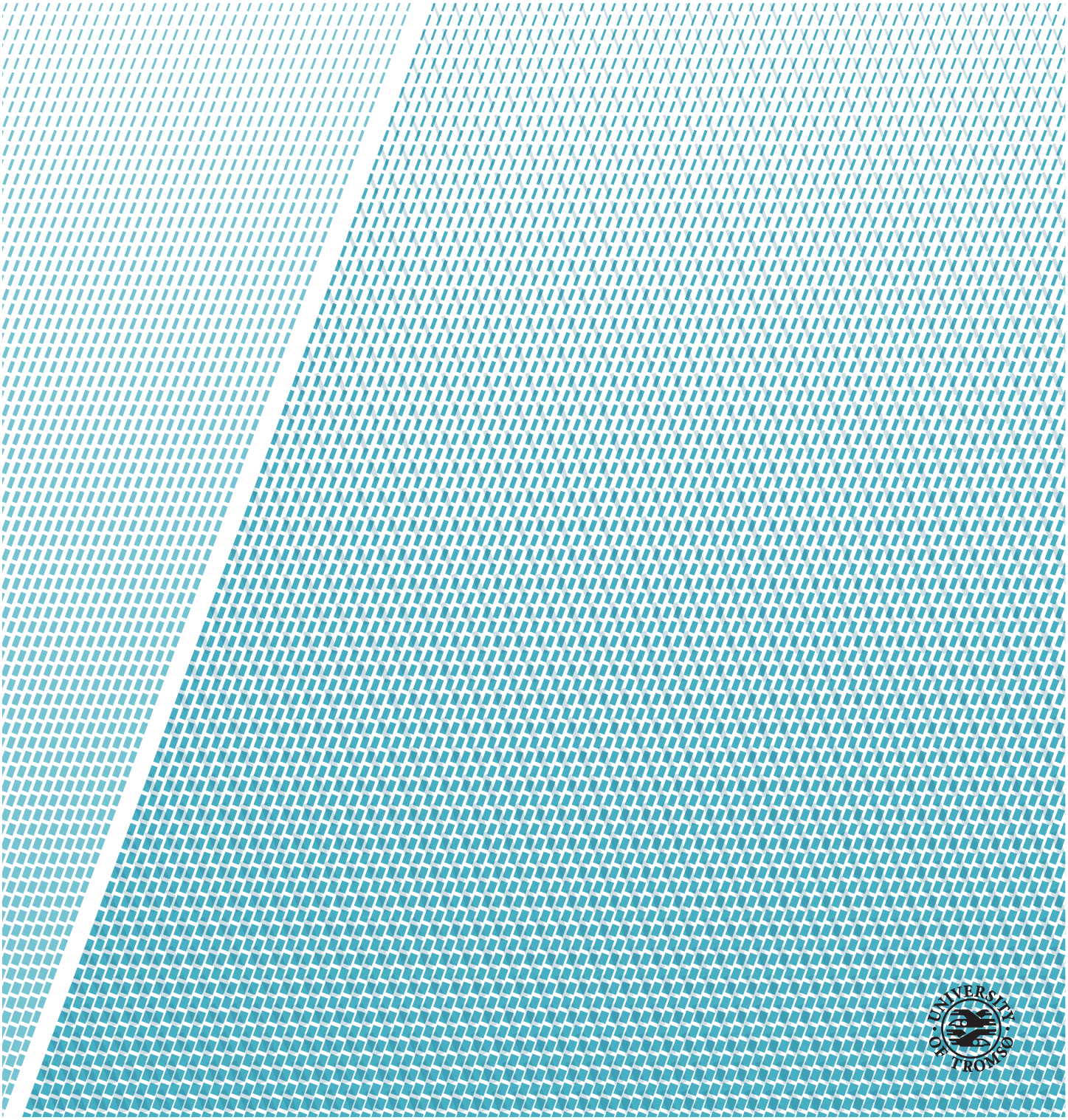
UiT
THE ARCTIC
UNIVERSITY
OF NORWAY

Faculty of Science and Technology
Department of Computer Science

A framework for building scalable web applications for high-resolution cluster-based display walls

Jason Tang

Master thesis in Computer Science October 2015



Abstract

As technology advances, researchers in the natural sciences collect ever-increasing amounts of data. While computer science research often focuses on effective ways to perform computations on large data sets, the visualization of large data sets can be just as important for achieving new insights. Just as cluster computing enables scalable computation on large data sets, so can cluster-based display walls enable scalable visualization of large data sets.

At the same time, visualization and user interface libraries are being most extensively developed for the web. Examples of such libraries include D3 for visualization and Bootstrap for user interfaces. Such libraries often contain built-in support for scaling down to small displays (i.e. on mobile devices), however, they have no such support for scaling up to cluster-based display walls, which require coordination among multiple display hosts.

This thesis presents a framework, provisionally named Browzawall, for building web applications that scale up to high-resolution cluster-based display wall environments. Browzawall consists of several JavaScript libraries and a WebSocket server. By using Browzawall, application developers can build web applications that scale from mobile and desktop environments up to cluster-based display wall.

Acknowledgements

I would like to thank my adviser, Associate Professor John Markus Bjørndalen, and co-adviser Bjørn Fjukstad for their input and guidance, and most importantly, encouragement.

Thanks to classmate Erlend Graff for assistance with LaTeX and the formatting of this document.

A special thank you to my family for their patience, love, and support.

Finally I would like to thank Live and Kjetil Gundersen, Sunniva Stette, Lars Sørensen, Åse Bjerkan, and Joe Niemi for their friendship and support, especially in the home stretch.

Contents

List of Figures

List of Tables

List of Abbreviations

AJAX Asynchronous JavaScript and XML

API application programming interface

CPU Central Processing Unit

CSS Cascading Style Sheets

D3 Data-Driven Documents

DD3 Distributed D3

DDOS Distributed Denial of Service

DOM Document Object Model

DSTM Distributed Software Transactional Memory

HTML5 version 5 of the HyperText Markup Language standard

HTML HyperText Markup Language

HTTP HyperText Transfer Protocol

IPC Inter-Process Communication

KEGG Kyoto Encyclopedia of Genes and Genomes

NOWAC Norwegian Women and Cancer

SVG Scalable Vector Graphics

TCP Transmission Control Protocol

UI User Interface

URL Uniform Resource Locator

UiT University of Tromsø

VNC Virtual Network Computing

1 Introduction

Scientific researchers collect ever-increasing amounts of data. Much work has been done to create architectures and frameworks to analyze large data sets using computing clusters. However, to gain insights into scientific data, it's often helpful to not just analyze the data, but also to visualize it. Meaningfully visualizing large quantities of data can be difficult because display technology has not advanced as quickly as other computing technologies, for example Central Processing Unit (CPU) technology. In order to achieve larger, higher-resolution displays, display walls have been developed by tiling multiple projectors or LCD displays on a physical wall[1]. Display walls controlled by a single computer are possible within certain limits, since modern computers can support multiple graphics cards, each supporting multiple displays. To scale up any further requires a cluster-based approach with multiple computers each controlling one or more of the tiled displays.

Meanwhile, software development has been revolutionized by the Internet and web technologies. The web has developed into a full-fledged application platform, as is reflected by the version 5 of the HyperText Markup Language standard (HTML5)[1] standard published October 2014. One of the major advantages of web applications is that they are accessible by anyone with a web browser and an Internet connection, regardless of operating system or CPU architecture. Web applications are thus relatively easy to deploy, update, and maintain, since application users usually don't need to install anything on their

1. http://www.w3.org/TR/html5/

own personal devices.

This ease of deployment has created incredible interest in the web as an application platform, which has in turn motivated the development of code libraries to make web applications easier to develop. On the client side alone, jQuery[2] is notable for its ubiquity, while Bootstrap[3] is notable for enabling User Interface (UI) development across diverse screen resolutions, and Data-Driven Documents (D3)[4] is notable for its ability to visualize data.

This thesis presents a framework for building web applications that can scale up to a cluster-based display wall, thus allowing application developers to combine the rich UI and visualization libraries of the web with the high resolutions available on display walls. This framework, still very much a prototype, has been given the provisional name of Browzawall, and will be referred to as such in this thesis.

1.1 Challenges

There are two main challenges in adapting interactive web applications to a cluster-based display wall environment. The first challenge is visually scaling the application: the application must appear correctly when multiple hosts are each responsible for displaying a small segment of the whole, and the application should be able to take advantage of the display wall's full resolution. The second challenge is that of data synchronization: any changes in the application state (or at least those changes that result in an updated display) must be communicated to all applicable display hosts.

These challenges are distinct. One can imagine a scenario where visual scaling is desired, but synchronization is unnecessary, such as viewing large static images or diagrams. One can also imagine a scenario where synchronization is desired, but visual scaling is not, such as a remote collaboration system. However, for interactive web applications on a cluster-based display wall, both of these challenges must be addressed.

When addressing these challenges, both transparency and opacity are valuable. An application programmer should be able to treat a display wall almost identically to a single display, yet should also be able to create applications that can detect when they are running in a high-resolution environment, in order

2. `http://jquery.com`
3. `http://getbootstrap.com`
4. `http://d3js.org`

to fully utilize the available displays.

1.2 Organization

The rest of this document is organized as follows. Chapter 2 provides background. Chapter 3 provides an exploration of the solution space for running web applications on cluster-based display walls and describes solutions considered but ultimately discarded. Chapter 4 covers the design of Browzawall while Chapter 5 covers implementation details. Example applications built with Browzawall are described in Chapter 6, and an evaluation of Browzawall is provided in Chapter 7. Chapter 8 contains concluding remarks and Chapter 9 presents opportunities for future work.

2 Background

This chapter provides background about the Tromsø Display Wall, the collaboration with Norwegian Women and Cancer (NOWAC), as well as previous work related to running web applications on display walls.

2.1 Tromsø Display Wall

The Tromsø Display Wall has been on ongoing subject of research at University of Tromsø (UiT) since 2004-2005[1]. The Display Wall currently consists of a Rocks[1] cluster of commodity PCs running Ubuntu Linux. There is one front-end node and 28 tile nodes. Each tile is connected to a 1024x768 pixel projector. These projectors project onto a screen arranged in a 7x4 pattern, thus creating a total resolution of 7168x3072 or 22 megapixels. The Tromsø Display Wall also has a motion-capture system to enable interaction via gesture. For an overview of applications developed for the Tromsø Display Wall, see [1] or `http://hpds.cs.uit.no`.

1. `http://www.rocksclusters.org`

2.2 NOWAC and Kvik

Ongoing work at the Department of Computer Science involves aiding epidemiology researchers in exploring and visualizing data from the NOWAC study. The NOWAC study seeks to discover the possible relationships between lifestyle and cancer risk, and requires analyses of biological data. The NOWAC study has collected questionnaire data from over 170,000 women, as well as over 60,000 blood samples and over 800 biopsies[4]. Kvik is a framework developing applications for interactive exploration of biological data from the NOWAC study together with knowledge from online databases such as Kyoto Encyclopedia of Genes and Genomes (KEGG)[4]. One application that uses the Kvik framework is Kvik Pathways[5]. Kvik Pathways is an application that epidemiologists can use to browse gene expression data in the context of biological pathways maps Biological pathways are graphical representations of biological processes in an organism. Researchers want to use these to get an overview of different progresses that change during development of cancer.

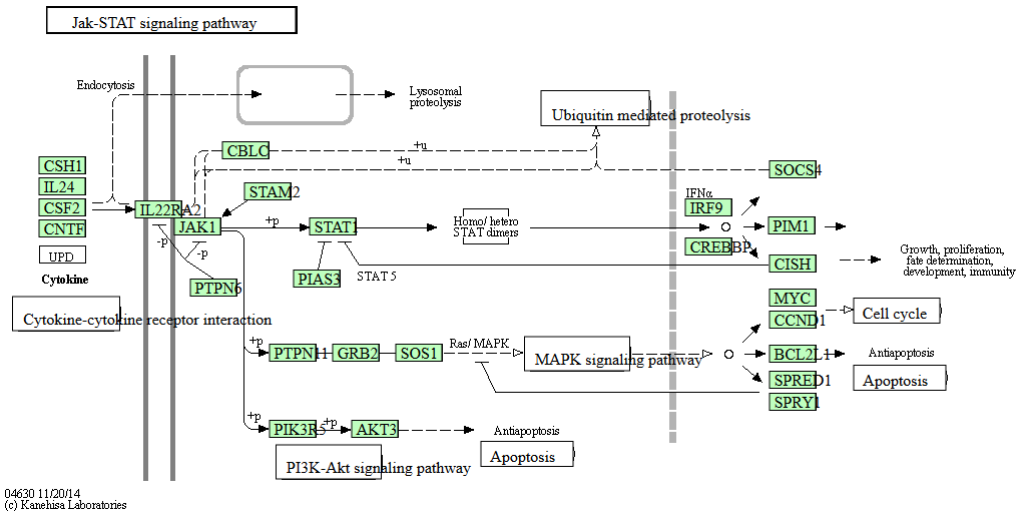


Figure 2.1: Example of a gene pathway diagram generated by Kvik.

2.3 HTML and the Document Object Model

The Document Object Model (DOM) is both a structural representation for HyperText Markup Language (HTML) documents, and a programming interface for accessing and modifying that structure[9]. This thesis assumes a general understanding of HTML and the DOM.

2.4 Related Work

This section describes work related to using web applications on display walls, whether directly or indirectly.

2.4.1 Multibrowsing

An early system to display coordinated web content in a multiple-display environment was Multibrowsing[8]. Multibrowsing allowed users to interact with one browser window, yet control the content displayed on other browser windows on remote displays. This involved sending a Uniform Resource Locator (URL) to a daemon running on the display hosts, which would then open the URL in a browser window. Browzawall's architecture resembles Multibrowsing in that both use a central server to facilitate many-to-many browser communication. However, Multibrowsing treats displays individually, while Browzawall tries to treat several tiled displays as a single display. Multibrowsing also treats web pages statically (once they have been loaded by the browser), as there is no way to modify content on a remote display without issuing a new URL. Browzawall has been able to take advantage of the more recent WebSocket[3] standard to push content updates without reloading an entire page.

2.4.2 Hydrascope

Hydrascope[7] is an existing approach to adapting existing web applications to display wall environments. Hydrascope uses a browser extension and meta-applications to coordinate the various browser windows. This requires various methods of reverse-engineering including DOM inspection and event injection. For example, a meta-application may watch for a changing page number in a presentation application, and trigger events in the other windows to update them accordingly. Hydrascope was designed to handle cases where the underlying application already existed and could not be modified (the application could even be owned by a third party, as in Google Docs[2]). We are interested in developing new applications, or adapting applications that we have built in the past, so these reverse-engineering techniques are complicated and unnecessary.

2. http://docs.google.com/

2.4.3 VNC

Virtual Network Computing (VNC) has been successfully implemented in a cluster-based display wall environment[6][11]. A user could thus run both a web browser and a VNC server on a personal computer, and relay the display to display wall. This may be sufficient for many applications, though it has several performance limitations. The computer that runs the VNC server becomes a bottleneck for both CPU and network usage; CPU usage because it must render the entire document, and network usage because it must transfer the segments of the rendered document to each of the display tiles. These segments are transferred as pixel data, which in many cases will require more bandwidth to transfer than the original HTML document.

2.4.4 AtomizeJS

AtomizeJS[3] is a JavaScript library and Node.js server implementation that aims to provide synchronization using a Distributed Software Transactional Memory (DSTM) model. We did some exploration of using AtomizeJS as a synchronization mechanism.

We ultimately abandoned AtomizeJS because it caused cross-origin security errors that we were unwilling to resolve or work around. It was also a more complex solution than we needed. AtomizeJS was designed for use with Direct Proxies[4]. For compatibility with browsers that do not yet support Direct Proxies, either the application JavaScript code must be run through a provided code translation tool to generate equivalent AtomizeJS application programming interface (API) calls, or the application programmer must write those API calls directly, resulting in verbose code.

3. `http://atomizejs.github.io/`
4. A feature of JavaScript that has not been formally adopted as of May 2015.

3

Solution Space

This chapter provides a brief analysis of possible approaches to addressing the challenges of visual scaling and data synchronization, followed by a discussion of two alternative solutions considered before settling on Browzawall's current architecture.

3.1 Visual Scaling

This section identifies two possible models for scaling a document up to a display wall, which throughout this thesis will be referred to as the "divide and distribute" method, and the "replicate and zoom" method.

3.1.1 Divide and Distribute

In this model, a master loads the document, and then divides it according to the arrangement of displays in the system. The master directly tells each individual display tile what it should display. Any changes to the document need only be sent to the tiles that display the relevant elements.

Divide and distribute could be implemented in one of the following ways:

- A standard web browser renders the entire document, then client-side

JavaScript code or a browser extension inspects the rendered document to calculate positioning of elements on the display wall.

- A stand-alone process implements rendering logic itself.
- Application developers use a custom API for positioning top-level elements on the display wall.

In the first two methods, the master becomes a performance bottleneck. The second method also requires excessive redundant programming effort. The third method prevents application developers from using normal HTML/Cascading Style Sheets (CSS) flow rules to define the document as a whole.

3.1.2 Replicate and Zoom

In this model, each display host loads a complete copy of the document. The display host then uses information it has about its position in the overall display to display the portion of the document it is responsible for, essentially "zooming in" on that portion of the document. Any changes to the document then need to be replicated to all display hosts in the system.

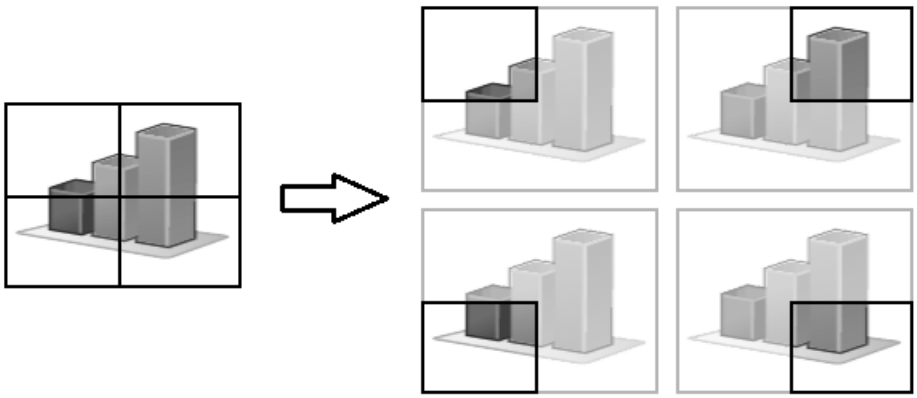


Figure 3.1: A graph replicated among four display hosts. Each display host has a copy of the entire graph, but displays only part of it.

The replicate and zoom method results in redundant network and CPU usage, because every display tile will load a copy of the entire document while displaying only a fraction of it. In the worst case, a display wall composed of N tiles would display N times as much information as a single display of the same resolution; thus each tile would load N times as many elements as it would if

it only loaded the elements it visibly displayed.

Replicate and zoom is also prone to synchronization problems when retrieving changing content. Since each display host retrieves content independently of each other, it's possible that some hosts could get different versions of the content. Figure 3.2 shows an example of this on the Tromsø Display Wall. While most of the content is the same, the advertisements on the top and right side clearly differ on different tiles.

Figure 3.2: Displaying the dagbladet.no home page on the Tromsø Display Wall.

3.2 Data Synchronization

Regarding coordination and synchronization, it's helpful to look at the behavior of web applications in general. Interactive web applications can be generalized as in figure 3.3: A user sends input, which is received by an event handler. The event handler executes code that modifies the DOM (or DOM elements such as canvasses), and the browser then renders the resulting DOM into an image to display. Synchronizing the display can be achieved by synchronizing the application at any one of these stages in the pipeline. What follows is a brief analysis of the benefits and drawbacks of synchronization at each of these particular stages.

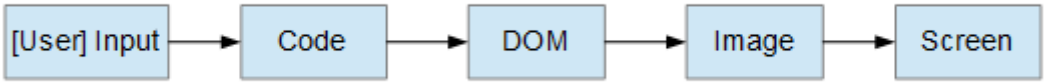


Figure 3.3: Generalized web application behavior.

3.2.1 Synchronizing Input

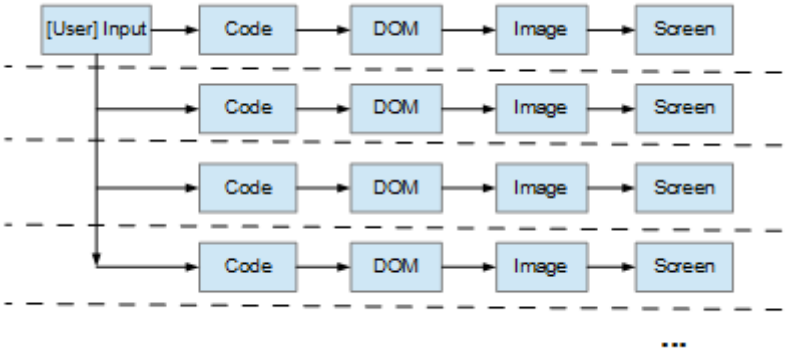


Figure 3.4: Synchronizing input between multiple display hosts.

Input can be synchronized by event handlers that broadcast events (or at least the relevant data from the events) to all display hosts. Display hosts thus must also listen for remote events, and invoke the appropriate application-specific code whether the event originated on the local host or on a remote host.

The primary drawback to synchronizing input is that it only works for applications that are completely deterministic; that is, the same sequence of input always results in the same output. Any application with randomness, including many kinds of games, will not work properly.

3.2.2 Synchronizing Code

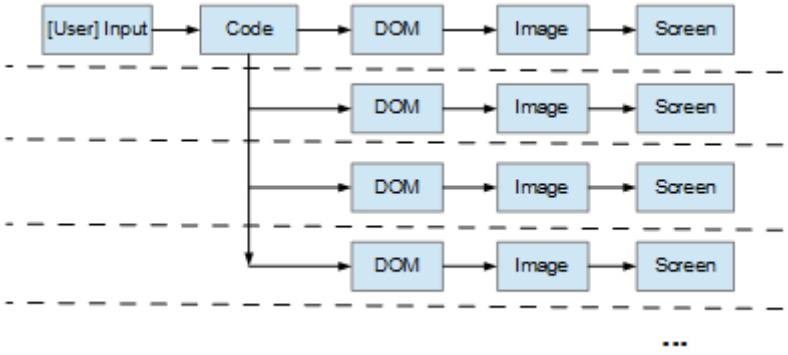


Figure 3.5: Synchronizing code between multiple display hosts.

Synchronizing code is the process of instructing remote display hosts to execute code to achieve the desired display. This can take an RPC-like form where the user interface invokes a pre-defined function on all display hosts. Alternatively, the user interface can send the code itself to be executed, either by serializing a function, or by sending a string of code to be interpreted via eval().

The primary challenge with this approach is being able to pass parameters as either values or references as necessary. In addition, this method may require writing separate application-specific code for both sides of the network (both before and after transmission), depending on the programming model used, and in the absence of libraries to handle one side automatically. For example, in an RPC model, one function must be defined to handle the event and then remotely call a second function to update the DOM or other application state.

3.2.3 Synchronizing DOM

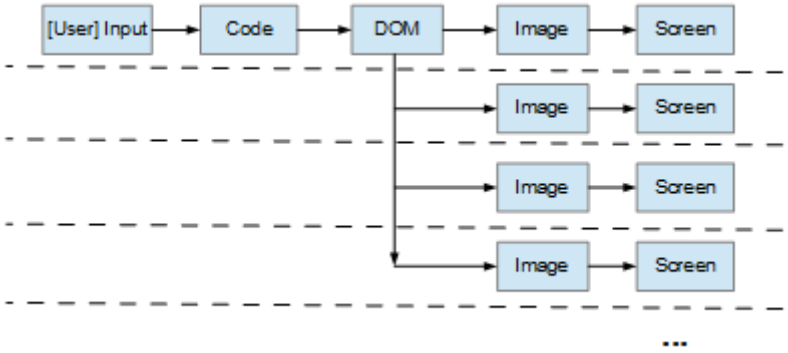


Figure 3.6: Synchronizing the DOM between multiple display hosts.

Synchronizing DOM elements themselves is a straight-forward and logical approach, particularly when input results in only a small amount of changes to the DOM. On the other hand, when a large number of elements are generated programmatically (for example, with D3), it can become incredibly inefficient to transmit these elements. Compression can help, and there exist freely-available compression libraries for JavaScript, but for some operations the compressed HTML will still be larger than the size of the original data, which can even be cached.

One major advantage of synchronizing the DOM is that it is a state-based method of synchronization rather than an event-based method. As such, it is particularly well suited for situations where fault-tolerance and fault-recovery are important, where events could be dropped.

3.2.4 Synchronizing Rendered Documents

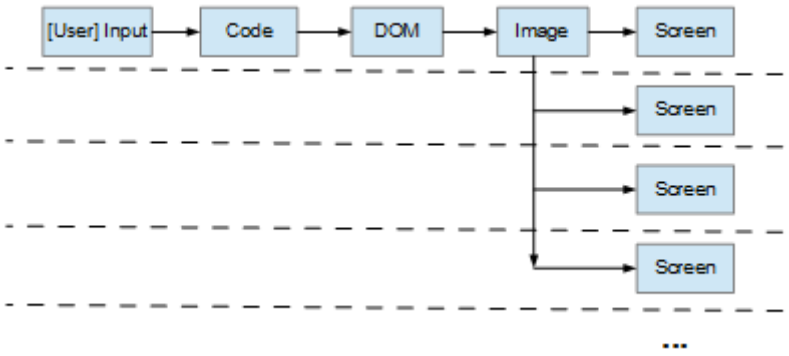


Figure 3.7: Synchronizing the rendered document between multiple display hosts.

Synchronizing the image resulting from rendering the document is included for the sake of completion. As mentioned above, this is what a VNC system does. Such a VNC system could display any application, so there is no point in developing a browser-specific system that would have the same benefits and drawbacks as a general-purpose VNC.

3.3 Alternatives Considered

Before settling on our current architecture and design, we explored other alternatives, such as creating a distributed browser process, or using properties specific to Scalable Vector Graphics (SVG).

3.3.1 Distributed Browser

We briefly considered creating a distributed browser process. Modern versions of at least one open-source browser already have an architecture in which each tab is rendered by a separate rendering process that communicates with the master process via an Inter-Process Communication (IPC) mechanism[10]. Figure 3.8, shows an architectural overview of the Chromium[1] open-source browser. The I/O thread (top left) handles all network communication, and passes data via IPC to the appropriate Renderer process. There is one Renderer process per browser tab.

1. `http://www.chromium.org/`

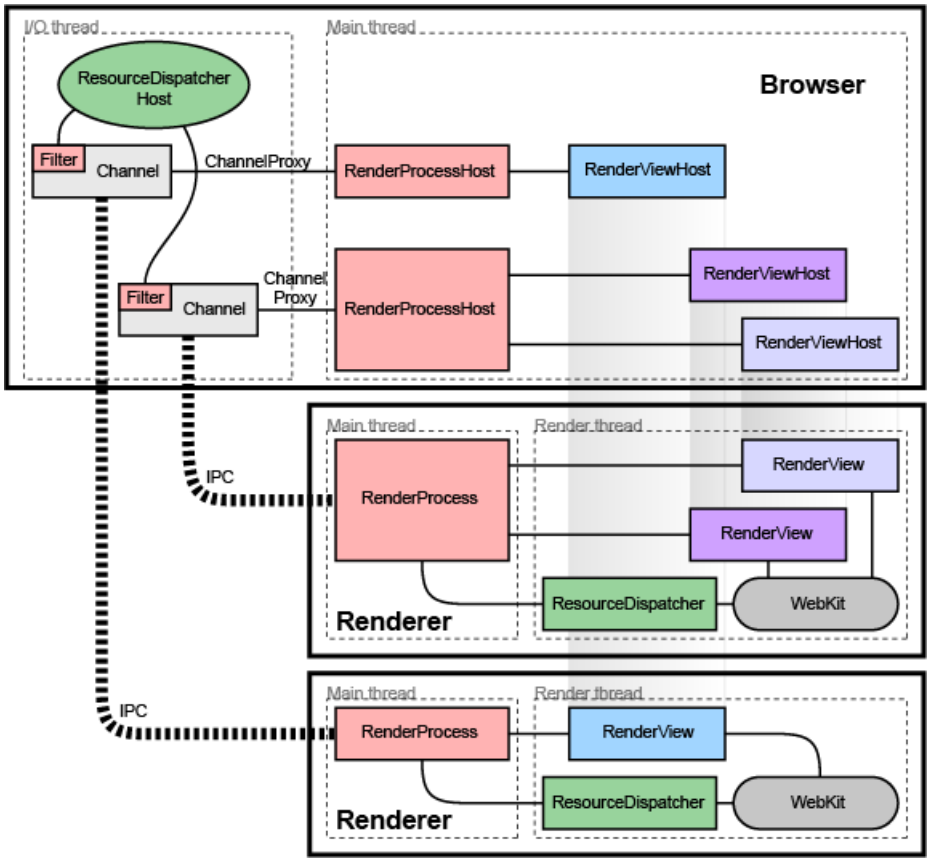


Figure 3.8: Architecture overview of the Chromium open-source web browser. From [10]

It is thus conceivable to scale up and distribute these rendering processes across all display hosts, replacing local IPC with network communication. Theoretically, such a browser would allow any existing web application to scale up to a cluster-based display wall. However, even if technically feasible, such a distributed browser would require maintenance in parallel with the original open-source browser in order to support new features, fix bugs, and patch security vulnerabilities.

3.3.2 SVG transform and viewBox

Early work on this project focused on displaying SVG diagrams on the display wall, since that is the format of the diagrams created by Kvik[4]. By using SVG's transform and viewBox[2] attributes, we were able to scale up (via "replicate and

2. `http://www.w3.org/TR/SVG/coords.html`

zoom") SVG diagrams to the display wall. However, once we found that CSS3 Transform functions can accomplish the same behavior for not only SVG but all visible HTML elements, we abandoned this SVG-specific solution.

4

Design

Browzawall provides a method for scaling up web applications to display walls, as well as several methods of synchronizing web applications through a central communication server. Browzawall is based on a replicate and zoom model of visual scaling. Replicate and zoom was chosen because of its ease of implementation, decentralization, and flexibility for the application developer.

- **Ease of implementation:** Replicate and zoom means that the system as a whole maintains a single state regardless of how many display hosts are participating.

- **Decentralization:** The scaling logic is contained entirely within the browser window. Browzawall does have a central component, the communication server, but this only handles basic communication and synchronization functionality. With the majority of the functionality in client-side JavaScript libraries, it's easy to add new functionality without risking the disruption of existing applications.

- **Flexibility:** Replicate and zoom models the entire display wall as one large HTML page, and can be treated as such by designers and application programmers. Normal CSS rules will work to lay out pages that span the entire wall.

The replicate and zoom method lends itself to a dual-channel approach to browser communication; Normal HyperText Transfer Protocol (HTTP) com-

munication is used to load HTML documents and other resources, while a WebSocket connection is used for coordination and synchronization. By maximizing the amount of data transferred via HTTP, the system takes advantage of the browser's ability to parallelize requests and cache frequently-requested files.

4.1 Architecture

Browzawall consists of four client-side JavaScript libraries and a communication server. The client-side libraries provide connectivity to the communication server and various methods of data synchronization, each with their advantages and disadvantages. In addition, the client-side libraries handle scaling applications up for display on a display wall.

The four client-side libraries are:

- comm.js – a library for basic communication and synchronization
- walldoc.js – a library for visual scaling on a display wall
- domsync.js – a library for synchronizing DOM elements, subtrees, and input across multiple hosts
- dd3.js – an experimental wrapper and extension for D3 that supports performing synchronized operations according to D3's programming idioms.

4.2 Communication Server

The communication server is a single-threaded WebSocket[3] server built on Node.js[1]. WebSocket is the ideal protocol for communication between browsers and the communication server, because once connected, either the browser or the server can initiate message transmission. This is in contrast with other approaches such as Asynchronous JavaScript and XML (AJAX), where the browser must always initiate an exchange.

Node.js was chosen for its ease of use. It has HTTP and WebSocket support, and it was easy to find example WebSocket servers on the Internet. Additionally,

1. `http://nodejs.org`

it is convenient to work in the same programming language (JavaScript) for both the client libraries and the server implementation.

Because of the replicate and zoom model, the communication server will typically receive incoming messages and broadcast them to all other applicable display hosts. However, some messages are directed to the server itself and thus do not get broadcast. These server-directed messages include messages for session identification and server-side synchronization, both described below.

4.3 comm.js

The comm.js library handles connection to the communication server. This happens automatically when the script file is loaded. The library supplies a session id in order to keep different applications from interfering with each other while connected to the same communication server. Comm.js several methods of synchronizing applications at the code stage. The most robust method is termed a Broadcast Function Call, as it allows one browser window to call a particular function, with particular arguments, on every participating browser. The comm library automatically handles marshalling and unmarshalling parameters.

Comm.js also has another feature for synchronizing display hosts in cases where fault-tolerance is more important. Session-persistent shared objects are objects that are stored in a key-value store in the communication server. Browsers can push updates to the server, which stores it and then pushes it to any other participating browsers. Any browser windows that join the session late (or recover from a crash), can receive the most recent values directly from the server.

Finally, the Comm library provides other server-side synchronization capabilities. It provides a sort of barrier implementation that is useful for keeping a smooth feel when browsers have intensive computation and I/O to perform before updating the view. Comm.js also provides a function for generating an HTML id that is unique session-wide.

4.4 walldoc.js

Walldoc handles the basic visual scaling of Browzawall. Basically, it turns the document into one giant document with dimensions as large as the display

wall, and then zooms in appropriately based on which tile it is configured to be.

4.5 domsync.js

Domsync aims to be a re-usable client-side JavaScript library that covers common use cases with distributed web applications. Domsync consists of two major components: DOM synchronization and distributed event handlers.

Domsync provides methods to synchronize DOM elements and subtrees directly. These methods cover common use cases such as modifying attributes, modifying style declarations, and updating child trees.

Distributed event listeners provide the ability to synchronize the web application by synchronizing user input, while using an API that is nearly identical to normal DOM event listeners. By registering a distributed event listener function for a particular event, the Domsync library will automatically broadcast any of those events that trigger locally. Domsync will then invoke the event listener function for both events triggered locally and events received from remote hosts.

4.6 dd3.js

DD3 was an experiment to automatically broadcast D3 functions. For more details, see the Implementation chapter below.

5 Implementation

This chapter describes various implementation details of interest in the Browzawall framework.

5.1 System Integration

The system for developing and testing Browzawall was the Tromsø Display Wall, described in Section 2.1. Existing open-source web servers were used to serve web pages; first Apache[1], then the http-server[2] Node package because of its easier deployment. The Browzawall communication server is a separate process and listens for Transmission Control Protocol (TCP) connections on a separate port.

The Tromsø Display Wall uses the Chromium browser[3]. Browsers are launched on the display tiles issuing a cluster-fork[4] command from the cluster front-end. Chromium's –app command-line parameter specifies the URL to load, and launches the browser window with the viewport fullscreen. The URL query string consists of the output of a hostname command. By embedding hostnames

1. `http://httpd.apache.org`
2. `https://www.npmjs.com/package/http-server`
3. `http://chromium.org`
4. Part of Rocks; `http://www.rocksclusters.org`

in the URL, browser windows can access this information and use it to display the correct contents for each tile.

5.2 Visual Scaling

Browzawall relies on each display host to supply information about that host's role in the system. A role may be either "tile", indicating a display tile that's part of a wall, and or a "default" role typically used by control interfaces. Display tiles also need to supply information about their relative position within the wall. On the Tromsø Display Wall, the host names of the tiles includes this positioning information, so it's sufficient to run a hostname command and paste the result into the URL's query string.

The JavaScript file responsible for visually scaling the document, walldoc.js, currently contains hard-coded parameters and logic that apply specifically to the Tromsø Display Wall.

The host names of all tiles is of the form tile-X-Y, where X is the horizontal position on the wall (0-6, left-to-right), and Y is the vertical position (0-3, bottom-to-top). Given these positions, it's simply a matter of multiplying the horizontal position by the horizontal resolution of each display to determine the horizontal offset. The vertical offset is calculated in a similar manner, however, the vertical positions must be inverted because the positions increase from bottom to top, while HTML Y-coordinates increase from top to bottom.

The appropriate scaling occurs automatically by including walldoc.js, which executes a function on page load to parse the query string. The zooming is then accomplished by simply using CSS 3 transform[5] functions to scale and translate the body of the document appropriately for each tile. Configuration of the wall currently requires manual changes to walldoc.js.

5.3 Session Initiation

Connection to the communication server is automatic when including the comm.js library. The connection routine assumes the communication server is running on the same host as the web server, and assumes a default port.

Upon connection to the server, the client application identifies the session it

5. http://www.w3.org/TR/css3-transforms/

desires to participate in. The session name defaults to the file name of the client page, which under basic use-cases is sufficient for preventing different applications from interfering with each other.

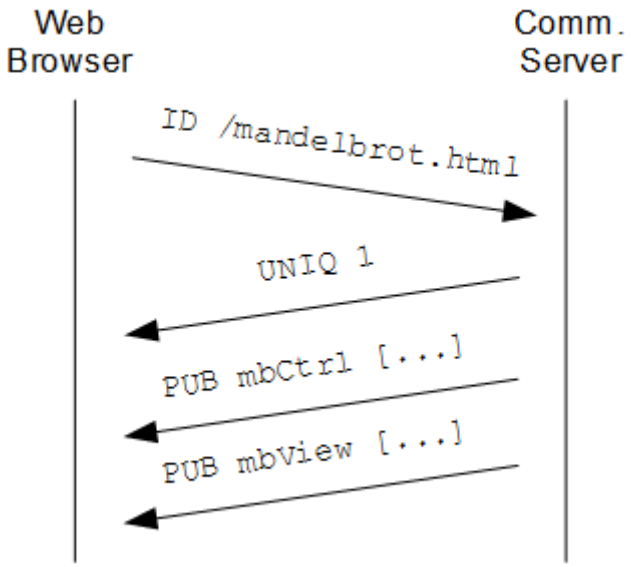


Figure 5.1: Session initiation between web browser and communication server for the Mandelbrot application (described in detail in Section 6.2).

5.4 Message Internals

Messages are sent to and from the communication server via unicode text, as this is the standard implementation of strings in JavaScript. Each message is composed of an operation identifier, followed by a space character and an optional payload of arbitrary text. Payloads are typically JSON-formatted data, since JSON is human-readable, more concise than XML, and built in to JavaScript. The reason the entire message is not JSON is that the server is usually not interested in contents of the payload itself, since most of the time that will just get broadcast to all connected hosts in the session. By separating the operation identifier at the start of the message by just a space character, the server only has to interpret the message up to the first space to determine what to do with a message; it does not have to parse all of the JSON in the message.

5.5 Broadcast Function Calls

Browzawall allows application programmers to invoke JavaScript functions on all display hosts in the current session, for the purposes of updating application state.

The first step is to call comm.register(), which associates a function key with a callback function. The function key is simply an identifier string. The function key is what gets encoded into messages broadcast to the other display hosts, and subsequently used by them to look up the appropriate function to call. Registering a new function under an existing function key will replace the function previously associated with the key.

After a function has been registered, it can be invoked by calling comm.callAll(). The first argument to callAll() can be either the appropriate function key or the function itself. Any remaining arguments are encoded into JSON and transmitted over the network. The callAll() function will also call the function locally.

Remote display hosts will receive the message, use the received function key to look up the appropriate function (if one has been registered), decode the JSON parameters, and then call the function with the received parameters.

There are two pitfalls to be aware of when using Browzawall's broadcast function calls:

- As with most other uses of callbacks in JavaScript, "this" in the function body will resolve to window. Function.prototype.bind() can work around this issue.

- The function's arguments must be passed by value. Most notably, this means that DOM Nodes (including Elements) can't be passed as arguments to a remote function call. Instead, pass a string such as an id string or an object key. A future enhancement may support Element arguments by automatically encoding and decoding Element selectors in the same manner as the domsync library.

5.6 Session-Persistent Shared Objects

For cases where applications could become unsynchronized if any display host connects (or reconnects) late, the communication server provides the ability to store JavaScript objects in a key-value store. A late-connecting display host

can thus receive objects from this store to obtain the current state of the application. Shared objects are available as long as the session is active, that is, as long as at least one host is connected to that session. After the last host disconnects, the server deletes any session-persistent objects so that they can be garbage-collected.

Shared objects are updated according to a publish/subscribe method. Client applications can push updated objects for a particular key to the communication server, which will store the object and push it to all other connected display hosts. Client applications can also subscribe to a particular key, supplying a callback to be invoked when an updated object is received.

Applications may be more interested in storing a log of events rather than the current state. A session-persistent object can be used to create a shared log by declaring an array as one of its members and appending to that array. Note however that the object will not be treated as a log internally, and any changes to the object (including appending to a member array) will result in retransmission of the entire object.

5.7 Barriers

Barriers are a synchronization method whereby all members of a group wait until all of them signal that they have completed some task. Barriers are typically used in parallel computation problems, but here they are available to synchronize displays in situations where intensive computation or network usage can result in inconsistent updating. An example of an application of barriers is the Mandelbrot set described in detail later; since the computation time of regions vary, a smoother user experience is achieved by performing all calculations, invoking a barrier, then updating the canvas when the barrier is complete.

It should be noted that in browsers, JavaScript's programming model does not allow for true barriers. A true barrier is a function that blocks until receiving a notification to proceed. JavaScript is single-threaded and relies on asynchronous programming for web pages to remain responsive. Therefore barriers are implemented by supplying a callback to the barrier function. Any code immediately following the barrier invocation will be executed immediately, before receiving the barrier notification.

5.8 Remote Execution and Troubleshooting

Modern browsers include a JavaScript console, allowing developers to experiment with and execute JavaScript code without editing and reloading a document containing JavaScript. To provide this capability for a display wall environment, Browzawall provides a reval() function. Short for "remote eval()", it accepts a string of JavaScript code as its only parameter. The string is broadcast to all display hosts in the session, where it is executed using the eval() function. The string is also executed locally with eval().

While reval() could be used in production code to satisfy the data synchronization needs of many applications, application developers will likely find other methods easier to use. The security implications of reval() (as well as the other forms of remote code execution in Browzawall) are discussed in detail in Section 7.5.

Browzawall also provides a reloadAll() function that, as expected, issues a command to all display hosts in the session to reload their current document.

5.9 DOM Synchronization

Domsync provides convenience methods replicating DOM elements to all display hosts. Domsync uses the Broadcast Function Calls The following functions are provided:

- syncAttr([name, value]) This method serializes all style defined on the element and broadcasts them to all display hosts. For convenience, users can supply a name and value to update the attribute by that name with that particular value before broadcasting all attributes.

- syncStyle([name, value]) This method serializes all style declarations defined on the element itself (not those defined in style sheets) and broadcasts them to all display hosts. For convenience, users can supply a name and value to update the style declaration by that name with that particular value before broadcasting all style declarations.

- syncInner() This method synchronizes all children of an element by broadcasting the element's innerHTML attribute.

- syncOuter() This method synchronizes an element and all of its children by broadcasting its outerHTML attribute.

5.10 Distributed Event listeners

Another feature provided by domsync.js is distributed event listeners. To the application developer, distributed event listeners behave almost identically to regular event listeners. The preferred way of defining event listeners[] in HTML5 is with addEventListener() and removeEventListener(), both methods from the EventTarget prototype[]. Domsync extends the prototype of EventTarget with addDistributedEventListener() and removeDistributedEventListener(), which accept the same parameter list as addEventListener() and removeEventListener().

Internally, adding a distributed event listener stores the user-defined event listener in a data structure while adding a meta-listener on the EventTarget for the event type in question. When the appropriate event is triggered, the meta-listener encodes the triggered event and sends it over the network. Then it looks up the actual user-defined event listeners (there can be multiple) and executes them locally.

Domsync also listens for remote event messages from the communication server. Upon receiving a remote event, it will look up the event listener for the correct EventTarget and event type (and useCapture), and execute the handler.

5.10.1 Encoding the Event

DOM Events contain references to DOM elements. It makes more sense to encode these references than to try to serialize the element itself. Domsync encodes element references as CSS selectors. The selector is generated by traversing up the DOM tree until reaching either an element with a defined id attribute or the body element. The algorithm relies on the Child Combinator ">" and :nth-child(n) pseudo-class[]. Allowing an id as a base case (instead of just the root of the document) not only prevents unnecessary DOM traversal, but also supports use cases where the entire document is not synchronized between hosts, but only select subtrees. For example, an end user might use a mobile tablet as an interface for the display wall, and that mobile interface might only display a portion of the document at a time.

DOM events can also be triggered on, and contain references to, document and window, which are not Elements, and thus can't be addressed by a CSS selector. Instead, each has a "magic string" to identify it: "#$win" for window and "#$doc" for document. These magic strings were designed to be invalid CSS selectors to avoid potential conflicts with user-defined elements. They are invalid because # denotes an id selector, and a valid id cannot begin with a $.

5.10.2 Decoding the Event

Decoding a remote event consists of two steps: decoding references and assigning dummy methods.

Decoding references at the remote hosts is simply the reverse of the process described above. For each applicable field, the decoder first looks for magic strings, and otherwise invokes document.querySelector() for DOM Elements.

Dummy methods must be assigned to the remote event to allow the same event listener to be executed for both remote and local events. For example, an event listener may want call event.stopPropagation(). On the host where the event is originally triggered, this will have the usual effect of stopping the propagation of the event to elements either higher or lower on the DOM tree (depending on whether or not the event listener is registered for the capture phase). Because remote events do not propagate, the remote event listener simply needs an empty function to call to avoid an error.

5.11 Distributed D3

Distributed D3 (DD3) is a lightweight wrapper to the popular visualization library D3. Distributed D3 provides the ability to execute D3 operations on all Browzawall hosts, while supporting D3's characteristic programming idioms, such as method chaining.

Distributed D3 is based on DD3Batch objects that appear as D3 Selection objects. Calling a method on a DD3Batch object encodes that method call and its parameters, and adds them to a queue. A DD3Batch can be executed only locally by calling local(), or it can be broadcast to all display hosts by calling remote().

A common usage pattern with D3 is to create a selection and assign it to a variable, so that it can be used later for multiple other operations. In a distributed environment where all display hosts are executing the same D3 operations, each display host must store its own copy of the selection, which may then be referenced by DD3 operations initiated by any other host. Distributed D3 allows for storage of selections by accepting an optional key string as an argument to remote() and local(). The selection can be recalled by calling dd3.load(), passing the desired key string.

6

Example Applications

This chapter describes some of the notable example applications developed using Browzawall.

6.1 Wall Controller

The Wall Controller is a prototype interface for directing content on the display wall. The intended use case is that a user operates a control interface from a personal computer, such as a laptop. (The system is designed so that multiple collaborating users could each have their own control interface, though this has not been extensively tested.) This control interface presents a miniaturized overview of the entire wall. The user can click and drag on empty space to specify the position and size of a new element (i.e. the point where the user clicks and the point where the user releases will be opposite corners of the rectangle that bounds the element). The user can then click and drag the element to move it around. The element will always appear in the corresponding location on the display wall. Two types of elements are currently supported, iframes and SVG gene pathway diagrams.

When creating an iframe, which is the default element, the Wall Controller will prompt the user for a web address that will become the src attribute of the new iframe, automatically prepending "http://". This allows the Wall Controller to display arbitrary web pages or other web-hosted files on the display wall. Wall

Controller iframes work great for displaying static content such as images or Portable Document Format (PDF) documents. It is also possible to load other web pages, though they may not display properly for two reasons. First, some web pages are designed to not display when loaded in an iframe in another document. Second, web pages may contain dynamic or changing content, and since all of the display hosts are independently requesting this content, they may retrieve different content. For example, a news web page may show different headlines, or may show different advertisements. Figure 3.2 in Section 9.1 shows this exact problem. Automatically-playing slideshows may get out of sync.

If the user holds the shift key while clicking and dragging, the Wall Controller will instead prompt the user for a gene pathway number, and then create an SVG drawing of the corresponding pathway diagram. These gene pathway diagrams are generated with code contributed from the Kvik[4] project. Adapting Kvik's interactive gene pathway diagrams for a display wall environment was a large part of the motivation behind Browzawall. Figure 6.1 shows the Tromsø Display Wall displaying such a pathway diagram in this way. Figure 6.2 is a closer view of four different pathways on the display wall.

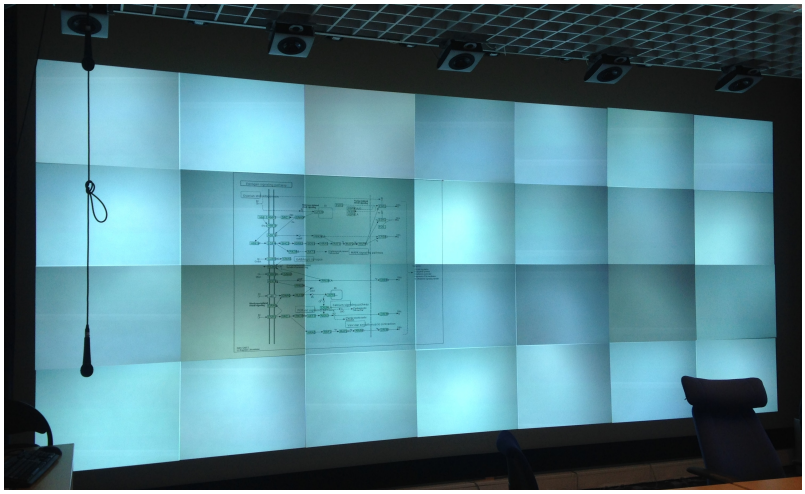


Figure 6.1: A gene pathway diagram on the Tromsø Display Wall

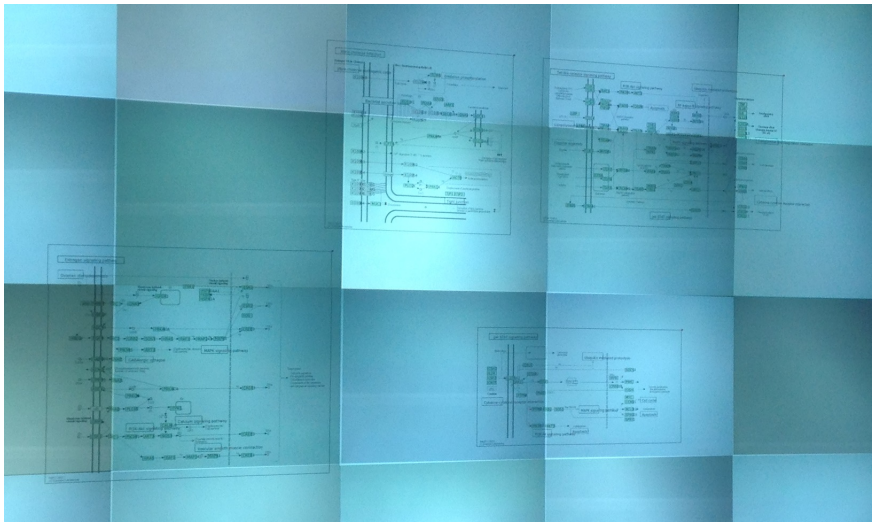


Figure 6.2: Close-up of four pathway diagrams on the Tromsø Display Wall

6.2 Mandelbrot Set

The Mandelbrot Set is a fractal frequently used as an example problem for parallel and distributed computing. The fractal is generated by iteratively performing a computation based on a point's coordinates in real and imaginary space. The number of iterations necessary to exceed a particular bound then gets mapped to a particular color for display. Interesting properties of the Mandelbrot set are that each point can be calculated independently of each other, and that different regions can take different amounts of time to calculate even if they contain the same number of points.

The Mandelbrot set application is interesting because it departs from the typical replicate and zoom model of Browzawall and demonstrates how Browzawall can synchronize browser windows that render their own individual content.

This Mandelbrot set application displays a single full-screen HTML canvas on each display host. Users can click on the canvas to pan and zoom around the Mandelbrot fractal. Display tiles use the communication interface to synchronize the coordinates of the bounding box of the full image, then use walldoc to calculate the coordinates of the bounding box of the tile's own region. After calculating values for each of its pixels, it invokes a barrier to wait for all display hosts to finish computation before writing the pixel data to the canvas element.

6.3 U.S. Counties Map

There are over 3,000 counties in the United States[2]. Many statistics are collected on a per-county basis, and a high-resolution display wall would be helpful for visualizing such statistics.

This application, adapted from a D3 example application[1], displays a vector graphic of the outlines of the counties. Clicking and dragging the graphic will pan it, while the mouse wheel zooms in and out.

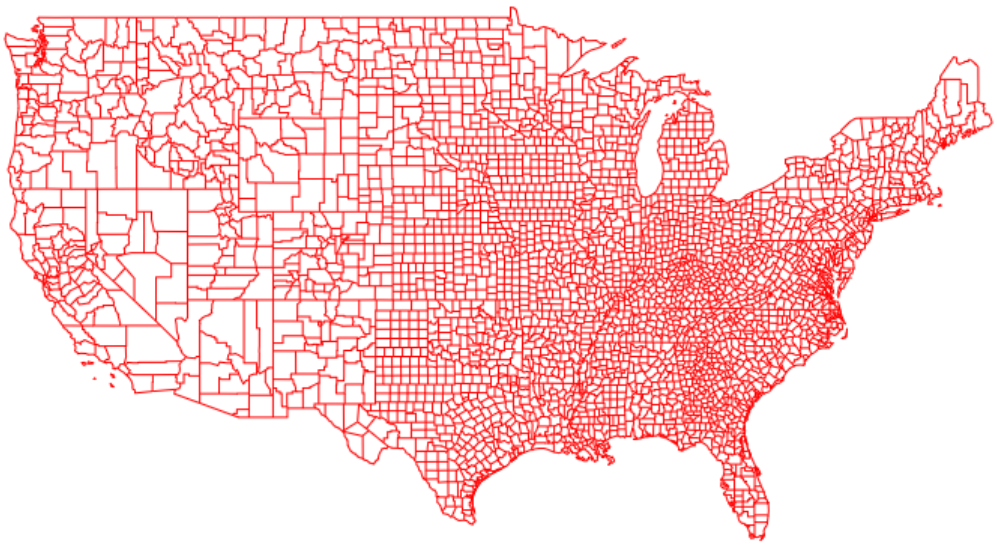


Figure 6.3: U.S. Counties Map

1. http://bl.ocks.org/mbostock/5914438

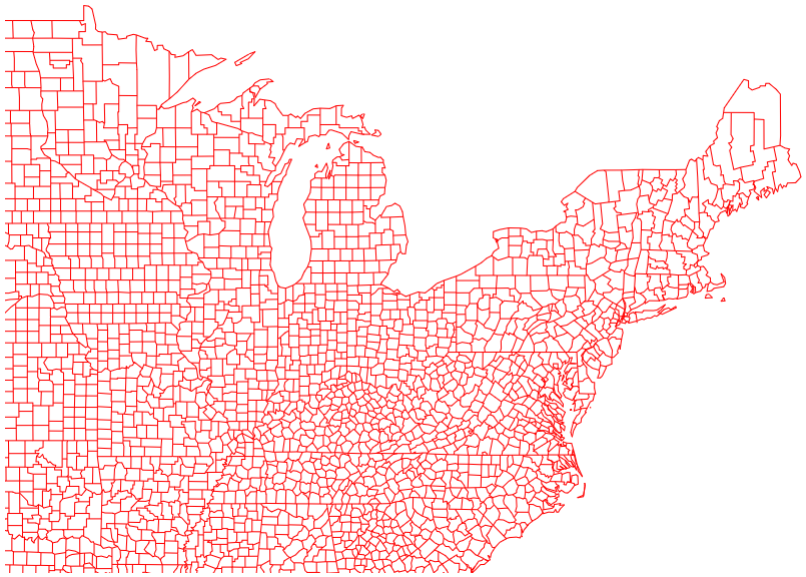


Figure 6.4: U.S. Counties Map zoomed in on the Northeast region

6.4 Hi-Res Gallery

This application is meant to showcase the flexibility and power of the replicate and zoom model, and the ability to treat the entire display wall as one large web page. It is a simple gallery of high-resolution .jpg images obtained from NASA's Jet Propulsion Laboratories/Caltech[2]. When launched on the display wall, the images will span across the entire wall, left to right, top to bottom. When launched on a typical personal computer (that isn't behaving as a tile), the web page will arrange the images downward in a column that the user can then scroll down as a normal-looking web page.

All that's happening behind the scenes is that JavaScript code detects whether or not the web page has been launched as a tile, i.e. the URL query string is the form "?tile-x-y". If it is a tile, then all it does is resize the document as a whole, then perform the appropriate CSS3 transforms to achieve the correct viewport. The standard HTML layout algorithm then handles the page layout according to the document size it's given.

2. http://www.jpl.nasa.gov/spaceimages/

6.5 YouTube Controller

This application combines the interface of the Wall Controller described above with the YouTube Player API[3]. Clicking and dragging will prompt the user for a YouTube video ID, then load a YouTube Player that will play the specified video.

3. https://developers.google.com/youtube/iframe_api_reference

7 Evaluation

This chapter provides a brief evaluation of various aspects of Browzawall.

7.1 File Size

Table 7.1 lists the file sizes of Browzawall's client-side libraries, as well as the file sizes of the commonly-used jQuery and D3 libraries. Browzawall's libraries are very small relative to them, so transferring them over the network should not significantly impact page load time; nor should retrieving them from the browser's cache. There is still room for potential optimization (if necessary) by combining the Browzawall JavaScript files (to reduce the total number of requests) and/or using a JavaScript minifier.

File	Size (KB)
comm.js	9
walldoc.js	6
domsync.js	7
dd3.js	6
jquery-2.1.4.min.js	83
d3.v3.min.js	148

Table 7.1: File sizes

File	Total Phys. Memory (MB)
(single blank tab)	92.4
memtestbase.html	103.0
memtest.html	102.3
wallcontroller.html	103.3

Table 7.2: Client Memory Usage

7.2 Client Memory Usage

Client memory usage was tested for the Wall Controller application, and compared against two "skeleton" applications as a reference point. The skeletons contained only basic HTML structure, and were each less than .5 KB in size. One, named "memtest.html" loaded the three core Browzawall libraries (comm.js, walldoc.js, and domsync.js), the other, named "memtestbase.html", loaded no external Javascript. (Note that simply loading the comm.js library will initiate the WebSocket connection to the communication server.) Wall Controller, or wallcontroller.html loads not just the Browzawall libraries, D3, the D3 colorbrewer library, and a library for generating gene pathway diagrams.

Measurements were taking with Firefox version 40.0.3 on a Windows 8 laptop, using the physical memory statistics provided by Task Manager. Each web pages was loaded by a fresh Firefox process that hadn't loaded any other tabs (except for the blank tab as the "home" page). As this measurement was intended to measure the baseline memory usage of Browzawall, the measurements were recorded once the page had loaded but before any user-initiated actions had taken place. Table 7.2 shows the results. While loading any web page appears to have an overhead of roughly 10.5 MB more than a starting blank tab, the pages that loaded the Browzawall libraries didn't use significantly more memory than the page that did not.

7.3 Responsiveness

Browzawall is the integration of display walls and web-based visualization, which are essentially two different user interface technologies. As such, responsiveness is the most important metric. We have used the Pathway Controller application described in the previous chapter to observe the responsiveness of clicking and dragging pathway diagrams to reposition them on the display wall. There was little to no visible lag on the display wall whether the Pathway Controller used custom messages, DOM Synchronization, or Distributed Event Listeners to synchronize the position of the pathway diagram.

7.4 Distributed D3

In order to evaluate Distributed D3, I turned to the example applications linked on `http://d3js.org`, with the intent to adapt them to function on a cluster-based display wall. However, all of these applications that I explored relied on features beyond the basic D3 Selection object, e.g. Transitions, and thus were not supported by the prototype Distributed D3. It will take further effort to determine if it's feasible and worthwhile to implement Distributed D3 for the entirety of D3.

7.5 Security Concerns

Any users of Browzawall should be aware of security concerns. While the typical security concerns of web applications apply, of particular note is the fact that Browzawall provides several methods of executing JavaScript code in a browser window on remote machines. These remote machines may be behind firewalls or otherwise have higher access privileges. Or, by executing code on all hosts that are part of a display wall, a malicious user could effect a Distributed Denial of Service (DDoS) attack. Adding encryption and authentication is an opportunity for future work.

Without encryption and authentication, a malicious user could simply connect to the communication server and flood it with messages. If other display hosts are connected to the same session, they would each also receive this traffic. Depending on the application and the messages sent, this could cause the browser window to create new Elements, and thus eat up memory.

Session-persistent shared objects currently leave the communication server vulnerable to excessive memory usage. The server does not enforce any limits on the number of such objects or their size, so a malicious user could cause the server to consume memory up to the limit allowed by Node.js.

Browzawall inherits any security vulnerabilities from the open-source or "off-the-shelf" software it relies on. Currently this includes Node.js, and whatever web server, web browsers, and operating systems it's installed with.

7.6 Usability

Any framework should be evaluated on its ease of use. This is one area where more work is required. With clear API documentation, other web application

developers can try building their own Browzawall applications.

7.7 Comparison to Other Solutions

Section 2.4 introduced existing solutions for displaying interactive web applications on cluster-based display walls.

Browzawall has several benefits over Multibrowsing[8]. Browzawall is capable of a greater degree of abstraction than Multibrowsing, since it can treat several displays, even across multiple display nodes, as one large display. Browzawall also benefits from the smoother performance that the WebSocket protocol offers, and its ability to push data to browsers without requiring the browser to completely refresh the page.

Hydrascope[7] has the ability to scale up and synchronize applications without modifying existing source code, though it relies on reverse-engineering, which can result in brittle solutions.

VNC and the Display Cloud[6][11] allow display of web applications from the VNC server's desktop to a cluster-based display wall. This is an incredibly versatile solution, as it can display not just web applications, but any applications on the VNC server's desktop. This is, however, subject to available bandwidth. Browzawall has an advantage in that it is able to transmit events, rather than just pixels. Transitions, such as menus sliding out, create multiple pixel changes from a single event. We have observed brief artifacts in such situations when using VNC.

One of Browzawall's largest weaknesses relative to other solutions is that it requires retrofitting client-side JavaScript code into existing web applications, or building new applications from scratch. However, many or most existing web applications and pages were designed for normal desktop or mobile resolutions, and do not produce any additional benefit on higher resolution displays. See Figure 7.1, where the dagbladet.no home page has enough horizontal space as the red bar at the top, but only uses a small portion of that space.

Figure 7.1: Displaying the dagbladet.no home page on the Tromsø Display Wall.

8

Conclusion

This thesis has described the challenges inherent to integrating web technologies with cluster-based display walls, and has presented an overview of possible solutions. This thesis has also described Browzawall, a framework providing several methods for addressing those challenges, which application programmers can use according to what is most appropriate.

9 Future Work

This chapter suggests potential future work, including improvements to current functionality and integration with other technologies.

9.1 General Improvements

The security concerns in Section 7.5 could be addressed by implementing the WebSocket Secure (wss://) protocol. The excess bandwidth and content desynchronization mentioned inherent to "replicate and zoom" (described in Section) could be mitigated by setting up a proxy cache for the display tiles to receive content from.

9.2 Distributed D3

As mentioned in Chapter 7, the current implementation of Distributed D3 is too limited to handle anything but the most basic use cases. Implementing additional features of D3 in Distributed D3 would allow us to better explore the potential of this approach.

9.3 Shared Motion

Browzawall was designed for scientific visualization applications in which the display would not be changing frequently, and thus even latencies (between display tiles) close to a second would be acceptable. For finer-grained synchronization on a display wall, Browzawall could be integrated with Shared Motion developed by Motion Corporation[1]. With Shared Motion, it may be possible to play high-resolution HTML video content on a display wall.

9.4 Gesture Support

The Tromsø Display Wall has a motion-capture system that detects motion in front of the display screen[1]. A future project could supply this motion data to the communication server and display hosts in order for applications to be controlled by gestures.

1. http://mcorp.no

Bibliography

[1] O Anshus, Daniel Stødle, T Hagen, Bård Fjukstad, J Bjørndalen, L Bongo, Yong Liu, and Lars Tiede. Nineyears of the tromsø display wall. In *Proceedings of Powerwall, SIGCHI Workshop*, 2013.

[2] U.S. Geological Survey/Audio by Steve Sobieszczyk. "how many counties are there in the united states?". `http://gallery.usgs.gov/audios/124`, 2008. Accessed 1-October-2015.

[3] Ian Fette and Alexey Melnikov. The websocket protocol. 2011.

[4] Bjørn Fjukstad. Kvik: Interactive exploration of genomic data from the nowac postgenome biobank. 2014.

[5] Bjørn Fjukstad, Karina Standahl Olsen, Mie Jareid, Eiliv Lund, and Lars Ailo Bongo. Kvik: three-tier data exploration tools for flexible analysis of genomic data in epidemiological studies. *F1000Research*, 4, 2015.

[6] Tor-Magne Stien Hagen. Interactive visualization on high-resolution tiled display walls with network accessible compute-and display-resources. 2011.

[7] Björn Hartmann, Michel Beaudouin-Lafon, and Wendy E Mackay. Hydrascope: creating multi-surface meta-applications through view synchronization and input multiplexing. In *Proceedings of the 2nd ACM International Symposium on Pervasive Displays*, pages 43–48. ACM, 2013.

[8] Brad Johanson, Shankar Ponnekanti, Caesar Sengupta, and Armando Fox. Multibrowsing: Moving web content across multiple displays. In *Ubicomp 2001: Ubiquitous Computing*, pages 346–353. Springer, 2001.

[9] Mozilla Developer Network. "document object model (dom) - web apis | mdn". `https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model`, 2015. Accessed 1-October-2015.

[10] The Chromium Projects. "multi-process architecture - the chromium projects". http://www.chromium.org/developers/design-documents/multi-process-architecture, 2008. Accessed 1-October-2015.

[11] Lars Tiede, John Markus Bjørndalen, and Otto J Anshus. Cloud displays for mobile users in a display cloud. In *Proceedings of the 14th Workshop on Mobile Computing Systems and Applications*, page 12. ACM, 2013.