The W12 Network Window System

Julien Lord

School of Computer Science
McGill University
Montreal, Canada

February 2012

A thesis submitted to McGill University in partial fulfillment of the requirements for the degree of Master of Science.

Dedication

For family and friends.

Acknowledgments

I would like to thank my supervisor, Professor Muthucumaru Maheswaran, for all his advice and guidance throughout the development of this project. Without him, this project would not have been possible.

I would also like to thank Derek Ingrouville for providing the basis upon which W12 was built and Lucas Terra for proofreading.

Finally, I would like to thank my parents, Denis Lord and Susan Copeland, for their encouragement and also for their help in editing and proofreading.

Abstract

Mobility is quickly becoming the normal usage pattern for application developers. How a user accesses a given application changes depending on present circumstances. User applications can exist on the local machine, on a web server, in a cloud infrastructure, or on a mobile phone. Deciding which platforms to support can be the difference between a successful application and one that doesn't get off the ground.

The *W12 Window System* is an attempt at solving the mobility problem. Each platform shares one ubiquitous piece of software: the web browser. When a new platform is developed, typically a web browser is among the first fully-formed applications released. W12 leverages HTML5 web browsers as thin-client computing terminals to display an application that can be running anywhere over the network. Windows are drawn into the browser and events can be sent back to the application for processing.

Résumé

Les utilisateurs d'applications informatiques ne sont plus restreints à leurs ordinateurs comme avant. Le mode d'utilisation d'une application peut changer selon les circonstances de l'usager. L'application peut être exécutée sur un ordinateur local, sur un serveur Web, dans un environnement d'infonuagique ou sur un téléphone mobile. La décision de la plateforme informatique sur laquelle une nouvelle application sera développée peut, à elle seule, décider du succès ou de l'échec du projet.

Le système *W12*, un environnement graphique en réseau, vise à régler les problèmes reliés aux multiples plateformes informatiques. Chacune des plateformes mentionnées partage un aspect en particulier : le navigateur Web. Le navigateur Web est souvent une des premières applications développées pour le lancement d'une nouvelle plateforme informatique. Le système W12 utilise la nouvelle standard Web HTML5 pour transformer un navigateur Web en un client léger pour permettre l'affichage d'une application peu importe où elle est exécutée. Le navigateur Web affiche le rendu de l'application et les évènements, tels ceux générés par le clavier et la souris, sont acheminés à l'application.

Contents

List of Figures

Chapter 1

Introduction

Application developers are faced with a wide range of available platforms. From Android and iOS for smartphones and tablets to Windows, Linux, and Mac OSX for desktops and laptops. Actively supporting more than a single platform can be a difficult challenge. Users further compound this by expecting their applications to follow them, personal data and all, across these platform boundaries. This thesis describes a new window system, the *W12 Window System*, which provides mechanisms to tackle these challenges.

1.1 Motivations

Developing applications for multiple platforms can be approached from two distinct angles. First, a core application can be developed for a target platform. Additional platforms can be supported by *porting* the core application to additional platforms. Depending on the platforms being targeted, this can mean recreating the entire application from scratch. The second approach is one whereby an application is designed

by separating its core functionality from its user interface. The core functionality can run within a dedicated environment, on a single platform, while the user interface can be presented to the user on the platform of her/his choosing. This approach is taken by standard web applications which develop their core functionality on the server and present the user interface as a separate entity, usually a web page. This method has the disadvantage of decoupling the application design from the user interface design. Applications need to supply their functionality in terms of services to which the HTML UI makes requests over HTTP.

Adding to the problem of platform cross-compatibility is the sheer number of devices that end-users have access to. Computing is no longer limited to computers. Tablets and mobile phones are readily accessible and increasingly powerful. Users often need applications designed for specific devices. "Mobile" versions of applications sometimes exist, but these are simply new versions of existing applications which aim to provide similar functionality on a different platform.

This thesis proposes an alternative approach, using a networked window system dubbed *W12*. This system aims to allow application developers to create rich user interfaces displayed within a web browser. In doing so, W12 maintains platform independence by design and provides a single application version usable on any type of device.

1.2 Contributions

This thesis contributes a new HTML5-based network window system implemented using C, Python and JavaScript. To the author's knowledge, it is the only such system using this combination of languages and the only such system to use the JSON message

format for communicating to the browser. The demonstration application outlined shows that W12 is clearly a feasible option for remote application deployment.

1.3 Topics Discussed

Before presenting W12, Chapter 2 discusses various window systems, both historical and current. Chapter 3 discusses the design and implementation of the various parts making up the W12 system. Chapter 4 outlines different methods for deploying W12 applications. Chapter 5 presents performance testing results. Chapter 6 outlines some systems closely related to W12. Chapter 7 discusses conclusions and potential directions for future development work on W12.

Chapter 2

Background Information

This chapter describes various existing window systems. Historically significant systems as well as current systems are described. Communication protocols designed for remotely viewing displays are also presented.

2.1 Network/Extensible Window System

The Network/Extensible Window System (NeWS) was developed in the 1980s by James Gosling and David Rosenthal of Sun Microsystems. Originally called *SunDEW* (Sun Distributed Extensible Windows), it was created to address issues perceived in the Andrew and X window systems, both of which the authors had extensive experience with [1]. NeWS display relies on Adobe's PostScript language, with special extensions added to accommodate several clients drawing to a single screen as opposed to a single printer (client) drawing on a sheet of paper, as PostScript was initially designed to do. By using the PostScript imaging model, NeWS allowed very complex imaging operations using complex paths, fills, and clipping planes. For instance, it's

relatively simple to define windows of arbitrary shapes, something rather rare even in modern window systems. This flexibility comes at the cost of pixel precision. Unlike X, which uses the pixel as the base unit, PostScript uses the *point* as the basic unit (roughly 1/72"). This means that draw calls have no real guarantees of which pixels will be painted.

The NeWS architecture is similar to other networked window systems. The NeWS server resides on the display machine with client applications communicating over a reliable communication channel (usually TCP/IP) [1]. Where NeWS differs from other window systems is its use of the PostScript language at the server level. The server supports all standard PostScript operations as defined by Adobe as well as customized extensions to allow Object-Oriented programming and multi-threaded operations [1]. Generic objects were added to represent canvases, lightweight processes (LWP), concurrency monitors (i.e. locks), events, and colors. A reference-counting garbage collection system was also added [1]. Canvas objects represent a window on which client applications can draw. LWPs allow multi-processing without the overhead required by full-fledged system threads. The NeWS display server achieves the LWPs by running the entire server within a single addressing space. Monitors are necessary to allow multi-threaded operations to prevent concurrency issues. Events are used both as generic messages passed between processes and external devices as well as templates against which incoming events can be compared to see if a client is interested in a specific event (i.e. to register listeners). Color objects extend the basic PostScript color model to give more flexibility. Messages passed between client and server are formatted as actual PostScript commands, with some compression in place to reduce bandwidth requirements.

Having a full interpreter on the display server allows NeWS applications to be written in three different styles [1, p. 39]:

1. Written entirely in PostScript and residing in the single address-space, multiple process world of the NeWS server. Similar to the Xerox PARC system environment.

2. Access the system using raw PostScript imaging operators and NeWS input operators. Similar to the X11 window system, but with a more powerful imaging model.

3. Access the system using a server-resident toolkit. Similar to systems like Andrew [2] with much higher-level operators, in which the network communication is in terms of objects such as menus and scroll bars.

In the first style, applications are downloaded entirely to the application server and run inside the interpreter only. This is much less network-oriented and is therefore a less interesting paradigm. In the third style, application programmers need to find an adequate toolkit provided by a third party. The developers of NeWS provided a toolkit called the Lite Toolkit but this was a very simple toolkit with little customization possible.

The second style, while allowing application programmers complete flexibility, relies on a special preprocessor tool, called *cps*, which would create the interface between the application and the NeWS server. The cps preprocessor takes as input a specification file which associates a PostScript code snippet to a C-style function declaration. The cps preprocessor outputs a C header file for the application to use and a PostScript programme for the server to load. The typical client-server

communication was as follows [1, p. 148]:

1. Establishing a connection to the server.
2. Sending PostScript code (downloading) to the server.
3. Invoking the PostScript code downloaded.
4. Sending replies from the sever back to the client.

While this made for a very flexible model, allowing arbitrary PostScript to be executed on the display server from within a client application, it also made for some unwieldy code. Defining custom PostScript methods required the creation of both a setup function and an execution function within the application code. The setup function had to be called before the execution function could be used. This did however lead to clever implementations allowing multiple arbitrary combinations of PostScript calls from within the application code [1, pp. 151-152].

```
12  cdef ps_newframe(x, y)
13      /current-x x store
14      /current-y y store
15      /paintclient win send
```

Fig. 2.1 Example cps code. Methods defined in this manner are available within a C program with the associated PostScript code being executed on the NeWS server when the C method is called [1, p. 187].

By far the most powerful feature in NeWS was its inherent ability to offload processing to the display server. By allowing arbitrary PostScript to be loaded and executed on the display server, event handlers could be coded that required no network roundtrip. Events generated on the display server (e.g. by a mouse or keyboard)

could be dealt with locally without need to propagate the event back to the client application. This can lead to significant performance gains in certain situations where several events are generated in quick succession, like when tracking mouse movement.

NeWS had difficulty gaining market share with the X window system gaining in popularity. A merged X11/NeWS server was proposed which would be able to connect with both NeWS and X11 applications on a single display server. The merged server failed to gain any traction in the market and Sun eventually dropped NeWS entirely. Exact reasons for NeWS's failure are speculation but license implications most likely played an important role. X was distributed freely and openly, that is free of charge and free of restrictive licensing, whereas NeWS was not. Also to NeWS's disadvantage was the lack of available libraries for specific language bindings. The NeWS team supplied only the Lite Toolkit and the cps preprocessor, expecting third-party developers to create additional toolkits and language bindings. Without these tools, application developers were required to code not only the application logic but also the PostScript code which was needed for the display server.

2.2 Adobe Display PostScript

The Display PostScript (DPS) system was developed by Adobe as a means to use the PostScript language for displaying on a bitmapped computer screen. Unlike NeWS, DPS was not a complete window system. It relied on an underlying system to accomplish its goals. A standard X extension was created as well as the Display PostScript NX package, which allowed a X window implementation lacking the DPS extension to correctly display DPS commands.

Like those created for NeWS, DPS added several extensions to the PostScript

language. Mechanisms were created for dealing with multiple execution contexts to allow multiple windows to be drawn to simultaneously. These context mechanisms are closely related to the LWP concept of NeWS. To reduce memory usage and improve performance of the system, DPS added means for encoding variable and method names as integers rather than character strings. Mechanisms were also devised for dealing with incremental screen updates and dealing with scrolling windows. Like in the NeWS cps preprocessor, arbitrary PostScript was also possible using a tool called pswrap. The tool allowed wrapping PostScript commands inside C function calls.

Architecturally, it's important to note that DPS did not deal with any network issues whatsoever. It relied on the underlying window system to accomplish this communication. Figure 2.2 shows both methods by which DPS could interact with an existing window system to display graphics.

DPS also relied on the underlying system for event management. It did not provide any additional means for detecting or processing events. Application programmers were required to use the underlying window system event management facilities to detect keyboard, mouse, and other events.

2.3 The X Window System

The X Window System, simply referred to as X, was developed at MIT primarily by Robert Scheifler, James Gettys and Ron Newman [4]. Initial development started in the 1980s [4] but continues today as an open source project led by the X.Org Foundation. X is considered the most widespread window system among the various Unix distributions. Versions are also available for Windows and Mac OSX, though they are less prevalent. At the time of writing, the current version is X11 Release 7.6,

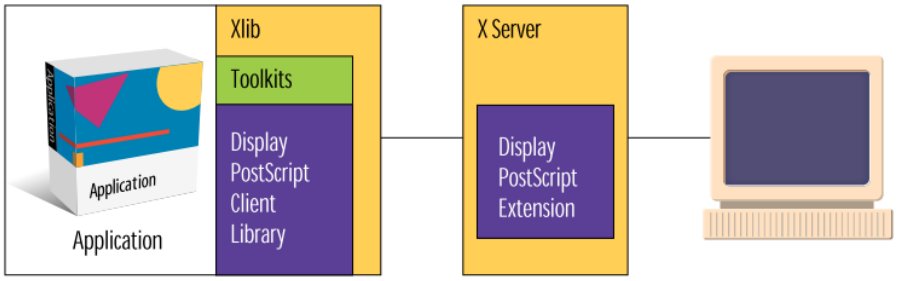

Figure 2. Communicating with the Display PostScript extension

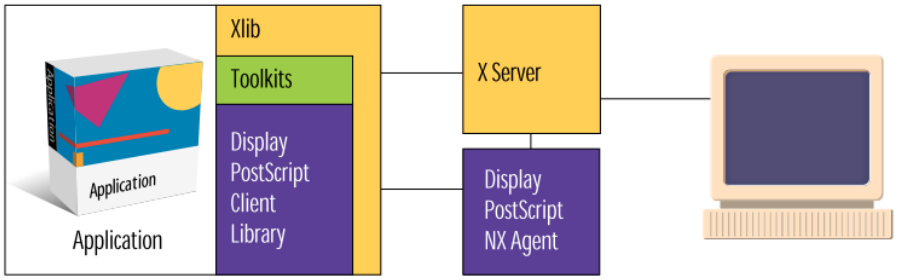

Figure 3. Communicating with Display PostScript NX software

Fig. 2.2 DPS had two separate systems for communicating with the display [3].

with Release 7.7 planned for “the second half of 2011”.

X was initially designed for distribution across MIT’s computer laboratories. The labs featured several thin-client terminals with a few powerful servers providing applications to them. First and foremost, X was designed to be network-independent. Applications and the display they drew to did not need to be on the same machine. Also, varying hardware across the many terminals meant that X needed to be hardware-agnostic. To these ends, X was designed as a client-server system with a specific communication protocol, the X Protocol, used to communicate between the two. Somewhat confusingly, the X server process runs on the thin-client terminal while the applications making use of the window system are considered the clients. Clients use the Xlib API to communicate with the server. Rather than impose specific styles on its users, X leaves most of the window management work up to the developers. This has led to the creation of a few different toolkits, among them GTK+ and Qt, which provide window management on top of X.

The X protocol is asynchronous, requiring only a “reliable communication” method. In most cases, TCP/IP is used though any protocol with reliable delivery would be in line with the specification [5]. This is important when considering applications being used across potentially slow network links. Clients send *requests* to the server and in most cases expect no reply. Of the 120 requests available to be called by clients, only a third of them generate replies. This reduces the number of network round-trips and alleviates latency-related issues. Similarly, *events* generated at the server, for example keyboard or mouse interaction, are usually only sent to clients which have expressed an interest in the event by registering an event handler. Some events like the EXPOSE event are sent regardless [5]. This eliminates the need for clients to poll

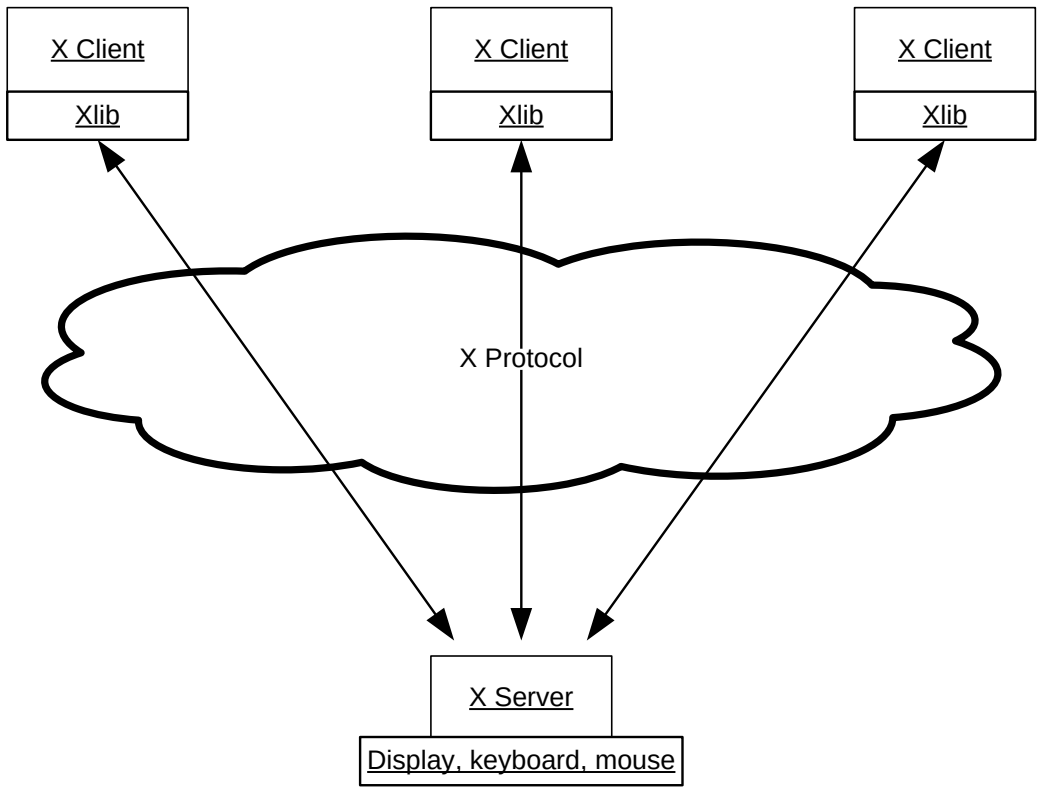

Fig. 2.3 X Window System architecture.

the server periodically to get events. There are roughly 30 classes of events that can be generated within X [5].

The X protocol operates by way of binary messages. These messages are separated into four categories [5]:

1. Request messages, sent by clients to servers, containing at least 4 bytes:
   - 1-byte Major OpCode
   - 2-byte Message Length
   - 1-byte Additional Data
   - A sequence number is implicitly given to each message once it arrives at the server
   - 0 or more data bytes depending on the request. The Message Length determines how many such data bytes are present in the request.
2. Reply messages, sent by servers to clients in response to a request, are at least 32 bytes long and contain:
   - 4-byte Message Length
   - 0 or more data bytes (these are present after the initial 32 byte reply meaning that the total length in bytes is 32 plus the Message Length)
3. Event messages, sent by servers to clients, are exactly 32 bytes long and contain:
   - 1-byte Type Code
   - Up to 31 additional data bytes, varying depending on the event

4. Error messages, sent by servers to clients when requests have failed, are exactly 32 bytes long and contain the following:

    - 1-byte Major Opcode of the failed request
    - 1-byte Minor Opcode of the failed request
    - Up to 30 data bytes giving additional information about the error

It should be noted that the X protocol explicitly states that unused bytes within messages are not guaranteed to be zero.

The flexibility of the X window system comes from its previously discussed lack of style imposed and also from X's Extension system. Each of the four major communication types reserves the upper range of its values for extensions. Minor Opcodes are also defined for extensions to use as they see fit. Extension developers are free to add additional capacity to an X implementation by way of these reserved messages. The Display PostScript system made use of this extension mechanism. This flexibility comes both as a blessing and a curse. Not imposing a specific style means that in most cases no two X applications share a common user interface [6]. When using multiple different applications at once, this can get confusing quite quickly. Toolkits like those mentioned above try to standardize look and feel to alleviate this. The Extension system also suffers from some problems : using an extension requires recompiling the system. Not only is this inconvenient (even impossible) for non-technical users, it can quickly lead to portability issues for application developers. Applications looking at broad distribution cannot rely on extensions necessarily being present on a given server implementation.

2.4 Virtual Network Computing

The Virtual Network Computing system (VNC) uses the Remote Frambuffer Protocol (RFB) to achieve remote graphical user interface computing. In the RFB client-server model, the client computer has the display, mouse, and keyboard and receives graphical updates from the server [7]. As might be implied from the name, RFB transfers only graphical updates to the client, essentially sending its local frambuffer over the network. The client side is only tasked with drawing the updates on its screen and relaying mouse and keyboard input to the server. Communication is expected to occur over a reliable channel, usually TCP/IP.

The core of the RFB display protocol is the single primitive concept *put a rectangle of pixel data [of given height/width] at a given x,y position* [7]. The client arbitrarily makes requests for screen updates to which the server responds by sending a series of these *draw rectangle* commands. This method of requesting updates allows clients to dictate their frequency and, as such, to regulate network traffic. Slower network connections will necessarily lead to lower number of requests being sent because of the transfer time required to send the reply. This makes the server free to send only key frames rather than updates for inconsequential/transient display changes. The input protocol for RFB states only that keyboard, mouse, or other synthesized I/O events (e.g. pen-based hand-writing recognition tool outputting artificial keyboard presses) are sent directly to the server as soon as they are received by the client [7]. There is no mention of any additional processes put in place to aid in transmitting user input. Highly interactive applications, requiring lots of user input, usually have trouble coping with high latency over RFB. RFB does, however, allow some leeway in how the server sends graphics update data back to the client. Several different encodings are

supported by default, from simple raw rectangle data to more complex schemes [7]. RFB also supports an extension mechanism to allow developers implementing the protocol to add their own arbitrary encoding schemes for graphics update.

RFB aims to be as simple a protocol to implement as possible and it achieves this goal quite well. While only incorporating simple security measures, like other protocols, RFB's security can be made more robust by tunnelling its connection over SSH. In this sense, RFB can be seen as being fairly complete. However, it does require both server and client components to be available and installed. This means a new platform requires a new client in order to connect to an existing server.

2.5 OpenXUP

The OpenXUP system, proposed by Yu et al., is based on the Extensible User Interface Protocol (XUP) which is itself based on the Simple Object Access Protocol (SOAP) [8]. The main goal of OpenXUP is to provide a means to display remote applications in a web browser while maintaining the "feel" of a local application. Specifically, OpenXUP allows incremental updates, freeing applications from the typical slow and tedious web page refresh between operations [8].

The OpenXUP system is separated into two components. The server component resides within a typical web application server and the client component resides within a web browser. The server acts as a bridge between an existing web application framework and an OpenXUP user interface (i.e. the OpenXUP client). The client portion renders the UI according to an XML markup document provided by the server. OpenXUP supports several such markup languages, among them XML User Interface Language (XUL), Extensible Application Markup Language (XAML), User

Interface Markup Language (UIML) and Simple User Interface Language (SUL). The client also takes user input, converts it to appropriate XML format and sends it to the server. It receives UI updates from the OpenXUP server. Events received at the server are mapped to back-end processing functionality of the existing web application. In simple terms, OpenXUP takes an existing web application and replaces the HTML/JavaScript front-end with a more "native-feeling" and "more responsive" [8] UI. This is because the client component is a full-blown WinForms based application [9]; "The client may be deployed as a stand-alone Windows program or as a browser plugin" [8].

This is a fairly significant drawback. Typical web applications, despite the failings outlined by Yu [8], are essentially platform-independant by design. Any web browser can access them and get fairly regular usage out of the application. OpenXUP provides better functionality but at the cost of platform dependence. The client is naturally limited in its distributability. While more platform-indepedent Java-based inplementations are planned [9], the Windows-only version is the only implementation available to date.

2.6 GTK+ Broadway

Announced as part of the GTK+ 3.2 release, the GTK Broadway back-end promises over-the-network display of GTK3 applications inside an HTML5 compliant browser. Very little has been formally published beyond blogs posts and demo videos submitted by its developer, Alexander Larsson, a software engineer at Red Hat Software [10]. It relies on HTML5 WebSockets for communication to the browser. On the backend, it seems as though Broadway is streaming the output of GTK3 applications over the

network as sequential image files, possibly PNG files. Details are still quite vague on the exact mechanisms being used. Demo videos show standard GTK applications like gedit and GIMP running inside web browsers with normal mouse and keyboard interaction [10]. No mention is made of actual rendering occuring inside the client browser, which leads us to believe that the rendering is done at the server level and communicated directly to the browser, similarly to VNC. It is not clear how events are sent back to the server as this isn't addressed in any of the available information. Demo videos seem to show a certain amount of "lag" in the applications receiving mouse input [10] (GIMP example video).

Chapter 3

W12 Window System

The W12 window system is a three-tier platform. The Wlib client library is the first tier of the system. Like Xlib in the X Windows System, it provides application developers with means to draw UI elements to and receive input from clients. The W-Server is the middle tier of the system. It acts as a meeting point for W12 applications and remote clients wanting to see the output of those applications. The final tier of the W12 system is WJSlib, a JavaScript client library used to render applications inside a Web Browser. Figure 3.1 shows a high-level view of the W12 system.

3.1 Wlib

Wlib is a C library containing all the necessary functions to connect W12-enabled applications, draw output, and receive events. It gives application developers access to drawing primitives that can be used to programatically create the desired UI. In its original concept form, Wlib was meant to be a feature-equivalent port of Xlib but with time this goal fell by the wayside. Xlib (and X11 in general) needs to deal

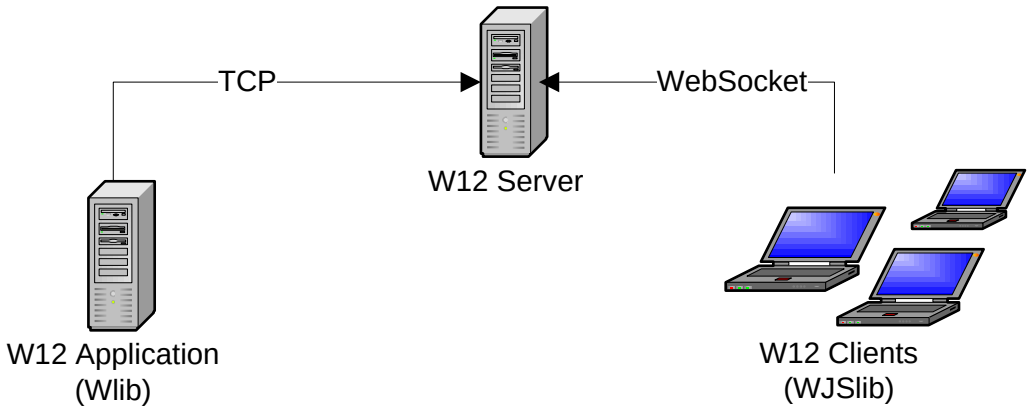

Fig. 3.1 W12 Window System Architecture.

with many concerns which can be ignored in the context of W12. Managing display hardware, colormaps, copy/paste buffers, and other X11 functionality can be ignored since, untimately, W12 applications will be runnning within a system that already has a functional, platform-specific, window system. The web browser that W12 will be displaying through is already running within this window system. W12 need only concern itself with what happens inside this restricted framework. To this end, Wlib as it currently exists is "inspired" by Xlib, but is not a port of Xlib.

The connection between W12 applications and the W-Server is managed by Wlib transparently. Opening a display is done using the W-Server host name and port with Wlib taking care of the underlying connection. Currently, the connection between Wlib and W-Server is done over TCP/IP, but any reliable communication could be used instead. The opened W12 *Display* object keeps this TCP connection alive until the application is terminated. Figure 3.2 shows a typical W12 main application loop.

With the display connected, applications can register callback methods for events in which they are interested. At present, Wlib provides events for handling keyboard

```
250  int main()
251  {
252      Display *display = NULL;
253      char *host = "localhost";
254      int port = 9090;
255
256      display = OpenDisplay("localhost", 9090);
257      if (display == NULL) {
258          fprintf(stderr, "Unable to connect to display %s:%d\n", host, port);
259          exit(1);
260      }
261
262      /* Register Callbacks */
263      RegisterCallback(display, ExposeEventType, expose_event, NULL);
264      RegisterCallback(display, ClickEventType, click_event, NULL);
265      RegisterCallback(display, SetupEventType, setup, NULL);
266      RegisterCallback(display, MouseDragEventType, mouse_drag, NULL);
267      RegisterCallback(display, MouseDragOutEventType, mouse_drag_out, NULL);
268      RegisterCallback(display, Resize, resize, NULL);
269
270      MainLoop(display);
271
272      CloseDisplay(display);
273      return 0;
274  }
275
```

Fig. 3.2 Example W12 application *main* method.

interaction, mouse interaction, window resize, window setup, window expose, font preloading, and file transfer. Data related to these events is marshalled by Wlib and provided to the application in pre-determined formats. Since applications will only be visible once a web browser connects through the W-Server, requests for drawing should only occur within a callback event.

Currently, Wlib does not provide standard widgets. Instead, it follows the Xlib approach in providing drawing primitives such as rectangles, lines, and points. This is done in the hopes that once W12 is fully completed, widget toolkits can be developed on top of it, as is the case of toolkits like GTK+ and Qt existing on top of X. Like in Xlib, W12's drawing primitives use the pixel as their unit of measure. This allows application developers to have pixel-precise control of the output on-screen. Marshalling and network communication of these requests is done transparently and the

communication is done asynchrously. Applications simply call the functions provided by Wlib and continue with their processing.

Appendix A shows the full source of the weyes application, a W12 version of the standard X11 xeyes application. Appendix B shows the Wlib API and explains each of the methods available.

3.2 W-Server

The W-Server is a multi-purpose web application server currently implemented in Python language using the Twisted [11] application framework. On the one hand, it is responsible for hosting static content required by W12 applications, like the WJS client library, and optional files such as CSS and custom fonts. On the other hand, it acts as a controller for W12 application interaction, relaying requests and events between W12 applications and web clients.

W12 applications connect to the W-Server using the TCP/IP protocol. Multiple W12 applications can connect to the W-Server simultaneously. These applications are free to connect and disconnect as they see fit. The server stores no information about connected applications beyond the connection parameters and a session-unique identifier; it is entirely up to the W12 application to manage persistent storage between connections, if necessary.

Web clients connect to the server over HTTP to browse available applications. Connecting to a specific application requires establishing a WebSocket connection using the appropriate handshake mechanism. While the WebSocket specification is still in draft form and likely to change, the current handshake scheme implemented in Google's Chrome v14 browser and supported by W-Server is the HyBi-10 hand-

shake protocol [12]. Once the handshake is complete and the WebSocket connection is established, W-Server acts as relay between clients and applications, forwarding requests to clients and sending events to applications.

Having the server separate from the application provides insulation at the cost of latency. The server adds an additional layer capable of securing the underlying applications if required. Having a single point of connection for clients was also deemed more convenient. Web clients could have access to multiple applications running at multiple different hosts while connecting to a single W-Server. Alternative versions of the W12 system could incorporate W-Server into Wlib to allow stand-alone application deployment.

3.3 WJS Client Library

The WJS Client Library is the client-side analogue of the of Wlib application library. Implemented in JavaScript, WJS makes use of existing JavaScript frameworks including jQuery [13], jQueryUI [14], and Processing.js [15]. These libraries were used to facilitate HTML DOM manipulation and HTML5 Canvas interaction.

WJS is tasked with creating and managing a WebSocket connection to W-Server, drawing Wlib requests on-screen, and generating events based on use interaction. In the X Window System model, WJS acts like the X Server. To provide pixel-based drawing functionality, WJS relies on the HTML5 canvas element and the Processing.js library. The jQuery library is used for DOM manipulation tasks as well as for event generation. The jQueryUI package allows W12 “windows” to be dragged easily within the web browser window.

WJS provides event management for several classes of events. Keyboard interac-

tion can generate KeyPress, KeyType, and KeyRelease events. KeyPress and KeyType differ from each other in the usual sense: KeyPress is a representation of the physical key being pressed on the keyboard while KeyType is a representation of the letter being typed. For example, KeyType events will not fire on special keys such as SHIFT and CTRL whereas KeyPress will fire and KeyPress events will not differentiate between lower-case and upper-case where KeyType events will (i.e. 'a' and 'A' will issue the same KeyPress event but different KeyType events). Mouse interaction can generate MouseMove, MouseClick, MouseDown, MouseDrag, and MouseDragOut events. As expected, MouseMove events fire when the cursor is moved. MouseClick events fire when any button on the mouse is pressed and released. MouseDown events occur when the mouse is pressed down only (and not necessarily released). MouseDrag events occur when the cursor is moved while a button on the mouse is held down. MouseDragOut events are fired if the cursor leaves the W12 application area (e.g. the HTML5 Canvas element) while a MouseDrag is in progress. Resize events will fire if the browser is resized. File transfer events will fire when a file is dropped onto the W12 application area. Finally, Setup and Preload events will fire when the application is first launched to allow intial preparation of the display and loading of external fonts to occur if desired.

WJS provides platform independance for the W12 system. Any HTML5-compatible web browser can use the WJS library to draw W12 applications on screen, regardless of the platform on which the browser is running.

3.4 W12 Communication Protocol

The communication protocol used by W12 is very simple. Each request generates a single message encoded in the JSON format [16]. Wlib function calls take their arguments and wrap them in a two-member JSON object, the first member being a name representing the method being called and the second being a list of arguments for that method. The JSON object is transfered over the network as a string and evaluated inside WJS request handler. The method name is looked up against a list of known methods and, if recognized, executed with the arguments provided. Events generated on the web client side are encoded as simple strings with event parameters separated by spaces. Wlib receives these string messages and automatically parses the various parameters to create Event objects which are passed back to the event handlers. It should be noted that events are only communicated if handlers have been registered for them. Events with no handlers do not generate network traffic.

File transfers operate slightly differently, generating a series of events. When the file transfer event is generated by dragging and dropping a file, an initialization event is fired containing general metadata describing the file being transferred (file name, file size, file type, etc). Subsequently, the file is sent over the network in pieces of 1 MB or less. These file chunks are sent as pairs with the first message of the pair being used to send metadata about the incoming chunk and the second message containing the actual bytes of the chunk being transferred. This 2-message communication model was chosen both for simplicity and also to avoid doing data manipulation within JavaScript. Having an initial, fixed-length messages is easier to deal with at the Wlib level. It allows the size, and subsequently the amount of data to read from the socket, to be known ahead of time. Also, file manipulation APIs in JavaScript are still very

much in their infancy and can be somewhat difficult to work with due to inconsistent implementation and poor documentation. Reducing the amount of data manipulation done at the JavaScript was deemed advantageous because of this. When the transfer is complete, a final event is generated to signal that the file transfer is complete. Note that these file transfer events exist in raw and Base64-encoded forms due to limitations in the current WebSocket API. Binary file transfer, though officially supported, is still rather problematic over a WebSocket connection. As such, WJS attemps to guess if the file is binary or not, based on file type. In the case of binary files, file chunks are Base64 encoded before being transferred with additional metadata to communicate the original as well as the encoded file sizes.

3.5 W12 Message Flow

W12 applications execute following a standard execution pattern. As shown in Figure 3.2, W12 applications first connect their `Display` to the server and register event handler methods. Once this is done, W12 applications must all call the `MainLoop()` method which, as expected, is the main application loop for programs using W12. At this point, the application is ready to draw when called upon. In fact, the W12 application is waiting on its connection to the server and will not execute further until a client web browser connects via the W-Server. Figure 3.3 shows this process as a sequence diagram.

Clients wanting to connect to a W12 application need to connect to the W-Server through a simple HTTP connection. By default, the W-Server will provide a list of connected, available applications. Only when the web client selects a specific application from the list will processing start within the W12 application. The first message

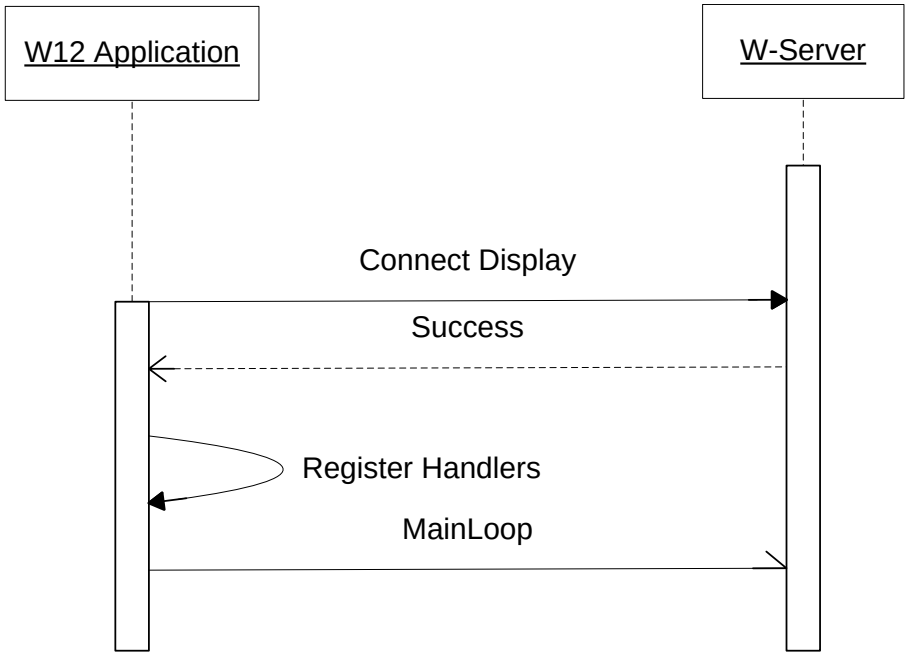

Fig. 3.3 Sequence of events when a W12 application connects to the W-Server.

sent to the application is the `PRELOAD` message. It allows W12 applications to load resources it may require such as fonts and images. The data sent by the W12 application in response to the `PRELOAD` message are used by the W-Server to generate the application template that will be sent to the web client. This template is an HTML file which contains links to the WJS library and to the libraries which it requires, all of which are hosted statically by the W-Server. Future versions of W12 could in fact generate specific versions of the WJS library on the fly. It could be adapted based on the client connecting to it, for instance. This possibility is explored in section 7.1.4. The HTML template also contains links to CSS and Processing.js template files which are generated automatically based on the preloaded data. This allows the look and feel of W12 application to be customized as the developer sees fit. The HTML template also initialises the WebSocket connection using JavaScript. This is done through the standard jQuery use of the `$(document).ready()` function, a system by which JavaScript code can be initiated by the web browser automatically when the content is sufficiently loaded but not necessarily fully rendered. This WebSocket connection is how the web client will communicate with the W-Server. The handshaking protocol, as explained in WebSocket protocol documentation [12], is done automatically by the W-Server. W12 application developers do not need to concern themselves with this process. Once this WebSocket connection is established, the web client can begin to receive requests generated by the W12 application.

Once the preloading is complete, the W-Server sends the W12 application a `SETUP` message, which includes the width and height of the browser window. The `SETUP` message generates up to two responses from W12 applications. The first response from the W12 application informs the web client of the events which it would like

to receive. This is done as an automated process at the Wlib-level. Any events for which listeners have been registered within Wlib will be communicated to the web client. This is done to limit network traffic; events which have no registered listeners do not need to be communicated over the network. The second response to the `SETUP` message is up to the W12 application itself. If the application registered a handler for the `SETUP` event, the registered method will be executed; otherwise nothing is done and processing will simply continue. The setup event allows W12 to do standard initial processing like determining window sizes, setting default draw colors, etc.

After the setup messages are finished, the web browser is ready to receive the full frame. An `EXPOSE` message is sent from the web client to the W12 application, via the W-Server. This is similar in nature to the `EXPOSE` event as described in [5]. Like with the second part of the `SETUP` response, the method registered as the callback for the `EXPOSE` Event is executed. As many requests as the developer wishes may be sent at this point to draw the required user interface. Once the `EXPOSE` handler is fully executed, the W12 application waits for events sent by the web client. W12 applications are free to respond to events as they see fit, sending as many or as few requests as they wish. The communication process between the web client, the W-Server, and the W12 application is shown in Figure 3.4.

3.6 Limitations and Concerns

The overall design of W12 creates some limitations to its adoption. Relying on HTML5 elements is tricky as the specification is in a state of constant flux. As an example, during W12's development, Google's Chrome browser, the main platform used in testing, changed its WebSocket handshake method from the Hixie-76 protocol to the

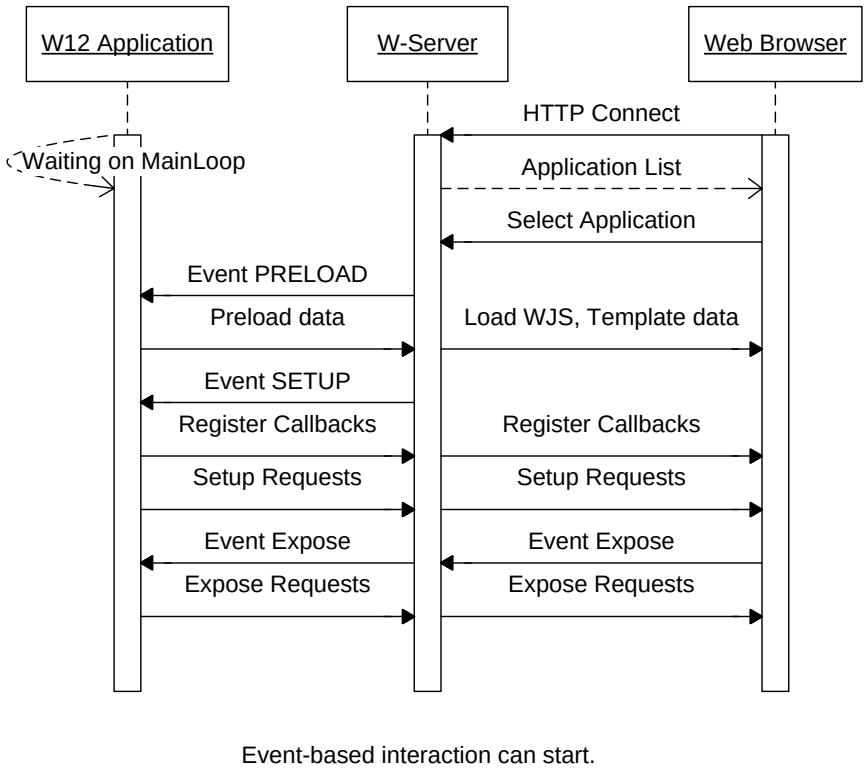

Fig. 3.4 Sequence of events when a web browser connects to a W12 application.

Hybi-10 protocol [17], disabling the entire system. Until the HTML5 specification is finalized, it's fair to assume that W12 will need to be periodically changed to accomodate the HTML5 specification. Also, adoption of HTML5 is still not complete across all major browsers. Certain features on which W12 relies may not be implemented as expected in a given web browser. To a certain extent, the W-Server can be used to alleviate this. A system could be devised to detect browser capabilities at connection time and adapt W12's behaviour to the client connecting to it.

Security is also a major concern. Currently, W12 runs over HTTP and WS protocols but it could be adapted to use HTTPS and WSS protocols to increase connection security. While the W-Server does insulate W12 applications from potentially malicious users, it is by no means a failproof system. Mechanisms can and should be put into place to monitor connections at the W-Server level to detect/prevent malicious users from executing attacks on W12 applications. Relying on JavaScript can also be considered a security concern because the code is sent entirely to the web client to be interpreted locally. While there are methods in place for making the code more difficult to read [18], these methods are aimed at reducing size and increasing speed, not at security. There are no guarantees that the code being run on client-side has not been tampered with in some way. Research papers such as [19] and [20] examine how JavaScript code can be made to run securely in the browser. Such methods are beyond the scope of W12 at present. This is further examined in section 7.1.1.

Chapter 4

Deploying W12 Applications

This chapter explains methods for deploying W12 applications. Different scenarios are discussed as well as how different deployment schemes can be leveraged effectively.

4.1 W12 as a Networked Display System

These methods use W12 as a remote display protocol. They allow applications to decouple processing logic from user interface. This allows applications to present their UI to remote users natively.

4.1.1 Cloud-based Deployment

The standard deployment, as envisioned during the design stages of W12, is one where W12 applications are available from cloud computing frameworks. In this scenario, shown in Figure 4.1, W12 applications and the W-Server run within the cloud. Clients wanting to use those applications point a web browser to the publicly visible W-Server component and are given a list of available applications. In this scenario, clients are

free to use any web browser they wish, from Safari running on an iPad to FireFox running on a laptop computer. The physical machine on which the W12 application is running does not need to be made publicly available. The W12 application only needs a single TCP port to communicate with the W-Server. In fact, the W12 application need not necessarily be executed within the cloud. It could be be running from an arbitrary location from which it can reach the W-Server over TCP/IP. The W-Server allows applications to connect and disconnect at any time, so a single W12 application could be running on different physical machines with no impact on the end user. If a failure occurs, a W12 application's data could be transferred to a new machine and the application could be restarted with little downtime.

This scenario makes use of any devices as a thin-client. Very little data is held on the actual device and the bulk of the processing and data storage is being done within the machine running the W12 application. Handling device failure becomes trivial. If the client's device fails, for example if the battery on a user's laptop runs out, she/he can reconnect from another device, like a smartphone for instance, and reconnect to the same application immediately with no loss of data.

In this scenario, the W-Server is not required to be independent of the W12 application. Both can just as easily be bundled within a single machine.

4.1.2 Mobile Application Projection

W12 can also be deployed entirely locally, with both the W-Server and the W12 applications connected to it running within the same machine. The W-Server may then project applications either locally or over a network. In the context of a personal computer, this can allow W12 to be used as an application framework of sorts. Multiple

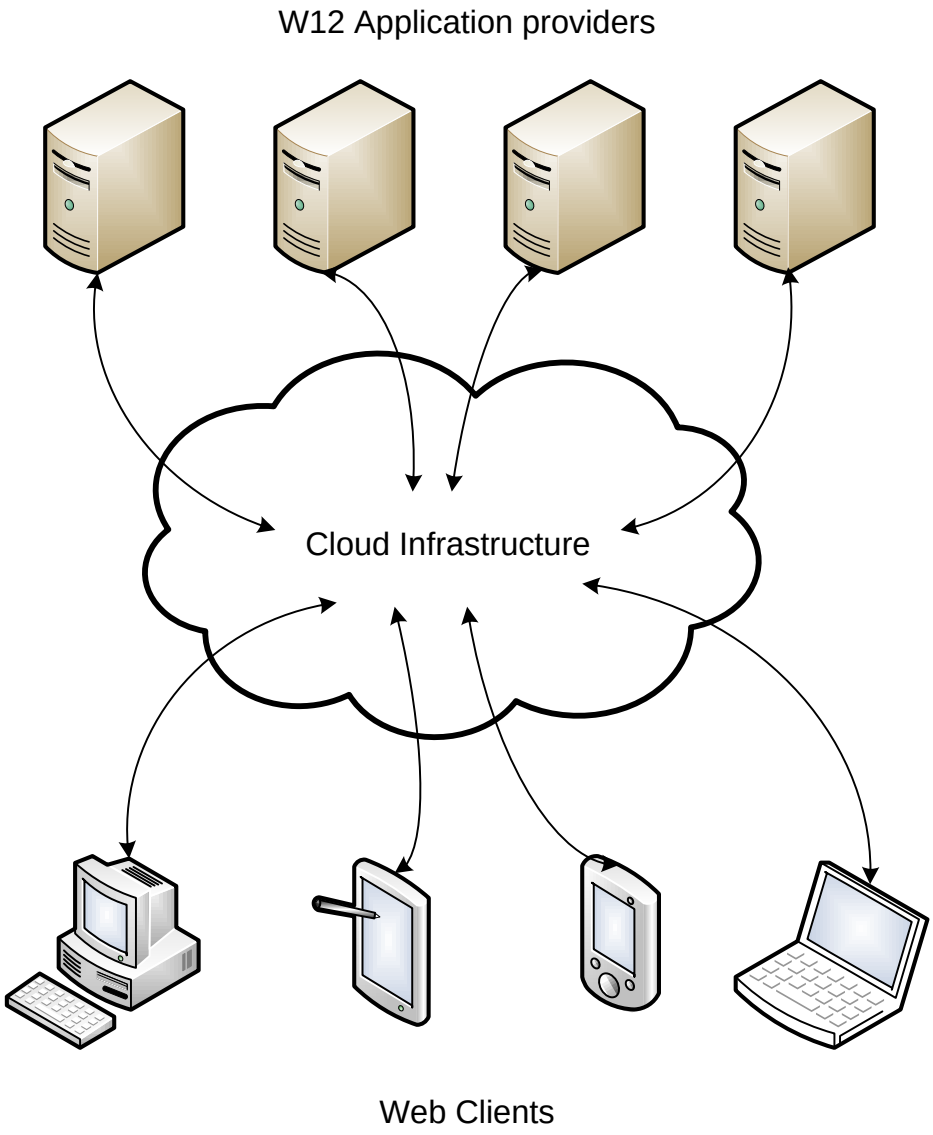

Fig. 4.1 Diagram showing W12 deployed to a cloud infrastructure.

applications be can be deployed within W12 with little additional installation required on the machine. This is similar to the above network scenario with every computer in the network potentially having its own local copy of the W12 applications.

This scenario becomes more interesting in the context of mobile devices. Application projection would require a version of the W-Server running on the mobile device. W12 applications would run on the device as well, connecting to the local W-Server. Users on the go could connect to their local applications through the device's web browser. This is already fairly standard practice in the real world with many mobile applications being implemented as a simple web-viewer widget that loads a local web page. Beyond this local application usage, the W-Server allows mobile device users to project the user interface of their application to any other terminal. By connecting the mobile device to a computer, for example, the web browser on that computer becomes the application user interface, bringing with it any attached peripherals like a keyboard and a mouse. Touchscreens are adequate at best for short messages but typing long documents on them is a tedious, error-prone affair. The ability to connect to screen and keyboard of any computer can greatly increase the usability of many applications, especially typing-intensive applications like text editor or those requiring more screen real estate, like spreadsheets. W12 stores no local information within the web browser on the device on which it is running, so disconnecting the device would leave no traces of the application behind.

4.2 W12 as an Information Overlay System

A novel deployment scheme could be for W12 to be used as an overlay system to heighten existing web content. In this scenario, the W-Server acts as a proxy for

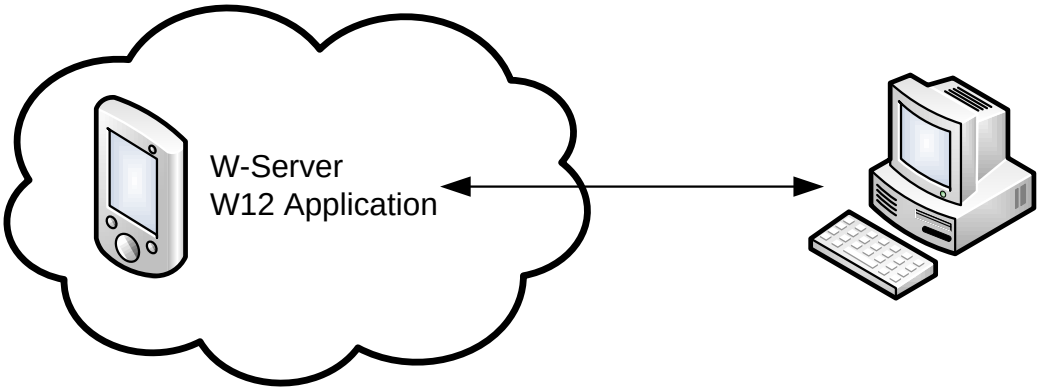

Fig. 4.2 Diagram showing W12 used for projecting mobile applications to an external terminal.

standard web browser requests, like loading a web page. Those requests are returned as expected but the W-Server embeds the content within the context of the WJS library and template. This allows W12 applications to be written as widgets to appear over traditional web content. These widgets could be simple sticky notes or calendar applications.

To develop this system, the W-Server would need to be extended to allow it to fetch arbitrary web content. Special care needs to be taken when creating this extension to prevent any potential cross-site scripting attacks either against the page being loaded through the W-Server or via W12 against the W-Server itself. W12 applications designed in this fashion would likely be widget-specific and not easily adapt themselves to other W12 deployment schemes.

Components of the W12 system could be placed at several points in the network to create this overlay system. The W-Server could be hosted by W12 application providers. Users could connect to the W-Server and select a list of desired overlays before beginning their session. Alternatively, users could run the W-Server and the

desired W12 applications locally. In this context, users could send requests through the W-Server, having the W12 overlay available at all times. This could be extended to only show the overlay for certain types of web requests. In the extreme case, the W-Server could even be integrated directly into a web browser. Doing this would make W12 a widget development platform for web browsers, similar to the extension platforms available for several current web browsers.

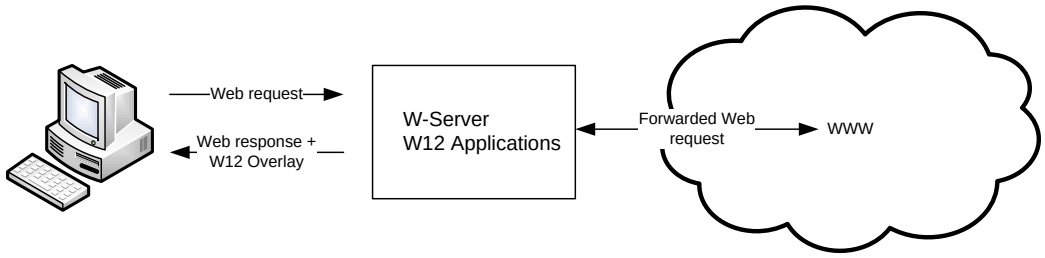

Fig. 4.3 Diagram showing W12 used as an information overlay system.

Chapter 5

Performance Test Results

This chapter presents the results of performance testing done on the current implementation of the W12 system.

5.1 Test Setup

The W12 system was tested both objectively and subjectively using two separate test applications. Test applications were implemented using several programming languages. The goal was to compare C-based W12 applications to other browser-based and native implementations. The first test application, an objective test, emits primitive draw requests, rectangles in this case, over several iterations. The second, subjective, test application is a programme called weyes, a W12 version of the classic xeyes application. All tests were run on a machine with a Pentium Core 2 Quad Q6600 CPU running at 2.4GHz equipped with 2 GB of system memory.

5.2 Objective Test

The objective tests were run using 1,000, 5,000, 10,000, 15,000, and 20,000 primitive requests. Four different versions of the test application were implemented. The first version was implemented using standard W12 calls. The second version was coded in JavaScript, making direct calls to the WJS library. This provides a simulation of a W12 application without network overhead. The third implementation was again coded in JavaScript but this time using only Processing.js native calls. This provides a reference for the speed of the PJS library upon which WJS is built. The final implementation was coded using Java and the native Processing library, of which PJS is a port. This provides a reference for native system performance outside of a web browser.

The tests were run in two separate instances. First, a 10 iteration run of each size was executed, measuring median response time. Second, 100 iterations of each size were executed, this time measuring average response time time. Figure 5.1 and Figure 5.2 highlight how each implementation fared.

Unsurprisingly, the java performance exceeds that of javascript and W12. Also unsurprisingly, the only results which have any noticeable standard deviation are those of the W12 implementation. This can be attributed to the nature of network communication being somewhat random in its response time. It's interesting to note that the JavaScript implementations show virtually identical performance. This implies that the overhead cost associated to WJS-specific code is minimal compared to cost of executing raw Processing.js commands. The speed at the browser side depends only on the speed of the javascript drawing library. Test results indicate that the W12 application scales very similarly to the JavaScript-only code. This implies that

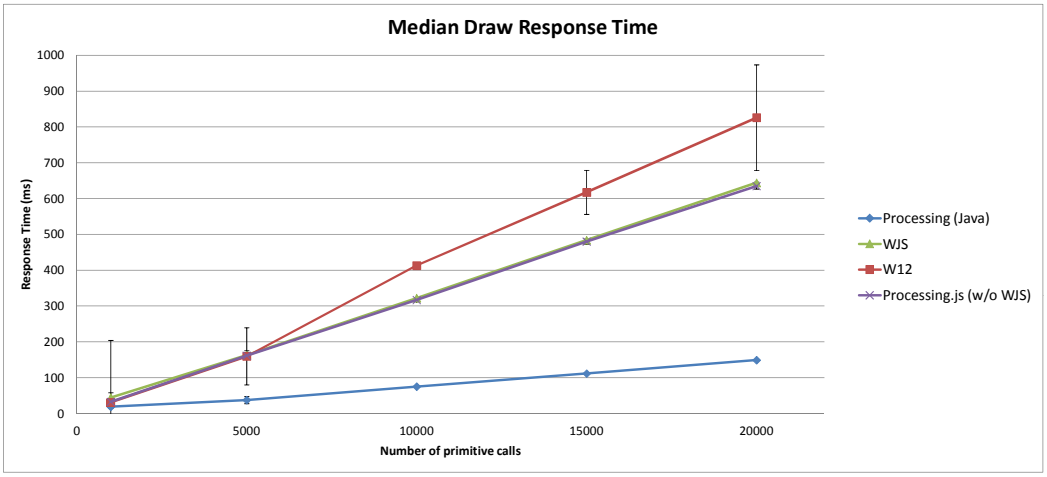

Fig. 5.1 Graph shows the response time of the test application when making various numbers of primitive requests. Data represents the median time over 10 iterations.

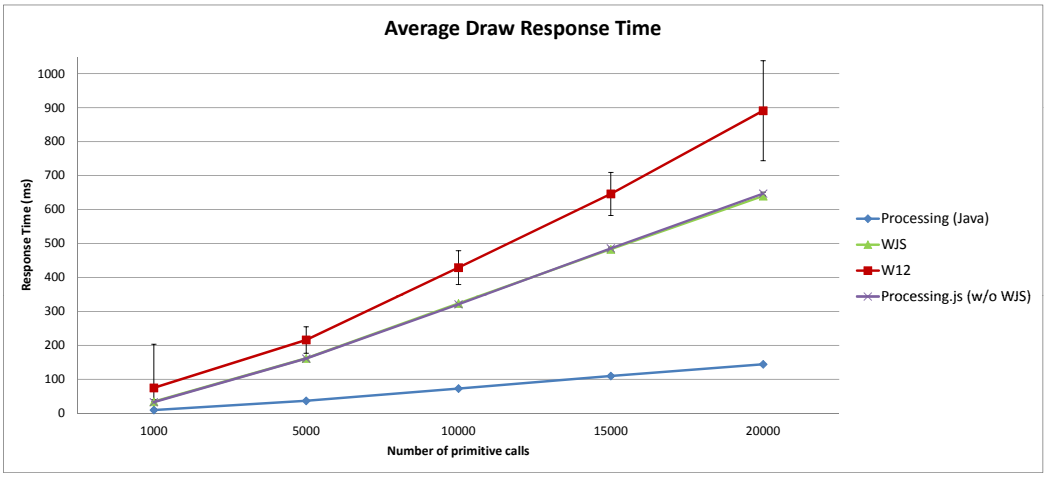

Fig. 5.2 Graph shows the response time of the test application when making various numbers of primitive requests. Data represents the average time over 100 iterations.

W12 applications incur a fixed overhead cost in submitting their draw requests. This makes sense since the second network trip, from the W-Server to the web browser, is masked from the application itself.

It should be noted that, in each case, the measured times represent the system execution time required for executing the draw requests only. They do not reflect the time required to actually draw the request on-screen. With the various levels of buffering, it is exceedingly difficult to obtain accurate "request-to-screen" objective measurements.

5.3 Subjective Test

Objective testing only tells part of the story. To gauge the responsiveness of W12, that is how well the system responds to user input, a W12-version of the X application xeyes was developed. The application, called `weyes`, draws cartoon eyes on the screen which follow the mouse pointer. This is an event-intensive application in the sense that mouse motion is constantly being tracked and the entire display is redrawn whenever the mouse moves. In this case, three separate versions of `weyes` were implemented. The first used standard W12 methods. The second used only the WJS library, simulating incoming calls with actual network interaction. The third method used the underlying Processing.js library only.

The response for both JavaScript-only implementations was essentially identical. The applications both responded well to mouse movements with no perceived lag. This suggests that the overhead of the WJS code is very small. It also shows that the Processing.js library is fast enough to support event-intensive applications. The W12 implementation, while not quite as responsive as the other two, did handle the

weyes application quite effectively. Lag was present but barely noticeable. Changes to the W-server did affect this amount of lag. Tests were run with logging enabled and logging disabled at the W-Server level. Tests with logging enabled showed much more noticeable lag than the test with logging disabled. This suggests that W-Server processing time can play an important role determining the overall responsiveness of W12 applications.

These results are favourable and show that W12 applications can be responsive enough to provide a positive user experience. Efforts aimed at improving performance at the JavaScript and/or W-Server levels could provide great gains in this respect.

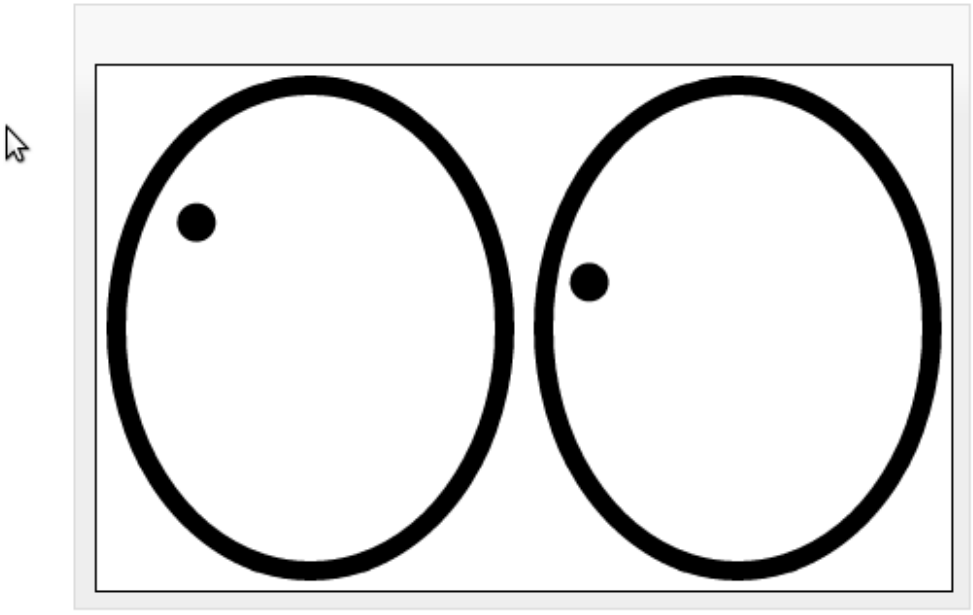

Fig. 5.3 `weyes`. A W12 test application.

Chapter 6

Related Works

W12 is by no means a unique idea. Technologies for displaying interactive GUIs over a network connection have existed for over 20 years. X and NeWS are prime examples of this. The advent and refinement of web technologies have shifted some of the technologies, but the core principles remain the same: decoupling an application's execution environment from its visualization environment. The following is a brief overview of other web-based UI libraries available.

6.1 Google Web Toolkit

The Google Web Toolkit (GWT) is a Java API with functionality provided to compile Java code directly to browser-executable JavaScript. The typical development paradigm for GWT is that of a "pure" client-side application, running entirely within the local web browser [21]. For applications needing more processing, several methods of client-server communication are provided. These range from remote procedure calls to a Java-based back-end server (specifically Java servlets), to HTTP requests

returning data in one of a variety of possible formats (e.g. JSON, XML, etc). These resemble standard AJAX calls as all methods provided in GWT are asynchronous (though not all of them return XML data as typical AJAX calls do) and require HTTP communication with a web server [21]. In some case, serialization/deserialization of Java objects is done automatically by GWT to facilitate back-end processes. GWT provides a more streamlined approach for existing server-based web applications to produce elegant and functional user interfaces. This differs from W12 which provides facilities for any application to project its user interface over the web natively.

6.2 Wt

Wt (pronounced witty) is a C++ library that provides HTML widgets for C++ applications [22]. Like W12, it allows applications to display themselves inside a typical Web Browser, relaying events back to the application for processing. It aims to abstract away development work in web-standard languages, like HTML, JavaScript, and CSS so that developers can code entire applications using only C++, as they would if they were developing a regular desktop application. Widgets are combined in a tree-hierarchy to create rich layouts on-screen. From a programmer's point of view, these widgets behave like any others from a standard GUI toolkit. Wt handles the process of sending them to the browser and creating HTML markup. Much as different toolkits exist for desktop application development, Wt and W12 represent different means to a similar end. Both platofrms allow applications to send their user interface over a web server to be viewed within a web browser.

Wt is very flexible in its rendering options. Like W12, it can output to a canvas element for pixel drawing, but it can also output to HTML and standard image formats

(JPG, PNG) including vector formats (SVG, PDF) [22]. Wt and W12 are very similar in goals while differing mostly in implmentation. Wt was developed to create HTML-style widgets available directly through C++ APIs. W12 was designed similarly to X to provide pixel-addressable drawing routine within a C API and allowing developers to create their own widget toolkits on top of W12. Wt makes use of either the Apache or its own wthttpd webserver, W12 uses the Python-based Twisted system.

6.3 Visual WebGUI

Visual WebGUI (VWG) is a commercial product aimed at developers using Microsoft's .NET platform. It creates a layer allowing desktop-oriented .NET applications to be deployed on the web using Microsoft's IIS application server [23]. The goal again is to abstract away web-standard technologies so that developers can deal only with objects in the developer's preferred language, in this case C#.NET or VB.NET. VWG wraps known .NET widgets into HTML to allow .NET Windows Form applications to be deployed through a web browser while maintaining a Windows-like look-and-feel. VWG is available as both a commercial product and as an LGPL-licensed open source package, both of which require Microsoft Visual Studio development environment.

6.4 Mobile Development Frameworks

The broad range of mobile devices and platforms has created a need for development kits that facilitate the creation of applications for multiple platforms. Several different such toolkits exist and each presents its advantages and disadvantages. Typically, these frameworks are aimed squarely at mobile devices and not personal computers

but they remain related in some respect to W12.

6.4.1 PhoneGap

The PhoneGap [24] platform allows developers to create applications entirely in HTML, JavaScript, and CSS. Application layout and general "look-and-feel" are created using HTML and CSS with JavaScript providing the application logic. PhoneGap provides JavaScript APIs to access various hardware functionality like GPS location, accelerometer data, and camera. This access is typically not granted to JavaScript code. PhoneGap packages exist for the following mobile platforms: iOS, Android, BlackBerry OS, WebOS, Windows Phone 7, Symbian, Bada [24]. Applications need to be packaged with PhoneGap to be usable, meaning that different versions of it exist for each platform. While the core application code is platform independent, the final package must include different versions of PhoneGap. This is different from W12 in which a single running application can be viewed across many platforms without needing to be repackaged.

6.4.2 Rhomobile Rhodes

The Rhodes platform [25], developed by Rhomobile, provides mobile cross-compatibility for ruby applications. Using Rhodes, a single application codebase can be easily compiled for a specific platform. It compiles code in a given platform's native format, meaning that no additional libraries or run-time packages need to be present in order for Rhodes applications to work. Rhomobile provides some additional features for syncing application data to cloud and for rapid application development. Unlike W12 applications, Rhodes applications must be executed locally.

Chapter 7

Conclusions and Future Development

This chapter presents development ideas to add more functionality to W12 and outlines various different ways W12 could be used in the future.

7.1 Future Work

W12, as a platform, is quite flexible. As has been shown in Chapter 4, the system can be adapted to many different uses. The system is currently still very much in development. While functional, it still lacks certain functionality to make it a broadly usable platform.

7.1.1 Security

Security was not taken into consideration when W12 was developed initially. To appeal to security-conscious users, W12 needs to be secured. First and foremost,

communication should be transfered to Secure Socket Layer communication using HTTPS and WSS (Secured WebSocket) protocols. A security review should also be performed on the W-Server and Wlib components to evaluate potential security holes in the system.

7.1.2 Performance Improvements

The weyes application showed that W-Server speed is central to the overall performance of the W12 system. The Twisted application platform was selected early in W12's development process, but no real performance research was done to compare alternative solutions. Platforms like lighttpd [26], Ngix [27], or teepeedee2 [28] could potentially outperform Twisted.

The current communication protocol is another potential performance improvement. The current version of the protocol was designed for simplicity more than performance. Proper analysis could minimize the size of the messages and improve the efficiency of communication in general.

7.1.3 Widget Library

In its current form, Wlib provides low-level drawing capability. Developing applications at this level can be a tedious process. A proper widget toolkit would facilitate application development; abstracting away low-level details inside high-level objects like buttons and labels. In this respect, Wlib would be following in the footsteps of Xlib, on top of which widget toolkits like Qt and GTK+ were developed.

Development of such a toolkit would like require components written in C and in JavaScript. The C-language component would be a intermediary library interacting

directly with Wlib. Application developers would include this library and forget about the underlying details inherent to Wlib requests. The JavaScript component would interact with the WJS library to further manange event generation. The broad events currently generated by WJS would need to be focused on the widget within which they occurred. This would yield widget-based events which are much easier to delegate for application developers.

7.1.4 Client-side Code

Taking inspiration from the short-lived NeWS developed by Sun, W12 could use the JavaScript interpreter more centrally to allow method deployment/execution directly on the client browser. Through a PostScript interpreter and C preprocessor, NeWS allowed arbitrary PostScript code to be loaded into the display server at runtime. This made it possible to have, for instance, event handlers present on both sides of the network connection. For W12, this would mean the potential to service events directly within the browser without the need for a network round-trip, making event handling much faster. Extending this idea even further could mean cloud applications capable of determining the computing power of the device connected. W12 could then decide, on the fly, whether event callbacks should be handled on the device or within the application, weighing network latency against computational power.

7.2 Conclusions

This thesis presents the design and implementation of the W12 Network Display system. The system addresses the problem of platform compatibility by allowing a single application to be rendered on any platform through a web browser. Application

developers using the W12 system would not need to maintain several codebases to support multiple platforms. The system also addresses the fact that users are not only mobile but have an increasing number of ways to connect to their applications. By targeting web browsers directly, W12 allows applications to blur the lines between different classes of devices.

User testing showed W12 to be functional and surprisingly effective. Tests showed that the system is capable of creating user interfaces which display across multiple platforms like standard PCs and iPad tablets. W12's multiple deployment scenarios make it an appealing platform for several different classes of applications, from simple widgets to full desktop applications. With mobile computing constantly evolving, W12 is a framework with great potential.

Appendix A

Weyes application source code

This appendix contains the full source of the *weyes* application. It shows the basic layout of a W12 application with heavy use of callback functions.

```
/*

The MIT License (MIT) Copyright (c) 2011 Julien Lord, Muthucumaru
Maheswaran

Permission is hereby granted, free of charge, to any person obtaining a
copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

*/


#include <hlib/hlib.h>
#include <stdio.h>
```

```
#include <stdlib.h>
#include <string.h>
#include <math.h>

#define STROKE_W  10.0
#define EYE_W    200.0
#define EYE_H    250.0
#define PUPIL_W    10.0
#define EYE_RAD    (EYE_W / 4.0 - PUPIL_W)

/* PROTOTYPES */
int draw(Display *display, int num, int mouseX, int mouseY);


int size_x     = 2*EYE_W + 4*STROKE_W;
int size_y     = EYE_H + 2*STROKE_W;
float *eyes   = NULL; /* Eye centre positions */
float *pupils   = NULL; /* initial pupil positions */


/** Event handlers **/
void setup(Display *display, Event *event, void *data)
{
  /* white bg */
  Background1i(display, 255);
  Size(display, size_x, size_y);

  /* black stroke and white fill */
  StrokeWeight(display, STROKE_W);
  Stroke1i(display, 0);
  Fill1i(display, 255);

  if (eyes == NULL) {
    eyes = malloc(4*sizeof(float));
    eyes[0] = 0.25 * size_x;
    eyes[1] = 0.50 * size_y;
    eyes[2] = 0.75 * size_x;
    eyes[3] = 0.50 * size_y;
  }

  if (pupils == NULL) {
    pupils = malloc(4*sizeof(float));
    memcpy(pupils, eyes, 4*sizeof(float));
  }
}

void mouse_move(Display *display, Event *event, void *data)
{
```

```
  ClearScreen(display);
  int x = event->val.mouse.x;
  int y = event->val.mouse.y;

  if (x < 0) x = 0;
  if (y < 0) y = 0;
  draw(display, 0, x, y);
  draw(display, 2, x, y);
}

void expose_event(Display *display, Event *event, void *data)
{
  draw(display, 0, eyes[0], eyes[1]);
  draw(display, 2, eyes[2], eyes[3]);
}

int draw(Display *display,int num, int mouseX, int mouseY)
{
  double angle1, angle2;
  double dx,dy;
  double tx,ty;
  double cosa, sina;
  double x,y,x2,y2;

  x = pupils[num];
  y = pupils[num+1];
  x2 = eyes[num];
  y2 = eyes[num+1];

  DrawEllipse(display,(int)eyes[num],(int)eyes[num+1],EYE_W,EYE_H);

  PushStyle(display);
  Fill1i(display, 0);

  dx = (double)mouseX - x;
  dy = (double)mouseY - y;
  angle1 = atan2(dy,dx);

  cosa = cos(angle1);
  sina = sin(angle1);
  tx = mouseX - cosa * EYE_RAD;
  ty = mouseY - sina * EYE_RAD;

  dx = tx - x2;
  dy = ty - y2;
  angle2 = atan2(dy,dx);

  cosa = cos(angle2);
```

```
  sina = sin(angle2);
  x    = x2 + cosa * EYE_RAD;
  y    = y2 + sina * EYE_RAD;

  PushMatrix(display);
  Translate2f(display,x, y);
  Rotate(display, angle1);
  DrawEllipse(display, EYE_RAD, 0, PUPIL_W, PUPIL_W);
  PopMatrix(display);

  pupils[num] = x;
  pupils[num+1] = y;

  PopStyle(display);
  return 0;
}

int main(int argc, char *argv[])
{
  Display *display = NULL;
  char *host = "localhost";
  int port = 9090;

  display = OpenDisplay("localhost", 9090);
  if (display == NULL) {
    fprintf(stderr,"Unable to connect to %s:%d\n", host,port);
    exit(1);
  }

  /* Register Callbacks */
  RegisterCallback(display, ExposeEventType, expose_event, NULL);
  RegisterCallback(display, SetupEventType, setup, NULL);
  RegisterCallback(display, MouseMoveEventType, mouse_move, NULL);

  MainLoop(display);

  CloseDisplay(display);
  free(eyes);
  free(pupils);
  return 0;
}
```

Appendix B

Wlib API Specification

This appendix provides an API specification for Wlib. It lists and explains the methods exposed by the library.

B.1 Data Structures

B.1.1 The `Display struct`

The main display object used in Wlib.

```
typedef struct Display
{
    char *hostname;
    int port;
    int width;  /*deprecated*/
    int height; /*deprecated*/
    Socket *socket;
    Callbacks *callbacks;
} Display;
```

The Display object holds the hostname and port information for the W-Server this display is linked to. It holds pointers to the socket through which all communication with W-Server will occur and to the list of all callbacks handlers.

B.1.2 `Event` Structures

The following are the data structures used for sending event data from the browser to the application.

```
typedef enum EventType
{
  SetupEventType,
  PreLoad,
```

```
  Resize,
  ExposeEventType,
  ClickEventType,
  MouseDownEventType,
  MouseMoveEventType,
  MouseDragEventType,
  MouseDragOutEventType,
  KeyPressed,
  KeyReleased,
  KeyTyped,
  FileDropInit,
  FileDropChunkReceived,
  FileDropEnd,
  b64FileDropInit,
  b64FileDropChunkReceived,
  b64FileDropEnd
} EventType;

typedef struct Event
{
    EventType type;
    union {
        MouseEvent mouse;
        KeyboardEvent keyboard;
        DropEvent drop;
        b64DropEvent drop64;
        WinSize win;
    } val;
} Event;
```

The Event object is a generalization of all events available. It contains the type, selected from the EventType enumeration listed above, and a more specific event structure.

```
typedef struct MouseEvent
{
    int x;  /* X coordinate of click */
    int y;  /* Y coordinate of click */
    int button; /* mouse button clicked */
    int dx; /* mouse movement deltas */
    int dy;
} MouseEvent;
```

The MouseEvent structure is used when reporting mouse-related events. The `x` and `y` coordinates represent the current position of the mouse pointer. The `dx` and `dy` coordinates represent the change in position since the previous mouse event was reported. The `button` field is to report which button was used for button-related mouse events. Constants are defined for `LEFT`, `MIDDLE` and `RIGHT` mouse buttons.

```
typedef struct KeyboardEvent
{
    int keycode;
} KeyboardEvent;
```

The KeyBoardEvent structure is used when reporting `KeyPressed`, `KeyReleased` and `KeyTyped` events. The `keycode` field is used to report the key related to the event. In the case of `KeyTyped` events, the value represents the ASCII code of the letter typed. The other keyboard events return the keycode of the key.

```
typedef struct DropEvent
{
    const char *name; /* filename */
    const char *type; /* filetype */
    unsigned int size;  /* filesize */
    unsigned int num_chunks; /* Number of transfer chunks */
    unsigned int chunk_size; /* current chunk size */
    unsigned int cur_chunk; /*current chunk number */
    char *chunk; /* deprecated */
} DropEvent;
typedef struct b64DropEvent
{
    const char *name; /* filename */
    const char *type; /* filetype */
    unsigned int o_size;  /* original filesize */
    unsigned int e_size;  /* encoded filesize ie total transfer size*/
    unsigned int num_chunks; /* Number of ENCODED transfer chunks */
    unsigned int chunk_size; /* current encoded chunk size */
    unsigned int cur_chunk; /* current encoded chunk number */
    char *chunk; /* deprecated */
} b64DropEvent;
```

The DropEvent structures are used when reporting file transfer events. As the names would suggest, the DropEvent structure is used for normal file transfers and the b64 variant is used for Base64-encoded file tranfers. Both structures hold metadata related to the file being transferred.

```
typedef struct WinSize
{
    unsigned int width;
    unsigned int height;
} WinSize;
```

The WinSize structure is used to report the size of the browser window, in pixels. It is used when reporting `SetupEventType` and `Resize` events.

B.2 Wlib API

Wlib is the client library used by W12 applications. It provides methods for connecting to the W-Server and sending requests to web clients. The API documentation is roughly separated by functionality. All calls use the pixel as the basic unit of measure.

B.2.1 Connection Management

General purpose methods used in Wlib.

```
Display *OpenDisplay(char *hostname, int port)
```

Initiates a connection to W-Server using the given hostname and port. On success, it returns a pointer to a fully usable Display object.

```
void CloseDisplay(Display *display)
```

Terminate the connection between to the given Display object and the W-Server.

```
Event *GetEvent(Display *display)
```

Block on the socket in the Display object until a new message arrives from the W-Server. Returns a pointer to the event generated.

```
void RegisterCallback(Display *display, EventType etype, EventCallback
cb, void *data)
```

Register a new callback handler on the Display object. The method pointer *cb* will be called when a method of type *etype* is received from the W-Server. The data pointer allows arbitrary data to be stored. This can be used to pass additional data to the callback method, for example.

```
void MainLoop(Display *display);
```

Entry point for all W12 applications. Runs a loop calling GetEvent. A NULL event will terminate the main loop.

```
int RegisterRemoteInt(Display *display, const char* name, int value);
```

This was an attempt to register variables with javascript. It's incomplete and should not be used as it is now.

```
int RegisterRemoteFloat(Display *display, const char* name, float value);
```

This was an attempt to register variables with javascript. It's incomplete and should not be used as it is now.

```
int DisableKeyList(Display *display, EventType etype, char* list);
```

Specifies a list of characters which should NOT be passed to the browser i.e. that sets the listeners to return false

```
char* keyboardListBuilder(int num, ...);
```

Helper method. Creates a list properly null-terminated list of characters for keyboard callbacks.

B.2.2 2D draw primitives

All 2D primitives draw using the current Stroke color for the outline and the current Fill color for the fill.

```
int DrawArc(Display *display, int x, int y, int width, int height, float start, float stop);
```

Draws an arc around the ellipse defined by the x,y,width,height parameters with the start and stop parameters defining the angles, in radians, between which the arc should run.

```
int DrawEllipse(Display *display, int x, int y, int width, int height);
```

Draw an ellipse defined by the point (x,y) and of size defined by width and height. By default, (x,y) represents the center point of the ellipse, but htis can be changed by calling EllipseMode.

```
int DrawLine2D(Display *display, int x0, int y0, int x1, int y1);
```

Draw a line from the point (x0,y0) to the point (x1,y1).

```
int DrawPoint2D(Display *display, int x, int y);
```

Draw a point at (x,y).

```
int DrawQuad(Display *display, int x0, int y0, int x1, int y1, int x2, int y2, int x3, int y3);
```

Draw an arbitrary quadrilateral shape defined by the four corners (x0,y0), (x1,y1), (x2,y2), and (x3,y3).

```
int DrawRectangle(Display *display, int x, int y, int width, int height);
```

Draw a rectangle from the point (x,y) and given size. By default, (x,y) is the top-left corner. Calls to RectMode can change this.

```
int DrawTriangle(Display *display, int x0, int y0, int x1, int y1, int x2, int y2);
```

Draw a triangle by connecting a line between the three points passed.

B.2.3 Colour Management

```
int Background1i(Display *display, int grey);
```

Redraw the background in the given shade of grey. Overwrites the canvas. Equivalent to calling Background3i with r=g=b.

```
int Background3i(Display *display, int r, int g, int b);
```

Redraw the background in the given color.

```
int ColorMode1i(Display *display, int mode, int r0);
```

Changes the way color calls are interpreted.

Sets the range for colors within the mode. Subsequent color calls will be intepreted within this range. Takes an integer range.

e.g. Calling ColorMode1i with mode RGB and r0 = 10 will make the maximum RGB values 10 instead of the default 255. Intermediate values are scaled appropriately. Valid modes are:

- RGB
- HSB

These are defined as constants.

```
int ColorMode3i(Display *display, int mode, int r0, int r1, int r2);
```

Changes the way color calls are interpreted. Sets the range for colors within the mode. Subsequent color calls will be intepreted within this range. Takes integers. e.g. Calling ColorMode3i with mode RGB and r0 = 10 , r1 = 20, r2 = 30 will make the maximum RGB values 10 for red, 20 for gree, and 30 for blue instead of the default 255 for each. Intermediate values are scaled appropriately. Valid modes are:

- RGB
- HSB

These are defined as constants.

```
int ColorMode1f(Display *display, int mode, float r0);
```

Changes the way color calls are interpreted. Sets the range for colors within the mode. Subsequent color calls will be intepreted within this range. Takes a floating point range. e.g. Calling ColorMode1f with mode RGB and r0 = 1.0 will make the maximum RGB values 1.0 instead of the default 255. Intermediate values are scaled appropriately. Valid modes are:

- RGB
- HSB

These are defined as constants.

```
int ColorMode3f(Display *display, int mode, float r0, float r1, float r2);
```

Changes the way color calls are interpreted. Sets the range for colors within the mode. Subsequent color calls will be intepreted within this range. Takes floating point values. e.g. Calling ColorMode3f with mode RGB and r0 = 1.0 , r1 = 2.0, r2 = 3.0 will make the maximum RGB values 1.0 for red, 2.0 for gree, and 3.0 for blue instead of the default 255 for each. Intermediate values are scaled appropriately. Valid modes are:

- RGB
- HSB

These are defined as constants.

```
int Stroke1i(Display *display, int val);
```

Set the Stroke color to a shade of grey. Equivalent to calling Stroke3i with r=g=b. Takes an integer value.

```
int Stroke3i(Display *display, int r, int g, int b);
```

Set the Stroke color to the given RGB value. Takes integer values.

```
int Stroke1f(Display *display, float val);
```

Set the Stroke color to a shade of grey. Equivalent to calling Stroke3f with r=g=b. Takes a floating point value.

```
int Stroke3f(Display *display, float r, float g, float b);
```

Set the Stroke color to the given RGB value. Takes floating point values.

```
int Fill1i(Display *display, int val);
```

Set the Fill color to the given shade of grey. Equivalent to Fill3i with r=g=b. Takes an integer value.

```
int Fill3i(Display *display, int r, int g, int b);
```

Set the Fill color to the given color. Takes a integer values.

```
int Fill1f(Display *display, float val);
```

Set the Fill color to the given shade of grey. Equivalent to Fill3f with r=g=b. Takes a floating point value.

```
int Fill3f(Display *display, int r, int g, int b);
```

Set the Fill color to the given color. Takes floating point values.

B.2.4 Draw Mode Attribute

```
int EllipseMode(Display *display, int mode);
```

Define the way Ellipses are drawn. i.e. which point do the passed coordinates represent. Valid values are:

- CENTER
- RADIUS
- CORNER
- CORNERS

These are defined constants.

```
int RectMode(Display *display, int mode);
```

Define the way Rectangles are drawn. i.e. which point do the passed coordinates represent. Valid values are:

- CENTER
- RADIUS
- CORNER
- CORNERS

These are defined constants.

```
int StrokeCap(Display *display, const char* mode);
```

Define the way strokes end. Valid values are:

- SQUARE
- PROJECT
- ROUND

These are defined constants.

```
int StrokeJoin(Display *display, const char* mode);
```

Define the way strokes join together. Valid values are:

- MITER
- BEVEL

- ROUND

These are defined constants.

```
int StrokeWeight(Display *display, int w);
```

Sets the width of the stroke when drawing primitives.

```
int NoStroke(Display *display);
```

Draw subsequent primitives with no Stroke.

```
int NoFill(Display *display);
```

Draw subsequent primitives with no Fill.

B.2.5 Vertex Functions

Methods dealing with vertex-based drawing.

```
int BeginShape(Display *display);
```

Using BeginShape/EndShape allows for the creation of more complex shapes. BeginShape starts the shape definition. Used in conjunction with Vertex methods

```
int BeginShapeMode(Display *display, int mode);
```

Using BeginShapeMode/EndShapeMode allows for the creation of more complex shapes using different types of rendering. BeginShapeMode begins the shape and determines the mode to use. Differs from BeginShape in setting how vertices are drawn. Valid modes are:

- POINTS
- LINES
- TRIANGLES
- TRIANGLE_FAN
- TRIANGLE_STRIP
- QUADS
- QUAD_STRIP

These are defined as constants. Invalid modes are ignored. Used in conjunction with Vertex methods

```
int EndShape(Display *display);
```

Using BeginShape/EndShape allows for the creation of more complex shapes. EndShape finalises the shape definition. Used in conjunction with Vertex methods

```
int EndShapeMode(Display *display, int mode);
```

Using BeginShapeMode/EndShapeMode allows for the creation of more complex shapes using different types of rendering. EndShapeMode finalises the shape. The only valid mode is CLOSE. Invalid modes are ignored. This is defined as a constant. Used in conjunction with Vertex methods

```
int Vertex2D(Display *display, int x, int y);
```

Used inside a BeginShape/EndShape block. Defines a vertex inside the shape.

B.2.6 Matrix Transformations

Methods dealing with transformation matrices.

```
int PushMatrix(Display *display);
```

Marks the beginning of a transformation matrix.

```
int PopMatrix(Display *display);
```

Mark the end of a transformation matrix.

```
int Translate2i(Display *display, int x, int y);
```

Add a translation of (x,y) into the transformation matrix. Integer-valued.

```
int Translate2f(Display *display, float x, float y);
```

Add a translation of (x,y) into the transformation matrix. Floating point-valued.

```
int Rotate(Display *display, float angle);
```

Add a rotation of angle RADIANS into the transformation matrix.

```
int Scale1f(Display *display, float scale);
```

NOT IMPLEMENTED

```
int Scale2f(Display *display, float x, float y);
```

NOT IMPLEMENTED

B.2.7 Font Management

Methods dealing with loading and selecting fonts.

```
int CreateFont(Display *display, const char *fontName, const char
*fontURL);
```

Used to preload fonts via css @font-face directives.

```
int LoadFont(Display *display, const char *fontName, int size);
```

Load a font by name and size.

```
int TextFont(Display *display, const char *fontName, int size);
```

Set the font to be used by name and size.

```
int TextAlign(Display *display, int h_align, int v_align);
```

NOT IMPLEMENTED

```
int TextLeading(Display *display, int dist);
```

NOT IMPLEMENTED

```
int TextSize(Display *display, int size);
```

NOT IMPLEMENTED

B.2.8 Text Output

Methods dealing with emitting text.

```
int SendText(Display *display, int x, int y, char *text);
```

Send an arbitrary string to be displayed on screen. Text will start at position (x,y) and be printed within the canvas. This text will not be user-selectable.

```
int OverwriteTextArea(Display *display, const char *target, const char
*text);
```

Overwrite the contents of the TextArea with id target.

```
int AppendTextArea(Display *display, const char *target, const char
*text);
```

Append the given text to the existing text in the TextArea with id target.

```
int CreateTextArea(Display *display, const char *id, int x, int y, int
width, int height, int readonly);
```

Create a text area over the canvas to receive text directly. This uses HTML textarea elements and positions them over the canvas.

```
int TextAreaSetFont(Display *display, const char *id, const char
*fontName);
```

Set the font to be used inside the TextArea with id id.

```
int TextAreaSetCss(Display *display, const char *id, const char *name,
const char *value);
```

This is a general purpose way to call jquery from C. Just a front for $("#id").css("name", "value");

B.2.9 Miscellaneous Functions

Various methods that didn't fall into another category.

```
int PushStyle(Display *display);
```

PushStyle allows temporary styles changes to occur. Current style settings are saved. Subsequent style changes remain in effect until acall to PopStyle.

```
int PopStyle(Display *display);
```

Return the style settings to what they were prior to calling PushStyle.

```
int Size(Display *display, int width, int height);
```

Change the size of the display.

References

[1] M. A. James Gosling, David S.H. Rosenthal, *The NeWS Book: An introduction to the Network/Extensible Window System*. Springer-Verlag, 1989.

[2] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. Rosenthal, and F. D. Smith, "Andrew: a distributed personal computing environment," *Commun. ACM*, vol. 29, pp. 184–201, March 1986.

[3] Adobe Systems Incorporated, 345 Park Avenue San Jose, CA 95110, *The Display PostScript System Communication Handbook*, 1994.

[4] R. W. Scheifler and J. Gettys, "The x window system," *ACM Trans. Graph.*, vol. 5, pp. 79–109, April 1986.

[5] R. W. Scheifler, *X Window System Protocol: X Consortium Standard*. The Open Group, version 11 release 6.8 ed., 2004.

[6] D. P. Grisby, *A Distributed Adaptive Window System*. PhD thesis, University of Cambridge, 1999.

[7] T. Richardson, *The RFB Protocol*. RealVNC Ltd, version 3.8 ed., November 2010.

[8] J. Yu, B. Benatallah, F. Casati, and R. Saint-Paul, "Openxup: an alternative approach to developing highly interactive web applications," in *Proceedings of the 6th international conference on Web engineering*, ICWE '06, (New York, NY, USA), pp. 289–296, ACM, 2006.

[9] OpenXUP.org, "Openxup official website." `http://openxup.org/`. [Online; accessed 5-December-2011].

[10] A. Larsson, "Gtk+ broadway author's blog." `http://blogs.gnome.org/alexl/`. [Online; accessed 5-December-2011].

[11] T. M. Labs, "Twisted matrix labs." `http://twistedmatrix.com/trac/`. [Online; accessed 5-December-2011].

[12] I. E. T. Force, "Bidirectional or server-initiated http (hybi)." `https://datatracker.ietf.org/wg/hybi/charter/`. [Online; accessed 5-December-2011].

[13] jQuery Project, "jquery official website." `http://jquery.com/`. [Online; accessed 5-December-2011].

[14] jQuery Project and jQuery UI Team, "jqueryui official website." `http://jqueryui.com/`. [Online; accessed 5-December-2011].

[15] J. Resig, A. MacDonald, and S. C. C. for Development of Open Technology, "Processing.js official website." `http://processingjs.org/`.

[16] D. Crockford, "Javascript object notation." `http://www.json.org/`. [Online; accessed 5-December-2011].

[17] Wikipedia, "Websocket — wikipedia, the free encyclopedia." `http://en.wikipedia.org/w/index.php?title=WebSocket&oldid=464069728`, 2011. [Online; accessed 5-December-2011].

[18] Google, "Minimize payload size." `http://code.google.com/speed/page-speed/docs/payload.html`. [Online; accessed 5-December-2011].

[19] M. Reynolds, "Using Lightweight Formal Methods for JavaScript Security," Tech. Rep. BUCS-TR-2010-021, CS Department, Boston University, July 23 2010.

[20] S. Guarnieri and B. Livshits, "Gatekeeper: mostly static enforcement of security and reliability policies for javascript code," in *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, (Berkeley, CA, USA), pp. 151–168, USENIX Association, 2009.

[21] Google, "Google web toolkit official website." `http://code.google.com/webtoolkit/`. [Online; accessed 5-December-2011].

[22] E. bvba, "Wt, c++ web toolkit." `http://www.webtoolkit.eu/wt/`. [Online; accessed 5-December-2011].

[23] V. WebGui, "Visual webgui official website." `http://www.visualwebgui.com/`. [Online; accessed 5-December-2011].

[24] A. S. Inc., "Phonegap." `http://phonegap.com/`. [Online; accessed 5-December-2011].

[25] I. Rhomobile, "Rhomobile." `http://rhomobile.com/`. [Online; accessed 5-December-2011].

[26] J. Kneschke, "Lighttp official website." `http://www.lighttpd.net/`. [Online; accessed 5-December-2011].

[27] I. Sysoev, "Nginx official website." `http://nginx.org/`. [Online; accessed 5-December-2011].

[28] J. Fremlin, "teepeedee2 official repository." `https://github.com/vii/teepeedee2`. [Online; accessed 5-December-2011].