8-2019

Rethinking the Delivery Architecture of Data-Intensive Visualization

Mohammad Raji
University of Tennessee

To the Graduate Council:

I am submitting herewith a dissertation written by Mohammad Raji entitled "Rethinking the Delivery Architecture of Data-Intensive Visualization." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

Jian Huang, Major Professor

We have read this dissertation and recommend its acceptance:

Jian Huang, Mark Dean, Audris Mockus, Russell Zaretzki

Accepted for the Council:

Dixie L. Thompson

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

Rethinking the Delivery Architecture of Data-Intensive Visualization

A Dissertation Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

MohammadReza AhmadzadehRaji

August 2019

Dedicated to Sara, the wind beneath my wings

Acknowledgments

First and foremost, I would like to thank my advisor, Dr. Jian Huang. Over the years, Dr. Huang has helped me see beyond what I could, has guided me with patience, and has taught me the tenacity to push through. His lessons on having a "heart of a champion" have been a guiding principle at every turn.

Second, I would like to thank my wife, Sara, as well as my parents. They have been inspirational and supportive through thick and thin. I am particularly grateful for my wife's support. Sara has believed in me when I could not, and has lifted me every time I would fall.

I would also like to thank my friends at Seelab. They were the best teammates one could ask for. In particular, I would like to thank my best friend Alok Hota, whom I am honored to have worked with. Alok's perseverance and perfectionism have been an inspiration for me and our collaborations have been instrumental in this work.

Finally, I would like to thank the members of my committee, Dr. Mark Dean, Dr. Audris Mockus, and Dr. Russell Zaretzki for their invaluable feedback, insights, and ideas regarding my work. I would also like to thank the staff of the Department of Electrical Engineering and Computer Science for their help and assistance through the years.

Abstract

The web has transformed the way people create and consume information. However, data-intensive visualization applications have rarely been able to take full benefits of the web ecosystem so far. Analysis and visualization have remained close to large datasets on large servers and desktops, because of the vast resources that such applications require. This hampers the accessibility and on-demand availability of data-intensive science. In this work, I introduce a novel architecture for the delivery of interactive, data-intensive visualization to the web ecosystem. The presented architecture, codenamed Fabric, follows the idea of keeping the server-side oblivious of application logic as a set of scalable microservices that 1) manage data and 2) compute data products. Disconnected from application logic, the services allow interactive and data-intensive visualization be simultaneously accessible to many users. Meanwhile, the client-side of this architecture perceives visualization applications as an interaction-in image-out black box, with the sole responsibility of keeping track of application state and mapping interactions into well-defined and structured visualization requests. Fabric essentially provides a separation of concern that decouples the otherwise tightly coupled clients and servers seen in traditional data applications. Results show that Fabric enables high scalability of audience, supports large data while maintaining interaction, supports scientific reproducibility, and improves control and protection of data products.

Table of Contents

List of Tables

List of Figures

List of Listings

Chapter 1

Introduction

The web has been revolutionary. It has transformed the way people create and consume information for over 20 years. With recent advancements in web technologies, cloud-based systems and on-demand services, many applications and products have moved to the web to leverage its numerous advantages such as high accessibility, availability, and sharability to name a few. These applications often share several themes. They typically serve a large number of simultaneous users, provide interactive experiences, and are highly accessible through various devices.

Along with it, the web has also brought the era of big data. An era that necessitates the development of new methods and techniques for data analysis and visualization. However, data-intensive visualization applications have faced many challenges in embracing the web ecosystem itself. They have remained on servers and desktops closer to the core of analysis, their datasets.

The primary reason for the lethargic development of web-based data-intensive applications has been the tight coupling in application architectures involving large data [73]. Applications containing large datasets have often had a monolithic architecture in which the entire state of the application has been bound to application logic, the data, and user actions. In traditional data-intensive applications, the client requires full control over a spawned server. In other words, a one-to-one connection is established for each user, limiting the scalability of resources.

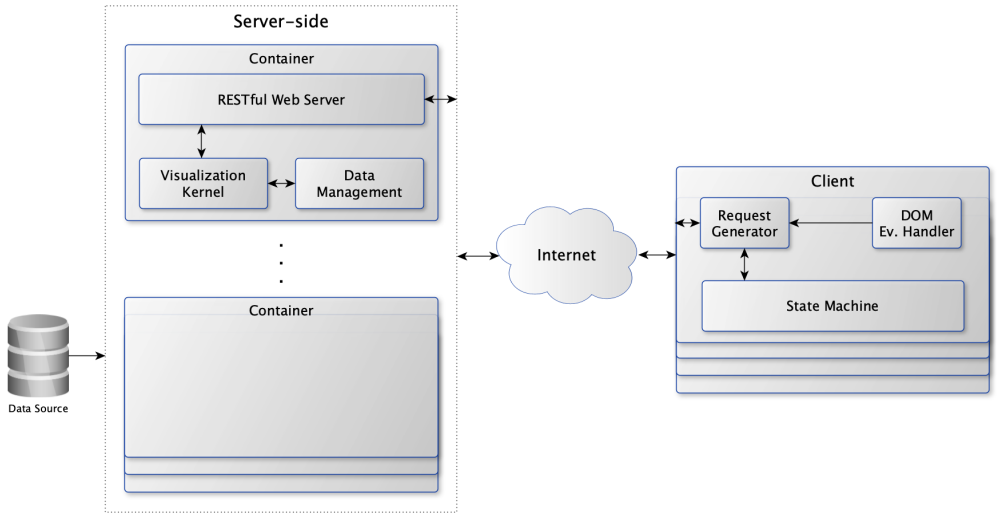

Figure 1.1: An overview of the Fabric architecture is shown. The server-side is composed of a swarm of containers that receive requests and compute visual data products, oblivious to application logic. The client-side has the responsibility of converting user interactions to visualization requests as well as managing application state.

In this dissertation, I introduce a new perspective on the delivery of interactive, data-intensive visualization to the web. The overarching method, named Fabric and shown in Figure 1.1, follows the idea of keeping the server-side oblivious of application logic as a set of scalable microservices that 1) manage data and 2) compute data products. Disconnected from application logic, the microservices allow interactive data-intensive visualization be simultaneously accessible to many users and allows horizontal scaling on the cloud.

The client-side in Fabric is untangled from computing data products and therefore perceives visualization applications as an *interaction-in image-out* black box. It has the sole responsibility of keeping track of application state and mapping user interactions into well-defined and structured visualization requests. The conversion of these visualization requests to visuals is only the responsibility of the server-side data-compute microservices. The model is essentially a separation of concern that decouples the otherwise tightly coupled client and server seen in traditional applications.

Fabric's server-side is realized based on the idea of software containerization and is built as a swarm of containers (Figure 1.1). Each container serves incoming requests using a web-server and computes data products using a stateless and independent program that we call a *visualization kernel*. Different visualization kernels are used for different application needs.

In this dissertation, I first describe a Fabric application that incorporates a simple visualization kernel that serves pre-rendered image responses to clients. I then introduce two other Fabric applications with kernels that perform live computation with high fidelity. We call these two types of visualization kernels *passive*, and *active* due to the time in which they compute data products.

A passive visualization kernel follows the idea that by limiting incoming requests and responses between the client and server to a finite number, a visualization request can be simplified as a call for a particular pre-rendered image, previously captured. With this restriction, one can consider a visualization application as a finite set of application states and their corresponding visual outputs. In other words, the application can be modeled as a deterministic state machine. Through this approach, my work allows interactive visualization applications be captured as independent standalone objects that are completely disconnected from the original data while still maintaining interactivity through a large yet finite and compressed set of pre-rendered images. One of the main usecases of this is the archival of interactive visualization.

While a pre-rendered set of responses can have many usecases, it limits the fidelity of the visualization. Active kernels alleviate this by rendering responses on the fly, at the cost of computational resources and more complicated client-side state machines.

At a high level, two types of visualization exist in the community: scientific visualization, and information visualization. My approach considers the needs of both of these sub-communities and supports three properties to provide a rich experience on the web when it comes to data-intensive visualization:

First is the size of data. Fabric supports the delivery of large datasets both in the information visualization and scientific visualization domains.

Second is the size of audience also known as the number of users. One of the main benefits of the web is that it provides a platform for multiple users to share and collaborate. Fabric takes this into account and offers support for high accessibility and sharability for multiple users.

Third is the degree of interaction. Interaction has always been one of the key components of visualization. In the past, large data visualizations have been delivered through pictures

and videos only, or have only supported single users. Fabric enables high interactivity in both cases where the original application and data are present and in the case where they are not.

This dissertation is organized as follows. Chapter 2 discusses the relevant background work in the literature. Chapter 3 gives an overview of a Fabric-based architecture with support for various visualization kernels. Chapter 4 presents an implementation of Fabric with a passive pre-rendering kernel. Chapter 5, and Chapter 6 describe two realizations of Fabric using active kernels for scientific and graph rendering respectively. Finally, Chapter 7 concludes the dissertation, and discusses future works.

Chapter 2

Background

Fabric aims at the delivery of an interactive experience with large datasets both in the information visualization and scientific visualization domains. This challenge vertically touches many aspects in computer science. At a low level, it involves building fast and reliable visualization kernels that can render visual data products in real-time. At an architecture level, the kernels are scattered throughout a swarm of containers among many machines. On the client-side, higher level aspects such as interactivity, and shareability are considered. This chapter discusses the background work in these areas with regard Fabric's approach. It also describes the background work of the three Fabric-based applications in this work, namely, Loom (Chapter 4), Tapestry (Chapter 5), and KnitGraph (Chapter 6).

2.1 Web-based Visualization

Over the years, visualization researchers have made much effort to make interactive visualizations work inside web browsers.

At an architectural level, visualization applications on the web can be divided into two categories based on where the data source is placed. One category is client-side applications that require the data be present in the web browser or streamed to the browser while users interact with the system. In these applications, the client-side is responsible for performing the rendering and visualizing the results. The second category is remote-visualization applications that perform visualization on the server-side and communicate

the visual product to the browser for viewing. These systems require dedicated servers other than typical web servers on the Internet.

Within the client-side category, the creation of D3.js in 2011 became one of the most recognized milestones [22]. D3.js provides a simple interface for mapping data to visuals. One of the reasons for its popularity is the plethora of examples that can be found online [30]. As a competing project, Vega has provided a reactive visualization grammar for the web [83, 82]. These works are typically based on web-based elements in the Document Object Model (DOM) and are limited to the number of DOM elements that can be supported and rendered in browsers.

Three dimensional graphics libraries and tools such as WebGL and ThreeJS [24] have also been used for visualization on the web [57]. While applications that use WebGL or the HTML5 canvas with D3.js alleviate the DOM elements limitation, they still require the data be present on the client-side and perform visualization tasks natively inside web browsers. As a result, they are more suitable for small datasets and rely on the user's machine for rendering performance.

In the remote-visualization category, dedicated scalable systems perform data processing and rendering on a remote server and transmit final or intermediate data to the client for further handling. While, some of these methods transmit intermediate data products, it is more common to send fully rendered images in general [32]. These solutions fit better with the high-end computing community [54, 53, 70, 97, 98], where server-side computing and networking resources tend to be abundant. However, these methods typically create a one-to-one connection between the client and the server-side and require large dedicated servers or clusters for a user's task. This limits these approaches in horizontal scalability, cloud deployment support, and as a result, the number of users that can be handled simultaneously.

At the extreme end of remote-visualization techniques, some image-space methods pre-render visual products, store them on disk, and communicate the results using light web-servers. As a recent success, ArcticViewer is a web visualization system that improves in-browser user experience by serving pre-rendered images of datasets on demand [4]. Paired with Cinema [12], ArcticViewer addresses needs by the in-situ visualization community particularly well. Pre-rendering typically generates a large amount of rendered images that

may or may not be used by the end-user and limits the number of interactions possible to only those that were pre-rendered.

Other notable successes in the remote-visualization category include: (i) visualization system interfaces, such as Visualizer and LightViz [54]; (ii) API-based scientific data management applications, such as MIDAS using the ParaViewWeb API [53]; (iii) plugin-based web browser systems backed by a high-end resource, such as ViSUS using an IBM BlueGene [70]; and (iv) plugin free implementations backed by custom clusters, such as XML3D [97, 98]. ParaViewWeb and LightViz make use of vtk.js [57], a web port of the popular visualization toolkit.

In this dissertation, I introduce Fabric as an image-space remote-visualization architecture that is stateless and can therefore be horizontally scaled and easily deployed on cloud platforms. Unlike previous works, Fabric is a web service composed of many copies of micro-visualization-kernels that work independently of one another and do not maintain a persistent connection to the client. I first present a flexible pre-rendering-based kernel and then present two visualization kernels for live rendering of large scientific data.

2.1.1 Architectural Designs of Client-Server

Regardless of whether a web browser is used, delivering visualizations with mobility should ideally combine the best of both client-side and remote-visualization designs. For example, it should be "expressive and flexible" like in data-space systems[22], and be "immediately available" for large data like in [12].

This need can benefit from having a more clear separation of concerns, where the expressive and flexible interactions are handled separate from the highly available and efficient computing.

Existing applications often take a monolithic approach (Figure 2.1-left). In result, the 1-to-1 mapping between client and server has become a standard design in existing systems, such as VisIt[29], ParaView[15], and web-based systems like ParaViewWeb. Previous works have also explored adding staging nodes in between the client and the server, in order to achieve better system performances [106, 89], while continuing to abide by the monolithic 1-to-1 mapping between client-server.

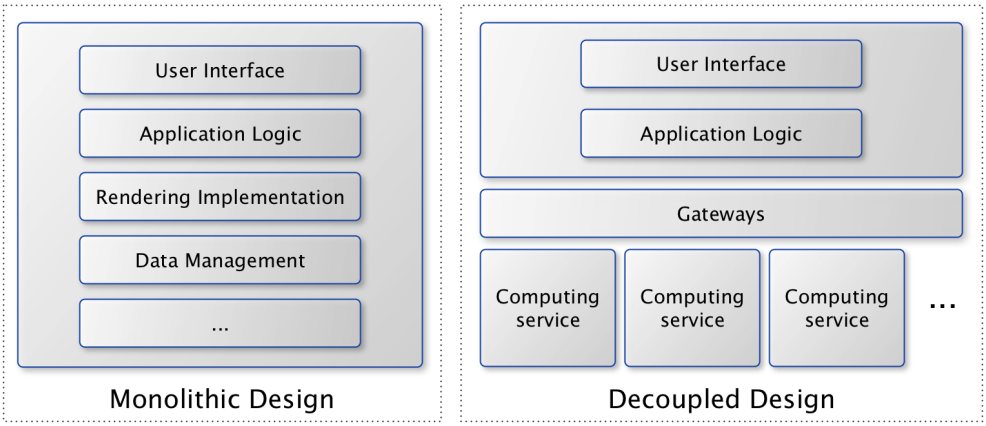

Figure 2.1: Comparing a monolithic design (left) and a decoupled design (right). In a monolithic design, application states exist throughout the component stack. In a decoupled design, compute-intensive tasks such as rendering and data management are encapsulated in stateless services and accessed through a unified cloud-hosted gateway.

A decoupled design (Figure 2.1-right) can separate the highly responsive visual application from the compute-intensive components such as rendering. In that regard, our recent work, Tapestry [74] was the first example to our knowledge that implemented such decoupling and allow the client-server to have an m-to-n relationship, where m and n can be any number above 1. Tapestry can be considered as a particular implementation of a Fabric architecture for scientific volume and iso-surface rendering and is discussed in detail in Chapter 5.

By simplifying the client-server interface into rendering requests, Tapestry's server-side is responsible for rendering and is completely oblivious to anything application related; that is, it has become stateless, free of application logic. The server can answer simultaneous requests from m different clients. On the client-side, a web-page can contain visualizations of many datasets, potentially hosted on n different servers.

While computing services commonly use HPC platforms, the Tapestry server was instead created as a web service managed by Docker [74], to reap the benefits of automatic load balancing, auto-scaling, and automatically parallelized data transfers that are standard in today's web technology.

In [73], we further extended Tapestry so that the server can run as an Amazon AWS deployed microservice, which is independently deployable, fine-grained, and lightweight.

Deployed on AWS, applications can achieve interactive performance at minuscule costs. Hence, scientific visualization can be more accessible and available than before.

We perceive Fabric as an adoptable method that established tools that have played a pioneering role in today's computational science, such as VisIt[29], ParaView[15], and Arctic Viewer [4], can incorporate for resource-scaling purposes. Chapter 3 gives more details into how a Fabric architecture is structured.

2.2 Pre-Rendering and its Applications

Chapter 4 discusses a Fabric implementation called Loom that uses a pre-rendering-based visualization kernel. Loom captures visualization applications and serves them to the web. This section covers the background work on pre-rendering, capturing, and its applications.

2.2.1 Capturing and Modeling Application Behavior

Interaction with applications and managing their state have been modeled with UMLs and other types of finite state machines for many years [80, 49]. UMLs provide a complex state machine that can represent how one can interact with a user interface and how the state of the application changes with actions. However, UMLs can quickly become very large and complex. In recent years, behavior trees have been used to model artificial intelligence in games [59] as simpler and more modular alternatives. Originally, behavior trees were designe d by Dromey et al. as a way to formalize requirements in designing systems [34].

Inspired by behavior trees, Loom uses a tree structure as a simple way to model interactions in a visualization application.

Capturing and storing an entire application along with its data and code has been possible for a long time with the help of virtual containers. Containers can be used to store applications along with their data and requirements [5]. While containers simplify installation and reuse, the size of the resulting container images can grow exponentially. Additionally, any system that is not based on processed products of the application requires the presence of the data source or at least a connection to it. However, this is infeasible for archival in which the data source may no longer be accessible.

2.2.2 Scientific Reproducibility

The topic of reproducibility has become a focal point in the scientific community in recent years. This is evidenced by the forming of a special joint project of the National Academies of Sciences, Engineering, and Medicine (October 2017) [2], publications in PNAS (March 2018) [38], and an entire special issue in Nature (October 2018).

Reproducibility involves techniques that help re-create the results of other scientists. It is a crucial attribute of scientific research that differentiates impactful work from paper-driven work. Reproducible research also increases the speed of scientific endeavor since trying to recreate and continue the works of others can cost a lot of time.

Within the visualization domain, provenance has been a key component of reproducibility. Tools supporting provenance, record and show how a researcher using the visualization arrived at a hypothesis or conclusion and what the process of arriving at the particular settings in the application were.

Tools such as VisTrails [17] and ParaView *lookmarks* [91] have been used for a long time within scientific visualization applications like ParaView itself. More recently SIMProv.js has been introduced as a library that simplifies adding provenance capabilities to web-based visualizations [25].

The Fabric-based Loom application in this work not only considers the provenance of user actions but also introduces methods for reproducing visualization results while disconnected from the original application and data. This feature can simplify sharing and storing hypothesis testings and research results as standalone objects.

2.2.3 Archival and Provenance

Archival of data and data products has been a crucial endeavor in advancing research [72, 100, 78]. As an example, research based on the Internet Archives' Wayback Machine has steadily increased since its birth in the year 2000 [14]. However, as an important insight interface to data, archival of visualization has been seldom studied. This is while software used for visualization can become obsolete as is evident by the discontinuation of Adobe Flash [9], and data ownership policies mean that data sources may not always be available.

Provenance is perhaps the closest area to archival in the literature and has been a key topic in visualization as a way to enable scientific reproducibility. VisTrails [17] systematically captures provenance information in a tree structure and is used in conjunction with other visualization tools to store and recreate workflows. In Paraview [11], Lookmarks have been used to store views of datasets similar to how bookmarks work for webpages [91].

The topic of provenance has also been looked at in the context of web applications. The Open Provenance Vision [64] has been presented as a general vocabulary and format that can be used by different semantic web applications. As follow-up work, the W3C presented the PROV standard as a set of specifications for storing and sharing transformations to data [102]. Many applications have been built on top of such web-based specifications, such as Komadu [95], and Karma [88].

Within web-based visualizations, SIMProv.js has been recently introduced as a general way of augmenting web-based applications to include provenance throughout the user's interactions and reasoning process [26].

Provenance deals with storing the history of user tasks within a visualization application. However, this is different than archival. We define archival as storing a visualization and restoring it at a time when the data or software infrastructure may not fully exist. An archived visualization may still include provenance data regarding how the visualization was utilized before.

One of the most related works to Loom is Graphical Histories [50], in which the states of a visualization application are stored as a hierarchy of images depending on user interactions. Similar works exist that store image transformations in hierarchical structures [47, 28]. Loom takes this idea further in that it also recreates the interactivity of the application at runtime so that the visualization can be used and explored while the data and code no longer need to exist.

2.2.4 Automation in Visualizations

The most common assumption with visualization is that it is visual, and interactive. Hence a Graphical User Interface (GUI) is assumed. When using these GUIs, a common user behavior is to search through a problem space in search of significant patterns.

As common in computer science, the search can be automated even for visualization. For example, automated compound boolean query based visualizations [93], automated regular expression based queries [46], even one of the most difficult tasks in visualization, the design of effective transfer functions [111, 67, 76]. These automations employ simple, powerful visualization specific languages, and as a result, have led to many visualization researchers considering the specification of visualization as textual.

In the above cases, researchers used special program-accessible interfaces to control the visualizations and found great successes. In a related way, other researchers have also built and used UI-bots to automatically go through a graphical user interface. One of the most recent works is the use of monkey testing to automatically stress-test web based visualizations [75].

In Loom, we have also developed a UI-bot and ways for a human expert (e.g. a visualization developer or an archivist) to guide the UI-bot to methodically go through the problem space as specified by the graphical user interfaces. Our key focus here is that, in a transparent way, the captured visualizations are organized according to guidance provided by the human expert.

2.3 Visualizing Volumetric Data

Chapter 5 discusses Tapestry, a Fabric-based architecture for scientific rendering of volumetric data. This section looks at the background of volume visualization.

Volume visualization is well understood from an algorithm perspective [61]. Highly efficient implementations using many-core processors, either GPU or CPU, are available as community-maintained open-source renderers [29, 15, 103, 1]. In this work, we use OSPRay [103] because of its rendering performance. Additionally, its software-only nature makes it easier to manage in a typical cloud-managed container. A GPU-based renderer that exhibits similar throughput to OSPRay can also be used.

Level-of-detail is a proven approach to manage the trade-off between speed and quality for time-critical visualization [109, 58, 17]. Tapestry uses a similar approach. When a user interacts with the 3D visualization in the web document, rendering requests are made at a

lower resolution. After a user pauses, rendering requests are made at a higher resolution. This is detailed in Section 5.1.1.

Parallel visualization generally takes three approaches: data-parallel, task-parallel, and a hybrid of the two [44, 108]. Our primary concern is system throughput (i.e. rendering requests/sec). We chose the task-parallel approach to process rendering requests in parallel. As is commonly done [40], we group worker processes into a two-level hierarchy: (i) the computing cluster as a whole, (ii) each computing node. Worker processes on the same node share datasets via memory-mapped regions of disk. Using known methods to resolve I/O bottlenecks [56], we have a dedicated I/O layer as the data manager on each node to manage pre-loading the data once Tapestry starts (detailed in Section 5.1.2).

2.4 Visualizing Large Graphs

Graphs are used in many areas of science to show relationships among entities. With the increase in recent decades, many works in the literature have tackled the challenges of visualizing and interacting with large graphs. Works in the literature can be divided into three main types: *layout calculation*, *clutter reduction*, and *rendering*. Chapter 6 covers a Fabric application for large graph visualization.

Earlier works on graph visualization revolved around optimal layouts for graphs. The Fruchterman-Reingold algorithm is one of the most famous force-directed layouts from earlier days [41] with its optimized parallel version having been introduced more recently [42]. Many other graph layout algorithms exist in the literature [18].

While most works outside of the visualization community, pertained to layouts, the visualization community often presented full-stack applications that also tackled the problem of graph rendering. For example Munzner et al. Munzner et al. visualized large graphs as minimum spanning trees and in a hyperbolic space to reduce clutter. Munzner also developed H3Viewer, capable of interacting with 100k node graphs [65].

Layout algorithms are at the heart of graph visualization. In this work, we use the SFDP [52], and ARF [45] layouts due to their inherent parallelism and availability in the *graph-tool* library that provides fast and efficient graph algorithms [71].

More recent works revolve around edge bundling and similar techniques that reduce vertex and edge clutter by altering the shape of edges. One of the first edge bundling algorithms for general undirected graphs was the work of Cui et al. [31]. More recently, multilevel edge bundling can create a layout for a million node graph in a matter of minutes [43].

In the area of graph rendering, applications such as Gephi [16], GraphViz [36], and Cytoscope [85] are commonly used. These applications facilitate the entire process of visualizing and analyzing a graph. However, they are mostly desktop-based and are meant for single-user exploration.

Many graph rendering applications rely on some technique to reduce the number of elements rendered at each point in time. One of the first such systems, ASK-GraphView, used the notion of hierarchies and clustering to limit the number of rendered elements [8]. GraphMaps facilitates the browsing of large graphs in tile-sized portions [66]. Other works used querying [19, 39], and machine learning [27] as ways to limit the size of the rendered data.

While some of these works have been effective in browsing graphs of over a million node in size, they have often visualized limited local views of graphs and expected prior knowledge about the graph for querying. In contrast, the Fabric-based architecture for graph visualization in this work is capable of rendering the entire global view of large graphs on the web, and provides live interactive tools for filtering and exploration.

A common critique of rendering large graphs in their entirety is how most layout algorithms tend to show large graphs as a giant hairball [39]. However, I believe that with interactive zooming and fast rendering, a global rendering can better communicate the overall structure while still allowing for the local dive-in option. Additionally, with live filtering tools, a global view can enhance exploration. In Chapter 6, we look at the effects of filtering at a large scale.

Chapter 3

Architecture Overview

The difference between a Fabric application and traditional visualization applications can be seen in Figure 2.1.

The Fabric architecture revolves around the idea of managing application state on the client and serving visualizations from a swarm of containerized stateless servers. Application state is the collective variables and values that define the state of the application and change often depending on user interaction.

In a visualization application architecture, two contrasting variables affect the location of application state. On one hand, interaction is what changes it, while on the other hand, how the system processes and renders the data depends on it. From a programming perspective, these two components, interaction and computation, pull application state towards themselves. In large-data systems, the data is often on the server-side due to its size and interaction is first handled on the client-side where the user is. As a result of this, traditional visualization applications manage state in both the client and server sides. This results in a tight coupling between the client and server and limits their scalability.

Tight coupling is a measure in software engineering that developers strive to reduce between the components of their system [51]. Reduced coupling increases the scalability of components and simplifies programming and debugging.

In the following, we look at the main components of Fabric.

3.1 Containerized Services

The overall idea of the stateless server model is that freeing the server-side from application state allows it to be easily replicated and scaled on multiple machines. This modularity increases performance by allowing the server-side to respond to various users simultaneously, instead of being bound to a one-to-one connection and responding to a single user.

To facilitate this distribution and the management of resources, we use containers to wrap the servers. Specifically, we use Docker containers in a Docker swarm [5]. We have experimented with this setup in our Tapestry project [74] and seen great performance.

Figure 3.1 shows a simplified overview of each visualization container. Every container includes a web server that handles incoming requests. Requests are parsed and sent to a visualization kernel that produces an output by visualization the data through the data management component. The output may take the form of an image or other format. The idea is that the output should need almost zero further processing for the client. A raster image, for example, would need no processing and can be handled natively in a browser.

A visualization kernel can either be *passive* or *active*. A passive kernel is a program that only serves pre-determined and pre-computed results such as pre-rendered images. An active kernel produces results at runtime, based on client requests.

At a higher level, a set of virtual containers reside on a physical node within the cloud. In this setup, the datasets involved can be replicated on all nodes if the nodes have sufficient memory. Data files can be memory mapped so that they only load in memory once per node. In cases where the datasets are too large to fit in memory, they can be distributed across the swarm. The visualization kernels in this case would have the responsibility of providing parallel computation and rendering.

In Fabric, the server-side is only responsible for repetitive computation (e.g. rendering and encoding). The client-side handles interaction and is in charge of application logic. When application state changes as a result of user interaction, the new state is temporarily sent to an instance of the server-side (i.e. a container) to create a new output (i.e. rendering) and return the result. After the server-side container has finished its task, it forgets the application state and becomes available for another task.

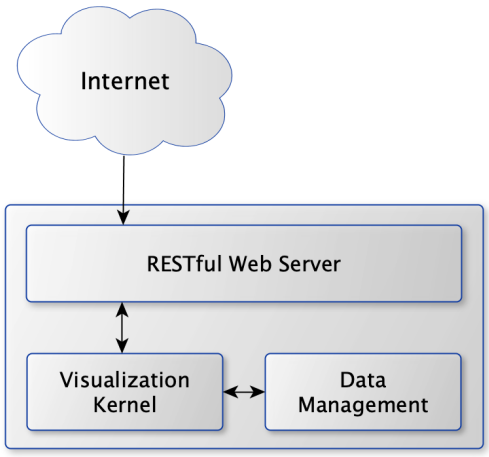

Figure 3.1: An overview of a visualization container. The web server within the container responds to request from the outside world. The kernel accesses the data manager and produces visual responses based on the incoming request from the web server.

Using Docker Swarm, the client-side only sees a single unified HTTP endpoint that it can communicate with. Requests to the endpoint are load-balanced and routed to available Docker containers. The response from the containers is also handled by the router and sent back to the correct client.

Containerization using Docker provides easy scaling and load balancing. Additionally, it provides fault tolerance. If a container fails due to software error, other containers take its place to respond to visualization requests.

3.2 Application Logic

The client-side in a Fabric implementation is responsible for handling user interaction. For example, as shown in Figure 3.2, in a browser-based application the client-side listens to changes in the Document Object Model (DOM) and invokes callback functions as a result of them. In Fabric, the interactions are mapped to visualization requests. While seemingly trivial, the mapping depends on the application at hand and the application state. For example, in Tapestry, the main supported interaction is 3D rotation through an Arcball algorithm and a new request is generated given the user interaction and the previous state of the application (a virtual camera's position in 3D space). Arcball is a known algorithm

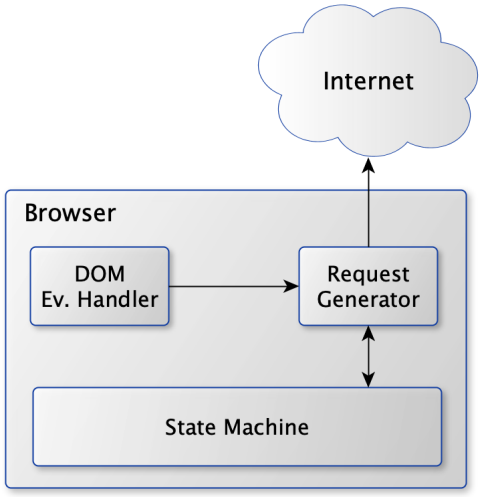

Figure 3.2: An overview of a the client-side of our architecture is shown. Even handlers handle interactions with the Document Object Model (DOM). They then invoke the request generator that creates a new visualization request by means of the application's state machine. The request is then sent to the server-side through the Internet.

in computer graphics that maps 2D mouse interactions to 3D rotations using quaternion calculations.

The generated request is then sent to the server-side as an HTTP request.

3.3 Features and Use Cases

3.3.1 Scalability of Audience

The main benefit of Fabric, is that it provides a basis for a system that can deliver large data visualization, as well as simultaneously supporting many clients.

While scalability in computation has been a theme in the HPC-Visualization community for a long time, Fabric takes this one step further and increases the scalability of audience so that large data visualization can be available on the web ecosystem.

It is worth noting that the audience is not limited to those on the Internet using a browser. The support of multiple clients and web tools allows a Fabric application to be used in other settings as well. Collaborative environments with heterogeneous devices, and augmented/virtual reality settings are two such cases.

3.3.2 Access Policies

Being able to set access policies in a visualization application is another advantage of a Fabric architecture. Because the server is the one providing visual output, it can easily limit what different users can request, with user-defined access policies. Consider an example of a Tableau visualization. The server can simply not respond to requests for information on some parts of the application. Authorized users however, can send hash tokens for authorization along with their requests and gain the complete experience. As another example, entire sub-menus of applications can be enabled/disabled using predefined access-policies.

Access policies can take a form of server-side configuration files mapping hashed user tokens to a list of application states that they have authority to interact with.

3.3.3 Reproducibility

Creating a reduced stateful interface between the client and servers through HTTP requests makes it easy to record all interactions and their responses by listening and saving the requests. Given an initial application state, a pre-recorded set of requests can reproduce the exact steps another scientist took while using a visualization tool. Unlike a video however, users are not limited to *just* those steps and have the freedom to stop and interact with the tool in other ways.

Additionally, given the ability to capture different states of an application, scientists can record their visual analysis as a set of finite states for others to use. Users can then not only see the results that the scientists reached or saw in the visualizations but also interact in the same way with the visuals. The paths of interaction as stored in the interaction tree can be searched and annotated for guided interaction.

3.3.4 Archiving & Sharing Applications

The mapping of interaction to request gives structure and organizes the states of a visualization. By discretizing the requests and responses in a visualization to a reduced finite set, one can easily construct a standalone version of an interactive visualization without its data and source code.

This provides a method for archiving data insights for the future. The captured application states can be endorsed with metadata allowing future users to learn more about specific visuals and insights. Externalization also allows applications to be easily shared and stored offline while still maintaining their interactivity.

Our first example of a Fabric architecture, called Loom, provides such archival capabilities and is further described in Chapter 4.

3.4 Production Environment

Implementations of Fabric have been deployed on both an institutional cloud and Amazon's AWS cloud.

The institutional cloud includes a total of six machines. Three of those machines contain Intel Xeon E5-2650 v4 CPUs and 128 GB memory each, while the other three contain 2x Intel Xeon E5-2660 v4 CPUs with 256 GB memory each. The total number of cores are 156 and the total memory of the cluster is 1.12 TB.

On AWS, based on our initial findings from Tapestry [73], a series of micro or medium EC2 instances are sufficient for running the system.

The client-side of this work mostly resides in user browsers, however due to the request-based nature of our approach, other tools that omit HTTP requests such as `cURL` should also be supported.

Chapter 4

Delivering General Visualizations for Archival

A simple visualization kernel can be considered as one that has pre-computed all possible request responses as images and simply serves them. This fits in the group of remote-visualization techniques that use pre-rendering as the driving mechanism. The types of interactions in such approaches are very limited.

However, we will show that pre-rendering can be greatly extended to capture entire visualizations. In this chapter, we introduce Loom, a system that incorporates a pre-rendering-based kernel that delivers highly interactive visualization applications to the web.

As opposed to live-rendering kernels that require specifically tailored code, Loom can be used to capture and deliver already existing visualization applications. This way, visualizations that were offline in the past can be experienced on the web and can be more accessible. Due to its pre-rendering nature, Loom can help archive interactive visualization without needing to keep the original data or source code.

For instance, a Loom object can capture portions of a visualization in the desktop-based Tableau application or the Paraview application and provide them in a browser, disconnected from its original data.

Note that while Loom objects can include exploratory visualizations that include many application states, Loom is mainly created for explanatory visualizations [92] in which the interactive workflow is pre-determined and known to the archivist.

Using a Loom object is simple. Our Loom viewer is Javascript-based and works in web browsers natively. To this end, archival visualizations can also be used as interactive web objects, regardless of whether the original visualization software is web-based or not.

Loom models user interactions as a tree structure. Given an interactive visualization tool, an archivist can use Loom's Overlay Application (LOA) to specify different UI areas and indicate to Loom, what interactions those UI areas support (e.g. clicking). Then our system uses OS-level UI automation to capture the different states of the visualizations as images. It then organizes and compresses them into a single object. Loom objects also contain information that associate recorded user interactions with the captured images. Loom objects provide a black-box perspective on visualization – accepting interactions as input and providing images as output.

We consider each view of an interactive visualization application as a frame. When a user changes some UI controls, the visualizations on the screen change to a new view, and hence a new frame. At 628 frames and Retina screen resolution (2560 x 1600), the Loom object of the NYT visualization is only 5.5MB in size. In this work, our maximum averaged storage overhead of Loom objects is below 40KB per interaction frame.

4.1 Architecture Design

Let us start with Loom's system workflow and show an example to capture a Tableau visualization of the superstore dataset [96].

Overall, there are three stages in a Loom object's lifecycle (Figure 4.1): (1) user annotation and specification, (2) construction, (3) interactive use. Details of each stage are in Section 4.1.1, Section 4.1.2, and Section 4.1.3, respectively.

In the first stage, Loom solely focuses on the visualization's interface that include user controls and various visualization displays. Loom needs to work non-invasively, and be able to capture interactive visualizations of an existing application without needing to modify the application, or in other ways hinder how the visualization application functions. To this end, we consider the data layer, transformations, and rendering components of the visualization application as a complete black-box.

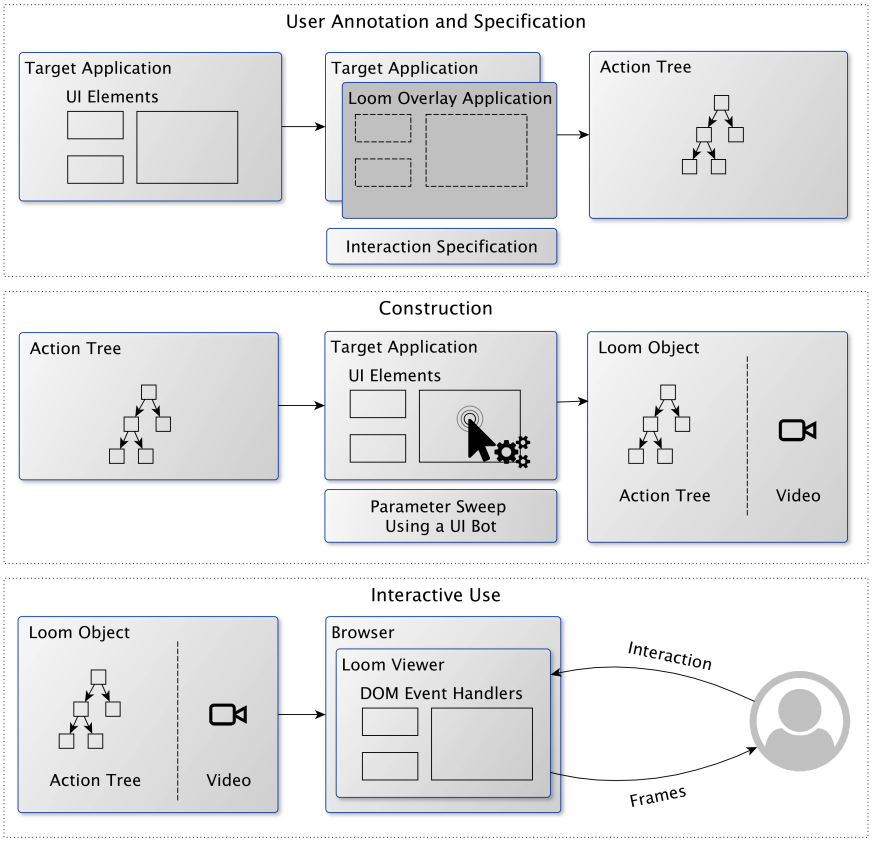

Figure 4.1: The overview of the Loom system. The three stages of a Loom object's lifecycle is shown.

For the first stage, we have designed a desktop application. When opened, the application creates a semi-transparent overlay that can be placed on top of visualizations. We call this the *Loom Overlay Application (LOA)*.

A domain expert uses LOA to annotate the UI controls of the visualization application that they would like to include in the final Loom object. Additionally, they specify the sequence of actions that users are expected to perform.

Using the Tableau visualizations in (Figure 4.2) as an example, these sequences could be as simple as: click on the "Overview" button to launch the overview window, and then mouse over individual states on the map for more information. Another sequence could be to click on the "Profit Ratio by City" button, and then scroll through the bar graph in the new window.

With every interaction with an application, users are presented with different options that can be picked and interacted with. This forms a hierarchy of options. Therefore, Loom

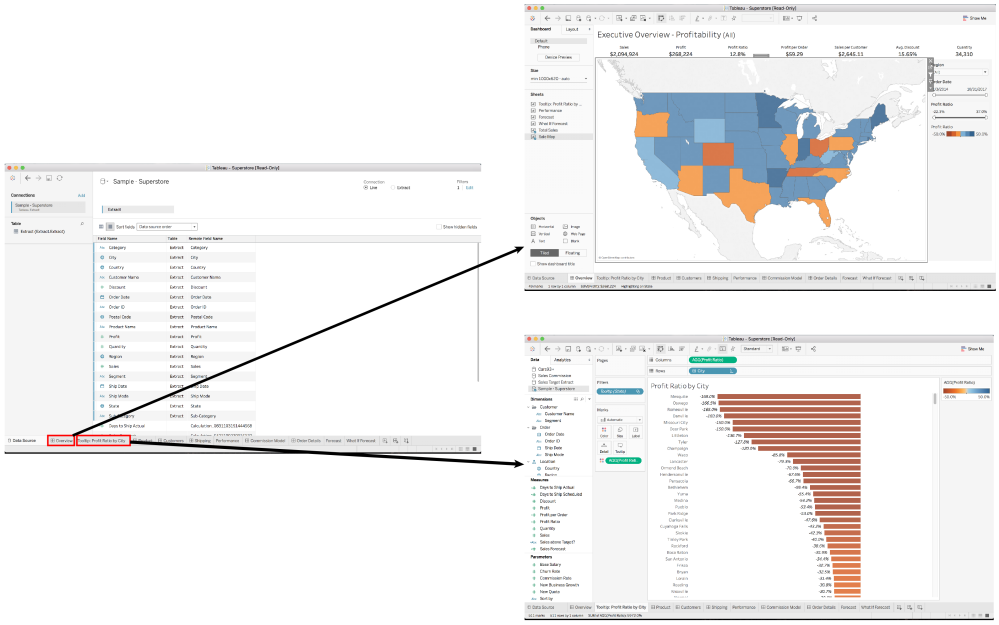

Figure 4.2: The Tableau application (left) showing the sample superstore dataset. Clicking on the first highlighted button at the bottom of the screen (Overview) shows a map view. The next button shows a bar graph of profit ratio by city. It is only after clicking the Overview button, that the map becomes available and the states will become clickable. This is an example case of a hierarchical behavior in a user interface.

stores the sequences of interactions into a tree-based structure. We call this the *action tree*. Earlier actions in the sequences become upper level nodes in the tree, and the latter actions are closer to the leaf level. By specifying particular sequences, the user is implicitly defining an intended user workflow using the visualization application. Hence, LOA outputs are user workflow specific. There is no limit on how many trees can be created and used as specifications. Neither is there a limit on the depth of the action trees. A simplified version of the action tree of the Tableau example is shown in Figure 4.3.

Specifying all actions and their order can be a time-consuming task for users. To alleviate this, LOA provides a suite of smart tools that help users in selecting visual components and specifying their actions. These tools are detailed in Section 4.1.1

In the second stage, Loom objects are constructed based on the specifications output by LOA. We built a UI-bot that automatically triggers UI events by invoking OS level mouse and keyboard control.

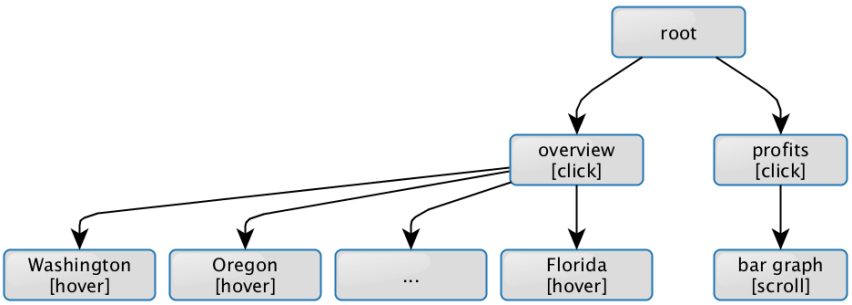

Figure 4.3: An action tree for the Tableau superstore example. Every node of the tree is a state in the application. The action that allows one to go to a state is written in brackets inside each node. For example, one can hover over Washington on the map only after they have clicked on the "Overview" button that reveals the map visualization.

Naively, one can capture all possible visualizations by having the UI-bot exhaustively "click" or interact through all available UI controls, and take screen snapshots of the visualization application along the way.

We optimize that naive approach by having the UI-bot conduct an orchestrated parameter sweep according to the action trees specified in LOA. This gives us two benefits. First, the user workflow specified by the domain expert using LOA provides an "access policy" of the visualization application. Whoever uses the Loom object is limited by that policy. This provides a new way to define, customize, and enforce fine-grained information access privileges for the first time. Second, we have discovered through experiments that such orchestrated parameter sweep as guided by the tree structure produces frames that have the maximal amount of similarities, and the cross-frame differences are usually just incremental. In result, the captured screen snapshots can be compressed very efficiently using widely available video compression technology.

In the third stage, to use the object, our browser-based code opens the Loom object and reconstructs the visualization application. Based on the user's interactions in the browser, appropriate images of the visualization application are then loaded and shown. To the end-user, the experience is as if they are interacting with the original visualization and data.

4.1.1 User Annotation and Specification

To capture a user's possible interactions with a visualization, Loom requires the specification of every interactive element in the application that they want to include. We call these elements "targets". Loom also requires the specification of several properties for every target. Specifying this information is done by the user and the assistance of LOA's toolset. The required properties for each target are as follows.

- **Action** represents the type of action that can be performed on an interactive element. Examples are "click", "brushing", "3D rotation", etc.
- **Position** represents the physical position of the interactive element on the screen
- **Area** represents the area of the region that the interactive element spans on the screen
- **Parent** represents the element that must be interacted with before the current element becomes accessible. An example of a parent target is a dropdown menu for the buttons inside the dropdown. Hierarchically, the dropdown itself must be interacted with before its options become available

The specification phase provides the user with a way to create an action tree based on the application interface. Rather than having the user script the interface manually, such as with a general purpose programming language, Loom shows the user the underlying application and allows the user to define the steps visually and intuitively. This is done through Loom's Overlay Application (LOA).

When the user starts LOA, they are presented with a semi-transparent overlay like the one shown in Figure 4.4. LOA allows the user to visually see the application and define areas for various targets. These targets have an associated shape defining their position and area, and an *action* property that specifies what event they perform. Additionally, every target has a name, an optional description, and a unique ID.

LOA supports rectangles, circles and complex polygons as target shapes. Created targets can be selected using a selection tool. Every target is accompanied by a setting menu where their properties can be set in. The menu appears in the toolbar whenever a target is selected.

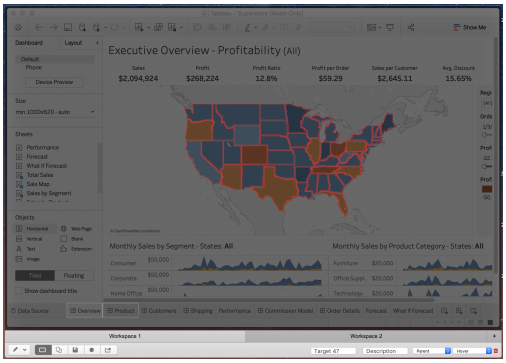

Figure 4.4: Loom provides an overlay application (LOA) that allows users to select different parts of their visualization and tell Loom how it should interact with each component. These selections are then used by Loom to construct an action tree that represents the possible interactions with the application. In this figure, the states of the US are target components selected in red.

If multiple targets are selected, all of their properties can be edited at the same time through the same menu. In other words, the changed property (e.g. the targets' parent) changes to the same value at the same time. This simplifies editing a large number of targets. Loom's toolbar is shown in Figure 4.5 along with a property menu for a selected target.

For each target specified by the user, Loom adds a node to an action tree. When the user is done with adding targets, they can "export" the Loom object, at which point Loom saves the action tree to disk as a JSON file.

Additionally, the user can add a name and description for each target. The name and description can be used at runtime to search for specific states within a visualization.

Assisted Target Specification

As the user adds more targets, LOA's display can become cluttered, making subsequent selections difficult. To organize these selections, LOA provides *workspace tabs* (shown in Figure 4.5). This way, users can use each workspace for a specific part of an application. To LOA, all workspaces still define the same visualization. This means that parent target relationships can be specified across workspaces.

LOA also assists users in specifying large numbers of UI components with a suite of tools, described in the following.

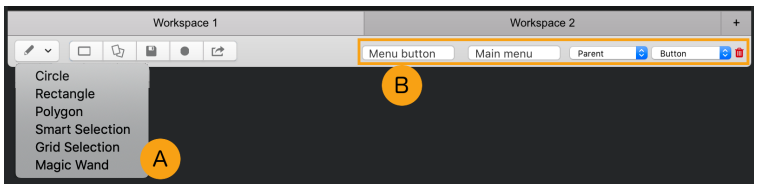

Figure 4.5: Loom's toolbar is shown. (a) shows the selection menu supporting circles, rectangles, and polygons, as well as three assistive selection tools. (b) shows the properties of a selected target. The first textbox contains the name of the target. The second textbox contains the description. Two dropdowns receive the parent state and the action of the target from the user. In the current toolbar, two workspaces are opened by the user.

- **Grid Selection.** In many user interfaces, menu items are horizontally or vertically distributed. The grid selection tool simplifies defining a series of rectangular targets based on a single outer rectangle selected by the user and the number of rectangles to automatically create in the X and Y directions.

- **Magic Wand.** Visualization elements can sometimes have complicated shapes making them difficult to select (e.g. a state in the US map). The magic wand tool uses a flood fill algorithm [99] to select a component, based on its boundaries and color. The states in Figure 4.4 were selected using this tool.

- **Smart Selection.** Visualizations can sometimes have hundreds of visual components. The smart selection tool uses a contour detection algorithm and automatically selects distinct visual elements. The user can then edit or delete the elements as needed. The magic wand and smart selection tools work by taking a screenshot of the underlying visualization, applying their respective image processing method to the image, and adding the selections to LOA.

- **Copy Tool.** UI elements can act differently depending on which target was interacted with before, meaning that the same UI elements can have multiple parents. For example, depending on the value of a radio button, the states on a map might show different information upon hovering. This necessitates adding multiple targets for the same area. The copy tool simplifies this. A user can select a series of targets and by clicking the copy button, new targets of the same shape will be created in a new workspace.

Using LOA on our Tableau example, the user creates a selection box around the two buttons at the bottom of the screen and sets their actions to "click". A default name is automatically associated with the selections. The user also has the option to change those names. In the Tableau application, the user then navigates to the "Overview" map visualization. Using the Loom overlay, they then select the physical position of each of the states on the map and set the action to "hover". To convey the hierarchical aspect of the tree, they additionally set the parent node of each of the state selections to the Overview button. This means that the states are only hoverable if the Overview button had been clicked.

To add the bargraph visualization, the user navigates to the page in Tableau. Using LOA, they then select the portion of the screen that includes the bargraph, and set the action to scroll. By now, the resulting overlay includes all of the selections for our example (Figure 4.4).

By default, Loom supports several action types such as click, hover, brush, 3D rotation, and sliding. Advanced interactions can be added by writing custom actions in small scripts and are discussed in Section 4.1.4.

4.1.2 Construction

During the construction stage, Loom traverses the action tree and interacts with the visualization application on behalf of the user using a UI bot that controls the mouse and keyboard. At every node of the tree, Loom captures the state of the application in the form of a screenshot and saves it to disk. An index number is associated with every screenshot and the number is stored in the visited node in the tree.

Starting from the root of the action tree, every branch down to the leaves is a sequence of possible interactions. For instance, in our Tableau example, a user can select the overview, then click on any of the states on the visualized map. Another path is for the user to select the profits button and view the bar graph. Due to this, the traversal algorithm should interact with only the nodes of a sequence starting from the root and ending at leaves and cannot jump to other branches along the way. As another example, consider an interface that requires the user to click on a dropdown, then click on an option of the dropdown to

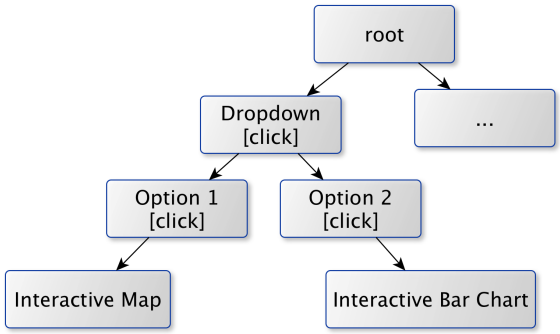

Figure 4.6: A simplified action tree for a dropdown button is shown. Different visualizations are shown based on the two options of the dropdown. After interacting with the interactive map, Loom's UI bot needs to start from the root of the tree again in order to reach the interactive bar chart. This requires restarting a pre-order traversal.

reveal a map visualization. An illustration of its action tree can be seen in Figure 4.6. The left most branch starts from the root, then based on the left most child of the root, Loom chooses to click on the dropdown. The dropdown opens at this point in time. Loom then has the option to go to the left most child of the dropdown and click on the map visualization button. Finally, Loom can click and interact with the map. Note that after interacting with the map, it is impossible to click on a different option of the dropdown simply because the dropdown is no longer open. In other words, the application has lost its previous state. Inevitably, Loom must start from the root again then choose a different branch along the way. This process is repeated until all leaves have been visited. This traversal process is equivalent to a pre-order traversal in which after every set of leaf nodes that share a parent, we go back to the root. The algorithm for this traversal is shown in Algorithm 1.

The `DO_ACTION` function in Algorithm 1 performs the specific action for the target (e.g. move mouse, click, etc.). The `CAPTURE_SCREENSHOT` function takes a screenshot and returns an index representing the screenshot. The number is then set to the frame number for the node and will be used in the interactive use stage.

When the tree is fully traversed, the screenshots taken in this stage are saved on disk and served to the web using a simple web-server.

Alternatively, when the number of frames is low, Loom can save all screenshots in a video format for compression. At this point, the frame numbers in the video correspond to the index that is stored in the nodes of the action tree. In other words, Loom linearizes the

various application states and stores them sequentially in the video. The video can then be sent with the web page, eliminating the need for any server communication. This option is particularly useful for archiving visualizations.

For the video format, we chose MP4 with H.264 compression. Other technologies such as WebM and H.265 could also be used. However, it's worth noting that H.264 is practically the first video compression technology that could enable the development of Loom.

When H.264 became a standard in 2003, besides significant coding efficiency improvements, there were crucial advances in the flexibility of using the technology over a much broader variety of application domains than before[104]. Efficient and high-quality out-of-order playback of video over the web would not be feasible with older technologies. We use the *ffmpeg* implementation of H.264.

Algorithm 1 Traversal of the action tree for automated UI interaction

```
1:  procedure TRAVERSE(tree)
2:      repeat
3:          n ←Find next non-visited leaf node
4:          Find the path from root to n
5:          for every node along the path do
6:              VISIT(node)
7:          for every leaf node that is a sibling of n do
8:              VISIT(node)
9:      until all nodes have been visited
10:
11: procedure VISIT(node)
12:     DO_ACTION(node.action)
13:     if node has not been visited before then
14:         frame_num ←CAPTURE_SCREENSHOT()
15:         node.frame_num ← frame_num
```

4.1.3 Interactive Use

Reconstructing the application as an interactive visualization takes place in a browser. The code for viewing a compressed Loom object is written in Javascript and therefore can be executed using any modern browser on different devices such as desktops, tablets and mobile phones.

In order to reconstruct the application's interface, the Loom viewer requires the action tree as a JSON file. The viewer then makes frame requests to the server based on user interactions.

In the case of the compressed-video option, the video that was created in the second stage is also needed. In which case, the viewer then essentially provides out-of-order playback of the constructed Loom video, based on user interactions.

Initially, when the viewer is opened, it traverses the action tree and creates an invisible HTML DOM object for each of the targets that the user made. These DOM objects will be responsible for handling user interactions. Then, based on the actions of the targets, appropriate event handlers are set up. For example, consider a target with the click action and a respective position P and area A that describe the selected area in the application. To add this target, Loom adds an invisible DOM element with the position and area of P and A. Loom then adds a Javascript "click" handler. The DOM element is then associated with its node in the action tree using a hash table. When the DOM element is clicked on, it finds the connected node in the tree, takes the frame number associated with the node and seeks the Loom video to the correct frame number or requests the frame from the server. Consequently, the image shown to the user is the same as what they would have seen if they had interacted with that button in the original visualization.

Although a DOM element is created for every target, not all targets should be interactable at all times. For example, the options of a dropdown should not be available before the dropdown is opened (clicked on). As another example, the DOM elements created for the states in the Tableau example should only be clickable if the user has previously activated the map visualization. In other words, an application state should only become accessible if its parents have been acted on beforehand. Loom handles this using the concept of state machines.

In the reconstruction stage, the action tree is converted to a state machine. The conversion between the action tree model and the state machine is done using the following rules.

1. An application state is created for every node in the tree

2. A transition is created between every state S and its children in the tree. The condition of the transition is set to the action belonging to the child node

3. A two-way transition is created between every two leaf nodes that have the same parent

Within the state machine, every state can be reached if its parent is the current state and its action is executed. Additionally, the root of the tree can always be accessed if the user clicks on anywhere outside of other targets. This is so that users can continue interacting with the system once they have reached a leaf node.

The number of DOM elements that handle interaction can grow very quickly depending on the complexity of the visualization. Additionally, some of the DOM elements can overlap on the screen in condense visualizations. In this case, DOM elements that are on top prevent bottom elements from receiving user events such as clicking. The state machine solves this issue by raising the DOM elements that should be accessible and lowering those that should not be accessible. This is done through CSS by changing the `z-index` style attribute of the DOM elements.

Guided Interaction and Provenance

A user may choose to not capture the entire visualization application and only specify portions of it. This means that images shown to users may have inactive sections. The Loom viewer provides several tools that hints at what is and what is not interactive. Figure 4.7-left shows the Loom viewer in the browser. The right panel is added by Loom. The panel includes a mini-map showing a gray overview of interactive regions. Additionally, a *hints* toggle button is provided. When hints are turned on, interactive regions are highlighted in the visualization (Figure 4.7-right).

Having captured the states and elements of a visualization creates a unique opportunity for Loom at runtime. The panel includes a search tool with which users can search through names and descriptions of UI elements using fuzzy searching [77]. When a user clicks on a search result, Loom switches to the appropriate application state and draws the user's attention to the element they searched for with a ripple effect. The state machine switches

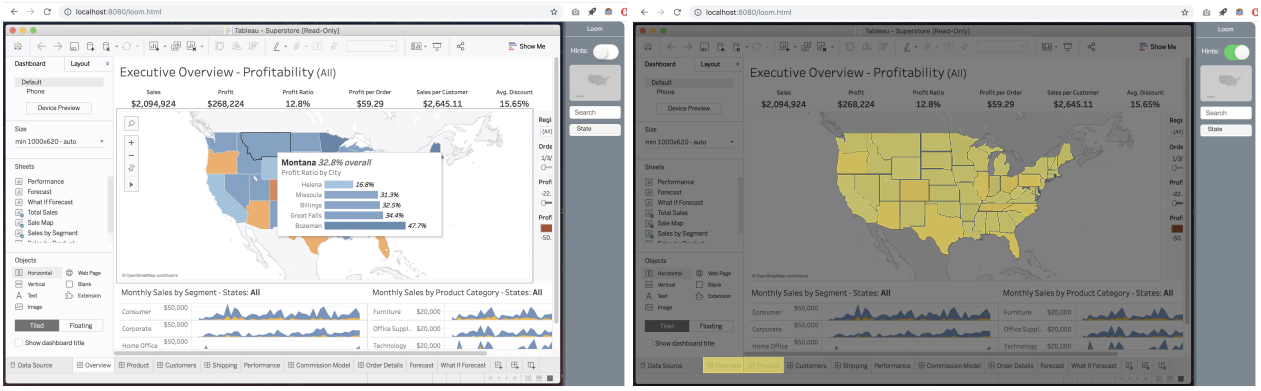

Figure 4.7: Both images show the Loom viewer within the browser. The right toolbar provides a mini-map that shows the position and shape of interactive elements. The right image shows the usage of the hints toggle button, highlighting interactive elements.

to the parent state of the element so that the user can choose to interact with the element or not. An example of this is shown in Figure 4.8.

4.1.4 Extending Interactions

Visualizations include interactions that are much more complicated than simply clicking and hovering over visual elements. Loom's interactions can be extended with custom plugins. A Loom plugin consists of two scripts. The first script tells Loom how to interact with a visual element whose boundary is defined by the user's selection box. In other words, it is essentially a `DO_ACTION` function. The second script tells Loom how to handle interactions in the browser and map them to the appropriate frames in a Loom video. Here, we give an example of how we implemented a plugin for a slider action and a brushing action. Plugins can support even more complex interactions. Section 4.2.2 explains how we added a plugin for 3D rotation around scientific visualizations.

Consider a slider in a visualization with its knob moved to the left. Given the position and area of a user's selection, a simple slider action moves the mouse to the inner side of the selected rectangle. It then performs a click event, holds the slider knob, and moves it incrementally to the right. At every defined interval, it takes a screenshot. In Loom's viewer, a DOM element is automatically created for event handling as mentioned in Section 4.1.3. The plugin for the slider sets a Javascript drag event on the DOM target element. Then,

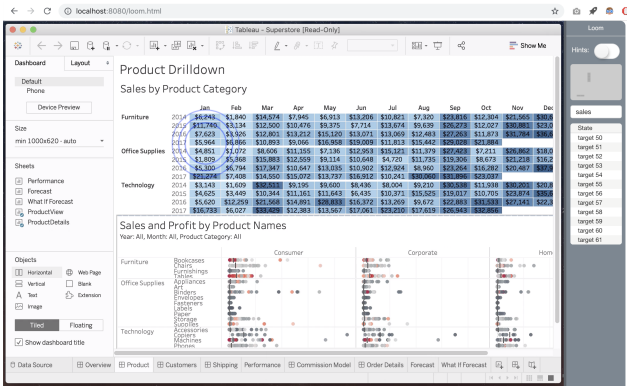

Figure 4.8: Searching through target descriptions populates a list of possible visual elements in the toolbar. Clicking on a target makes Loom navigate to the appropriate application state and highlights the searched target with a ripple effect (in blue)

based on the position of the mouse in the target, it seeks the Loom video to the correct frame.

Brushing is a common interaction in visualization applications. Consider a rectangular brushable area A. Users can pick a location in A and press the mouse button. They can then drag the mouse elsewhere within A and finally release the mouse. They have essentially selected a box S that can be defined with a start and an end position. To support brushing, the brushing plugin discretizes the interaction. In other words, it divides area A horizontally and vertically into a set of cells. It then starts by capturing every possible combination of selections for the divided cells. For n cells ($\sqrt{n}$ columns and $\sqrt{n}$ rows), a total of n^2 selections can be made. For example, an area that is divided into 16 cells (4 columns and 4 rows) leads to 256 different selections. The captured frames are then linearly indexed and added to the Loom object. In Loom's viewer, a DOM element is created for handling the brushing. The handler registers mouse press and releases, calculating the cells encompassed by the user's brushing. Based on the starting cell and ending cell of the selection it re-calculates the linear index of the suitable frame and seeks the Loom video to the frame.

This technique can be used to support many other interactions such as panning, scrolling, dragging, etc. The general mechanism is to capture possible image responses from the visualization and index them linearly, and finding the linear index based on interactions in the viewer to seek to the correct frame. Loom's current code-base includes support for *clicking, hovering, sliding, brushing, and 3D rotation.*

4.1.5 Control of Privacy

Typically, the control of privacy is related to the data. Many visualizations that could be public are not shared, simply because their data sources cannot be shared due to various regulations and policies. Sometimes, we see non-interactive visualizations of a protected dataset, but never see an interactive version online, simply because the website would require direct access to the dataset.

Separating a visualization from its data creates an opportunity to look at the privacy and security aspect of visualizations. An interactive Loom visualization only contains images, making it safer to share. Additionally, it is possible to encrypt certain frames and only enable them for authorized users, providing a finer control on privacy.

In our current prototype, Loom uses AES encryption to encrypt video frames. The archivist can select frames using a query on their target description and choose to encrypt those frames. The frames are extracted from the video into a separate encoded file. The remaining Loom video will no longer support the extracted interactions, unless patched with the decrypted file. Loom uses the OpenSSL implementation of AES, however other types of image encryption techniques can also be used.

While a data-connected visualization can also encrypt its dataset, it will be required to decrypt all or most of it at runtime to create the interactive visualization. However in Loom visualizations, a frame will only need to be decrypted if the user has authorization to view it.

4.2 Applications

4.2.1 Archiving Journalistic Visualization

Unlike many other types of media that can be easily saved as images, videos, or PDFs, interactive visualizations are often difficult to be archived. For example, visualizations used in online data journalism are often bound to a news agency's servers and can be taken down at any point.

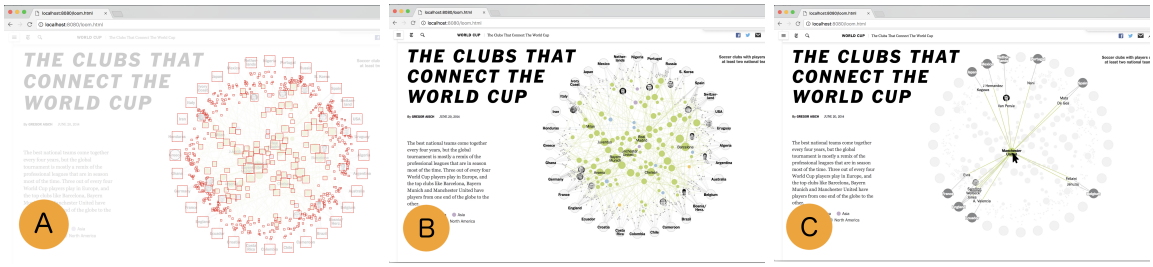

Figure 4.9: A Loom object in the browser showing the 2014 Soccer World Cup visualization from the New York Times. The visualization no longer depends on the original website or the data source, yet is fully interactive. The bounding box of the selected targets are highlighted in red in the top image.

Using Loom with the video option, many types of interactive visualizations can be saved, archived, and then independently used long after the original visualization has become inaccessible. Figure 4.9 (B, C) shows a visualization from the New York Times [68], that has been captured using Loom and re-opened in a Chrome browser.

The complete graph in the visualization has 628 interactive nodes that highlight connected neighbors when hovered on. The size of the resulting Loom object is 5.5MB.

Selecting 628 nodes using LOA can be a tedious task. In this example, we used Loom's magic wand and smart selection to assist in selecting the interactive elements of the page. Figure 4.9 (A) exposes the boundary of the selections.

Although the size of the Loom object is 5.5MB, the full Loom video is not downloaded to the client immediately. Loom utilizes HTML5's video streaming and only loads a small portion of the video when the page loads. As the user interacts with the visualization and as a result, seeks to the various frames of the video, the remaining portions are downloaded. Therefore, unlike the original version of the visualization that requires loading and running many assets and algorithms, the Loom object becomes available almost immediately when the page starts. This serves as a proof of concept for progressively streaming an application.

4.2.2 Capturing Large Scientific Visualizations

To showcase Loom's capability on capturing interactions with scientific visualization, we picked the Paraview application as a subject, and opened a volumetric heptane dataset in

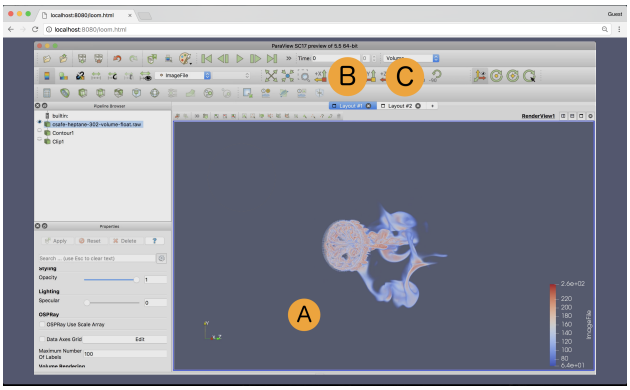

Figure 4.10: A reconstructed version of the Paraview interface in the Chrome browser. In this example, the volume can be freely rotated using Loom's arcball extension (A). The two tabs at the top (B, C) change between volume rendering and surface rendering modes. The Loom object for these interactions is approximately 10MB.

Paraview. The dataset had a size of 105MB. Figure 4.10 shows Paraview with the heptane dataset reconstructed within a browser.

One of the most widely used types of interaction in scientific visualization is 3D rotation around an object. To support this, we added a custom action as an extension. In the capture stage, the action simply uses the mouse to exhaustively drag around an object in Paraview in an organized way. The action first rotates the object so that the camera looks at the zenith. It then rotates the object around the X axis towards the nadir. Loom takes screenshots along the way. Going from zenith to nadir once spans 180 degrees. The action script then re-centers the object, and incrementally rotates the object along the Y axis and continues the first step again. This process continues until the complete object has been captured. Figure 4.10 contains two fully rotational targets in the middle of the screen.

In cases where one has access to the underlying application's API, one can rotate the camera or the object with incremental angles in the custom action script. However, in the case of this example, we aimed for an application-agnostic way of capturing the rotations around an object. Due to this lack of access to the underlying application in this case, we initially measured how many pixels the cursor must travel to complete 180 degrees around the object in our Paraview instance, and then used this to complete the rotations in our custom script. Our action script takes 500 images around an object. That is 20 intervals

around the object, each of which includes 25 images from the zenith to the nadir of the object.

In the reconstruction stage, the extension implements a standard arcball algorithm that maps mouse movements to the 500 captured images based on the yaw and pitch of the arcball algorithm. In our example with Paraview and the heptane dataset, we included two sets of rotations, one for a volume rendering and one for a surface rendering. Figure 4.10 shows the reconstructed Paraview interface within a Chrome browser. What the user sees in the browser is a single frame of the Loom video showing a screenshot of Paraview. Clicking on options B, and C switch between volume rendering and surface rendering. For each option, the rendering in the middle of the screen updates appropriately and can be rotated. As a user drags the mouse cursor on the rendering, their mouse movement is converted to angles using the arcball algorithm. The angles are then mapped to the appropriate image among the 500 captured images from the object, and the image is shown to the user. It is important to note that this is a quantization over the possible rotations around the data and is less smooth than the original experience. However, it stands as an example of complex interaction reconstructed in the browser using Loom, independent of the original data and application.

4.2.3 Capturing Information Visualizations

Throughout the years, many visualizations have been created using Adobe Flash. The discontinuation of Flash and its lost support in modern browsers, makes them a great candidate for archival. Figure 4.11 shows LOA on top of an Flash-based information visualization from the Senseable City Lab at MIT [6].

The visualization shows medical record data. The circular keywords in the visualization have been captured with the assistance of the smart selector. The complete Loom object includes 336 targets created in 3 workspaces. The size of the Loom object is 4.2MB.

As another example for information visualization, Figure 4.12 shows a more complicated version of the Tableau superstore example in Section 4.1.

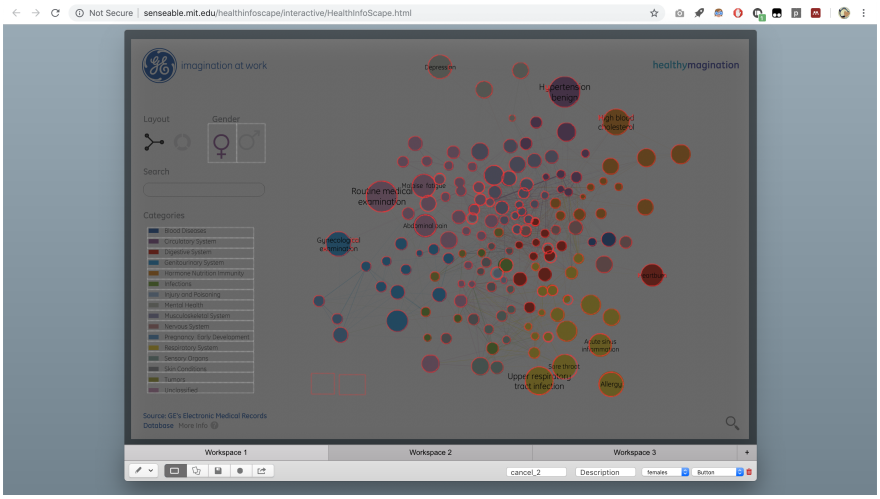

Figure 4.11: The Loom Overlay Application is shown on top of an Adobe Flash visualization. The Loom object includes 336 selected targets and has a size of 4.2MB.

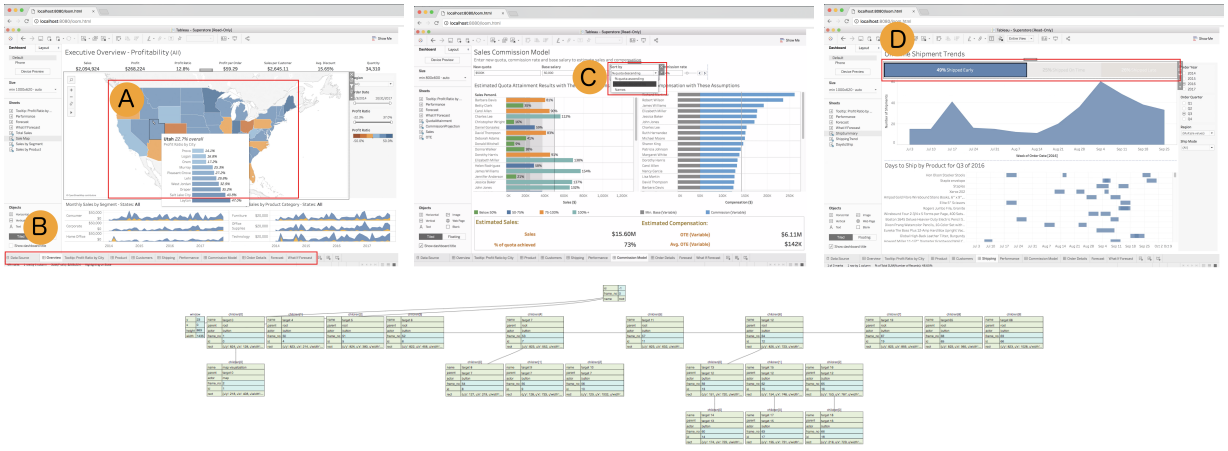

Figure 4.12: A Loom object of the Tableau superstore dataset is shown in the browser. An interactive map and clickable tabs are shown in (A). A functional dropdown that changes the order of the data is shown in (C). Three different line graphs can be picked in (D). The action tree of the Loom object can be seen at the bottom. All of the interactions resulted in a 2.6MB file with 72 frames at a resolution of 2560x1600.

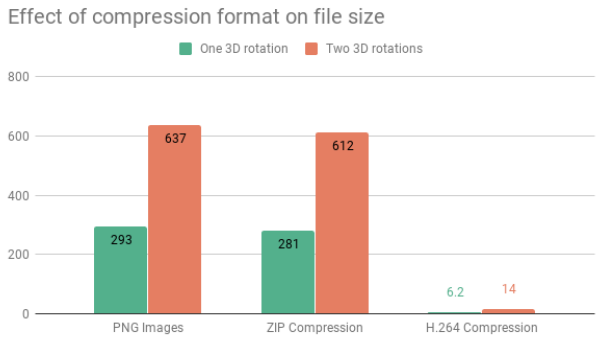

Figure 4.13: The effect of H.264 compression on two Loom objects is shown. The size of the object is at a minimum, 38 times smaller than the raw images. This is mainly due to the similarity between consecutive frames.

The final Loom object included 72 frames with an interactive US map visualization, 10 tabs with various static visualizations, and a dropdown with 3 buttons. The resulting object had a size of 2.6MB at a resolution of 2560x1600.

4.3 Results and Discussions

In contrast to typical videos, subsequent frames in the videos of Loom objects are extremely similar to one another. This results in great compression of the frames and small Loom objects. Figure 4.13 compares the compressed and uncompressed versions for two Loom objects that included volumetric visualizations in Paraview. We can see that Loom objects are 38 times smaller when compressed, compared to the raw images.

The size of Loom objects change not based on the size of the visualized data, but by the amount of interaction that a user needs. This provides an alternative control on the size of visualizations.

Based on the definitions in [21], an interaction is an action by a user with an intent to change the state of an application. Every frame in a Loom video is a new visual response to a state change. Therefore, to quantifiably measure the amount of interaction Loom provides, we consider each frame an interaction.

Although increasing the number of interactions also increases the size of Loom objects, it can also help with video compression. In Table 4.1, the Loom video size and the number

Table 4.1: Results of comparing a Loom object's video size and the number of interactions it provides (measured as new frames). The similarity between frames in cases where there is more interaction has contributed to how well the video compresses. In all cases, the cost of adding an interaction to the object was less than 40KB.

Test Case	Loom Video Size (MB)	Number of Interactions	KB/Interaction
New York Times	5.5	628	8.96
Tableau Superstore	2.6	72	36.97
Senseable City Lab (Adobe Flash)	4.2	336	12.5
Paraview (two 3D rotations)	9.9	1003	10.10
Paraview (two 3D rotations + clipping)	14	1015	14.12

of interactions for five different test cases is shown. Additionally, the KB/Interaction ratio shows how much an interaction is taking space in the Loom object. The Tableau examples have much less interaction and subsequently less number of frames. However, their KB/Interaction ratio is much larger than the Paraview examples that compress better. This is simply because the frames that involve 3D rotation are not drastically different from one another.

In the Table, we can also see the effect of adding new types of interactions. The last row shows an example that includes a clipping interaction that clips a surface rendering. The added detail of the surface renders has affected the KB/Interaction ratio.

Despite these relationships, it is important to note that in all of our tests, the ratio was always below 40KB per interaction.

Loom's UI-bot periodically waits after every interaction in order to let the underlying visualization update if it needs to. It also waits after every mouse movement to prevent incorrect mouse clicks. Therefore, generating Loom objects takes time depending on the number of interactions. The most time consuming case in our examples was for the Paraview object with two full 3D rotations and a surface clipping slider action. Loom's UI-bot took approximately 40 minutes to capture the interactions. That is less than 2.5 seconds wait time between every two interactions. The duration of the waitings can be changed in Loom.

4.3.1 Limitations

Making a visualization independent of the original code and data via pre-rendering can induce some limitations on the types of interactions possible. While many application states

Table 4.2: Loom's support for different types of interactions based on the taxonomy of Brehmer et al. [23] is shown. In general, discrete interactions can be captured, while continuous and undetermined interactions cannot be captured by Loom.

Taxonomical Unit	Support	Comments
Select	✓	Discrete Selections Supported
Navigate	✓	Navigations such as panning are supported if discrete
Arrange	✓	-
Change	✓	-
Filter	✗	Filtering is usually undetermined (based on user input)
Aggregate	✗	Aggregation typically exponentially increases the state space

can be captured, there can only be a finite amount, making it impossible to completely replace Turing complete code. Therefore, it is important to discuss what is and what is not possible to capture with Loom.

Many works have introduced interaction taxonomies and organized the types of interactions used in visualizations. With regard to the taxonomical *dimensions* of interaction [90], Loom supports *stepped*, *passive*, and *composite* interactions and does not support *continuous*. As mentioned in Section 4.1.4, some continuous interactions can be imitated with discrete alternatives. For example, scrolling can be discretized such that the application scrolls in steps. We classify Loom's interaction *types* with regard to the taxonomy of Brehmer et al. [23]. Table 4.2 shows the results. In essence, Loom supports interactions that are pre-determined and discrete.

4.3.2 Comparison to Virtual Containers

There is a tradeoff between Loom and other application archival options such as virtual containers (e.g. Docker). On one hand containers can provide the entire ecosystem needed for an application meaning that complete interactivity is preserved. On the other hand, containers require the entire data and application to still be present within an operating system along with all requirements installed. The size of the containers can quickly rise. Loom alleviates this at the cost of reducing the possible interactions. Moreover, Loom provides a mechanism with which offline visualizations can be partially available online without the need of any server, whereas containers make it more difficult to access a

visualization even in offline modes often requiring internal network configurations and display forwarding to run a graphical application.

4.3.3 Comparison to the Web

Web-based information visualizations of small datasets are sometimes independent of external data sources and servers by default, making it easy to simply save the files for archival. However, most visualizations rely on technology and language standards that change often. When these technologies become obsolete, browsers remove their support and the visualizations fail to run. This has been seen in the discontinuation of Adobe Flash [9] as well as frequent changes in the Javascript and WebGL standards. While Loom also uses Javascript, it relies on the idea of showing images, a very basic structure that can live on for a very long time. Even without the proper Loom code, individual visualization frames are still retrievable from Loom objects.

4.3.4 Suitability for Visualization vs. Other Applications

Visualization applications often rely on external and pre-defined data sources. This creates a unique opportunity for systems like Loom. Loom cannot be used with general utility applications such as Microsoft Word simply because their data source is provided by users (i.e. text) at runtime and is not pre-determined. Moreover, the types of interactions with visualizations are well- and pre-defined, making capturing much simpler. General applications on the other hand support interaction with components that are created on the fly based on user data.

Chapter 5

Delivering Volumetric Visualizations in Real-time

While Loom pushes the boundaries of pre-rendering, the resulting visualizations can suffer from low interaction fidelity and limitations in explorability. When it comes to volumetric data that is often used in scientific visualization settings, explorability, and high fidelity are very important.

In this chapter, we present Tapestry as a different implementation of a Fabric-based architecture. Tapestry delivers explorable scientific visualization of volumetric data to the web. It incorporates an active kernel that renders visual responses live and sends them to the client-side.

Different from Loom, the visualization requests in Tapestry take the form of conventional visualization parameters as opposed to simple frame numbers. Additionally, due to the high interaction fidelity provided, the client's state machine in Tapestry takes the form of Javascript functions.

5.1 Architecture Design

As a Fabric-based architecture, Tapestry decouples the client and the server and separates the application space from the system space.

We do so by formalizing rendering requests as a reduced and restricted interface, and the only interface, between the two spaces. As shown in the system diagram (Figure 5.1), the generation of rendering requests in the application space is asynchronous and distributed. On the server side, rendering requests are automatically distributed to many disparate endpoints through typical web server load balancers and ensures scalability.

The application space maintains the *dynamic states* related to the application and interaction. The system space is dedicated to answering rendering requests and stays stateless without maintaining any application state information.

The two spaces have different life cycles. The system space stays up as long as the cloud service is up. The application space exists as individual instances, with one instance per each session when a user accesses the application, e.g. a web page with embedded 3D visualizations. The application space can have many instances. The system space is a single entity shared by all instances of the application space.

In the application space, a hyperimage is the universal interactive visualization object. Each hyperimage is controlled by an attached Tapestry object in JavaScript, which presents the 3D interactions and automatically requests services from the server, by way of issuing rendering requests. Details in Section 5.1.1.

The system space is cloud hosted on a cluster of nodes. These nodes comprise a Docker Swarm [5]. The swarm abstracts handling of rendering requests into a cluster of microservices implemented in virtualized containers, which the swarm manages altogether as a collection. The system also includes elastic task handling, request routing, and automatic resource scaling. Details in Section 5.1.2.

Connections between the two spaces are simple, short, and transient rendering requests. An application instance can generate many rendering requests concurrently. The system space can answer a large amount of rendering requests simultaneously. The system space does not relate one rendering request with another, and treats each request independently, even when the rendering requests are from the same application instance.

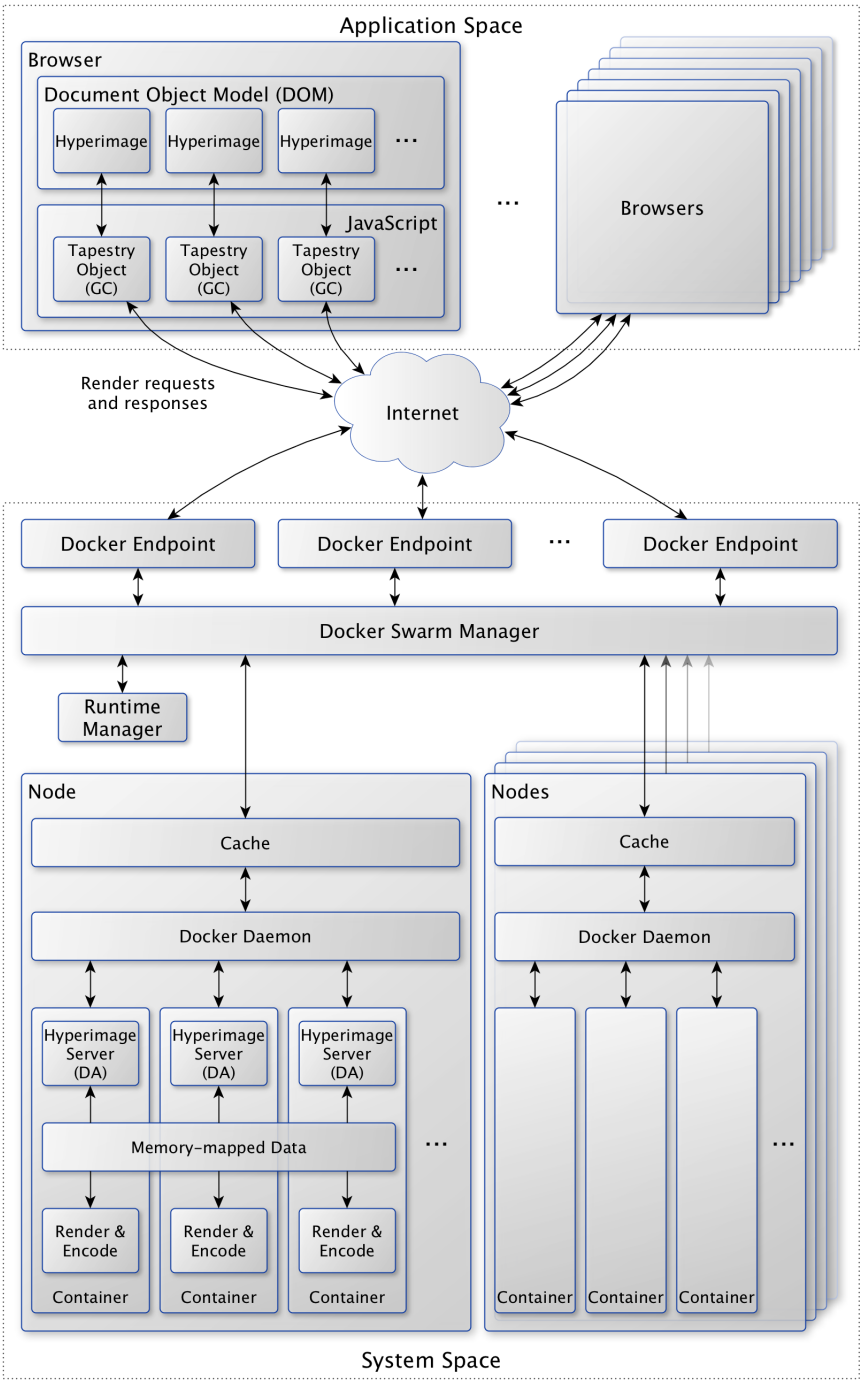

Figure 5.1: The Tapestry system architecture, which separates the application space and system space.

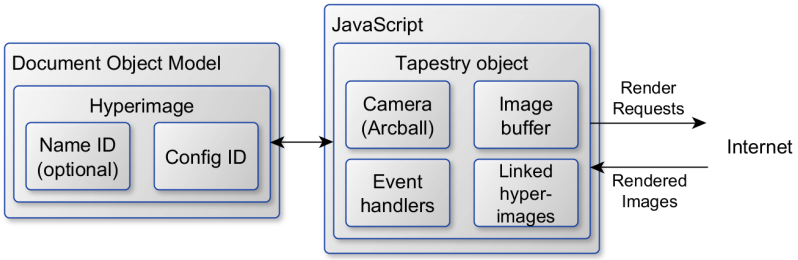

Figure 5.2: Hyperimages reside in the DOM. In the application space, each hyperimage element is paired with a Tapestry object, which handles user interaction and communicate with the Tapestry server.

5.1.1 Application Space

Using Tapestry, the presentation of the visualization resides in a desktop/mobile web browser as an embedded object.

Within a browser, we could consider using the HTML5 canvas or the 3D-enhanced WebGL canvas [37]. However, we chose to use a simple image tag (`<img>`) instead for several reasons. First, HTML5 and WebGL canvases are heavyweight elements with initialization costs. Their performance also relies on the user's hardware. Second, the output of many visualizations is an image and therefore an `<img>` tag is a natural medium that does not need any post-processing and is widely used across the web ecosystem. We refer to our enhanced `<img>` tags as *hyperimages*.

Control of Visualization Objects

Figure 5.2 shows a closeup of Tapestry's application space in a web setting. An application can use as many hyperimages as the developer desires. In this example, we show a single hyperimage in the DOM, but multiple may be present. In essence, a hyperimage is a simple `<img>` tag with extended capabilities. As a user interacts with a hyperimage, a controlling JavaScript object generates and submits rendering requests to the server automatically, updates the received renders and updates the hyperimage's `src` attribute.

The Graphics Context (GC) of each hyperimage is controlled by an attached Tapestry object in the `tapestry.js` JavaScript code. The GC information includes: camera management through arcball, an image buffer for received images, event handlers and a

list of other hyperimages that may be linked to the object. Optional settings such as initial camera position can be sent to the Tapestry constructor if needed.

Listing 5.1: Sample code for adding a hyperimage into a webpage

```
<script> $(".hyperimage").tapestry({}); </script>
<img class="hyperimage" data-dataset="supernova"/>
```

Listing 5.1 shows the full HTML code to embed a 3D visualization on a web page. The second line of Listing 5.1 shows a simple hyperimage of a supernova. The `class` attribute identifies the tag as a hyperimage, and the dataset being rendered is added in the `data-dataset` attribute. Note, `data-*` is the standard prefix for custom attributes in HTML5 [101]. Hyperimages become interactive by replacing the source attribute of the tag. When the user is not interacting, a hyperimage is effectively a simple image.

For time varying data, a hyperimage can take an optional `data-timerange` attribute. The value of this attribute represents the time step range through which the volume can animate. This range is formatted as `<integer>..<integer>`. For example, a value of 5..15 would mean that the hyperimage cycles through time steps 5 to 15 when animated.

In addition to mouse and hand gestures, Tapestry allows a customizable type of interaction: *hyperactions*. Hyperactions provide a way for the DOM to manipulate a hyperimage without user intervention. A simple use case of a hyperaction is a hyperlink in a text that rotates a hyperimage to a specific viewpoint. Hyperactions essentially provide a simple connection between textual content and volume renderings. Any standard DOM element can be converted to a hyperaction by adding three attributes: the class `hyperaction`, a `for` attribute that denotes which hyperimage should be associated with the action, and a `data-action` attribute describing the action itself. For example, a hyperlink that sets the camera position of a hyperimage is shown in Listing 5.2.

When clicked on, this hyperaction sets the camera position of the hyperimage with the `id` of teapot1 to $(10, 15, 100)$. A list of supported actions and their syntax is shown in Table 5.1. The logic behind what hyperactions do is also controlled by Tapestry objects. When a Tapestry object is initialized, it looks at the DOM for hyperimages and their

Table 5.1: Tapestry's list of supported hyperactions

Action	Description
`position(x, y, z)`	Sets the position of the camera
`rotate(angle, axis)`	Rotates the camera `angle` degrees about the given axis
`zoom(z)`	Sets the relative camera Z position
`link(id1, ...)`	Links the viewpoint of other hyperimages to the current hyperimage's camera
`unlink(id1, ...)`	Unlinks the viewpoint of other hyperimages
`play()`	Animates the time steps of a time series dataset
`stop()`	Stops the time series animation
`time(t)`	Changes the timestep to t
`switch_config(name)`	Switches to a new hyperimage configuration

corresponding hyperactions and sets up event handlers for the hyperactions' action. Two example applications in Section 5.2 make use of hyperactions.

Listing 5.2: An example hyperaction that sets the camera position to the given position for the teapot dataset.

```
<a class="hyperaction" for="teapot1" data-action="position=10,15,100">a
   new viewpoint</a>
```

Generation of Rendering Requests

The DOM defines the structure of a web page, and the JavaScript provides interactivity and control. The relationship between a hyperimage (a DOM element) and the related Tapestry object is no exception to that. When a user interacts with a hyperimage through mouse or touch gestures, the corresponding Tapestry object manages callback functions and generates rendering requests as needed. While interaction is happening, it continues to send new requests to the server-side and asks for updated renders.

During interaction (e.g. when rotating), the object requests interaction resolution images (256^2 by default) to allow for smoother movement. When interaction stops, the object requests a viewing resolution image (1024^2).

Rendering requests are sent using the HTTP GET method. As a result, renderings can be saved or shared after interaction just like any image with a valid address. A rendering request takes the form of `http://HOST/DATASET/POS_X/POS_Y/POS_Z/UP_X/UP_Y/UP_Z/RESOLUTION/OPTIONAL`. The `DATASET` parameter denotes which configured dataset should be rendered. The camera position is given by `<POS_X, POS_Y, POS_Z>`, and the up vector is given by `<UP_X, UP_Y, UP_Z>`. `RESOLUTION` denotes the rendering's resolution.

Finally, additional optional parameters can be added as a comma separated string of key-value pairs. For example, to specify the time step in a temporal series.

Listing 5.3: Two rendering requests for a well-known supernova simulation [21]. The values represent camera position, up vector, and image size, respectively. The second request includes an optional time step parameter.

```
http://host.com/supernova/128.0/-256.0/500.0/0.707/0.0/0.707/256
http://host.com/supernova/128.0/-256.0/500.0/0.707/0.0/0.707/256/
   timestep,5
```

Tapestry objects also control the volume of rendering requests. For example, a user's mouse can typically emit up to 125 *move* events per second (on a common 125Hz mouse). We set a default policy: let every fifth event trigger a rendering request. This policy generates up to 25 rendering requests per second.

Due to the minimal interface between the client and server, requests can also be generated in batches and by scripts, for more complicated applications. Section 5.2 shows this in more detail through several applications.

Non-Invasive Embedding

From an application developer perspective, Tapestry provides non-invasive integration in clients. In other words, it is simple to integrate and customize and does not cause any global changes in the host web application.

More specifically, hyperimages in the client are self-contained and do not share state with each other. This means that they can be independently added or removed in a page.

Another aspect of non-invasiveness are hyperactions. Hyperactions are behaviors, not objects. In other words, they can be added to a variety of HTML elements (e.g. buttons, hyperlinks, images, etc.) and enable interaction with a hyperimage. Those HTML elements can be freely styled and edited by the developer.

Users of scientific visualization often need to tweak and edit visualization tools to add new capabilities. To facilitate this, the Tapestry server can take an optional *app* directory as input at runtime. JavaScript, HTML, or CSS source code in the *app* directory overrides

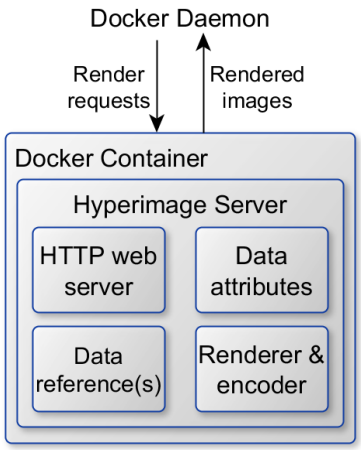

Figure 5.3: A container is the basic processing unit in Tapestry's system space. Each container runs an instance of the hyperimage server.

those of Tapestry's default, allowing for easy hot-swappable functional changes. In other words, client-side changes to a user's application do not require a re-compile or restart of the Tapestry service.

5.1.2 System Space

The sole concern of the system space is to process rendering requests. It is a task-parallel computing system, using distributed resources that auto-scale on demand.

In system space, we make a distinction between a *physical node*, a *Docker container*, and a *hyperimage server instance*. A physical node refers to the real machine on which multiple Docker containers may be launched. There may be multiple physical nodes. A Docker container is an in-memory virtual operating system.

Figure 5.3 shows a single Docker container. Each container includes an instance of a hyperimage server, which is a web server that manages attributes of given datasets, and handles any rendering requests it receives in sequence.

Container-Based Rendering Services

Virtualization and containerization are classic concepts in software architecture [69]. Open-source software container platforms have become popular, including for HPC computing services [94].

We chose Docker [5] containers because they are lightweight, and provide a robust and simple interface. Each Docker container includes a small, stripped-down version of an operating system as well as all the dependencies needed to run an application independently. Multiple containers can run on the same node.

Each physical node runs a local Docker daemon, which manages all running containers on that node. Across nodes, we use Docker Swarm as another layer of abstraction on top of a collection of physical nodes, allowing a pool of containers to have unified entry points as well as leverage Docker Swarm's load balancer.

In Tapestry, each Docker container is based on a stripped down version of Ubuntu, which runs a hyperimage server instance inside. The Docker Swarm Manager monitors and manages the containers, routes incoming rendering requests, and load balances the containers using its internal Ingress load balancer [55].

When a hyperimage server starts, it loads all pre-configured datasets into memory using a memory-mapped loading operation. In other words, containers that reside in the same worker node only load the data once and only during system startup.

Hyperimage Server and Data Attributes

A hyperimage server is initialized once and lives for the lifetime of the cloud service. A hyperimage server takes a configuration directory during initialization. All valid configuration files – properly formatted JSON files – within this directory are used to provide data attributes for the server instance. These configuration files, provide basic information about the datasets. An example configuration file is shown in Listing 5.4.

Listing 5.4: Example configuration file providing data attributes

```
{
    "filename" : "/path/to/data/magnetic.bin",
    "dimensions" : [512, 512, 512],
    "colorMap" : "cool to warm",
    "opacityAttenuation" : 0.5,
    "backgroundColor" : [38, 34, 56]
}
```

The configuration files are a list of key-value pairs. A complete list of keys and possible values for configuration files can be found in our previous work [74]. These parameters are standard visualization data attributes. Basic information about the dataset, such as `filename` and `dimensions` are required, but most others are optional and can revert to default values. Different transfer functions require different configuration files. However, they can all point to the same dataset. Memory-mapping assures that the dataset used by different configurations are only loaded to memory once for each node.

Additional configuration keys available also include `isosurfaceValues` and `specular` to control isosurface rendering if desired. Note that Tapestry uses OSPRay's *implicit isosurface rendering* to provide images of surfaces. Implicit isosurfaces avoid the need to explicitly compute and store surface geometry, which allows the server to remain stateless.

Currently, the server handles raw binary and NetCDF files, two common formats for scientific data. The filename provided may be a path to a single file, i.e. a static volume, or a path with wildcard characters to describe multiple volumes, i.e. a time-varying series. Example `filename`s for a time-varying series could be: `"~/supernova/*.bin"` for all available time steps or `"~/supernova/time_[2-7].bin"` for 5 specific time steps.

During initialization, the datasets referred to by the configurations are loaded. Since each physical node may run multiple server instances, we memory-map the datasets when loading. This allows the physical node's host operating system to maintain an in-memory map of a file that can be given to each server instance. This reduces I/O costs and allows using multiple configuration files to reference the same dataset without additional overhead.

Attributes about the dataset from the configuration, such as transfer function or data variable, are kept alongside the reference to the data. Multiple configuration files may reference the same dataset, for example, using varying transfer functions. This flexibility allows for more power in the rendering requests.

Handling of Rendering Requests

After being routed from a unified endpoint to a specific Docker container, a rendering request is handled by a hyperimage server. Rendering requests from the client ask for an image URL in which various parameters are embedded. Image requests are processed by the C++ web

server, built with the Pistache library [60], by first parsing the options and then rendering the requested image using the OSPRay renderer.

Each incoming rendering request contains the dataset, camera position, up vector, the resolution of the render, and potentially time-step. Camera and renderer settings are updated accordingly.

OSPRay performs the rendering according to the above parameters. The life-cycle of the OSPRay rendering objects in each server are equal to that of the hyperimage server itself. Data and rendering attributes are pre-configured per volume during hyperimage server initialization. When the render completes, we composite the OSPRay framebuffer onto the appropriate background color and encode as a JPG image. There is no need to store the image to disk on the server, so the encoding is done to a byte stream in memory. At this point, all information about the camera position and other dynamic state parameters are no longer needed nor held.

The web server sends the rendered image as JPG byte stream (e.g. `image/jpg` MIME type) from the rendering module. The Docker Swarm Manager, which routed the request to this container, handles responding to the appropriate user. *The hyperimage server itself remains oblivious to whom it has communicated with.*

Elastic System Operation

Job Assignment and Runtime Management.

Using a single container, rendering requests from n users will be queued up by the web server. Each request will occupy the container until rendering and network transfer of the image is complete. With multiple containers, any container available can be selected for any given rendering request. Sequential requests from a single user can be routed to different containers on different physical nodes. This has two main benefits: (i) new rendering requests can be processed while other requests are blocked for I/O, network transfer, or rendering; and (ii) elastic routing provides fault tolerance when a hyperimage server or physical node goes down.

The volume of rendering requests is variable over time and hard to predict. We monitor the current load on all containers and scale the number of containers up or down accordingly, through the runtime manager (RM) shown in Figure 5.1.

Our RM, like RMs on typical cloud platforms, implement elasticity by periodically checking CPU usage across all containers, and start new containers or close idling containers as needed. In our previous work, we showed how Tapestry leveraged such auto-scaling on an institutional cluster [74]. In this work, we deploy Tapestry on Amazon AWS as a microservice and, to this end, benefit from Amazon's auto-scaling RMs transparently.

Cache Container. In each physical node, we have added an Nginx cache container intercepting all messages between hyperimage servers and the outside. In a completely transparent manner, this enables caching for the Tapestry microservice instances. Server responses are now cached based on the incoming request. This improves efficiency and scalability for many use cases. For example, commonly used view angles, isovalues, etc. in repeated batches of renderings for hypervideos and tiled renderings can now be simply reused, saving hyperimage servers to handle new rendering requests. Note that client-side caching inside web browsers also take place transparently by browsers themselves.

Controllable Granularity. Tapestry's server-side is a task-parallel engine. As known for task-parallel systems in general, the granularity of the tasks can affect the parallel efficiency of the overall system. In this work, we have added a tiling mechanism to Tapestry as an option so that an application can choose to use finer granularity to achieve better performance.

With tiling, a single hyperimage can be divided into many `<img>` tags on the client-side. Each tile represents a portion of the final render and is rendered on a different container in parallel to other tiles. Using tiling, the client-side creates a render request for each tile and sends them to the server-side. Once the response comes back, the appropriate `<img>` is updated with the result.

The setting "`tiling,TILE_NUMBER-N_TILES`" is an optional parameter in the rendering request to specify tiling. For example, `tiling,0-16` denotes that the rendering request is for the first tile out of a 16-tile render. Once this rendering request reaches a hyperimage

server, the server calculates the portion of the volume that it needs to render and updates the OSPRay camera's clip space.

When rendered tiles are returned to the client-side, the tiles are placed in the DOM in their own corresponding `<img>` tag. Because each tile request can be sent independently and routed to the correct position in the hyperimage, there is no explicit compositing step required. That is, we provide *stitch-free* tiling.

Multiple Endpoints. Docker Swarm uses an Ingress load balancer [55]. The setup allows any physical node to be an endpoint for incoming requests. The requests are then routed to a free container. As a new addition, in this work, we have added support for multiple endpoints in the client (`tapestry.js`). The `host` parameter in a Tapestry object can be set to an array of host addresses. Endpoints are then chosen using a round-robin approach in the client in Tapestry objects. This achieves two purposes. First, the problem of bottlenecking at a node's inbound traffic is alleviated. Second, browsers typically only open a limited number of sockets per host address (e.g. Chrome currently defaults to opening 6 connections per destination host (endpoint) [3].) By using multiple endpoints, Tapestry objects can take advantage of more open sockets.

In the case of Amazon's cloud, AWS also has a load balancer that provides the same effect as Docker Swarm's and is called the Elastic Load Balancer (ELB). Multiple ELBs can target the same set of machines to provide a similar effect on AWS as on our institutional Docker Swarm. The address of the ELBs can be used as endpoints in Tapestry clients.

5.1.3 Deployment on Institutional Clouds

Tapestry's source code comes with a command-line interface (CLI) named `tapestry.sh` that simplifies setting up and running the backend on institutional clouds. Linux and Docker Swarm are the only requirements for running the Tapestry system. With Docker Swarm installed, users can simply run `./tapestry.sh build` and `./tapestry.sh run` to run the system. Since Tapestry is built inside Docker containers, the build is guaranteed to be successful on machines that run Docker. In that regard, Docker has simplified portability. The command-line interface also contains other sub-commands such as `scale` (for manually scaling the system), `example` (to download and run the examples), `cache_report` (to view

the number of cache hits and misses) among others. Extra features of the interface can be seen using the `help` subcommand.

5.1.4 Deployment on Amazon AWS as a Microservice

Although the achieved performance metrics on public clouds may be lower than on institutional clouds, public facing cloud platforms, such as Amazon AWS, provide true Internet-scale availability and accessibility at very affordable cost levels.

To create a Tapestry service on AWS from scratch, only a few steps are needed. AWS provides a load balancer that is instrumental in distributing rendering loads across multiple machines. For the setup, an AWS load balancer needs to be started with its listening port set to a publicly accessible port for the service; typically the default HTTP port 80. The load balancer must then be configured to forward traffic to some alternative port (e.g. 8080).

After that, an AWS Elastic Container Service (ECS) service can be created. Tapestry's Docker image then needs to be uploaded to Amazon's cloud-based registry and needs to include any necessary data and configurations. The ECS service needs to point to this image and use the previously specified private port (8080). Finally, the user needs to scale the service as necessary; often a higher number than would be used on an institutional cloud because AWS shares the resources with other users and services.

In studying the performance of Tapestry on Amazon AWS, we were mostly interested in choosing the optimal type of machine and measuring the price for a desired frame-per-second performance. In our tests, we spawned various numbers of different machines and sent rendering requests of different image sizes and measured the round trip time. As a summary of the outcome, we found to support a large number of simultaneous users, using a large number of small T2 type instances is more cost effective. However, for super resolution renderings for a few users, the Compute-Optimized machines are more suitable. More detailed results are shown in Section 5.3.4. Additionally, to simplify usage on cloud services, we have released a Docker image of Tapestry [1].

[1] https://hub.docker.com/r/seelabutk/tapestry

5.2 Applications

In this section, we describe three application development settings enabled by using the Tapestry microservice. Specific application performance results are in Section 5.3.6.

5.2.1 Embedding Visualizations into Web Pages

Hyperimages can be easily added to a web page using HTML tags and a short JavaScript function call. To integrate hyperimages into a page, the developer must include the tapestry.js file and its dependencies: arcball.js, sylvester.js, math.js and jQuery.js. Then, one line of JavaScript needs to be called to initialize all hyperimages: `$(".hyperimage").tapestry();`

This call creates a Tapestry object per hyperimage tag. Parameters such as default size of the hyperimage and camera position can be sent to the object through the constructor.

Time-Varying Data Animation (Wikipedia Example)

Listing 5.5 shows the changes needed to include a hyperimage of a time-varying dataset into a Wikipedia page.

Figure 5.4 shows the Wikipedia page on tornadoes after the modification. The page includes a hyperimage linked to a series of time steps from a tornado simulation dataset. Two hyperactions can be seen in the code. Users can click a hyperaction to play or stop the animation, while still having the ability for 3D interaction with the volume rendering.

Listing 5.5: Code for adding a hyperimage of a time varying simulation into the Wikipedia tornado page.

```
$(".hyperimage").tapestry({
    "host": "http://host.com:port/",
    "width": 256, "height": 256, "zoom": 300, "n_timesteps": 20
});
<img id="timeseries" class="hyperimage"  data-volume="tornado"
    data-timerange="0..20"/>
<a class="hyperaction" for="timeseries" data-action="play()"></a>
<a class="hyperaction" for="timeseries" data-action="stop()"></a>
```

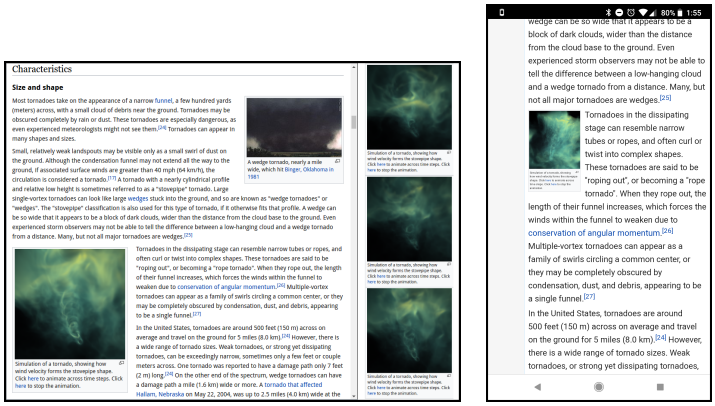

Figure 5.4: Left: embedded a volume rendering of tornado (dataset details in Table 3) in a Wikipedia page on tornadoes. Users can start and stop an animated temporal sequence. Right: The same page also works on mobile phones. The page used to hold a static image showcasing the shape of a stovepipe tornado. Now users can interactively see the temporal progression of the natural phenomenon.

Multiple Linked Views (NASA Example)

Here we show a NASA educational outreach page explaining supernovae. The relevant code changes are in Listing 5.6. The modified page is shown in Figure 5.5.

The page now contains four hyperimages showing consecutive time steps of a supernova simulation. The views can be linked and unlinked with the hyperaction in the caption. When linked, all four hyperimages move together when a user interacts with any one of them.

5.2.2 Controllable Movies of Scientific Visualization

By unifying the interface of the Tapestry microservice as simple rendering requests, we can achieve more complex application logic, for example, for making movies of scientific visualization.

Traditionally, making a visualization movie requires creating the keyframes first. Then, a movie is created by rendering all of the intermediate frames sequentially. Making changes to an already-made movie requires a user to have access to significant computing resources, and is usually a very time consuming process.

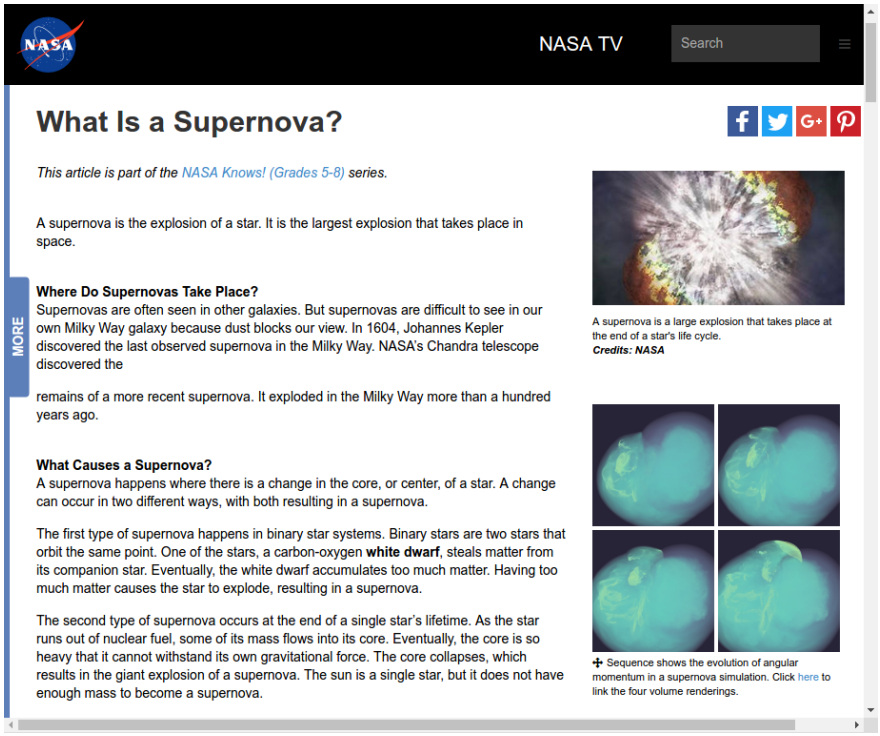

Figure 5.5: Embedding four time steps of a supernova simulation into a NASA educational web page (dataset details in Section 5.3). The four hyperimages (bottom right) can be linked or unlinked using the hyperaction in the caption below it. Previously, the page had only a static figure (top right) showing an artist's rendition. Now users can also interactively explore how a supernova evolves over time.

Listing 5.6: Code needed to insert the four linkable hyperimages and hyperaction into NASA's supernova web page

```
<script>
$(".hyperimage").tapestry({
"host": "http://host.com:port/",
"width": 128, "height": 128, "zoom": 300
});
</script>

<img id="s1" class="hyperimage" data-dataset="nova1" />
<img id="s2" class="hyperimage" data-dataset="nova2" />
<img id="s3" class="hyperimage" data-dataset="nova3" />
<img id="s4" class="hyperimage" data-dataset="nova4" />

<a class="hyperaction" for="s1" data-action="link(s2,s3,s4)"></a>
```

Using the Tapestry microservice, we can make the movie-making process interactively controllable by a user from within a simple web browser. While offloading all rendering tasks to the microservice, we simplify the application space of the movie-making process to just the textual representations of the keyframes (i.e. the corresponding rendering requests). We call these application-space constructs, *hypervideos*.

Hypervideos can be embedded in HTML with the `class` attribute set to `hypervideo`, and their `data-keyframes` set to a JSON file. Alternatively, developers can set the `data-keyframeid` attribute to the `id` of a script tag that contains the JSON. Listing 5.7 shows an embedded hypervideo with two keyframes.

Using and interacting with hypervideos is different from traditional movies in important ways.

First, each keyframe can be presented on a web page as a hyperimage, which has all of the interactivity described in Section 4.1, including allowing the user to alter the keyframe by changing the view. The generation of intermediate frames is automatic. We use linear interpolation for changes in timesteps, isovalues, and zoom levels; we interpolate camera rotations using slerp [86].

Listing 5.7: Sample script for a hypervideo with two keyframes

```
<script id="video" type="text/json">
{
"keyframe0": {
"rotation": [-0.72, 0.30, 0.62, 0.51, 0.83, 0.19, -0.46, 0.45, -0.75],
"zoom": 500, "timestep": 0, "isovalue": 0.2
},
"keyframe1": {
"rotation": [0.44, -0.16, 0.88, 0.43, 0.90, -0.05, -0.78, 0.40, 0.46],
"zoom": 200, "timestep": 20, "isovalue": 0.7
}
}
</script>

<div class="hypervideo" data-keyframeid="video" data-dataset="supernova"
   ></div>
```

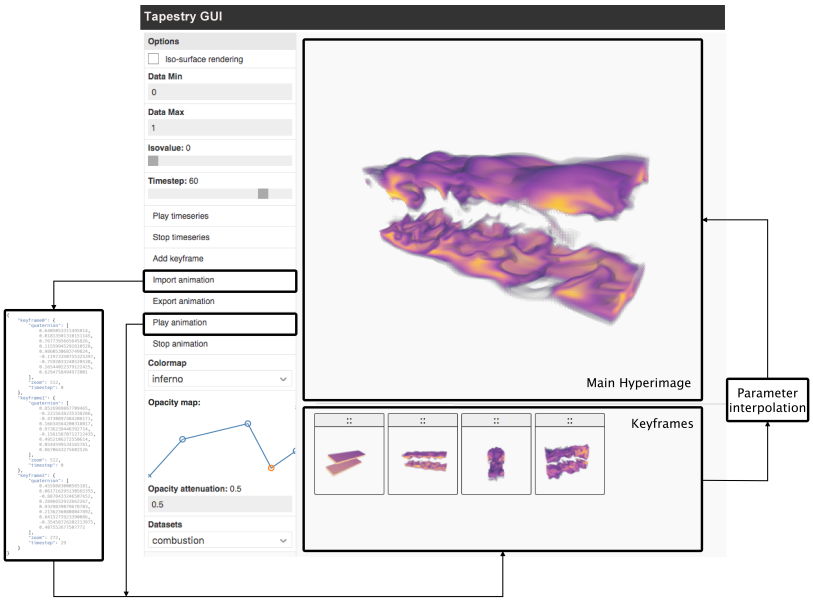

Figure 5.6: A webpage for creating and manipulating hypervideos. A user can add keyframes, edit existing keyframes, and export movies. Editing the movie can be to modify camera angle, time step, isovalue, color map, etc.

Second, in a traditional movie, only the keyframes are controllable. In contrast, due to the Tapestry microservice treating all rendering requests in the same way, we turn each individual frame in the movie into a hyperimage. In this way, when a viewer watches the movie, he or she can pause the movie at any time to interact and navigate around the dataset freely.

Third, because of the microservice's availability, the movie, i.e the hypervideo, can remain text-only, and hence remain compact, easily editable, sharable, and version controlled. In addition, while changing number of frames, screen resolution, splitting and re-joining movies etc., are hard for traditional movies, they are trivial tasks for hypervideos.

For creating hypervideos, Figure 5.6 shows a GUI that is essentially a web page. A user can interactively add and control the key frames. When the keyframes are set, the user can play the animation or export the video in the form of JSON text or as MP4 (rendered and encoded server-side using *ffmpeg*). At all times, the Tapestry microservice serves as the rendering engine.

The performance of hypervideo renderings is presented in Section 5.3.6.

Figure 5.7: A volume rendering of the turbine blade dataset shown through HoloLens.

5.2.3 Augmented Reality and Power-Wall

The endpoints of the Tapestry microservice is served by Docker Swarm following standard HTTP protocols. This kind of generality allows any application to simply access the endpoints (e.g. via Linux's `curl`). When using the Tapestry microservice, the application space does not have to be related to web browsers at all. We further provide two demonstrative examples as follows.

For the first example, we developed HoloTapestry, a C# application for augmented reality using the Tapestry microservice. This prototype runs on a Microsoft HoloLens device and performs stereo renderings using two textured planes, rotated so they stay normal to the viewer's eyes. Each plane independently updates its texture by making rendering requests to the microservice based on the current camera parameters from the HoloLens. Transparency is achieved by setting the background color of the renders to black as is standard in HoloLens applications.

In result, Tapestry microservices allowed us to deliver volume renderings of a 7.5GB dataset to an AR device with 2GB of memory by writing about 100 lines of code. Figure 5.7 shows a view of the turbine blade dataset on a desk. The performance of HoloTapestry is in Section 5.3.6. HoloTapestry is open-source [2].

For the second example, we target power-wall displays, which is arguably one of the most prized tools for demonstrating advances in science and engineering. Traditionally, each power-wall facility is accompanied by its own computing cluster. Due to the typical tiled

[2]https://github.com/seelabutk/holotapestry

Figure 5.8: A user using Tapestry to inspect a 3D printed wind turbine on a 4×3 power-wall. Renderings are 2048×2048 in resolution.

nature of power-walls, producing super-resolution renderings using the Tapestry microservice is straightforward. One can use a short Shell script that batch-generates rendering requests through `curl`. Or, one can run a web browser across the power-wall and have the browser transparently issue the batch of rendering requests, one per each tile in the image, in order to achieve parallel acceleration on the server side. In both cases, a lightweight single-node can deliver data-intensive visualizations onto the whole power-wall.

Figure 5.8 shows a user using Tapestry to inspect defects in a 3D printed wind turbine blade on a 4×3 power-wall display. The volume is created by scanning the actual 3D printed model using neutron scattering [20]. The renderings are at 2048^2 resolution, rendered in 256 tiles (128^2 pixels per tile) in parallel. The tiles are synchronized using a global barrier. Tiled-based performance is detailed in Section 5.3.1.

5.3 Results and Discussion

Our testing platforms include our institutional cloud and Amazon AWS instances. Our institutional cloud setup includes three machines each with 24 physical cores (dual-socket Xeon E5-2650 v4, 2.9 GHz, 128 GB memory) and three machines each with 28 cores (dual-socket Xeon E5-2650 v4, 2.9 GHz, 256 GB memory).

On AWS, we tested seven different types of instances. Table 5.2 shows the detailed list. The "d" suffix (e.g. c5d.xlarge) refers to AWS instances with SSDs. For our system, the SSDs do not affect the runtime performance, only microservice initiation time.

Table 5.2: Amazon AWS instances used in this work. The t2 prefix (e.g. t2.micro) refers to general purpose instances, while the c5 prefix refers to compute optimized instances. The containers column shows the maximum number of containers allowed by AWS on each particular instance.

Instance	Core Cnt	Memory	# Containers
t2.micro	1 vCPU	1 GiB	1
t2.medium	2 vCPUs	4 GiB	2
c5d.large	2 vCPUs	4 GiB	2
c5d.xlarge	4 vCPUs	8 GiB	3
c5.2xlarge	8 vCPUs	16 GiB	3
c5d.2xlarge	8 vCPUs	16 GiB	3
c5.9xlarge	36 vCPUs	72 GiB	7
c5d.18xlarge	72 vCPUs	144 GiB	14

Our testing includes: (i) using 1 single container to serve 1 rendering request (Section 5.3.2), (ii) using an institutional cluster to serve a varying number of emulated streams of rendering requests (Section 5.3.3), (iii) using Amazon AWS cloud to serve a varying number of emulated request streams (Section 5.3.4), (iv) using AWS cloud to serve a varying number of simulated users (Section 5.3.5), and (v) performance of demonstrative applications as experienced by a user (Section 5.3.6).

Among the above tests, (i) - (iii) are to understand how the Tapestry server performs, independent of user behavior. (iv) is to understand the quality of service received by a cohort of simultaneous users performing exactly the same kinds of operations. (v) is to understand how a single user experiences applications supported by the Tapestry microservice. Note that end-users are not affected by dataset load time in these tests because all datasets are pre-loaded before the service starts.

5.3.1 Configuring the Tapestry Microservice

This section discusses application policies to consider when deploying Tapestry on the cloud.

When deploying on Amazon AWS, because virtual instances have to share their physical nodes with others, Amazon by default sets a low cap on the number of containers. For example (as shown in Table 5.2), on c5d.18xlarge (with 72 vCPUs), the Amazon imposed container count cap is 14, which translates to a 0.2 container/core ratio. Because this is much lower than the 0.8 ratio on institutional cloud (explained in Section 5.3.3), we use the max number of containers allowed by AWS.

For applications to run optimally on the cloud, there are three accelerations to consider, all of which are independent of Tapestry. Instead, they are solely application-side policies.

First, use tiling. Instead of sending a rendering request for a 1024^2 image, send 16 rendering requests of 256^2 tiles. These per-tile rendering requests will be answered by the Tapestry microservice in parallel. For example, a t2.medium instance has 2 vCPU and 2GB memory, each available for 4.6 cents/hour. It's easily affordable, and beneficial for fault tolerance, to get a cohort of 100 t2.mediums to use for Tapestry.

We have found a simple and general heuristic to set tiling factor to 16. A tiling factor of 4 still limits the amount of parallelism that can be exploited. A tiling factor of 64 creates too much management overhead for the client. Based on our tests, a tiling factor of 16 reliably leads to 3 to 4 times faster rendering performance, as compared to when tiling is not used. Tile size or image size of 64^2 or smaller is to fine grained. In all our demo applications, we lower bound tile size to 128^2.

Second, use a lower interaction-resolution and a higher viewing-resolution. As discussed in Section 2.3, level-of-detail is very effective to ensure user-experience. Specifically, when needing a visualization at a viewing resolution of 1024^2, during interaction for faster response time, it is helpful to use a lower interaction resolution. Regardless of whether rendering for interaction- or viewing-resolutions, all of our demo applications use tiling (to benefit from parallel server-side rendering).

Third, use multi-threaded downloading. Most modern web browsers implement this by default. For example, Chrome automatically opens 6 asynchronous socket connections for each destination host. When accessing Tapestry from a non-browser client (e.g. `curl`), we have also found parallel connections helpful.

Hence, we have set up our tests of Tapestry microservices, in Sections 5.3.4, 5.3.5, and 5.3.6, using the following assumptions: (1) each user has 6 concurrent request streams, (2) tile-based rendering requests, (3) when testing for user experience, use a viewing-resolution of 1024^2 and a interaction-resolution of 256^2.

5.3.2 Rendering Pipeline Performance

We benchmarked the rendering and encoding process using three variables that affect render time: image size, level of attenuation of a ramp opacity map, and number of samples per pixel. We used 6 image sizes (64^2, 128^2, 256^2, 512^2, 1024^2, and 2048^2), 4 attenuation values (1.0, 0.5, 0.1, and 0.01), and 4 sampling rates (1, 2, 4, and 8). The target hardware was a 24-core node of our institutional cluster with a single container. We then tested each combination of these parameters, resulting in 96 test cases. We repeated each of the 96 cases 10 times with the camera at a randomized positions to simulate the effects of the volume being at different distances and angles. We calculate the average time taken for 10 renders for a given test case. To see the effect of image sizes, we then averaged the times for each image size. This simulates possible variation in image quality within same-sized images.

The target datasets were: supernova, isotropic turbulence, and magnetic reconnection (described in Table 5.3). All three datasets are structured grids of floating point values. To measure rendering time, each image was rendered to OSPRay's internal framebuffer and was then discarded to avoid buffer copy or encoding time. We then tested the encoding time (without saving to disk) separate from render time. Results are shown in Table 5.4. Note that rendering time does not necessarily increase linearly with image size (a known characteristic of ray-tracing [62]).

The fastest rendering case was unsurprisingly 64^2 image size. Within the test cases that used a 64^2 image, attenuation of 0.1 and sample rate of 1 resulted in the fastest renders at 0.001 seconds, approximately 1000 frames per second. On the other hand, the slowest renders occurred with 2048^2 images.

We also compared the encoding time of PNG vs JPG (at 100% quality). PNG was the image format used in our previous work [74]. On average, JPG was 2.5 times faster in encoding than PNG and generated byte streams were generally smaller.

In our experiments, the size of the rendered images varied between a few kilobytes for low resolutions up to under 300 KB for 2048^2 images. The exact size of the generated images depends on the content of the rendering.

Table 5.3: The datasets used in this work. For time-varying data, varying time steps were used during testing.

Dataset	Size per Volume	Spatial Resolution	Time Steps
Boston teapot with lobster	45 MB	$356 \times 256 \times 178$	1
Isotropic turbulence[33]	64 MB	$256 \times 256 \times 256$	1
Jet flames[107]	132 MB	$264 \times 396 \times 66$	122
Superstorm[81] (1 run)	201 MB	$254 \times 254 \times 37$	49
Tornado [105] (wind velocity)	257 MB	$480 \times 480 \times 290$	600
Supernova[21]	308 MB	$432 \times 432 \times 432$	60
Magnetic reconnection[48]	512 MB	$512 \times 512 \times 512$	1
Turbine blade[20]	7500MB	$1589 \times 698 \times 1799$	1

Table 5.4: Average benchmarking results for rendering requests using the supernova, isotropic turbulence, and magnetic datasets. The round-trip time for each request includes render, encode, and transfer time to and from the server with JPG encoding.

Image size	Rendering time (s)	PNG Enc. time (s)	JPG Enc. time (s)	Round-trip time (s)
64×64	0.003	0.005	0.003	0.009
128×128	0.004	0.011	0.005	0.016
256×256	0.009	0.035	0.012	0.030
512×512	0.024	0.122	0.037	0.092
1024×1024	0.083	0.452	0.147	0.284
2048×2048	0.338	1.651	0.580	1.066

5.3.3 Tapestry Server Throughput

In order to evaluate our system's throughput, we implemented a stress test of Tapestry microservices running on our institutional cluster. We orchestrated multiple test machines to send rendering requests to Tapestry simultaneously. In other words, each test machine sends a different request stream to the server.

The testing master starts by spawning testing workers on the test machines. The master then waits until all test workers have finished their tests. Test workers use `curl` to send rendering requests at a rate of 25 requests/second, while randomly changing rendering parameters (e.g. camera position) for each request. Finally, the master reads off the test logs from a shared queue and saves to disk. The logs list request-sent and response-received times that allow us to measure the average time it takes our system to respond to rendering requests. This throughput testing suite is written in Python and is included in the Tapestry repository.

To increase the load on the system, we simply increase the number of test workers. Like in Section 5.3.2, initially our test target was one Tapestry container in a single 24 core node of our cluster. We ran each test 100 times on the supernova, turbulence and tornado datasets

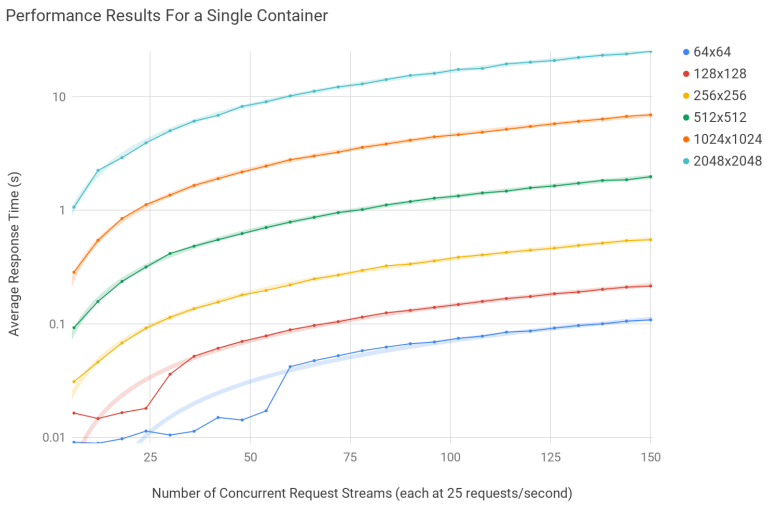

Figure 5.9: System throughput results showing request rate vs. response time for various image sizes in log scale. The linear regression trendlines are over-plotted indicating the linear growth of response time in relation to the number of concurrent request streams.

(Table 5.3). For each dataset, we generated rendering requests for six image sizes: 64^2, 128^2, 256^2, 512^2, 1024^2, and 2048^2.

We then averaged the response time collected, to show an overall system throughput under a mixture of different sizes of rendering jobs. Figure 5.9 shows the scaling curves for various image sizes. When doubling image size, average response time approximately increased by a factor of 4, which is expected.

Then, we tested for the effect of the number of containers per node. In this test, we kept the number of testing workers constant (150), and varied the number of Tapestry containers. Figure 5.10 shows the results for three image sizes. For all image sizes, as we gradually increase the number of containers from 1 towards 20, average response time improves. After reaching 20 containers, adding more containers did not yield noticeable improvements.

With the hardware being a single node with 24 physical cores, getting best performance with roughly 20 containers suggests roughly a 0.8 container/core ratio. Through additional testing, we found this ratio to be quite consistent on institutional cloud.

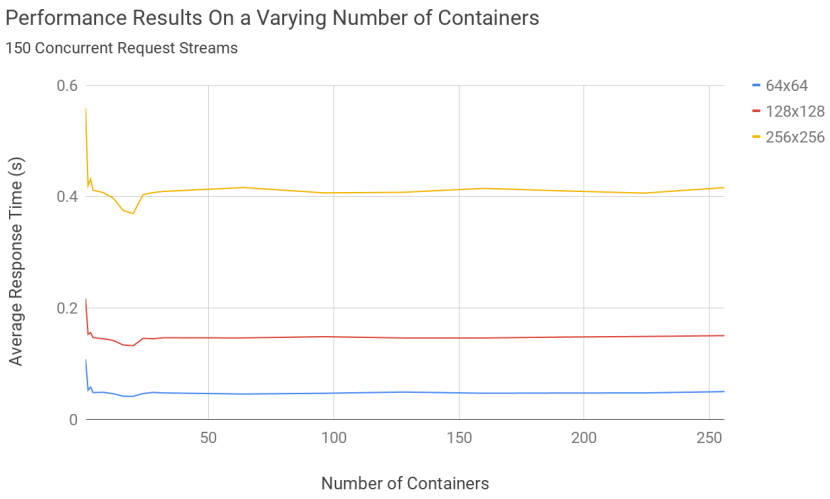

Figure 5.10: Results showing the relationship between the number of containers in our institutional cloud and average response time. The optimal number of containers is shown to be 20 for a machine with 24 physical cores.

5.3.4 AWS Microservice Throughput

Next, we evaluated Tapestry's performance on Amazon AWS. In particular, we looked at the relationship between FPS vs. Price over various tile sizes: 64^2, 128^2, and 256^2.

Since Tapestry is a compute-intensive service, we tested Amazon's Compute-Optimized instances as well as T2 Performance instances [13]. We chose T2 machines because of their ability to sustain CPU workload and low costs [13]. For each instance type, we ran different number of machines. For more powerful machines we were limited to lower quantities due to Amazon's policies.

For the supernova dataset, Figure 5.11 shows FPS vs Price for 6 and 120 concurrent request streams with all of our tested AWS instance types. Each point in the scatter plot represents an AWS instance type and configuration.

For example, Figure 5.11-top shows the cost to sustain 10 FPS when rendering tiles of 256^2 is approximately \$4/hour. Please note, tiling lets applications transparently leverage server-side parallel rendering; when an application requests tiles of 256^2, the target image resolution is actually 1024^2.

To evaluate the choices of AWS instances, we used 120 concurrent request streams and a tile size of 128^2 (i.e. targeting a typical desktop visualization resolution of 512^2).

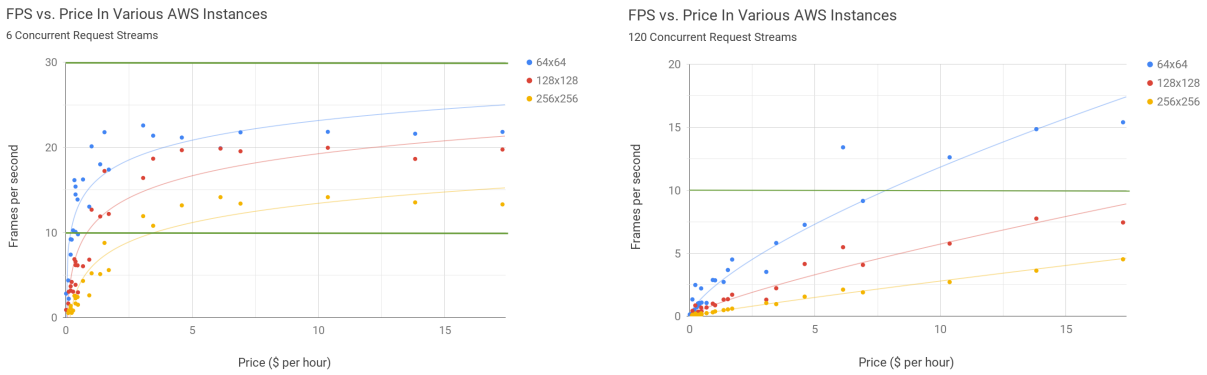

Figure 5.11: Graphs showing FPS vs. price on Amazon AWS for 6 (top) and 120 (bottom) concurrent request streams. Each point in the scatter plots belong to a different AWS instance and configuration. 10 FPS and 30 FPS are marked in green.

Figure 5.12 shows the performance of different instance types in blue for 120 streams. The cost of these instances can be seen in 5.12-right. It appears that the cost correlates quite well with the desired FPS. The two graphs also show that although large Compute-Optimized machines (towards the right) perform better, they are less cost-efficient. A reason may be that larger machines are more suited for fewer users and large tile sizes.

Figure 5.12 shows that by lowering the number of request streams to 6 (red bars), the rendering speed of the Compute-Optimized instances grew much more than a large number of smaller machines such as 100 *t2.medium* instances.

Furthermore, we compared the performance of 100 *t2.medium* machines and 3×72 core *C5D.18xlarge* machines. Based on the changes in the number of concurrent requests from 6 to 120, we used the least squares fitting model to estimate where the performance of the two meet. The fitness of the model had a root mean square error of 0.019. Figure 5.13 shows that at 380 concurrent request streams (i.e. about 60 simultaneous uses), 100 *t2.medium* instances become more cost-efficient.

5.3.5 User Experience Benchmarking

To test our system's performance under realistic workloads, we used "monkey testing", a standard approach to stress-test web pages. Monkey testing involves simulating interactions across elements of the page. We used this on hyperimages to simulate user interaction.

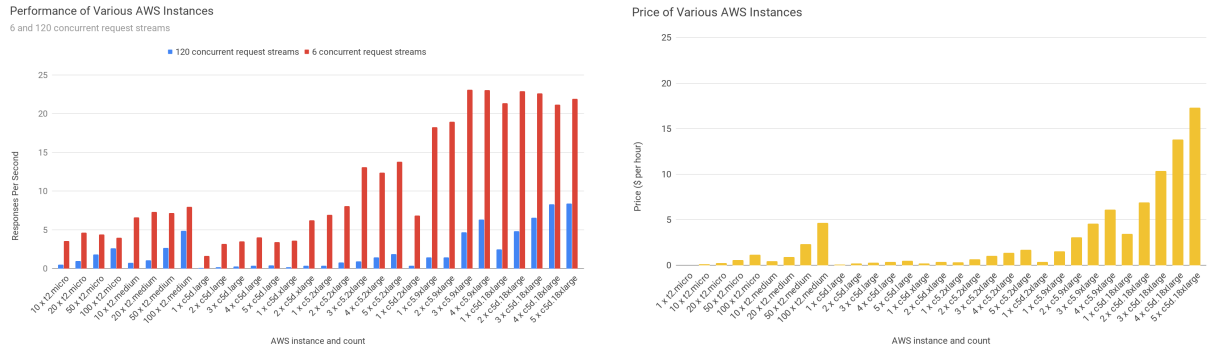

Figure 5.12: Left image shows a comparison between the rendering performance of various AWS instances for 120 and 6 concurrent request streams (both at a request rate of 25 FPS). In a Chrome browser that uses 6 request streams per host, the former results in 20 users while the latter results in 1 user. Compute-optimized instances perform better with 6 request streams. Right image shows the cost of different AWS instances.

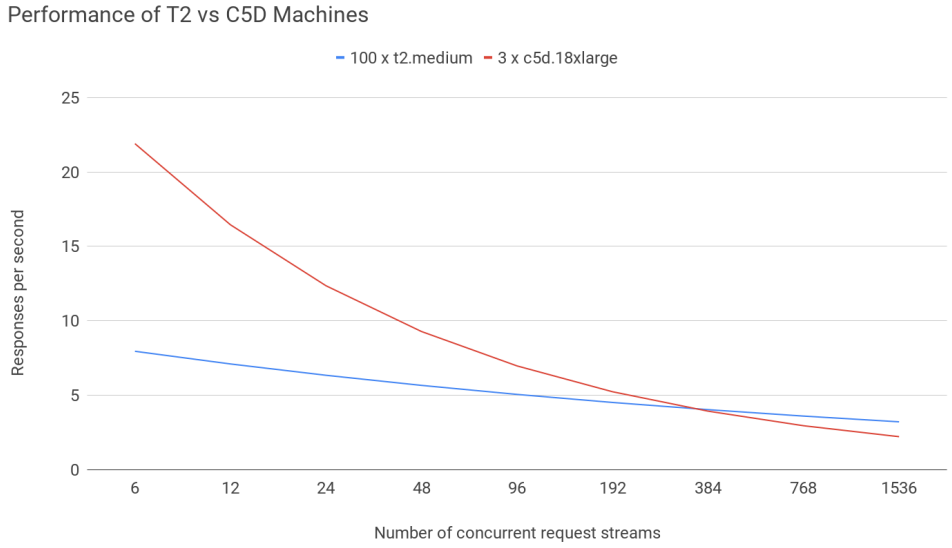

Figure 5.13: Graph showing the estimated point at which T2 instances surpass compute-optimized instances at efficiency.

We ran the "natural monkey testing" scripts in the same configuration as before [74], only that in this work 100 Amazon *t2.micro* instances were acting as testing clients. The datasets used were supernova, turbulence and magnetic (Table 5.3).

Each of the *t2.micro* instances ran a lightweight version of Ubuntu and a headless Chrome browser. Our testing script used SSH to connect to all 100 instances and run our hyperimage test page within the headless browser, and with monkey testing controlling the interactions.

When the monkey testing interactions were done, the JavaScript code within the page sent timing results to a simple Python server that log the results to a file. The timing results included request times, response times, and the resolution of requested images. On average, 3.46% of the images were at viewing resolution (1024^2), and the rest were interaction resolution (256^2).

Figure 5.14 shows average response time for a varying number of testing clients. The blue line shows when the testing clients are deployed on Amazon AWS, and the red line shows when the testing clients are on the local area network as the institutional cluster. The result shows diminishing differences due to network proximity as the number of testing clients increase, which can lead to network congestion regardless of proximity.

Figure 5.15 shows the same test repeated to reveal resource-scalability of our platform. We expanded the deployment from 3 nodes (72 cores, blue curve) to 6 nodes (156 cores, red curve). In both of these two cases, the testing clients were deployed on AWS.

5.3.6 Application Performance

To test the performance of the applications in (Section 5.2) with a single user, we used three C5.9xlarge AWS instances as server.

For hyperimage embedding, we conducted a single monkey-testing user test on a web page with a visualization of a dataset selected randomly (full list in Table 5.3). On average, interaction-resolution renderings (256^2) were rendered at a speed of 9.43 FPS, while viewing-resolution renderings (1024^2) achieved 2.08 FPS. In other words, when a user stops interacting, a high quality rendering is provided in less than 0.5 seconds.

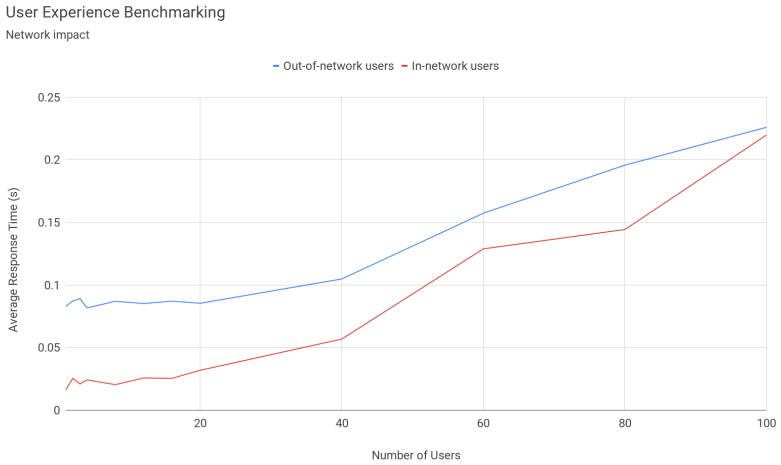

Figure 5.14: Response time for a varying number of users is shown. The slower out-of-network results are from 100 simulated users on AWS, accessing our institutional cloud. The in-network results are from 100 simulated users in our local 1 Gbps network.

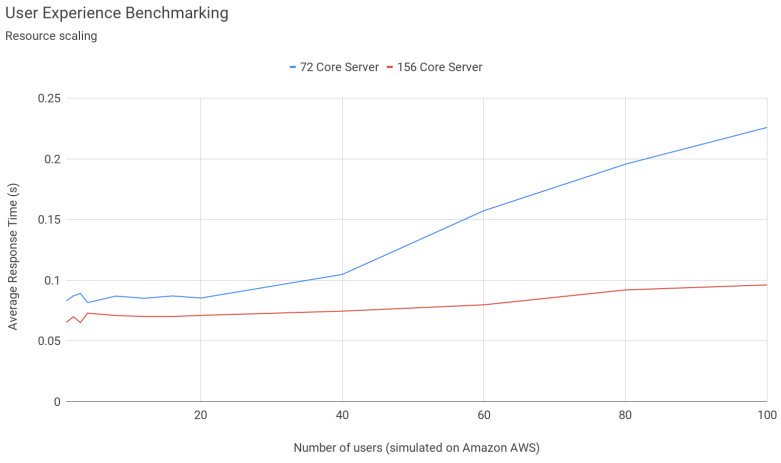

Figure 5.15: Graph showing the scalability of the system. The 72 core cluster is the same as the one used in our previous work [74]. In both cases, we used 100 users (simulated on AWS with monkey-testing).

We also looked at the overhead of including the client-side JavaScript code for Tapestry. On average, pages with Tapestry enabled loaded 1.29 times slower than pages without Tapestry included. For example, a Wikipedia page without hyperimages, loaded in 510ms, while with hyperimages, it took 659ms. Most of this overhead is due to the jQuery library.

Hypervideo performance essentially depends on the server throughput since interpolation has a negligible cost. In our tests, we created three hypervideos for different datasets (5 keyframes each). We chose to generate 50 frames between every two keyframe and therefore 200 frames were rendered for each video. Our video playback speed was set to 30 frames per second; the 200 frame videos were approximately 6 seconds long. The keyframes were chosen at random with different angles, and zoom levels. On average, it took 70.66 seconds to render a full video.

When changing one of the keyframes, on average, the readjustment of a keyframe took 21.15 seconds, since most of the intermediate frames were auto-cached by the cache container (Section 5.1.2). A user can watch the video as it renders albeit at the rendering speed. Any subsequent playback is at 30 frames/second. All hypervideo tests were done using a resolution of 1024^2.

We also tested the speed of our augmented reality application. To view volume renderings of the 7.5GB sized turbine dataset on a HoloLens (Figure 5.7), HoloTapestry can update renderings at a sustained speed of 4.5 FPS. The viewing-resolution in the tests was 512^2 (stereo, without explicit synchronization of left and right eye images), using all 6-nodes of our institutional cluster. While the speed of our prototype implementation is not sufficient for practical use yet, we believe as hardware performance on AR devices improves, better results can be achieved, and HoloTapestry can be utilized in situations where the data is large and cannot be rendered on the device.

5.3.7 Discussion

A visualization that allows real 3D interaction can achieve better user engagement and provide more information than a still image or video can provide. In this respect, Tapestry helps make 3D visualization more accessible. The model used by Tapestry also simplifies

Table 5.5: Summary of the pros and cons between client-side rendering, stateless, and stateful server-side rendering.

Architecture	Pros	Cons
Client only	Does not require external server, existing frameworks	Requires data transfer initial overhead, relies on potentially inadequate local resources, relies on approximated volume rendering techniques via WebGL
Client & Stateful server	Does not rely on client resources, no transfer time, low interaction overhead, dedicated server resources	Requires server-side setup, requires consistent connection to server, does not scale well for many users
Client & Stateless server	Does not rely on client resources, no transfer time, low interaction overhead, multi-user m-to-n mapping	Requires server-side setup, requires consistent connection to server

how a visualization can be hosted as a web service using open-source industry standards, such as Docker, jQuery, and OSPRay.

Comparison to VTK.js. As previously mentioned, client-side systems such as VTK.js have limitations on dataset size and render quality. They also rely on potentially inadequate local resources. Additionally, client-side solutions have significant load time and runtime overheads. For example, a 308 MB supernova volume would need to be pushed to each user. If the user is on a mobile device, this is infeasible. Render performance would be slow on a mobile device as well, leading to an unresponsive web page. Table 5.5 summarizes the pros and cons between client-side rendering versus the stateless remote rendering in Tapestry.

As an example, we informally compared a page with a Tapestry hyperimage with a VTK.js page. Both pages visualized the CHI variable from the jet dataset and a similar interaction pattern was executed across both pages. The Tapestry page took 1.12 seconds to load on average across three tests, while the VTK.js page took 3.21 seconds. The test was executed on a local network, therefore the data download time was not measured. However this should be considered in a real-world scenario. Furthermore, at its peak, the single Tapestry page used 24.4 MB memory while VTK.js used a maximum of 56.8 MB. Additionally, we tested a 308 MB supernova volume with VTK.js. The load time was 13.11 seconds on average with the data residing locally. In contrast, with Tapestry, the NASA supernova page with four supernova volumes took 2.04 seconds to load.

Comparison to ArcticViewer. Perhaps from a client's perspective, one of the most similar works to Tapestry is ArcticViewer [4]. Paired with Cinema [12], ArcticViewer enables the exploration of a dataset through pre-rendered images. While this technique is well suited for in-situ visualization, it generates a large amount of rendered images that may or may not

be used by the end-user. Distributed rendering on-the-fly in Tapestry however, allows users to perform unplanned interactions (e.g. changing transfer functions) without the burden of having generated and stored a large amount of data. Additionally, distributed tiling allows enables users to render large resolution images mid-exploration.

Chapter 6

Delivering Graph Visualizations in Real-time

Graph is the universal model for representing relationship among entities. With the explosion of big data in recent years, the size of graphs have also exponentially increased. The challenge to view and work with such graphs takes three different dimensions. First, the shear size of the graphs necessitate more computational resources and efficient methods in storing, visualizing and communicating them. Second, due to the prevalence of public data, the potential audience has also increased. While gaining insights from a graph may have been an offline practice in the past, it now calls for efficient web access, so that many people can benefit from the insights at the same time. Third, when the size of a graph increases, statically rendering and viewing it is no longer a viable action. New interactions need to be thought of and implemented.

In this chapter, I introduce a Fabric-based architecture that visualizes large graphs for multiple audiences on the web at the same time while also enabling live interactions. The system, codenamed KnitGraph is comprised of an active visualization kernel responsible for rendering a layout of a graph, a client-side that distributes rendering request responses among a tiling system. KnitGraph utilizes the idea that OpenGL's Shader language (GLSL) is external to the typical graphics pipeline, can be manipulated externally and then compiled on the fly. Therefore, the client-side of KnitGraph alter the server-side graph rendering on the fly by sending GLSL code for compilation based on user interactivity.

Note that the definition of large graphs in the literature is subjective and depends on time. KnitGraph considers a large graph as a graph with above one million vertices and edges.

6.1 Architecture Design

6.1.1 Minimal Graph Rendering Kernel

Similar to the OSPRay-based visualization kernel used in Tapestry, KnitGraph also employs a micro-kernel that is stateless and other than having a few graph datasets loaded in memory, does not maintain the state of interaction that the user has with the system. Instead, the state of the visualization is sent from the client-side to the server with each rendering request.

As a result of being stateless, KnitGraph servers are distributed in Docker containers within a load-balanced Docker Swarm. Every KnitGraph server renders a portion of the full graph and correctly positions the graphics camera based on the incoming request.

In Tapestry, the kernel used an external rendering engine for scientific visualization (OSPRay), and therefore, the format of the requests was customized for that engine and its usecases. However, in KnitGraph, we use standard OpenGL, a well established graphics programming programming interface [87]. This design choice allows us to have a more standard request structure. Each rendering request from the client includes two OpenGL shader snippets. These snippets are codes written in GLSL, edited on the client-side based on user interactions and compiled on the server on the fly. GLSL code is platform agnostic and even runs in WebGL if needed. Moreover, GLSL is compiled based on the graphics hardware available on a machine with compilers that are optimized by the hardware vendor. Sending GLSL code as a request allows for a lot of flexibility in how the rendering can change.

Figure 6.1 shows an overview of a KnitGraph server.

6.1.2 Client-Side Interaction

The client-side of KnitGraph uses Leaflet's tiling system [10]. Leaflet is a commonly used Javascript library for creating tile-based map applications. Every tile in has an `x, y, z`

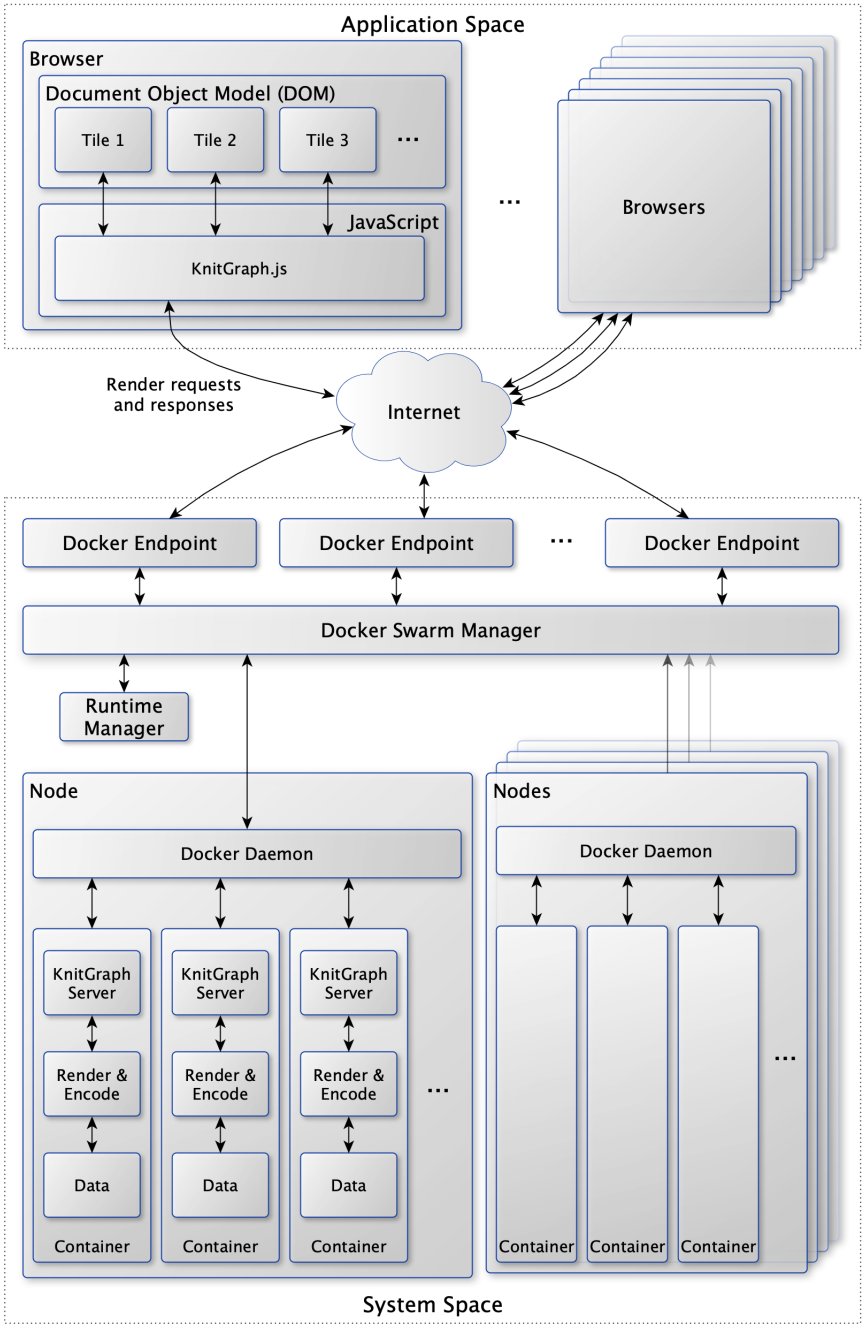

Figure 6.1: An overview of KnitGraph's architecure. Note the similarity to Tapestry's architecture. The visualization kernel in this case is implemented in OpenGL and performs the graph rendering. The client-side is comprised of tiles that invoke rendering requests to the server.

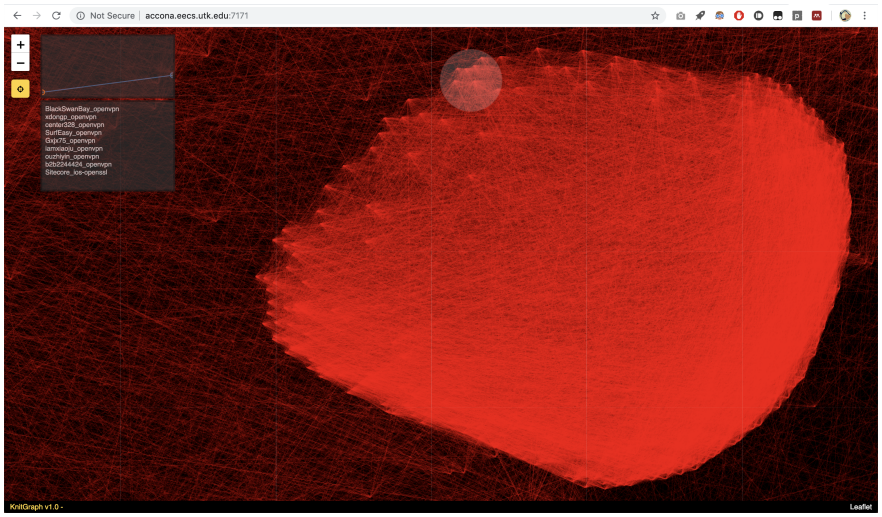

Figure 6.2: A view of KnitGraph within a browser.

coordinate. The `x`, and `y` coordinates denote the 2D position of a tile with the origin set to the top-left side of the screen. The `z` coordinate denotes the zoom level. In map-libraries `z` is typically an integer from 0 to 16.

Figure 6.2 shows a view of KnitGraph within the browser. The image shows a zoomed in view of a large software dependency graph. A series of vertices are selected with the cursor and their names attribute is displayed in a box on top-left side of the screen. A transfer function editor provides filtering capabilities and is described in the next section.

Opacity Transfer Function

One of the main challenges in rendering a large graph is how edges overlap one another until the visualization resembles a blob with no particular feature visible. As a first remedy, graph rendering applications decrease the opacity of the edges and use alpha-blending when drawing edges on top of one another. While this solution helps significantly in displaying graph features, it does not completely solve the problem. In dense enough graphs, the edges ultimately stack up to create completely opaque areas.

Another solution in the literature is edge-bundling in which edges that take similar routes are bundled together to clear the way for other structures in the graph. Many edge-bundling algorithms exist in the literature [110], however, they come with significant computational overhead making this approach infeasible for large graphs.

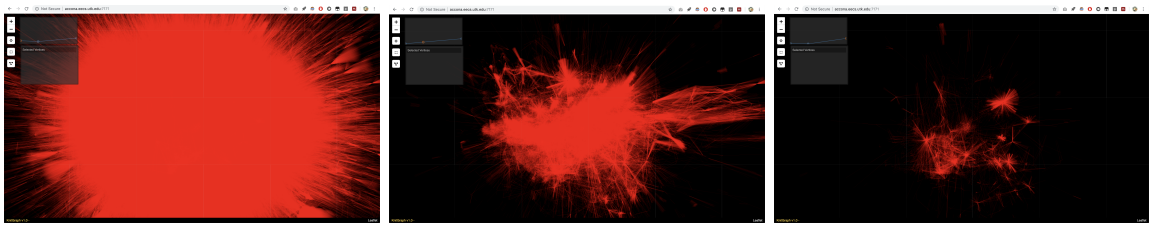

Figure 6.3: Three images show how changing the opacity transfer function (TF) helps structures appear in the graph. With a uniform transfer function, the graph becomes extremely cluttered (left image), while the middle and right images show different structures with the help of a TF.

A third approach is providing a filtering mechanism to the user so that they can explore the different features in the graph based on their attributes of interest. Filtering perfectly matches KnitGraph's architectural design for two reasons. First, the choice of using OpenGL shaders as a request format gives full control to the client in displaying and emphasizing edges based on data attributes. KnitGraph uses the GLSL fragment shader for this purpose, and can therefore alter the color and opacity of each edge based on a data attribute. Second, the scalable server-side means that a new rendering of the entire graph can be served in realtime, allowing the user to freely explore different viewing parameters.

Filtering in KnitGraph's client-side is done through an Opacity Transfer Function (OTF). An OTF is a function mapping a data attribute's value to an opacity value. This allows users to hide or emphasize edges based on an attribute of interest. While OTFs are widely used in scientific 3D rendering applications, to our knowledge, this is the first time that they are used as a filtering mechanism for large graph rendering.

As done traditionally in visualization software, the `x` axis in a transfer function editor represents the range of a selected numerical data attribute. The `y` axis represents opacity from 0 to 1.

Figure 6.3 shows the transfer function editor being used in KnitGraph. The left image in the figure shows all edges with uniform opacity. The other two images in the figure show internal structures in the graph using a transfer function that highlights edges with a high attribute value and dims those with a low value. The attribute used in this example was the number of maintainers in software projects.

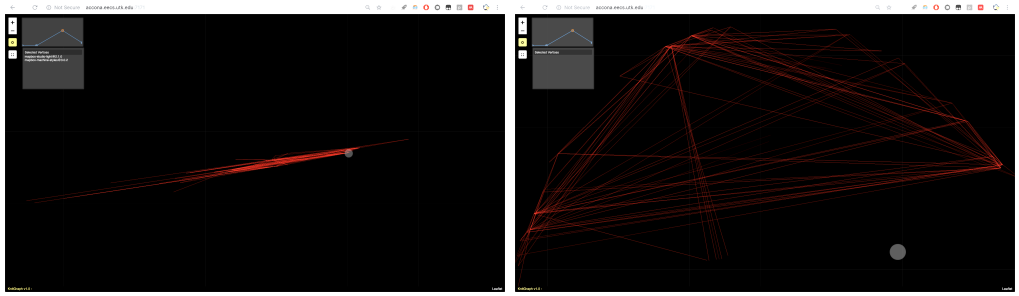

Figure 6.4: An example of using the fisheye tool in KnitGraph is shown. The left image shows a dense and cluttered area in a graph. The right image is the result of using the fisheye tool on this area. The edges and vertices are expanded from the center.

Layout Manipulation

In addition to a fragment shader, OpenGL's programmable pipeline includes another type of shader called a vertex shader. Vertex shaders are responsible for manipulating the physical position of vertices in a rendering. While KnitGraph follows the idea of calculating a graph layout once and then distributing its rendering without layout changes that are often costly, the vertex shader can be used to make small adjustments to the layout on the fly.

As a prototype for this feature, KnitGraph includes a fish-eye view that expands dense areas when clicked on, so that the internal structures can be better viewed. The adjustment is computationally simple. When a user clicks on a region in the graph with the cursor, the client side code alters the vertex-shader such that all vertex positions within a chosen radius of the cursor are moved further out based on their proximity to the cursor position. The code is then sent to the server-side for compilation and rendering.

Figure 6.4 shows the effect of the fisheye tool on a dense region (left). The vertices and edges are expanded and can be better viewed (right).

Vertex Selection

A graph cannot be explored if the vertices are unknown. A vertex-selection method is therefore an absolute necessity. KnitGraph includes a vertex-selection mode that is toggled using the left bullseye icon. When active, users can use a cursor to select graph vertices. The "name" attribute for the selected vertices is then fetched from a random server container and displayed in the second top-left box.

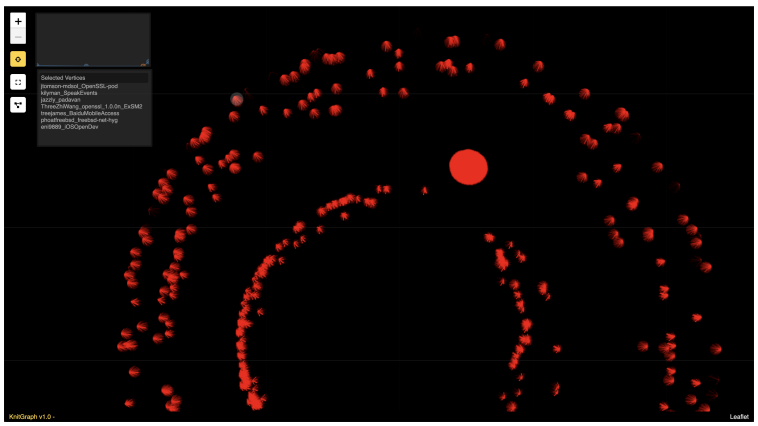

Figure 6.5: A small cluster of vertices is selected towards the top-left side of the graph using the vertex-selection tool. The name of the selected vertices is shown in the left box.

Because of the large size of the graph, and proportional size of the vertices, it may be difficult for a user to exactly select a vertex. To alleviate this, the vertex-selection mode has a cursor whose radius can be adjusted by scrolling up or down. This interaction mechanism is similar to that of interactions in the Blender application [79]. The radius of the cursor is also sent to the server-side when requesting vertex names.

When a user clicks on a region in the graph using the selection tool, the location of the click event as well as the radius of the selection cursor are sent to the server-side. The location of the click event includes the tile's coordinates, the zoom level, and the offset of the click event within the tile. One of the graph kernels on the server-side receives the request and using the location, and maps the event location from the image space to a region in the data space based on the range of the data points. A function then looks through all vertex points and collects vertices that fall within the calculated region. The attributes for the found vertices are then sent back to the client.

Figure 6.5 shows a view of a graph with many small clusters. The selection cursor can be seen towards the top-left of the graph. Having selected one of the clusters, the names of the vertices is shown in the left *selected-vertices* box.

6.2 Applications

KnitGraph aims at increasing the supported number of users, while still providing a degree of interactivity with a graph. Therefore in this section, we have picked two graph datasets that can be beneficial to a large number of users throughout the Internet. Both of the datasets are revolve around software packages and pertain to the vast community of developers. In the following sections, we introduce each dataset, and show how KnitGraph has been used in exploring various features within each of them.

6.2.1 The Heartbleed Network

The open source community has immensely grown in size in recent years and the network of various open source projects on the web, holds many types of relationships among entities such as projects, developers, code snippets, and languages to name a few. Each of these relationship networks pertains to different aspects of the community and understanding them can have great impact [112].

One of the many aspects in open source networks is security and the propagation of vulnerabilities throughout the ecosystem. If developers find a vulnerability in a software package, it is time-consuming but not difficult to find projects that cite the vulnerable project as a dependency. Many software packaging systems already track dependencies. However, finding similarly vulnerable packages due to code-reuse and copying is a much more difficult endeavor.

In this section, we look at a portion of Github that has been either known to have been directly vulnerable to the Heartbleed bug [35] or is indirectly vulnerable due to have copies of vulnerable code. We obtained the data for this network from our collaboration with the software supply chain group at the University of Tennessee [63].

The graph dataset includes 7478 projects represented as vertices and 25 million edges. An edge between two projects indicates that they share one or more files that are identical. Common files such as copyright notices and licenses are not included.

Figure 6.6 shows a view of the heartbleed network using the ARF layout [45]. The transfer function is set such that repositories that were originally reported vulnerable are shown. The graph shows many groups of repositories in clusters that share content. The large cluster towards the middle of the graph consists of copies of the OpenSSL and Wireshark projects.

Realizing that the large cluster includes copies of Wireshark using hte vertex selection tool, the user can further filter the view. The user selects those vertices using KnitGraph's neighbor selection tool. The result is shown in Figure 6.7. The graph now only shows the selected Wireshark copies and their immediate neighbors. In the image, a group of these neighbors are selected and their names are shown in the vertex attribute box.

As another example, Figure 6.2 shows a zoomed in version of the same dataset. In this view, a cluster of OpenVPN projects are shown. Among the selected vertices, an outlier is visible (`ios-openssl`) that shares content with the other projects and could potentially be vulnerable.

6.2.2 NPM Dependency Network

With the introduction of modern Javascript, NodeJS and the NPM package manager, the Javascript community has embraced the idea of reusing code by depending on micro-projects that do one thing and one thing well. However, this has not been without issues. In 2016, thousands of projects on NPM broke because a package named *leftpad* that only included 11 trivial lines of code was taken down by the developer [7].

We believe, it is therefore important for developers to know exactly on which projects they are depending down the dependency tree and know the properties of such packages. For example: Do they include any known vulnerabilities? Are they only reliant on a small number of maintainers? Are they prone to become obsolete due to lack of updates?

While these questions can be easily answered for first-level dependencies, they become very difficult to answer for deeper levels as well as for the whole NPM community as a whole.

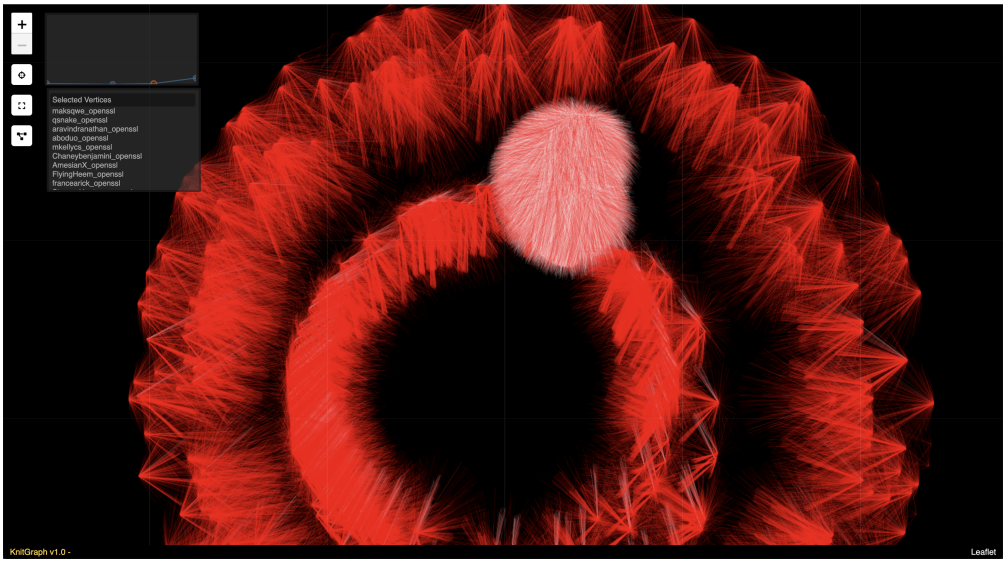

Figure 6.6: A view of the heartbleed network is shown. The transfer function is set so that only those vertices that were originally found vulnerable are shown. The large cluster towards to the middle represents copies of the OpenSSL and Wireshark projects.

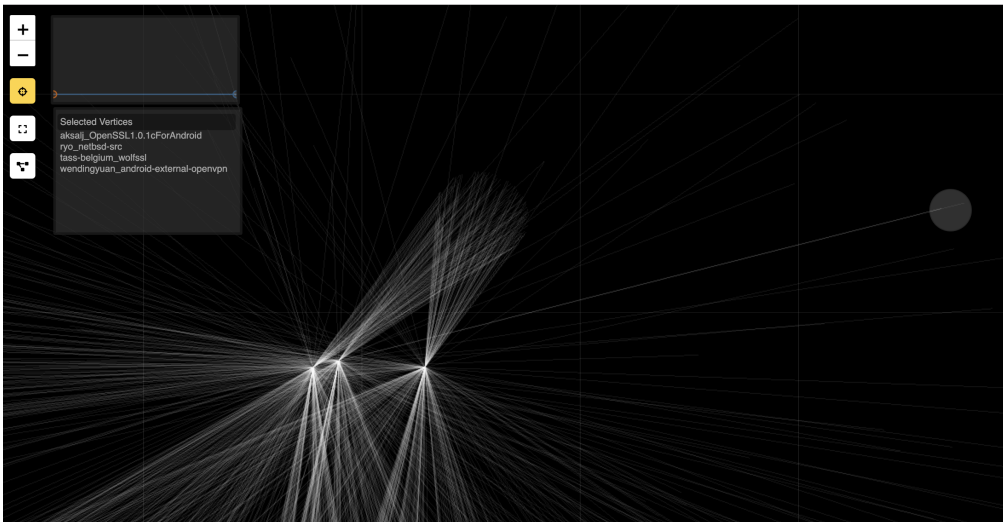

Figure 6.7: A group of Wireshark-related projects are shown using the neighbor-selection tool. A group of neighbors are further selected (towards the right). Their project names are shown in the vertex-attribute box.

In this section, we look at the NPM dependency graph. The graph includes approximately 1.1 million packages represented as vertices, and about 6.1 million edges showing dependencies among the packages. Each vertex also includes properties such as its name, number of maintainers, last update time, and whether it has been flagged with a known vulnerability.

Figure 6.8 shows a zoomed-in view of the NPM graph, in which the transfer function is set such that projects with the highest number of maintainers are shown. The vertex selection tool has selected a few nodes indicating that the `cli` project is highly maintained.

6.3 Results and Discussion

We tested KnitGraph's performance on our institutional cloud described in Section 3.4.

Our tests considered the NPM dataset. On average, KnitGraph showed an average response time of 1.47 seconds for a tile request. This was in the case of asking for the entirety of the graph at a zoomed out view. In a more detailed view, where the rendered images were less dense, an average response time of 1.01 seconds were observed per tile.

We also measured the performance of vertex attribute queries done using the vertex selection tool. On average, it took 0.5 seconds to retrieve the attributes of less than 200 selected vertices.

The main benefit of having a stateless architecture is being able to horizontally scale the server and support more users. Therefore, we tested the effect of multiple containers in the face of 100 concurrent requests. Figure 6.9 shows how increasing the number of containers helps lower response times. On average, 20 containers showed a response time of 3.8 seconds per tile. Note that this is the time it took for a tile request to be answered and its response rendered on the client's screen. It does not mean that users have to wait 3.8 seconds between each new tile.

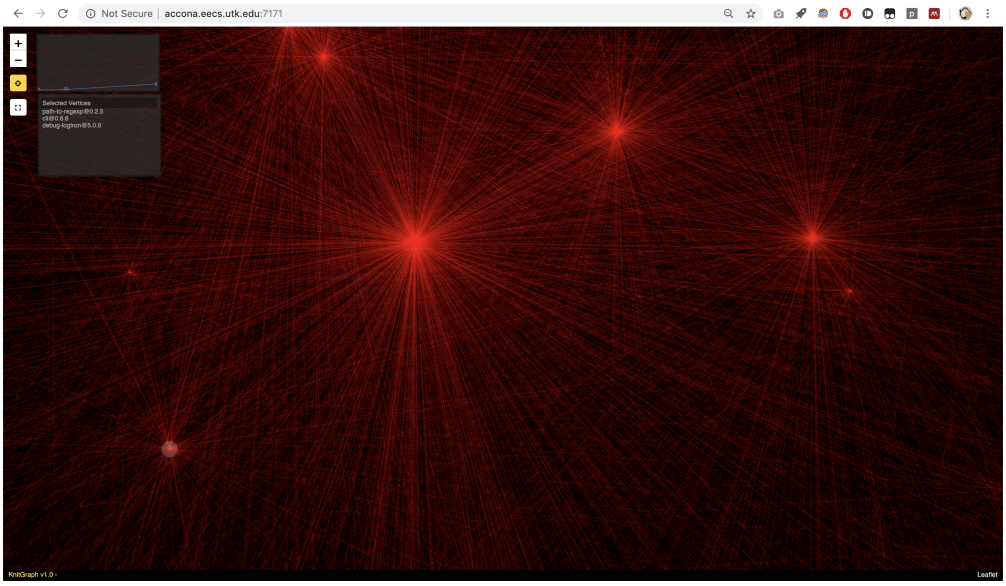

Figure 6.8: A zoomed-in view of the NPM graph is shown. Projects with a large number of members have been set as visible by the transfer function.

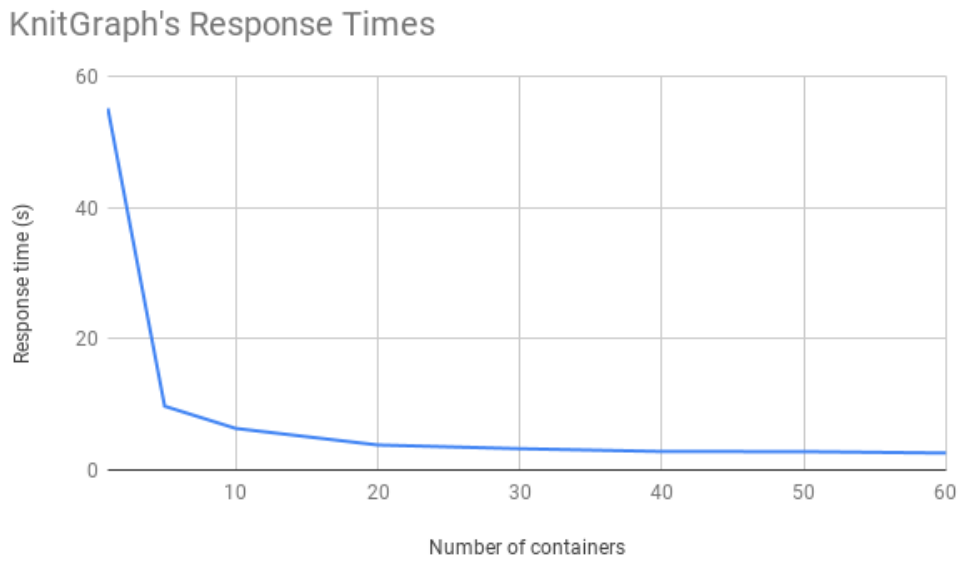

Figure 6.9: KnitGraph's response times are shown for 100 concurrent requests while varying the number of containers. Similar to Tapestry, more containers help handle more requests. With 20 containers, tiles took an average 3.8 seconds to appear.

Chapter 7

Conclusion and Future Works

Many applications that were previously offline have embraced the web ecosystem for better availability and accessibility. While the movement of data analysis and visualization to the web has been tackled before, large data-oriented applications have faced many obstacles in adopting the web ecosystem.

I believe this has mostly been due to the monolithic and highly-coupled systems that served such applications. Many such applications have been based on remote-visualization architectures that were originally built for HPC systems and not originally built with the web in mind.

In this dissertation, I presented a novel perspective on the delivery of interactive visualization to the web ecosystem. At a high level, my approach, named Fabric, separates application logic and interaction from rendering and data management. This allows application logic to reside closer to where interaction is initiated from (the client-side). This also means that the application logic is able to stay oblivious of the servers that compute data products and perform rendering, and as a result allows the server-side to be dynamically switched from one server to another. On the server-side, the mentioned separation of concern allows for statelessness and hence, horizontal scalability in response to a varying workload. In this chapter I summarize the work in this dissertation, and discuss potential future directions.

7.1 Delivering General Visualizations

My work on capturing interactive visualization, and its accompanying prototype named Loom, has shown that the behavior of many interactive applications can be captured as a state machine along with a set of finite images that represent each application state. On one hand, this has resulted in being able to detach a captured application from its original source code, computational needs, and data source. On the other hand, this has lays the basis for the separation of the client-side and server-side in complex interactive applications.

Loom's detachment from the source code and data has enabled the archival of interactive visualization. In addition, we showed how Loom can help with reproducibility of scientific visualizations, provided better sharability.

The ideas behind Loom can be extended in many ways. One potential direction is combining the passive pre-rendering in Loom with active rendering in order to increase the interactive fidelity of applications on-demand.

Considering that specifying UI components is the most time-consuming step in Loom's process, another direction is using machine learning for automatic interaction with applications in the capture phase.

7.2 Delivering Volumetric Scientific Visualization

Replacing the passive pre-rendering kernel in Loom with live rendering enabled scientific visualization of large data with high-fidelity become possible for a wide-audience. The resulting system called Tapestry, showed that scientific visualization can be hosted on the cloud as a micro-service. The cloud-based solution abstracted high-end visualization performance to dollars per frame. Additionally, we showed how Tapestry can serve various devices such as regular desktops, mobile phones, AR/VR devices, and large powerwalls.

Utilizing the on-demand rendering kernel in Tapestry, we further showed how the system can help in building scientific movies and sharing them as small textual snippets that can be modified and rendered on the fly. In recent advancements, Tapestry has been used to

render an entire 3D scene of the Moana movie on the web [84]. The complete scene has an approximate size of 200GB and was rendered on the cloud with Amazon's AWS.

For future directions, I believe Tapestry can be used as a basis for taking high-performance movie rendering to the cloud and allow small teams to build animations and only pay for the resources at render time. The shareability aspect of Tapestry's videos can be very useful in teams that would like to collaborate.

7.3 Delivering Graph Visualization

In Chapter 6, we showed that extremely large graphs can be rendered and interacted with on the web. KnitGraph's approach is general in that it makes no assumptions about the underlying graph. It is therefore suitable for initial explorations of large graphs. I foresee KnitGraph being used in scenarios where a large audience can benefit from understanding the underlying data. Examples are project dependency graphs, and large social network data that belongs to the large number of people that created it.

The core contribution of KnitGraph however, is the type of visualization request it utilizes. Unlike the other Fabric kernels, KnitGraph's requests are complete code snippets in GLSL. Results have shown that GLSL functions can be written, manipulated, and sent across the web for remote rendering, in each request, while still maintaining an interactive speed. This provides a great amount of flexibility to end users in a stateless architecture. It also serves as a good example of the kind of flexibility that the OpenGL programmable pipeline provides. Additionally, to our knowledge, KnitGraph is the first application to use shaders for filtering and interacting with graphs.

The current implementation of KnitGraph serves as a prototype and can benefit from many new functionalities such as cluster-coloring, and applying more complicated local layouts on clusters. At a higher level, KnitGraph's approach can be used for non-graph visualizations such as interactive arc diagrams, hive plots, and parallel coordinates.

Bibliography

[1] (2016). *NVIDIA IndeX*. 12

[2] (2018 (accessed November 23, 2018)). National Academies of Sciences, Engineering, and Medicine: Reproducibility and Replicability in Science. 10

[3] (2018 (accessed October 14, 2018)). Maximum Number of Open Connections Per Browser. 57

[4] (2018 (accessed October 14, 2018)). Paraview ArcticViewer. 6, 9, 77

[5] (2018 (accessed October 14, 2018)). Software Container Platform - Docker: https://www.docker.com/. 9, 16, 46, 53

[6] (2019 (accessed January 21, 2019)). Senseable City Lab - MIT. 39

[7] Abdalkareem, R., Nourry, O., Wehaibi, S., Mujahid, S., and Shihab, E. (2017). Why do developers use trivial packages? an empirical case study on npm. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 385–395. ACM. 87

[8] Abello, J., Van Ham, F., and Krishnan, N. (2006). Ask-graphview: A large scale graph visualization system. *IEEE transactions on visualization and computer graphics*, 12(5):669–676. 14

[9] Adobe (2019 (accessed Jan 16, 2019)). Flash & The Future of Interactive Content. https://theblog.adobe.com/adobe-flash-update/. 10, 44

[10] Agafonkin, V. (2019 (accessed April 10, 2019)). An open-source JavaScript library for mobile-friendly interactive maps. https://leafletjs.com/. 80

[11] Ahrens, J., Geveci, B., and Law, C. (2005). Paraview: An end-user tool for large data visualization. *The visualization handbook*, 717. 11

[12] Ahrens, J., Jourdain, S., O'Leary, P., Patchett, J., Rogers, D. H., and Petersen, M. (2014). An image-based approach to extreme scale in situ visualization and analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 424–434. IEEE Press. 6, 7, 77

[13] Amazon (2018 (accessed October 14, 2018)). Amazon AWS Instance Types. 71

[14] Arora, S. K., Li, Y., Youtie, J., and Shapira, P. (2016). Using the wayback machine to mine websites in the social sciences: a methodological resource. *Journal of the Association for Information Science and Technology*, 67(8):1904–1915. 10

[15] Ayachit, U. (2015). *The ParaView Guide: A Parallel Visualization Application*. Kitware, Inc. 7, 9, 12

[16] Bastian, M., Heymann, S., and Jacomy, M. (2009). Gephi: An open source software for exploring and manipulating networks. 14

[17] Bavoil, L., Callahan, S. P., Crossno, P. J., Freire, J., Scheidegger, C. E., Silva, C. T., and Vo, H. T. (2005). Vistrails: Enabling interactive multiple-view visualizations. In *Proc. of IEEE Visualization*, pages 135–142. IEEE. 10, 11, 12

[18] Beck, F., Burch, M., Diehl, S., and Weiskopf, D. (2017). A taxonomy and survey of dynamic graph visualization. In *Computer Graphics Forum*, volume 36, pages 133–159. Wiley Online Library. 13

[19] Bikakis, N., Liagouris, J., Krommyda, M., Papastefanatos, G., and Sellis, T. (2016). Graphvizdb: A scalable platform for interactive large graph visualization. In *2016 IEEE 32nd international conference on data engineering (ICDE)*, pages 1342–1345. IEEE. 14

[20] Bilheux, H., Crawford, K., Walker, L., Voisin, S., Kang, M., Harvey, M., Bailey, B., Phillips, M., Bilheux, J., Berry, K., et al. (2013). Neutron imaging at the oak ridge national laboratory: present and future capabilities. In *7th International Topical Meeting on Neutron Radiography*. Phys. Proc. 65, 69

[21] Blondin, J. M. and Mezzacappa, A. (2007). Pulsar spins from an instability in the accretion shock of supernovae. *Nature*, 445(7123):58–60. 51, 69

[22] Bostock, M., Ogievetsky, V., and Heer, J. (2011). D3 data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309. 6, 7

[23] Brehmer, M. and Munzner, T. (2013). A multi-level typology of abstract visualization tasks. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2376–2385. 43

[24] Cabello, R. et al. (2010). Three. js. *URL: https://github. com/mrdoob/three. js*. 6

[25] Camisetty, A., Chandurkar, C., Sun, M., and Koop, D. (2018). Enhancing web-based analytics applications through provenance. *IEEE transactions on visualization and computer graphics*. 10

[26] Camisetty, A., Chandurkar, C., Sun, M., and Koop, D. (2019). Enhancing web-based analytics applications through provenance. *IEEE transactions on visualization and computer graphics*, 25(1):131–141. 11

[27] Chau, D. H., Kittur, A., Hong, J. I., and Faloutsos, C. (2011). Apolo: making sense of large network data by combining rich user interaction and machine learning. In *Proceedings of the SIGCHI conference on human factors in computing systems*, pages 167–176. ACM. 14

[28] Chen, H.-T., Wei, L.-Y., and Chang, C.-F. (2011). Nonlinear revision control for images. In *ACM Transactions on Graphics (TOG)*, volume 30, page 105. ACM. 11

[29] Childs, H., Brugger, E., Whitlock, B., Meredith, J., Ahern, S., Pugmire, D., Biagas, K., Miller, M., Harrison, C., Weber, G. H., Krishnan, H., Fogal, T., Sanderson, A., Garth, C., Bethel, E. W., Camp, D., Rübel, O., Durant, M., Favre, J. M., and Navrátil, P. (2012). VisIt: An end-user tool for visualizing and analyzing very large data. In *High Performance Visualization–Enabling Extreme-Scale Scientific Insight*, pages 357–372. 7, 9, 12

[30] Christophe Viau (2019 (accessed April 4, 2019)). List of d3 examples. https://christopheviau.com/d3list/. 6

[31] Cui, W., Zhou, H., Qu, H., Wong, P. C., and Li, X. (2008). Geometry-based edge clustering for graph visualization. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1277–1284. 14

[32] Ding, J., Huang, J., Beck, M., Liu, S., Moore, T., and Soltesz, S. (2003). Remote visualization by browsing image-based databases with logistical networking. In *SC03: Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 34:1–34:11. 6

[33] Donzis, D., Yeung, P., and Pekurovsky, D. (2008). Turbulence simulations on $O(10^4)$ processors. In *TeraGrid 2008 Conference*. 69

[34] Dromey, R. G. (2006). Formalizing the transition from requirements to design. In *Mathematical frameworks for component software: Models for analysis and synthesis*, pages 173–205. World Scientific. 9

[35] Durumeric, Z., Li, F., Kasten, J., Amann, J., Beekman, J., Payer, M., Weaver, N., Adrian, D., Paxson, V., Bailey, M., et al. (2014). The matter of heartbleed. In *Proceedings of the 2014 conference on internet measurement conference*, pages 475–488. ACM. 86

[36] Ellson, J., Gansner, E. R., Koutsofios, E., North, S. C., and Woodhull, G. (2004). Graphviz and dynagraphstatic and dynamic graph drawing tools. In *Graph drawing software*, pages 127–148. Springer. 14

[37] Evans, A., Romeo, M., Bahrehmand, A., Agenjo, J., and Balt, J. (June, 2014). 3d graphics on the web: A survey. *Computers and Graphics*, 41:43 – 61. 48

[38] Fanelli, D. (2018). Opinion: Is science really facing a reproducibility crisis, and do we need it to? *Proceedings of the National Academy of Sciences*, 115(11):2628–2631. 10

[39] Fang, D., Keezer, M., Williams, J., Kulkarni, K., Pienta, R., and Chau, D. H. (2017). Carina: Interactive million-node graph visualization using web browser technologies. In *Proceedings of the 26th International Conference on World Wide Web Companion*, pages 775–776. International World Wide Web Conferences Steering Committee. 14

[40] Frey, J., Tannenbaum, T., Livny, M., Foster, I., and Tuecke, S. (2001). Condor-g: A computation management agent for multi-institutional grids. In *High Performance Distributed Computing, 2001. Proceedings. 10th IEEE International Symposium on*, pages 55–63. IEEE. 13

[41] Fruchterman, T. M. and Reingold, E. M. (1991). Graph drawing by force-directed placement. *Software: Practice and experience*, 21(11):1129–1164. 13

[42] Gajdoš, P., Ježowicz, T., Uher, V., and Dohnálek, P. (2016). A parallel fruchterman–reingold algorithm optimized for fast visualization of large graphs and swarms of data. *Swarm and Evolutionary Computation*, 26:56–63. 13

[43] Gansner, E. R., Hu, Y., North, S., and Scheidegger, C. (2011). Multilevel agglomerative edge bundling for visualizing large graphs. In *2011 IEEE Pacific Visualization Symposium*, pages 187–194. IEEE. 14

[44] Gao, J., Huang, J., Johnson, C. R., and Atchley, S. (2005). Distributed data management for large volume visualization. In *Proc. of IEEE Visualization*, pages 183–189. IEEE. 13

[45] Geipel, M. M. (2007). Self-organization applied to dynamic network layout. *International Journal of Modern Physics C*, 18(10):1537–1549. 13, 87

[46] Glatter, M., Huang, J., Ahern, S., Daniel, J., and Lu, A. (2008). Visualizing temporal patterns in large multivariate data using modified globbing. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1467 – 1474. 12

[47] Grossman, T., Matejka, J., and Fitzmaurice, G. (2010). Chronicle: capture, exploration, and playback of document workflow histories. In *Proceedings of the 23nd annual ACM symposium on User interface software and technology*, pages 143–152. ACM. 11

[48] Guo, F., Li, H., Daughton, W., and Liu, Y. H. (2014). Formation of hard power laws in the energetic particle spectra resulting from relativistic magnetic reconnection. *Physical Review Letters*, 113(15):1–5. 69

[49] Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274. 9

[50] Heer, J., Mackinlay, J., Stolte, C., and Agrawala, M. (2008). Graphical histories for visualization: Supporting analysis, communication, and evaluation. *IEEE transactions on visualization and computer graphics*, 14(6). 11

[51] Hitz, M. and Montazeri, B. (1995). Measuring coupling and cohesion in object-oriented systems. 15

[52] Hu, Y. (2005). Efficient, high-quality force-directed graph drawing. *Mathematica Journal*, 10(1):37–71. 13

[53] Jomier, J., Jourdain, S., Ayachit, U., and Marion, C. (2011). Remote visualization of large datasets with midas and paraviewweb. In *Proceedings of the 16th International Conference on 3D Web Technology*, Web3D '11, pages 147–150, New York, NY, USA. ACM. 6, 7

[54] Jourdain, S., Ayachit, U., and Geveci, B. (2010). Paraviewweb, a web framework for 3d visualization and data processing. In *IADIS Intl Conf. on Web Virtual Reality and Three-Dimensional Worlds*, volume 7, page 1. 6, 7

[55] Kanuparthy, P., Matthews, W., and Dovrolis, C. (2012). DNS-based ingress load balancing: An experimental evaluation. *CoRR*, abs/1205.0820. 53, 57

[56] Kendall, W., Huang, J., Peterka, T., Latham, R., and Ross, R. (2011). Toward a general i/o layer for parallel-visualization applications. *IEEE Computer Graphics and Applications*, 31(6):6–10. 13

[57] Kitware (2017 (accessed June 10, 2017)). Vtk.js. https://github.com/Kitware/vtk-js. 6, 7

[58] Li, X. and Shen, H.-W. (2002). Time-critical multi-resolution volume rendering using 3d texture mapping hardware. In *Proc. IEEE/ACM Symp. on Volume Visualization and Graphics*, pages 29–36. IEEE. 12

[59] Lim, C.-U., Baumgarten, R., and Colton, S. (2010). Evolving behaviour trees for the commercial game defcon. In *European Conference on the Applications of Evolutionary Computation*, pages 100–110. Springer. 9

[60] Mathieu Stefani (2017 (accessed June 16, 2017)). Pistache http server. http://pistache.io. 55

[61] Meißner, M., Huang, J., Bartz, D., Mueller, K., and Crawfis, R. (2000a). A practical evaluation of popular volume rendering algorithms. In *Proc. of IEEE Symp. on Volume Visualization*, pages 81–90. ACM. 12

[62] Meißner, M., Huang, J., Bartz, D., Mueller, K., and Crawfis, R. (2000b). A practical evaluation of popular volume rendering algorithms. In *Proceedings of the 2000 IEEE symposium on Volume visualization*, pages 81–90. ACM. 68

[63] Mockus, Audris and Zaretzki Russell and Bichescu, Bogdan and Bradley, Randy (2019 (accessed April 6, 2019)). Open source supply chains and avoidance of risk: An evidence based approach to improve floss supply chains. https://www.nsf.gov/awardsearch/showAward?AWD_ID=1633437&HistoricalAwards=false. 86

[64] Moreau, L. (2010). The foundations for provenance on the web. *Foundations and Trends in Web Science*, 2(2–3):99–241. 11

[65] Munzner, T. (1998). Drawing large graphs with h3viewer and site manager. In *International Symposium on Graph Drawing*, pages 384–393. Springer. 13

[66] Nachmanson, L., Prutkin, R., Lee, B., Riche, N. H., Holroyd, A. E., and Chen, X. (2015). Graphmaps: Browsing large graphs as interactive maps. In *International Symposium on Graph Drawing*, pages 3–15. Springer. 14

[67] Nguyen, B. P., Tay, W.-L., Chui, C.-K., and Ong, S.-H. (2012). A clustering-based system to automate transfer function design for medical image visualization. *The Visual Computer*, 28(2):181–191. 12

[68] NYTimes (2018 (accessed March 28, 2018)). The New York Times visualization on the 2014 World Cup. https://www.nytimes.com/interactive/2014/06/20/sports/worldcup/how-world-cup-players-are-connected.html. 37

[69] Pahl, C. (2015). Containerization and the PaaS cloud. *IEEE Cloud Computing*, 2(3):24–31. 52

[70] Pascucci, V., Scorzelli, G., Summa, B., Bremer, P.-T., Gyulassy, A., Christensen, C., Philip, S., and Kumar, S. (2012). The visus visualization framework. *EW Bethel, HC (LBNL), and CH (UofU), editors, High Performance Visualization: Enabling Extreme-Scale Scientific Insight, Chapman and Hall/CRC Computational Science*. 6, 7

[71] Peixoto, T. P. (2014). The graph-tool python library. *figshare*. 13

[72] Piwowar, H. A., Vision, T. J., and Whitlock, M. C. (2011). Data archiving is a good investment. *Nature*, 473(7347):285. 10

[73] Raji, M., Hota, A., Hobson, T., and Huang, J. (2018). Scientific visualization as a microservice. *IEEE transactions on visualization and computer graphics*. 1, 8, 20

[74] Raji, M., Hota, A., and Huang, J. (2017a). Scalable web-embedded volume rendering. In *2017 IEEE 7th Symposium on Large Data Analysis and Visualization (LDAV)*, pages 45–54. 8, 16, 54, 56, 68, 74, 75

[75] Raji, M., Hota, A., and Huang, J. (2017b). Scalable web-embedded volume rendering. In *2017 IEEE 7th Symposium on Large Data Analysis and Visualization (LDAV)*, pages 45–54. IEEE. 12

[76] Raji, M., Hota, A., Sisneros, R., Messmer, P., and Huang, J. (2017c). Photo-Guided Exploration of Volume Data Features. In Telea, A. and Bennett, J., editors, *Eurographics Symposium on Parallel Graphics and Visualization*. The Eurographics Association. 12

[77] Risk, K. (2019 (accessed January 23, 2019). Lightweight fuzzy-search, in JavaScript. https://github.com/krisk/Fuse. 33

[78] Roche, D. G., Lanfear, R., Binning, S. A., Haff, T. M., Schwanz, L. E., Cain, K. E., Kokko, H., Jennions, M. D., and Kruuk, L. E. (2014). Troubleshooting public data archiving: suggestions to increase participation. *PLoS biology*, 12(1):e1001779. 10

[79] Roosendaal, T. and Selleri, S. (2004). *The Official Blender 2.3 guide: free 3D creation suite for modeling, animation, and rendering*, volume 3. No Starch Press San Francisco. 85

[80] Rumbaugh, J., Jacobson, I., and Booch, G. (2004). *Unified modeling language reference manual, the*. Pearson Higher Education. 9

[81] Sanyal, J., Zhang, S., Dyer, J., Mercer, A., Amburn, P., and Moorhead, R. (2010). Noodles: A tool for visualization of numerical weather model ensemble uncertainty. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):1421–1430. 69

[82] Satyanarayan, A., Moritz, D., Wongsuphasawat, K., and Heer, J. (2017). Vega-lite: A grammar of interactive graphics. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):341–350. 6

[83] Satyanarayan, A., Russell, R., Hoffswell, J., and Heer, J. (2016). Reactive vega: A streaming dataflow architecture for declarative interactive visualization. *IEEE transactions on visualization and computer graphics*, 22(1):659–668. 6

[84] Seelab (2019 (accessed April 13, 2019)). Rendering Moana Using Tapestry. https://github.com/seelabutk/tapestry-moana. 93

[85] Shannon, P., Markiel, A., Ozier, O., Baliga, N. S., Wang, J. T., Ramage, D., Amin, N., Schwikowski, B., and Ideker, T. (2003). Cytoscape: a software environment for integrated models of biomolecular interaction networks. *Genome research*, 13(11):2498–2504. 14

[86] Shoemake, K. (1985). Animating rotation with quaternion curves. In *Proc. ACM SIGGRAPH*, volume 19, pages 245–254. 62

[87] Shreiner, D., Sellers, G., Kessenich, J., and Licea-Kane, B. (2013). *OpenGL programming guide: The Official guide to learning OpenGL, version 4.3*. Addison-Wesley. 80

[88] Simmhan, Y. L., Plale, B., and Gannon, D. (2006). A framework for collecting provenance in data-centric scientific workflows. In *Web Services, 2006. ICWS'06. International Conference on*, pages 427–436. IEEE. 11

[89] Sisneros, R., Jones, C., Huang, J., Gao, J., Park, B.-H., and Samatova, N. (2007). A multi-level cache model for run-time optimization of remote visualization. *IEEE Trans. on Visualization & Computer Graphics*, 13(5). 7

[90] Spence, R. (2007). *Information Visualization: Design for Interaction, 2nd Ed.* Prentice Hall. 43

[91] Stanton, E. T. and Kegelmeyer, W. P. (2004). Creating and managing" lookmarks" in paraview. In *Information Visualization, 2004. INFOVIS 2004. IEEE Symposium on*, pages p19–p19. IEEE. 10, 11

[92] Steele, J. and Iliinsky, N. (2011). *Designing Data Visualizations.* O'Reilly Media, Inc. 21

[93] Stockinger, K., Shalf, J., Wu, K., and Bethel, E. W. (2005). Query-driven visualization of large data sets. In *VIS 05. IEEE Visualization, 2005.*, pages 167–174. 12

[94] Stubbs, J., Moreira, W., and Dooley, R. (2015). Distributed systems of microservices using docker and serfnode. In *Science Gateways (IWSG), 2015 7th International Workshop on*, pages 34–39. IEEE. 52

[95] Suriarachchi, I., Zhou, Q., and Plale, B. (2015). Komadu: A capture and visualization system for scientific data provenance. *Journal of Open Research Software*, 3(1). 11

[96] Tableau (2018 (accessed March 28, 2018)). Tableau's Superstore dataset. https://community.tableau.com/docs/DOC-1236. 22

[97] Tamm, G. and Slusallek, P. (2015). Plugin free remote visualization in the browser. In *SPIE/IS&T Electronic Imaging*, pages 939705–939705. International Society for Optics and Photonics. 6, 7

[98] Tamm, G. and Slusallek, P. (2016). Web-enabled server-based and distributed real-time ray-tracing. In *Proceedings of the 16th Eurographics Symposium on Parallel Graphics and Visualization*, pages 55–68. Eurographics Association. 6, 7

[99] Torbert, S. (2016). *Applied computer science.* Springer. 28

[100] Vines, T. H., Andrew, R. L., Bock, D. G., Franklin, M. T., Gilbert, K. J., Kane, N. C., Moore, J.-S., Moyers, B. T., Renaut, S., Rennison, D. J., et al. (2013). Mandated data

archiving greatly improves access to research data. *The FASEB journal*, 27(4):1304–1308. 10

[101] W3C (2017 (accessed March 21, 2017)). Embedding custom non-visible data with the data attributes. https://www.w3.org/TR/2011/WD-html5-20110525/elements.html#embedding-custom-non-visible-data-with-the-data-attributes. 49

[102] W3C (2019 (accessed January 23, 2019)). An Overview of the PROV Family of Documents. 11

[103] Wald, I., Johnson, G., Amstutz, J., Brownlee, C., Knoll, A., Jeffers, J., Günther, J., and Navratil, P. (2017). Ospray-a cpu ray tracing framework for scientific visualization. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):931–940. 12

[104] Wiegand, T., Sullivan, G. J., Bjontegaard, G., and Luthra, A. (2003). Overview of the h.264/avc video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):560–576. 31

[105] Wilhelmson, R., Straka, M., Sisneros, R., Orf, L., Jewett, B., and Bryan, G. (2013). Understanding tornadoes and their parent supercells through ultra-high resolution simulation/analysis. 69

[106] Wu, Q., Gao, J., Zhu, M., Rao, N. S., Huang, J., and Iyengar, S. (2008). Self-adaptive configuration of visualization pipeline over wide-area networks. *IEEE Transactions on Computers*, 57(1):55–68. 7

[107] Yoo, C. S., Sankaran, R., and Chen, J. H. (2007). Direct numerical simulation of turbulent lifted hydrogen jet flame in heated coflow. Unpublished manuscript. 69

[108] Yu, H., Ma, K.-L., and Welling, J. (2004). A parallel visualization pipeline for terascale earthquake simulations. In *Proc. of the ACM/IEEE Supercomputing Conference (SC'04)*, pages 49–49. IEEE. 13

[109] Zach, C., Mantler, S., and Karner, K. (2002). Time-critical rendering of discrete and continuous levels of detail. In *Proc. of the ACM Symp. on Virtual Reality Software and Technology*, pages 1–8. ACM. 12

[110] Zhou, H., Xu, P., Yuan, X., and Qu, H. (2013). Edge bundling in information visualization. *Tsinghua Science and Technology*, 18(2):145–156. 82

[111] Zhou, J. and Takatsuka, M. (2009). Automatic transfer function generation using contour tree controlled residue flow model and color harmonics. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1481–1488. 12

[112] Zhou, M., Chen, Q., Mockus, A., and Wu, F. (2017). On the scalability of linux kernel maintainers' work. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 27–37. ACM. 86

Vita

Mohammad Raji received a BS and an MS in Computer Engineering from Razi University, Iran in 2008 and 2012 respectively. In 2014, he started working under Dr. Jian Huang at The University of Tennessee, Knoxville (UTK) in the area of data visualization. During his studies at UTK, he completed a Summer internship at the National Center for Supercomputing Applications where he worked on the line between visualization and deep learning. In 2017, he received his second MS in Computer Science, and in August 2019, Mohammad graduated with a Doctor of Philosophy degree in Computer Science with a focus on large web-based data visualization. Mohammad's research interests include web-based data visualization systems, large scale visualization architectures, and deep learning.