LivelyViz: an approach to develop interactive collaborative web visualizations

by

Voltaire Bazurto Blacio
B.Sc., Universidad Politécnica Salesiana, Ecuador, 2009

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

© Voltaire Bazurto Blacio, 2016
University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part, by photocopying or other means, without the permission of the author.
LivelyViz: an approach to develop interactive collaborative web visualizations

by

Voltaire Bazurto Blacio
B.Sc., Universidad Politécnica Salesiana, Ecuador, 2009

Supervisory Committee

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

Dr. Chris Upton, Co-Supervisor
(Department of Biochemistry & Microbiology)

Dr. Rick McGeer, Departmental member
(Department of Computer Science)
Supervisory Committee

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

Dr. Chris Upton, Co-Supervisor
(Department of Biochemistry & Microbiology)

Dr. Rick McGeer, Departmental member
(Department of Computer Science)

ABSTRACT

We investigate the development of collaborative data dashboards, comprised of web visualization components. For this, we explore the use of Lively Web as a development platform and provide a framework for developing web collaborative scientific visualizations.

We use a modern thin-client approach that moves most of the specific application processing logic from the client side to the server side, leveraging the implementation of reusable web services. As a web application, it provides users with multi-platform and multi-device compatibility along with enhanced concurrent access from remote locations.

Our platform focuses on providing reusable, interactive, extensible and tightly-integrated web visualization components. Such visualization components are designed to be readily usable in distributed-synchronous collaborative environments. As use case we consider the development of a dashboard for researchers working with bioinformatics datasets, in particular Poxviruses data.

We argue that our thin-client approach for developing web collaborative visualizations can greatly benefit researchers in different geographic locations in their mission of analyzing datasets as a team.
Contents

Supervisory Committee ........................................... ii
Abstract .................................................................. iii
Table of Contents .................................................... iv
List of Tables .......................................................... vii
List of Figures .......................................................... viii
List of Listings .......................................................... x
Acknowledgements .................................................... xi
Dedication ................................................................ xii

1 Introduction .......................................................... 1
   1.1 Motivation ......................................................... 2
   1.2 Problem statement ............................................ 4
   1.3 Approach ........................................................ 4
   1.4 Thesis organization .......................................... 5

2 Background .......................................................... 6
   2.1 The Viral Bioinformatics Resource Center (VBRC) ....... 6
   2.2 Viral Orthologous Clusters (VOCs) .......................... 7
   2.3 Genome Map ................................................... 7
   2.4 Genome map plotting in VOCs .............................. 9
   2.5 Further visualization with external tools .................. 11
   2.6 Limitations ...................................................... 11
   2.7 Proposed architecture using Lively ....................... 13
6.2 Future Work ................................. 73

Bibliography .................................. 76
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Desired features in the LivelyViz visualization components</td>
<td>32</td>
</tr>
<tr>
<td>5.1</td>
<td>Number of LOC used in the implementation of common features in VOCs and LivelyViz</td>
<td>67</td>
</tr>
</tbody>
</table>
List of Figures

Figure 2.1  The main window of VOCs client ................................. 8
Figure 2.2  VOCs Genome Map showing genes in a selected genome and highlighting conserved genes by using a color scale .... 9
Figure 2.3  Current architecture of VOCs client communication with the VOCs MySQL database ................................. 10
Figure 2.4  The new architecture using Lively Web ......................... 13
Figure 3.1  The evolution of the web platform over the years as shown in [38] 17
Figure 3.2  Examples of network biology visualizations .................... 19
Figure 3.3  The JBrowse interface ............................................... 20
Figure 3.4  GenomeD3Plot example ............................................. 20
Figure 3.5  Pileup.js example .................................................. 21
Figure 3.6  The MSAViewer component ....................................... 21
Figure 3.7  A jHeatmap example ................................................ 22
Figure 3.8  A scatterplot visualization built using d3.js ................... 22
Figure 3.9  The Epiviz application .............................................. 24
Figure 3.10 The CompPhy web interface ..................................... 25
Figure 3.11 PBrowse genome browser ....................................... 27
Figure 4.1  Time-Space matrix showing the possible scenarios of collaborative visualization ........................................ 34
Figure 4.2  A sequence depicting how actions can be executed remotely by demand .................................................. 36
Figure 4.3  An example showing the default behaviour when two different morphs register actions using the same name ......... 37
Figure 4.4  The session manager component .................................. 42
Figure 4.5  Annotations in the genome map component ....................... 44
Figure 4.6  Generic Awareness Architecture .................................. 45
Figure 4.7  User awareness morph example .................................. 46
Figure 4.8  Morph showing the list of participants involved in a collaborative session ................................................. 47

Figure 4.9  Two morphs connected through the Lively Morphic Connection API ............................................................ 47

Figure 4.10  Linked interactions between LivelyViz components .......................................................... 50

Figure 5.1  The DB Chooser component ........................................................................................................ 54

Figure 5.2  The Genome Map component ......................................................................................................... 55

Figure 5.3  Changing the color of a gene ............................................................................................................. 55

Figure 5.4  The Scatterplot component ............................................................................................................. 56

Figure 5.5  Previewing a genomic dataset ........................................................................................................ 56

Figure 5.6  Selecting numerical columns in the dataset as axes .............................................................................. 57

Figure 5.7  The formula builder dialog ............................................................................................................. 57

Figure 5.8  Plotting a scatterplot using a new column created with the formula editor ........................................... 58

Figure 5.9  The Protael visualizer incorporated in LivelyViz .............................................................................. 59

Figure 5.10  Bar chart showing total LOCS of VOCs project ............................................................................ 60

Figure 5.11  VOCs top ten files with the major number of LOC ........................................................................ 61

Figure 5.12  Bar chart showing total LOCS of VOCs project ............................................................................ 61

Figure 5.13  VOCs’ top ten files number of LOC ............................................................................................ 62
List of Listings

Listing 4.1 Defining a L2L service in a world ........................................... 35
Listing 4.2 Sending a message to request a remote action execution .... 35
Listing 4.3 Proposed approach to define collaborative isolated actions per morph of the same type .......................................................... 39
Listing 4.4 Rename is also necessary when sending a message to a remote peer 39
Listing 4.5 Function for associating a morph name with an action name . 40
Listing 4.6 The registerGeneralEvents() method ........................................ 40
Listing 4.7 The executeAGeneralFunction() method ................................. 41
Listing 4.8 Example of connecting two morphs with the Lively bindings API 46
Listing 4.9 Proposed approach to interconnect LivelyViz morphs ............ 47
Listing 4.10 The notifyLocalConnectedWidgets() method ...................... 49
Listing 4.11 The processCommandFromConnectedWidget() method ...... 49
Listing 5.1 Minimal required code for implementing mouse events in a canvas in Java ................................................................. 62
Listing 5.2 Minimal code required to handle the click event in javascript . 63
Listing 5.3 Fluent interface in javascript ......................................................... 64
Listing 5.4 Font formatting related code in VOCs ..................................... 64
Listing 5.5 Font styling in Lively using CSS ................................................. 65
Acknowledgements

I would like to thank:

**Dr. Ulrike Stege**, for her mentoring, encouragement, optimism and unconditional support. Without her collaboration I could not have completed this work.

**Dr. Chris Upton**, for his support, patience and guidance on the bioinformatics part of this thesis. He and the members of his lab always provided me with valuable support.

**Dr. Rick McGeer**, for his time, comments and suggestions to improve my work.

**Matt Hemmings, Marko Röder and Robert Krahn**, for supporting me every time I had a question about Lively.

**PITA and Rigi groups**, for their useful feedback.

**My family**, for always encouraging me and supporting me to pursue my goals.

**The Government of Ecuador and the SENESCYT**, for funding me with a Scholarship.
Dedication

To my mother for her support, sacrifice and dedication on providing me always with the best possible education. To my father for stimulating my curiosity of learning more about science and technology since I was a child. To my brother Fabio and my sister Zulemma for always encouraging me to pursue my goals.
Chapter 1

Introduction

We live in an era where technology is present in almost every aspect of our daily lives. The ubiquitous presence of electronic devices in our society contributes to the capturing, generation, storing, processing and exchange of data. As a consequence, we could come to the conclusion that data is present everywhere.

In a time where we have massive amounts of data at our fingertips, humans need an effective way to view and understand data easily. For human beings, the visual channel is known to be very effective for acquiring large amounts of information from the environment. All perceived information is then processed by the brain \[30\] [15] [90]. By making use of effective visualizations \[35\] derived from large amount of data, information can be delivered to humans using representations simpler to understand rather than looking at the raw data directly.

Nowadays, the use of effective visualizations benefit a variety of fields \[32\] [4] (e.g., entertainment, health sciences, news, education, transportation), aiding people who wish to gain insights about a specific subject through the examination of data [92]. Visualizations can be powerful allies in tasks such as summarizing, observing trends, discovering new patterns, planning and decision making.

Scientific visualizations \[52\] come to the aid of researchers who require to understand, modify, compare, derive and plot data. Moreover, interactive visualizations are helpful to experts in a specific field who are trying to find hidden traces or patterns in their data. Such visualizations are intended to aid in an exploratory analysis of a dataset \[105\] obtained from an experiment.

With the advent of services in the cloud, especially the so called Software as a Service (SaaS) \[60\], users can upload and store their documents or save their work on remote servers and share that information with other users, encouraging them to
collaborate with each other. Adding collaborative features to interactive visualizations provides a faster way to perform analysis and discuss their results among a group of experts.

The constant evolution of web protocols, web browsers and server side technologies allow developers to create complex applications and make them available around the world when shared via the internet. However, because of the plethora of available technologies, programming languages, web browsers and the required amount of expertise to develop a web application from end-to-end (client side and server side), developers have a steep learning curve to deal with when they start learning how to develop such kind of applications. In order to help harness such challenges, the Lively Web project [13] (in this thesis, short: Lively) was created. Formerly known as the Lively Kernel [99], this project attempts to provide a full-featured web development platform that hides the complexities that arise when programming in the various layers of web applications, and when encapsulating as many lines of code as possible into web objects that can be programmed in an intuitive visual way. The purpose of this thesis is to explore and demonstrate the uses of Lively in providing a framework for developing web collaborative scientific visualizations. Specifically, we investigated the development of a collaborative dashboard tool for the visualization and understanding of genomic data.

1.1 Motivation

Exploratory analysis of data (EDA) [43]—where the source of the data could be either from the results of an experiment or from a simulation—is a powerful tool to gain insights into a particular phenomena. In 1977, Tukey [104] proposed to analyze the data with the aid of visual representations in order to discover patterns that could lead to formulate some hypotheses. His approach is very useful when we do not know exactly what to look for in the data. In such scenarios, we would rather carefully inspect the data and try to figure out what it is trying to tell us [105].

Usually, experts need to look carefully through a dataset when searching for patterns, correlations or anomalous behaviours among elements, in order to come up with a hypothesis. In many disciplines this is achieved by the use of visual data representations.

For example: when we know exactly what are we looking for in a dataset and we have a defined methodology, procedure or a formal model to test a hypothesis,
an algorithm can be designed to compute the results. But if we do not know what
to expect in advance, exploring the dataset through visual representations using the
human eye can be really powerful to discover traces and patterns that can help us to
gain a better understanding of the data and come up with new research questions and
ideas.

In the area of bioinformatics, EDA is a powerful tool to help to understand genomes
and their structure, as well as their genetic properties [85][102]. Depending on the
nature of the research and the dataset analyzed, data can be represented in different
ways [100] by using abstract visualization objects (AVO) sometimes also referred as
idioms [72].

Collaboration plays an important role in research. Sharing research results and
analysis findings with other experts is a common practice among bioinformatics
researchers. Very often research collaborators are located in different geographic areas;
therefore, it is a common practice to rely on the use of external tools for achieving
some degree of collaboration. Such kind of tools include: chat rooms, forums [93][83],
social networking [71], webpages [47], wikis [3] and databases.

However, these external tools are usually general-purpose applications designed
for sharing content or to achieve some level of communication among users. They
were not designed to integrate closely with the tools used during an exploratory data
analysis session. Moreover, integrating such external collaborative tools with existing
applications or platforms used for bioinformatics analysis in a lab might require some
significant additional work and technical knowledge, specially if such existing running
applications are outdated (legacy systems). Examples of such additional efforts could
be: setting up a parallel IT infrastructure for communications; learning how to use
a new tool to integrate them into a research pipeline; programming in a specific
non-standard application program interface (API) for achieving a very limited level of
integration with existing applications.

Another approach to improve communication and collaboration among researchers
in a team is adding built-in collaborative capabilities, e.g., when developing specific
bioinformatics applications. In this way, the entire team and research workflow
get an integrated mechanism to communicate results to other users of the same
platform without establishing additional barriers. Unfortunately, developing this
new communication layer from scratch often leads to more time and effort for the
application development team, while instead one wishes to put the focus on the
features that are more inherent to the particular bioinformatics research problem.
1.2 Problem statement

The aim of this thesis is to investigate and demonstrate the use of Lively Web as an effective development platform for developing web collaborative visualization tools. As a use case, we developed a web dashboard application to showcase a bioinformatics dataset, in particular poxvirus genomic data. We propose a set of reusable visualization components readily to use for real-time collaboration scenarios that we call LivelyViz.

This leads us to the following research questions:

- RQ1: Can Lively provide a platform to develop effective interactive web visualizations?
- RQ2: How can Lively contribute to the developing a collaborative oriented bioinformatics dashboard application? What makes it stand out from traditional development platform or methodologies?
- RQ3: Can Lively integrate pre-existing datasets and third-party visualization libraries into its workflow to extend the dashboard with additional visualizations?

1.3 Approach

In this thesis we describe the implementation of a dashboard application for visualizing viral data using the Lively Web platform. This dashboard connects to the Viral Orthologous Clusters (VOCs) database project developed by the Viral Bioinformatics Resource Center (VBRC), to obtain the data that feed the visualizations.

In order to address the previously formulated research questions, the following approaches are taken:

- Develop a web dashboard of visualizations using Lively Web platform: The dashboard should contain at least two different types of reusable visualization components, especially designed to plot poxvirus genomic data. Such components should be linked to each other when dealing with the same genome reference.

- Define an essential layer of communication: this is done using Lively network capabilities based on HTML5 WebSockets, to allow connected users to the same webpage to interact simultaneously to perform EDA using reusable visualization components.
- Provide a real example showing how Lively can connect to external data sources such as databases and serve this data to feed the visualization components. In our example we are using the public VOCs MySQL database.

1.4 Thesis organization

The thesis is organized in the following manner:

Chapter 2 provides background information related to the VBRC lab and the VOCs tool. It also describes the current features, architecture and limitations of the VOCs application; along with the proposed architecture to design the dashboard application using Lively.

Chapter 3 describes notable literature related to previous work on web collaborative visualizations.

Chapter 4 explains the reasons for choosing Lively as a development platform and the challenges found during the development. The proposed framework/methodology to use Lively to develop the application and the visualization design concepts are also discussed here.

Chapter 5 discusses the developed visualization components and their features. In addition, a code metrics evaluation is presented to compare LivelyViz and VOCs.

Chapter 6 describes the conclusions obtained from our research along with ideas and suggestions for future work.
Chapter 2

Background

Nowadays, it is very common that researchers based in different geographic locations are interested in analyzing and visualizing the same dataset simultaneously. However, in most cases such analysis tools lack collaborative features. This poses a big challenge and limitation for researchers who wish to share their analysis results, especially when the size of their datasets makes it unfeasible to transmit the whole data over the network. This usually results in the usage of external communication applications to achieve some degree of collaboration. Given these external communication tools are not directly integrated with the research and analysis pipeline, this may results in inefficient, repetitive and time consuming tasks, such as the reformatting of data to be transmitted using external tools, every time new results are available.

2.1 The Viral Bioinformatics Resource Center (VBRC)

The VBRC focuses on large DNA viruses, with a prominent interest in the poxvirus family. This research group has developed several tools [65] over the past twenty years that contribute to the study of such viruses [40]. On their website they provide access to public databases and tools, mainly for comparative analysis of virus genomes [96][111]. Among those tools are: VOCs [64], VGO [107], Base-by-Base [51], JDotter [50], GATU [101] and other additional applications that can be found on the virology.uvic.ca website under the menu option “Tools”. All these tools were specially designed having in mind, as primary users, virologists rather than computer scientists. Thus their graphical interfaces are rather easy to use to perform analysis tasks on the genomic data, in comparison with command line tools.
2.2 Viral Orthologous Clusters (VOCs)

The Viral Orthologous Clusters (VOCs) [64] is a Java GUI client that can connect to and access the information stored in the VBRC databases. This software operates on a client-server architecture. The Java application can be downloaded from the virology.uvic.ca website and is installed locally on a PC through the web browser by using the Java-Web Start service [10]. This means that in order to be able to run the client software, the user’s computer will require to have installed the Java Platform Standard Edition 6 [8] (or a superior version).

The application can be accessed and launched from the virology.uvic.ca website by selecting the VOCs option under the “tools” menu. After clicking the button for launching the application from the website, the Java-Web Start service will prompt the user to accept the download of the application. If there is any newer version of the software, the Web Start service will take care of everything for the user and it will download all the required files onto her computer.

The server side component of VOCs consists of a MySQL [22] database that stores fully annotated poxvirus genomes, proteins and genes. In addition, the database stores information related to: open reading frames (ORFs), predicted isoelectric points (PI), molecular weights (MW), nucleotide and amino acid frequencies, and codon usage.

From the main window of VOCs (2.1), queries can be run against the database by specifying certain search parameters and filters. The tool also integrates some other analysis tools [61] that can be executed as pipelines with the input data obtained from the VOCs database queries. Among these tools are: TBLASTN, BLASTP, BLASTX, BLASTN [45, 44], JDotter, Base-By-Base, VGO, Genome Map. One example of a pipeline might involve these steps:

1. Select two or more sequences from VOCs and use them as inputs
2. Align the selected sequences using ClustalO or any other similar software to produce results
3. View the produced results in Base-By-Base

2.3 Genome Map

One of the tools that can be launched from the VOCs interface is Genome Map (2.2). Genome Map allows users to visualize the location of every gene in a selected genome,
Figure 2.1: The main window of VOCs client

showing also information such as the numbering of each gene, size of the gene, distance among genes, strand sense and a color. The color in every gene helps to provide additional information about it. The tool provides a few default coloring schemes for the genes. The default coloring scale represents how conserved is a gene in other viruses of the same family.

The entire genome is shown in a linear way that is divided in several tracks of fixed length of base pairs (bp). These tracks are stacked one over another to achieve the presentation goal of fitting in one page. Every gene is visually represented with a geometric shape along a track, the length of the shape represents the length of the gene from a specific start to an end position. These shapes are usually arrows representing the orientation of the gene. Genes that are considered to be in the positive strand are placed at the upper part of every track line and the negatives ones below it. In addition, every gene arrow is clickable to show detailed information recovered from the database about it and allowing the user to edit the color of the selected gene.

Additional features include: being able to save the entire genome map as a png image, launching additional windows with detailed information about the genome,
importing text files to change the color of multiple genes at the same time.

2.4 Genome map plotting in VOCs

VOCs provides the option to draw the entire map of a selected viral genome for the purposes of displaying that map in one page.

In the current architecture (Figure 2.3), the VOCs application is delivered to the user device by serving a JNLP file. After the JNLP file has been downloaded and opened, the Java Web Start application will start downloading the VOCs application onto the user PC. Once the VOCs application is downloaded and loaded, the user must select a family virus database to begin to work.

When the user has selected the database, the application will download a public xml file (DBPref.xml) from the server, containing the connection information of every available viral database. The DBPref.xml-file will then be parsed and the application will retrieve the connection parameters for the selected virus family.

After the xml code is parsed, a connection is created with the database server.
Then the application will be ready to send SQL queries to the database in order to retrieve information.

It is worth to noting that the current architecture has some disadvantages. From the perspective of the user, in order to start using using the application, the PC client must have a Java virtual machine in order to read JNLP files. The time required for loading the entire application at once can be very long.\(^1\)

In addition, the VOCs client attempts to connect directly from the users PC with the database server, using non-standard port numbers. There is a possibility that this will lead to accessibility problems when using the application in a network that is behind a firewall.

All queries and SQL logic necessary to retrieve the information from the database

---

\(^1\)We measured the time required by the VOCs application to load and run after being executed for first time on a laptop. In order to run VOCs for the first time on a computer, around two minutes were required only to download the required packages, then a dialog prompting the user to choose a viral database to start working was shown. On the subsequent tries, around 14 seconds were required to show such viral selection dialog. After selecting a database to start working, a splash screen is shown while VOCs is getting ready to display its main window. This loading process took around 48 seconds.

The reported times required for loading the entire application are understandable given that all modules included in VOCs need to be loaded. However, these loading times can be excessive if the user only wishes to perform simple visualization tasks, such as drawing a genome map.
are stored in the VOCs client Java application, allowing to send queries to MySQL or Microsoft SQL Server databases. In the original design of VOCs, there was a consideration for supporting several Relational Database Management Systems (RDMS). Therefore, custom libraries were developed to work as an abstraction layer to support only the two databases previously mentioned. Currently, on the server side a MySQL database server is used.

Having all the logic for the SQL queries configured as part of the client application means that, for any single change in the database queries logic, the whole Java application must be recompiled and uploaded to the server. This in turn means that users must download a new version of the application every time the JNLP file is opened. Although thanks to the Java Web Start system the deploy and versioning system of the client application is managed transparently (making things easier for users in this aspect), every time when there is a new version of the application, the whole client must be downloaded again by the user.

2.5 Further visualization with external tools

In some cases, the VBRC researchers are interested in investigating a particular feature shared among several viruses, so they use numerical and statistical analysis of the data stored in VOCs database. To get a better understanding of the results of their analysis, they usually need to plot the data in a different way than the ones provided by VOCs. In this scenario they might rely on external applications to generate bar charts, scatterplots or other kind of visual representation of measurable data. They typically export their data from VOCs into a Microsoft Excel spreadsheet and take advantage of the prebuilt graph generator features. However, every time the data is updated in the VOCs database, a new dataset has to be exported and then loaded by the external plotting tool.

It is a natural goal that all these analysis results and plotted graphics are shared among colleagues in order to discuss them and in order to pursue research.

2.6 Limitations

The genome map tool has limitations that could be overcome to provide a better user experience:
• The initial load time of the data from database before plotting was around 11-14 seconds, even if this time is not significant, it is clearly a point that can be optimized.

• Although additional windows can be launched from the genome map showing detailed information from the database, or allowing to pipeline certain data from the genes (such as the nucleotide sequence) to other VOCs tools such as JDotter, there is not a direct integration between those tools and the genome map. There is no integrated actions and visual feedback such as linking and brushing highlighting or dynamic loading content under demand.

• There is no ability to collaborate with other users using the same tool at the same time. Every user has their own copy of data and they are not allowed to communicate or send feedback from inside the application, therefore they need to rely on external applications such as email, chat or Skype.

• Although more than one window of the genome map can be launched, it is uncomfortable to arrange them to compare them visually simultaneously. This is mainly because while interacting with one window, the other windows lose the focus and can get hidden from the user’s sight. In addition of that, there is no way for those windows to interact, such as coloring all the common genes in every genome map when clicking on a specific gene in one window.

• The current state of the tool does not allow to integrate into other external visualizations or tools made by third parties.

• For every new change implemented in the tool, the whole Java VOCs project needs to be rebuilt from source code and uploaded to the web server so it can be redistributed to the users.

• The Java Runtime Environment (JRE) must have been installed previously in the computer where VOCs is going to be executed. A user might need run VOCs in a computer or environment where he/she does not have administrative privileges to install new software.

• In some networks behind a firewall, the connection to the VOCs database might fail because the MySQL communication port might be blocked by the firewall. This happens because the database connection is initiated by the VOCs client.
requesting a connection to a remote MySQL server port. It is common in various scenarios such as corporate intranets this kind of ports are blocked by default. The VOCs user may need to contact the network administrator to open those ports.

2.7 Proposed architecture using Lively

As an alternative of using the VOCs client to launch visualization tools in a pipeline style, we propose in a new architecture (Figure 2.4) the use of a web application client. This web application will be served to the user as a web page, stored on a Lively Web server. In this way, the user will only require a web browser to access the new web application.

In order to maintain the current scheme for connecting to the database, the proposed architecture maintains the use of the DBPref.xml. This file can safely be moved onto the Lively Web server, independent of whether the intention is to maintain it as public or to make it private. Like this, the new system can work in parallel with the legacy system without affecting the current production set-up.

One big difference of our new architecture when compared to the old architecture is that connections to the database are not made directly from the client application.
side. Here, the Lively Web server will be responsible to connect to the database and to send the SQL queries to the database server. This has several advantages: if any change is made to the logic that retrieves information from the database, there is no need for recompilation of the code, because the server side programming consists of javascript language, which is interpreted by a Nodejs server [24]. Any change that will be made to the logic on the server side will immediately be reflected when the code is executed again. Another advantage is the ease of maintenance of the source code: all this logic belongs to the backend operations, and it is best not to mix with the frontend application code. This separation of the code benefits the maintainability and scalability of the application.

The maintainability is improved because it is easier to find the root cause of a problem in the whole application when frontend and backend are separated. Both frontend and backend layers provides error consoles to review the error messages produced on each part of the application. Also, when introducing new changes or modifying one specific layer, the other layer is not affected and it can continue operating. Functional scalability is achieved because the modular nature of server side code in Lively. New server code can be created as an individual endpoint that can be consumed by applications. The creation or modification of a single endpoint will not affect the services yielded by other endpoints. This eases the way in which new code can be added to the server to meet with requirements of implementing new features in the application.

It is also worth mentioning that additional storage services can be added to the Lively Web server such as incorporating SQLite database files, installing a relational database management system or using plain files. This can benefit the entire VOCs project by having isolation of application related data, test result data or just totally new datasets from the main MySQL database containing the curated poxvirus information.

Another benefit that our approach achieves is reusability. Having the queries in a single accessible point such as the Lively Web server, any other application that can make an HTTP request can use the previously developed methods for retrieving information from the database.

Thanks to the capability of Nodejs to add additional functionality through the use of modules, a wide range of libraries is available to the developer to perform different kinds of tasks on the server side. For achieving the goal of supporting different databases, the Sequelize [33] module is used. Sequelize acts as an Object Relationship
Mapper that maps every table on the viral database to a javascript Object class. By doing this, Sequelize allows to abstract the structure and the language details of the database making easy to switch from one database to another. To date, Sequelize supports MySQL [22], PostgreSQL [28], MariaDB [14], SQLite [34] and Microsoft SQL Server [16].

By serving the frontend as a web page, the user can easily access it by using a supported web browser. One big advantage of this is that the application can be executed in multiple operating systems and platforms as long as a capable web browser is available. For example, the application could be executed on a tablet, laptop or desktop PC. This is a new important feature because VOCs cannot be executed in tablets.

The frontend of the application is organized using graphical components called morphs that perform a specific task such as selecting the database related to a virus family, showing the list of genomes from a selected family or plotting a complete genome.

2.7.1 Summary

This chapter covered general background information related to the VBRC lab, their current analysis tools and their database VOCs. By reviewing the current architecture of VOCs, we identified limitations and shortcomings that researchers face when trying to use their analysis tools. We presented an alternative architecture using Lively to overcome the difficulties found in the original one.
Chapter 3

Related work

The purpose of this chapter is to provide a brief introduction about interactive visualizations on the web and previous work related to our thesis. Visualization applications are used almost on any kind of field or discipline that can be imagined, but since our research use case is focused on the bioinformatics field, we concentrate on describing related work on visualizations and collaboration in this discipline.

In this chapter we cover topics such as: how the web has been used for delivering visualizations to users, the role of technological improvements in HTML and javascript to benefit the way that visualizations can be developed nowadays, the use of web visualizations applied to the bioinformatics field and a few examples about web collaborative visualization applications for bioinformatics.

3.1 The web as a visualization platform

Nowadays, the web has become a very popular alternative to deliver interactive remote visualizations in contrast to the traditional approach of using desktop applications. Mwalongo et al. [91] presented a survey in 2016 with the current trends about the development of web-based visualization applications.

Although desktop applications are still being used for heavy intensive computation visualization duties on the client side (such as in simulations or complex 3D modelling), this has started to change slowly with the advent of technologies (Figure 3.1) such as WebGL [89] and Graphics Processing Unit (GPU) programming capabilities that have been incorporated into modern web browsers.

Web applications provide to users some practical benefits such as the simplicity
of only relying on using a modern browser to consume such applications from any operating system, location or compatible devices (e.g. desktop, laptops, tablets and cellphones). In addition, users will always get the last version of the web application just by visiting the corresponding link of that application, no additional installs or downloads are necessary. From the developer perspective, it is easier to maintain the source code of the application because the same code will run on different platforms and devices as long there is a capable web browser installed. Moreover, the process of deploying the application to final users is as easy as sharing a link with them.

HTML5 also provides a convenient way to create 2D visual representations through the Scalable Vector Graphics (SVG) html element and the Canvas html element [95]. SVG is based on XML and allows to create graphical objects that can be resized without losing quality on the image; on the other hand, the images in Canvas are made of pixels(raster based) and are created via programming.

Apart from the 3D and 2D rendering capabilities provided by WebGL, the addition of other features to the javascript web API have taken the traditional web applications to the next level that provides users with a improved experience in terms of interactivity, usability and remote collaboration. Among these features we can highlight: the ability to store binary information with the data structure ArrayBuffer [20] [19], the execution of background code through Web Workers, bi-directional network communication with server using websockets [84] and peer-to-peer communication without relying on a central server with WebRTC [48].

But not only developments in the client-side have contributed to richer and more sophisticated web applications but also advancements on the server-side technologies.
Grid computing [68] and cloud computing [86] allow to move intensive and complex computational processing to the server side fostering in this way a thin client architecture. This heavy and distributed computing power helped to optimized the amount of data transmitted to the client side, either by passing already generated output visualizations as images to display after demanding scientific data processing; or by sending compact and reduced data structures to be used for reconstructing visualizations on the client side. Either way, the aim is to transmit only the data necessary to show visualizations on the client side. This approach encourages minimizing the amount of data transmitted, especially when dataset sizes make the transmission of them over the network impossible. However, the main challenges that remain on web-based visualizations applications are related to network latency and bandwidth [91].

### 3.2 Javascript visualization libraries

On top of the basic features that HTML5 provides for creating 2D and 3D graphics, developers have built specialized javascript libraries to deal with common and repetitive tasks to initialize, produce, modify and animate different types of visualizations on the web. Libraries such as D3.js [49] to create 2D visualizations or Three.js [55] to produce 3D renderings in the browsers are just two examples to mention.

D3.js is a library for creating 2D interactive visualizations based on data, it leverages SVG, CSS and HTML to build visualizations on a webpage. Almost any type of visual encodings used on information visualizations can be constructed with this library such as pie charts, bar charts, line charts, scatterplots, parallel coordinates, etc. Even custom designed visual representations can be developed with d3.js, anything that can be represented with a geometric or polygonal shape.

However, sometimes in certain areas it is required to visualize data in a particular way depending on the domain of the data, for example: bioinformatics, cartography and exploratory data analysis. In the field of bioinformatics, there are several initiatives to create a wide range of visualization applications, components and libraries that help to create visual representations of systems biology and genomes, Pavlopoulos et al. covered this in their survey in 2015 [94]. Visualizations in bioinformatics could be categorized according to the type of data that is intended to be represented, such as network biology visualizations, genomic visualizations or visualization for analyzing expression data [94].

In network biology visualizations, it is important to show the relationship among
biological entities or bio-entities (e.g., proteins, genes, pathways, molecules). Examples of libraries (Figure 3.2) for representing such kind of data are Cytoscape.js [70] for representing network relationships using networks or KEGGViewer [109] for signaling pathways.

![a) Cytoscape example](image1)

![b) KEGGViewer example](image2)

**Figure 3.2:** Examples of network biology visualizations. 1) Cytoscape example as depicted in Franz et al. paper in 2015 [70]. 2) The KEGGViewer example. Image adapted [109].

Tools related with genomic visualizations can be divided in four categories: genome browsers, genome assembly visualization tools, genome alignment visualization tools and tools intended for comparative genomics [94]. There are several genome-related visualization libraries for the web nowadays, such as Pileup.js [108], GenomeD3Plot [82], JBrowse [53] or MSAViewer [112].

JBrowse (Figure 3.3) is a highly customizable genome browser that supports visualize several type of genomic tracks at the same time, load several genome sequence file formats, consume external datasources via REST, add functionality through a plug-in system and redefining UI events. It can be considered a full web genome browser that can be used not only for visualizing but also for analyzing genomic data.

GenomeD3Plot (Figure 3.4) is a library for plotting genomic data written on top of D3.js. It supports four type of tracks (standard, stranded, plot and glyph) and allows to plot not only linear genomes but also circular ones. It was designed to be used a visualization component to be integrated in other web applications.

Pileup.js (Figure 3.5) is a recent genome viewer library designed to study and analyze genomic variants. This viewer leverages the latest techniques and technologies available in the javascript world in order to provide, not only up-to-date compatibility
with current technical specifications of web applications, but also to yield a robust library for visualization and analysis of genomic data. The library can be embedded in larger web applications and it can be customized via javascript programming and CSS styling.

The MSAViewer visualization component (Figure 3.6) allows to load and display multiple sequence alignments on the web. It can read FASTA [79] or CLUSTAL [58] data either from the local disk or from a remote computer. In addition, visual interactions to help inspect the data such as zooming and panning are possible. In the visualization area, not only the entire sequence alignment is shown. Consensus information can be displayed either in the form of big letters or as bar charts in a different track. Alignments can be exported to a text file.
Popular visual encodings to present gene expression data are scatterplots and heatmaps [80]. Figure 3.7 shows an example using jHeatmap, a web-based interactive visualization component to visualize heatmaps [63]. Scatterplots can be constructed relatively easy using the d3.js library as shown in Figure 3.8.

The growing interest of migrating visualization applications to the web has led to the creation of projects such as BioJS [59], which is a javascript framework containing a set of reusable visualization components to represent biological data. This project features also a centralized online repository where new visualization components can be added. MSAViewer and the KEGGViewer, described previously in this section, have been already included in the BioJS framework.

3.3 Collaborative visualizations in bioinformatics

In this section we describe briefly three different web-based bioinformatics visualization tools used for data analysis and exploration.
3.3.1 Epiviz

Epiviz [56] was planned as a tool that could bring together popular computing environments where bioinformatics data is pre-processed, and interactive visualizations and collaboration of the data are derived from the interactive analysis.

In an early version of Epiviz [57], a web-based genome browser component and a package called Epivizr were provided. The Epivizr package was developed to serve as a bridge between the Epiviz visualization web application and the Bioconductor [2] tool for the R-language computing environment [29]. Bioconductor is a popular set of tools for statistical analysis of high-throughput biological data using the R programming language.
The original idea was to tightly couple both tools providing an environment for analysis of data and reproducible visualizations. The Epivizr package allows to take all the advantages of the Bioconductor analysis environment and make all the results available using WebSockets to communicate and integrate with the Epiviz genome browser. The package is also responsible for mapping all the data types existing in R to Epiviz data types, in order to be used for the visualizations. It is worth to mention that the communication protocol using WebSockets allows to incorporate other data sources different from the Bioconductor platform. In such way, other platforms that could act as a data provider can be added as long they implement the same communication protocol. Other possible environments that could act as data providers could be for example Python and PHP serving data from MySQL.

Collaboration is supported by generating persistent URLs that can be shared with other users. These URLs reproduce data related to previous performed analysis, customization in visualizations components and computed measurements.

Epiviz (Figure 3.9) provides the ability to extend the available visual components with D3.js user defined visualizations. Additionally, there is a Data Provider component that allows to integrate further data sources. Moreover, external scripts that can extend the functionality of the different visualizations can be incorporated. Such scripts are stored and retrieved from GitHub gist.

The concept of workspaces introduces the ability of storing UI/code customizations and results from analysis operations in order to share them with other users. Thanks to this feature, other users can replicate the steps of the analyses that have been shared with them in their own workspace.

Other notable features in Epiviz are:

- The ability to customize the behaviour of a visualization component through a javascript code editor.
- The data can be transformed and derived into new datasets directly from the UI with javascript code. This increases the flexibility for users who want to modify the data displayed in a visualization. Such kind of transformations include: filtering by an object property, changing colors based on coordinates or measurements, group by measurements and order by measurements.
- The javascript code used for customizing both data and visualizations can be stored per user in the application database. All the javascript code is previously
checked and sanitized before storing it into the database to avoid potential security threats.

- Unified data format to simplify the reuse of visualizations among different type of data sources.
- The visualization API provides an easy way to create new visualization plugins.
- The data provider API allows to integrate new data sources.
- Visualization components support linking-and-brushing features (when the user moves the mouse pointer over an object in a visualization component, related data in other visualization components get highlighted).

Although Epiviz provides a convenient platform for visualization and analysis of genomic data, it lacks support for live collaboration where other users can get immediate feedback of others’ actions. Another shortcoming is the lack of a central repository where users can store their custom code, so later such code can be shared and reused by other users. In general, the collaborative approach of Epiviz was designed for asynchronous-distributed scenarios.

![Figure 3.9: The Epiviz application available at their demo website](image)

### 3.3.2 CompPhy

CompPhy (Figure 3.10) is a web-based application for comparing phylogenetic trees allowing real-time remote collaboration [67]. It is built on top of PHP and MySQL.
A script called ScriptTree, located on the server side, parses the input text data to generate the tree images that will be delivered to the client side, these tree pictures are generated as SVG images. In addition, a set of Perl scripts residing also on the server side allow to make modifications on the tree structure. Trees, projects and any other information that need persistence are stored in the database.

In terms of collaboration, the platform provides both synchronous and asynchronous mechanisms of communication among users. For asynchronous collaboration, the application provides a forum per project where researchers can discuss topics related to the analyses, a todo-list, a history log of completed task on the project timeline and the ability of uploading external documents related to the project.

During synchronous collaborative sessions, only one user at time is allowed to make changes to the shared workspace. This is enforced by a system where participants have to request manually the control of the interface in order to make changes to the data. However, all other participants that are connected will receive almost immediate updates when a change has been completed. This manual approach of a user requesting control is how the developers of this platform tried to address the problem of concurrency while editing in a shared workspace.

Even that CompPhy yields synchronous collaboration in distributed scenarios, its scope is heavily targeted on the task of comparing and analyzing phylogenetic trees, with no further possibility at the moment of adding or supporting other types of visual encodings to enrich the data analysis.
3.3.3 PBrowse

PBrowse is a web-based genome browser tool (Figure 3.11) that allows to visualize and inspect large genomic data by providing live collaboration among users on a group session. Its collaboration system is based on the existence of users, roles and groups. A user admin can create a group where other users can join. While users are sharing a group their genome browser views will be synchronized whenever one of the users perform an action that update the current view such as loading a file or editing a track.

Synchronization of the shared view is orchestrated by a central server and the communication with the clients is done through websockets. Notifications of any change occurred on the shared view in the genome browser are sent from the server to every client with a message containing a particular instruction to be executed. The genome browser client recognizes 35 different types of instructions that can be executed.

Users with the respective privileges can create comments on the loaded tracks in the genome browser. In addition, an online chat is provided to foster the discussion among participants. A MySQL database is used as a central storage for users metadata related to their comments, files and session identity.

PBrowse provides a convenient way of holding real-time collaborative sessions while visualizing genomic data. However, in the same way as CompPhy, its scope is limited only to analyze data visualized in the genome browser. It does not provide a way to extend or incorporate other types of visualizations.

3.4 Summary

Relevant literature related to previous work was covered in this chapter. We examined the evolution of the web technologies and their impact on delivering visualizations to users. Moreover, javascript libraries for visualizing bioinformatics data were reviewed. At the end of the chapter, we presented three bioinformatics web applications that provide collaborative visualizations.

The three presented bioinformatics web applications (Epiviz, CompPhy and PBrowse) contain limitations in two major areas: synchronous collaboration and extensibility of their visualizations. Epiviz does not provide synchronous collaboration at all. CompPhy and PBrowse do provide synchronous collaboration but both are re-
Figure 3.11: PBrowse genome browser. A collaborative session is shown as depicted in [98].

Restricted to a specific type of visual encodings (phylogenetic trees and genome browsers). They do not provide any mechanisms to incorporate other types of visualization into their shared view.

LivelyViz (as described in Chapter 4) copes with these limitations by providing synchronous collaboration in its visualization components and the possibility of incorporating third-party developed visualization libraries.
Chapter 4

Technical approach

In this chapter, we provide the reasons for selecting Lively as our development platform. We also cover the technical details of the implementation of our collaborative visualization components. In addition, we present the visualization design concepts behind the collaborative features that LivelyViz components provide.

4.1 Why visualizations using Lively Web?

Lively Web is a web development environment that uses javascript as its main programming language. The aim of this platform is to provide an environment to develop web applications in an easy and friendly way using the least amount of programming code, because the platform already provides pre-built components called morphs [88] that can be manipulated using just mouse actions such as clicking, dragging and dropping. Most of the properties of these objects—such as scaling, coloring, rotation, position, combining and grouping with other elements—can be modified through an intuitive interaction by using only the mouse. In addition, every component can also be customized via a code editor using javascript language to achieve more advanced and specific behaviours.

Morphs can be modified, extended and combined with other morphs to create more complex objects that can be then saved as new morphs and then shared through a central morph repository in Lively called Partsbin. Once a component is saved into the Partsbin, it can be reused countless times by any other Lively users. This is a powerful feature that fosters collaboration and reusability of components through the entire platform and it is a great aid in developing UI components and visualizations.
In Lively, every user has their own workspace or folder on the server where they can create webpages called worlds. Worlds can be used as a workspace where users can drop and manipulate morphs. It is important to highlight that every world and morph saved into the Partsbin is versioned through a wiki mechanism that allows any user to submit changes. Further, every change is recorded. Like this, users can go back and review or revert to a previously saved version of a world or morph saved in Partsbin. This feature benefits the collaboration among component developers, and even regular users who wish to copy and extend other users’ worlds.

Because Lively is a web platform that relies heavily on javascript, it can be accessed from any computer or device that has installed a compatible web browser such as Google Chrome. This yields a wide range of accessibility, both in terms of devices and operating systems. Javascript code can be directly evaluated as text in a Lively world (by highlighting the text and pressing CTRL+D or CTRL+P), this feature along with the built-in presence of a code editor, debugger and a code versioning system provide a complete functional online development environment to developers without installing any additional software, and also simplifying greatly the arduous task of debugging and testing. Also, it is worth to note that by choosing Lively/javascript as a development platform –rather than a compiled language (such as Java and .Net) and a local development environment– reduces the typical development cycle of coding→compiling→debugging→testing to just coding→debugging→testing. For these reasons applications developed in Lively will always reflect their last version of changes without the user having to install or download anything.

The development of web applications with Lively using javascript opens the door to a plethora of mature libraries developed in the same language for achieving multiple purposes. Over the past years, developers had became interested in providing more libraries to enhance the user experience of web applications by leveraging the capabilities of HTML5 [46].

It is possible to import and use external javascript libraries in Lively. We used this feature to import the popular visualization library D3.js [49] for developing our own visualization components that will be described later. Moreover, external visualization libraries that are ready to use can also be incorporated into our platform, like we did with the external visualization component called Protael viewer [97].

Another important component of the Lively server architecture is the possibility of creating subservers. Subservers are server side code that can be executed onto the Lively server allowing to provide REST web services or web sockets services. Each
subserver is a server side endpoint based on NodeJS which means that javascript language can be used to code such services. This is a really powerful feature that provides access to a wide range of javascript server side libraries and projects that have been already developed and tested by third party developers. The ability of NodeJS to install plugins and third party libraries through the Node Package Manager tool (npm) makes easy to add additional libraries support that ease the development of large applications. For the example case used in this thesis, a few specific libraries have been used from the npm repository for coping with specific requirements/tasks such as sequaize, a package that yields database abstraction supporting several database engines and database communication.

Finally, Lively provides its own network communication protocol and libraries to create live collaborative web applications. This very important feature will be explained with more detail in section 4.4.1.

4.2 LivelyViz: visualization collaborative components

This thesis intends to propose, explore and develop a set visualization components in Lively that yield a user-friendly, collaborative and intuitive visualization experience when dealing with datasets regardless of the nature of the information to be portrayed.

These components will be treated as new Lively morphs and should meet the following criteria:

- **Reusable**: the components should be able to be ready to use and have as many copies of them as desired into the screen.

- **Individual**: each component should encapsulate all the functionality required to operate as standalone components when dragged into the workspace area

- **Be able to interconnect**: components should be able to be connected to each other to exchange data and UI events locally, where local means inside the same webpage or workspace area.

- **Collaborative**: the exchange of data and some UI actions should be possible between two or more components that are placed in different workspaces/webpages, browsers, devices or physical locations.
• **Extensible:** customization via programming should be allowed to advanced users and developers who wish to extend, modify and incorporate new behaviours or functionalities to every component.

After considering these keys features that every component should have, we decided that Lively provides a suitable software architecture to develop such kind of elements. Table 4.1 shows every desired feature for the visualization components discussed in this thesis and which elements Lively provides as a base to achieve those desired features.

### 4.3 Methodology

The key element in Lively used for developing the visualization components described in this thesis is the HTMLWrapperMorph. Although Lively morphs encapsulate most of the technical details involved when developing web applications –usually combining pieces of code written in different scripting languages such as HTML5, javascript and CSS3– by providing intuitive visual interactions and friendly dialogs, there are still cases where it is desirable to have a fine-grained control on how the application is built through the traditional web programming/design approach. Among the cases that can benefit from using the traditional web development approach are:

• Better control when formatting and positioning in the screen dynamic generated content

• Importing, reusing and injecting into a webpage external javascript libraries for creating mashups and complex user interfaces

• Implementing complex interactive visualizations using Scalable Vector Graphics (SVG) or Canvas HTML5 tags

• Improving performance by building components and visualizations that do not rely on too many submorphs

The HTMLWrapperMorph renders its content as plain HTML, which can be parsed and displayed by the browser. However, it also provides all the features and benefits that a morph can provide such as graphical manipulation, encapsulation, duplication, be able to be shared and programming via javascript.
<table>
<thead>
<tr>
<th>LivelyViz Component Features</th>
<th>Lively elements that can be used to build the desired visualization components</th>
</tr>
</thead>
</table>
| Reusable                    | ● Morphic architecture  
                               ● Partsbin                   |
| Individual                  | ● Morphic architecture  
                               ● Code editor                |
| Be able to interconnect     | ● Morphic architecture  
                               ● Lively events binding architecture |
| Collaborative               | ● Lively Network architecture  
                               ● Lively-To-Lively protocol (in short L2L) [31]  
                               ● Partsbin                   |
| Extensible                  | ● Code editor             |

Table 4.1: Desired features in the LivelyViz visualization components and the Lively Web core components that will be used for meeting such desired features.
With all the benefits that the HTML morph yields, the D3.js visualization toolkit is imported as a required library to create visualizations that will be displayed as the morph content. All these visualizations are based on SVG HTML tags, providing the advantages of being able to reside in memory as objects and modified on-the-fly without having to erase and redraw objects painted in the visualization area like when using the canvas HTML tag. Another advantage of using SVG is that the dynamic content of the generated visualization is just plain XML text, making it easy to store and transfer. This feature is leveraged to send the content of every visualization to the server to be processed and allow download of the content for the visualization as an SVG or PNG image.

The format conversion is managed by a subserver that was implemented in Lively for this specific purpose. The subserver makes use of the npm packages svg2png for managing the conversion from svg XML text to a png image and phantomJS, a server side implementation of a webkit browser and it is used to parse the SVG XML text that is sent from the visualization UI. It is worth noting that svg2png did not generate the PNG image properly when using the Lively world server located on https://lively-web.org because the package requires a higher version of NodeJS server than the one currently used by the global server. However, the implementation was tested on a local installed Lively server using the most recent stable version of NodeJS and it worked properly.

4.4 Visualization design concepts

Isenberg et al. [78] describes the scenarios where collaborative visualization can occur in terms of time and space using the matrix portrayed in Figure 4.1. LivelyViz components were designed with aim of primarily be used in distributed and synchronous collaborative scenarios by leveraging all the collaborative features that Lively as a platform yields. Of course, every LivelyViz component can also be used in the other collaborative scenarios described by the time-space matrix.

In 2008, Heer [73] proposed some design guidelines for developing asynchronous collaborative visualizations. Some of these guidelines were applied during the development process of this thesis, adapting them to a distributed synchronous collaborative environment. These design considerations proposed by Heer that were used in this thesis are:
Figure 4.1: Time-Space matrix showing the possible scenarios of collaborative visualization.

- Division and allocation of work
- Common ground and awareness
- Reference and deixis
- Group dynamics
- Consensus and decision making

The remaining part of this section explains how every design principle was applied in our research to develop our visualization components.

4.4.1 Division and allocation of work

LivelyViz components were designed to work as individual standalone visualizations, but also ready to be used in a distributed synchronous environment. The design
considered that users connected to a collaborative session, can receive instantly visual feedback on the actions performed by a remote peer. In every component there is a set of tasks—such as coloring a gene, loading a genome and creating an annotation—that are expected to be notified and replicated to the other peers connected in a group session in order to collaborate effectively. All these tasks need to be identified by the developer beforehand and then the respective programming inside the component must be implemented.

Following the principle of dividing the work into small units—either just for notifying other users about a performed action or a small task that creates new content such as annotating—can help developers to easily define how their widgets are going to react to a received remote notification or to a remote interaction that needs to be replicated locally. This also encourages modularity and reusability of the code and at the same time encapsulates all the technical complexities required to work in a collaborative environment inside the component, making them ready to use by users. By leveraging this design principle, technical barriers for establishing a distributed collaborative session among users are minimized fostering the discussion, analysis and production of content based on collaborative efforts.

In Lively, we achieve this feature by making use of the method `lively.net.SessionTracker.registerActions` for defining services in a world. This method allows to register actions that will triggered when a specific instruction is received remotely by a peer (Figure 4.2). Listing 4.1 shows how to define a new service in a Lively world to respond an incoming remote message called “myHelloService”.

```javascript
var myDefinedServices = {
    myHelloService: function(msg, session){
        alert("Hello" lia);
    }
}

lively.net.SessionTracker.registerActions(myDefinedServices);
```

**Listing 4.1:** Defining a L2L service in a world

In addition, to send a message to a remote peer requesting the execution of a specific action, the method `lively.net.SessionTracker.getSession().sendTo()` can be used as shown in Listing 4.2.

```javascript
var message = {};
var sess = lively.net.SessionTracker.getSession();
```
Figure 4.2: A sequence depicting how actions can be execute remotely by demand. 1) Shows two users viewing the same world. The morphA registers an action that shows the message “Hello” when receiving the instruction “myHelloService” from a remote user. 2) User2 sends an instruction to user1 asking for the execution of the action “myHelloService”.

Listing 4.2: Sending a message to request a remote action execution

```java
3 //request the execution of myHelloService() in the remote peer specified by remoteClientId
4 sess.sendTo(remoteClientId, 'myHelloService', message);
```

However, the aforementioned method will register actions with a unique name globally in the whole world. If an action is defined using the same name of a previously registered action, this will override the definition of any previously defined action with that name.

An example of this limitation is illustrated by Figure 4.3. The example portrays the following case: there is a world called worldA that will be used by two different users (user1 and user2). worldA contains two morphs: morphA and morphB. Each of these morphs encapsulates the code to register the action myHelloService. When user2
Figure 4.3: An example showing the default behaviour when two different morphs register actions using the same name. 1) Both morphs register the same action. When morphB registers its own copy of the action “myHelloService”, it overrides the action defined by morphA because both are using the same name and actions are registered globally per world. 2) User2 sends an instruction to user1 asking for the execution of the action “myHelloService”, the action is only executed once even if each morph had registered the action as a service of the world.

This default behaviour can be a problem for achieving modularity and isolation principles when designing components for LivelyViz. For example, having two copies of the same type of morph (e.g. two copies of a circle) will result in a name collision when each of them try to register their own copy of their actions, resulting in the execution of only one set of registered actions for one morph. Another problem would arise in the scenario where having two morphs of different kind (e.g. a square and
a circle) but both having a common set of methods with the same name but with a
totally different implementation such as \texttt{getArea()} or \texttt{getPerimeter()}. In some
cases, it may be desirable to have this case where the name of methods should be
the same and standardized across different types of morphs. In such scenario, if a
remote user sends an instruction to a worldX having a square and a circle requesting
the execution of the method \texttt{getArea()} only one method will be executed (the one
that was most recently registered).

For LivelyViz components, these scenarios are present because we want that every
component be collaborative and be ready to execute code under remote demand, but
at the same time to be able of having as many copies of the same type of component
as required by the user. Also, different types of visualizations component should be
able to coexist in the same world and interact one with each other without interfering
with the normal functioning of any component on the world. It is also desirable for
modularity and code maintainability to have an standardized set of methods across
any type of visualization component to perform common tasks such as the inter-
communication among components in the same world and the remote communication
with other components in other worlds.

To cope with such scenarios, we propose two different approaches: registering
actions inherent to every kind of LivelyViz components and registering general ac-
tions where any type of visualization component can subscribe. These two cases
describe remote communication among components located in different worlds. Local
communication among components in the same world will be covered in section 4.4.3.

In order to have every morph of the same type responds to a common remote
instruction, we proposed the implementation of a method where every morph will hold
the definition of all the actions to be executed on remote demand. This definition should
be in the same way as defined previously on listing 4.1 however before registering
the object containing the definition of all actions, every action should be renamed by
appending the name of the current morph to every action name. Once the renaming
process is finished, then the object containing the renamed actions can be passed to
the registerActions method. Obviously, if we have two morphs having the same name,
this would provoke the same behaviour we are trying to avoid. A possible option to
deal with this situation could have been solved by using a unique identifier that every
morph have under the property named “id”, however we decided to stay away from
that property and use the morph name instead because the field “name” can be easily
modified in every morph using a visual dialog launched from the morph halo.
menu. Also, Lively automatically renames a new morph that is placed into a world if there is an existing morph using the same name as the default name of the new morph. Listing 4.3 shows an example with the minimum code required to implement the approach previously described.

Listing 4.3: Proposed approach to define collaborative isolated actions per morph of the same type

In the same way that actions are renamed when they are defined, the same renaming scheme should be used before sending a message to a remote peer (Listing 4.4).

Listing 4.4: Rename is also necessary when sending a message to a remote peer

For the scenario when is desirable to request the remote execution of a method that
is shared by several components of different types, we propose defining general events to which every component must subscribe in order to be notified. The idea is to have a global available data structure per world to hold a relationship between the name of a specific general action and a list of morph names that will subscribe to that action. In this way, when a remote request to execute a general action is received, a search can be done in the world-available data structure and lookup for all morphs subscribed to the requested action and trigger such action in every morph subscribed. This is special useful when the source of the message is originated from the server side. With this approach, after an event occurred in the server side, the event can broadcasted to all the subscribed components in the world no matter what type of morph they are. Listing 4.5 shows the function used to subscribe a morph with a specific action name. Listing 4.6 define the list of general events that will be shared among component that invoke this function, while listing 4.7 shows the function responsible to execute an action in every morph subscribed to a general event.

```
function registerForGeneralFunctionExecution(morphName, actionName){

  // The data structure for holding the general actions and subscribed morphs will be stored in the $world object
  if(!$world.generalFunctionsWidgets){
    $world.generalFunctionsWidgets = {}; 
  }

  if(!$world.generalFunctionsWidgets[actionName]){ 
    $world.generalFunctionsWidgets[actionName] = {}; 
  }

  if(!$world.generalFunctionsWidgets[actionName].widgets){ 
    $world.generalFunctionsWidgets[actionName].widgets = [];
  }

  $world.generalFunctionsWidgets[actionName].widgets.
  pushIfNotIncluded(morphName);

}
```

**Listing 4.5:** Function for associating a morph name with an action name. A global available data structure is used for this.

```
function registerGeneralEvents(context){
```

```
var self = context;

// Define a set of common events (actions) to be shared among any component that subscribe.
var generalEvents =
{
    notifyPeersGroupDisbandedByAdmin: function (msg, session){
        // Just pass the current call control a to a local widget method implementation
        if (! isCurrentUserWorldAuthor()){
            executeAGeneralFunction("generalDisconnectGroupSession", []);
        }
    }
};

// This line call the function to subscribe a morph to a specific actionname defined in the object generalEvents
registerForGeneralFunctionExecution(self.getName(), "generalDisconnectGroupSession");

lively.net.SessionTracker.registerActions(generalEvents);

Listing 4.6: This function define all the general events available in the world and also associate the morph that invoked this function with every general event

function executeAGeneralFunction(actionName, arrParams) {
    // Get the list of components that are registered to the action specified by actionName
    var arrMorphNames = $world.generalFunctionsWidgets[actionName].widgets;
    if(arrMorphNames && arrMorphNames.length){
        // Execute this method with all registered morphs
        arrMorphNames.forEach(function (morphName){
            $morph(morphName)[actionName].apply($morph(morphName), arrParams);
        });
    }
}

Listing 4.7: Execute a specific method in all the morphs subscribed to an actionName
4.4.2 Common ground and awareness

As described by Heer [73], common ground is about sharing some basic knowledge that will be used as a base to have an effective interaction among peers.

He proposed the ability to share the current state of a visualization among other users through the use of parameters that can be passed via URL in the context of collaborative asynchronous environments. With such kind of approach, the visualization could also be embedded into external websites, blogs or forums.

Given that we focus on collaborative synchronous distributed environments, our approach is different. We propose the use of world URLs as our mechanism to start the collaboration and discussion inside of a Lively environment. For example, a world containing a collaborative dashboard to visualize genomic data will have its own URL. The acknowledgment of this URL to all participants will constitute the common ground to start the collaborative session.

Each world has also an author who is the user who created that world. We designed a LivelyViz component that will be the responsible to handle all the collaborative session management, we called this component the group session manager component (Figure 4.4).

The group session manager component only allows the author of the respective world to create a collaborative session and to invite other online Lively users to join to such session. Other users different than the author will receive a chat instant message containing an invitation to visit the world URL where the collaborative session will start.

Figure 4.4: The session manager component. 1) Shows the interface for the world author. There is a dialog that allows to invite people to the collaborative session. 2) This is the interface for other users different than the author. A dialog to select a session to join is shown.
be held. Non-author users can visit the link and view the same visualization interface and decide if they would connect to the group session. If a non-author user would like to hold their own session and invite people, they can just save-as that world in their own personal folder and now they will be the author of an identical copy of the original world containing the visualization components, allowing them to invite other users.

Additionally, content of the state of every visualization component can be exported to a PNG or SVG image file that can be distributed and reused externally in other platforms (such as blogs, wikis, forums, personal websites, digital documents) and publications.

In order to support a discussion model on our visualizations, we provide a simple annotation model (Figure 4.5) that allow users to create comments in a specific visualization view. In the genome map component, users can create comments in any area of the map. These comments are stored in a SQLite file at the Lively server delivering the web application, and they are always associated to a specific genome. Annotations are only editable by their authors but can be seen by any user.

For every component, there is a button to enter in the “create annotation” mode. Under this mode, users can create a new annotation when clicking on any region inside the visualization component. Another button allows to list all the annotations related to the current selected genome. Annotations are represented with a small icon that is placed into the visualization area of a component. Usually, all the annotation icons won’t be visible until the user click the “List annotations” button, which will show the window containing all the annotations for the selected genome and it will display all the annotation icons in the visualization area. Every one of these icons are clickable and allow to edit a selected annotation, however, when users have entered in the “create annotation” mode, click events on existing annotations will be ignored. Additionally, when an annotation icon is hovered by the mouse, a popup dialog containing the details of the selected annotation will be displayed.

During a collaborative session, remote peers are notified when a user creates, edits or deletes an annotation. If a peer receives a remote notification that an annotation event has occurred, he will be acknowledged in two different ways depending if annotations icons are visible in his visualization component(and therefore the annotations list window too). When the annotation icons are hidden, peers will be notified by a small popup dialog at the right-bottom corner of the visualization component. If the annotations list window is visible, the visualization area will be automatically updated.
in that peer by creating or removing new icons depending on the type of annotation event received (create, edit or delete).

Figure 4.5: Annotations in the genome map component

User awareness is very important when dealing with synchronous collaborative web visualizations, because it provides a sense of what other connected users are doing during a collaborative session. In 2012 Heinrich et al. [74] proposed a generic awareness infrastructure (GAI) and a blueprint to build reusable awareness widgets.

The GAI (shown in Figure 4.6) proposes that in every client there must be a Generic Awareness Adapter (GAA) component, which is responsible of serializing/deserializing messages containing information about events fired in the client side and send them to the server side where an awareness service exists. The awareness service will be responsible to process the messages and redirect them to the other clients. In this way, the GAA components that are present in the other clients can receive these messages, process them and update their visual interfaces with the changes made by the user who fired the event originally.

We used some of the ideas proposed by Heinrich et al. when designing our collaborative widgets. We developed a basic user awareness morph (Figure 4.7) that can be reused through all LivelyViz components to produce a visual feedback of who is performing an action during the collaborative session. The widget will show the
name of the user performing the action and a specific color that will identify the user during the entire session. This morph also encapsulates all the logic to detect the peers involved in the collaborative group and it will take care of updating the status of the participants in the group, for example if a new user joins the component will be able to track that change and render correctly the name and the color assigned to the new user.

A subserver was also developed to handle directly the requests received by the awareness component. The subserver handles all tasks related with the managing of colors assigned to the users in a group session. It uses the color scales defined in the npm package d3-scale [25]. When a new user joins into the collaborative session, a new random color is picked from a predefined scale of colors.

This subserver also sends a broadcast message to all morphs implementing the user awareness morph when an event has occurred in the server side, e.g., if a user requested a new color or wants to modify their own color. Under this kind of scenario, the architecture to handle this kind of broadcasting messages by the LivelyViz components discussed in section 4.4.1 comes very handy.

Another mechanism of awareness was implemented in the general session manager with a button to show a list of participants (Figure 4.8) taking part in the current collaborative session. The list is automatically updated as participants join or leave the session.

4.4.3 Reference and deixis

When people share a common workspace of any type (desk, table, room, screen, etc), it is very common to reference objects or spaces by pointing them and use words such
Figure 4.7: The user awareness morph is shown on top of the LivelyViz component when a user is dragging the window around the world. In this particular example the red color assigned to the user performing the action is also used to color the border of the window to accentuate the visual cue of who is performing that action in the shared workspace as “this” or “that”. Usually these kind of words requires additional information to be completely understood, so it is necessary establishing first a referential system of time, person or space depending of the circumstances in order to achieve full understanding among all the participants (common grounding).

In our visualization components we use the linking-and-brushing technique as a way to point and emphasize regions of interest inside a visualization component. For example we can hover a gene inside the gene map component and have that particular gene highlighted at the scatterplot(Figure 4.10). This is achieved by connecting the two components through events using the Lively Morphic Connection API.

Morphic connections allows to connect morphs either by attributes or by methods. The idea is having a “source” morph which will have an attribute or method that will trigger the execution of a method on the “target” morph. This connection can be done either visually or programatically. Listing 4.8 shows the a simple connection between two morphs: a list and a circle (Figure 4.9).

```
1  $morph('List').setList(['Red', 'Blue', 'White']);
2  lively.bindings.connect(this, 'selection', $morph('myCircle'), 'setFill', {
3      converter: function(colorName){
4        console.log(colorName.toLowerCase());
    })
```
Figure 4.8: Morph showing the list of participants involved in a collaborative session. This functionality was implemented in the general session manager morph.

Figure 4.9: Two morphs connected through the Lively Morphic Connection API. When a user click on the list, the circle changes with the selected color.

Listing 4.8: Connecting a list morph(source) with an ellipse morph(target). When the property selection changes in the source morph, it triggers the execution of the method setFill in the target morph. A converter function is used to convert the plain color string into a Color object recognized by the method setFill.

Although this is a pretty simple way of connecting two morphs, we proposed a different and more convenient approach for connecting our LivelyViz components. The idea is defining at least three methods in every component: one for registering a list of target morphs to notify, a method for notifying all the morphs registered as targets and a hub method for processing any request received from any other morph.

The method `connectWidgets()` (Listing 4.9) will be responsible to register connection between the current morph and every target morph defined in a list. The connection will be set between the local method `notifyLocalConnectedWidgets()` in the current morph and the execution of the `processCommandFromConnectedWidget()` method in every target morph.

```javascript
function connectWidgets(){
  var notifyThisWidgets = ['D3GeneMap',
```
Listing 4.9: This method specifies a list of target morphs and then establishes a connection between the current morph and every target morph. Every time the current morph executes the method `notifyLocalConnectedWidgets()`, the `processCommandFromConnectedWidget()` method will be executed in every target morph. The method in the target is executed with the returned value from the source method as an input parameter.

When the morph connection has been created between the source component and one or various targets components, the execution of the `notifyLocalConnectedWidgets()` method in the source component triggers the execution of the `processCommandFromConnectedWidget()` method on every target component. Listings 4.10 and 4.11 shows the basic implementation of these methods. This approach helps to maintain a simple way of connecting visualization components by always relying on the connection of the two methods previously described in a pair of source/target morphs. By simplifying the connection approach, it is easier for the developer to add, maintain and track all the methods that can be requested for execution by another morph. Moreover, by using a list of target components, events that fire multiple actions in other components are possible mimicking a broadcast event message sent
by the source component to all the target morphs. This approach is useful to build event-oriented actions in the UI that can help integrate different type of visualization components to display more useful information at the same time. Additionally, by encapsulating in each type of visualization component this mechanism to react to other components actions, increase modularity and reusability of the components allowing the users to be able to compose their own type of dashboard view.

Listing 4.10: This code sends a notification to a target morph requesting the execution of a method with the given arguments.

```javascript
this.addScript(function notifyLocalConnectedWidgets (action,
   argumentsF) {
   var self = this;
   return {
      sender: self.getName(),
      data: {
         action: action,
         argsF: argumentsF
      }
   };});
```

```javascript
function processCommandFromConnectedWidget(msg) {
   /*{
      DOC:
      sender: morphNameSender
      data: {
         action: '', // action requested to execute
         args: {} // arguments for action to execute (An object)
      }
   }*/
   var self = this;
   var action = msg.data.action;
   var argsF = msg.data.argsF;
   var listActionsAllowed = {
      loadGenome: function (argsF) {
         self.doSelectAfterGenome(argsF);
      },
```
Listing 4.11: This method intercepts any notification from the source morph, decompose the message object and execute a local method specified by the "action" parameter. All actions that can be requested to execute by a source morph must be defined here.

Figure 4.10: Linked interactions between LivelyViz components. The linking-and-brushing technique is used to highlight a gene in the scatterplot.

4.4.4 Group dynamics

One of the core features of LivelyViz components is the ability to be used in distributed collaborative scenarios by creating group sessions where collaborators can join and obtain instant feedback of some of the actions that their peers perform on the visualization components. The client side part has been described previously by explaining the session manager morph to create collaborative sessions and invite
people to join. A key component in the whole architecture is played by the server side which implements a subserver for orchestrating all the concurrent capabilities required to host a collaborative group session.

In 2016, Hemmings [75] proposed a scalable communication protocol for handling group messaging among applications in Lively, this service was built on top the regular L2L messaging services and it was called “Group tracking”. The Group tracking protocol relies on a central tracker that will hold groups of users. These groups can be defined by an admin user and they work similarly as a chat room application would, where the “rooms” are represented by the name of the applications that will be shared by a group of users. A user can define as many groups as desired and a group can hold virtually any number of users. Even subgroups can be defined inside groups to segment users that would wish to work inside the same group but with a more restricted number of participants, this subgroups are called “partitions”.

We built on top of this communication protocol, our server side collaborative architecture to yield LivelyViz components with the ability to communicate among them and work collaboratively. However, we extend the original protocol and add a few features that are useful for our kind of application but at the same they will serve as a contribution to enrich the group tracking protocol. In addition to the existing basic functionality of the protocol (creating, joining, abdicating and resetting a group), we introduced the following new features:

- A simple list of invited people that will be used to enforce a policy that only users in this list are authorized to join the group. All the basic services to handle this list were also implemented (register invitations, validation when a user try to join, delete invitations, etc.)

- Broadcast messaging notification to all participants from the server side. This is a really important feature that we used a lot in our architecture to notify all users when an event has occurred on the server side or some important event originated from the client side (e.g. the admin user disbanding the group and ending the collaborative session).

- Methods for synchronizing the list of group members and the status of the collaborative session across all participants in the group.

As described previously, the session manager will allow the author of a world to create a collaborative session by inviting people to join. In this case, the world
author will become an admin user of a group which name will be taken from the name of the world present in the world URL. For our components we do not make use of the partitions features and we always assume that all users that join to a group are working on a default partition, so actually this feature is totally transparent for application users. This was made for the sake of simplicity but of course our approach can be extended to support subgroups interactions on the LivelyViz components as future work.

4.4.5 Consensus and discussion

LivelyViz components were designed to promote and encourage collaborative interactions around a set of visualizations and to help users to start discussions about their analysis and results. At this stage, these discussions can be done through the built-in Lively instant messaging chat and the basic system of annotations that has been implemented. For sure, more work can be done to make every component more discussion oriented where a conclusion could be taken among all group members after implementing some kind of mechanism to present ideas and a way to rate or score those proposed ideas and come up with a group position regarding a particular subject. We believe that this can be part of the future work and research on collaborative interactions.

4.5 Summary

Throughout this chapter we discussed the technical reasons that led us choose Lively as a development environment for building our visualization platform. The required features that our visualization components should provide to users were described along with the visualization design considerations that guided our implementation. In addition, we explained how every visualization design principle was applied in our components using Lively.
Chapter 5

LivelyViz: Use, evaluation & discussion

In this chapter we describe the visualization components that we developed for our bioinformatics data dashboard. A brief description of the most notable features of every visualization component is given.

In addition, LivelyViz is compared against VOCs using a code metrics evaluation. Moreover, we discuss the benefits of LivelyViz not only as an analysis and visualization application, but also as a platform that can be extended by power users\footnote{Users who are able to use advanced features present in software applications, such as basic scripting programming, but who are not considered computer scientists or formal programmers.} and developers. Finally, some identified limitations in Lively and LivelyViz are addressed.

5.1 LivelyViz visualization components

LivelyViz visualization components that conform to our bioinformatics dashboard application are discussed in this section. Our platform was presented in 2016 at CASCON and CSER conferences\textsuperscript{[42, 41]}.\footnote{Users who are able to use advanced features present in software applications, such as basic scripting programming, but who are not considered computer scientists or formal programmers.}

Two visualization widgets were designed to illustrate how Lively can be used to develop collaborative visualizations. One widget is a genome map that shows the positions of every gene in a selected genome. The other one is a scatterplot that allows to select numerical fields from a given dataset and assigns them to the x-axis and the y-axis. These two components were developed from scratch using D3.js library. A third component was developed, using an external javascript library called Protael\textsuperscript{[97]}.\footnote{Users who are able to use advanced features present in software applications, such as basic scripting programming, but who are not considered computer scientists or formal programmers.}
This was done to illustrate how to import external javascript visualization components into Lively.

5.1.1 Genome Map

The Genome Map component (Figure 5.2) allows to connect to the VOCs database, select a genome from a specific family and plot the location, size and length of every gene of the selected genome. The selection can be done by clicking the button “Select genome”, which will invoke a component that we call “database chooser” (Figure 5.1). This procedure for selecting a genome is also used in the scatterplot component. It notifies other users during a collaborative session about this action.

Figure 5.1: The DB Chooser component. The left window allows to select the virus family and the right one allows to select a genome to plot.

Additional information about every gene is encoded using a color scheme. These color scheme encodes a numerical scale that represents a feature about a gene, for example: how common is a gene in viruses of the same family. When hovering with the mouse over a gene, detailed information is shown in a popup.

When clicking on any gene a menu allowing to select options related to the selected gene will be displayed. One option is to change the gene color (Figure 5.3), this action notifies remote peers during a collaborative session. The other option is displaying the amino acid sequence of the selected gene.

The Genome map component provides the possibility of downloading the current content of the visualization component as a PNG or SVG image. Creating annotations per genome is also possible, the mechanisms of how annotations work were described in Section 4.4.2. This component was also created to work in collaboration with the scatterplot component. In other words, users will get visual feedback in the scatterplot when interacting with the genome map component. In this way, researchers can
5.1.2 Scatterplot

The Scatterplot component (Figure 5.4) was designed to help researchers to explore visually their datasets in order to find patterns and trends. In the same way as the genome map component, the scatterplot component connects to the VOCs database and allows to load a viral genomic dataset. The process of loading the data is exactly the same as described previously in the genome map component, by using the “Database Chooser” dialog. Once the dataset is loaded, users can preview the loaded
dataset (Figure 5.5) and their columns. Every row in this dataset represents a gene in the loaded viral genome. The preview table showing the dataset rows and columns was developed using the handsontable javascript library [7].

![Figure 5.4: The Scatterplot component.](image)

![Figure 5.5: Previewing a genomic dataset.](image)

After loading the dataset, the next step is selecting from the dataset the numerical columns that will be used as the x-axis and y-axis in the scatterplot (Figure 5.6). This can be done by using the button “Select axis & plot” from the menu bar of the component. During a collaborative session, actions such as loading a genome, selecting axes and plotting will notify remote peers and their respective components will be updated with the new data.
Another important and useful feature in this component, is the ability of deriving a new column from the existing ones in the dataset. This new column can be used as axes when plotting a new scatterplot (Figure 5.8). This can be done by using the “formula editor” component, which is accessible from the menu bar by clicking its button (Σ). Figure 5.7 shows the formula editor dialog that allows to create formulas using the numerical columns from the original dataset and store the result as a new named column. On the left side, there is the list of all numerical columns available in the current loaded dataset. As future work, we plan to extend this dialog by allowing every user to store and retrieve their own formulas from the Lively server.

The scatterplot component also provides the feature for exporting the current visualization content to an image. Its events can also be interconnected to work at the same time with the genome map, to produce visual feedback when interacting with dots in the scatterplot. For example, when hovering a gene dot in the scatterplot, that specific gene can be highlighted in the genome map.
5.1.3 Amino acid sequence visualizer

In order to demonstrate the capabilities of Lively to import third party visualization libraries, we decided to incorporate the external javascript visualization library called Protael to visualize the amino acid sequence (Figure 5.9) of a selected gene. This option was added to the menu displayed when a gene is clicked in the genome map. The amino acid sequence is recovered from the VOCs database using the identifier of the selected gene. Protael supports loading more than one genomic sequence or quantitative data as tracks, creating annotations, visualizing alignments and adding markers in the sequence.

The use of Protael to visualize the amino acid sequence is not collaborative at the moment, however it can be easily extended to have that capability, as we did with our other two components, we leave this as future work. In fact, currently we do not leverage all capabilities of Protael. However, also this can be addressed, with the goal of developing more complex visualizations, in future work.

5.2 Evaluation

In this section, we cover the details of a code metrics evaluation that was conducted in order to find out if Lively, as a development platform, can benefit the productivity of developers when building interactive data visualizations.
Figure 5.9: The Protael visualizer incorporated in LivelyViz. This external library was used to show the amino acid sequence of a selected gene in the genome map.

We compare LivelyViz and VOCs projects, by using as a metric the number of physical lines of code (LOC) present in their respective source files.

5.2.1 Code metrics evaluation

This section describes a static code evaluation we did on source-code files for VOCs and LivelyViz projects\(^2\). As a metric of our evaluation we chose the numbers of physical lines of code (LOC). We present three bar charts for every project, depicting information about the number of LOC in every file: a general chart representing all files in the project, a chart depicting the top ten files with the greatest number of LOC, and a list of files involved in the visualization process.

Tools and methodology

As we have mentioned in previous chapters, VOCs was developed using the Java language and LivelyViz using the javascript language. The first step was to strip all comment lines and white lines from all source files in both projects. For our analysis we only took into account .java and .js as valid source files. In addition, any external library developed by third-party vendors were ignored. This means that only code developed from scratch by VBRC developers was taken in account in the case of VOCs. In the same way, we only measured our own code developed for LivelyViz, stripping away any external developed libraries files.

We used the tool called CLOC [1] to generate new versions of every original source code file that were free of comments and white lines. The second step involved running

\(^2\) VOCs was created more than 10 years ago and more than 15 co-op students have participated in its development since it was created. LivelyViz was developed within one year by a single graduate student, who is the author of this thesis.
CLOC again but this time against the newly generated source files to generate a file reporting the number of lines of code per file.

Results

VOCs client Java code is composed of 635 files and a total of 136223 LOC. These files are distributed in the packages:

- **ca.virology.vocs**: contains all the code related to the VOCs client.

- **ca.virology.vocs.lib1**: packages containing several libraries related to the client and server side. This package contains legacy code to connect to the VOCs database.

- **ca.virology.vocs.lib2**: packages related to the database connection.

Figure 5.10 shows a bar chart depicting all VOCs source files with their respective number of LOC, being the file **VOCSSQLSelect.java** the highest ranked with 3336 LOCS. This file contains the most common SQL Queries to retrieve information from VOCs database, it is extensively used in the GenomeMap viewer option.

![Bar chart showing total LOCS of VOCs project. The chart shows all sources files that compose the VOCs client](image)

**Figure 5.10:** Bar chart showing total LOCS of VOCs project. The chart shows all sources files that compose the VOCs client

In Figure 5.11 we can appreciate the top ten files containing the major number of LOC. The three first positions on this ranking are files containing source code related to database communication and SQL queries.
We found also that a total of 2907 LOC are responsible for the GenomeMap visualization in VOCs. They are distributed among the files `GenomePlot.java` and `GenomePlotter.java`.

After analyzing LivelyViz project code, we found that the total number of LOC for the project, including both client-side and server-side code, were 12723 lines. Even though most of the morph code is embedded into the component itself and accessible through Lively javascript code editor, we decided to distribute the code among javascript files depending on their role inside the application. Figure 5.12 shows the bar chart showing the general overview of number of LOC of all files in the LivelyViz project.

Figure 5.12: Bar chart showing total LOCS of VOCs project. The chart shows all source files that compose the VOCs client

In the same spirit as we did with VOCs, we provide a top ten chart, shown in Figure 5.13, depicting the files containing the most number of LOC. This time, the top three ranked files are the ones related to every major LivelyViz visualization component: the GenomeMap, the scatterplot and the gene amino acid visualizer based
on the Proatel library.

![VOCs top ten files with the major number of LOC](image)

**Figure 5.13:** VOCs top ten files with the major number of LOC

In the D3GeneMapTestVersion.js, only 857 LOC are direct responsible for plotting the entire genome in contrast to the 2907 LOC in VOCs. This means that VOCs source code for plotting the genome map is nearly 3.39 times larger than the amount of code required in LivelyViz. We took a closer look at two specific parts of the implementation of genome map plotting methods in both projects.

We found that for implementing the mouse event handlers around the arrows representing every gene in the map, in VOCs we required 135 lines versus 88 lines in LivelyViz. That is, in this particular case, LivelyViz requires less code for achieving the same purpose (Listing 5.2). This is mainly because events in Java require an implementation of the `ActionListener` class and when it is defined in place of the invoker method, the resulting code can be very verbose (Listing 5.1). In addition, in VOCs there is only one mouse event handler for the whole canvas, which means that in order to detect which gene was clicked, additional computations need to be performed to identify whether the position where the mouse was clicked is contained inside any of the gene arrows. On the contrary, in LivelyViz every gene arrow is an SVG object, so every of these objects can define their own mouse handlers directly to respond mouse events without requiring additional LOC of computations to determine if a click over one gene has happened.

```java
myCanvasObj.addMouseListener(new MouseListener() {
    @Override
    public void mouseReleased(MouseEvent e) {
    }
    @Override
    public void mousePressed(MouseEvent e) {
```
Listing 5.1: Minimal required code for implementing mouse events in a canvas in Java. Because the method `addMouseListener()` expects a class implementing the interface `MouseListener`, several methods must be implemented even if they have no content. In our example, an anonymous class is implicitly defined when defining a new `mouseListener` in place.

```java
@Override
public void mouseExited(MouseEvent e) {
}
@Override
public void mouseEntered(MouseEvent e) {
}
@Override
public void mouseClicked(MouseEvent e) {
    System.out.println("Mouse clicked!!");
}
```

Listing 5.2: Minimal code required to handle the click event in javascript.

```javascript
document.getElementById('myGene').onclick = function() {
    alert('Gene clicked!');
};
```

Another issue is related to the fact that in VOCs the graphics are plotted using the Java Graphics2D capabilities. This means that a lot of code is required to compute the position of elements, or to access a specific plotted gene shape to modify its features and then repaint the whole canvas.

In LivelyViz this is simpler because the visualization is based on SVG objects that are part of the Document Object Model (DOM) of the HTML webpage, and they can be referenced and accessed easily compared to the method used in VOCs, which actually traverses all the gene list elements. This produces repetition of code, especially nested for loops. In addition, the use of D3.js makes even easier to access, select and modify in batch features on genes that have been already plotted. D3.js implements fluent interface methods. A common code convention of using fluent interface is writing one chained method per line (Listing 5.3). With this kind of code convention, clarity in the source code is achieved at the price of producing additional artificial physical lines. So this actually means that the implementation using LivelyViz could achieve even fewer physical lines if it would have followed a
different code style convention where chained methods would be written in one line.

```java
var myObj = new myObj();
myObj.methodA().methodB().methodC().methodD().methodE();
```

**Listing 5.3:** Fluent interface in javascript. Notice that all calls to the chained methods could be written in just one logical line. However, for clarity sake, methods D and E execute in their own new line. This has created two additional physical lines.

The other code section that was compared among both implementations focuses on methods for drawing the entire genome map. In this case, VOCs required 780 lines versus 545 in LivelyViz. In this scenario, lots of extra lines were required in VOCs for formatting the font (Listing 5.4) that was going to be used for plotting labels inside the genome map visualization and for defining the gene coloring scheme. In LivelyViz, most part of the styling is left out of the programming and is achieved by leveraging the CSS editor that every morph has (Listing 5.5). Additionally, D3.js allows to define with a few lines color scales based on numerical ranges that can be used for coloring genes.

```java
private FontMetrics pickFont(Graphics2D g, String str, int xSpace, int ySpace) {
    boolean fits = false;
    Font font = g.getFont();
    FontMetrics fontMetrics = g.getFontMetrics();
    int fontSize = font.getSize();
    String fontName = font.getName();
    int fontStyle = java.awt.Font.BOLD;
    curFontSize = fontSize;
    while (!fits) {
        if ((fontMetrics.getHeight() <= ySpace)
            && (fontMetrics.stringWidth(str) <= xSpace)) { 
            fits = true;
        } else {
            if (fontSize <= minFontSize) {
                fits = true;
            } else {
                font = new Font(fontName, fontStyle, --fontSize);
                g.setFont(font);
                fontMetrics = g.getFontMetrics();
            }
        }
    }
    return fontMetrics;
}
```
private FontMetrics setFixedFont(Graphics2D g) {
    Font font;
    FontMetrics fontMetrics;
    font = new java.awt.Font("Courier", java.awt.Font.BOLD, 
        fixedFontSize);
    g.setFont(font);
    fontMetrics = g.getFontMetrics();
    curFontSize = fixedFontSize;
    return fontMetrics;
}

private FontMetrics setFixedFont(Graphics2D g, int p) {
    Font font;
    FontMetrics fontMetrics;
    curFontSize = fixedFontSize + p;
    font = new java.awt.Font("Courier", java.awt.Font.BOLD, 
        curFontSize);
    g.setFont(font);
    fontMetrics = g.getFontMetrics();
    return fontMetrics;
}

public void setFontSize(int s) {
    boolean rp = false;
    if (!fixedFont) { fixedFont = true; rp = true; }
    if (fixedFontSize != s) { fixedFontSize = s; rp = true; }
    curFontSize = fixedFontSize;
    if (rp) super.repaint();
}

public void enablePickFont() {
    if (fixedFont) {
        fixedFont = false;
        super.repaint(); // Repaint the whole canvas
    }
}

Listing 5.4: Font formatting related code in VOCs
In general, code sections related to the management, positioning and styling inside the visualization area, will be shorter in javascript code, at least if SVG is being used as the render technology. This is because the objects being drawn as SVG elements become part of the DOM (consequently residing in memory) and are independently accessible through a simple API. This means that the programmer does not need to take care of details such as making complicated computations to determine the position of a drawn object that we want to modify, or repainting the whole visualization area every time a new object is added or an existing object property is updated. In addition, building graphical user interfaces in VOCs using the Java Swing classes can require lots of LOC compared to Lively where the UI can be built graphically using mouse interactions—thanks the Morphic class UI library provided by Lively and that was originally implemented in Self—and then saved as a component for reusing it multiple times.

Morphic is characterized by inheritance from prototypes, composition, and a combination of direct manipulation and programming. In contrast, toolkits based on Swing rely on code configuration and have limited support for composition. That is why popular Java integrated development environments provide plugins that allow designing user interfaces in a graphical way.

Table 5.1 shows the numbers of LOC required for implementing common features present in both platforms such as gene mouse event handling, gene click event and genes plotting. Additionally, two non-common features related to building a part of the GUI in every platform were provided. Since these two features cannot be compared between VOCs and LivelyViz, a N/A (not applicable) message indicates that a feature was not implemented. Even though these two features cannot be compared directly, we provide numbers to have an idea about how many LOC are required when building user interface components for both platforms.

LOC might not be always a perfect metric when comparing projects implemented
<table>
<thead>
<tr>
<th>Features implemented</th>
<th># LOC VOCs</th>
<th># LOC Lively</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defining mouse event handlers for plotted genes</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>Implementation of the mouse click event handler</td>
<td>121</td>
<td>71</td>
</tr>
<tr>
<td>Drawing genes inside the visualization area</td>
<td>780</td>
<td>N/A</td>
</tr>
<tr>
<td>Building a menu bar in Swing with 61 clickable options</td>
<td>868</td>
<td>N/A</td>
</tr>
<tr>
<td>Building a menu based on HTML Buttons with 8 clickable</td>
<td>N/A</td>
<td>17</td>
</tr>
<tr>
<td>options</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 5.1**: Number of LOC used for the implementation of some features in VOCs and LivelyViz. Features not implemented in one of the platforms are represented with the text N/A (Not applicable).
in different programming languages, because this measure can be affected even by coding convention styles. However, it can help when comparing shared features among both projects. In addition, it can help having a better understanding about the language implementation differences and how this could impact the growth of the source code. Further analysis of numbers of LOC could lead to improve code quality by splitting and modularizing large source code files.

Even though number of LOC can be an imperfect metric in some cases, LivelyViz scored better with fewer number of LOC than VOCs. A reason for this may be that Java is a “well typed” language that requires a more strict definition on its data types when importing packages. This can contribute to the creation of extra physical lines in contrast to javascript, a loosely typed dynamic script language.

5.3 Discussion

LivelyViz is a proof-of-concept implementation that addresses the limitations of the collaborative visualization platforms described in Section 3.3. By leveraging the possibility of importing third-party javascript visualization libraries, LivelyViz is capable of extending its available options of visual encodings allowing it to compose useful and diverse data dashboards. LivelyViz was developed as an inherently collaborative platform. This was possible thanks to the collaborative features inherited by the Lively platform (Partsbin, L2L network protocol, Lively event bindings API), but also due to the work developed and described in this thesis in the technical chapter (Chapter 4). In particular, we provide synchronization of the shared view of all participants during a collaborative group session.

In addition, when developing web applications a higher level of abstraction is provided through the use of the morphic architecture. In this way, the developer does not have to deal with low level implementation details in a web application (combination of HTML, javascript and CSS distributed across multiple files). All the required javascript code is embedded into every morph component.

The web nature of LivelyViz provides immediate benefits such as ubiquitous access, simplified deployment and updates to users, code portability (javascript code can be also used to develop non-Lively based web applications) and code maintainability (less amount of files translates into less code to maintain). Additionally, LivelyViz architecture is firewall-friendly.

Moreover, Lively poses as a great alternative for quick application prototyping
and exploratory development research. Its features as a dynamic web development environment (liveliness, direct code execution and debugging, developer tools) can benefit development teams that need to build functional prototypes in short time to investigate the feasibility of a design on the web platform. This can save the valuable time of developer groups before embarking on choosing a development platform that could end in not being suitable for the entire project, wasting precious time and software development resources.

5.4 Limitations

Certainly, Lively has some points that we would like to see improved. One of these is the lack of a set of complex built-in GUI controls and a more friendly and effective layout system inside morphs elements. Currently, a set of basic UI morphs is provided (combo list, buttons, sliders). However, it would be great to have also complex UI objects (such as grids, tree lists, layout components) out-of-the-box. Of course, these kinds of components can be incorporated using external javascript libraries. However, it would be better to have an official set of these complex UI controls as morphs developed and maintained by the Lively core development team.

Security is another point that would be nice if addressed by the Lively team. Right now, users can log into Lively by simply choosing a username with no password. Again, a user friendly mechanism should be provided to allow authenticate in Lively. In addition, an authorization API could greatly save time for developers that require to restrict access specific parts of their components being developed.

Furthermore, even though server-side code is allowed to be developed in Lively through the use of subservers, these files are not versioned in any way. It would be very useful if subserver code could also be versioned to manage and retrieve previous versions of such code.

In the case of the current implementation of LivelyViz, there are some limitations that could be addressed in future versions. Currently when collaborating, there is no mechanism of concurrent control when interacting with the visualization components. Even though in our use case we are only reading the VOCs dataset, there might be scenarios where strict concurrent write access control would be necessary. We partially cope with this when users annotate genomes using our SQLite database located in the Lively server, by restricting that only the author of an annotation can edit or delete it.

Currently the use of LivelyViz on tablets is possible. However, only a few specific
collaborative actions—such as loading a genome, moving the components in the shared view or plotting in the scatterplot—are available. This is mainly because some of the core UI controls of Lively are not optimized to be used and displayed out-of-the-box on tablet devices. Also, some Lively components and options rely on the right-click button of the mouse, making them inaccessible or difficult to use from only touch based devices. Further work could be done both on the Lively UI components and on LivelyViz to adapt the UI to make it more user friendly and responsive when displayed on such kind of devices.

In the Genome Map and Scatterplot components, a zoom option could prove useful for inspecting easily specific areas of the visualization. This could benefit, for example, when there are dense clusters of points in the Scatterplot.

5.5 Summary

LivelyViz visualization components and their features were described in this chapter. A code metrics evaluation was presented comparing the amount of lines of code required in VOCs and LivelyViz to implement the same feature. We found that using LivelyViz resulted in fewer LOC than VOCs. We concluded the chapter with a discussion on benefits and limitations of LivelyViz & Lively.
Chapter 6

Conclusions and Future Work

In this chapter we summarize the work described in this thesis, highlighting the contributions of our research and finally addressing the future possible directions that this work can take. As a reminder, the research questions addressed here and which we formulated in the first chapter are as follows:

- RQ1: Can Lively Web provide a platform to develop effective interactive web visualizations?
- RQ2: How can Lively contribute to the developing of a collaborative oriented bioinformatics dashboard application? What makes it stand out from traditional development platform or methodologies?
- RQ3: Can Lively integrate pre-existing datasets and third-party visualization libraries into its workflow to extend the dashboard with additional visualizations?

6.1 Summary

Our research focused on providing a framework to develop effective web collaborative visualization tools that can assist, e.g., geographically distributed research teams with their data analyses in a visual manner.

We described in Chapter 2 the challenges experienced by researchers at the VBRC lab using desktop visualization applications that do not support live collaboration out of the box. We discussed the current architecture of the VOCs application along with its limitations, and we proposed an alternative architecture based on Lively that can help to achieve collaborative applications using the VOCs database. This
answers our first research question affirmative, namely whether or not Lively is a suitable platform for building collaborative web applications.

In Chapter 3 we discussed the role of the web as a platform for delivering interactive visualizations. We also reviewed some visualization libraries used for building 2D visualizations, in particular visualizations used in bioinformatics. The most common visual encodings used in the analysis of genomic and biology systems data were discussed in this chapter. Finally, at the end of the chapter we presented three web visualization applications that deal with bioinformatics datasets and that provide some degree of collaboration.

Chapter 4 answers the second research question by providing technical details about the methodology we followed to build a visualization dashboard tool that connects to a bioinformatics dataset, in this case to the VOCs database. We explained in detail the technical features present in Lively that made possible the development of our collaborative visualization platform. In addition, the visualization concepts that were used to design our components are also discussed in this chapter.

We developed two collaborative bioinformatics visualization components—the GenomeMap and the Scatterplot components—that can communicate with each other when visualizing the same dataset. Both components are described in detail in Chapter 5, along with the description of an external visualization library called Protael that we have incorporated in our platform. Such incorporation of an externally developed javascript visualization along with the fact of connecting to the VOCs database as our main datasource, answers our third research question. At the end of the chapter we also provide a brief analysis about how convenient Lively is to develop applications, we did this by measuring the number of lines of code required. We measured the number of lines used both in LivelyViz and in VOCs client project, and compared the results. Our findings show that using Lively javascript code can produce less physical lines of code. We illustrated this comparing the Lively javascript implementation to the VOCs Java implementation of the genome map visualization.

Given all the information provided in this present thesis, we can identify the main contributions of our research, which are:

- We provide a framework to develop a set of web visualization components that are reusable, extensible and ready to use in collaborative distributed synchronous environments. As a use case we developed collaborative bioinformatics visualization components. However, our work can be easily applied to a different kind of
situations and datasets. Components can be extended through programming and shared among other users and developers fostering the reusability and collaboration.

- Our platform is based on a thin-client architecture which provide benefits of simplifying the maintenance, deployment and accessibility to the visualization applications. This also reduces the amount of data travelling over the network by sending only small messages to keep synchronized the shared workspace and provide live awareness feedback to all participants during a group collaborative session.

- The web-based architecture of our platform provides ubiquitous access over the internet and allows the access from any operating system and device (desktop or mobile) that supports a capable web browser.

6.2 Future Work

The conducted research opened new questions that can lead to further work over our platform.

What is the impact of our platform in the productivity of large research groups when analyzing data remotely? A empirical study [62] would be necessary in order to assess how a research group using our platform can get benefited in terms of time to achieve specific tasks involving interaction with remote peers versus the use of their current lab tools setup.

How does our platform scale with large groups of concurrent remote users? How does the location of users affect the collaborative interactions? Such a study could investigate the threshold value for simultaneous participants in a collaborative session. The study could further investigate best technical approaches and strategies to avoid degrading the collaboration service as the number of participants grow to a certain point.

Can LivelyViz visualizations and web services be consumed by and interoperate with non-Lively based applications? We know that our platform can connect to external datasources, as demonstrated with the VOCs example. However, there must be valid scenarios where external non-web applications might need to consume LivelyViz services. For example, imagine a particular desktop application that would require connecting to the same collaborative
session where researchers analyze a dataset, and are able to interact with them. It could be even a script or batch process sending data in real time to all connected participants to visualize that data.

Aside from the possible future work related with the arising new research questions, there are other areas of improvement specifically related to our bioinformatics dashboard that can be addressed.

Certainly, more visualization components could and should be incorporated into our platform, either built from scratch or imported from external libraries, developed by a third party. The VBRC has shown interest in RNA-Sequencing analysis with the aid of visual tools similar to MochiView [76]. Some of the visual libraries for bioinformatics discussed in Section 3.2 could be used as a base to develop more collaborative components to make our platform more versatile.

In addition, further development for refining the collaborative interactions in the existent components could be done. More collaborative actions should be added to the current visualization components to make them respond to more remote interactions. For example, the amino acid visualizer based on Protael could be modified to support the creation of tracks, features, annotations and markers along the entire sequence. This new functionalities of course should be totally collaborative providing other participants with awareness of concurrent editing.

Regarding the group dynamics described in Section 4.4.4, allowing the creation of subgroups inside a collaborative group should be added. This is already supported at the level of the network protocol proposed by Hemmings [75], so adding this kind of functionality at the component level might benefit the division and allocation of work when the number of users is big and they have to deal with large scale genomes. A possible use of this feature could be a genome annotation task where subgroups can be assigned to edit only a limited region of the entire genome but allowing all people in the group see the progress of the annotation along the entire genome. Another possible scenario where this functionality could be useful are the events called annotation Jamborees [103]. In such events, a community of diverse experts gather during a few days to work in collaboration to produce, update and refine the available knowledge of a target organism. Researchers are usually divided in groups according to their area of expertise to work on a specific task to help to construct the consensus information about the target organism.

The simple annotation system could be extended to support threaded comments, in order to track easily the evolution of the discussions. There should be also the
possibility at the end of an analysis session that every participant could provide a final comment, observation or hypothesis. These hypotheses should incorporate a vote or grade system that would allow every participant to cast a vote in order to come to a consensus after the analysis has finished. Finally, the formula editor could be extended to allow retrieval and saving of user-defined formulas from the server. In this way, users can avoid recreating complex formulas every time they want to start an analysis session with the scatterplot component.
Bibliography


[94] Georgios A Pavlopoulos, Dimitris Malliarakis, Nikolas Papanikolaou, Theodosis Theodosiou, Anton J Enright, and Ioannis Iliopoulos. Visualizing genome and


