jsVNC - A Case Study of Engineering Cloud-Based Network Applications

Simon Andreas Frimann Lund

June 4, 2010

Contents

1 Introduction **3**
1.1 Terminology . . . 4
1.2 Related Work . . . 4

2 Analysis **4**
2.1 VNC Application . . . 4
2.2 Infrastructural Challenges and Browser Limitations . . . 6
2.3 Techniques . . . 6
2.3.1 Initiate Retrieval . . . 7
2.3.2 Server Push . . . 7
2.4 Technique Implementations . . . 7
2.4.1 Bayeux . . . 8
2.4.2 BOSH . . . 8
2.5 Wire Protocols . . . 8
2.5.1 WebSockets . . . 8
2.6 Security Concerns . . . 9
2.7 Frame-buffer Rendering . . . 9
2.8 Conclusion . . . 9

3 Architecture & Design **10**

4 Implementation **11**
4.1 Hobs . . . 11
4.1.1 Session Creation . . . 11
4.1.2 Server to Client Messages . . . 12
4.1.3 Client to Server Messages . . . 12
4.2 jsVNC . . . 13
4.3 MIFCHO . . . 14
4.4 MIFCHO Protocol . . . 16
4.4.1 Handshake . . . 16
4.4.2 Tunnel Setup Request and Response . . . 16
4.5 Using MIFCHO . . . 17

5 Experiments **18**
5.1 Results . . . 18

6 Conclusion **20**
6.1 Future Work . . . 20

References **21**

A MIFCHO Configuration Example **22**

B Message Samples **22**
B.1 Hobs Session Creation . . . 22
B.2 Hobs Sending Message . . . 22

B.3 Hobs Receiving Message 23
B.4 WebSocket Initialization 23
B.5 WebSocket Send 23

C Physical Medium **24**
C.1 Online 24

Abstract

Cloud computing provides a means for seamlessly making computing resources available as a service, on-demand, everywhere. The work in this paper studies the engineering challenges of making a VNC client available as a service, on-demand in an Internet-browser.

A VNC client has been engineered and middleware has been implemented which encapsulates the infrastructural challenges of providing cloud-based network applications.

Experiments show promising results that it is possible to engineer network applications in the browser which require high throughput and low latency. Experiments however also show that the price of cloud-based VNC client is paid with significantly higher CPU utilization, memory consumption and bandwidth consumption than a traditional VNC client.

1 Introduction

Internet-browsers support the paradigm of cloud computing. In cloud computing the goal is to provide computing resources as a service, on-demand via the Internet. Internet-browsers facilitate the cloud-computing paradigm of providing a means for on-demand delivery of applications via the Internet without requiring any further installation or maintenance of software on the users device.

Browser-based applications has multiple advantages that makes them more attractive than traditional applications to both application-users and application-developers. They can be accessed from any device available to the user as long as the device has an Internet browser. This can be their laptop, smart phone or a device the user can borrow such as a library PC or a friends computing device.

A browser-based application consists of HTML, CSS and JavaScript. HTML and CSS are used to declaratively define the graphical presentation of the application and JavaScript is used to define the actual application-logic. Popular browser-based applications for application-domains such as mail-clients (Hotmail, G-Mail), social networking (Facebook, LinkedIn) and many others exists.

Browser-based applications however do not exist for all application-domains since technical limitations of Internet-browsers hinders them. Applications requiring efficient manipulation of 2D or 3D graphics, or low-level network communication. Such application-domains include games, Computer-Assisted-Design and network-clients.

The Internet-browser was not designed to handle these application-domains but since the browser is a facilitator for the cloud computing paradigm of applications as a service provides a strong incentive for pushing the boundaries of which application-domains the browser should handle.

One approach to pushing the boundaries is to expand the capabilities of the browser by using third-party plug-ins. Such plug-in technology include Java Runtime Environments, Flash, ActiveX components, Microsoft Silverlight and others. The third-party plug-in must be installed and maintained on the users device which is a compromise of the cloud computing ideal of being able to seamlessly provide applications as a service. Another aspect of third-party plug-ins are that the plug-ins can be proprietary and conflict with distribution licensing of the many different potential platforms. A security aspect of third-party plug-ins are that they provide additional attack-vectors for remote-code execution. Since third-party plug-ins provide additional access to local res sources they are also vulnerable to exposing local res sources when exploited.

Another approach to expanding the capabilities of browser-based applications are to expand the capabilities offered by the browsers themselves and to stretch and combine current capabilities of the browser in new ways to achieve the desired functionality. By doing so the design goal of being able to seamlessly provide applications as a service can be maintained.

The work in this report focuses on using the latter approach in an effort to engineer a browser-based VNC-client, named jsVNC (JavaScript VNC). A VNC client is not well-suited to be run in an Internet-browser since it requires an efficient way to render a frame-buffer on the clients device and it needs to communicate with a VNC server over TCP. An analysis is provided in section 2 on how current Internet-browser capabilities and current work-in-progress on expanding Internet-browser capabilities can be combined to obtain the functionality needed for the engineering of jsVNC.

Engineering the browser-based application is however not the only challenge in engineering a cloud-based network application. The browser is only a facilitator for providing the client-side application as a service to the user. For a network client to be usable it must be able to contact the corresponding server, establishing end to end connections on the Internet is non-trivial. The Internet consists of many hosts directly connected to the Internet with a public IP-address establishing connections to such hosts are trivial, the problem persists in the many hosts on private networks connect to the Internet but not accessible from the Internet due to firewall restrictions and network address translators.

Existing tools and applications suites exist that can be configured and combined to solve such issues but doing so can be complex. The work in this paper documents a MIddleware For Connection Handling and Orchestration (MIFCHO) unifies the connectivity and protocol translation issues for cloud-based networking applications in a simple solution.

The rest of the report is structured as follows. Section 1.1 describes the terminology used in this report. Section 1.2 describes the essential differences between jsVNC/MIFCHO and similar VNC-clients and middleware solutions. As previously mentioned then section 2 analyzes the possibilities and challenges with current technology. Section 3 describes the architecture and design of jsVNC and MIFCHO. Section 4 describes the implementation of jsVNC and MIFCHO. In section 5

the performance of jsVNC is evaluated and compared to a traditional VNC-client, the section also discuss the results of the experiments. Lastly a conclusion on the work in this report is provided in section 6.

1.1 Terminology

When referring to browser, Internet-browsers such as Internet Explorer, Firefox, Google Chrome, Opera is the programs referred to.

When mentioning *modern* browsers, these are references to browsers that support the HTML5 canvas tag.

1.2 Related Work

The work documented in this report can be separated into three major problem areas. The first is the actual implementation of the VNC-client in JavaScript. Second is the client-side libraries and protocol implemented to emulate bidirectional communication and third is the implementation of the middleware that supports the client-side code.

jsVNC is focused on not having any third-party requirements and to function in a strict browser environment. Other approaches implement the entire VNC-client as a third party component, such approaches include: TightVNC Java-Viewer[16](Java) and FlashLight-VNC[4](Flash). Another approach is to write the VNC client in JavaScript but rely on third-party components via a bridging technique to enable socket communication in JavaScript. Bridging techniques include [8](Flash) or [22](Java).

jsVNC is more closely related to the two projects CARDE[3] and Guacamole [21]. Both of these projects are like jsVNC implemented entirely in JavaScript without relying on any third-party plug-ins.

CARDE however relies on the work-in-progress of browser-based WebSockets[19]. jsVNC also utilize WebSockets when available but also implement an emulation of WebSockets named Hobs which is used as a fail-over in case the browser does not support WebSockets.

jsVNC, CARDE and Guacamole all differentiate in how they encapsulate protocol messages of the RFB protocol. CARDE uses a JSON-RFB encapsulation, Guacamole uses an XML-RFB encapsulation and jsVNC does not use any encapsulation. jsVNC does not use any encapsulation since which minimizes message encoding and decoding overhead.

To support abstract communication in the jsVNC implementation a communication library named HOBS has been implemented. It implements a communication protocol similar in technique to BOSH[7]. The MIFCHO middleware that translates HOBS and WebSockets and aids client-server connectivity is related in functionality to Kaazing Gateway[9] and Orbited[12]. It however also has an additional feature to support access to firewalled resources similarly to Google SDC[6]. MIFCHO unifies solutions to the different connectivity challenges of browser-based applications in a cloud context.

2 Analysis

The starting point for the analysis is to determine the capabilities required for a VNC client. To uncover this an in-depth analysis of the VNC application is provided in section 2.1. In the succeeding subsection the infrastructural challenges of cloud-based network applications are analyzed and the limitations of the Internet-browser are described.

Different techniques to overcoming limitations and implementations thereof are described in subsections 2.3 and 2.4.

At the end of this section the different challenges are concluded upon and a recommandation for a solution-model is provided which is used for the basis of the architecture, design and implementation.

2.1 VNC Application

Virtual Network Computing (VNC) is an application that provides the display of and interaction with, a remote computer over a network. It does so by using client-server based communication where the server sends display-output to the client and the client sends mouse and keyboard input to the server.

Historically VNC was invented at Olivetti Research Labs and was based on the Remote FrameBuffer (RFB[17]) protocol, currently VNC and RFB are officially maintained by a company named RealVNC. Many VNC/RFB implementations has been engineered by others than the original creators and current maintainers. Both proprietary and open-source implementations exist with non-standardized extensions to the original protocol. Offering more features such as file-transfer[15][18]. Even the official maintainers of the RFB protocol provide proprietary VNC implementations.

A VNC implementation named TigerVNC[14] has a high focus on improving the performance of VNC and also provide a community maintained RFB specification as an alternative to the official specification. The contribution of the community maintained specification is that it attempts to document and unify third-party protocol extensions. Alternatives to VNC/RFB are among others NomachineNX[11]/FreeNX[1] and Remote Desktop (RDP[10]).

Engineering all functionality of the community maintained RFB specification is however out of scope for this project. Since focus of this project is on the challenges of engineering browser-based network applications and not with challenges of optimizing the VNC application features and performance. It is however worth noting for future work that a documentation for many protocol extensions exists such that it can be used in the further development of jsVNC.

The most essential messages (illustrated in figure 1) are *frameBufferUpdateRequest*, *keyEvent*, *pointerEvent* send from the client and *frameBufferUpdate* send from

the server. The essential idea of VNC/RFB is to send the display of the server to the client in the form of a *frameBufferUpdate* where the data representing the display is encoded with one of the encodings supported by both the VNC client and server. During a handshake phase the server sends a set of supported encodings to the client and the client can choose to send a subset of the encodings to the server to inform the server of which encodings the client supports. The server is not allowed to send updates with an encoding that the client has not explicitly informed the server that he supports. The RAW encoding however must be supported by a valid VNC client and server, the other encodings are optional.

FrameBufferUpdates with RAW encoding sends display data as a BGRA-bitmap such a representation requires high throughput capabilities of the underlying network.

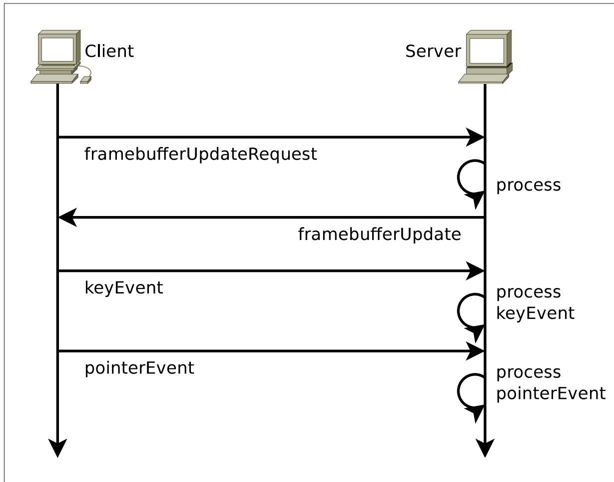

Figure 1: Essential messages of VNC/RFB.

In table 1 an overview of the sizes of framebufferUpdates is given in megabytes and in table 2 and overview of the theoretical maximum amount of *frameBufferUpdates* per second is provided. The updates are updates of the entire display where the display has a resolution as described in the head of the table and a color-depth of 4byte per pixel. The tables gives an indication as to what could be theoretically possible or impossible uses of a VNC-client with RAW encoding.

If the VNC-client accesses a VNC-server playing a video then this would require a frame-rate of about 25fps for stutter-free playback. Playback of HIGH-definition video is thus theoretically impossible. Low-resolution video might be possible when high-throughput interconnects are available.

	480x320	800x480	1024x768	1920x1200
MB	0.585	1.831	3	8.789

Table 1: Size of a single frame-buffer update with RAW encoding.

Mb/S	480x320	800x480	1024x768	1920x1200
1	0.21	0.09	0.04	0.01
10	2.13	0.85	0.42	0.14
100	21.33	8.53	4.17	1.42
1000	218.45	87.38	42.67	14.56

Table 2: Theoretical maximum amount of frame-buffer updates per second with RAW encoding.

Due to these high throughput requirements of RAW encoding it might seem that VNC with RAW encoding is useless on the Internet where interconnects can be as slow 1-2Mbit. The general use of desktop does however have much lower requirements to the rate of updates and can even for many updates settle with only updating small parts of the frame-buffer or re-use existing parts of the frame-buffer that the client already have. The RFB protocol therefore support incremental frame-buffer updates which only send a sub-rectangle of the display to the client and by using the CopyRect encoding the server can instruct the client to copy a rectangle of the frame-buffer to different coordinates. Such uses makes VNC viable on low-throughput interconnects for uses such as text-editing, mail-reading, file-browser and other common tasks.

Another aspect of the protocol design is that it is asynchronous in message delivery. This might seem counter-intuitive by the *frameBufferUpdateRequest* and corresponding response in form of a *frameBufferUpdate*. And it effectively means that a client sending a *frameBufferUpdateRequest* should not expect a *frameBufferUpdate* to arrive on the wire immediately after. The sending of *framebufferUpdates* is regulated by the server and the server chooses when to send the *frameBufferUpdate* message. Additionally the amount of *framebufferUpdates* send by the server is less than or equal to the amount of *frameBufferUpdateRequests*.

This protocol property is however quite essential for the implementation of a VNC client, the client must decide upon a scheme for requesting updates and take into consideration the throughput requirements when deciding upon a polling scheme for the frameBufferUpdateRequests. The engineering of jsVNC is focused on implementing the most essential messaging of the RFB protocol to summarize this involves an application capable of performing the following:

Decode the rectangle-encodings of the *frameBufferUpdate* messages.

Render the decoded rectangles on the local display and copy an area of the frame-buffer to different coordinates.

Grab local mouse input and transform them into *pointerEvent* messages.

Grab local keyboard and transform keystrokes to *keyEvent* messages.

FrameBufferScheme Implement a sensible scheme for sending *frameBufferUpdateRequests*.

How jsVNC handles the above tasks are described in section 4.2. In the following sections the capabilities of Internet-browsers to enable the exchange of RFB protocol messages are discussed.

2.2 Infrastructural Challenges and Browser Limitations

As described in the previous section then VNC/RFB is a network based application and the RFB protocol sets up some requirements to the throughput capabilities of the interconnects. In addition to the requirements induced by the VNC application itself then three essential challenges for engineering a cloud-based network application are:

Deployment In the ideal of cloud computing then jsVNC should be provided as a service on-demand, this poses some infrastructural challenges which are dependent on the architectural choices which is described in further detail in section 3 and section 4.

Connection-Establishment Cloud computing is based on communication over the Internet, the Internet provides a wide range of hosts identified by public IP-addresses which a client can connect to directly. The Internet however also facilitate connectivity with other networks where the hosts in the network do not have public IP-addresses but are connected to the Internet via a gateway and are therefore not directly available from the Internet. jsVNC should able to connect to any host on the Internet and any host on a different network which is somehow connected to the Internet. How connection-establishment is solved is described in further detail in section 3 and section 4.

Browser-Limitations The last essential challenge is that a VNC/RFB client needs a reliable transport protocol such as TCP to exchange messages with the VNC server. Internet-browsers however do not provide access to low-level communication primitives such as sockets via JavaScript. The remainder of this section addresses this problem and describes the communication primitives available in JavaScript and different techniques and implementations of techniques which can potentially be used to *emulate* sockets in JavaScript.

Browser does not provide low-level access to a socket API but they do provide different means for network communication based on the HTTP protocol via JavaScript. Since RFB and HTTP are both application-level protocols an initial idea is to find out if the two protocols have enough similar characteristics such that HTTP could be used to directly emulate RFB by parsing the semantic meaning of HTTP messages differently. An example of this idea is to emulate an RFB *pointerEvent* message in HTTP as illustrated in figure 2.

HTTP is a stateless protocol and the RFB protocol specification describes RFB as being stateless. The state referred to in the RFB specification is the state of the display on the remote screen, not of the protocol. The protocol itself is stateful and several messages are send between client and server in the handshake and initialization phase prior to sending frame-buffer output.

```
POST /pointerEvent HTTP/1.1
...
http-headers
...

button-mask: 00000000
x: 123
y: 321
```

Figure 2: Direct emulation of a RFB pointerEvent with HTTP.

Another more pressing incompatibility is that the HTTP protocol is based on a synchronous request/response messages. The client sends a HTTP Request to the server and the server then sends a HTTP Response back to the client. The HTTP Server is only capable of sending data to the client in the response to HTTP Request, HTTP is in this sense one-way communication. This poses a conflict with RFB protocol which is asynchronous and it must be able to receive messages from the server when data is available such as *framebuffer-Updates* and *serverCutText* messages, RFB requires bidirectional asynchronous communication. It therefore does not seem feasible to directly emulate RFB by applying a different semantic meaning to HTTP messages. Another approach to establishing bidirectional communication must thus be found.

In the following sections two different approaches to obtaining bidirectional communication are described. One approach is based on using different techniques to emulate asynchronous bidirectional communication based on synchronous HTTP. The other approach is based on current work in progress of standardizing a browser-supported communication protocol enabling bidirectional communication.

2.3 Techniques

As previously described then the currently available methods of communication is based on the synchronous HTTP request/response messages. When a page is loaded in the browser it is retrieved by a HTTP request and all resources in the retrieved document is retrieved by further HTTP requests.

To improve the loading time of a page browsers use a combination of HTTP-pipe-lining and utilizing mul-

tiple underlying TCP connections. With HTTP-pipeling a client sends multiple HTTP requests before waiting for the corresponding responses. The advantage of HTTP-pipe lining is that the server can process multiple requests concurrently it must however still return HTTP responses in the same order as the corresponding requests were received. The browser can use multiple underlying TCP connections to partition the set of the HTTP requests, the amount of underlying TCP connections is implementation specific but has historically been limited to two connections per domain, more recently this limit has been increased to six underlying TCP connections.

The problem is that once the browser has retrieved the current document and resources referred to within the document then the server will not send anymore data to the client.

The techniques for initiating retrieving data after the page is finished loading are two-part they use some method to provoke a HTTP request and they utilize features of the HTTP protocol to avoid polling for data but instead let the server push data to the client when data becomes available.

2.3.1 Initiate Retrieval

The techniques for retrieving data from the server evolve around using JavaScript to dynamically add elements to the current document by expanding the Document Object Model (DOM). The element added must as a side-effect require the retrieval of a resource. This can be accomplished by adding an IFRAME to the DOM which will result in a HTTP GET request to the URL in the IFRAME's SRC tag. The retrieved document will then contain a piece of JavaScript code that will be executed upon retrieval. It is a bit more involved to send data to the server using this method but it can be accomplished by expanding the DOM with a IFRAME containing a FORM element, populate the FORM with the data that one wants to send and submitting the form.

Another approach is to use the XML HTTP Requests (XHR), XHR support asynchronous execution of HTTP request by providing a simple interface for constructing requests, sending them and binding event-listeners for responses.

2.3.2 Server Push

When a technique for performing HTTP-requests is chosen a technique for enabling the server to push data to the client instead of forcing the client to poll for data must be decided upon. Two different techniques referred to as *Hanging Get* / *Long Polling* and the other *HTTP Streaming* can be used.

Long Polling involves creating a loop of HTTP requests and letting the server wait with sending it's response until it has data ready. This is essentially still a polling method but with a poll cycle that matches with data being available. The clear advantage is that no unnecessary requests are invoked. There is however a practical limitation as to how long the poll cycle can be allowed to wait. Since the normal behavior of a HTTP request/response is that the server will start sending the response as soon as it has received the client request, when intermediaries such as HTTP proxies sees a HTTP response not sending any data they can choose to close the connection based on a timeout parameter. When using the long poll method it is thus a good practice to negotiate a cycle timeout value which is lower than the most common HTTP proxies. By doing so the Long Poll will simply send an empty response and the cycle will execute another Long Poll.

The *HTTP Streaming* approach utilizes HTTP Chunked encoding. A Regular HTTP response sends data to the client by adding a content-length header describing the length of response-body. With Chunked encoding the content-length header is skipped and the response body is send in chunks. Where each chunk is prefixed with a textual length indicator. The intention of chunked encoding is to support sending responses where the total size of the response is unknown, but it is intended to send finite length responses. Therefore the chunked-encoding has a way to indicate that current chunk is the final chunk. It is therefore not a true data stream of infinite length as the name *HTTP Streaming* indicates but it can be emulated to behave as an infinite stream by never sending the final chunk.

There are many advantages to the *HTTP Streaming* vs *Long Polling*, it can be used for both GET and POST requests, which means that it can be used to emulate a stream from both server to client and also from client to server. There is also a much smaller overhead for each message send with chunked encoding, the only overhead is the chunk-size indicator where *Long Polling* has to ship the entire HTTP request-line and headers for each payload.

HTTP Streaming however has a big disadvantage that intermediaries such as proxies are likely to alter lengths of chunks and buffer chunks until they see the final chunk indication or until an output buffer is filled. Even when no proxies interfere with chunked encoding then browsers can behave in the same manner, such that instead of pushing small amounts of bytes to the browser for rendering. Instead they wait until there a buffer-threshold is exceeded. Such behavior is critical for many networking applications that send many small messages during handshaking/initialization phases such as the RFB protocol.

2.4 Technique Implementations

An umbrella term *comet* has been proposed by software engineer Alex Russel[13] for identifying the previously described techniques. Multiple different implementations of the comet-techniques exists with varying degree of generality and applicability. Two approaches stand out: Bayeux and BOSH[7].

Bayeux is protocol specification for comet-based communication with a wide variety of supported implementations. BOSH is a standardization of bidirectional communication using synchronous HTTP, it has

been developed by XMPP primarily for use with their chat program Jabber since they needed a way to create browser-based clients and also to have a way to tunnel their other protocols over HTTP for firewall traversal.

2.4.1 Bayeux

Bayeux offers a higher-level communication protocol related to the publish / subscribe communication paradigm. It provides a means for web-application developers to implement applications using the semantics of publishing and subscribing events and abstracts all the lower-level issues of the comet techniques.

A message in Bayeux is specified in JSON and has a set of reserved fields (*channel, clientId, id, data, advice, ext, successful, error*) of which only the field *channel* is mandatory. An example of the specification of a Bayeux message is provided in figure x.

```
{
  channel:   "/a/channel",
  data:      "Message payload/Arbitrary Object"
}
```

Figure 3: Example of Bayeux Message.

The channel field defines a communication-channel between client and server, special meta-channels exists for performing protocol handshake(/meta/handshake), event subscription/unsubscription (/meta/subscribe|unsubscribe).

Bayeux is very well-suited for implementing new applications in browser compatible with the publish / subscribe paradigm. Bayeux is maturing and is supported by Java servers such as Jetty.

2.4.2 BOSH

Where Bayeux defines a higher-level publish/subscribe protocol BOSH attempts to stay low-level and instead emulate the semantics of a regular long-lived TCP-connection based on an efficient use of multiple synchronous HTTP request/response pairs without the relying on chunked responses.

Where the Bayeux protocol supports many different Comet-based transports BOSH focuses only on Long Polling and using specific utilization of Long Polling described as the BOSH Technique.

Messages in BOSH are not wrapped in JSON as with Bayeux but wrapped in HTML <BODY /> elements where the attributes of the element are message fields. An example from the BOSH protocol specification is provided in figure 4.

BOSH has some very strong requirements which make it usable in environments such as mobile/browser-based clients, compatibility with proxies that buffer partial responses, backwards compatibility with HTTP/1.0, usable in environments where access to HTTP-headers is denied and many others.

```
POST /webclient
HTTP/1.1
Host: httpcm.jabber.org
Accept-Encoding: gzip, deflate
Content-Type: text/xml; charset=utf-8
Content-Length: 104

<body content='text/xml; charset=utf-8'
  hold='1'
  rid='1573741820'
  to='jabber.org'
  route='xmpp:jabber.org:9999'
  ver='1.6'
  wait='60'
  ack='1'
  xml:lang='en'
  xmlns='http://jabber.org/protocol/httpbind'
      />
```

Figure 4: Example of a BOSH message.

2.5 Wire Protocols

The previously mentioned techniques and implementations thereof have come into existence due the the fact that browser are not capable of performing communication in the way that web-developers need modern web-applications to communicate. The above are one approach to solving the problem of missing communication another approach is to expand the browsers capabilities.

2.5.1 WebSockets

The protocol is a simple text-oriented frame-based protocol, connection setup is initially done by the client sending an initial handshake message compatible with HTTP. WebSockets are not like raw TCP based sockets, TCP based sockets supports streaming where WebSockets are frame-based. Each frame/message send on a WebSocket has an initial frame-type header followed by the payload and depending on the frame-type also a end-of-message character.

The frames in the WebSocket-protocol closely resembles the type of communication made available by using chunked-encoding on HTTP requests. WebSockets however has two clear advantages to the Chunked Encoding technique, they are truly full-duplex requiring only one socket for sending and receiving. Also WebSockets are a standards initiative designed for the purpose of bidirectional communication in the browser, this means that practical implications such as proxies should not choke the communication channel because of misinterpretation of the data exchanged.

The protocol is work-in-progress and constantly changing the latest version is available from [20].

WebSockets are work-in-progress but some browsers have however has implemented different versions of the protocol draft. Firefox 3.7, Chromium, Google Chrome has experimental WebSocket support of what seems to be based on draft-specification 75.

2.6 Security Concerns

Traditional desktop applications are vulnerable to incorrect memory-management which can be exploited as attack-vectors for manipulating the behavior of the application, crashing it or making it execute code residing in other parts of system memory.

Browser-based applications are not concerned with performing accurate memory-management since this is handled by the browser. Browser-based applications are however vulnerable to much simpler methods of manipulating application behavior. One such attack method is called Cross-Site-Scripting (XSS[24]), it exploits applications which does not filter user-input but instead directly sends user-input to the browser for rendering. This can be used to inject HTML, CSS, JavaScript or Flash into the application which will then be executed when rendered by the browser.

Imagine a social networking site which did not perform proper user-input checking, a malicious user could inject JavaScript code into their profile page. Every visitor watching the profile page would then execute the JavaScript code injected by the malicious user, which would enable the script to execute actions in the context of the victim, sending messages to everybody in their social network with messages such as "you are so foo, bar" or other messages that the victim probably did not intend on sending.

Another common threat for browser-based applications are Cross-Site-Request-Forgery (CSRF[23]), it is based on a hostile web-page creating fake requests for a target website. Continuing with the example of the social networking site. A CSRF can be composed by providing image on a hostile site performing a forged request on the social-networking site.

```
1 <img src='http://social-networking/
      updateStatus?status=iAmSoFooBarToday' />
```

To protect against such attacks browsers implement Same-Origin access policies.

2.7 Frame-buffer Rendering

When the complex issues of enabling browser-based bidirectional communication has been solved an equally important problem must be handled: how to efficiently render rectangles of *frameBufferUpdates* in the browser?

Browsers are capable of efficiently rendering images in form of <img> tags and browsers support decoding images of different file formats such as: PNG, GIF, JPEG and some variations of 16bit bitmaps. The 32bit BGRA representation of the RAW encoding using truecolor is however not supported. One approach would be to do real time conversion of the RAW encoded rectangle format such as JPEG or PNG. Doing so in JavaScript would probably be too demanding. Another approach would be to let the middleware perform realtime encoding of rectangles, such a solution has some interesting perspectives.

The middleware could be applied in other scenarios where a native VNC-client supports more advanced encodings such as the TightVNC encoding which is based on JPEG compression. The TightVNC encodings-scheme provides a trade-off between CPU utilization and throughput requirements. By significantly lowering the throughput requirements but also requiring a much higher CPU-utilization on the VNC server. By providing encoding in the middleware the VNC server could use the simple RAW encoding and offload the expensive JPEG compression to the middleware.

Such a solution based on <IMG> tags would however handle incremental *frameBufferUpdates* and *CopyRect* encodings poorly since there is no means for copying a subrectangle of the image contained within the <IMG> tag.

Another approach would be to use the <CANVAS> tag which is made available in HTML5, the canvas supports efficient operations on such as *putImageData* and *getImageData* to copy sub-rectangles and do partial updates of the frame buffer.

2.8 Conclusion

A combined summary of the challenges are provided as a list of recommendations for the Architecture, Design and Implementation of the cloud-based network application jsVNC.

Choosing a method for obtaining bidirectional communication is quite essential, WebSockets are promising they provide a low-overhead and fully bidirectional communication primitive with a clean API. They are however work-in-progress and not very widely support so it would be recommended to create a minimalistic communication library in JavaScript which provide a fail-over when WebSockets are not available. This fail-over library could take advantage of the BOSH specification. BOSH is quite attractive since it is designed to work in a very strict browser-environment and could thus provide for a robust alternative to WebSockets.

The approach of letting middleware handle realtime encoding of *frameBufferUpdate* rectangles has some interesting perspectives but lack support of copying sub-rectangles. The idea of off-loading middleware could provide potential food for though for future work. The canvas element seems like a better recommendation for this project and if the browser does not natively support it the ExCanvas[5] project could be used a failback.

Want to use Canvas, since we rely on canvas, browsers which are new enough to support canvas will also support XHR. However support for WebSockets is still very limited since it is constantly changed so a fallback communication protocol when WebSockets are not available.

3 Architecture & Design

The architectural considerations and design-choices are concerned with client-server connection establishment and enabling browser-based bidirectional communication.

The challenges in client-server connection establishment are that network address translators (NAT) hinders a client from directly establishing a connection to a server behind a NAT-enabled device, since NAT only translates outgoing connections. The simplest solution is to let the NAT-device forward all traffic on a specific port to the server. Such an approach is however quite cumbersome since it requires access to a device which is likely to be out of administrative scope. Another solution is to use hole-punching, a technique used by UDP-based peer-to-peer applications such as VOIP, real-time-games and others. Hole-punching techniques however rely on implementation specific properties of NAT-devices which make them slightly unstable. Work[2] has been made to adapt hole-punching techniques to TCP, their results show that only 64% percent of the tested Nat-devices support the TCP-hole-punching techniques.

Since browsers are not able to establish raw sockets but rely on WebSockets or Comet-based to enable bidirectional communication. Thus a means for performing protocol-translation from WebSockets/Comet-based communication to raw sockets must be provided. This could architecturally be placed as the responsibility of the VNC server, by expanding the implementation of the VNC server to be able to run RFB on top of WebSockets/Comet-based communication. The advantage to this is that jsVNC could connect directly the VNC server without requiring an intermediary to translate protocols. Two architecturally different solution models could be applied in aiding jsVNC:

Decentralized use hole-punching-techniques as described in [2] for establishing client-server communication and implementing protocol translation in the VNC server.

Centralized use a middleware platform for protocol-translation and aiding client-server communication.

The labeling of a centralized vs decentralized architecture is referring to a centralization of the concerns vs decentralization of the concerns. There is nothing that prevents the middleware platform from being implemented in a distributed fashion. The main criteria that marks the architecture as centralized is that all the concerns are centralized in the middleware and not decentralized to be handled by the VNC client and server.

The work in this project is based on a centralized middleware architecture which primary purpose is to perform protocol translation as illustrated in figure 5 and to aid connection establishment as illustrated in figure 6.

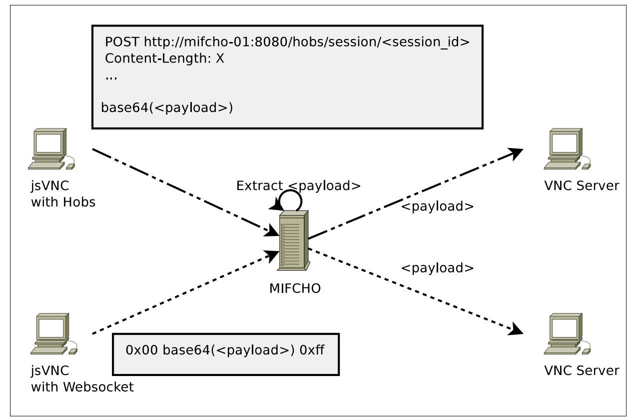

Figure 5: MIFCHO primary purpose: protocol translation.

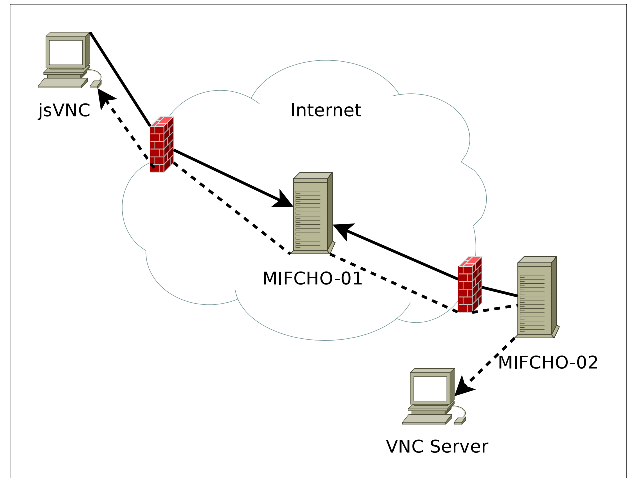

Figure 6: MIFCHO secondary purpose: aiding client-server connection establishment.

4 Implementation

The implementation of jsVNC is multi-part, implementing the actual VNC application is only a small part of the total required engineering effort. The combined engineering effort consists of:

Web-Application The actual jsVNC browser-based application consisting of HTML, CSS and JavaScript that forms the graphical user-interface and implementation of the RFB protocol. The implementation is described in section 4.2.

Bidirectional-communication library and protocol must be implemented since WebSockets at the time of writing are labeled as work-in-progress and therefore have very limited browser-support. Therefore a fail-back for providing bidirectional-communication must be implemented in form of a protocol definition and a JavaScript library to support it. The JavaScript communication library and protocol is named Hobs its implementation is described in section 4.1.

Middleware must be implemented to support the architecture described in section 3. The implementation responsible for these things is named MIFCHO (MI*ddleware* F*or* C*onnection* H*andling* and O*rchestration*) and its implementation is described in section 4.3.

4.1 Hobs

Hobs provides a fail-back for bidirectional communication when the browser does not support WebSockets. Hobs is interface compatible with the WebSocket interface described in [19], this provides for a means of using Hobs as a drop-in replace in an application using WebSockets to provide backward compatible operation. And it is designed to be just that, a simple implementation of a WebSocket.

Hobs uses two concurrent TCP connections to obtain bidirectional message exchange, one connection for sending messages from client to server as described in section 4.1.3 and another connection for server to client messages as described in section 4.1.2. Two connections are necessary since Hobs uses the Long Polling technique in order to be able to receive data. If one only connection was used then the send of a message would have to wait for the Long Poll to finish and hereby adding a high amount of latency for sending data. Therefore two connections are used, one for client to server messages and another for server to client messages.

Where other comet implementations/protocols encapsulate meta-data in the HTTP body and encoding it in either JSON(Bayeux) or XML(Bosh), Hobs encapsulate meta-data in the HTTP request-line and reserve use of the HTTP body only for payloads. The naming convention used by Hobs is illustrated in figure 7.

```
1 /<prefix>/<msg_id>/<arg1>/<arg2>/.../<argN>
```

Figure 7: Meta data path encapsulation.

Each item of meta-data is therefore separated with a "/". All Hobs meta-data has <prefix> which is used used as a namespace-separator such that Hobs can coexist with web servers and other uses of the HTTP protocol on the same host and port. <msg_id> identifies the type of messages carried, valid message-identifiers are *"create"* and *"session"*. The message-identifier is followed by N arguments.

4.1.1 Session Creation

```
1 GET /hobs/create/<rid>/<wait>/<ep_host>/<
     ep_port> HTTP/1.1
```

The Hobs meta-data for initialization consists of a *prefix* followed by the message-identifier *create* indicating session creation and the arguments *request_id*, *wait*, *endpoint_host*, *endpoint_port* and an optional *mifcho_id*. An example of the session creation request/response is provided in appendix B.1.

The session creation argument *request_id* is an arbitrary integer which will be incremented for each message send from Hobs, the *request_id* is meant to be used on the server-side as a way to order incoming messages to ensure correct ordering. The *wait* argument is an integer larger than zero that informs the server-side how long it should wait in seconds before timing out a Long Poll. *endpoint_host*, *endpoint_port* together identify the address of the host which the client wants to communicate with.

The optional *mifcho_id* can be used to indicate that the endpoint address should be contacted via another MIFCHO instance identified by *mifcho_id*. This will establish a tunnel between the current MIFCHO instance and the MIFCHO instance identified by *mifcho_id*.

The meta-data of the session creation request just described is delivered to the MIFCHO instance encapsulated in the PATH part of the request-line of a HTTP GET request as illustrated in figure 8. When the MIFCHO instance receives the creation-request it will attempt to establish a socket to the endpoint specified. It will upon success return a HTTP Response with a *session_id* in the body. The *session*_id is a unique integer generated by the MIFCHO instance which identify the session created.

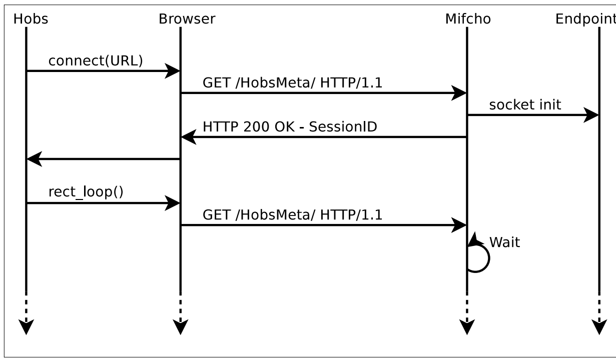

Figure 8: Hobs initialization.

Hobs will on retrieval of the *session_id* initiate a RECV_LOOP implementing a Long Poll which is described in the following section.

4.1.2 Server to Client Messages

```
1 GET /hobs/session/<sid> HTTP/1.1
```

After the session creation the server will send payloads to Hobs encapsulated in the response to a GET request, Hobs provides meta-data in the HTTP GET requests in the form a *prefix*, the message-identifier *session* and finally a *session_id* retrieved during session creation. An example is provided in the appendix section B.3.

To ensure a constant stream of data from the server, Hobs sends the GET requests in a Long Polling loop as illustrated in figure 9. When a payload is received Hobs immediately executes the RECV_LOOP function again to retrieve further payloads. Additionally Hobs executes the function bound to ONMESSAGE.

Notice that the MIFCHO instance can wait in two scenarios, either waiting endpoint payloads or wait for a GET request in which to send the payload. When waiting for endpoint payloads MIFCHO needs to implement the Long Polling timeout but equally important is it for MIFCHO to support buffering of payloads when waiting for Hobs GET requests. Buffering should be bound by an upper limit such data does not clog up at the intermediary.

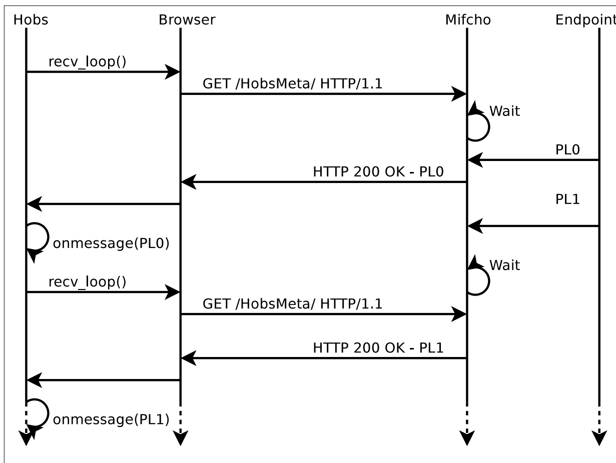

Figure 9: Hobs receiving two payloads PL0 and PL1 from endpoint to Hobs.

4.1.3 Client to Server Messages

```
1 POST /hobs/session/<sid>/<rid+1>/ HTTP/1.1
2 Content-Length: <payload_length>
3
4 <payload>
```

Payloads send from client to server are encapsulated in HTTP POST requests with Hobs meta data consisting of a *prefix*, message-identifier *session, session_id/sid* identifying the Hobs session and an incrementation of the *request_id/rid*. A short example is given in the figure above and a more elaborate example is provided in the appendix section *B.2*.

The handling of the HTTP encapsulation is illustrated in figure 10 where two payloads PL0 and PL1 and send from Hobs. When MIFCHO receives the POST request it extracts the payload and forwards it to the endpoint. Notice that the HTTP response to the HTTP POST occurs immediately and does not provide any information on the delivery of the payload to the endpoint. The HTTP Response is only an indication of whether MIFCHO will eventually forward the payload and as the figure shows then the actual forward can occur immediately prior to the HTTP Response of later after the HTTP Response has been sent.

Hobs is limited to only use two underlying connections, Hobs therefore counts the amount of outstanding POST requests, if a POST is currently ongoing the send will store the payload in an output buffer. When the ongoing POST returns it will immediately POST all payloads in the output buffer.

This behavior has the advantage that it lowers communication latency when the JavaScript application has a high frequency of small payloads. This occurs in VNC when the client must update the cursor position on the server in form of a *pointerEvent*.

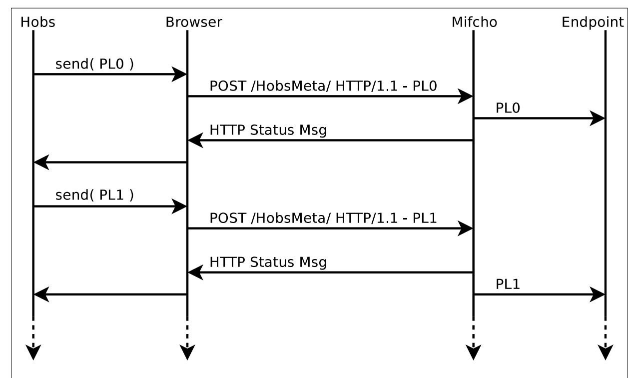

Figure 10: Hobs sending two payloads PL0 and PL1 from Hobs to endpoint.

4.2 jsVNC

jsVNC is organized into multiple layers as illustrated in figure 11. The lower layers consists of an implementation of the Hobs protocol as described in the previous section. The logic required for VNC/RFB messages are encapsulated in its own layer and is built on top of the Hobs library and WebSockets to provide the bidirectional communication primitive. It uses a simple method to check which communication method to instantiate by simply checking for availability which is done i JavaScript as illustrated in figure 12.

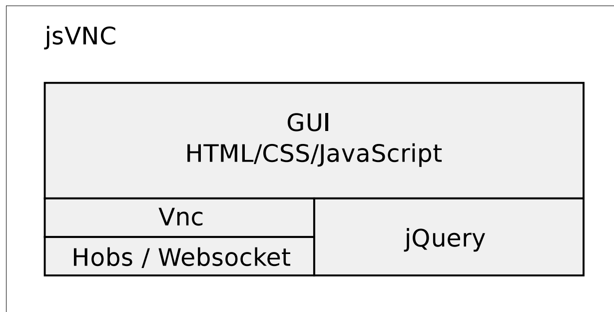

Figure 11: Layered structure of jsVNC.

Many different frameworks for aiding JavaScript application development exists to name a few Google Web Toolkit (GWT), jQuery, Prototype etc. The lower layers of jsVNC are not coupled to any frameworks since it would greatly reduce portability and interoperation of jsVNC, It is only the top-level binding of HTML elements to JavaScript events which utilize a framework. This means that Hobs and Vnc can be reused in different projects and integrated with other browser-based applications without enforcing the use a any particular framework.

```
1 if ("WebSocket" in window) {}
2 else if ("Hobs" in window) {}
3 else return false;
```

Figure 12: Checking for capabilities in JavaScript.

An overview of the code layout can be inspected in table 4, the list of essential capabilities of an VNC-client which was derived in section 2.1 is reproduced here with a description of how the capabilities are implemented. A complete overview of the feature-completeness of jsVNC is provided in table 3.

Decode the rectangle-encodings of the *frameBufferUpdate* messages.
This task is handled by the *process_buffer* in the Vnc layer. Based on the type of encoding used different methods are executed, for the RAW encoding the function *draw_rectangle* is used. *Draw_rectangle* basically decodes the RAW encoding by transforming the BGR representation to RGB and at the same time converts the byte-data to a numerical number which the canvas element can understand.

Render the decoded rectangles on the local display and copy an area of the frame-buffer to different coordinates.
This task is also handle by the *draw_rectangle* method for RAW encoding and for incremental *frameBufferUpdates.* But the essential part is that the frame-buffer uses the canvas tag to render the frame-buffer.

Grab local mouse input and transform them into *pointerEvent* messages.
This task is handled by catching all *onmousemove, onmousedown* and *onmouseup* events and update a global representation of the state of mouse, including position on screen and the bitmask of buttons pressed. This is done to provide a data structure which can be use to poll for the current state of the mouse, this is needed to send correct *pointerEvents.* When receiving onmousemove events the current state of the buttons are unknown if jsVNC in these situations send the incorrect bitmask then features such as drag-and-drop would not be possible.

Grab local keyboard and transform keystrokes to *keyEvent* messages.
The *keyEvent* message is easily mapped to the onkeyup/onkeydown events since the *keyEvent* message simply consists of a *down* flag indicating whether the key was pressed or released and the *key* itself. The *keyEvent* message is however only partially implemented since it requires a manual mapping of an browser-specific integer-values representing a key and the keysym that the VNC server will interpret. The overlapping set of characters are only the alphanumeric characters in the ASCI char-set.

FrameBufferScheme Implement a sensible scheme for sending *frameBufferUpdateRequests.*
This tasks is implemented by the *fbur_poll* function it uses a global *FburPoll* structure with the fields *frequency* (int) and *polling* (bool). fbur_poll calls itself recursively as long as *FburPoll.polling* is true, each recursive call is delayed *FburPoll.frequency* seconds by the use of the *setTimeout* method.
Only the first frameBufferUpdate request is a full request, every succinct request is incremental. It was found that prepending a *pointerEvent* to each *frameBufferUpdate* improved user-perceived performance.

User perceived performance is difficult to measure but one fact is that users expect that interacting with a system would result in a some sort of reacting within a short period of time. If the too long time passed before the application responds with any type of feedback the behavior is interpreted as an error by the user. With RFB this requires that the polling cycle matches with the exact time of when something changes on the server display.

This is very hard to predict on the client side, except for the case when the client does something that could lead to changes in the frame-buffer, such as moving the mouse over an graphical element with hovering effect.

To achieve good user-perceived experience this could be taken advantage of such that each pointerEvent generated by mouse movement is send together with a *frameBufferUpdateRequest*.

This might seem like an excessive amount of update requests and one concern is that *frameBufferUpdateRequests* are 10bytes in size and pointerEvent are only 6bytes in size. Using this technique would indirectly increase the message-size of each *pointerEvent* with ~166 percent.

However the Hobs Encapsulation requires about 513bytes in HTTP Request-line and headers, the actual message-size increase when bundling *pointerEvent* and *frameBufferUpdateRequest* is therefore neglectable due to the protocol overhead of Hobs.

Using Hobs and WebSockets increases the lengths of messages due to payload encapsulation. Payload encapsulation is required in order to safely transfer binary data over XHR requests in environments where the browser does not allow the client to the change of the HTTP-headers and therefore cannot change the mime-type to application/raw. WebSockets also need to perform payload encapsulation, the protocol specification describes a binary framing type, the WebSocket API however does not provide for at any means to enable the use of the binary framing.

Therefore both WebSockets and Hobs use a base64 encoding of payloads to ensure safe transfer. Base64 encoding induces an overhead on the payloads since it transforms three bytes into four and if the message is not a multiple of 3 padding must be used.

It is not possible to accurately define the overhead of Hobs since the HTTP request-line and headers differentiate depending on the browser and the

CHAP.	Feature	Status
6.1.1	Handshake - Protocol Version	OK
6.1.2	Handshake - Security	OK
6.1.3	Handshake - Security Result	OK
6.2.1	Security Types - None	OK
6.2.2	Security Types - VNC Auth.	-
6.3.1	ClientInit	OK
6.3.2	ServerInit	OK
6.4.1	SetPixelFormat	OK
6.4.2	SetEncodings	OK
6.4.3	FrameBufferUpdateRequest	OK
6.4.4	KeyEvent	PARTIAL
6.4.5	PointerEvent	OK
6.4.6	ClientCutText	-
6.5.1	FrameBufferUpdate	OK
6.5.2	SetColourMapEntries	OK
6.5.3	Bell	OK
6.5.4	ServerCutText	OK
6.6.1	Encodings - RAW	OK
6.6.2	Encodings - CopyRect	OK
6.6.3	Encodings - RRE	-
6.6.4	Encodings - Hextile	-
6.6.5	Encodings - ZRLE	-
6.7.1	PseudoEncodingCursor	PARTIAL
6.7.2	PseudoEncodingsDesktopSize	OK

Table 3: Feature-completeness of jsVNC.

Component	Path	Description
Vnc	js/vnc.js	Implementation of RFB messagehandling, sending and GUI bindings
Hobs	js/hobs.js	Implementation of bidirectional Hobs protocol.
jQuery	js/jquery.js	Framework for aiding the graphical user interface.
GUI	vnc.html	HTML and JavaScript for instantiating and binding Vnc to Html elements.
CSS	css/*.css	Stylesheets for HTML presentation.
Images	images/*.png	Images used in the graphical user-interface.

Table 4: Organization of jsVNC code.

4.3 MIFCHO

The idea of MIFCHO is to provide middleware for overcoming the client-server communication establishment issues and protocol translation from Hobs/WebSockets to raw sockets, enabling browser-based bidirectional communication with endpoints being available on the Internet or on private networks connected to the Internet.

The following is a description of the abstract organization of MIFCHO after-which a section is provided describing the MIFCHO protocol which enables daisy-chaining of MIFCHO instances which is the key enabler to solving the connectivity issues. Lastly in section 4.5 a brief description on how to use MIFCHO is provided.

MIFCHO is written entirely in Python. Python has many frameworks to accelerate development and ease the maintenance of networked applications. The third-party Twisted framework is a popular choice and so is the built-in SocketServer framework. In the early stages of development both frameworks were experimented with to see how they could assist the development. It was found that both provide too many layers of indirection, MIFCHO is generally concerned with providing a means for managing outgoing connections and to bind on sockets and handle incoming connections to sockets. Instead of using one of these existing frameworks MIFCHO is implemented as a minimalistic and specialized framework based on the experience

gained from using Twisted and SocketServer. MIFCHO tries not be a framework useful for any type of of network-application such as Twisted and SocketServer. It is instead simply a decoupling of the issues of handling (binding, listening, accepting and tearing down) sockets/connections, and to obtain concurrent processing of multiple sockets/connections by using threads. This is done such that implementing the application-specific logic, such as writing WebSocket to socket translation can focus on just that without being distracted about implementation issues regarding concurrency, connection binding/listening and shutdown.

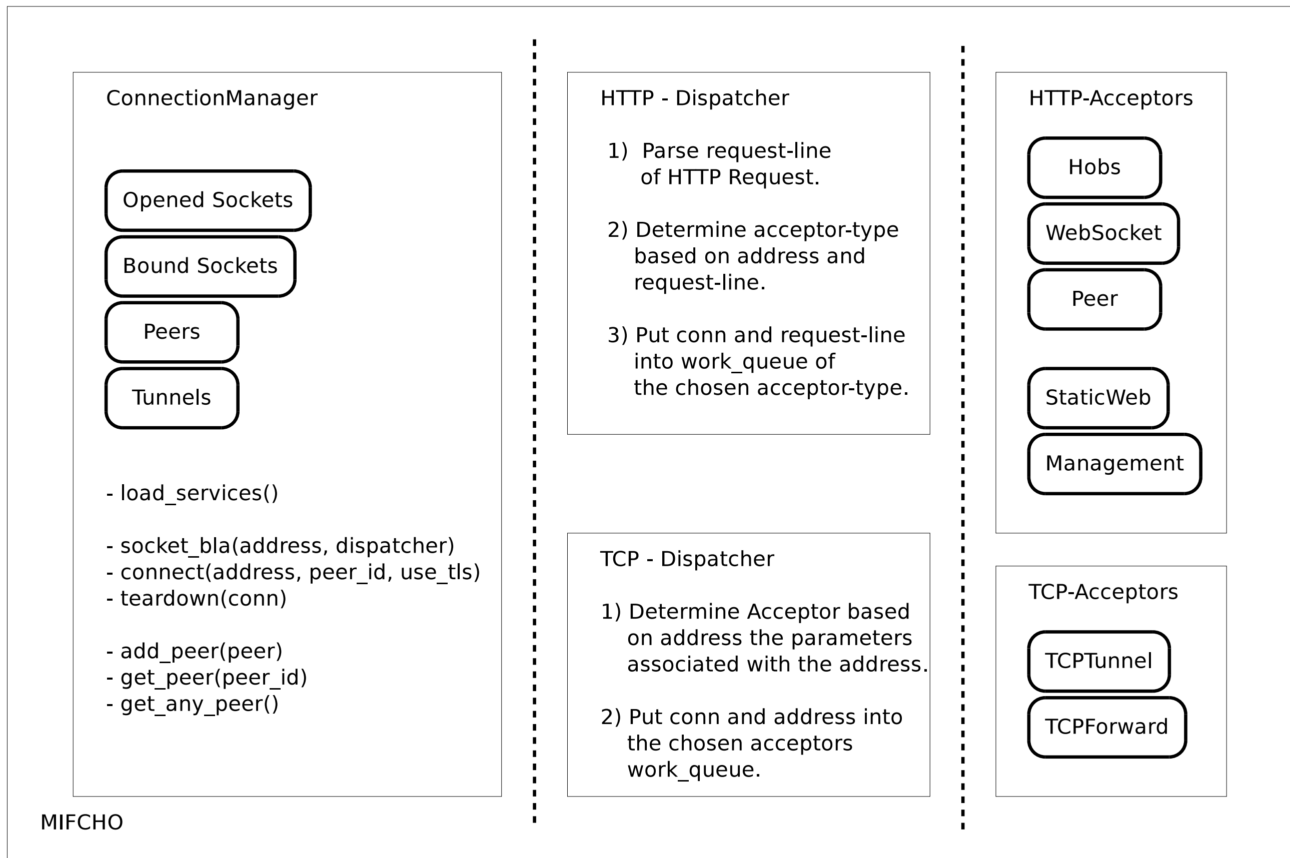

Figure 13: Abstract organization of MIFCHO.

The framework part or non-application-specific part of MIFCHO consists of the classes *ConnectionManager*, *Connector*, *Acceptor*, *Piper*, *Connection*. The organization of MIFCHO is illustrated in figure 13

The *ConnectionManager* is a central entity that provides helper functionality for establishing outgoing connections via the *connect()* method and to bind/listen/accept incoming connections via the *socket_bla()* method. The *ConnectionManager* holds references to all bound and opened connections and provides a means for gracefully tearing down connections via the *teardown()* method. The *ConnectionManager* also maintain lists of *Peers* and *Tunnels.* Peers are other MIFCHO instances which the *ConnectionManager* can use to create connections through. Tunnels are lists of connection pairs (A, B) where the output of connection A is copied to the input of connection B and the output of connection B is copied to the input of connection A.

Acceptors contain application-specific logic. An acceptor is based on the well-known design pattern of a worker-pool. Work in the context of MIFCHO consists of a 3-tuple/triplet on the form:

```
(conn, address, aux)
```

Where conn and address are the result of a socket.accept() call with the addition that the conn is a socket encapsulated in a Connection object. aux provides for auxiliary data.

A skeleton example of implementing an acceptor is provided here:

```
class MyAcceptor(Acceptor):
  def work(self, job):
    (conn, address, aux) = job
        ...
```

For the reader familiar with the SocketServer framework distributed with Python then Acceptors are similar to RequestHandlers. Specifically the work() method of an Acceptor is equivalent to the handle() method of a RequestHandler.

There is however substantial difference in the life-cycle of RequestHandler and Acceptor objects and in their coupling to the source producing the connection which they "*handle*" or "*work*".

Dispatchers are the glue between the *ConnectionManager* and the *Acceptors. Dispatchers* take ingoing

connections as input and contain the logic to route the ingoing connection to the appropriate *Acceptor* and to create the work triplet described earlier. The aux part of the work-triplet can be used to provide additional data to the *Acceptor*. MIFCHO has two dispatchers (*TCPDispatcher* and *HTTPDispatcher*) implementing different dispatching strategies, most interesting is the *HttpDispatcher*. It dispatches request to different *Acceptors* based on the request-line of an ingoing connection containing a HTTP Request. This strategy makes it possible in a simple way to have multiple different uses of the HTTP protocol on the same (host, port) pair.

MIFCHO has several *Acceptors* the Hobs and WebSocket Acceptors performs protocol translation from Hobs/WebSocket to raw sockets. *Acceptors* however are not limited to doing protocol-translation, MIFCHO has three other Acceptors: *Peer*, *StaticWeb* and *Management*. *Peer* acceptor implements part of the protocol described in the following section, *StaticWeb* implements a simple *WebServer* serving static files, *Management* provides performance data such as CPU utilization of the MIFCHO instance. The *StaticWeb Acceptor* provides a convenient way to distribute the jsVNC web-application while maintaining same-origin compliance.

These Acceptors are all based on the HTTP protocol, two other acceptors exists with the purpose of providing simple TCP forwarding and Tunneling.

The final entity in MIFCHO are the *Connectors,* a is the reverse of an *Acceptor*. Instead of waiting for incoming connections the *Connector* itself established outgoing Connections. MIFCHO contains one *Connector*, the *PeerConnector* implements the other part of the MIFCHO protocol.

4.4 MIFCHO Protocol

The primary purpose of the MIFCHO protocol is to enable distinct MIFCHO instances communicate and establish connections via each other.

MIFCHO is a simple protocol with three messages: *handshake*, *tunnel-setup request* and *tunnel-setup response*. The MIFCHO protocol messages are encapsulated in HTTP in an RPC-like fashion with meta-data provided via the PATH of the HTTP Request-line. The encapsulation of meta-data in th PATH was also chosen in the Hobs protocol, this approach is attractive when the amount of arguments are few and provides a simple API when the messages a similar to remote procedure calls. The prefix could be considered an object instance reference, *hello* and *tunnel* method names and everything else "/" separated arguments.

In the following sections the MIFCHO messages are described and to names MFC-0 and MFC-1 are used to identify two MIFCHO instances.

4.4.1 Handshake

```
1 POST /peer/hello/<peer_id> HTTP/1.1
```

The handshake as illustrated in figure 14 is initiated by the entity establishing the connection to the other party. In the illustration this is MFC-1 connecting to MFC-0. MFC-1 identifies itself by sending the hello command containing MFC-1s *peer_id*. The *peer_id* is not a network address but a unique identification of MFC-1.

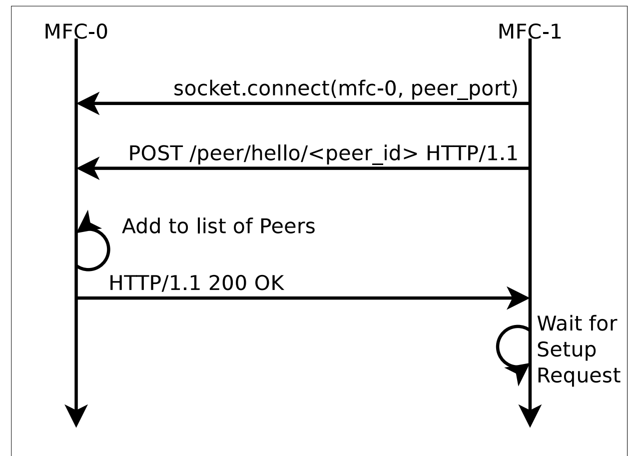

Figure 14: MIFCHO handshake.

When MFC-0 receives the *hello* message it stores a reference to the underlying socket of the HTTP POST request, mapped to by the *peer_id* and sends a HTTP 200 OK to MFC-1. The underlying socket is called the *control connection* of MFC-1. The *control connection* is maintained by MFC-1 meaning that if it should be disconnected it will attempt to reconnect. After MFC-1 has received the 200 OK response to it's handshake method it starts to listen on the *control connection* for tunnel setup requests.

MIFCHO then switches direction of HTTP requests, such behavior is specified under the Reverse HTTP protocol specification.

4.4.2 Tunnel Setup Request and Response

The meta-data for a tunnel setup request consists of *prefix*, fixed message-identifier string *"tunnel"*, followed by a *tunnel_id*, *endpoint_host* and *endpoint_port*. The meta-data for tunnel setup response consists of a *prefix* and a fixed message-identifier string *"tunnel"*.

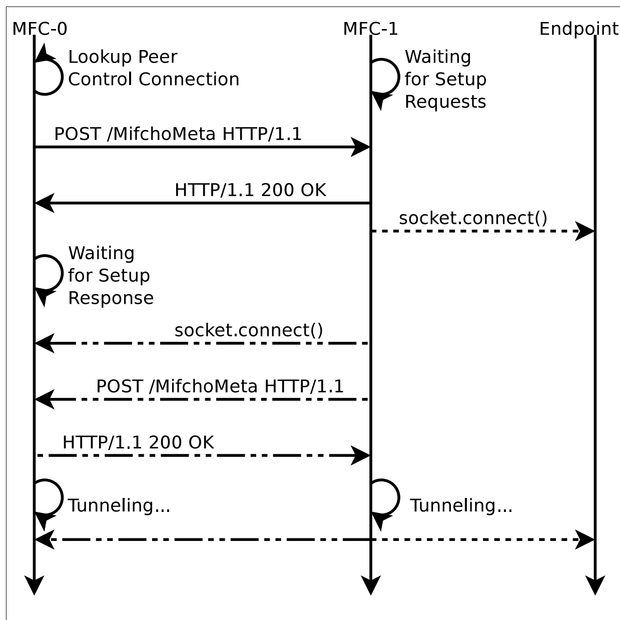

Figure 15: MIFCHO tunnel setup request and response.

As described in the previous section then MFC-1 listens on its control connection for setup requests. When MFC-0 needs to tunnel data through MFC-0 it looks up the control connection of MFC-1 and sends the setup request, MFC-1 responds immediately indicating not that the status of contacting the endpoint but the status of whether the MFC-1 will attempt to contact the endpoint.

MFC-0 then attempts to contact the endpoint specified in the meta-data, after-wards it establishes a new socket with MFC-0 and sends a tunnel setup response on the new socket. When MFC-0 has verified that the tunnel_id provided is valid it responds 200 OK.

MFC-0 and MFC-1 hereafter tunnels all payloads on these two newly created sockets.

4.5 Using MIFCHO

Deploying jsVNC and MIFCHO can be done by:

```
1 cd ~
2 mkdir deploy
3 svn export http://mifcho.googlecode.com/svn/
      trunk/ mifcho
4 svn export http://jsvnc.googlecode.com/svn/
      trunk/src jsvnc
5 cd mifcho
6 # adjust the configuration file
7 python bin/mifcho.py -c etc/web_gw.conf -l var
      /log/web_wg.log -v ERROR
```

Figure 16: Starting a MIFCHO instance with configuration in *web_gw.cfg*, logging to *web_gw.log* at log-level ERROR.

5 Experiments

The experiments should reveal the difference in resource utilization and consumption between using a traditional VNC client and the browser-based VNC client jsVNC.

Two experiments were designed to measure the CPU utilization and memory consumption on the device running the VNC client and another experiment to see the effect of the protocol overhead as described previously. To perform these two experiments tools are needed to be able to: reproducing a desktop interaction, measure CPU/memory utilization and measure the amount of bytes transferred.

Two tools (*record.py* and *play.py*) were developed for reproducing the desktop interaction. *record.py* does as the suggest perform a recording user-input events from the X window system such as mouse-keydown/keyup/move and keyboard keydown/keyup. *record.py* is based on an example application from the Xlib python bindings and enhanced with functionality to store all events to file with a timestamp such that events can be replayed.

The replay of events is performed by *play.py* and in addition to executing events in a timely manner it also samples CPU utilization and memory consumption of a target process.

It was established in the analysis that a VNC client using only RAW, CopyRect and pseudo-encoding is not able to playback video in a decent quality without stutter. The desktop interaction record with *record.py* was therefore a utilization of a desktop where the interactions comprised of a set of operations common to the use of a desktop environment. This included moving the mouse over items to trigger hover-effects, clicking of pop-up menus, writing text in a text-editor, draw a simple drawing in a drawing program, maximize and minimize windows to trigger the transfer of larger rectangles than those transferred with the other uses and lastly drag windows around to provoke the use of the CopyRect encoding.

A screen-recording of the this interaction is provided in the appendix section C, links to an online version of this video demonstration is also available in the appendix.

For the first experiment the desktop interaction was played back with *play.py* using the following VNC clients: TightVNCs vncviewer, jsVNC in Google Chrome using WebSockets, jsVNC in Google Chrome using Hobs and finally jsVNC in Firefox using Hobs.

For the second experiment the previously described desktop interaction is played back with *play.py* but additionally used Wireshark to record the message-exchange between the VNC client and MIFCHO.

The VNC server was in both cases running on a machine with Microsoft Windows XP and the reference VNC-server implementation *VNC Free Edition 4.1* from RealVNC. The desktop was using a resolution 1280x712 with 32bit colors.

5.1 Results

In figure 17 the CPU utilization is plotted as a function of time, the graph shows a peak for all clients in the beginning of the desktop interaction this is due to the server sending the entire frame-buffer to the client requiring the most amount of work during this period of time. The graph of the browser-based clients stabilizes around 30% CPU utilization within about 5 seconds. The graph of the native VNC-client stabilizes around 1% CPU utilization within about one second.

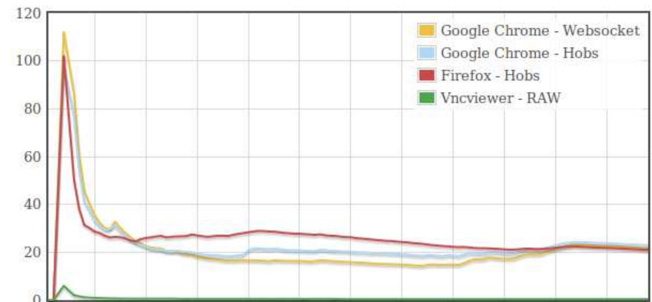

Figure 17: CPU utilization in percent of jsVNC and vncviewer.

The graph quite clearly shows that the browser-based clients have a significantly higher CPU-utilization with a peak of 112% where the native VNC-client never exceeds 6% utilization. It is interesting to observe that there does not seem to be any clear advantage in terms of CPU-utilization in using browser-supported WebSockets.

To uncover what the source of this significantly higher CPU-utilization is a CPU-profiling was run on jsVNC, revealing that 45% of the time was spend of base64 decoding. This explains both why CPU-utilization is so much higher than the native VNC client, since the native client does not need to do any base64 encoding and it explains why there is not anything to gain with the browser-supported WebSockets since the dominant operation of base64 decoding is required for both Hobs and WebSockets.

Figure 18: Memory consumption of jsVNC and TightVNCs vncviewer.

The memory consumption of jsVNC in Google Chrome and Firefox and TightVNCs vncviewer is plotted as a function of time in figure 18 . The graph shows that the memory consumption Google-Chrome

quickly consumes about 3.5MB which corresponds somewhat to the size of the frame-buffer = 1280*712*4 / (1024*1024) ~= 3.47MB. The graph rises at the end of interaction which complies with the maximization and minimization performed which leads to somewhat larger *framebufferUpdates*. It is mildly surprising to see that native vncviewer uses less than a MB of memory. The graph again clearly shows that the browser-based VNC client has a significantly higher resource consumption than the native client.

The amount of bytes transferred in absolute values and relative values to the native client is provided in table 5.

Bytes	Send	Received	Total
Vncviewer	706312	60903223	61609535
jsVNC WS	777814	74756330	75534144
jsVNC WS %	10.1%	22.7%	22.6%
jsVNC Hobs	4109996	81286663	85396659
jsVNC Hobs %	481.8%	33.4%	38.5%

Table 5: Bytes transferred in absolute values and percentile increase in relation to the native client.

The graph shows the total amount of bytes transferred for jsVNC using WebSockets, jsVNC using Hobs and for TightVNCs native VNC-client. The graph shows that Hobs has a higher overhead than WebSockets. This is also to be expected since Hobs encapsulates payloads in HTTP request/response pairs and WebSockets only use a constant four bytes for the beginning/end of message indicators.

The data a client receives with RFB mostly contains large messages where the data a client sends with RFB is mostly small messages. Since a client receives *frameBufferUpdates* and sends frameBufferRequests and *pointerEvent*/*keyEvents*. The table shows that the overhead is most significant when sending packages due to the fact that the sending is comprised of many small messages.

6 Conclusion

In this paper a case study of engineering cloud-based networking applications has been conducted. Cloud computing provides a means for seamlessly making computing resources available as a service, on-demand, everywhere. The work in this paper studies the engineering challenges of making a VNC client available as a service, on-demand in an Internet-browser.

A VNC client has been engineered and experiments show promising results that it is feasible to engineer network applications in the browser which require high throughput and low latency. Experiments however also show that the price of cloud-based VNC client is paid with significantly higher CPU utilization, memory consumption and bandwidth consumption than a traditional VNC client.

The work in this paper also analyze the challenges of enabling socket-like communication for the browser and a middleware platform for aiding connection-establishment, enabling protocol translation and performing browser-based application deployment has been implemented.

It was also found that one of the contributors to significant higher CPU-utilization of the browser-based VNC client is base64 decoding of data. This discovery stressed the importance for future WebSocket APIs and protocols to support binary framing to safely and efficiently transport binary data in browser-based applications.

6.1 Future Work

Experiment with implementing more encoding-schemes based on the community maintained RFB specification. The TIGHT encoding-scheme is based on JPEG compression this is interesting since Internet-browsers are well-equipped for rendering JPEG images. It could be interesting to evaluate the difference in performance of jsVNC with TIGHT encoding vs the RAW manually handling decoding.

It was observed during the experiments that a large part of the CPU-utilization was caused by base64 encoding and decoding of messages, it could be interesting to find a way to enable Hobs to transfer binary data in a safe way without requiring the base64 encoding.

Another observation was made that MIFCHO could be used as an offload proxy for the CPU-intensive tasks of JPEG encoding *frameBufferUpdates*.

References

[1] FreeNX. http://freenx.berlios.de/, 2010.

[2] P. S. Bryan Ford and D. Kegel. Peer-to-peer communication across network address translators. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 13–13, Berkeley, CA, USA, 2005. USENIX Association.

[3] carde. carde - Pure Web Standards based Remote Desktop. http://code.google.com/p/carde/, 2010.

[4] M. Fucci. FlashLight-VNC. http://www.wizhelp.com/flashlight-vnc/, 2010.

[5] Google. ExplorerCanvas. http://excanvas.sourceforge.net/, 2010.

[6] Google Inc. Google Secure Data Connector. http://code.google.com/securedataconnector/, 2010.

[7] P. S.-A. J. M. Ian Paterson, Dave Smith. XEP-0124: Bidirectional-streams Over Synchronous HTTP (BOSH). XEP-0124 (Informational), 2009.

[8] jssockets. jssockets. http://code.google.com/p/jssockets/, 2010.

[9] Kaazing. Kaazing Gateway. http://www.kaazing.org/confluence/display/KAAZING/Home, 2010.

[10] Microsoft. Windows Remote Desktop. http://support.microsoft.com/default.aspx?scid=kb;EN-US;q186607, 2010.

[11] NoMachine. NX. http://www.nomachine.com/products.php, 2010.

[12] Orbited. Orbited. http://orbited.org/, 2010.

[13] A. Russell. Comet Low Latency Data for the Browser. http://alex.dojotoolkit.org/2006/03/comet-low-latency-data-for-the-browser/, 2010.

[14] TigerVNC. TigerVNC. http://tigervnc.org/, 2010.

[15] TightVNC. TightVNC. http://www.tightvnc.com/, 2010.

[16] TightVNC. TightVNC Java Viewer. http://www.tightvnc.com/ssh-java-vnc-viewer.php, 2010.

[17] R. L. Tristan Richardson. The RFB Protocol. www.realvnc.com/docs/rfbproto.pdf, 2009.

[18] UltraVNC. UltraVNC. http://www.uvnc.com/, 2010.

[19] W3C. The WebSocket API. http://dev.w3.org/html5/websockets/, 2010.

[20] W3C. The WebSocket Protocol Specification - Latest. http://www.whatwg.org/specs/web-socket-protocol/, 2010.

[21] S. G. Ware. Guacamole. http://sourceforge.net/projects/guacamole/, May 2010.

[22] S. G. Ware. Java Socket Bridge. http://stephengware.com/projects/javasocketbridge/, 2010.

[23] Wikipedia. Cross Site Request Forgery. http://en.wikipedia.org/wiki/Cross-site_request_forgery, 2010.

[24] Wikipedia. Cross Site Scripting. http://en.wikipedia.org/wiki/Cross-site_scripting, 2010.

A MIFCHO Configuration Example

```
1  [Hobs Gateway]
2  url=http://tile-0-0.local:8000/hobs
3  instances=15
4  component=HobsAcceptor
5
6  [Websocket Gateway]
7  url=http://tile-0-0.local:8000/wsocket
8  instances=15
9  component=WebsocketAcceptor
10
11 [Peer Interface]
12 url=http://tile-0-0.local:8000/peer
13 instances=15
14 component=PeerAcceptor
15
16 [Management Interface]
17 url=http://tile-0-0.local:8000/admin
18 instances=15
19 component=ManagementAcceptor
20
21 [jsVNC App Deploy]
22 url=http://tile-0-0.local:8000/jsvnc
23 instances=15
24 component=StaticWebAcceptor
25 path_prefix=../jsvnc/src
26
27 [TCP Forward]
28 url=tunnel://tile-0-0.local:5900/localhost/59000
29 instances=15
30 component=TCPForwardAcceptor
31
32 [TCP Tunnel via Peer]
33 url=tunnel://tile-0-0.local:8001/1234/localhost/59000
34 instances=15
35 component=TCPTunAcceptor
```

Figure 19: Example MIFCHO configuration file.

B Message Samples

B.1 Hobs Session Creation

Session creation request of 443 bytes.

```
1  GET /hobs/create/3527051141/50/jsvnc-01/59000 HTTP/1.1
2  Host: tile-0-0:8000
3  User-Agent: Mozilla/5.0 (X11; U; Linux x86_64; en-US; rv:1.9.2.5pre) Gecko/20100528 Ubuntu
       /10.04 (lucid) Namoroka/3.6.5pre
4  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
5  Accept-Language: en-us,en;q=0.5
6  Accept-Encoding: gzip,deflate
7  Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
8  Keep-Alive: 115
9  Connection: keep-alive
10 Origin: null
```

Session creation response of 110 bytes.

```
1  HTTP/1.1 200 OK
2  Content-Length: 39
3  Access-Control-Allow-Origin: *
4
5  198118126074926987294597228863060066306
```

B.2 Hobs Sending Message

Sending a *pointerEvent* and *frameBufferUpdateRequest*, total message-length: 595 bytes.

```
1  POST /hobs/session/198118126074926987294597228863060066306/3527051188 HTTP/1.1
2  Host: tile-0-0:8000
```

```
3  User-Agent: Mozilla/5.0 (X11; U; Linux x86_64; en-US; rv:1.9.2.5pre) Gecko/20100528 Ubuntu
       /10.04 (lucid) Namoroka/3.6.5pre
4  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
5  Accept-Language: en-us,en;q=0.5
6  Accept-Encoding: gzip,deflate
7  Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
8  Keep-Alive: 115
9  Connection: keep-alive
10 Content-Type: text/plain; charset=UTF-8
11 Content-Length: 24
12 Origin: null
13 Pragma: no-cache
14 Cache-Control: no-cache
15
16 BQAAAAAAAwEAAAAABQACyA==
```

Session usage response of 70bytes:

```
1  HTTP/1.1 200 OK
2  Content-Length: 0
3  Access-Control-Allow-Origin: *
```

B.3 Hobs Receiving Message

Request

```
1  GET /hobs/session/198118126074926987294597228863060066306 HTTP/1.1
2  Host: tile-0-0:8000
3  User-Agent: Mozilla/5.0 (X11; U; Linux x86_64; en-US; rv:1.9.2.5pre) Gecko/20100528 Ubuntu
       /10.04 (lucid) Namoroka/3.6.5pre
4  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
5  Accept-Language: en-us,en;q=0.5
6  Accept-Encoding: gzip,deflate
7  Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
8  Keep-Alive: 115
9  Connection: keep-alive
10 Origin: null
```

Response

```
1  HTTP/1.1 200 OK
2  Content-Length: 307200
3  Content-Type: text/plain
4  Access-Control-Allow-Origin: *
5
6  <partial-framebufferupdateresponse-base64-encoded>
```

B.4 WebSocket Initialization

HTTP compatible setup request.

```
1  GET /wsocket/1234/jsvnc-01/59000 HTTP/1.1
2  Upgrade: WebSocket
3  Connection: Upgrade
4  Host: tile-0-0:8000
5  Origin: null
```

Response:

```
1  HTTP/1.1 101 Web Socket Protocol Handshake
2  Upgrade: WebSocket
3  Connection: Upgrade
4  WebSocket-Origin: null
5  WebSocket-Location: ws://tile-0-0:8000/wsocket/1234/jsvnc-01/59000
6  WebSocket-Protocol: sample
```

B.5 WebSocket Send

Sending a *pointerEvent* and *frameBufferUpdateRequest*, total message-length: 26 bytes.

```
1  00BQAAAAAAAwAAAAAABQACyA==FF
```

C Physical Medium

The source-code for jsVNC and MIFCHO are provided on the enclosed physical medium. The content of the medium is organized as described in table 6.

Path	Description
/jsvnc/*	All source code related to the jsVNC browser-based application.
/mifcho/*	All source code related to the MIFCHO middleware.
/demo/jsvnc_chrome.avi	Video demonstration of jsVNC in H.264 encoding.
/demo/jsvnc_chrome.ogv	Video demonstration of jsVNC in OGV format.
/demo/screenshots/*.png	Screen-shots of jsVNC.
/report.pdf	A PDF-version of this document.

Table 6: Organization of physical medium.

C.1 Online

The resources described above are also available online via Googles project-hosting service and the video is available on Youtube. Links are provided in table 7.

	URL
MIFCHO	http://code.google.com/p/mifcho/hosted
jsVNC	http://code.google.com/p/jsvnc/
Video-Demo	http://www.youtube.com/watch?v=TocE4MzsD-c

Table 7: Online availability of ressources.