Thin Client Collaborative Visualizations Using the Distributed Cloud

by

Matthew Hemmings
B.Sc., University of Victoria, 2014

Supervisory Committee

Dr. U. Stege, Co-Supervisor
(Department of Computer Science)

Dr. Y. Coady, Co-Supervisor
(Department of Computer Science)

Dr. R. McGeer, Departmental Member
(Department of Computer Science)
Supervisory Committee

Dr. U. Stege, Co-Supervisor
(Department of Computer Science)

Dr. Y. Coady, Co-Supervisor
(Department of Computer Science)

Dr. R. McGeer, Departmental Member
(Department of Computer Science)

ABSTRACT

This thesis describes the research, design, implementation, and evaluation of a collaborative visualization system that models large data sets in thin clients using the Lively Web development environment. A thin client is a computing device with light resources, depending heavily on remote computational resources for any large scale data processing. A thin client could be a cellular phone, a tablet or a laptop with insufficient resources to perform heavy computing locally. The applications of this technology form part of a new class of application where large data sets are being visualized and collaborated on with low latency where the users are geographically separated. The primary motivation of this research is to show that large data sets can be viewed and interacted with on any device, regardless of geographic location, in collaboration with other users with no setup required by the user. In addition, it shows the strengths of the Lively Web in developing impressive thin-client visualizations in a flexible, straight-forward manner. For deployment, Lively Web servers are brought up using docker containers on the distributed cloud using virtual machines allocated by the Global Environment for Network Innovations (GENI) and Smart Applications on Virtual Infrastructure (SAVI) networks.
# Contents

[Supervisory Committee](#) ii  
[Abstract](#) iii  
[Table of Contents](#) iv  
[List of Tables](#) vi  
[List of Figures](#) vii  
[List of Listings](#) viii  
[Acknowledgements](#) ix  
[Dedication](#) x  

## 1 Introduction  
1.1 Motivation ........................................... 1  
1.1.1 Examples of Applications .......................... 1  
1.2 Collaboration is not Presentation ...................... 2  
1.3 Solution Design and Approach ........................ 2  

## 2 Architecture  
2.1 What is the Lively Web? ................................ 4  
2.1.1 Interacting with The Lively Web .................... 6  
2.1.2 Scene Graph ........................................ 6  
2.1.3 Version Control .................................... 8  
2.1.4 External Library Support ............................ 8  
2.1.5 Messaging Service .................................. 8  
2.2 What is GENI? ......................................... 12
2.2.1 Why GENI? ................................................................. 12
2.3 Deployment ................................................................. 13
  2.3.1 Performance Requirements ......................................... 13
  2.3.2 Lively Server in Docker Container .............................. 13
  2.3.3 Plugability .............................................................. 13
  2.3.4 Data Server for Pollution Data ................................. 13

3 Methodology ................................................................. 15
  3.1 Cost of Computation vs Communication .......................... 15
    3.1.1 Impact of Cost Disparity ......................................... 15
  3.2 Layered Communication .............................................. 16
    3.2.1 Scalability ............................................................ 16
    3.2.2 Group Namespaces .................................................. 17
    3.2.3 Groups and Partitions ............................................ 17
    3.2.4 Group Messaging ................................................... 18
    3.2.5 User Groups ........................................................ 18
    3.2.6 Group Actions ....................................................... 18

4 Applications and Experimentation .................................... 20
  4.1 Pollution Visualization ............................................... 20
    4.1.1 Experiments and Results ......................................... 23
    4.1.2 Pollution Quadtree Server ....................................... 25
  4.2 NBA Shot Chart Visualization ...................................... 25
    4.2.1 SportVU Cameras and Data Gathering ........................ 26
    4.2.2 Experiments and Results ......................................... 28

5 Evaluation, Analysis and Comparisons .............................. 29
  5.1 Comparisons to Other Messaging Services ...................... 29
    5.1.1 Beanstalkd [28] .................................................... 29
    5.1.2 RabbitMQ [29] ..................................................... 30
    5.1.3 Lively2Lively Group Messaging Protocol ..................... 30

6 Conclusions and Future Work ........................................ 34
  6.1 Summary ............................................................... 34
  6.2 Future Work ........................................................... 35

Bibliography ............................................................... 37
List of Tables

Table 4.1 Latency and Bandwidth Estimates . . . . . . . . . . . . . . . . 24
Table 5.1 Delays in Joining a Group in ms . . . . . . . . . . . . . . . . . 33
Table 5.2 Delays in messages (Round Trip Times) in ms . . . . . . . . 33
List of Figures

Figure 2.1 Deployment Sites ................................................. 5
Figure 2.2 UI Elements ....................................................... 6
Figure 2.3 Ways to interact with a morph .............................. 7
Figure 2.4 A morphic layout and its corresponding scene graph 9
Figure 2.5 A map showing GENI POPs ................................. 12

Figure 4.1 Pollution Visualizer ............................................. 21
Figure 4.2 NBA Shot Chart ................................................. 27
List of Code Samples

2.1 Sample Session .................................................. 11
3.1 Sample Group Sessions ........................................ 19
4.1 Pollution Message ................................................ 22
4.2 Example Shot Data .............................................. 28
4.3 Shot Chart Message ............................................. 28
ACKNOWLEDGEMENTS

I would like to thank:

**Jennifer Hemmings**, for supporting me in the high moments as well as the low, and for helping me keep perspective when things seemed out of control.

**Ulrike Stege, Rick McGeer and Yvonne Coady**, for mentoring, support, encouragement, and patience. Without them, I never would have put this into a coherent design.

**My father**, for providing the inspiration to pursue my dreams, and for showing me that you don’t have to lose yourself in your work. I miss you, dad.

**My mother**, for providing me strength, and emotional support. Thanks for everything you do that I’ll never remember to say.

*It’s kind of fun to do the impossible.*

Walt Disney
DEDICATION

For my father. I miss you every day. Thank you for showing me the many wonders of the world and the curiosity and resourcefulness to find the answers I wanted.
Chapter 1

Introduction

1.1 Motivation

Collaboration tools are necessary to enable users to work together on data to do many tasks, from simple presentations to the more complex research done to acquire new insight from the data in front of them. As data sets grow in size, they become problematic to distribute wholesale across the internet. There are systems in place that are built with the express purpose of allowing users to collaborate on large sets of data over large areas, but these tend to be both expensive and very specialized.

When the internet was first designed, it was done primarily as a tool to transport data from one place to another, but has grown instead to be a ubiquitous component of our lives. The browser, present on almost every computing device with a connection to the internet, offers a unique opportunity for us to develop tools that can be made available to a larger audience than ever before and that require very little or no setup to be made use of.

1.1.1 Examples of Applications

There are, of course, many applications that could benefit from this type of approach. In this thesis, we will see two examples. The first is a geographical dataset that models the concentration of airborne pollution data over the land areas of the planet using a large dataset comprising 16 years of data, on a monthly time scale. The second
uses sports statistics in the form of basketball data to model where players are most successful in scoring.

These examples do not, however, form an exhaustive list and in future there are plans to model epidemiological data focusing on the spread and containment of diseases and health care information showing mortality rates, both with the ability to be manipulated live and be made visible to all other participants in a collaboration session.

1.2 Collaboration is not Presentation

There are numerous tools available to present data from a single point to many other viewers, from Skype[43] to online presentation tools like Prezi[16]. These allow you to present your information either over a voice-over-ip (VOIP) call or a web service from a single point to many others. These work well when the information you are presenting needs only to be expressed by a single person and when the information flow is one-way, that is from presenter to listeners. The limitation becomes apparent, however, if there needs to be manipulation to the data. Instead of being able to manipulate it directly, in a presentation situation, you are forced to ask the presenter to make any changes you would like to see and wait to see what happens. This can be frustrating, especially if the interruptions are frequent, complex, or both. Google docs can allow this sort of interaction, but only in a limited scope, whether that be text only (docs) or spreadsheet oriented (sheets). There is, however, a need for more complex visualizations sometimes and to make these collaboratively interactive is the focus of this work.

1.3 Solution Design and Approach

To accommodate the idea of making a low resource collaboration system that would work on any device that has a web browser, we chose the Lively Web [30] as a development environment. This is an online Integrated Development Environment (IDE) that uses primarily JavaScript as the language being developed in. In this thesis, we will first introduce the Lively Web, and explain how it is used.

To accommodate large scale distribution, necessary for reasons expressed in detail in
Section 2.2, the Global Environment for Network Innovations (GENI) platform was chosen. This platform allows for rapid deployment of virtual machines over much of the area of the United States, and with partners around the world, easy access to many global sites. Since the Lively Web server can be deployed as a container-based service on any of these nodes, it was both easy and fast to deploy them in many places.

Finally, having chosen a development tool, it was necessary to design a protocol for communications, layered on top of the existing web socket-based communication present in the Lively Web, called Lively2Lively. This service allowed collaboration messages to be sent to other users easily and also allowed for large numbers of users, possibly all using the same application, to have distinct sessions running simultaneously and without interference from each other. This protocol is described in detail and allows for future expansion.
Chapter 2

Architecture

The first challenge undertaken when a solution to a problem is designed is to choose an appropriate tool for the task at hand. In the case of this project, it is separated into two distinct tools. For creating a compelling and light weight visualization, the Lively Web development environment was used. To take advantage of the distributed cloud, we need at least one provider of resources. This project took advantage of two:

1. Global Environment for Network Innovations (GENI) [3]
2. Smart Applications on Virtual Infrastructure (SAVI) [11]

Both GENI and SAVI are environments where it is possible to request a virtual machine (VM) upon which you are able to deploy applications for testing. In this chapter, we will discuss the GENI research platform as well as give brief introductions to their usage. SAVI operates in a very similar fashion to GENI, creating virtual machines for use in research at disparate locations across Canada, so we will not discuss it in detail here.

2.1 What is the Lively Web?

The Lively Web is a web-based open source development environment. It is built using JavaScript and requires no tool installations other than a web browser that supports JavaScript, like Chrome, to use and develop applications in [30]. The Lively Web abstracts away the HTML and CSS that are normally part of developing a web application [31] leaving the user to interact only with JavaScript and direct manipulation of the Morphic environment [36]. To access the Lively Web development
Figure 2.1: Deployment Sites
environment, one needs only open the Lively Web home page [5]. JavaScript being a part of any modern browser, and the development taking place entirely in it, means that development is independent of platform. Being stored in the cloud, work on a project can be started, saved, and resumed anywhere.

2.1.1 Interacting with The Lively Web

The Lively Web, from the perspective of a developer, is made up of a container called “world”. In the world, you are able to create objects, called morphs [36]. A user is able to interact with a morph either programatically or interactively; including their creation. To work interactively with a morph, the halo, shown in Figure 2.2a is used. The icons floating around the morph show the different ways to interact with it.

Figure 2.3a shows a list of the properties of the morph [30]. Figure 2.3b indicates how to interact with the associated morph.

2.1.2 Scene Graph

Each morph has exactly one owner morph, forming a parent-child relationship. The world morph, which has no owner, is the only exception to this rule. This creates
(a) An object inspector

(b) An object editor

Figure 2.3: Ways to interact with a morph
a graph with the world as root, as shown in the example in Figure 2.4, meaning all morphs have the world morph as one of their ancestors. A morph has knowledge of both its owner and its children. By knowing these relationships, it is possible for a morph to find any other in the underlying tree structure.

2.1.3 Version Control

The Lively Web is based on a wiki, and maintains versions of any world, every time you save it. Stepping back to a previous version is done directly through the menu system. If a project stops working, it is easy to check previous versions and visit them without having to change the world in place, so you can determine which rollback point to move to. It is important to note that each of these previous versioned worlds is, itself, a world and is fully interactive and functional.

2.1.4 External Library Support

The Lively Web allows a user the ability to import any JavaScript library they choose and use it within the context of their world. For this project, Lively did not have any mapping software built in that would meet the needs of the pollution visualization project, so Polymaps was used [9]. Polymaps is a JavaScript library that is designed to allow the user to make interactive map-based visualizations and experiences in a modern web browser. Other applications based on the same framework need not be map-based, any libraries necessary can be used.

2.1.5 Messaging Service

Lively contains a messaging service based on WebSockets called “Lively2Lively”, which can be used to pass messages of any kind from world to world and user to user. These can even be used to invoke remote functions. This was done in this project to, among other things, update remote users of the map state and other contextually necessary information.

The advantages of using the Lively2Lively messaging service instead of using raw web sockets or creating your own communication protocol are that, because this service is abstracted, the underlying connection methods are invisible and the invocation is all that is left to the developer. Further, it is built into Lively itself, so each world
(a) An example of morphs laid out in a Rectangle

(b) The corresponding Scene Graph

Figure 2.4: A morphic layout and its corresponding scene graph
has access to it by default. The application needs only be told:

1. The user or group of users that the messages need to be sent to
2. The type of message being sent
3. Arguments for the receiver

These items are bundled up in a simple JavaScript Object Notation (JSON) structure and then sent to the Lively2Lively protocol for delivery. By default, received messages are not necessarily understood by the recipient, the world must register a handler for any message types it is to handle. The Lively2Lively service assigns a session GUID to each world as it contacts the central tracker and each session connected to the tracker can be queried. The changes made to facilitate the creation of more applications will be discussed in more detail in Chapter 3.

Listing 2.1 is a sample set of sessions information:
There are many ways to filter this information such that you only send messages to intended recipients and to avoid blanket message sending. Later in this paper, a means of group management is introduced that simplifies this process.
2.2 What is GENI?

The Global Environment for Network Innovations (GENI) is a network of 50 points-of-presence (POPs) across the United States. Agreements with partners give at least an additional 50 worldwide.

2.2.1 Why GENI?

GENI offers not only access to virtual machines on which to conduct experiments and deploy programs, but also the ability to easily deploy a site near a user, wherever that user might be. This is important for the type of application discussed here, as there is a requirement for large data sets to be hosted near the home. Glenn Ricart \[41\] refers to this as locavore computing, that is, one who consumes computation locally whenever possible to improve user experience. Since, for our research, there is a very real need for local computation to have high local bandwidth and simultaneously low latency, GENI is a very good solution that meets these needs.

Figure 2.5: A map showing GENI POPs
2.3 Deployment

2.3.1 Performance Requirements

Studies have shown that if an application does not respond in 150 ms, users begin to turn away [46]. Because our intent is for a thin-client device such as a laptop, cell phone or tablet to be able to display and interact with the data, the user could be anywhere. Since the user can be anywhere and there must be a database near the user, the data must be available to be brought up everywhere with very little advance notice.

2.3.2 Lively Server in Docker Container

A docker container is pulled down from Docker Hub [13], which contains all the elements required to run the Lively Web server. This container does not actually contain the Lively Web server itself, just the file system to support it. The Lively server is pulled from GitHub [6] and mounted as an external volume to the container and a port forwarded from the host machine to the container to allow external access.

2.3.3 Plugability

In the case of the Pollution Visualizer presented in Section [4.1], the server can be replaced by any other server reachable in the same fashion as the existing flask server that accepts a bounding box and returns the data points in the same form as the existing server, allowing it to be replaced with a different data server if performance needs change. The visualizer can be replaced by anything that sends a bounding box and expects points returned in an array, allowing another application to be built that interacts with the same data in another way. Both could be replaced entirely, of course, and have been in other applications that will be discussed in detail in Chapter [4].

2.3.4 Data Server for Pollution Data

A python flask [2] server is started on the same host machine as the lively server on a different port to minimize transit distance between users and the data they are requesting. The server itself uses a Quad Tree to locate the data that is stored in this
format: (latitude, longitude, pm2.5 concentration)
That is, the server maintains a position and a measure of the concentration of particulate matter in the atmosphere with size 2.5 micrometers or smaller \[33\] in multiple files on disk that represent the data found on the leaves of the Quad Tree structure. This will be discussed in more detail in Section \[4.1.1\].
Chapter 3

Methodology

When beginning to design a visualization or application, it is important to look at the size of the data versus the size of the program. We will examine how changes in cost for both communication and computation have influenced the landscape of program design and how, in turn, this means that our approaches to solving problems need to adapt as well. In addition, we will describe how we layered additional functionality on top of the Lively2Lively communication protocol to expand the possibilities of collaborations.

3.1 Cost of Computation vs Communication

When the Internet was designed, one of the primary design principles was to facilitate data transfer [22]. The costs associated with computation were quite high [14] in the earliest days of the internet, so it became common practice to ship data to programs. However, as data sets grew in size, this became problematic. Soon, it was necessary to re-examine what was being sent over the wire. While computation costs have dropped dramatically, the corresponding costs of communication decreased much more slowly [1]. Because of this, we have begun sending programs to where data is and acting upon it locally instead of the reverse.

3.1.1 Impact of Cost Disparity

As mentioned, because the disparity in cost has increased so dramatically between computation and communication, it now makes sense to take advantage of remote computation and put programs in the same place as the data. This has changed the
way solutions are being designed to allow for smaller programs that can more quickly and cheaply be shipped near to the data sets and return only what is absolutely required. In the case of our Pollution Visualizer, we explore a different approach designed to reduce latency while accepting that a new site setup will take some time. When a new site is needed, both the data and program are shipped as close as possible to the new users to minimize transit time of all data.

3.2 Layered Communication

The existing Lively2Lively [5] protocol allows transmission of messages of various types from session to session, and this is indeed the method by which the Pollution Visualizer communicates. Upon loading, the world registers an action listener so that when an incoming message is received, it will know what to do with the data contained therein. Then a stepping (repeating) service is started that, every iteration interval, contacts the central tracker, gets a list of all current sessions and then filters them for the recipients desired. Once this list of sessions is obtained, the message is sent out to all of them and each world updates accordingly, in whatever way it was designed. The worlds need not necessarily mimic the behavior of the sender.

3.2.1 Scalability

The existing communication protocol did, however, bring to light a degree of inefficiency and lacking functionality to expand beyond the single application. With a large number of sessions on a large number of Lively servers, the time taken to filter the list of sessions every time a message is sent becomes unwieldy. This is because the current model is to request from the registration tracker a list of all connected sessions and look through it for sessions that match intended recipients. If the list were large it could take an extremely long time to filter intended recipients from it. In addition, the application would need to know, in advance, what type of filtering would be needed, limiting the flexibility of the application by not allowing expansion to the application without modifying it first. It would be better to have a list of targets to send to that could be joined by interested parties, regardless of their source, to allow for many uses of the same data being sent.

To better handle this problem, we created and layered a second tracker on top of
the first that will maintain lists of users that are taking part in various collaborative activities. This tracker handles all the maintenance that goes along with making sure connections remain up to date and, when a connection terminates, updating accordingly. We call this service “Group Tracking”. To receive messages from a group, a user needs to send a request to the tracker with the name of the group they wish to join. Upon receipt, the tracker will check if a group by that name already exists. If it does, the user is added to the group user list. If it does not exist, a group is created and the user added to it.

In addition to this, simplifying the behavior of a collaborative application by allowing a tracker that handles most of the work outside of the original world speeds things up by having pre-built lists that would not require filtering before messages to be sent. Filtering can, of course, still be done based on user roles, url of the world, or any other number of parameters, but the agreement in joining a group was that you expect to receive messages from them.

### 3.2.2 Group Namespaces

When considering namespaces for the group, scalability again was a concern. The more users decide to use the collaborative service, of course, the higher the risk of collisions when attempting to create a new group. To prevent this, the namespace standard is modelled after the DNS standard [38] from 1987. As shown in Listing 3.1, the group is defined in a tree hierarchy in the format: username.groupname.partitionname. In the example, this would be MattH.shotchart.default.

### 3.2.3 Groups and Partitions

A user is free to have any number of groups that they are responsible for. They can easily create a new one for every application that they intend to run collaboratively by making a call to the group tracker. There is, however, no need to create a second group for the same application, despite the desire existing to have two distinct groups of people interacting on the same application at the same time. Instead, the second group of users can be distinguished by partitioning the original group. For instance, if two groups of people are trying to discuss pollution data at the same time without interfering with each other, it is possible for a single user to create a single group with two partitions in it, one partition for each group of users.
3.2.4 Group Messaging

As mentioned in Section 3.2.1, being a member of a group indicates your willingness to receive messages from this group. The messages sent, however, are deliberately not instructions on what to do, but rather what was done. For example, if the color of a morph is changed in one world, a message could be dispatched announcing that the color was changed and what it was changed to. Another user would be free, however, to do whatever they chose with the information once it was received. You could, instead of changing a matching morph, plot the point on a color picker, for instance, or create a text string indicating the RGB values. It was important in the design of the group service that no instructions be sent, because I wanted the user to be able to build two distinct applications using the same messages, instead of having to create a second application. This method also allows for non-lively services, such as a session running on a python connection, to connect to the group tracker and receive the data being sent as well. If the group protocol was changed slightly to recognise non-lively connections and deal with them accordingly, there would further be no need to alter the lively2lively messaging service. This could easily be included in the abstracted group communication layer.

3.2.5 User Groups

Currently, there are two groups of users in the Group Service. One is the administrator of the group, and the other is the user. The intent behind the user groups is to allow there to be control of the flow of messaging and other activities such as limiting joins to only people approved by an administrator or higher tier user. Currently, the administrator is reassigned to the next oldest user in the group if he or she departs, or they are able to abdicate responsibility to another user of their choosing. In future, there will be support for any number of tiers of user per group, determined by the administrator, and each of them will have their own set of actions they are able to make.

3.2.6 Group Actions

In addition to groups having user tiers, they also have action lists that are approved. When a message is sent to the group, it is checked against the list of approved actions and, if it is not on the supported list, a response will be fed back to the requesting
user indicating that the action is not allowed. Administrators control whether or not an action is on the list of approved actions. To register an action, a request is made by the administrator to the Group Tracker, indicating the action in question and the group it should be added to. If the action is not already in the list of actions for that group, it is added and the administrator notified. If it is a duplicate request, the administrator is warned and no action is taken.

Listing 3.1: Sample Group Sessions

```json
{
    ConnectedSessions: {
        Matt: {
            shotchart: {
                default: {
                    users: [
                        "client-session: 4845D329-2AC1-41EE-A594-E4EFC58100AC",
                        "client-session: 6DDB3FFB-8949-40B3-BB1E-7B1DE31BFC55"
                    ],
                    actions: ["setList"],
                    admin: "client-session: 4845D329-2AC1-41EE-A594-E4EFC58100AC"
                }
            }
        }
    }
}
```
Chapter 4

Applications and Experimentation

This chapter discusses two collaborative visualizations built on top of the Lively2Lively group messaging protocol and the GENI infrastructure.

4.1 Pollution Visualization

The pollution visualizer was published in "The Ignite Distributed Collaborative Scientific Visualization System" [19] and in The Geni book [26] and was demonstrated live at the Geni Engineering Conference [27] in both plenary and in the demonstration hall.

This visualization, depicted in Figure 4.1, is constructed of a type where there is a large data set stored locally. In this case, a pollution database, created from information gleaned from several sources by Dr. David Lary [33]. The database stores a series of triples containing the latitude, longitude and concentration of pollution every 10 km across the land area of the planet. A snapshot exists for each month from 1997 to 2015, resulting in a database several gigabytes in size.

Because of the structure of the data described in Section 2.3.4 the data for each interaction is so large that sending it across the wire directly is impractical. At roughly 27 bytes per point, and a single interaction using up to 30,000 points, this is approximately 6.4 megabits. It is too large to send quickly enough to meet the established requirements of 150 ms [46] very far, so we were forced to find an alternate solution. Instead, we fit the group messaging protocol described in Section 3.2 As
Figure 4.1: Pollution Visualizer
we discussed in Section 3.2.4, the payload of the messages sent by the Lively2Lively group messaging protocol are, instead of the results of the database query, the query itself. We, in effect, ship the program to the data. The user interaction is reported as shown in Listing 4.1

Listing 4.1: Pollution Message

```json
{ "action": "changeMap",  
  "data": {  
    "user": "MattH",  
    "zoom": "2.2",  
    "opacity": "0.5",  
    "resolution": 10,  
    "center": {  
      "lon": 388.63963305359715,  
      "lat": 23.761342425080898  
    },  
    "maxVal": "40",  
    "month": 9,  
    "year": 2004,  
    "monthUpdate": true  
  },  
  "sender": "client−session: FE3CA50C−DE19−412F−8B66−3217DC5E7B09",  
  "target": "client−session: D9DBC085−12E9−42D1−9945−C6C78D94D0F",  
  "messageId": "client−msg: 95209C36−6C21−4501−8767−451D6EB76684",  
  "messageIndex": 17}
```

The only information sent across the wire is that contained in the data field of the message and the name in the action field. The wrapping surrounding the data is the Lively2Lively protocol information, containing who sent the message, and what the message index was on the server. When a user performs any sort of UI action, such as moving the map, zooming in or out, changing the maximum pollution density, changing the time being viewed or even the opacity of the points being drawn, a message containing all this information is sent to all other members of the collaborative group. When the user receives it, this application is written by default to duplicate the action taken by the triggering user, but there is nothing preventing you from taking any action you choose with the information being given by adding instructions to the message receipt script.

The application has a pre-registered action that listens for messages of the type "changeMap" and passes the data portion along to it. The action is passed to the same script that is run when a change is processed locally, including all the same information. In this way, if a change is made to the changeMap script, the only change that needs to be made is to ensure that the data sent across the wire is of the same format as that expected from a local change.

Critically, this application was built to meet the metric of having an action be received, looked up and rendered in 150ms, as this has been established as the amount
of time before users begin to turn away [46].

4.1.1 Experiments and Results

Measurements were taken regarding how many points Polymaps [9] was able to render in 100ms, and the result of 30,000 points was obtained. With an eye towards limiting seek time to retrieve the data, different solutions for data management were examined. The standard SQL and NoSQL databases were attempted, beginning with SQLite[12], MySQL[8], Postresql[10], and MongoDB[7]. MongoDB, specifically, was evaluated first alone, and then with both the postgis and other gis extensions, but this resulted in a very large database on disk. Since we were looking for a total time of 150 ms, and 100 of that was already devoted to drawing points on the map and we also had to be mindful of the transmission time (shown in Table 4.1), we began looking for data structure solutions fast enough to serve the points in a period of time less than 25ms. In addition, the constraints imposed because of the use of a cloud service needed to be considered. This imposed a limitation of 4GB of physical memory and limited disk space. Precise disk space varied depending on the cloud service (primarily GENI [3], but also iMinds [4] in Europe, and SAVI [11] in Canada), but it was never larger than 40GB. This meant that we were unable to load the entire dataset into an in-memory SQLite database, since it was larger than the physical memory provided and were also unable to properly index a large geospatial database, as the indexing caused unreasonable growth in disk space. Since none of the out-of-the-box solutions were working, a QuadTree solution was devised, as this is one of the recommended data structures when using geospatial data [37]. In the case of our current application, the use of the QuadTree resulted in the existence of approximately 600,000 of files on disk. While not occupying an extraordinary volume of disk, 9GB of data is still used. Together, these results indicate there is room for improvement, as mentioned in Section 6.2.

This same number of points could be returned from the QuadTree search in approximately 20ms. Since network latency is the last limiting factor, it was necessary to determine how much data we could ship from the database to a client in four situations, shown in Table 4.1.
<table>
<thead>
<tr>
<th>Scenario</th>
<th>Latency</th>
<th>Estimated Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Campus</td>
<td>1 ms</td>
<td>1 Gb/s</td>
</tr>
<tr>
<td>City</td>
<td>5 ms</td>
<td>1 Gb/s</td>
</tr>
<tr>
<td>Continent</td>
<td>50 ms</td>
<td>100 Mb/s</td>
</tr>
<tr>
<td>World</td>
<td>250 ms</td>
<td>100 Mb/s</td>
</tr>
</tbody>
</table>

Table 4.1: Latency and Bandwidth Estimates

The various scenarios are explained here. “Campus” assumes that the user is accessing a server on the same campus where the viewer is located and gigabit bandwidth present. “City” assumes a server within approximately 100 km, but not sharing campus internet with the viewer. “Continent” assumes that there are one or two servers per continent with the viewer connecting to the one with the lower latency, similar in style to Amazon’s Elastic Cloud service. Finally, “World” assumes that there is a single server for global use and that everyone who sends a data request is sending it to the same one.

As you can see, with the expectation of 20 ms for a round trip from the request to the receipt of data and at 27 bytes / point, 30,000 points gives approximately 810 kilobytes, or 6.48 megabits. With this data, combined with the latency and bandwidth experiments, it is clear that the only two viable scenarios are “Campus” and “City”.

In order to allow for this type of connectivity to anyone in the world, there must be a server available within 100km of anywhere. To approximate this on a continent scale, the GENI infrastructure, mentioned in Section 2.2 and whose POPs (shown in Figure 2.5) covers the majority of the continent and can quickly be used to bring up a new site at any of them. The result of the combination of the GENI infrastructure, the Lively Web and the Group Messaging Protocol, described above, result in an application that feels like a desktop-application-like application on a thin client, collaboratively, anywhere. That is, an application written in its own ecosystem using larger resources.
4.1.2 Pollution Quadtree Server

For this visualization, a Quadtree \cite{42} is being used to store the data points. The map is divided into different sized grids, which we call resolutions. The resolutions chosen are to limit the number of points being drawn at a given time because, as we discussed in Section 4.1.1, we are restricted to rendering 30,000 points on each query. The resolutions are at a point density of one per 10km\(^2\), 25km\(^2\), 50km\(^2\), and 100km\(^2\). In this way, depending on the zoom level, the application will search different Quadtrees looking for the correct leaf. Each leaf contains the list of points in a given square.

In a Quadtree, to locate to the correct leaf, the Quadtree takes a bounding box given to it by the Pollution Visualization application in the form of lat/lon pairs of the northwest and southeast points and begins to traverse the tree. Since, in a Quadtree, each node has exactly four children, it lends itself nicely to geographical data: Each child can be thought of as north, south, east and west. When looking for a point in the tree, you begin in the center, and determine if your sought point is north, south, east or west of that point. Then do the same with the new region mapped out by that quadrant. This pattern continues until you reach the appropriate leaf, where you return the value stored there. Since a bounding box is a known size, it is broken into pieces of the size of the appropriate resolution and then searched for in the Quadtree, returning all results as points to be rendered by the mapping software.

This particular visualizer has a server running in Flask \cite{2} on a separate port, but on the same machine as the visualization, so network transit time between the visualization and the database is minimized. When a bounding box is sent, the world makes a request from the server on behalf of the user and renders the points it receives.

4.2 NBA Shot Chart Visualization

The second application constructed based on our framework is an example of visualizing sports statistics. In the National Basketball Association (NBA) \cite{17}, one measure of determining efficacy of a player is to look at their made and missed shots. With a high enough sample set, it becomes apparent that certain players have hot zones
where they shoot from more effectively than other spots on the floor. When choosing a defensive strategy to counter these players, hot zones are one of the primary things taken into consideration. After all, there is no use spreading your defense thinly to guard a player who has difficulty with long shots in preference of guarding closer to the basket where more shots are made.

In Figure 4.2, you can see an application built precisely for this purpose. It has varying ways of narrowing down the selections available, but in the final box, a selection of a player’s name will display on the chart above their made and missed shots. This data is not stored locally, although we have proven in the previous experiment that such a thing is possible. In future, this application will be built to have the same style of setup that the Pollution Visualization in Section 4.1 where there is a database replicated at each site to reduce the potential load on the NBA stats servers. Each night, when the last game of the night is finished, a script will be set up to gather the data from the day, format it correctly and insert it into the databases at all the sites. For now, however, this data is queried live from the NBA stats API [34].

4.2.1 SportVU Cameras and Data Gathering

Beginning in the 2013-14 NBA season, the NBA began installing SportVU [35] cameras, 6 per arena, in the catwalks. These cameras each record an image every 40 milliseconds, feed this data into specialized software and then made available to various applications both in and out of the arena. This data is available throughout the game and historically.

The data returned from a query is, by default, a JSON object containing all the shot detail for a requested player. In Listing 4.2 you can see an example of the shot detail available and what information is contained for each shot, in this case a hook shot made by Tim Duncan. Each shot object has a collection of properties, but in this case only three are used. The co-ordinate position fields and the field that remarks on whether the shot was made or missed. The x, y co-ordinate pairs are fed to the visualization chart along with the made or missed flag and each is rendered on the shot chart as a scatter plot.
Figure 4.2: NBA Shot Chart
4.2.2 Experiments and Results

In this application, the messages sent, as shown in Listing 4.3, are extremely small. The action of clicking a list box item sends the name of the box clicked and the index of the item to all collaborators. Each of them separately sends a query to the NBA statistics API for their own data and renders it appropriately. Since the NBA has cloud hosted their statistics using Akamai, the localization is handled for us (15 measured GENI sites gave an average speed of 0.083s for request, processing on the server side and download of the data), so for this application, there is only a thin client for distribution to the nearest site to the user, without the large set of data. The visualization in question, using a JavaScript library called Data-Driven Documents (d3) [20], is able to render 600 points in 9 ms, or 67 points/ms, so with a remaining 70ms before hitting the desired threshold, we could conceivably render 4690 shots, more than enough to handle a season for any player on record. At scale, of course, to minimize impact to the NBA statistics servers, the data set could be created and manipulated in the same was as the Pollution Visualizer in Section 4.1.

Listing 4.2: Example Shot Data

```json
"Shot Chart Detail",
"0021500013",
21,
1495,
"Tim Duncan",
1610612759,
"San Antonio Spurs",
1,
9,
59,
Made Shot",
"Hook Shot",
"2PT Field Goal",
"Restricted Area",
"Center(C)",
"Less Than 8 ft.",
3,
-1,
39,
1,
1
```

Listing 4.3: Shot Chart Message

```json
{"action":"setList",
 "data":
  {"listname":"List2",
   "idx":4},
 "sender":"client-sessions:DDIDE9CA-FAB3-4AB5-ADE6-7C545F0483B8",
 "target":"client-sessions:525228B1-EC8A-41DE-8AF8-D9ED18FBAE31",
 "messageId":"client-msg:1BEBD4D4-293A-4229-A642-APC87DAAP405",
 "messageIndex":4}
```
Chapter 5

Evaluation, Analysis and Comparisons

5.1 Comparisons to Other Messaging Services

In order to support the collaborative visualizations discussed in this thesis, the proper messaging protocol is an integral part of the design of the system. It is important to choose a protocol that handles security, distribution to a list of collaborators and recovers from crashes and failures gracefully.

There are, of course, other means by which messages can be passed between clients. Some use servers and queues, others direct messaging. We will compare our messaging protocol to selected important and similar products such as Beanstalkd [28] and RabbitMQ [29] and outline how the Lively2Lively group messaging protocol differs from them and what advantages and disadvantages are present in each of the systems.

5.1.1 Beanstalkd [28]

Beanstalkd is ideally suited to be a work queue, maintaining the order of actions received. It is assumed to live in a single tenant environment, where there is no security present in terms of restricting access to either push or pull items from the work queue. Since it lacks the functionality of a push service and, by default, only maintains a single queue per server, clients accessing the jobs must periodically check to see if a new message is present. Further, in order for a single client to deal with multiple messages, the first one must be complete and removed before the second one
is accessible. Since the queue contains all messages, client filtering is also not done by the sender. It would be up to a receiving client to determine if the message was meant for them or not.

Since there is no default behavior in Beanstalkd for a message to be sent to multiple clients, there would need to be bookkeeping on the end of the application to regulate the messages as to how and when they are to act on it. At scale, this could prove to be a problem, since if there were two groups interacting, either they would need to each have their own Beanstalkd queue or they would need to wait until their turn to retrieve a message and act on it.

5.1.2 RabbitMQ [29]

RabbitMQ is an open source middleware program focused on message brokering and which implements the Advanced Message Queueing Protocol (AMQP) [44]. Out of the box, RabbitMQ handles redundancy and failover, resulting in high availability. Unlike Beanstalkd, it is designed to be hosted in a multi-tenant environment and supports multiple queues and types of queue, specifically four types: direct, fanout, topic, and headers. The messaging protocol, AMQP, requires very specific and sometimes complex message construction and semantics.

Unlike Beanstalkd, RabbitMQ supports message pushing and has multiple levels of security. Because it was built on top of the Erlang Open Telecom Platform (OTP) [24], it will run on any platform where Erlang OTP itself will run, so the spectrum is quite broad. Also, since the service is built on top of a standard protocol (AMQP), it is possible to change the RabbitMQ service for any other service that also implements the same protocol, with only the possibility of minor API changes.

5.1.3 Lively2Lively Group Messaging Protocol

The protocol discussed in this subsection was accepted to the ITC 2016 publication [25].

Since neither of these fit the needs of the program or environment, I created a new messaging protocol on top of the Lively2Lively protocol built into the Lively Web service. This service needed to have some of the benefits provided by the other protocols discussed above.
To maintain security, the group protocol has a permission system in place where, in order to join a group, a message is sent to the administrator of that group asking for permission. Once granted, the user is added to the group. In addition, only certain actions are permitted in a group setting in this service, so they must be registered. These actions can only be added by an administrator of the group. When a message is sent, it is checked against the allowed messages and only sent if it passes the check.

Because it was layered on top of the Lively2Lively protocol, it is extremely plug-gable. So long as the underlying service supports the same format of API, no change would be necessary to a collaborative application built in Lively. This allows the protocol underneath to be changed and the application itself won’t notice.

The group tracker is set up as a node.js subserver built in the Lively Web environment. Because the subservers are loaded when the server starts and reloaded when either they or the server crash, failures, when they occur, are recovered very quickly. The group tracker maintains a list of all currently connected sessions, by default, broken up into a JSON structure with the same heirarchy mentioned in Listing 3.1. This can be removed later or placed behind an additional layer of security if required, but during development and testing, it is useful to have.

Although written for use in the Lively Web, the intent with this communication protocol is that it be usable in other environments as well. As such, it was constructed in such a way as to be an expansion on the existing communication protocol. Without Lively2Lively, it can be retooled very quickly to be a tracker and communication abstraction usable by another environment that has existing client to client communication.

Listing 4.1 shows a sample communication sent using the group messaging protocol. Without considering the payload of the message (stored in “data”), this is approximately 250 bytes in size. Depending on the body of the message sent, the amount this will grow varies. In the case of the pollution application, the bounding box is sent along with the text values stored in a few fields, whereas in the shot chart, the value sent is only a string for the name of a listbox and the index of an item selected in it.
When a request to join a group is made, there is a delay between the request being sent and the time messages will be delivered to the new participant. To determine this time, we measured the round trip time between a request being made and the response being delivered from the server. This was measured to be an average of 36.1 ms (Shown in Table 5.1), making the estimated one-way trip to be 18.05 ms, since the messages will be delivered when the tracker receives the messages, not when they are returned to the user.

To test message delays, we measured communication between two collaborators using messages that contained no information other than the timing. This was done between one collaborator in Victoria, BC and two in San Francisco, USA through a central tracker at www.lively-web.org. These results are stored in Table 5.2. To show how this increased the delay, a ping was measured from each participant to www.lively-web.org and added together to show the estimated time of a message without the protocol being used. Once again, underscoring that this is round trip time being displayed, the one way time from participant to participant is less than 100 ms when sending collaboration messages between users. In the Lively2Lively protocol, messages are sent across web sockets one at a time instead of batched, so this likely increases the delivery time. In other protocols, this will vary.

Since the protocol is designed to be layered on top of another type of communication, the tracker’s job is to maintain a list of participants to be messaged directly by the communication layer beneath. This is done so that after the original join message is done, subsequent communications sent are done without additional baggage added to the message. The tracker can be asked directly for a list of participants and then each of them messaged directly or singly, as needed by the specific application. The result of this is that it does not slow down communication once a group is joined and participants are requested.
Table 5.1: Delays in Joining a Group in ms

<table>
<thead>
<tr>
<th></th>
<th>Round Trip</th>
<th>One Way</th>
</tr>
</thead>
<tbody>
<tr>
<td>37</td>
<td>18.5</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>16.5</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>17.5</td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>23.5</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>18.0</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>18.0</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>17.5</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>16.5</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>18.0</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>16.5</td>
<td></td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>36.1</strong></td>
<td><strong>18.05</strong></td>
</tr>
</tbody>
</table>

Table 5.2: Delays in messages (Round Trip Times) in ms

<table>
<thead>
<tr>
<th></th>
<th>Participant 1</th>
<th>Participant 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>108</td>
<td>133</td>
<td></td>
</tr>
<tr>
<td>125</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>169</td>
<td>120</td>
<td></td>
</tr>
<tr>
<td>189</td>
<td>114</td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>101</td>
<td></td>
</tr>
<tr>
<td>192</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>152</td>
<td>152</td>
<td></td>
</tr>
<tr>
<td>189</td>
<td>117</td>
<td></td>
</tr>
<tr>
<td>175</td>
<td>114</td>
<td></td>
</tr>
<tr>
<td>108</td>
<td>196</td>
<td></td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>151.8</strong></td>
<td><strong>130.3</strong></td>
</tr>
</tbody>
</table>
Chapter 6

Conclusions and Future Work

This chapter of this thesis is devoted to summarizing the previous chapters and presenting my ideas for future work to be done on this research and where I can see improvements that I could make to my design going forward. The group messaging protocol has been added to the Lively Web [5] code and is now available for use by any user when they create a world that requires collaborative behaviors.

6.1 Summary

This research focused on the idea that, while there are many tools exist to present to multiple people in disparate locations, there are few that allow you to work on a dataset together in a truly collaborative fashion. Further, there are even fewer tools available that allow this to be done on any browser enabled device. In Chapter 2 we discussed the requirements for the application to feel interactive and allow the users to effectively interact, including the requirements of the deployment system.

In Chapter 3 the discussion centered around the proposed design of the solution. We discussed how cost models had changed, making it cheaper for larger volumes of data to have programs shipped to them instead of the traditional, and opposite, method. The paper then went on to describe the proposed communication protocol and how it would operate under changes in scale, including limiting the searching required to send messages to the appropriate parties and how the groups described are separated in namespace to prevent accidental collision. Finally, we discussed how group messaging would operate. For examples of applications and experiments done
to determine timing and effectiveness, Chapter 4 described two applications designed to allow users to collaboratively manage datasets, one with a distributed data model and another without.

The messaging protocol itself was compared to other, similar, tools in Chapter 5. We discussed that, while other tools had some features needed, none had them all, so either additional book keeping or finessing would need to be done to get them to do the required job, or another protocol could be written entirely. The latter was chosen and then described in detail, including showing examples of how messages were passed and how security was maintained.

The inclusion of the Lively Group Messaging Protocol allows a user to quickly and easily create worlds that communicate with each other on a subscription-based service. The low-overhead tracker service maintains the connection list to send messages to and requires no direct filtering to prevent messages going to unintended participants in a session. The examples shown were not exhaustive examples of the work possible in this space, but offered a sample of what is possible.

6.2 Future Work

In the future, a few things need to be taken on. Primarily, in the space of the existing Pollution Visualizer mentioned in Section 4.1, the number of files on disk need to be addressed. This could be done by finding a suitable database structure that would provide the response times required or reducing the size of the data to allow some or all of it to fit in memory. In addition, while the mapping software used was able to render the number of points in 100 ms, it is possible we would be able to find a faster drawing map software suite and shift additional time from drawing to fetching data and allow a more standard database solution to be used.

The intent with the group-oriented messaging service is to provide developers a quick and easy way of developing interactive, collaborative applications. To do this, further projects should be created and, at the end of each project, determine what additional tools needed to be added to the group messaging protocol for that application in order to make it feasible and, if they look like something that will be needed in a general fashion going forward, that they then be incorporated into the protocol at large. Repeating this method will result in a smoother experience for other developers not as familiar with the inner workings of the system.
For the sports visualization focused on the shot charts for NBA players, we discussed in Section 4.2 that to reduce the load on the NBA servers, we would duplicate the statistics with permission to our own database or data structure and serve them to our own application. This would also allow a more consistent experience for the user, in that a change in the API or structure of the data served by the NBA or the removal of the API entirely would not cause a change in the functionality of the application. In particular, I believe that an interactive map showing either current disease and outbreak reports or a tool allowing the model of the spread of a disease could be modeled, given a suitable data source, and that this would allow experts around the world to experiment with different scenarios to prevent or limit outbreaks.
Bibliography


