISO 639-1* code of the language and replace all values of the object in the file with translations for each phrase.

To demonstrate this, the *Diorama* proof-of-concept application currently supports the languages, English (United Kingdom), English (United States) and German.

7 Conclusion

Implementing distributed algorithms and experimenting with them by modifying them and observing the effects of such changes is an important and effective way for students to gain a greater understanding of them. We have presented the design and implementation of a new application, *Diorama*, which not only facilitates this, but has it as its sole focus. *Diorama* minimises time wasted by users on things not directly related to distributed algorithms by being quick and easy to get up-and-running, and by avoiding configuration and features which are not directly relevant — instead — providing a rich and powerful platform on which students and teachers can create their own algorithm implementations, define their own network topology and run, analyse and demonstrate it.

We have shown through our evaluation that *Diorama*, even in its current proof-of-concept state, but certainly after a few changes and with the addition of more features, has excellent potential to be used for teaching and learning in the real world. We hope to develop the *Diorama* project further over the coming months and years to work towards achieving this, and invite members of the open source and computing education communities to contribute both code and ideas in order to progress it further.

7.1 Future work

We identify several ways in which we can move the *Diorama* project forward, from a functional rough-around-the-edges proof-of-concept implementation which has some deficiencies, towards a production-quality open-source application, ready for use by students and teachers.

An active collaborative open-source project

Currently, *Diorama* is technically open-source, in that all source code is freely available on *GitHub*. We can make it a well-maintained open-source project which accepts contributions from the community (pull requests on *GitHub*) by creating a contribution guide and managing the feature roadmap of *Diorama* using *GitHub* projects. This would likely help to improve *Diorama* with improved functionality and stability, making it more ideal for teaching and learning.

Additional node program runtimes

Our proof-of-concept application currently provides users the Python 3 runtime to use to program their node programs. In future, we want to make more runtimes available so that users are able to use a range of languages for this. This will involve creating a new base node program for each new runtime and publishing it as a new git repository. We have described how to do this in Section 6.4.

Live network topology editing

One major desired and planned feature which we did not implement in our proof-of-concept was live editing of a network topology while a user's simulation is running. This is a feature we want to implement in future. One way we could implement this would be to create an extra *Docker* container for each simulation — a *control* container. This control container would send commands to all node *Docker* containers, informing them of changes to the network topology, and then nodes would update their configuration accordingly, for example, by making changes to their `nid_connections` file.

Usability improvements

We could, with permission, collect and analyse data from users about how they use the web interface in an effort to improve its usability. One way we could do this is by using a user feedback tool. *Hotjar* is an example of such a tool, whose features include collecting heatmaps and recordings of user activity, such as scrolling, clicking and navigation [41].

We could also make additions to the user interface to make it more accessible to users. We currently support user-chosen colour schemes, but the two currently included colour schemes have been designed with a focus on aesthetic. To assist visually-impaired users, we could create a high-contrast colour scheme, following guidelines suggested by organisations which specialise in this area. We could also create a visual mode where text is made easier to read by, for example, making text bigger and increasing font weight.

A hosted service

In future, *Diorama* could be offered as a hosted service, either by us, or by educational institutions. The motivation behind this is that multiple users would simply log in to their respective user accounts on the *Diorama* service and use it as normal. A big advantage with this is that users would not need to perform any installation. It could also allow educators to access students' work or results to asses it or track progress, as well as remotely collaborate with them. To implement this, we could spin-up a virtual machine for a user on the server each time they log in. We may also need to centrally manage IP address spaces for node *Docker* containers in order to prevent clashes; we could do this by assigning each user a different address space their containers can use.

A

Example network topology code in JSON

```
{
  "single_nodes": [
    {
      "nid": "alice",
      "program": "prog"
    }
  ],
  "node_groups": [
    {
      "type": "ring",
      "number_nodes": 4,
      "program": "ring",
      "nid_prefix": "r",
      "connections": [
        {
          "from": "r0",
          "to": "alice"
        }
      ]
    },
    {
      "type": "star",
      "hub_program": "hub",
      "hub_nid": "hubert",
      "number_hosts": 3,
      "host_program": "host",
      "host_nid_prefix": "ho-",
      "host_nid_suffix": "-st",
      "host_nid_starting_number": 1,
      "host_nid_number_increment": 2,
      "connections": [
        {
          "from": "ho-1-st",
          "to": "alice"
        }
      ]
    },
    {
      "type": "tree",
      "number_levels": 3,
      "number_children": 2,
      "programs": [
        "tr_root",
        "tr_l1",
        "tr_l2"
      ],
      "nid_prefixes": [
        "root",
        "a",
        "b"
      ],
      "connections": [
        {
          "from": "root0",
          "to": "r2"
        }
      ]
    }
  ]
}
```

B

Installation guide

B.1 Pre-made images

Instructions can be found at `https://mauriceyap.github.io/diorama`.

B.2 For local VMs

For example, *VMWare Workstation Player* or *VirtualBox*

This is taken from the README at ***`https://github.com/mauriceyap/diorama-deploy`.***

B.2.1 Pre-requisites

Your real machine

You need Python (either 2.7 or 3.5+ are okay) installed on your Unix machine, as well as pip. Unless you're using Windows 10's Subsystem for Linux, Windows isn't supported, so sorry if that's what you're using. You can find out if you have these already installed by running `which python` and `which pip`.

Next, install *Ansible* by running `sudo pip install ansible`.

If you don't have permissions to run sudo, run it without, and then inside the `deploy.sh` file, replace `ansible-playbook` with the path to your ansible playbook executable (for example, `/.local/bin/ansible-playbook`).

Your virtual machine

Set up an Ubuntu VM - you could use VirtualBox or a cloud VM provider like Microsoft Azure. These scripts have been tested on *Ubuntu Server 18.04.2 LTS*.

Make a user account on your VM which has **passwordless sudo access**. There's a helpful article here if you're not sure how to do that: `https://www.cyberciti.biz/faq/linux-unix-running-sudo-command-without-a-password`.

B.2.2 Deploying

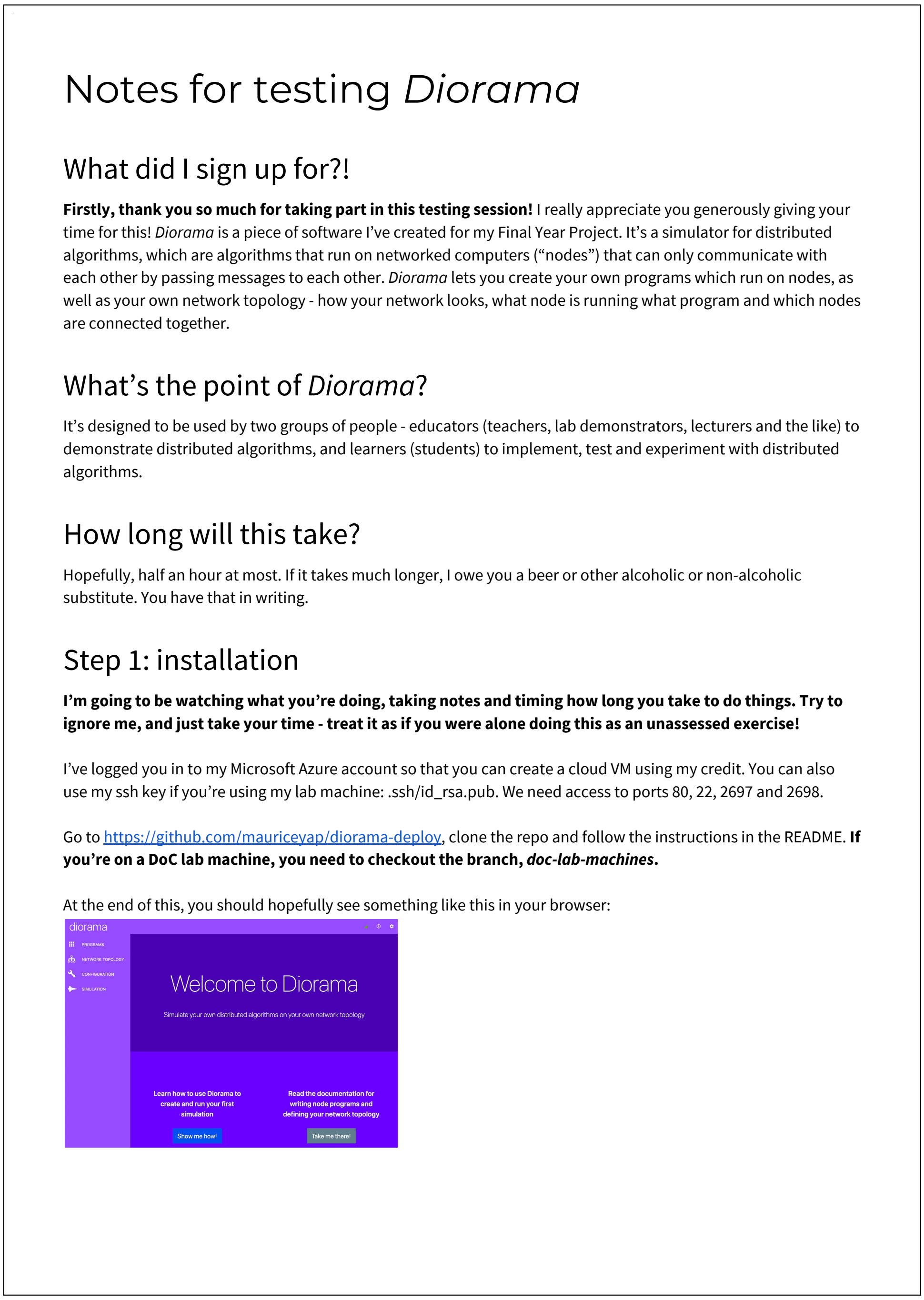
```
./deploy.sh IP_ADDRESS_OF_YOUR_VM USER_ACCOUNT_NAME
```

For example, `./deploy.sh 192.168.0.123 alice`.

Hopefully, that will run. After it's finished, you can open the web interface by visiting your VM's IP address in your browser.

C

Guide for test users



Notes for testing *Diorama*

What did I sign up for?!

Firstly, thank you so much for taking part in this testing session! I really appreciate you generously giving your time for this! *Diorama* is a piece of software I've created for my Final Year Project. It's a simulator for distributed algorithms, which are algorithms that run on networked computers ("nodes") that can only communicate with each other by passing messages to each other. *Diorama* lets you create your own programs which run on nodes, as well as your own network topology - how your network looks, what node is running what program and which nodes are connected together.

What's the point of *Diorama*?

It's designed to be used by two groups of people - educators (teachers, lab demonstrators, lecturers and the like) to demonstrate distributed algorithms, and learners (students) to implement, test and experiment with distributed algorithms.

How long will this take?

Hopefully, half an hour at most. If it takes much longer, I owe you a beer or other alcoholic or non-alcoholic substitute. You have that in writing.

Step 1: installation

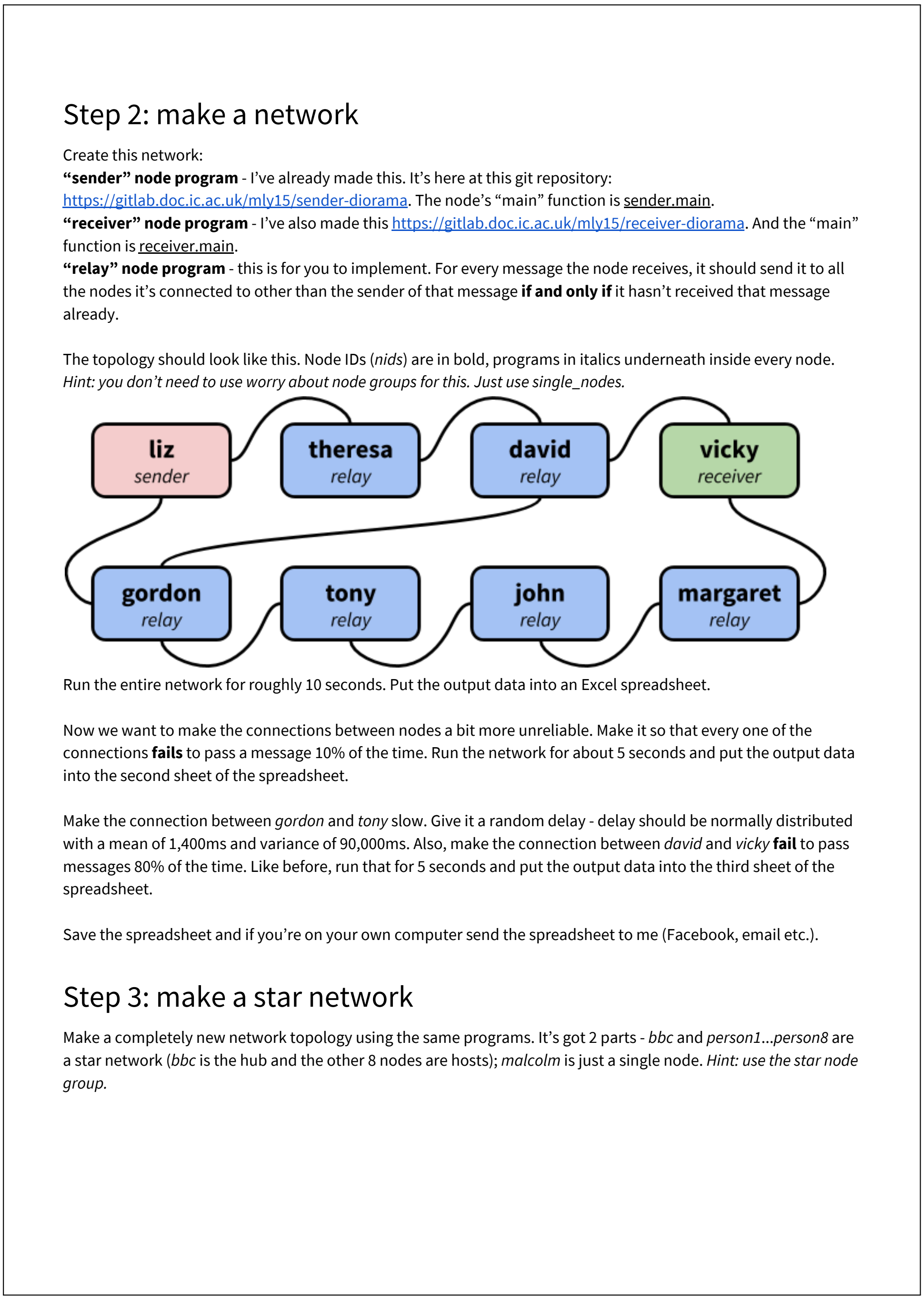
I'm going to be watching what you're doing, taking notes and timing how long you take to do things. Try to ignore me, and just take your time - treat it as if you were alone doing this as an unassessed exercise!

I've logged you in to my Microsoft Azure account so that you can create a cloud VM using my credit. You can also use my ssh key if you're using my lab machine: .ssh/id_rsa.pub. We need access to ports 80, 22, 2697 and 2698.

Go to https://github.com/mauriceyap/diorama-deploy, clone the repo and follow the instructions in the README. **If you're on a DoC lab machine, you need to checkout the branch, *doc-lab-machines*.**

At the end of this, you should hopefully see something like this in your browser:

diorama
PROGRAMS
NETWORK TOPOLOGY
CONFIGURATION
SIMULATION
Welcome to Diorama
Simulate your own distributed algorithms on your own network topology
Learn how to use Diorama to create and run your first simulation
Show me how!
Read the documentation for writing node programs and defining your network topology
Take me there!



Step 2: make a network

Create this network:
"sender" node program - I've already made this. It's here at this git repository: https://gitlab.doc.ic.ac.uk/mly15/sender-diorama. The node's "main" function is sender.main.
"receiver" node program - I've also made this https://gitlab.doc.ic.ac.uk/mly15/receiver-diorama. And the "main" function is receiver.main.
"relay" node program - this is for you to implement. For every message the node receives, it should send it to all the nodes it's connected to other than the sender of that message **if and only if** it hasn't received that message already.

The topology should look like this. Node IDs (*nids*) are in bold, programs in italics underneath inside every node. *Hint: you don't need to use worry about node groups for this. Just use single_nodes.*

Run the entire network for roughly 10 seconds. Put the output data into an Excel spreadsheet.

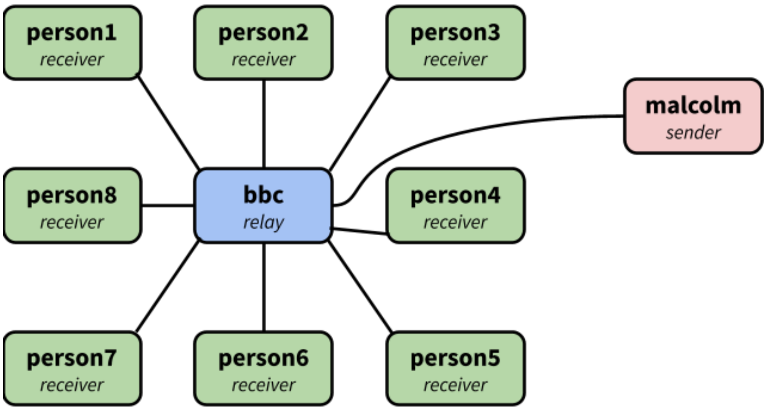
Now we want to make the connections between nodes a bit more unreliable. Make it so that every one of the connections **fails** to pass a message 10% of the time. Run the network for about 5 seconds and put the output data into the second sheet of the spreadsheet.

Make the connection between *gordon* and *tony* slow. Give it a random delay - delay should be normally distributed with a mean of 1,400ms and variance of 90,000ms. Also, make the connection between *david* and *vicky* **fail** to pass messages 80% of the time. Like before, run that for 5 seconds and put the output data into the third sheet of the spreadsheet.

Save the spreadsheet and if you're on your own computer send the spreadsheet to me (Facebook, email etc.).

Step 3: make a star network

Make a completely new network topology using the same programs. It's got 2 parts - *bbc* and *person1*...*person8* are a star network (*bbc* is the hub and the other 8 nodes are hosts); *malcolm* is just a single node. *Hint: use the star node group.*



Run all the nodes for 5 seconds.

Final remarks

Thank you thank you.

D

User testing node programs

sender.py

```
from time import sleep

def main(peer_nids, my_nid, send, receive, storage):
    counter = 0
    while True:
        for nid in peer_nids:
            message = f"Sender message {counter}".encode()
            send(message, nid)
        sleep(0.5)
        counter += 1
```

receiver.py

```
def main(peer_nids, my_nid, send, receive, storage):
    messages_received = set()
    while True:
        raw_message, nid = receive()
        message = raw_message.decode("utf8")
        if message not in messages_received:
            print(message)
            messages_received.add(message)
```

E User testing notes

For each of our volunteer test users, we give a raw transcription of the notes we took during testing, followed by verbal feedback that was given during testing or afterwards by the tester.

E.1 “Matt”

- 3:55 to install ansible and get set up with repo readme
- 2:35 to install Azure VM
- 4:14 to install everything onto the VM ansible
- 5:50 to create programs
- 7:43 to create network topology
- 1:50 simulation 1 - run and export
- 4:10 simulation 2 - edit connection parameters
- 3:05 simulation 3 - edit connection parameters
- 8:15 step 3. Took a long time to read API. Optional stuff added to text. Found example code and ASCII art illustrations in the documentation useful.

- *API documentation was good for making programs and for the topology. Bit less detailed than most stuff I’ve used, but there’s definitely enough information and is well-written.*
- *(About the programs page and editor) It was easy to navigate because the interface is nicely designed. The direct git import feature is really cool, it’s probably going to really useful for students, especially if they have group projects.*
- *(About the network topology editor page) A small annoyance but it’s really minor. The nodes in the visualisation for the topology move every time I make changes to it so it was hard to immediately see what I had changed. The interface was nice and simple and easy to use. I liked that there wasn’t anything unnecessary there. I thought it was weird not putting the connection editing stuff [delays and failure rates] in the code bit, but it’s probably my fault for missing the massive bold text that tells you to double click on the lines to change stuff.*
- *(About the simulation viewer) Layout is good. I really liked the "new changes have been made" badge in the "stop and reset" button. The checkboxes for selecting multiple or all nodes is useful. The copy to clipboard button on the output viewer tab*

is nice because it means I don't have to download a CSV file if I don't want to clutter my Downloads folder with yet another single-use file.

- *(Overall comments) It takes a while for the simulation to load. Overall I really like it and the interface looks really nice! It's really easy to use because it's obvious what to click to do stuff.*

E.2 “Nick"

- Never set up a cloud VM or used cloud services before, so found it difficult to set up a VM instance on Azure.
- 10:48 to create VM, set it up with Diorama and get it all running from starting to read README file.
- 12:28 to create all three programs. Had to correct him with which URL to use for the git repo. It's the .git one when you git clone, not the address to the web viewer for the GitLab repo.
- Had to correct algorithm - forgot about unique messages. I suggested using a set to store received messages.
- Had to help with writing Python. For-loop and if-statement syntax, API for set() object.
- 10:30 to create network topology. Had to help with YAML syntax - indentation especially. Had to remind him of two-way connections. He initially started by defining every connection both ways.
- 4:08 - run and export. Images took a long time to create.
- Didn't fully do tasks for editing connection parameters, but I made sure he was able to do it for one. Spent ages reading topology documentation looking for how to add delays and failure rates, but I had to point him in the right direction.

- *Setting up VM is difficult, you should specify system requirements for it. Installing Diorama is fine though.*
- *Program editor page, API guide accordion isn't immediately obvious.*
- *Put “all nodes connected to themselves” into the syntax of the topology.*
- *The graph viewer when you click save is useful to see what you've done.*
- *Like the programs interface, it's easy to use.*
- *Like the UI overall. It's generally easy to use.*

E.3 “Alan"

- Alan used a VM I had created with Diorama already installed. I asked him to have a quick read through of the installation instructions and he said it was good and he'd probably be able to follow it.
- Testing was very unfocused. He wanted to play with the application and explore as much as possible instead of focusing on following the tasks I had set. He was able to do everything eventually though and found everything impressive.
- Alan defined a very large network, with 30 fully-connected nodes. This caused the nodes in the graphic network topology viewer to spontaneously bounce around quickly and uncontrollably.

- *You shouldn't show the start simulation button when you don't have programs or a network yet.*
- *This is much better than what we did for the coursework [last year] with Docker. I spent lots of time on that making it work.*

E.4 “Gordon"

- Gordon was able to do all the tasks. For sender and receiver programs, he pasted the wrong link into the input box - gave URL of GitLab, not of the repository, ending in .git.
- Spent a lot of time reading the documentation for both the programs and network topology APIs. I had to give him some pointers about what to ignore (and was irrelevant to the tasks set) to save time.
- Simulation page - didn't realise you could select nodes and start all nodes at once.

- *It's pretty nice, I quite like it.*
- *Self-connected nodes thing [switch] isn't obvious enough.*
- *Interface and instructions were very clear.*
- *Strange that you can't add delays in the topology code.*
- *The network topology visualiser is useful.*

E.5 “Amy"

- Setup was quick and no problems. I had to prompt her to change port rules because she forgot about this.
- Spent quite a while reading everything before starting.

- Tried to copy and paste code from git repo into text editor (and use raw code as source). I let her do this for “sender” but pointed out the import from git function for “receiver”. Had to correct her implementation or “relay” because it originally didn’t check that message hadn’t already been received and also sent back to the sender of the original message each time message received.
- I coded the rest of the network topology to save time after she had done the first two nodes to prove she understood the API.
- On the simulation page, she initially tried to start all nodes by scheduling events. Took some time to find where to click for output tab.
- No problems adding delays and success rates to connections in the network. She immediately saw the bold text saying to double click connections to add these parameters.
- Took a long time to implement the star network (over 10 minutes) but eventually did it. Initially confused about the structure of the YAML file. She had started simply listing all the nodes and node groups at the top level, but I pointed out the first section of the API documentation about the overall structure of the YAML code.

- *Setting everything up was easy.*
- *The instructions are easy to follow.*
- *I like how easy it is to install. It’s nice that it’s just one script and you’re done.*
- *API notes are good and I could follow everything after spending a bit of time reading. It’s quite long but there’s nothing unnecessary there.*
- *The network topology API seems really powerful.*
- *Was this inspired by AWS Lambda?* ***(Yes!)***
- *The interface is pretty. I like the purple. But seriously, the design is good. It’s easy to use.*
- *The instructions [for performing user-testing] you’ve written are clear.*

Bibliography

[1] Dan Abramov. *Redux: A Predictable State Container for JS Apps*. 2019. URL: https://redux.js.org (visited on 13th June 2019).

[2] Agendaless Consulting. *Supervisor: A Process Control System*. 2019. URL: http://supervisord.org (visited on 14th June 2019).

[3] Airbnb, Inc. *Polyglot.js*. 2019. URL: http://airbnb.io/polyglot.js (visited on 14th June 2019).

[4] Almende B.V. *vis.js - A dynamic, browser based visualization library.* 2017. URL: https://visjs.org (visited on 14th June 2019).

[5] Amazon Web Services, Inc. *Amazon EC2 Pricing*. 2019. URL: https://aws.amazon.com/ec2/pricing/on-demand (visited on 8th June 2019).

[6] Amazon Web Services, Inc. *AWS Console: AWS Lambda Functions*. 2019. URL: https://eu-west-1.console.aws.amazon.com/lambda (visited on 10th June 2019).

[7] Amazon Web Services, Inc. *AWS Console: CloudWatch*. 2019. URL: https://eu-west-1.console.aws.amazon.com/cloudwatch (visited on 10th June 2019).

[8] Amazon Web Services, Inc. *AWS Lambda*. 2019. URL: https://aws.amazon.com/lambda (visited on 10th June 2019).

[9] Amazon Web Services, Inc. *AWS Lambda Documentation*. 2019. URL: https://docs.aws.amazon.com/lambda (visited on 10th June 2019).

[10] Amazon Web Services, Inc. *AWS Lambda Runtimes*. 2019. URL: https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtimes.html (visited on 10th June 2019).

[11] Amazon Web Services, Inc. *Custom AWS Lambda Runtimes*. 2019. URL: https://docs.aws.amazon.com/lambda/latest/dg/runtimes-custom.html (visited on 10th June 2019).

[12] Jim Armstrong. *Get to Know Docker Desktop*. 2018. URL: https://blog.docker.com/2018/09/get-to-know-docker-desktop (visited on 4th Feb. 2019).

[13] Bas Buursma and Ives van Hoorne. *CodeSandbox: Online Code Editor Tailored for Web Application Development*. 2019. URL: https://codesandbox.io (visited on 22nd Jan. 2019).

[14] Cambridge University Press. *Meaning of diorama in English*. 2019. URL: https://dictionary.cambridge.org/dictionary/english/diorama (visited on 3rd June 2019).

[15] Canonical Ltd. *Ubuntu: The leading operating system for PCs, IoT devices, servers and the cloud*. 2019. URL: https://www.ubuntu.com (visited on 8th June 2019).

[16] Arnaud Casteigts. "JBotSim: a Tool for Fast Prototyping of Distributed Algorithms in Dynamic Networks". In: *SIMUTools '15: Proceedings of the 8th International Conference on Simulation Tools and Techniques*. Ed. by Georgios Theodoropoulos, Gary Tan Soon Huat and Claudia Szabo. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). Brussels, Belgium: ICST, Aug. 2015.

[17] Arnaud Casteigts. *JBotSim Project documentation*. 2015. URL: https://jbotsim.io/javadoc/1.0.0/index.html?overview-summary.html (visited on 10th June 2019).

[18] Arnaud Casteigts. *The JBotSim Library*. 2015. URL: https://jbotsim.io (visited on 10th June 2019).

[19] Codeanywhere Inc. *Codeanywhere - Cross Platform Cloud IDE*. 2019. URL: https://codeanywhere.com (visited on 22nd Jan. 2019).

[20] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani et al. "Dynamo: Amazon's Highly Available Key-value Store". In: *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*. Ed. by Thomas C. Bressoud and Marinus Frans Kaashoek. ACM SIGOPS. New York, NY, USA: ACM, Oct. 2007, pp. 205 –220.

[21] Hoa Do. *Network Embedded Systems: Reliable Broadcast (et al.)* Slides for university lecture course. 2007. URL: https://ti.tuwien.ac.at/ecs/teaching/courses/nes/slides/hoa_do_reliablebc_final.pdf (visited on 8th May 2019).

[22] Docker Inc. *About images, containers, and storage drivers*. 2017. URL: https://docs.docker.com/v17.09/engine/userguide/storagedriver/imagesandcontainers (visited on 9th Feb. 2019).

[23] Docker Inc. *Develop with Docker*. 2019. URL: https://docs.docker.com/develop (visited on 9th Feb. 2019).

[24] Docker Inc. *Develop with Docker Engine SDKs and API*. 2019. URL: https://docs.docker.com/develop/sdk (visited on 3rd June 2019).

[25] Docker Inc. *Docker SDK for Python*. 2019. URL: https://docker-py.readthedocs.io/en/stable (visited on 12th June 2019).

[26] Docker Inc. *Networking overview*. 2019. URL: https://docs.docker.com/network (visited on 7th Feb. 2019).

[27] Docker Inc. *View logs for a container or service*. 2019. URL: https://docs.docker.com/config/containers/logging (visited on 9th Feb. 2019).

[28] Docker Inc. *What is a Container*. 2018. URL: https://www.docker.com/resources/what-container (visited on 25th Jan. 2019).

[29] Docker Inc. *Why Docker?* 2018. URL: https://www.docker.com/why-docker (visited on 24th Jan. 2019).

[30] Isaac Eldridge. *What Is Container Orchestration?* 2018. URL: https://blog.newrelic.com/engineering/container-orchestration-explained (visited on 4th Feb. 2019).

[31] Kayhan Erciyeş. *Distributed Graph Algorithms for Computer Networks*. London: Springer, 2013. ISBN: 978144715173. DOI: https://doi.org/10.1007/978-1-4471-5173-9.

[32] Johannes Ewald, Sean Larkin, Kees Kluskens and Tobias Koppers. *webpack*. 2019. URL: https://webpack.js.org (visited on 14th June 2019).

[33] Facebook Inc. *React — A JavaScript library for building user interfaces*. 2019. URL: https://reactjs.org (visited on 13th June 2019).

[34] Elliot Forbes. *UDP Client and Server Tutorial in Python*. 2017. URL: https://tutorialedge.net/python/udp-client-server-python (visited on 9th Feb. 2019).

[35] Google. *Google Compute Engine Pricing*. 2019. URL: https://cloud.google.com/compute/pricing (visited on 8th June 2019).

[36] Lukasz Guminski. *Orchestrate Containers for Development with Docker Compose*. 2018. URL: https://blog.codeship.com/orchestrate-containers-for-development-with-docker-compose (visited on 4th Feb. 2019).

[37] Howard Hamilton. *Distributed Algorithm*. Notes for university course: "Introduction to Operating Systems". 2007. URL: http://www2.cs.uregina.ca/~hamilton/courses/330/notes/distributed/distributed.html (visited on 23rd Jan. 2019).

[38] Oded Har-Tal. *A Simulator for Self-Stabilizing Distributed Algorithms*. Student's final-year BSc project. 2000. URL: https://www.cs.bgu.ac.il/~projects/projects/odedha/html/ (visited on 8th May 2019).

[39] Oded Har-Tal. *A Simulator for Self-Stabilizing Distributed Algorithms: Project Presentation*. Presentation slides for student's final-year BSc project. 2000. URL: https://www.cs.bgu.ac.il/~projects/projects/odedha/html/final.ppt (visited on 9th May 2019).

[40] Marijn Haverbeke. *CodeMirror*. 2019. URL: https://codemirror.net (visited on 14th June 2019).

[41] Hotjar Ltd. *Hotjar - Heatmaps, Visitor Recordings, Conversion Funnels, Form Analytics, Feedback Polls and Surveys in One Platform*. 2019. URL: https://www.hotjar.com (visited on 15th June 2019).

[42] James Hrisho. *React-Ace*. 2017. URL: http://securingsincity.github.io/react-ace (visited on 14th June 2019).

[43] Imperial College London DoC Computing Support Group. *DoC's Private IaaS Cloud Service*. 2019. URL: https://www.doc.ic.ac.uk/csg/services/cloud (visited on 8th June 2019).

[44] Matt Johnson. *Moment.js*. 2019. URL: https://momentjs.com (visited on 14th June 2019).

[45] Ajay D. Kshemkalyani and Mukesh Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, 2008.

[46] Insup Lee. *Introduction to Distributed Systems*. Slides for university lecture course. 2017. URL: https://www.cis.upenn.edu/~lee/07cis505/Lec/lec-ch1-DistSys-v4.pdf (visited on 4th Feb. 2019).

[47] Filipe Manco, Costin Lupu, Florian Schmidt et al. “My VM is Lighter (and Safer) than your Container”. In: *Proceedings of the Twenty-Sixth ACM Symposium on Operating Systems Principles*. Ed. by Haibo Chen, Lidong Zhou, Lorenzo Alvisi and Peter Chen. ACM SIGOPS. New York, NY, USA: ACM, Oct. 2017, pp. 218 –233.

[48] Partha Sarathi Mandal. *Distributed Algorithms*. Slides for university lecture course. 2016. URL: http://www.iitg.ac.in/gkd/aie/slide/Distributed%20Algorithm-PSM.pdf (visited on 23rd Jan. 2019).

[49] Ellis Michael. *Distributed Systems Labs and Framework*. Code repository for DSLabs project. 2018. URL: https://github.com/emichael/dslabs (visited on 10th May 2019).

[50] Ellis Michael, Doug Woos, Thomas Anderson, Michael D. Ernst and Zachary Tatlock. “Teaching Rigorous Distributed Systems With Efficient Model Checking”. In: *Proceedings of the Fourteenth EuroSys Conference 2019*. Ed. by George Candea, Robbert van Renesse and Christof Fetzer. ACM SIGOPS. New York, NY, USA: ACM, Mar. 2019.

[51] Microsoft. *Linux Virtual Machines Pricing*. 2019. URL: https://azure.microsoft.com/en-gb/pricing/details/virtual-machines/linux (visited on 8th June 2019).

[52] Microsoft. *Monaco Editor*. 2019. URL: https://microsoft.github.io/monaco-editor (visited on 14th June 2019).

[53] Alberto Montresor. *Distributed Algorithms: Reliable Broadcast*. Slides for university lecture course. 2016. URL: http://disi.unitn.it/~montreso/ds/handouts/04-rb.pdf (visited on 8th May 2019).

[54] Mozilla Corporation and Amazon Web Services, Inc. *Ace - The High Performance Code Editor For The Web*. 2019. URL: https://ace.c9.io (visited on 14th June 2019).

[55] Neoreason. *Repl.it*. 2018. URL: https://repl.it (visited on 22nd Jan. 2019).

[56] Oracle. *Stack Overflow Developer Survey Results 2019 (Developers’ Primary Operating Systems)*. 2019. URL: https://www.virtualbox.org/wiki/End-user_documentation (visited on 8th June 2019).

[57] Oracle. *VirtualBox: End-user documentation*. 2019. URL: https://www.virtualbox.org/wiki/End-user_documentation (visited on 8th June 2019).

[58] Severin Perez. *Writing Flexible Code with the Single Responsibility Principle: SOLID Principles and Maintainable Code*. 2018. URL: https://medium.com/@severinperez/writing-flexible-code-with-the-single-responsibility-principle-b71c4f3f883f (visited on 13th June 2019).

[59] Sergey Pimenov. *Metro UI CSS*. 2018. URL: https://metroui.org.ua (visited on 13th June 2019).

[60] Pusher Ltd. *Channels Protocol*. 2019. URL: https://pusher.com/docs/channels/library_auth_reference/pusher-websockets-protocol#events (visited on 10th June 2019).

[61] React Training. *React Router: Declarative Routing for React.js*. 2019. URL: https://reacttraining.com/react-router (visited on 14th June 2019).

[62] Red Hat, Inc. *Ansible is Simple IT Automation*. 2019. URL: https://www.ansible.com (visited on 14th June 2019).

[63] Red Hat, Inc. *Ansible: Windows Frequently Asked Questions*. 2019. URL: https://docs.ansible.com/ansible/latest/user_guide/windows_faq.html (visited on 8th June 2019).

[64] Guido van Rossum, Barry Warsaw and Nick Coghlan. *PEP 8 – Style Guide for Python Code*. 2001. URL: https://www.python.org/dev/peps/pep-0008 (visited on 30th May 2019).

[65] Markus Siemens. *TinyDB*. 2019. URL: https://tinydb.readthedocs.io/en/latest (visited on 10th June 2019).

[66] TETCOS. *NetSim Brochure*. 2017. URL: https://tetcos.com/downloads/NetSim_Brochure.pdf (visited on 24th Jan. 2019).

[67] TETCOS. *NetSim Standard*. 2018. URL: https://tetcos.com/netsim-std.html (visited on 25th Jan. 2019).

[68] The Tornado Authors. *Tornado Web Server*. 2019. URL: https://www.tornadoweb.org/en/stable (visited on 10th June 2019).

[69] Tiqa. *redux-polyglot*. 2018. URL: https://www.npmjs.com/package/redux-polyglot (visited on 14th June 2019).

[70] Travis CI, GmbH. *Travis CI - Test and Deploy Your Code with Confidence*. 2019. URL: http://travis-ci.org (visited on 14th June 2019).

[71] Michael Trier and Sebastian Thiel. *GitPython Documentation*. 2015. URL: https://gitpython.readthedocs.io/en/stable (visited on 12th June 2019).

[72] Doug Woos. *Oddity*. Code repository for Oddity project. 2019. URL: https://github.com/uwplse/oddity (visited on 10th June 2019).