CHALMERS |  UNIVERSITY OF GOTHENBURG

Visualizing Distributed Algorithms on the Seattle Platform

Master's thesis in Computer Science

JAKOB KALLIN

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Gothenburg, Sweden, June 2014

Visualizing Distributed Algorithms on the Seattle Platform

JAKOB KALLIN

Examiner: OLAF LANDSIEDEL

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden, June 2014

Abstract

This report describes Seastorm: a visualizer for distributed algorithms running on the Seattle platform. Seastorm displays the execution of algorithms as interactive sequence diagrams, intended to make reasoning about and debugging these algorithms easier. In order to do this, Seastorm augments the behavior of algorithms to also log events of interest, such as messages being sent and received. Most notably, this involves the addition of logical timestamps to messages, in order to avoid common problems related to event ordering in distributed systems. We designed Seastorm for students in courses on distributed systems and thus aimed to make its barrier to entry as low as possible: it requires very little installation, runs in the browser on any platform, and requires no manual modification of user algorithms. We performed small-scale qualitative testing to assess the value of Seastorm's visualization, with participants reporting an improved debugging experience compared to only using textual logging.

Acknowledgments

I would like to thank my supervisor Olaf Landsiedel, whose guidance and support confirmed that I made the right decision when asking him to be my supervisor before selecting a thesis topic. I would also like to thank Justin Cappos and the rest of the Seattle team, who provided me with invaluable help. Finally, I would like to thank my opponents Hongchao Liu, Lars Tidstam, and Xinwei Wang for their feedback.

Contents

Chapter 1

Introduction

1.1 Problem

A central challenge for computer scientists is learning and understanding the behavior of algorithms. One way of classifying algorithms is as either sequential, parallel, or distributed. Of these, a sequential algorithm is the easiest to understand, because it is a single sequence of actions taken in order. A parallel algorithm is more challenging to understand, because it involves several processes taking actions simultaneously. A distributed algorithm is even more difficult to understand, because it involves inter-process communication with fewer or no guarantees about timing and reliability.

Visualization can be an effective way of learning and understanding algorithms. In computer science textbooks, algorithms are often visualized as various forms of graphs that allow readers to see important behaviors in addition to reading about them. In a typical programmer's tool set, however, visualizations are rare. This is especially problematic for those working with distributed algorithms, which often cannot be used with traditional debuggers. A possible reason for the lack of visualizations in common developer tools is the complexity introduced by the fact that such tools must be able to visualize every type of algorithm—as well as every execution of those algorithms—and not just a single case as in a textbook.

There are some visualizers for distributed algorithms, both for simulators and for real environments, but many environments lack such tools completely. One example of an environment lacking visualizations is the Seattle platform for peer-to-peer computing [1], which is used for education and research related to distributed algorithms.

At present, users of the Seattle platform are limited to textual logging when inspecting the behavior of their algorithms. In such a scenario, they must manually cross-reference the data from different processes in order to determine which messages were sent, which were lost, and in what order it happened. This is a tedious and error-prone process, especially since no data is logged by default and each logging statement must be added manually.

1.2 Goal

The goal of this project was to improve education on distributed systems by implementing a visualizer for algorithms running on the Seattle platform. We named the resulting visualizer Seastorm.

We developed Seastorm with the following design goals:

- It should be targeted at students and educators making use of Seattle, such as those taking and giving the Distributed Systems course at Chalmers University of Technology (where Seattle is used).
- It should provide generic visualizations that can be used to understand a large number of algorithms, rather than specific visualizations that are only useful for a small number of algorithms.
- It should focus on relatively small algorithms that do not involve a large amount of data and activity, as such algorithms are often used in education. Examples of such algorithms include leader election algorithms (like the bully algorithm [2]). Examples of the opposite include long-running servers with complex behavior or many clients.
- It should have a low barrier to entry, which means that it should be platform-independent and that installation, configuration, and updating should be as simple as possible. It should also require little to no modification of existing programs in order to be able to visualize them. Finally, it should not require users to learn a large number of new concepts. This is important because students taking a single course in distributed systems may not have time for a lengthy setup and learning process.

We acknowledge the problem of software visualization systems being frequently constructed and then quickly abandoned without seeing much use (referred to by Hundhausen [3] as "system roulette"). As the design goals above suggest, we developed Seastorm to fill a specific niche rather than as a proof-of-concept or generic framework, increasing the chances that it will ultimately become useful.

1.3 Research questions

- How can visualization improve the understanding of distributed algorithms and make them easier to debug?
- How can monitoring of activity be added to programs running on the Seattle platform?
- How can the result of the project be made as accessible as possible for users who might benefit from it?

1.4 Outline

Chapter 2 provides background information for the rest of the report, including an overview of software visualization and a description of relevant aspects

of distributed systems. It also briefly describes the technologies used in the development of Seastorm.

Chapter 3 describes some prior projects related to Seastorm, either because they also focus on visualization or because they were developed for the same platform or environment.

Chapter 4 presents the design of Seastorm, without concern for implementation details. It describes the sequence diagram that Seastorm uses to visualize algorithms, including some ways in which it differs from other common sequence diagrams. It also gives the criteria that Seastorm uses to model algorithms as sequences of events, which allows them to be presented as sequence diagrams. Finally, it gives a brief overview of the Seastorm user interface.

Chapter 5 details how Seastorm was implemented using HTML5 and Python. It begins by describing the "heavy client" architecture that makes Seastorm as easy as possible to install and update yet still allows it to bypass some significant browser limitations. It then provides details on how Seastorm monitors the activity of Seattle programs without requiring user modification of source code. Finally, it outlines how Seastorm's sequence diagram was implemented in HTML5.

Chapter 6 presents the results of the project. It begins by giving an illustrated overview of Seastorm's main functionality. It then describes the small-scale qualitative testing of Seastorm that we performed, in which users reported a positive experience compared to having no visualization available. Finally, it describes the findings of a small performance test that we performed to identify possible issues with the speed of the user interface.

Chapter 7 summarizes the project and relates the final result to the design goals given above. It then suggests several ways in which the design, implementation, and testing of Seastorm could be improved in the future.

Chapter 2

Background

This chapter describes the most relevant background material needed for reading this report. It begins with an overview of software visualization and a discussion on its benefits. It then describes distributed algorithms and their properties, which need to be taken into account when designing development tools for them. It also gives an overview of the Seattle platform itself, focusing on the concepts that are important for Seastorm. Finally, it provides details on some parts of HTML5, which was used to implement much of Seastorm's functionality.

2.1 Software visualization

Software visualization is the act of presenting the behavior of software visually, in order to benefit learning or otherwise make the software easier to work with.

2.1.1 Benefits of software visualization

Intuitively, the benefits of visualization seem obvious, as evidenced by the widespread use of diagrams and charts in contemporary as well as historical teaching material. In modern development environments, however, visualization is exceedingly rare, possibly due to the difficulty of implementing dynamic and interactive visualizations compared to creating static visualizations for non-interactive media.

While critical of the methodology used by many developers of software visualization systems, Hundhausen [3] concludes that software visualization is effective in an educational context when the students interact with the visualization in some way. Merely seeing visualizations was not deemed useful, but making use of them in activities like programming exercises turned them into "catalysts for learning".

Wu [4] found significant benefits to teaching distributed algorithms with visualization as an aid. One of the drawbacks mentioned, however, is the potential difficulty of creating visualizations that map to the student's mental model of the algorithm. Rajala et al. [5] found some support for the benefit of visualization and solid evidence that it is especially beneficial for students with no prior programming experience.

We should note that the studies mentioned here evaluate vastly different types of visualizations, and directly applying their conclusions to Seastorm may not be relevant in all cases.

2.1.2 User adoption

Hundhausen [3] discusses a large body of research and concludes that, despite many software visualization systems having been developed, very few have seen widespread use. While the situation may have changed since Hundhausen's study, there are still very few, if any, well-known software visualization systems. In comparison, there are many notable development tools of other kinds, both old and new, with large audiences. Examples include Emacs, Vim, Eclipse, Visual Studio, Xcode, and Firebug.

An important issue highlighted by Hundhausen is the risk that visualizations themselves could contain errors, causing programs to be incorrectly visualized and possibly obscuring real errors in them. In general, this only applies to software visualization systems that require users to manually map program behavior to visualizations. The study found that programmers preferred textual debugging over such visualization systems, because they could then be confident that all errors were in their own programs rather than in the tools they were using. In other words, even useful visualizations could turn out to be rarely used in practice, as the barrier to entry and risk of complications is too high.

2.1.3 Types of visualizations

This section discusses some types of visualizations that are commonly used to visualize distributed systems and thus relevant to this project. The following visualizations will be outlined:

- Sequence diagram
- Communication diagram

2.1.3.1 Sequence diagram

The sequence diagram presents an algorithm as a set of vertical lifelines connected by arrows, with the vertical axis representing time, the lifelines representing different processes, and the arrows representing messages. A simple example of a sequence diagram as used in UML is given in figure 2.1. A more detailed description of the sequence diagram as used in UML is given by Bell [6].

Because it describes algorithms on the level of messages, the sequence diagram can be used for visualizing practically any distributed algorithm. Another significant benefit of the sequence diagram is that all available data can be visualized in a single image, as opposed to visualizations that present algorithms as a sequence of images. The sequence diagram is also pervasive in teaching material for distributed systems and thus already familiar to many students.

A drawback of the sequence diagram is that messages can become difficult to distinguish when the number of processes and messages increases. The same problem was identified by Nessa [7]. Another drawback of the sequence diagram is the lack of a spatial dimension: it does not present the distances and routes between processes.

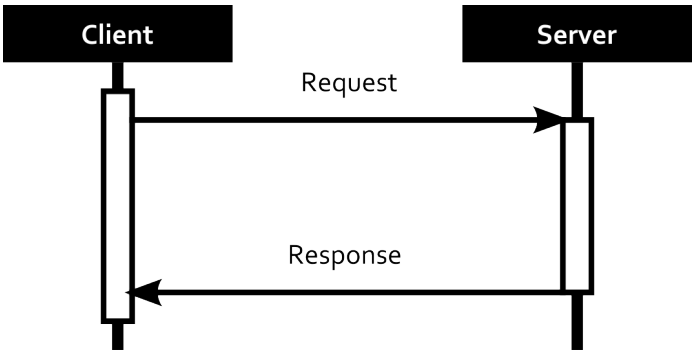

Figure 2.1: A sequence diagram

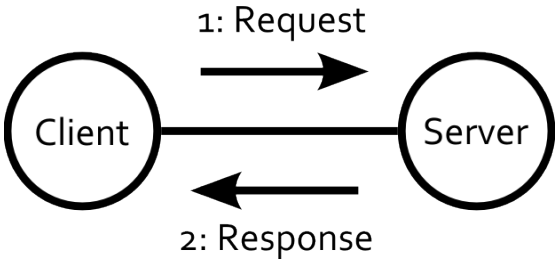

Figure 2.2: A communication diagram

2.1.3.2 Communication diagram

A communication diagram (used in UML and based on the older collaboration diagram) displays the same information as a sequence diagram but has no separate axis representing time. Instead, processes are placed in arbitrary positions, and messages passed between two processes are represented by arrow drawn between the processes, with one label for every message sent. The order of messages is presented using numbers next to the messages. A simple example of a communication diagram is given in figure 2.2. A more detailed description of the collaboration diagram (the predecessor of the communication diagram) is given by Karlsen [8].

A benefit of the communication diagram is that processes can be placed in any formation, potentially visualizing the topology of the network that the algorithm runs on. Like the sequence diagram, it is also general enough to be suitable for a large number of distributed algorithms.

A drawback of the communication diagram is that its presentation of ordering is relatively subtle: determining the order of messages requires comparing numbers rather than perceiving it graphically. A solution to this could be to instead use multiple network topology diagrams, with each individual diagram displaying only some of the messages. This approach, however, could make the overall behavior of the algorithm more difficult to understand, as information would be spread out over many diagrams.

2.2 Distributed algorithms

A distributed algorithm is an algorithm designed to run on multiple hosts in a network, with hosts communicating by passing messages to each other. These algorithms are generally subject to problems not encountered by sequential algorithms running on a single host, including [2]:

- Absence of a globally synchronized clock, which means that ordering of events cannot be determined simply from comparing physical timestamps.
- Unreliable means of communication, which means that events such as network failure or congestion can cause messages to be delayed or lost.
- Possibility of process failure, which means that processes have to take into account the possibility of other processes going down temporarily or permanently.

The exact nature of these problems depends on the specific network that the algorithm is running on. For the purposes of this report, algorithms are assumed to run on the Internet, which adds the additional complication that the author of the algorithm cannot control all hosts and network links involved.

2.2.1 Modeling distributed algorithms

At a conceptual level, the most general abstraction for representing activity in a distributed algorithm is a sequence of events [9]. This abstraction can also be used to describe a sequential algorithm (with each executed statement being considered an event), but it is more common in distributed algorithms, where it forms the basis for tracking time and causality.

In a distributed algorithm, there are two main types of events to consider:

1. events representing computations within a single process (internal events)
2. events representing communication between the processes (messaging events), which can be further divided into:
   (a) events representing the sending of a message (send events)
   (b) events representing the receipt of a message (receive events)

The reason for making this distinction is that the messages sent from a process make up the public interface of that process, whereas the internal events of the process are generally considered private implementation details.

2.2.2 Time and event ordering

Due to well-known limitations inherent in distributed systems, such systems commonly do not have globally synchronized clocks. This causes notable problems for systems that base their event ordering on the physical clocks of the different processes in the system. While protocols such as NTP employ algorithms to improve synchronization using approximations, inevitable clock drift makes it impossible to guarantee global synchronization.

We illustrate the problems caused by relying on physical clock synchronization with a minimal example program: a process A sending a single message to

another process B. If A reports the message departure time based on its physical clock and B reports the message arrival time based on its physical clock, the message can be reported as having been sent from the future to the past if the B's clock is sufficiently behind A's clock.

2.2.2.1 The happened-before relation

The rules governing event ordering are formalized by the happened-before relation [10], which holds for two events a and b if any of the following is true:

- a and b take place in the same process and a takes place before b. This rule might seem obvious, but it needs to be highlighted because only within a single process can such observations be made trivially.
- a is a send event and b is the corresponding receive event. This rule enforces the fact that a message must always be sent before it is received, regardless of whatever the physical timestamps might indicate.

In addition, the happened-before relation is transitive.

In order to avoid violations of the happened-before relation, an alternative to physical timestamps is required for ordering events. A common approach to that is described below.

2.2.2.2 Logical clocks

The first step towards improving event ordering is to replace physical clocks with logical clocks [10]. A logical clock is a counter that generates strictly increasing values with every event that occurs, allowing us to determine the order of events. Unlike a physical clock, the value of a logical clock is abstract and only intended to convey ordering. As an example, an event occurring at logical clock value 1 could have taken place one second earlier than event occurring at logical clock value 2, or it could have occurred five minutes earlier; with logical clocks, the distinction is irrelevant.

2.2.2.3 Lamport timestamps

With logical clocks implemented, the second step towards improving event ordering is to provide a method of synchronizing the logical clocks of different processes, so that one clock's value is relevant even when comparing it to that of a clock in another process. One of the simplest methods of doing this is the Lamport timestamp algorithm [10]. This algorithm synchronizes the logical clocks of two processes whenever a message is exchanged between them, by performing the following steps:

1. When a process sends a message, it includes its logical clock value in the message.
2. When a process receives a message, it sets its logical clock value to be greater than both its current value and the value included in the message. An example way of doing this is to set the logical clock value to the maximum of the two values and then increase it by one.

Lamport timestamps ensure that a message will always be reported as having been received after being sent, since every received message sets a process's clock to greater than the maximum of the two clocks involved.

2.2.2.4 Vector clocks

Vector clocks [2] are an alternative to Lamport timestamps based on each process having its own logical clock and maintaining local copies of the clocks in other processes. As vector clocks are not used by Seastorm and only briefly appear in our discussion on future work, a more detailed description of them is beyond the scope of this report.

2.2.2.5 Concurrent events

Lamport timestamps allow us to establish a happened-before relation between a send event and its corresponding receive event, but it does not give us guarantees about the ordering of subsequent events in the two processes where the send and receive events took place. In such a scenario, we consider the events to be concurrent.

For instance, if five events subsequently take place in each of the two processes, with no further synchronization, then we cannot determine the exact order in which these ten events took place (apart from knowing that events within a single process are ordered according to their timestamps). In this case, each of the five events in the first process could have taken place before each of the five events in the second process. The opposite could also be true, or any other interleaving.

It should be noted that, given accurate knowledge of the physical timestamp of every event in the system, we would be able to accurately order events that we would otherwise have thought to be concurrent. Because we do not have such knowledge, however, event concurrency is the most reliable assumption that we can make.

2.2.2.6 Inconsistent cuts

An inconsistent cut [11] is an invalid view of the state of a distributed system. Such a cut is obtained when a monitor process requests the state of each individual process in the system without taking precautions to ensure that each process records its state at roughly the same time.

A common property of an inconsistent cut is that one process reports to have received a message from a process that does not report to have sent it. This can in turn lead to incorrect analysis of what has actually happened in the system.

There are methods for avoiding inconsistent cuts (instead obtaining a consistent cut), but descriptions of those are beyond the scope of this report.

2.2.2.7 Causality violations

A causality violation [12] is when a process in a distributed system incorrectly orders events for which the happened-before relation holds. This can occur when messages from one process are received by another process in a different order than they were sent. The effect of a causality violation is that a message

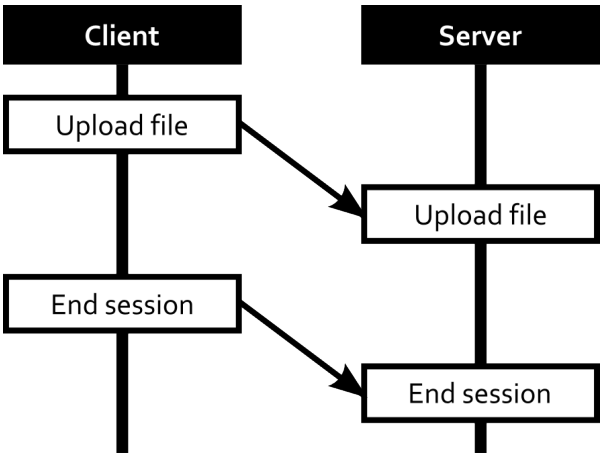

Figure 2.3: Client–server interaction with intended behavior

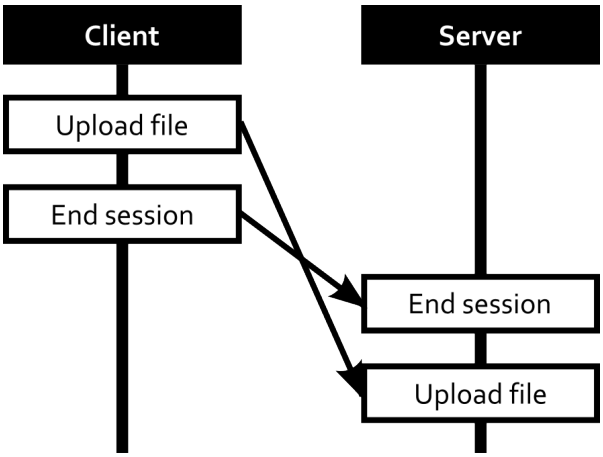

Figure 2.4: Client–server interaction with causality violation

b, whose existence may have been caused by the sending of a message a that was sent before b, is received by another process before a, potentially causing unintended behavior.

As an example, consider a client uploading a file to a file server and then ending its session with the file server. Figure 2.3 depicts the expected scenario, in which the file is uploaded first and the session is ended last. Figure 2.4 depicts a potential scenario that contains a causality violation: the session is ended before the file is uploaded (which in practice would prevent the file from being uploaded at all). The causality violation is visually indicated by the crossing arrows.

Causality violations can be avoided by delaying delivery of messages until all messages sent earlier have arrived, and then reordering them to remove all causal violations. Such a strategy thus enforces a logical ordering of messages rather than ordering them according to physical delivery times. Many common networking APIs, including those used in Seattle, do not prevent causality vio-

lations. In such APIs, causality violations must be prevented on the application level.

2.3 Seattle

Seattle is a platform for educational cloud computing that allows users to run distributed algorithms on a large collection of computers on the Internet.

2.3.1 Architecture

A computer participating in the Seattle platform does so by donating resources, which means providing other users with *vessels*. Vessels are virtual machines used for running *experiments*. An experiment is a set of individual programs running simultaneously on multiple vessels, acting as a distributed system. A Seattle user can acquire vessels on different computers in order to run experiments on them.

2.3.1.1 Vessel IDs

For the purposes of this report, a vessel is uniquely identified by the combination of its IP address and the port number assigned by the Seattle Clearinghouse (described in section 2.3.4) to its user.

Each vessel also has a name, which is a string used to distinguish vessels on the same node from each other. Seastorm assumes the use of the Seattle Clearinghouse to acquire vessels, however, and thus does not include the vessel name in the ID. This is because it is not possible for a user to acquire multiple vessels on the same node through the Seattle Clearinghouse.

2.3.2 Repy

Programs running on Seattle are written in Repy [13], which is a subset of the Python language. Repy denies access to much of the built-in functionality of Python and its libraries but provides replacements to some of these through the Repy API, including support for UDP and TCP communication.

There are two versions of Repy, with breaking API changes between them. Seastorm supports only Repy version 2, which is also the one described below.

2.3.2.1 Modules

Repy does not support Python's import statement. Instead, modules are included in a program using the dylink module [14], which dynamically includes the module at runtime.

2.3.2.2 UDP communication

Repy provides two functions for communicating over UDP:

- `sendmessage` sends a single message.

- `listenformessage` sets up a server socket that listens for messages. The server socket object in turn contains a `getmessage` method, which returns a single message if one is available (in a non-blocking fashion), as well as the IP address and port number that the message was sent from.

2.3.2.3 TCP communication

Repy provides two functions for communicating over TCP:

- `openconnection` attempts to open a connection and returns a connection socket if the connection is accepted before a specified timeout.
- `listenforconnection` sets up a server socket that listens for connections. The server socket object in turn contains a `getconnection` method, which returns a single connection socket if one is available (in a non-blocking fashion).

Connection sockets take the form of objects with three methods: `send`, `recv`, and `close`. The behavior of these is analogous to non-blocking TCP connection sockets in Python, which are in turned based on Berkeley sockets.

2.3.3 Node manager

A computer donating resources to Seattle runs a *node manager*, which manages the vessels available on that computer. For each vessel, the node manager keeps track of which user is allowed to control the vessel and handles commands issued by the user to control the vessel.

The Node Manager provides an API [15] based on a custom communications protocol on top of TCP that implements remote procedure calls. The supported procedures comprise public procedures, which can be called by anyone, and private procedures, which can only be called by the owner and possibly the users of a vessel. For private procedures, authentication and encryption are handled by means of public-key cryptography. Keys are provided to a user through the Seattle Clearinghouse website, described below.

2.3.4 Clearinghouse

The Seattle Clearinghouse [16] is a website that enables Seattle users to acquire vessels. The number of vessels that a user is allowed to acquire is based on the number of resources that they have donated to Seattle, or that others have donated on their behalf, but all users can acquire up to 10 vessels without donating. Each Clearinghouse user is assigned a port number that determines which port number the user's programs are allowed to use for communication on a vessel.

The Clearinghouse provides an API implemented using XML-RPC (briefly described in section 2.4.5). Authentication is handled by means of an API key, which is provided to a user through the Seattle Clearinghouse website. Encryption is provided by HTTPS.

2.4 HTML5

This section describes some relevant parts of HTML5 that are not central components and thus less well-known. A description of core technologies like HTML, CSS, and JavaScript is beyond the scope of this report.

In this report, we use the term HTML5 to refer to the collection of technologies used in modern web development and standardized by the W3C. The official term is Open Web Platform; HTML5 is colloquially used to more concisely describe the same concept.

2.4.1 Browser restrictions and limitations

At this point in time, browser applications are subject to some notable restrictions and limitations.

- Security restrictions: Due to the same-origin policy, an HTML5 application cannot communicate with another host unless that host explicitly allows the communication with CORS. It also cannot access the local filesystem except in very restricted ways, such as a user explicitly selecting files with a file selector controlled by the browser.
- Network restrictions: Even within the same origin or with CORS enabled, HTML5 applications cannot communicate with other hosts using arbitrary protocols; only a limited subset is allowed, most notably including HTTP.
- Cryptography limitations: HTML5 applications do not have access to the primitives needed to reliably perform cryptographic operations.

2.4.2 The same-origin policy

In order to increase security, browsers restrict webpage communication using the same-origin policy, which dictates that pages may only communicate with other pages if they have the same *origin*. An origin is defined by Mozilla as follows: "Two pages have the same origin if the protocol, port (if one is specified), and host are the same for both pages." [17]

Cross-Origin Resource Sharing (CORS) is a protocol that allows webpages to bypass the same-origin policy. Browsers with support for CORS allow webpages to access resources on servers of other origins that explicitly allow it. These permissions are implemented as headers sent in HTTP responses from the server to the webpage trying to access a resource.

Before a webpage in a browser with CORS support can access a resource from another origin, the browser checks the CORS headers sent by the server. If the webpage should be allowed access, the response is transparently delivered to the webpage. If the webpage should not be allowed access, the response is withheld and an error is raised.

We should emphasize that the default behavior of browsers in absence of CORS headers is to disallow cross-origin communication. This means that services designed without CORS in mind cannot be accessed by webpages of other origins.

We should also note that there is an older, widely used alternative to CORS for enabling cross-origin communication: JSONP. The JSONP technique has

the same limitations as CORS, however, in that it must be explicitly supported by the server in question.

2.4.3 Graphics

HTML5 offers three main technologies for drawing graphics:

- Canvas [18], which is a JavaScript API for drawing bitmap graphics on part of an HTML page.
- HTML/CSS, which are designed primarily for styling documents, although support for layout has been present in some form for many years and improved in recent years.
- SVG [19], which is an XML-based image format for creating scalable vector graphics that can be embedded into an HTML page.

While Canvas offers a great degree of freedom, it is complex (because it does not provide many abstractions) and does not allow for lossless scaling (because it draws bitmap graphics) compared to HTML/CSS and SVG.

HTML/CSS provides many layout facilities, but very few ways of drawing graphics. Conversely, SVG provides many facilities for drawing graphics, but very few for handling layout. For instance, HTML/CSS can create rectangles whose dimensions are determined by their contents but not draw arrows between two rectangles, while SVG can draw arrows between two rectangles but not have them automatically adapt to their contents. Either approach imposes limitations: HTML/CSS restricts graphics to essentially a grid of boxes, while SVG requires significant amounts of code in order to recreate the fluidity of HTML/CSS.

2.4.4 JSON

JSON [20] is a text-based data format, especially common on the web, used for serializing data and transferring data between different applications. Its data model includes several types of data common to many programming languages, including strings, numbers, boolean values, lists, and dictionaries.

2.4.5 XML-RPC

XML-RPC [21] is a protocol for performing remote procedure calls over HTTP. It uses a small subset of HTTP to achieve this: only the POST method is used, and only the 200 status code is used (unless a lower-level error occurred). Arguments, return values, and error messages are sent as part of the request body in XML format.

2.4.6 Server-sent events

Server-sent events are a web technology that allows servers to push data to web pages, in contrast with the traditional model of web pages requesting data from servers.

On the server, events are created by incrementally writing output to the client, using special character sequences to delimit events.

On the client, events are received by opening a special type of connection to the server. This connection triggers events as they arrive and allows for listening to events in the same way as with other JavaScript events.

Chapter 3

Related work

This chapter gives an overview of prior work related to Seastorm. It begins by describing a few existing projects that visualize distributed systems and highlighting notable ways in which they differ from Seastorm. It then briefly covers related projects that, like Seastorm, were developed for the browser. Finally, it describes existing tools used for running experiments on the Seattle platform.

3.1 Distributed system visualization

3.1.1 Cooja TimeLine

Cooja is a network simulator for Contiki, which is an operating system for the Internet of Things. Cooja includes the module TimeLine [22], which visualizes the behavior of wireless sensor networks. The visualization takes the form of a set of timelines, with each separate timeline showing the power state of a node as well as its transmissions and receptions.

As it focuses on wireless sensor networks, Cooja TimeLine needs to consider some low-level concepts that are not present in Seattle, such as packet collisions and power consumption. In fact, Cooja TimeLine focuses on these concepts, unlike Seastorm, which focuses on the messages being sent and their contents.

3.1.2 Ericsson EJBActorFrame trace monitor

Ericsson EJBActorFrame is a framework for developing services based on distributed state machines. Nessa [7] developed a visualizer that presents the behavior of such services as sequence diagrams. One of the problems identified in the analysis of the system was the difficulty of understanding these diagrams in systems with a large number of messages or processes. This was improved to some degree by providing filtering capabilities. The potential for improving visualization of such systems with the use of other diagrams was also proposed (and later implemented [8]).

This project is very similar to Seastorm but contains a couple of notable differences. First of all, it is not compatible with Seattle, although it has mechanisms for adding support for other environments. Secondly, its installation

process is much more involved, requiring the installation of Eclipse and two Eclipse plugins, among other things.

3.1.3 LYDIAN

Lydian [23] is an educational environment for distributed algorithms that provides multiple ways of visualizing algorithms. It uses a simulator to execute the algorithms, which means that the user is also in control of factors like network topology and timing. Algorithms are written using a C-like language.

LYDIAN was developed with the same goals as Seastorm and with the same target audience but offers a more comprehensive set of visualizations. On the other hand, LYDIAN uses a simulator and thus does not have explicit support for Seattle (although execution data from other systems could be compiled into the trace file format that LYDIAN uses). Additionally, the installation process is more involved than Seastorm's, at least when used on Windows and possibly other platforms.

3.2 Browser projects

3.2.1 D3.js

D3.js is a general-purpose visualization library written in JavaScript that is used to visualize data as SVG images. It has been used for many different visualizations, both interactive and non-interactive. Although the library allows for creating completely custom visualizations, it also provides layout algorithms for common diagrams, such as pie charts and trees.

In contrast to the general nature of D3.js, Seastorm specifically visualizes the execution of distributed algorithms. Although we considered using D3.js for Seastorm, the visualizations that we ultimately decided to implement were simple enough that we did not need the generality of D3.js and could implement a custom solution fairly quickly.

3.2.2 Cloud9 IDE

Cloud9 IDE [24] is a browser-based integrated development environment. It offers features common to many other development environments but also provides support for collaboration and online storage of projects.

Seastorm has much less sophisticated support for editing files and managing projects than Cloud9, and generally assumes that users perform these tasks in other applications. The focus of Seastorm is instead to provide visualization, which Cloud9 does not.

3.3 Seattle-related work

3.3.1 Seash

Seash (Seattle Shell) [25] is a terminal-based experiment manager for Seattle. It is the only experiment manager included in the standard distribution of Seattle (the *demokit*) and thus the most notable one. Seash allows the user to control

and inspect vessels. In addition to offering control over individual vessels, Seash can be used to create groups of vessels. Batch operations, such as running a file with a list of arguments, can then be performed simultaneously on all vessels in a group.

Seash runs entirely on the command line and performs no visualization or automatic logging, unlike Seastorm. It is not a subset of Seastorm, however, as it does allow for more fine-grained control over vessels.

3.3.2 Try Repy

Try Repy [26] is a web-based experiment manager for Seattle. Its aim is to provide a simple environment where users can try writing and running Repy programs without having to spend time learning to use more advanced experiment managers like Seash. Try Repy uses a client–server model: the client provides the user interface while the server runs the user's experiments. Unlike Seash, Try Repy only runs experiments on the server, rather than on nodes participating in Seattle.

Both Try Repy and Seastorm run in the browser and make use of a server. The installation process is similar for both, but Try Repy supports running the server on a different host, potentially sparing the user from doing any installation at all. Unlike Seastorm, Try Repy performs no visualization.

Chapter 4

Design

This chapter describes the design of Seastorm, starting with a conceptual architecture that describes on a high level how Seastorm creates visualizations from Seattle experiments. It then discusses which visualizations we decided to implement in Seastorm and on what grounds we made the decision. After this, it goes into detail about Seastorm's definition of events in a Seattle experiment, and further describes how these events are monitored. The chapter ends with an overview of how Seastorm's user interface exposes this functionality to the user.

4.1 Conceptual architecture

This section describes the architecture of Seattle on a conceptual level, free of concerns about platform and other implementation details. The technical architecture is described in section 5.1.

The two main tasks that Seastorm performs are gathering data from experiments running on Seattle and interactively visualizing this data for the user. On a conceptual level, the architecture of Seastorm is based on this division and thus consists of two core components: the monitor and the visualizer.

To achieve as much decoupling between these components as possible, Seastorm compiles all execution data into trace files using an intermediate data format that can be used for visualization with a small amount of preprocessing. This data format is the only link between the two components, and is free of implementation details from the system that is being monitored as well as the system that is doing the visualization. The result is that changes to the monitor do not affect the visualizer, and vice versa; only changes to the data format itself can force changes to both components. This architecture is illustrated in figure 4.1.

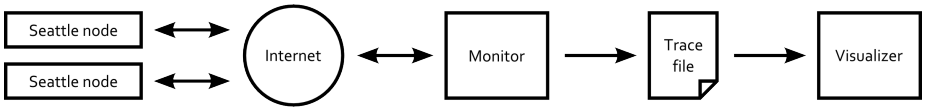

Figure 4.1: Conceptual architecture of Seastorm

Decoupling monitoring from visualization by using an intermediate data format has the following benefits, in descending order of importance:

1. **Modularity:** Insulating each part from changes in the other part makes development simpler, because each part has only one responsibility that is affected by outside factors. For the monitor, this is to create a data file with the correct format. For the visualizer, it is to correctly read a data file in this format.

2. **Flexibility:** Using an intermediate data format makes execution data modifiable and reproducible. This is useful, for instance, when synthesizing data, such as a teacher providing students with a reference file showing the intended behavior of an algorithm.

3. **Extensibility:** Similar to the second point, using an intermediate data format allows any system—not just Seattle—to produce trace files that can then be visualized with the same visualizer. For instance, compatible data could be generated from JavaScript to visualize a system of web workers [27] or from Erlang [28] to visualize a system of Erlang nodes. Although this extensibility is not a goal of the project itself, it is an incidental benefit of the architecture described.

4.2 Visualization

4.2.1 Choice of visualizations

In the interest of keeping Seastorm's implementation and user interface simple, we decided to implement a single visualization to begin with and only add others if we had time and found it useful. As such, we began by selecting which visualization to implement based on the design goals of Seastorm, listed in section 1.2. We considered two visualizations: the sequence diagram and the communication diagram.

The communication diagram is better than the sequence diagram at visualizing network topology and other information about network links. Given Seastorm's focus on education, however, we did not find this a significant problem, since education on distributed systems generally emphasizes algorithms that do not make assumptions about latency, bandwidth, and reliability. Further, the lack of route information does not affect Seastorm, as Seattle runs with no topology restrictions on the Internet, where full connectivity between all hosts is generally assumed.

Although the design goals do not require Seastorm's visualization to handle large algorithms, we also looked at how the two diagrams scale in general, as scaling issues can appear even in small algorithms. From this perspective, a benefit of the sequence diagram is that it can always grow on the vertical axis (representing time), unlike the communication diagram.

Possibly the most significant drawback of the communication diagram is that it does not visualize the order of events as clearly as the sequence diagram. Using multiple separate diagrams could make ordering more explicit, but it would instead hide more information and possibly make the diagram tedious to navigate.

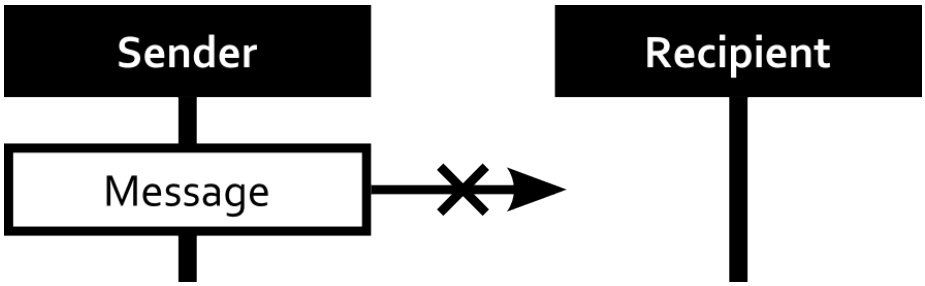

Figure 4.2: Visualization of unreceived messages in Seastorm

For these reasons, the sequence diagram was the first and only visualization that we implemented for Seastorm. In the interest of keeping the implementation and user interface simple, we ultimately did not implement any others.

4.2.2 The sequence diagram in Seastorm

This section describes the sequence diagram that we designed specifically for Seastorm, which differs somewhat from that used in UML. Notable differences between Seastorm's sequence diagram and that in UML are described in section 4.2.2.5.

4.2.2.1 Layout

When visualizing an experiment, Seastorm presents each vessel taking part in the experiment as a lifeline. Along a vessel's lifeline, Seastorm presents the events taking place in that vessel as boxes, ordered by their timestamps. Seastorm presents each send event as an arrow leading to the box in another vessel that represents the corresponding receive event. Each box contains a string describing the event, such as a log line in the case of internal events or the payload of a message in the case of messaging events.

4.2.2.2 Unreceived messages

If a message was sent but not received (due to a network failure, for instance), Seastorm draws a cross on the arrow's midpoint to indicate this. Additionally, the arrow is perpendicular to the lines and leads to the line of the receiving vessel, rather than to a box along that line. Figure 4.2 illustrates this.

Intuitively, it may seem unrealistic and misleading that an unreceived message is presented as an arrow perpendicular to the timelines, implying that the message would have arrived instantly if it had not been lost. We could not find a more general and less misleading representation, however. The reason for this is that in general, due to the inherent properties of distributed systems, we do not know when and where the message in question was lost, and thus cannot accurately decide what the arrow would have looked like if the message had been received.

We considered a potential solution to this based on calculating the average delivery time of other (successfully received) messages in the experiment and using this to draw an arrow with a more realistic angle. Because Seastorm

measures time using logical clocks rather than physical clocks, however, this would not yield a meaningful result.

Further, even if we added physical timestamps for this purpose, the approach would fail in executions that contain no successfully delivered messages and thus provide no data for estimates to be made. Finally, and perhaps most importantly, this method of drawing arrows would risk misleading users by giving the illusion that something is known about how the unreceived message traveled through the network.

In many sequence diagrams, arrows representing unreceived messages are also shorter than other arrows, to indicate that the message was lost somewhere between the sender and the recipient. Seastorm, however, draws such arrows all the way to the recipient. If it did not, there would be many scenarios in diagrams involving more than two vessels where the recipient of the message would not be clearly pointed out by the arrow. In other words, only drawing an arrow halfway to its recipient could make it seem like the message was intended for a vessel between the sender and the recipient in the diagram.

4.2.2.3 Undelivered messages

There is another category of messages that Seastorm draws in the same manner as lost messages: those that were successfully received by the receiving host but not delivered to the application on that host because the application did not call `getmessage` or `recv`.

In theory, Seastorm could draw these messages in some other fashion to distinguish this scenario from messages lost in transition, such as the end of the arrow "bouncing" off an event box in the other vessel. In practice, we did not find a practical way of doing this, due to the fact that Repy programs are not informed of undelivered messages.

4.2.2.4 Concurrent events

Seastorm draws each event box on a separate row in the sequence diagram, even when it may be concurrent with other events. Compared to displaying concurrent events on the same row, we found that this avoids a set of problematic edge cases: when two or more event boxes are placed on the same row and one or more of those events are send events whose messages were lost, the horizontal "lost message" arrows may well overlap almost entirely with other such arrows on the same row, or be covered to a large degree by other event boxes on that row.

When deciding in which order to present concurrent events, Seastorm orders them based on their timestamp. When their timestamps are equal, it breaks ties based on the IDs of the vessels.

4.2.2.5 Differences from the UML sequence diagram

One notable difference between Seastorm's sequence diagram and the UML sequence diagram is the fact that arrows are not always horizontal, in order to represent the delay between a message being sent and received. In that sense, it more resembles the sequence diagrams often used to illustrate TCP connections, such as the one in figure 4.3.

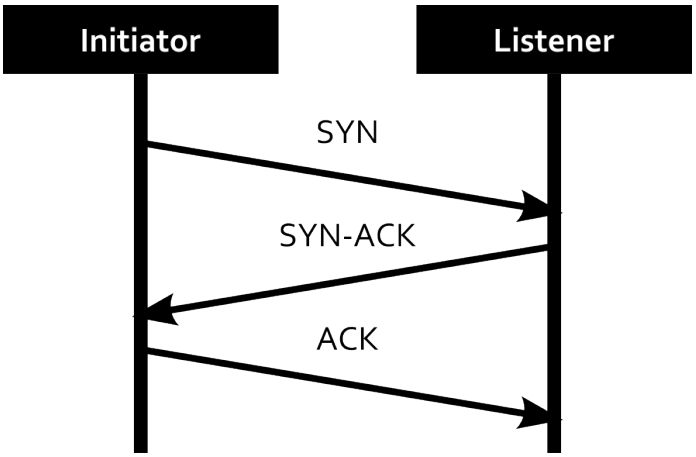

Figure 4.3: TCP sequence diagram

Another difference is the usage of synchronous messages: while the UML sequence diagram allows for them, Seastorm does not use them at all, since all of the communication in Seattle is asynchronous.

Finally, in Seastorm's sequence diagram, each event has its own description inside its box. In contrast, each message in a UML sequence diagram (consisting of a send event and a receive event) has a single description presented alongside the message's arrow.

4.3 Events

Based on the event abstraction described in section 2.2.1, this section outlines how Seastorm defines events in a Seattle experiment. This in turn affects the design of Seastorm's monitor, described in section 4.4.

4.3.1 Event sources

4.3.1.1 Messaging events

Seastorm recognizes two sources of messaging events in Seattle:

- UDP messages
- TCP connections

UDP messages In Seastorm, one call to `sendmessage` generates one send event. Similarly, one call to `getmessage` generates one receive event.

TCP connections Identifying messaging events from a TCP connection is significantly more complex than for UDP messages. This is because a single TCP connection is often used to send multiple messages, even though the connection itself includes no concept of discrete messages; all data being sent is a single stream of bytes. When application code uses custom logic to send

multiple discrete messages over the connection, a monitor working with the TCP abstraction cannot distinguish between them.

While we could solve this problem by letting the user explicitly signal the beginning and end of messages using Seastorm-specific function calls, this goes against the goal that Seastorm should have a low barrier to entry, because it would require the user to learn a new API, and to modify existing programs to work with Seastorm.

We thought at first that the sockets used for TCP communication in the Repy API provided a solution: each individual call to `send` and `recv` could generate one event. We realized that this approach was infeasible for several reasons, however, the primary one being that the calls are not symmetric. In other words, the bytes sent with one call to `send` could be received with multiple calls to `recv`, and the bytes sent with multiple calls to `send` could conversely be received with a single call to `recv`. This means that calls to socket operations do not map to any concept of messages used by the user, prompting us to consider this approach unsuitable.

Instead, Seastorm keeps the monitoring of TCP communication transparent to the user by imposing a significant limitation on how this communication is interpreted: each TCP connection is treated as an exchange of only two messages, regardless of the amount of data being sent, the number of calls to `recv` and `send` being made, and the time taken to send the data. These two messages are the entire contents of the two byte streams sent over the connection. The send event is generated when the socket is opened on the local end, and the receive event is generated when the socket is closed on the remote end. If one end of the connection does not send any data over the connection, no send event and corresponding receive event is generated.

The effect of this is that Seastorm is less than optimal for forms of TCP communication involving multiple discrete messages (as defined by the user) being sent by one or both sides of the connection. This is because each collection of messages will be visualized as a single message, based on when the entire byte stream started and stopped being sent.

4.3.1.2 Internal events

While any number of sources could potentially provide internal events in Seattle, we found one particularly suitable because of its simplicity and flexibility: the `log` function, which is used to generate arbitrary text output from a Repy program. In Seastorm, one call to `log` generates one internal event.

We identified one other source of important internal events: exceptions, which are important for debugging. When a vessel terminates due to an exception, one internal event is generated.

4.3.2 Event data and metadata

4.3.2.1 Data

Each event is associated with some data. In the case of messaging events, this is the payload of the message that was sent. In the case of internal events, this is a string logged by the user (with a call to `log`) or an error message and a stack trace (from an exception).

4.3.2.2 Metadata

Each event also contains metadata. The most crucial piece of metadata in an event is its timestamp, which is used for ordering.

Messaging events Messaging events also contain the following metadata:

- ID of the sender
- ID of the recipient
- Time of departure

Finally, receive events also contain the message's time of arrival.

Seastorm needs this metadata in order to match send events with their corresponding receive events. The IDs are required to identify the sender and the receiver, while the times are required to disambiguate between multiple messages sent between the same sender/receiver pair. This requires timestamps to be unique within each vessel.

4.4 Monitoring

4.4.1 Detecting events

Seattle itself makes available only a small amount of data on vessel activity, such as a vessel's log and its status. This means that, in order to record the events taking place in an experiment running on Seattle, Seastorm must add extra logic to the programs in the experiment.

We could do this by providing a library containing alternate versions of functions in the Repy API that perform the extra logic as well as invoke the original functions. The user would then call these library functions instead of the original ones. A significant problem with this approach is that it would require new programs to be written with Seastorm in mind and existing programs to be modified in order to be of use with Seastorm, which goes against the design goal that Seastorm should have a low barrier to entry.

We instead selected an approach that is less intrusive for the user: to implement a preprocessor that automatically inserts the extra logic in all the required places in the source code.

4.4.2 Ordering events

An initial prototype of Seastorm used the NTP-synchronized physical clocks of Seattle nodes to order events, partially to achieve basic functionality as quickly as possible, and partially to confirm the need for alternative synchronization methods in future versions. Minimal example programs encountered the clock synchronization problems described in section 2.2.2, and they did so frequently enough to confirm that an alternative or complementary synchronization method was required.

We concluded that the crucial property of a sequence diagram is that events are visually ordered according to the happened-before relation. In other words, events that took place before other events must be visualized as doing so. As

per the definition of the happened-before relation, this means correctly ordering events within a vessel and making sure that send events are always ordered before their corresponding receive events. Seastorm uses Lamport timestamps to achieve this ordering. There are notable alternatives to this choice, however, as we discuss in section 7.2.1.5.

Concurrent events and causality violations, on the other hand, do not need special treatment, as explained below.

- Concurrent events can be ordered in any way, as nothing is known about their true ordering.
- Causality violations are not prevented by Repy's networking API, which delivers messages as soon as they arrive. Since Seastorm only intends to monitor the behavior of Repy programs, not alter them, it likewise does not prevent causality violations. Messages are instead ordered according to their actual arrival times, and causality violations can thus be identified by crossing arrows, as in figure 2.4.

We should note that Seastorm can detect causality violations, even though this is generally impossible with only Lamport timestamps. The reason for this is that Seastorm orders events after-the-fact, with ordering data from all vessels available. This process is described in section 5.2.2.

4.4.3 Augmenting the Repy API

Whenever an event takes place in a vessel, Seastorm performs the following additional logic, automatically inserted by a preprocessor:

- The vessel's logical clock is incremented.
- The event is logged. (The exact meaning of this is described in section 5.2.2.)

For messaging events, Seastorm also takes the following steps, as per the Lamport timestamp algorithm:

- For send events, the logical clock value of the vessel, after being updated, is included in the message.
- For receive events, the clock value included in the received message is used to update the vessel's logical clock before incrementing it.

4.4.3.1 Messages to and from external processes

Seattle vessels can send messages to and receive messages from processes that are external to Seattle (such as web browsers) and whose activity thus cannot be directly controlled and observed by Seastorm. Since external processes are not configured to use Lamport timestamps in the way that vessels are, Seastorm cannot properly order the events caused by these processes and thus ignores them.

4.5 User interface

The Seastorm user interface consists of three parts: the file panel, the vessel panel, and the visualization panel.

4.5.1 File panel

The file panel lists the files that the user has included for use in Seastorm. These files come from one of two sources:

1. A directory on the user's computer, if the user chooses to enable this functionality. In this case, the user edits the files outside of Seastorm, and the user interface simply lists the files available.

2. From the browser's local storage, if the user does not enable the above functionality. In this case controls are provided to create, remove, and edit files directly inside the browser.

4.5.2 Vessel panel

The vessel panel lists the vessels that a user has acquired through the Seattle Clearinghouse website. Apart from displaying the IP address of each vessel, the vessel panel allows the user to associate the following information with each vessel:

- File: A program (selected from the files available in the file panel) that the vessel will execute when started.

- Arguments: An optional list of arguments that will be passed to the program when the vessel is started.

- Title: A string that will be used in place of the vessel's IP address when visualizing the activity of the vessel.

- Active: Whether the vessel will take part in the next execution.

The vessel panel allows the user to execute three different commands on the vessels: start, stop, and reset.

4.5.2.1 Start

The start command starts an experiment by uploading all of the user's files to each vessel and then starting the vessels at approximately the same time. While the vessels are running, the results of the experiment are continuously displayed in the visualization panel.

4.5.2.2 Stop

The stop command stops an experiment by stopping all of the vessels and then displays the final results of the experiment in the visualization panel.

4.5.2.3 Reset

The reset command stops all vessels and removes all files on them, just like the reset command in Seash. It is only intended for exceptional situations where unexpected errors cause the other commands to not work properly.

4.5.3 Visualization panel

The visualization panel contains the sequence diagram, which is described in section 2.1.3.1. In addition, it contains a table that lists all the events in an experiment as raw data. Finally, it contains a panel displaying the details of a single event selected by the user.

The visualization panel also enables the user to save the current visualization to a file, and to open other files saved in this way.

The visualization panel can be opened in a new window in order to provide the user with more room when analyzing the data.

4.5.3.1 Interactivity

In order to make the visualization more effective, Seastorm lets the user interact with the sequence diagram.

When an event is given focus by the keyboard or hovered over with the mouse pointer, the event's rectangle is highlighted. In addition, if the event is a send event, the arrow leading from the event's rectangle and the rectangle of the related receive event are also highlighted; vice versa if the event is a receive event. The sequence diagram and the table are linked, so that focusing on an event in one of them also highlights that event in the other.

When an event is selected with the keyboard or mouse (in the same way that a hyperlink is activated), highlighting takes place as described above and the details of the event are displayed in a separate panel.

The user can also zoom the sequence diagram in and out in order to get a broader or more narrow view of it.

Chapter 5

Implementation

This chapter describes how we implemented the design described in the previous chapter. It begins with a description of Seastorm's technical architecture, including the distinct software components that together form the application and how these components communicate. It then goes into detail about how Seastorm monitors Seattle experiments in order to gather execution data. Finally, it provides an overview of how Seastorm draws the sequence diagrams used to visualize experiments.

5.1 Technical architecture

On a technical level, Seastorm consists of two separate applications: an HTML5 application running in the user's browser (the client) and a Python application running as a process on the user's computer (the server). Nearly all of the functionality is contained within the HTML5 application; the Python application is used solely to bypass a few restrictions and limitations of HTML5. This architecture is illustrated in figure 5.1.

We deliberately chose this "heavy client" model in order to make the barrier to entry as low as possible. Keeping the server lightweight makes it less likely that it will need to be updated when a new version of Seastorm is released, which in turn means that most new versions can be distributed through the automatic update mechanism already present in the browser. The effect is that the user will rarely, if ever, have to download and run a new version of the Python application.

Another benefit of this architecture is that it becomes easier to remove

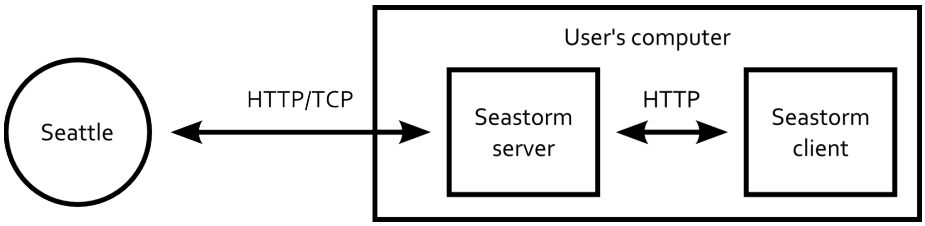

Figure 5.1: Technical architecture of Seastorm

the need for the server entirely if future developments remove the browser restrictions and limitations currently affecting Seastorm. For instance, improved support for cryptography in the browser and alternate versions of the Seattle APIs could remove the need for the server. In such a situation, moving all of the functionality into the client is much easier if the server contains very little functionality to begin with.

Requiring the use of a server works against the goal that Seastorm should have a low barrier to entry, because it requires a program to be downloaded and run outside of the browser. This is mitigated, however, by the fact that users of Seattle already have to install Python (in order to use Seash), which means that the only additional requirement is to download and run a single Python script.

5.1.1 Platform

Seastorm is developed for HTML5. At present, and for the foreseeable future, we consider HTML5 to be the most practical way of delivering a platform-independent application that requires no installation. Alternative platforms, such as Flash and Java, may have large market shares, but they still do not match the penetration of browsers. The market share of those alternative platforms are also at risk of a significantly reduced market share in the next few years, as the latest HTML5 standards become more widespread.

5.1.1.1 Bypassing browser restrictions and limitations

In order to get around the restrictions and limitations described above, Seastorm uses a Python script that runs on the user's computer and implements the required functionality. The Python script exposes the functionality by running a local web server that the HTML5 application can communicate with.

5.1.2 Client–server interaction

The client interacts with the server by sending HTTP requests to it, in order to access its unique functionality. Using different port numbers on `localhost`, the server exposes a proxy for the Clearinghouse API, a proxy for the Node Manager API, and a local filesystem server. The server uses CORS to allow the client access to it.

5.1.2.1 Clearinghouse API proxy

In order to access the Clearinghouse API, the client sends a request to the server, which forwards the request to the Clearinghouse API, waits for the response, and finally passes the response on to the client. The client application never directly communicates with, and has no knowledge of, the Clearinghouse API server. Both the proxy and the actual API use exactly the same XML-RPC protocol, however, so we could trivially remove this proxy from Seastorm if CORS support were ever added to the Clearinghouse API. (The possibility of adding CORS support was discussed with the developers of Seattle, but this approach was ultimately disregarded due to the other browser restrictions and limitations mentioned in section 2.4.1, which are more significant.)

5.1.2.2 Node Manager API proxy

The Node Manager API proxy exposes the functionality of the Node Manager API over HTTP. Unlike the Clearinghouse API, the Node Manager API uses a proprietary protocol that is not supported by browsers, which means that the server must perform some degree of translation in order to expose the API to the client.

Seattle provides a client library for the Node Manager API, written in Repy, that takes care of the protocol specifics. The proxy makes use of this library in order to be as lightweight as possible. The proxy exposes the methods of a node manager to the client through URL paths containing the IP address and port number of the node manager to contact as well as the name of the method to call. The client calls a method by making a POST requests to one of the URLs, with arguments passed in the request body. The server passes the return value or error message in the response body, with the status code distinguishing errors from return values.

We made one decision when exposing the Node Manager API over HTTP that makes the server more complex than necessary: the client calls methods with named arguments rather than positional arguments. This makes the server more complex because the methods available must be provided by a table hardcoded into the proxy, mapping argument names to argument positions. It also means that any future changes to the Node Manager API will require changes to the proxy. As the API had not changed in five years when we developed Seastorm, however (with the exception of a single method added to provide support for Repy version 2), we deemed it stable enough to consider future-proof. We thus decided that this drawback was outweighed by the benefit to development of being able to pass named arguments.

5.1.2.3 Local filesystem server

In order to allow users to write programs in their editor of choice rather than the one embedded in the web page, the Seastorm server optionally serves files to the web page from a single, user-selected directory on the local filesystem. This works around the client's otherwise limited access to the local filesystem.

The client accesses a list of the available files by connecting to the server and listening for server-sent events. Upon the initial connection, one event containing the filename of each available file is sent from the server. The client downloads each of these files from the server by sending a request to the server for each file, specifying the name of the file to retrieve.

In order to detect changes to the available files (such as when the user edits and saves a file in a text editor, or deletes a file in a file manager), the server polls the filesystem regularly (at a one-second interval). Whenever a change is detected—which includes file modification, creation, and deletion—an event is created, prompting the client to request the file. If the file exists, the client updates its file list with the latest version of the file. If the file has been removed, the client removes it from its file list.

5.2 Monitoring

5.2.1 Detecting events

Seastorm adds monitoring to a Repy program by uploading a wrapper library to each vessel and then adding a single line to the beginning of each user file that dynamically includes this library at runtime. The library replaces the built-in functions and methods with wrappers that perform extra logic in addition to calling the built-in function. The main metaprogramming features of Repy used to accomplish this are the ability to store a reference to a function in a variable and the ability to redefine previously defined functions.

We could have done this on a textual level by employing a Python parser to alter the actual source code of the programs, but this would add external dependencies by requiring the use of a parser. Making use of Repy's metaprogramming features to dynamically insert the extra logic a runtime, on the other hand, requires no external dependencies.

5.2.1.1 The wrapper library

This section describes how the wrapper library uses metaprogramming to add monitoring to the functions in the Repy API.

Standalone functions When adding monitoring to a standalone function (like `sendmessage` and `log`), the wrapper library takes the following steps:

1. A reference to the original function is saved in a variable.
2. A wrapper function with the same name as the original function is defined, effectively replacing the original function with the wrapper function.
3. The wrapper function performs the monitoring logic described in section 4.4.3.
4. At some point inside the wrapper function, the original function is called through the previously saved reference to it.

Object methods When adding monitoring to functions returning objects whose methods should in turn be monitored (such as `listenformessage` and `listenforconnection`), the process is similar. The only addition is that before being returned, the objects are wrapped in wrapper objects whose methods are monitored much in the same way as standalone functions.

5.2.1.2 Exceptions

When an exception occurs, the program running on a vessel terminates, which means that the monitoring performed by the vessel also terminates. As such, Seastorm cannot detect exceptions from within Repy.

Seattle still makes information about the exception available, however, via a vessel's log file. Thus, in order to detect exceptions, the client checks the status of the vessel. If the status is "Terminated" (which means that the program has exited, but not necessarily due to an exception), the client reads the vessel's log. If the log ends with a pattern matching that of a stack trace, the client

extracts the information about the exception and adds an internal event with the information to the trace file. This event is given a timestamp one greater than the last of the vessel's events, so that it will appear at the very end of the vessel's lifeline.

5.2.1.3 Logical clocks

The wrapper library stores the value of a vessel's logical clock as an integer in the global `mycontext` dictionary available to every vessel.

When a vessel's logical clock is sent in a message to another vessel, it is encoded as a 4-byte numeric string padded with zeroes, preceding the payload of the message. This means that the maximum supported logical clock value is 9999. We made the clock value fixed-length because it simplifies parsing, as the recipient knows exactly how many bytes to consider metadata. We chose a numeric string for this purpose because the Repy functions for receiving messages return strings, which means that an integer can easily be obtained by calling Repy's integer conversion function on the numeric string. Since we developed Seastorm with small algorithms in mind, we did not consider the limit of 9999 to be a pressing concern.

If clock values of more than 9999 are ever needed, the timestamp could instead be encoded bytewise as an integer, providing that the sender and the recipient use the same string encoding when converting the clock value to and from a string. Additionally, or alternatively, the length of the timestamp could be increased.

5.2.1.4 Messages to and from external processes

When an experiment starts, Seastorm uploads a text file to each vessel containing the IP addresses of the vessels taking part in the experiment. Whenever a vessel sends or receives a message, it also compares the message's recipient (when sending a message) or sender (when receiving a message) with the IP addresses and port numbers in this file. Only if the recipient or sender matches one of the IP address and port number pairs does the wrapper library actually log the messaging event.

5.2.1.5 Hiding timestamps from the user

When monitoring messaging functions, the wrapper library makes sure that the presence of timestamps in the messages being sent and received does not visibly affect the program for the user.

When sending a message (whether with UDP or TCP), the wrapper library returns to the user the number of actual payload bytes sent. This means that the length of the timestamp must be subtracted from the actual number of bytes sent over the network. If only the timestamp or part of the timestamp was successfully sent, the wrapper library returns 0 and no event is logged.

When receiving a message, the wrapper library removes the timestamp from it and only returns the payload to the user. If the message contains only a timestamp or part of a timestamp, no event is logged and the message is ignored.

5.2.1.6 Maintaining state in TCP connections

When monitoring TCP connections, the wrapper library takes into account the fact that a message can be sent or received with multiple calls to `send` or `recv`.

When `send` is called, the wrapper library sends the timestamp before sending any of the payload. If some or all of the timestamp fails to be sent, the wrapper library saves the remaining portion. When `send` is called again, the wrapper library attempts to send the remaining part of the timestamp before sending the payload. This process is repeated until the entire timestamp has been successfully sent, upon which the wrapper library begins sending the payload.

The wrapper library performs a similar process when `recv` is called.

5.2.1.7 Monitoring sessions

Whenever Seastorm starts a vessel, the wrapper library resets the state associated with a monitoring session (including the logical clock and the list of vessels taking part in the experiment). This means that a monitoring session lasts for the duration of an experiment.

5.2.2 Logging events

All events that the wrapper library detects must be saved somewhere so that they can ultimately be made available to the visualizer.

One way of doing this would be to have the vessels themselves send data about their events to a separate monitor process while they are running, and having the monitor process store the data. This approach would incur significant overhead, however, as every event taking place would cause a message to be sent to the monitor. This overhead would be especially significant if the messages were sent using TCP, which would be required in order to ensure a correct view of the experiment's execution. Additionally, the possibility of a TCP timeout would require vessels to also keep a local log of messages that have not yet reached the monitor, increasing the overhead even more. Finally, while this would provide as much of a real-time monitoring process as is possible, the benefits of this are small in a debugging environment, where most inspection will take place after-the-fact and not require instant results while the experiment is running.

The approach that Seastorm uses is instead based on vessels recording their events in locally stored activity log files, which Seastorm then downloads on demand when the data is needed. (The exact interval that Seastorm uses is three seconds.) This requires a vessel to only write an entry to its activity log whenever an event takes place (rather than sending messages), which incurs less overhead and less complexity while still being suitable for Seastorm's purposes.

Note that these activity logs are different from the logs that every Seattle vessel has automatically, and that a user can write to using the `log` function. For that reason we specifically use the term *activity log* when referring to the logs created by Seastorm.

5.2.2.1 File format

The file format that the wrapper library uses to log events is a subset of the de facto comma-separated values format (CSV). For each event, the wrapper

library saves all the data and metadata that will eventually be needed to create a valid trace file. Data that is provided by the user (such as message payloads), which could corrupt the CSV file if not properly escaped, is encoded using Base64. We selected Base64 for this rather than conventional CSV escaping methods because implementations of Base64 were already available in Repy (which is used to create the activity logs) and in JavaScript (which is used to parse them), unlike implementations of CSV.

5.2.2.2 Ordering of activity log entries

The wrapper library logs internal events and UDP messaging events immediately, which means that those will always be sorted according to their logical clock value. It does not, however, log TCP messaging events until the connection sending the message has been closed, since only then is the entire payload of the message known. Because the logical clock value of a TCP messaging event is the time at which the TCP connection was opened, this means that other events can be logged before the connection closes. When the TCP messaging event is finally logged, it can thus end up later in the activity log than events with a higher clock value. As a result, the entries in an activity log are not guaranteed to be sorted. Maintaining a correct ordering is instead the responsibility of the visualizer, as described in section 5.3.1.

5.2.2.3 Inconsistent cuts

Seastorm does not guarantee that the events recorded from an experiment represent a consistent cut. This is because it downloads the logs of the processes in the experiment without ensuring that the states they represent were all recorded at valid times.

We did not consider this a significant problem, as the errors in an inconsistent cut generally only appear toward the end of the sequence of recorded events, and are usually resolved the next time that Seastorm downloads the logs. Unless an experiment runs indefinitely, Seastorm will eventually download the final logs of the processes and construct a consistent cut. Since we designed Seastorm primarily for after-the-fact inspection of experiments, we did not consider this risk of temporary inconsistencies a significant problem.

5.2.2.4 Trace file

With a small amount of processing, the Seastorm client combines the Seastorm logs from all vessels taking part in an experiment into a JSON trace file. Events are serialized as a list of objects whose properties map to the event data and metadata described in section 4.3.2.

Apart from the list of events, the only other component of a trace file is a dictionary mapping vessel IDs to titles, provided by the user through the user interface.

5.3 Visualization

When deciding on a way to implement the sequence diagram visualization in HTML5, we used two criteria to make the decision: simplicity and responsive-

ness. The implementation should be simple, and the resulting visualization should be responsive. In this context, "responsive" means that the visualization fluidly adapts to the user's browser and makes use of the available space to provide information as clearly as possible. (This is the same meaning of the term as used in modern web design.)

We decided that the flexibility of Canvas was not required, and that its lack of responsiveness made it less desirable than the alternatives.

We found that HTML/CSS and SVG better met the criteria, but we also identified important limitations: HTML/CSS is not suitable for arbitrary graphics (such as arrows) while the lack of layout facilities in SVG makes it more complex than HTML/CSS.

In order to gain the benefits of both, we opted for a hybrid approach: the layout of the sequence diagram is created with HTML/CSS while non-rectangular graphics (such as arrows) are created with SVG and superimposed over the HTML/CSS layout.

5.3.1 HTML, CSS, and SVG

Seastorm represents the sequence diagram in HTML as a table of data with one column for each process. The table header contains a single row where each cell contains the title of the process associated with that column. The table body contains one row for each event in the trace, sorted by their timestamps, with ties broken by vessel IDs. The cells in this row are empty, except for the cell in the column of the process where the event took place. This cell contains the data associated with the event.

Using CSS, Seastorm gives each cell in the table body a background consisting of a line going from the top of the cell to the bottom of the cell along its horizontal middle, forming the timeline of the process associated with that column. Each non-empty cell in the table body is styled as a box, visually setting it apart from the other cells in the same row (which are empty).

Each cell in the HTML table corresponding to a send event contains an SVG object that in turn contains the arrow representing the sending of the message associated with that event. The dimensions of the arrow are calculated with JavaScript based on pixel coordinates acquired through the DOM. The SVG is finally positioned relative to its containing event box using CSS.

5.3.2 Arrows to self

Although rare in practice, it is possible for a vessel to send a message to itself. If it does, it is not guaranteed that it will be received immediately, as other events might take place in-between. Consequently, Seastorm needs to draw an arrow of arbitrary length from a vessel to itself (an "arrow to self").

Drawing an arrow to self is not as straightforward as drawing one to another vessel, for a few reasons. Perhaps most notably, multiple such arrows from the same vessel in close proximity may end up either overlapping almost completely or running out of horizontal space, depending on how they are drawn.

Figure 5.2 depicts the first situation, in which multiple arrows to self overlap and thus become indistinguishable. Figure 5.3 depicts the second situation, in which multiple arrows to self can potentially require so much horizontal space that the entire diagram becomes difficult to interpret.

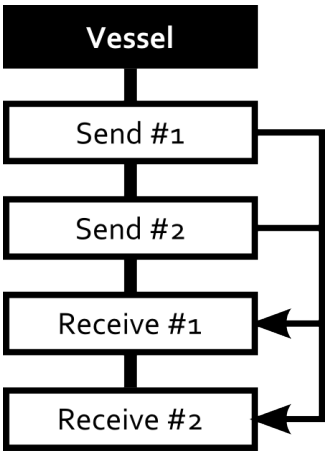

Figure 5.2: Overlapping arrows to self

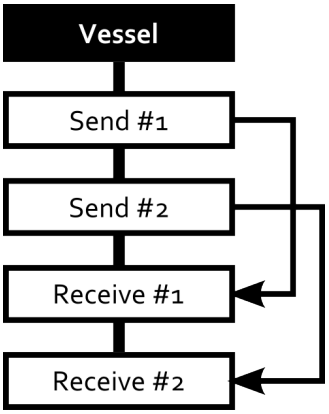

Figure 5.3: Adjacent arrows to self

While these problems can likely be solved either partially or completely, we decided that arrows to self were not a common enough use case to warrant the time that would be required to implement a solution to the problems described above. As such, Seastorm does not draw arrows to self.

Based on the results of our user testing, section 6.2.5.2 discusses why arrows to self may still be worth implementing.

5.3.3 Event titles

In some cases, event data might be too large to easily fit inside an event box. Seastorm allows the user to avoid this problem by giving optional titles to events. When an event has a title, that title, rather than the event data, is displayed in the event box.

A user gives a title to an event by calling the event-generating functions (`log`, `sendmessage`, and `getmessage`) with a `title` keyword argument. When this is done, the wrapper library logs the event title in addition to the event data. The title is likewise included in the resulting trace file. It should be noted that this is the only scenario where a user's code needs to be modified in order to achieve a certain visualization.

When an event has a title, the event's data can still be inspected in a separate panel displayed when the event is selected. This panel is described in section 4.5.3.1.

Chapter 6

Results

This chapter describes the results of this project from three different perspectives. First, it gives an illustrated overview of the main functionality of Seastorm. It then describes the user testing that we performed in order to assess the value of Seastorm's visualization based on user impressions. Finally, it describes a limited amount of performance testing that we carried out in order to identify possible bottlenecks in Seastorm, especially related to its user interface.

6.1 Functionality

6.1.1 User interface

Figure 6.1 shows the complete Seastorm user interface, with the vessel panel, file panel, and visualization panel in the same window. The file panel lists files from the local filesystem. It also shows a visualization demonstrating most of the features of Seastorm's sequence diagrams.

Figure 6.2 shows the same user interface except with an embedded text editor accessing files stored in the browser rather than on the local filesystem.

6.1.2 Visualization

Figure 6.3 shows the visualization of lost messages in Seastorm. As pointed out in section 4.2.2, the arrow representing the lost message sent from Vessel A to Vessel C necessarily misrepresents the speed with which the message moved through the network. Likewise, the arrow is as long as it would have been if it had been received, in order to avoid the risk of users interpreting it to be intended for Vessel B rather than Vessel C.

Figure 6.4 show the visualization of causality violations in Seastorm. The diagram does not explicitly highlight these violations in any way. Instead, the presence of crossing arrows allows the user to visually interpret this information when necessary.

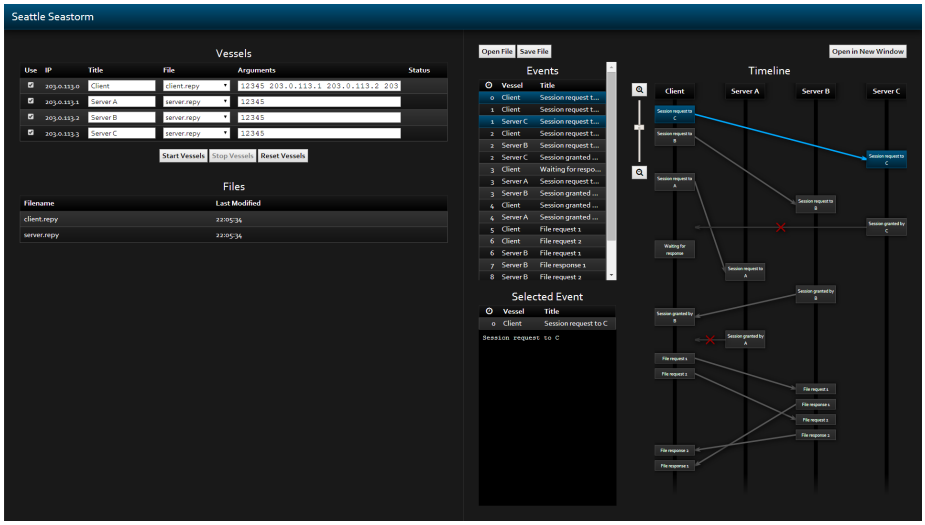

Figure 6.1: The Seastorm user interface with files from the local filesystem

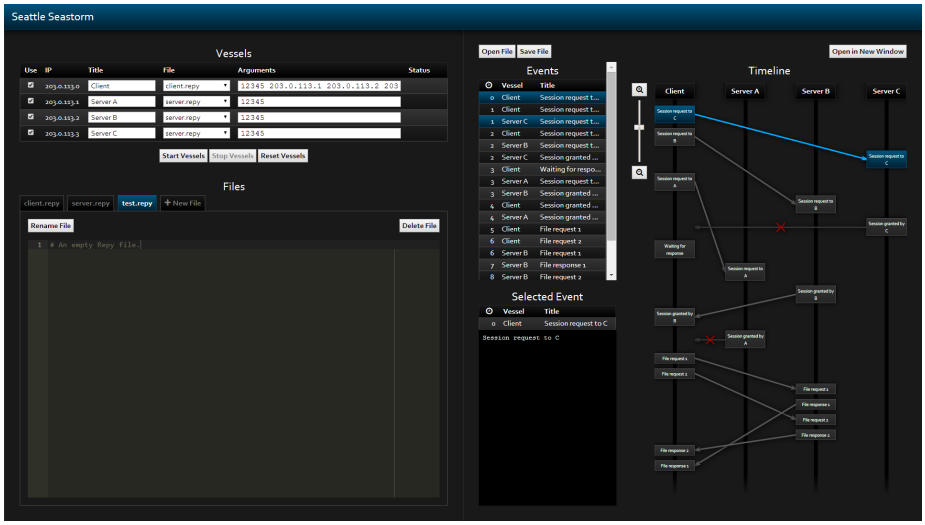

Figure 6.2: The Seastorm user interface with an embedded text editor

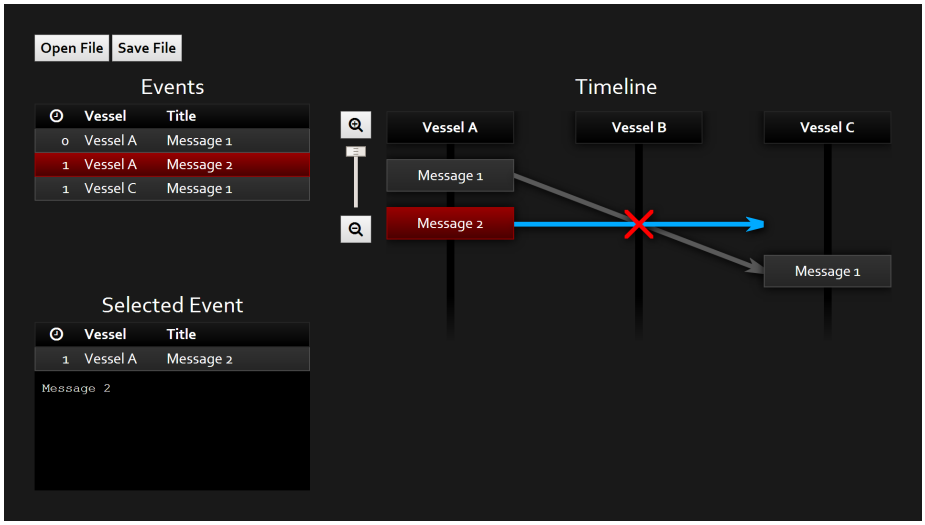

Figure 6.3: Visualization of a lost message in Seastorm

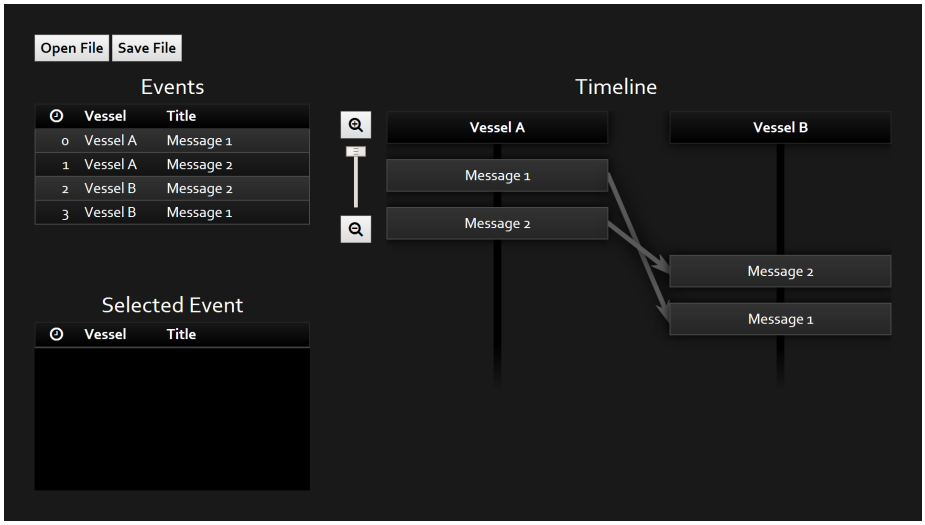

Figure 6.4: Visualization of a causality violation in Seastorm

6.2 User testing

In order to assess the value for users of having access to Seastorm's visualization while working with the Seattle platform, we performed user testing. Because there was no existing Seastorm user base from which to identify specific testing goals, and because resources for testing were limited, we opted for qualitative rather than quantitative testing. Our main goal was to gather overall user impressions of the software, which could then form the basis for future tests.

6.2.1 Test outline

The method that we selected for testing was one-on-one testing with one participant and one test leader at a time, due to the ease of performing and gathering results from such tests with limited resources. For the main part of the test, we gave participants two programming tasks to perform, one of them using Seastorm and the other using Seash. This was followed by an interview where we asked the participants questions about their experience and solicited them for feedback.

6.2.2 Design of tasks

Due to the limited amount of time per test session and the probability that some participants would need to refresh their memory on Seattle and Repy during the test, the tasks that we gave to the participants did not require them to write programs on their own. Instead, we provided them with two complete programs, each containing an error, and asked them to find and correct the errors. Given small enough programs, we deemed that the participants would be able to read and understand the programs in less time than needed to write similar programs on their own.

6.2.2.1 Design considerations

In order to better evaluate some of the unique features of Seastorm compared to other tools for Seattle, we designed the tasks to contain the type of errors that would be distinctly visible in Seastorm's visualizations. Two of the concepts that Seastorm visualizes most distinctly are lost or undelivered messages (indicated by crossed-out arrows) and messages being delivered in a different order than they were sent (indicated by two or more arrows crossing). Because of their distinctive visual representations, we identified these two concepts as particularly desirable to include in tasks assigned to the users during user testing.

We found a major obstacle to including either of them in a test, however: both of them are affected by the inherent unreliability of a distributed system and thus cannot consistently be reproduced. Put differently: it is not possible in practice to write a program that consistently loses a particular message or consistently delivers one message before another one that was sent earlier. Without this kind of guarantee, it is possible that a user by chance observes a correct execution of the program and (correctly) fails to identify any errors, possibly rendering the entire test useless. Although we could in theory introduce this kind of guarantee by modifying the testing environment to manipulate message delivery in some fashion, this would give the program unrealistic properties and

risk removing the user's trust in the environment. For this reason, we found it unfeasible to include these two concepts in user testing.

We identified one variant of the above concepts, however, that can be reliably reproduced: messages that are received by the destination host but ignored by the vessel on that host (because the user does not call `getmessage` or `recv`). Seastorm visualizes these in the same manner as messages lost in transition, and they indicate similar error conditions to the user. This means that crossed-out arrows can reliably be introduced into the visualization by introducing a defect into the program that causes it to not call `getmessage` or `recv` at the appropriate time.

It should be noted that the failure to reliably introduce certain types of errors into the user test indicate that some benefits of Seastorm could in practice only be available in real-world scenarios, where Seastorm could help to identify the errors when they happen to appear.

6.2.2.2 Resulting tasks

Based on the considerations described above, we designed two tasks. The errors that we introduced into the tasks both had a visual representation: a crossed-out arrow in the first task and an arrow leading to the wrong place in the second task.

Task 1: Calculator This task involved two vessels: a client sending requests for simple mathematical calculations, and a server performing these calculations and replying with their results. The error that we introduced into the code was an off-by-one error that caused one message to be ignored by its intended recipient.

Task 2: Auction This task involved four vessels: a seller offering an item for sale, and three bidders placing bids on that item. The error that we introduced into the code was an ordering error that caused one message to be sent to the wrong recipient.

6.2.3 Test execution

Eligible participants were readily available from the pool of Chalmers students who had previously taken the Distributed Systems course. We recruited participants from two sources: M.Sc. students who had taken the course, and Ph.D. students who had both taken the course and acted as teaching assistants for it. The only requirement of participants was that they have previous experience with Seattle.

We recruited a total of four M.Sc. students and three Ph.D. students in this way. One of the M.Sc. students participated in a pilot test used to identify problems with the test or the tasks given. Although the tasks were slightly redesigned based on the results of the pilot test, we still included the pilot test in our qualitative findings.

In order to make the test sessions as unobtrusive as possible for the participants, we set the length of a test session to 45 minutes, and no preparation of any kind was required of the participants. Of the 45 minutes, 15 were allotted

	Participants	Unsuccessful	Successful
Task 1 with Seash	3	3	0
Task 1 with Seastorm	3	2	1
Task 2 with Seash	3	2	1
Task 2 with Seastorm	3	2	1

Table 6.1: Quantitative results of Seastorm user testing

to the first task, 15 to the second task, and 15 to the interview and instructions given at the start of the test session.

During each test session, the voices of the the participant and the test leader were recorded and screenshots of the test computer were automatically taken every 30 seconds (with the participant's consent), for use as reference data.

The main objective of the user test was to assess the value of Seastorm's visualization rather than the design of Seastorm's user interface, but we also noted comments about the user interface when given.

While Seastorm was designed for use with Repy version 2, the participants only had experience with the version 1. For this reason, we created a separate version of Seastorm with partial support for Repy version 1, to the degree needed for the tasks that we designed. We then used this alternate version of Seastorm in the user test.

6.2.4 Quantitative results

While the test itself was qualitative, we present the quantitative results of the test in table 6.1 for completeness. These results do not include the one participant who took part in a pilot test with slightly different tasks given.

We quantified success with one of two results: failing to locate and correct the error ("unsuccessful"), and successfully locating and correcting the error ("successful"). This means that participants who located the error but were not able to correct it were placed in the first group.

The only clear-cut observation that we made from these results was that the tasks given were too difficult for most participants to complete in the time given. While we do not necessarily consider this a problem in a qualitative study, we would likely strive towards making the tasks simpler or giving the participants more time in any future quantitative studies.

6.2.5 Qualitative findings

6.2.5.1 Automatic logging

The very first step taken by most users for most tasks, whether they were using Seastorm or Seash, was to start the programs and observe any effects, without reading the source code.

When this was done with Seash, participants quickly realized that the programs provided to them contained no logging statements, practically rendering the result of the first execution useless and prompting them to start reading the code and possibly insert custom logging statements before starting the programs again.

When this was done with Seastorm, on the other hand, the participants always saw a result that they could analyze, and based on this were often able to identify the general nature of the error that they were looking for. Further, one of the most common benefits of Seastorm pointed out during the interviews was the automatic logging that Seastorm performed without requiring any modification of the source code by the user.

It should be noted that this benefit by itself is independent of the visualization provided by Seastorm; it's possible that a similar logging feature provided by Seash would be considered equally useful even with no visualization.

Even then, we see reason to believe that the source code is not always the most representative view of a program's behavior to many users. In fact, one participant started out by translating the intended behavior of the program to a sequence diagram by hand, then comparing that to the visualization provided by Seastorm, and only then inspecting the source code. This also highlights the generality of the sequence diagram, and how familiar it is to many programmers.

6.2.5.2 Usefulness of arrows to self

One participant, when using Seash for task 1, accidentally started the client with incorrect arguments, causing messages to be sent from the client to itself, rather than to the server. The participant was not able to identify this mistake in the allotted time using only custom logging statements.

Seastorm would have clearly visualized this error, as the client's timeline would contain both the sending and the receipt of the message, whereas the server's timeline would be empty. Since Seastorm does not draw arrows between send events and receive events in the same process, however (due to complications discussed in section 5.3.2), the error would not be as self-evident as it could be. This finding indicates the value of implementing arrows to self, if only to help users avoid similar errors.

6.2.5.3 Batch vessel operations

Several participants pointed out the value of being able to perform batch operations on vessels, such as simultaneously starting all available vessels with different files and different arguments. Similar functionality is either not available or not readily apparent to users in Seash.

At the same time, participants expressed a desire to retain the fine-grained control over individual vessels available in Seash (such as starting a single long-running server vessel only once), which Seastorm provides only to a small extent.

6.2.5.4 Ambiguity of crossed-out messages

The unfortunate similarity between lost messages and ignored messages (both visualized as a crossed-out arrow) is pointed out in section 4.2.2 and did indeed cause some degree of confusion among participants. Although all of them correctly identified the crossed-out arrows as indicating some form delivery failure, the initial reaction of most was to assume a network failure rather than the message being ignored by the recipient, in some cases prompting them to run the program again in order to confirm their suspicions.

This confusion was expected and points to the value of differentiating between the two errors, but we do not at this point know whether there is a practical way of doing so.

6.2.5.5 Risk of noisy diagrams

In some instances, the layout of the sequence diagram presented to a participant caused problems when interpreting the data. One example of this is an arrow that was partially concealed by boxes, causing it to not stand out as much as other arrows. Another example is a crossed-out arrow whose midpoint aligned closely with the midpoint of another arrow, making it ambiguous which arrow the cross belonged to. Approaches to reducing the impact of this type of noise are discussed in section 7.2.1.1.

6.2.5.6 Familiarity with Repy

A significant challenge for many participants was to recall the commands and functions available in Seash and Repy. When the user testing was performed, a few months had passed since the participants last used Seash and Repy. (When the tests were performed, almost half a year had passed since the last time the Distributed Systems course was given at Chalmers, making it difficult to find participants with recent Seattle experience.) Although the participants were provided with a summary of Seash commands, the Repy API documentation, and the opportunity to ask questions about Seash and Repy, their need to recall Seattle functionality was an unfortunate problem that introduced distractions to the tests.

6.2.5.7 Abstract nature of time and ordering

On at least one occasion, a participant interpreted the ordering of events in the sequence diagram literally and made an unwarranted assumption about latency in the network. The participant's assumption was that events are ordered using (correctly synchronized) physical timestamps, and Seastorm's sequence diagram did not provide any indication that logical clocks were in fact used instead, potentially yielding a different visualization than a corresponding visualization based on physical timestamps.

Although the same line of thinking was not explicitly highlighted by other participants, we find it reasonable to assume that most users will intuitively interpret the sequence diagram in this literal sense.

6.3 Performance testing

In order to identify possible bottlenecks in the design or implementation of Seastorm, we performed a small amount of performance testing. Due to the decoupled nature of Seastorm, it is possible to test each of its two major features—visualization and monitoring—in isolation.

6.3.1 Visualization

Because Seastorm creates visualizations from trace files, any conceivable sequence diagram can be created and tested by creating a corresponding trace file. The impact of the network or previous trace files is not relevant even when visualizations are created during an experiment. This is because the entire trace file is compiled and provided in a single operation to the visualizer, which then visualizes the trace file from scratch.

The most relevant aspect of Seastorm's visualization that we identified from a performance perspective was the user interface's speed when the user interacts with the sequence diagram. Under ideal conditions, events should highlight without a visible delay when the user hovers over them with the mouse pointer. Similarly, the diagram should smoothly resize when the zoom functionality is used.

The performance of an HTML5 application usually depends on both the computer's hardware and the browser's JavaScript engine. Although our performance testing was not intended to be rigorous, we still wanted to establish a metric of the browser's performance on any given computer, in order to make the results more relevant. For this purpose we used Dromaeo [29], a work-in-progress test suite for browser performance.

6.3.1.1 Test execution

We used two different computers in the performance test. In order to compare their browser performance numerically, we used the total number of runs per seconds given after the running the suite of recommended tests with Dromaeo in Google Chrome 35. On computer #1, running Windows, the number was 449.31. On computer #2, running Linux, the number was 839.95.

In order to test the performance of the Seastorm sequence diagram on these two computers, we generated trace files containing two vessels: one that continuously sends messages, and one that continuously receives those messages in the order that they were sent. The result is a diagram consisting of n event boxes and $n/2$ arrows between those event boxes, where n is the number of messages sent and received in the trace file. (The fact that all arrows in the diagram will have the same dimensions should not improve performance, as Seastorm still calculates the dimensions of and draws each arrow individually.) We generated such files for several different values of n: 50, 100, 150, 200, 500, and 1000.

In order to get a rough impression of when the user interface starts to visibly slow down, we opened the files in Seastorm on both computers. We then interacted with the diagram and manually judged when there was a visible drop in performance.

6.3.1.2 Findings

On computer #1, performance issues were noted at $n = 150$ and above. On computer #2, the corresponding number was $n = 200$. When these numbers were reached, interactivity was noticeably slower, including the highlighting of events under the mouse pointer and the zoom functionality.

As we designed Seastorm with small algorithms in mind, these numbers would not negatively affect the most common use cases on the computers that

we tested. In order to enable more use cases, however, Seastorm's visualization would need to be optimized for performance.

We should note that we did not implement Seastorm's visualizer with performance in mind, which means that there are most likely ways in which it could be considerably improved. It is also possible that Canvas could be used partially or entirely in place of HTML/CSS and SVG in order to increase performance.

6.3.2 Monitoring

We identified one primary aspect of monitoring in Seastorm that can be tested for performance: the overhead caused by the monitoring logic inserted into the Repy programs in an experiment. This overhead includes the processing cost of performing monitoring logic as well as the processing and bandwidth cost of the node manager sending the activity log over the network when polled for it.

We did not carry out any performance testing of Seastorm's monitor, as we did not consider it crucial given the design goals. We did not encounter performance problems in the experiments run during our user test, and we concluded that there are enough algorithms used in education that would yield experiments of similar sizes and thus not be negatively affected by performance concerns.

If performance testing of the monitoring were to be attempted in the future, one challenge to overcome would be the fact that users are given very limited control of Seattle vessels and thus cannot run custom applications on them, such as Python profilers. In order to have access to the tools needed for testing, one or more custom Seattle nodes would likely have to be set up.

Chapter 7

Conclusions

This chapter begins by summarizing the key points of this report, with a focus on the results of the project. It then outlines several suggestions for future work related to Seastorm, in two different areas: improvements to the design and functionality of Seastorm, and expanded user testing that can identify benefits and drawbacks of Seastorm in a more rigorous way.

7.1 Summary

We set out to create a visualizer for the Seattle platform aimed at students learning about distributed systems. Our main goals were to give it a low barrier to entry yet still make it useful for a large number of use cases.

We achieved the low barrier to entry by making the visualization seamless to the user, not requiring any changes to program source code. Further, we made the installation and update process as simple as possible within the constraints, requiring only the use of a web browser and a small Python script.

We made the visualizer useful for many different situations by generating sequence diagrams, which can represent any algorithm based on processes passing messages. In addition, the diagram is familiar to students from common course material and thus often maps well to their mental models of such algorithms.

Based on qualitative user testing with users having previous Seattle experience, we found that users quickly learned to use the visualizer and gave positive assessments of it. Although testing was not rigorous enough to provide clear evidence of the visualizer's usefulness, comments from users provided several examples of how it simplified the development process.

We believe that future work should focus on two different areas. First, the visualizer should be improved for current use cases and expanded to become suitable for new use cases, such as larger and more complex algorithms. Second, user studies should be conducted to investigate what qualities make visualizers attractive for use in real-world scenarios, given the positive reactions to such tools during testing but low adoption rates in the wider developer community.

7.2 Future work

7.2.1 Design and implementation

7.2.1.1 Filtering and sorting

In many cases, inspecting the behavior of a complex program is not a matter of using abstractions that can present all the information at once, but rather about viewing smaller parts of the program in isolation. This is how traditional debuggers and unit tests approach program complexity.

Seastorm does not provide features for filtering and sorting visualizations, and we did not consider such features crucial for the small programs that we designed it for. Even in educational settings, however, programs can grow too large to become impractical to visualize. One such example is a program that involves file transfers: showing every single message being sent in the transfer of large files would quickly clutter up the visualization.

We believe that filtering and sorting capabilities could make Seastorm a viable tool even for larger programs, in the same way that advanced filtering features makes tools like Wireshark useful even for traffic-heavy network connections. Relevant factors to filter on could be event data and metadata in general, including message payloads and event timestamps.

There would also be challenges in implementing features like these. The features would need be intuitive enough that users can make use of them without much training. They would also need to handle the absence of filtered events and vessels without presenting misleading information. For instance, if an entire vessel is hidden by the user, a decision needs to be made on what to do with messages being sent to that vessel.

7.2.1.2 Lower barrier to entry

Our original goal was for Seastorm to not require any installation at all for the user, and to be fully accessible from a single URL in the browser. While it currently only requires the additional step of running a script with Python (which the user already needs to install in order to use Seattle), we still consider it desirable to lower the barrier to entry, especially in an educational setting.

One way of doing this would be to run the Python script at a computer provided by the educational institution, and allowing users to connect to that server instead. In order to make this possible, an alternative to transmitting the user's private key over the network would have to be devised.

Another way would be to make a number of changes to the Seattle platform itself, namely to provide its APIs over HTTP (including CORS). This approach would also require an alternative to exposing the user's private key.

7.2.1.3 Support for other environments

The Seastorm visualizer is already modular: it can be used to generate sequence diagrams from a trace file, no matter how that trace file was produced. If Seastorm is found to be a useful tool for working with the Seattle platform, there is good reason to believe that it would be equally useful in other distributed systems, or other systems based on message passing (such as JavaScript web

workers, or a locally run Erlang program). In this case, it could be desirable to investigate ways of integrating Seastorm with other environments.

We recommend, however, that this only be done after getting positive results from a single environment. Trying to generalize the system before knowing in detail how it should behave would likely lead to duplicate or even wasted effort.

7.2.1.4 Alternate implementation of wrapper library

When seamlessly adding monitoring to Repy programs, Seastorm currently dynamically includes a wrapper library that redefines the functions that are to be monitored. An alternate approach that might be more robust, efficient, modular, or otherwise better is to make use of the security layers [30] available in Repy version 2. A security layer allows users to intercept function calls and potentially perform the same type of monitoring that Seastorm currently does. Because we did not have previous experience with security layers, and because Repy version 2 was still being worked on during the development of Seastorm, we did not look closer at the viability of this approach.

7.2.1.5 Alternatives to Lamport timestamps

Although Lamport timestamps allow Seastorm to correctly order events as required for creating a sequence diagram, there are even simpler methods that provide the same information. We base this on the observation that Seastorm's use of Lamport timestamps is equivalent to vector clocks, due to the fact that events are ordered after-the-fact, when all timestamps from all vessels are available. These timestamps can be converted to vector clocks as follows:

1. For each vessel, a new set of timestamps is created from the existing timestamps such that the ordering is the same but counting starts at 1 and each timestamp differs by exactly one. Figures 7.1 and 7.2 give examples of timestamps in an experiment before and after such a transformation, respectively.

2. Starting from the first event in each process, vector clocks are created for each event in the same way as in a running system, based on the new timestamps acquired in the previous step. Figure 7.3 gives an example of vector clocks acquired in such a way, with values of zero representing the clocks of vessels not yet contacted.

The timestamps created in the first step above are effectively per-vessel sequence numbers, which are easier to implement than Lamport timestamps, as the only operation needed for updating a sequence number is incrementing it. Thus, using sequence numbers instead of Lamport timestamps would simplify the implementation of Seastorm's wrapper library.

On the other hand, sequence numbers would require the Seastorm client to perform more preprocessing when creating sequence diagrams. This is because per-vessel sequence numbers do not capture ordering of events between processes as directly as Lamport timestamps; the client would instead need to create vector clocks as described above and use these to get equivalent information. As an example, consider the first event in vessel C in figure 7.1, which can easily be ordered relative to the first event in vessel A by simply comparing

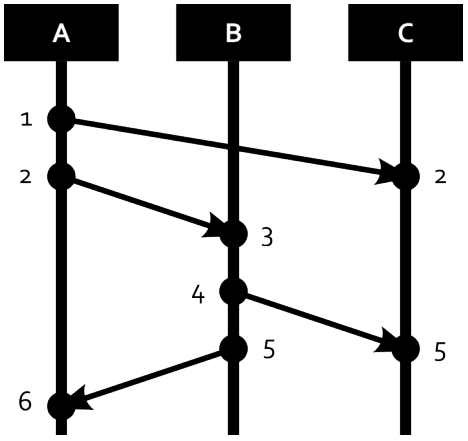

Figure 7.1: Example of Lamport timestamps

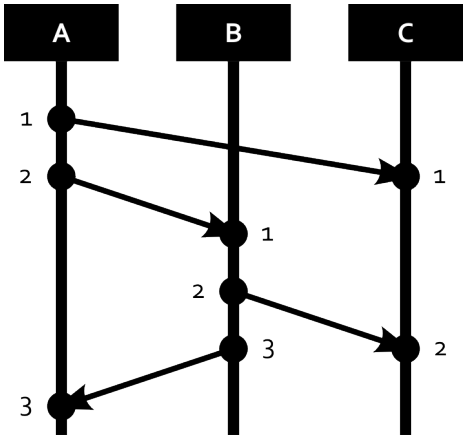

Figure 7.2: Example of sequence numbers acquired from Lamport timestamps

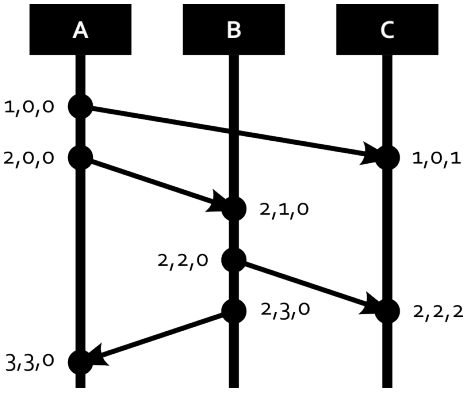

Figure 7.3: Example of vector clocks acquired from sequence numbers

timestamps. By contrast, the same events in figure 7.2 cannot be compared quite as easily.

Simplifying Seastorm's wrapper library could be beneficial in other future work, such as improving monitoring performance or adding support for other environments. In these scenarios, the benefits of doing so would likely outweigh the drawbacks of making the Seastorm client more complex.

7.2.2 Testing

7.2.2.1 Quantitative testing

In order to assess the usefulness of Seastorm for its intended purposes, quantitative testing should be performed. These tests should preferably focus on different use cases, such as understanding existing programs, debugging existing programs, and developing new programs. If Seastorm were to be found more useful in some of these use cases than in others, that would point to aspects of the system that should be improved, given less emphasis, or removed entirely.

7.2.2.2 Adoption

A truly useful system is one that users will adopt and start using on their own rather than only using as part of a user test. We believe that adoption rates should be examined in their own right when evaluating Seastorm, because even systems with a high level of usability may end up not being used in practice. This is especially likely if they involve a complicated setup process or are only useful for a small number of use cases.

This type of testing could be performed by introducing Seastorm at the start of a course in distributed systems (such as that at Chalmers) but not mandating its use. Questionnaires or interviews could then be used at the end of the course to see how many students voluntarily started using Seastorm and how many kept using it throughout the course, as well as finding out their reasons for doing so.

If all other forms of testing were to indicate that Seastorm is useful, but adoption rates among students are low, this would provide a good opportunity for identifying specific features or circumstances that make such tools attractive in practice.

7.2.2.3 Larger tests

The tests that we performed in the course of this project involved only seven participants, each of which only spent 30 minutes using Seastorm (15 minutes for each task). While small tests like these can be useful for providing indications on what to focus on in the future, they are generally not comprehensive enough to draw significant conclusions from.

One important factor that should be increased in future tests is the amount of time that students spend on a single task. Fifteen minutes is not representative of how much time developers spend on a single defect in an unknown program, and the limited amount of time can itself cause the participant to behave differently than normal.

7.2.2.4 Performance testing

If in the future Seastorm were used for larger experiments than the ones tested during this project, performance testing would become much more of a priority. In such experiments, the performance of both the monitor and the visualizer could become relevant. In order to identify and fix potential problems, more rigorous performance testing would be an important first step.

Appendix: User testing material

This appendix includes the material that was provided to users participating in the user testing of Seastorm.

Task 1: Calculator

Description

In this task, you will debug a program simulating a simple calculator that involves one client requesting calculations and another server performing them.

Task

The source code for the program has already been written, but it contains a defect that causes it to behave incorrectly. Your task is to find the defect, correct it, and verify that the program behaves correctly after your fix. The intended behavior of the program is described below.

Note: The program might fail if a message is lost due to congestion or network failure. In this task, you should look for a defect that can occur even when no messages are lost for this reason.

Intended behavior

1. The client sends an "AddOne" message to the server.
2. The server replies with "Value: 1".
3. The client sends another "AddOne" message to the server.
4. The server replies with "Value: 2".
5. The client sends a "Double" message to the server.
6. The server replies with "Value: 4".
7. The final value of `mycontext['number']` on the client is 4.

Files and arguments

One vessel: `calculator-server.repy port`, where:

- `port` is your assigned port number (63173)

One vessel: `calculator-client.repy port server action1 action2 action3`, where:

- `port` is your assigned port number (63173)
- `server` is the IP address of the other vessel
- `action1`, `action2`, `action3` are the requests that the client should send to the server
  - The following three values should be used, in order: `AddOne`, `AddOne`, `Double`

Note: If you are running the program using Seash, make sure to start the server before starting the client, so that the client doesn't start sending requests too early.

Source code

`calculator-server.repy`

```
def on_receive(other_ip, other_port, msg, handle):
  if msg == 'AddOne':
    mycontext['number'] += 1
  elif msg == 'Double':
    mycontext['number'] = mycontext['number'] * 2

  sendmess(
    other_ip,
    mycontext['port'],
    'Value: ' + str(mycontext['number']),
    getmyip(),
    mycontext['port']
  )

if callfunc == 'initialize':
  mycontext['port'] = int(callargs[0])
  mycontext['number'] = 0
  recvmess(getmyip(), mycontext['port'], on_receive)
```

`calculator-client.repy`

```
def on_receive(other_ip, other_port, msg, handle):
  mycontext['number'] = int(msg.split(': ')[1])

  if mycontext['next_action'] == len(mycontext['actions']) - 1:
   stopcomm(handle)
```

```
  if mycontext['next_action'] < len(mycontext['actions']):
    send_action(mycontext['next_action'])
    mycontext['next_action'] += 1

def send_action(index):
  sendmess(
    mycontext['server'],
    mycontext['port'],
    mycontext['actions'][index],
    getmyip(),
    mycontext['port']
  )

def start():
  recvmess(getmyip(), mycontext['port'], on_receive)
  mycontext['next_action'] = 1
  send_action(0)

if callfunc == 'initialize':
  mycontext['port'] = int(callargs[0])
  mycontext['server'] = callargs[1]
  mycontext['actions'] = callargs[2:]
  # Make sure that server has started before sending requests.
  settimer(2, start, [])
```

Task 2: Auction

Description

In this task, you will debug a program simulating an online auction that involves one seller, three bidders, and one item being sold.

Task

The source code for the program has already been written, but it contains a defect that causes it to behave incorrectly. Your task is to find the defect, correct it, and verify that the program behaves correctly after your fix. The intended behavior of the program is described below.

Note: The program might fail if a message is lost due to congestion or network failure. In this task, you should look for a defect that can occur even when no messages are lost for this reason.

Intended behavior

1. The seller sends an "Auction has started" message to each bidder.
2. Each bidder replies with a "Bid: xyz" message, where `xyz` is the amount that the bidder wants to bid (provided as an argument to `auction-bidder.repy`; see below).
3. After 3 seconds, the seller compares the bids that it has received, and:

(a) sends a "Sold!" message to the bidder that offered the highest bid.

(b) sends a "Not sold!" message to every other bidder.

Files and arguments

Three vessels: `auction-bidder.repy port bid`, where:

- `port` is your assigned port number (63173)
- `bid` is the amount that this bidder will bid
  - The following three different values should be used: 500, 600, 800

One vessel: `auction-seller.repy port bidder1 bidder2 bidder3`, where:

- `port` is your assigned port number (63173)
- `bidder1`, `bidder2`, `bidder3` are the IP addresses of the other vessels

Note: If you are running the program using Seash, make sure to start the bidders before starting the seller, so that the seller doesn't start sending messages too early.

Source code

`auction-bidder.repy`

```
def on_receive(other_ip, other_port, msg, handle):
  if msg == 'Auction has started':
    sendmess(
      other_ip,
      other_port,
      'Bid: ' + str(mycontext['bid']),
      getmyip(),
      mycontext['port']
    )
  else:
    stopcomm(handle)

if callfunc == 'initialize':
  mycontext['port'] = int(callargs[0])
  mycontext['bid'] = int(callargs[1])
  recvmess(getmyip(), mycontext['port'], on_receive)
```

`auction-seller.repy`

```
def on_receive_bid(other_ip, other_port, msg, handle):
  amount = int(msg.split(': ')[1])
  mycontext['bids'].append({
    'ip': other_ip,
    'amount': amount
  })
```

```
def start():
  mycontext['handle'] = recvmess(
    getmyip(),
    mycontext['port'],
    on_receive_bid
  )
  for bidder in mycontext['bidders']:
    sendmess(
      bidder,
      mycontext['port'],
      'Auction has started',
      getmyip(),
      mycontext['port']
    )
  settimer(3, send_results, [])

def send_results():
  mycontext['bids'].sort(cmp=sort_bids)

  sendmess(
    mycontext['bids'][0]['ip'],
    mycontext['port'],
    'Sold!',
    getmyip(),
    mycontext['port']
  )

  for bid in mycontext['bids'][:len(mycontext['bids'])-1]:
    sendmess(
      bid['ip'],
      mycontext['port'],
      'Not sold!',
      getmyip(),
      mycontext['port']
    )

  stopcomm(mycontext['handle'])

def sort_bids(x, y):
  return cmp(x['amount'], y['amount'])

if callfunc == 'initialize':
  mycontext['port'] = int(callargs[0])
  mycontext['bidders'] = callargs[1:]
  mycontext['bids'] = []
  # Make sure that bidders have started before sending requests.
  settimer(3, start, [])
```

Appendix: Deployment

Overview

When deploying Seastorm, distribution of two separate applications must be set up: the client and the server.

The client is distributed to the user like other browser applications: from a single entry-point URL that allows the user to run the application by simply visiting the URL in a browser. This URL should be served to the user from some trusted source (preferably over HTTPS), such as the user's educational institution.

The server is distributed to the user from the client, which links to a URL where a compressed folder containing the server can be downloaded. This URL points to a file included in the client application itself; in other words, it is served from the same location as the files that make up the client.

We should note that, in theory, the server could run on another computer than the user's own, and be modified to support serving multiple users at the same time. For security reasons, however, this is not a viable approach: in order to make calls to the Node Manager API, the server must have access to the user's private key, which should not leave the user's computer. Because of this, every user must run a personal instance of the server.

Requirements

- Python 2.6+
- A web server that can serve static files, preferably over HTTPS.

Instructions

On the host that will serve the Seastorm client and server to the users, perform the following steps:

1. Download the Seastorm source code from GitHub: `https://github.com/JakobKallin/Seattle-Seastorm`.
2. In the root directory of the Seastorm source code, run `python build.py output_path cors_origin`, where:
    - `output_path` is a directory served by a web server from this host.

- `cors_origin` is the origin from which the `output_path` directory is served.

3. Instruct your users to visit the host's web server in their browser and follow the instructions given in order to install Seastorm.

Example

In this example, we want to distribute Seastorm from `example.com`. We have set up a web server to serve static files from the directory `/home/jakob/www`.

1. We download the Seastorm source code to `/home/jakob/downloads/seastorm`.
2. We enter `/home/jakob/downloads/seastorm` and run `python build.py /home/jakob/www example.com`.
3. We instruct our users to visit `http://example.com/` in their web browser and install Seastorm by following the provided instructions.

Bibliography

[1] J. Cappos, I. Beschastnikh, A. Krishnamurthy, and T. Anderson, "Seattle: A Platform for Educational Cloud Computing," in *SIGCSE'09*, March 2009.

[2] A. S. Tanenbaum and M. van Steen, *Distributed Systems: Principles and Paradigms*. Pearson Prentice Hall, 2007.

[3] C. D. Hundhausen, S. A. Douglas, and J. T. Stasko, "A Meta-Study of Algorithm Visualization Effectiveness," *Journal of Visual Languages and Computing*, March 2002.

[4] R. Wu, "Visualization as An Aid for Understanding Distributed Algorithms: An Evaluation," Master's thesis, The University of Georgia, 2001.

[5] T. Rajala, M.-J. Laakso, E. Kaila, and T. Salakoski, "Effectiveness of Program Visualization: A Case Study with the ViLLE Tool," *Journal of Information Technology Education*, vol. 7, 2008.

[6] D. Bell, "UML basics: The sequence diagram." `http://www.ibm.com/developerworks/rational/library/3101.html`, February 2004. Accessed: June 19, 2014.

[7] R. Nessa, "Trace Visualisation for distributed State Machines," Master's thesis, Norwegian University of Science and Technology, 2005.

[8] L. E. Karlsen, "Providing a Birds Eye View on the Execution of Distributed, Reactive Systems using Collaborations," Master's thesis, Norwegian University of Science and Technology, 2006.

[9] O. Babaoğlu and K. Marzullo, "Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms," tech. rep., Laboratory for Computer Science, University of Bologna.

[10] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, vol. 21, pp. 558–565, July 1978.

[11] F. Mattern, "Virtual Time and Global States of Distributed Systems," in *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, 1988.

[12] P. A. Ward, "Algorithms for Causal Message Ordering in Distributed Systems." `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.92.1137&rep=rep1&type=pdf`. Accessed: June 19, 2014.

[13] "Repy Programming Guide." https://seattle.poly.edu/wiki/RepyApi. Accessed: June 19, 2014.

[14] "dylink.repy." https://seattle.poly.edu/wiki/SeattleLib/dylink.repy. Accessed: June 19, 2014.

[15] "Node Manager Design Document." https://seattle.poly.edu/wiki/UnderstandingSeattle/NodeManagerDesign. Accessed: June 19, 2014.

[16] "Seattle Clearinghouse." https://seattleclearinghouse.poly.edu/. Accessed: June 19, 2014.

[17] "Same-origin policy." https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy. Accessed: June 19, 2014.

[18] "HTML Canvas 2D Context." http://www.w3.org/TR/2dcontext/, August 2013.

[19] "Scalable Vector Graphics (SVG) 1.1." http://www.w3.org/TR/SVG/, August 2011.

[20] "The JSON Data Interchange Format." http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf, October 2013.

[21] D. Winer, "XML-RPC Specification." http://xmlrpc.scripting.com/spec.html, June 1999. Accessed: June 19, 2014.

[22] F. Osterlind, J. Eriksson, and A. Dunkels, "Demo Abstract: Cooja TimeLine: A Power Visualizer for Sensor Network Simulation," in *SenSys'10*, 2010.

[23] B. Koldehofe, M. Papatriantafilou, and P. Tsigas, "LYDIAN: An Extensible Educational Animation Environment for Distributed Algorithms," *Journal on Educational Resources in Computing*, vol. 6, June 2006.

[24] "Cloud9 IDE." https://c9.io/. Accessed: June 19, 2014.

[25] "Seash: The Seattle Shell." https://seattle.poly.edu/wiki/SeattleShell. Accessed: June 19, 2014.

[26] L. Pühringer, "Try Repy!," Bachelor's thesis, University of Vienna, October 2011.

[27] "Web Workers." http://www.w3.org/TR/workers/, May 2012.

[28] J. Armstrong, *Making reliable distributed systems in the presence of software errors*. PhD thesis, The Royal Institute of Technology, December 2003.

[29] "Dromaeo." https://wiki.mozilla.org/Dromaeo. Accessed: June 19, 2014.

[30] "Writing and Using Custom Security Layers in Repy V2." https://seattle.poly.edu/wiki/RepyV2SecurityLayers. Accessed: June 19, 2014.