X server in web browser

PEREZ HERNANDEZ Daniel
<tuvistavie@dcl.cs.waseda.ac.jp>

Thesis submitted in partial fullfillment of

the requirements for the degree of

Bachelor in Information and Computer Science

Student ID	1W09C753-8
Submission Date	August 27, 2013
Supervisor	Tatsuo Nakajima

Department of Computer Science
School of Science and Engineering
Waseda University

Abstract

Most new users to Linux have a lot of problems to set up a Linux installation, which is however often necessary for students in computer science. Using a remote server with already all the necessary tools prepared would therefore be a good alternative. However, SSH is the most common way to connect to a remote server, and there is no secure way to connect to a remote server with a graphical user interface. However, having to do everything over SSH would probably be too difficult for someone who is not yet used enough to programming and UNIX/Linux. The purpose of this research is to develop an X server running in any web browser and able to connect to a remote UNIX/Linux server, so that one could access a remote server using a GUI without any need to install anything on his or her computer.

Contents

List of Figures

Listings

Preface

This report is written in partial fulfillment of the requirement for the degree of Bachelor in Computer Science.

The project presented here is still under active development and will continue to be at least during the next six months. When writing this paper, the application is at a state where the main framework is operational and basic examples can be run, but is still very far from being able to respond to a real world use case. The integrity of the project source code is available on GitHub and under MIT license.

Chapter 1

Introduction

1.1 Motivations

1.1.1 Easy access to Linux

From its first releases, and during several years, Linux has been quite complicated and not very intuitive to use (eg. no graphic installer) for beginners. In the last few years, Linux has become much simpler to install, and use in general, with distributions such as Ubuntu or Fedora integrating a user friendly installer and even tools such as Wubi[1] to make the installation possible from Windows.

However, even if the installation process has become much simpler, the fact is that a lot of beginners are still having a lot of problems to get a usable Linux environment. While teaching C programming language as a teacher assistant, I wrote on my personal blog all the steps to get a usable Linux environment, using the possibly simplest setup: giving a disk image file to open with VirtualBox. Despite this, a large number of students still could not get Linux to work properly.

One of the greatest motivation for this project was therefore to create a system to make Linux available to anyone, even with no computer knowledge at all.

1.1.2 Access from mobile devices

Another interesting possibility for this project is a mobile access from any modern phone or tablet to a remote machine. By using this system, one could check anything, for example the status of a running task, on a given machine without having to create or use a dedicated API for it. This could be useful when the creation of a dedicated tool is not worth.

1.1.3 Evaluation of new web technologies

Web technologies have been evolving during the last few years, and a lot of new tools and technologies have been released. With the introduction of websockets, a full duplex communication between a browser and a web server have become possible. The last

[1] `http://www.ubuntu.com/download/desktop/windows-installer`

motivation for this project was to try and evaluate these new web technologies, in particular websockets, to see how well it can be integrated in a resource demanding project.

1.2 About the project

We will here be giving a short overview of what was tried in this project, what can be done and what still needs to be done.

1.2.1 Achieved tasks

One of the main task of this project was to make a communication possible between an X client and a modern browser. This is now possible and though not all X requests and events are supported yet, they can be added simply enough. A simple application can already be ran using the system, although the number of supported requests and replies still very limited.

1.2.2 Tasks to be achieved

Still only very few requests, replies and events are supported by the actual system. In order to be able to respond to real world requests, not only those requests will be needed, but some requests and other messages that are not defined by the core X protocol will be needed. We are now working on extending the base system to support these messages, but given the number of message that must be handled, the stable release of these is going to take some time.

1.3 Chapters summary

In chapter 2, About X protocol, we will not directly describe the project itself but rather give some general explanations about the X protocol, which the project is based on. We will describe the fundamental concepts and the key notions of the protocol so that the reader can have a sufficient understanding of it to continue reading through this paper.

In chapter 3, General architecture, we will give a general overview of the project and its architecture. We will explain how the project is split, as well as the responsibilities of the different part of it.

In chapter 4, Server frontend application, we will explain how this part of the project is working, and explain more in detail what it is supposed to do and how. We will also be discussing about the login manager part of the application which is handled in this application.

In chapter 5, Server backend application, we will give more detailed of how the system is communicating with the X clients, and how the data received and sent to the X clients is handled.

In chapter 6, Web browser application, we will describe how the frontend roles of the X server, rendering and event handling, are handled and implemented.

In chapter **??**, Evaluation, we will discuss about the general performances of the system, principally in a qualitative manner, and will also try to evaluate it from an end-user point of view.

In chapter 8, Related work, we will present two projects that are in some way related to this project and have some goals in common with it.

Chapter 2

About X protocol

The whole application is using the X protocol[4], we are therefore going to give a brief presentation of this protocol, for the reader to be able to understand the concepts in this paper.

2.1 Generalities

The X protocol is a network-transparent protocol for bitmap display. It uses a client-server architecture, running the server on the computer with the display, and makes the connection with local and remote clients possible. It is a binary protocol and can be built on top of any reliable byte stream (eg. TCP).

2.2 Messages

The protocol uses four main types of messages to communicate between the server and the clients. Those types are

Requests A request is the basic way to communicate from the client to the server and can be used either to query for some information or to update the display. Some requests need a reply while other do not. Requests needing a reply can be asyncrhonous or not.

Replies A reply is the basic way for the server to send back information to the client, and is usually used to send information about the server.

Events Events are generated by the server and sent to the client to notify it of some event in the handled display. Event can be used for graphic events (eg. when a window has been rendered) or for device events (eg. keyboard or mouse event).

Errors Errors can be generated from the client as well as the server for many different reasons. One of the main reason for errors generated from the server side is for access to non available resources.

An overview of how these messages are sent is shown in figure 2.1.

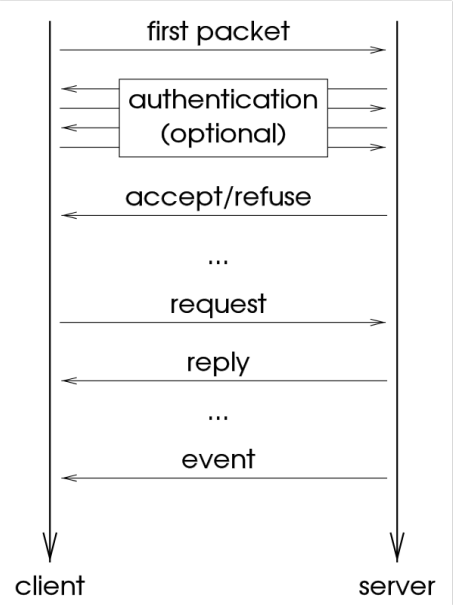

Figure 2.1: Overview of the X protocol [1]

2.3 Resources

The protocol uses a number of different resources which we will shortly introduce here.

Drawable A drawable is an abstract entity used to draw. It can be a window or a pixmap.

Window There are two types of window: top-level window and subwindows. A top-level window is the main container for an application and usually contains all the other components of the application. A subwindow is a window contained in the top-level window of an application and can be used for anything, from the titlebar to a button in the application.

Pixmap A pixmap is a region used to draw, but on opposite to a window, it is not shown until explicitly requested. The content of a pixmap or part of it can be displayed on a window.

Graphic context A graphic context is a structure containing basic graphic information to apply to a given request, for example the background and foreground colors, or a transformation to apply to the shape to draw.

[1] Diagram from Wikipedia X Window System Core Protocol

2.4 Example

To end up with the presentation of the X protocol, we will here give a short example of a communication between an X server and an X client. In this example, we will explain what happens at the protocol level when the following code, using Xlib[2], is ran.

```
1  #include <X11/Xlib.h>
   #include <stdio.h>
3  #include <stdlib.h>
   #include <string.h>

   int main(void)
7  {
       Display *display;
9      Window window;
       XEvent event;
11     char *msg = "Hello, World!";
       int s;

       display = XOpenDisplay("localhost:2.0");
15     if (display == NULL)
       {
17         fprintf(stderr, "Cannot open display\n");
           exit(1);
19     }

21     s = DefaultScreen(display);
       window = XCreateSimpleWindow(display, RootWindow(display, s), 10, 10,
           200, 200, 1, BlackPixel(display, s), WhitePixel(display, s));

       XSelectInput(display, window, ExposureMask | KeyPressMask);
25     XMapWindow(display, window);

27     while (1)
       {
29         XNextEvent(display, &event);

31         if (event.type == Expose)
           {
33             XFillRectangle(display, window, DefaultGC(display, s), 20, 20,
                   10, 10);
               XDrawString(display, window, DefaultGC(display, s), 50, 50, msg
                   , strlen(msg));
35         }

37         if (event.type == KeyPress)
               break;
39     }

41     XCloseDisplay(display);

43     return 0;
   }
```

Listing 2.1: Simple example of an X application

[2] Xlib is a library to implement client application using the X protocol. It takes care of all the low-level details of the protocol, such as byte-swapping, or choosing ids for allocated resources.

2.4.1 Connection setup

On line 14, the client tries to open the screen 0 of display number 2 for the X server running on `localhost`. In the X window system, for TCP connections, like in this case, the display number 0 should be listening on the port 6000, and other display should increase the port number by 1. Therefeore, the display number 2 is supposed to be listening port 6002. If no connection can be established to this host and port, the program fails. If the connection is established, the client send connection information to the server. The client first specifies the byte-order (LSBF or MSBF), then the major and minor version number of the used X protocol, and finally if needed, authorization protocol name and data. If the connection is rejected, the server sends back a message indicating this, as well as the reason for rejecting the connection. If the connection is accepted, the server sends back a message with the necessary server and screen information.

2.4.2 Requests

The `XOpenDisplay` function actually offers some abstraction for the end user and does not only initialize the connection but also query for extensions, and setup a basic graphic context.

The first request to be sent is a `QueryExtension` request, querying for the `BigRequest` [3] extension. The client then blocks until receiving a reply from the server. This is however not decided by the protocol itself, and depending on the request and reply involved, the client may choose to send the request asynchronously.

The next request to be sent is the request to create a default graphic context. The created graphic context is the one that will be used when the client uses the `DefaultGC (Display*, int)` function.

The client then creates a window, using the information received during the connection initialization, such as the root window and the black and white pixels information. This generate a `CreateWindow` request which does not need a reply, the client therefore sends the next request straight away.

`XSelectInput` is used to select the kind of events the client is listening. This is important to avoid receiving events that will never be hanlded. However, this request does not exist in the X protocol and `ChangeWindowAttributes` request with the `event-mask` set with the proper value is sent instead.

Finally, a `MapWindow` request is sent to map the created window on the screen.

[3] The `BigRequest` extension is an extension to allow requests of more than the 262140 bytes.

2.4.3 Events

The program works around a simple event loop, in which only two kinds of events are handled: the `Expose` and `KeyPress` events. The `XNextEvent` is a function that blocks the execution of the program until an event is received from the server, and set the event when received.

The first received event should be the `Expose` event, which is generated when a window is shown on the screen, so that should be generated if the `MapWindow` request was handled properly.

The other handled event is the `Keypress` event, which is generated whenever a key is pressed on the keyboard the server is listening. In this example, the `Keypress` event is not really handled, in the sense that neither the keycode nor the keysims are checked and the program simply breaks from the event loop, but the event does contain all the necessary information in order to properly handle it, and those information are usually indeed checked in a real world application.

2.4.4 Closing the display

As the client creates a number of resources while running, here a graphic context and a window, those needs to be destroyed when the client program ends. The `XCloseDisplay` is a high-level function that takes care to send the appropriate `DestroyWindow` and `FreeGC` requests to the server in order for the resources to be freed properly.

Chapter 3

General architecture

In this project, everything is based on the X protocol, but given the requirements of the system, there are a lot of things that needed to be handled in alternative ways. The core X protocol has only remained in the backend of the system, while the other parts and especially the frontend used in the browser is relying on much more modern technologies.

We will here discuss about the general architecture of this project, including the differences with the basic X protocol as well as their reason to be.

3.1 General overview

We will here be giving a general overview of the project.

The project is mainly divided in three different parts, and the general communication pattern is shown in figure 3.1.

Server backend This application main role is to communicate with the X clients, to transfer the requests sent by the X clients to the frontend of the system, and to transfer back the replies and events sent by the frontend to the X clients. Another important role of this application is to cache information held by the client and the frontend in order to respond directly to the X clients and avoid unneeded communications with the frontend.

To illustrate this caching system, let us think about a requests sent by a client to get the background and foreground pixels of a particular subwindow (the root window information being sent on connection). If the backend server was not doing any cache work, the request would have to be sent to the frontend application and then to the client web browser to get a reply. This would of course increase latency. By using the cache system, the reply can be handled by the backend and is therefore almost instant.

Server frontend One of the roles of this application is to act as a bridge to communicate between the backend application of the system and the web browser JavaScript application. Another of its role is to act as a login manager. The application receives an HTTP request with the client Linux/UNIX creditentials check those creditentials,

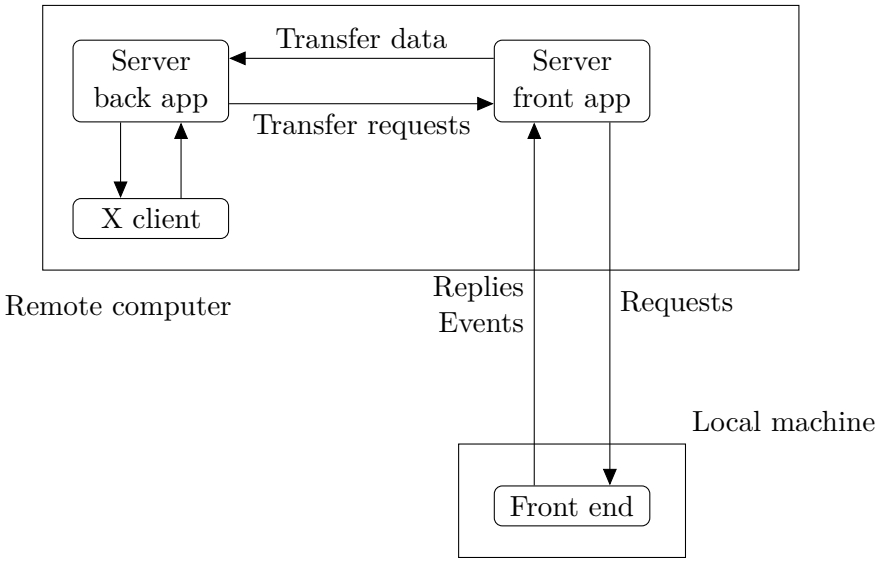

Figure 3.1: System general communication pattern

and on success, initializes the server backend for the particular client as well as the websocket connection to communicate with the web browser.

Web browser application The web browser application is the application running in the client browser, and has several responsibilities in the system. The first responsibility of this application is to handle the requests received from the server and to render them in the web browser. The second responsibility is to handle events occuring in the browser, and to send them back to the server, for them to be handled by the X client.

3.2 Differences with a normal X server

In this section, we will discuss between the main differences of this project with a normal X server.

3.2.1 Divided server

In a normal X client/server architecture, the client connects to the server, and communicates directly with it. The X server needs to have control over the keyboard, mouse, screen, and other hardware used to display graphics, and handle events.

This system is very different as first of all, there is not really *an* X server. Given the system requirements, the application that communicates with the client and the application that listens to the events cannot be the same; handling all the TCP directly in the

web browser is not realistic. There is therefore a need to communicate with clients, and make communication possible with the application acting as the X server frontend.

3.2.2 Communication through websockets

As a consequence of the above difference, another difference is that to implement this system, we need not only to communicate with X clients using a bytestream such as TCP sockets, but we also need to use some communication protocol to communicate with a web browser. The chosen protocol to achieve this is websockets, we will be discussing further about this decision later in this paper.

3.2.3 Integrated login manager

This project goes further from a normal X server in the sense that the login manager is integrated and therefore does not depends on the X Display Manager Control Protocol.

3.3 Language and tool choice

The web browser application is written in JavaScript, and we made the choice to avoid any DOM libraries that could potentially slow down the application and would not be very useful in this context as the application is almost only using Canvas.

For rendering Canvas, the application uses KineticJS[1]. KineticJS is a Canvas library that allows layering and easy event handling.

The application running on the remote server has been almost entirely developed in Scala. In this section, we will try to discuss objectively about the motivations for choosing this language. Some of the arguments given are based not only on Scala but also on the library Akka which integrates very well with the design and syntax of Scala.

In the following paragraphs, we will only give some quite abstract arguments, to avoid entering in too many details.

3.3.1 Multi-threading

The system is developed to of course support several X client connections for a single user, but also to accept several user at the same time. This therefore requires a fairly high amount of threads.

Multithreading is often the source of numerous mistakes when used naively using libraries such as pthread for C and C++, or `java.lang.Thread` for Java. One of the main reason for this is that it is quite difficult to efficiently keep trace of the threads

[1] `http://www.kineticjs.com/`

accessing a given resource, and this can sometimes lead to deadlocks, starvations, or other undesired effects[3].

To overcome this problem, Scala makes extensive use of the actor model, where basically only a single actor (thread) ever accesses the resource and the other actors query this actor by sending messages, and eventually waiting for responses from it. An actor assures to treat only a single message at a time, and to treat messages in order they arrive. If a message arrives while the actor is treating another message, this message will be queued, and treated afterwards. By using this model, access to a resource can be controled very easily.

Of course, Scala is not the only language using the actor model. To give an example between others, Erlang uses this communication system extensively. Furthermore, the library used in this program, Akka, is also available for Java, however the impossibility of writing DSL like code in Java due to the inexitance of operator overloading makes the general syntax less consise and harder to read.

3.3.2 Inter-process communication

This system uses a lot of Inter-process communication(IPC) to communicate between the backend and frontend applications on the server machine. How easy this communication is made was also an important criteria in the choice of the language and library.

Based on the above model, inter-process is made very easy, using the same model over a TCP connection. The Akka library allows a great abstraction for the programmer, and while IPC would generally require to manually serialize data, and make some other verification can be done in a completely transparent fashion.

Chapter 4

Server frontend application

The server frontend is the part of the application that ensures the communication between the backend and the web browser is done properly. It communicates through TCP sockets with the backend and through web sockets with the web browser.

It also has the important role of login manager. On a client connection, it first checks the login and password of the user, and launches a new backend instance for the user is the connection is accepted.

In this chapter, we will give more detailed explanation of how this application works, the reason for which it is needed, the different pitfalls when implementing it, and some possible improvements.

4.1 General explanations

This application is, as explained above, a server web application used to communicate with the web browser. However, the application really has two roles, which are very different from each other. We will here give a brief description of how these two roles are handled by the application.

A global overview of the execution of the program seen from the frontend application can be found in figure 4.1.

During the login phase, the application just acts as a normal web application. The client sends its credentials through an HTTP request, with some additional information, such as the size of the web browser window. The application checks the credentials of the client, and just sends back an error in case they did not match the UNIX/Linux credentials. If the credentials were correct, two actions happen simultaneously.

- The application launches a process of the backend application with the user id of the user connecting. When the backend application initializes, it sends a message to the frontend application to register itself, sending a unique identifier in order for both application to communicate together. This process usually ends up a little faster than the websocket connection and the first messages received from the backend application are queued until the websocket connection is opened. This choice have

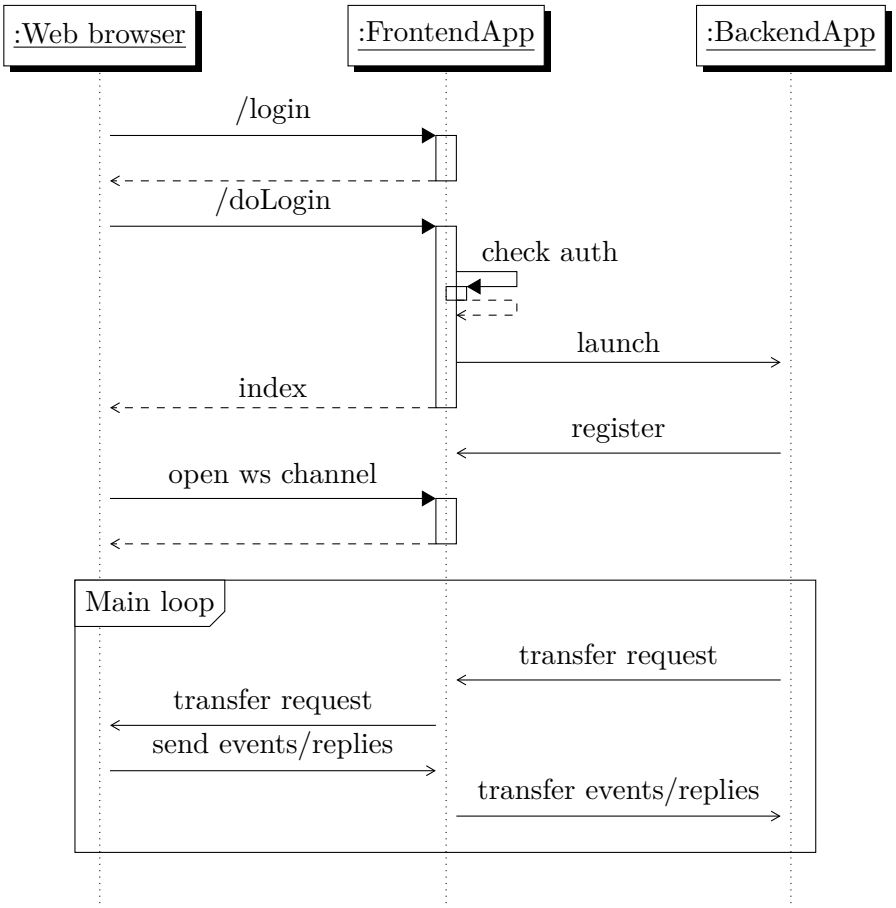

Figure 4.1: System initialization and execution

been made over waiting for the websocket to initialize, in order to improve the connection speed.

- It also sends the JS application for the web browser through the HTTP response with a secret token to authenticate the client. When the web browser receives this response, it initializes a websocket connection with the application, sending the secret token with it. The application checks for the authenticity of the token, and if the token is valid, the application accepts the websocket connection and starts transferring the messages that have been queued while waiting for the websocket connection to open.

After the having been initialized, the system enters in its *main loop* as shown in figure 4.1. The backend application transfers the requests received from X clients to the frontend application, which then transfer them to the web browser. When a reply or an event is generated by the browser, it is first sent to the frontend application which then send the message to the backend application for it to inform the X client accordingly.

4.2 Need for the frontend

Given the above explanations about the frontend application, one could think that the communication could be directly be done between the server backend and the web browser, which is partly true. We will therefore briefly discuss about the need for this application.

4.2.1 Security concerns

The most important reason of this application to be is basic security concerns. The main point being

- If the backend application and the web browser were communicating directly, as what user should the backend application be run?

which is a question with probably not any answer. If only a single user was to use the system, the application could be run as this user, which depending on the port the application is running on, could be done without having to change any privilege. However, this is no longer a valid approach when several users want to use the application at the same time. In this case, the application could be running as a special user created specially for that purpose, but in this case, for the user to be able to access their home, this user should be allowed to access it too, which clearly is not reasonable.

The best solution to solve this problem was therefore to create a kind of dispatcher application, which act as a frontend, and starts the backend application in a new process as the given user for each connection. Using this method, the only requirement for the frontend is to have a way to check the password authentication and to start a new process as this user. We will discuss about this in the next section.

4.2.2 Distributed system

Another motivation for adding a frontend is that by letting it act as a dispatcher of the backend application, adapting the system for it to run on several remote computers should be greatly simplified.

With the current system design, the frontend application and the backend application have no need to be running on the same machine, and the frontend could therefore act not only as a dispatcher to launch different process on the active machine, but also as a dispatcher to run the processes on different machines, on the same local area network, for example. As the communications between the frontend and backend are already done through TCP, this part of this application would work without any change, except for the hostname configuration.

However, in future versions, we might allow the backend application itself to listen for websocket connections after the frontend application has ended up the login process. This would allow the web application to connect directly to the backend application and could probably reduce the latency.

4.3 Login manager

As explained previously, one of the main role of the frontend application is to act as a login manager. That is, to check for the user credentials and to start a *session* for the user when these are correct.

A large difference with a normal web application is that the information used in this are completely volatile and only saved on memory, as the only needed ones in the frontend application are the user name and token to authenticate it. Furthermore, once the connection is closed, these information can be deleted safely and therefore do not need permanent storage.

When the user connects to the frontend application, the user name and password are sent through a normal HTTP request. The credentials are then checked using a small program written in C++[1], which simply returns an error code if the password was not correct or the user not found, or 0 when everything went fine. The calling application can therefore get sufficient information just by checking the exit code of the program.

The weakness of this approach is that when checking the password, the application needs to be ran as root, and for obvious security reason, it is better to avoid having an application with open ports running as root when possible.

When writing this paper, the feature has not been integrated in the application, and the frontend application needs to be ran root, however, we will soon be re-factoring the system to take the following approach.

1. Create an authentication server to check the credentials.
2. Set the authentication server to accept only connections from the frontend application.
3. Send a request from the frontend application to the authentication server to check for the credentials.
4. Launch or not the backend application or not depending on the answer from the authentication server.

[1] nix-password-checker is a very small program to check a password (on UNIX/Linux systems) by reading the `/etc/passwd` and the `/etc/shadow` files.

By using the above process, it is possible for the frontend application to be ran as a non-root user, and to have a total control on the authentication server running as a root.

Chapter 5

Server backend application

The server backend is the part of the system that communicates directly with the X clients, and takes care to transfer the necessary requests to the frontend application, as well as transferring back the replies and events sent from the frontend.

Another important point about this application is that it caches the information received from the X client and back from the frontend application, and using this cache tries to handle requests coming from the X client locally, without transferring them when not needed.

We will here discuss about how this application is designed and implemented.

5.1 General explanations

The application is launched asynchronously from the frontend application. Therefore, when starting up, the backend application first sends a message to the frontend to let it know that the application is now started and ready to receive and transfer messages. This is a single-way communication and the application does not wait for any answer from the frontend.

The second step when starting up the backend is to start listening on the port the X clients will be connecting to. The hostname to listen is statically set using a configuration file, however as the port to listen to changes depending on the current screen number, the port number to use to listen for incoming connection is sent as a command-line argument of the application. Typically, the first application to start will be given the screen number 0, and will therefore be listening on the port 6000 (see 2.4.1).

Finally, the application reads the initialization file of the current user to start launching applications (eg. a window manager or desktop environment). This file should be placed in the user home directory, and is called `.scalaxsrc`, however we plan to use the file `.xinitrc` to try to imitate the behavior of `startx` [1] or SLiM [2].

Once the application have finished initializing, it should received the first requests from X clients launched when reading the initialization file.

[1] `startx` is a script used to start the X Window System.
[2] SLiM (Simple Login Manager) is a graphical login manager for X.

When a request is received, the information given by the request, if any, is cached by the application, and the request is handled locally when possible, otherwise transferred to the frontend application. We will discuss about this further in the next section.

To handle requests sent by the X clients, the application first parses the requests and transforms it in a normal Scala object. It then checks if the requests needs to be handled locally, if the content of the requests needs to be cached, and finally if the request needs to be transferred to the frontend and sends it to the frontend through a TCP connection when needed.

5.2 Caching

As said in the previous section, an important responsibility of the backend application is to cache data received by both the X clients and the frontend application, and to use this cache efficiently to try to reduce the most possible the requests and replies transfers to the frontend application.

The caching system has not yet been implemented in the system when writing this paper, but the strategy to use is straightforward and should not be too difficult to implement.

The first step is to check if the requests content needs to be cached or not. As the requests are implemented as Scala case classes, we could easily add a trait to the requests that need to be cached. A possible example of this implementation is shown in listing 5.1.

```
trait NeedsCaching { // define trait for caching
  def cacheRequest(): Unit
}

// mixin traits to class needing to be cached
case class CreateWindowRequest (
  ...
) extends Request
  with NeedsTransfer // needs to be transfered to the frontend
  with NeedsCaching { // needs to be cached
  ...
  def cacheRequest() {
    // do caching logic
  }
}

// handle requests needing caching
request match {
  case r: NeedsCaching => r.cacheRequest()
  ...
}
```

Listing 5.1: Possible implementation of requests needing cache

Of course, the listing above is quite minimalist, but this should be enough to check if requests need caching and handle them.

The content of the `cacheRequest` method being too different for each request, we will not enter in the possible implementation details here, but at least for the different resources, the idea would be to save them as an appropriate model instance after checking that the resource id is valid.

Using this method, the errors of access to unavailable could be done in the backend application without any need to transfer the request further.

Chapter 6

Web browser application

The web browser application is the interface through which the user connects to the system and then controls it. It receives messages from the server frontend application and renders them on the browser screen. It also handles the browser's events and transfer them to frontend application.

6.1 Rendering

As mentioned above, the first role of the application is to render the requests received from the frontend application, and therefore indirectly from the X client running for the user's X session. Listing 6.1 shows a sample request that is received through the websocket channel and rendered by the application.

```
{
  "type":"request",
  "content": {
    "clientId": 1,
    "opCode": 70,
    "action": "PolyFillRectangleRequest",
    "request": {
      "drawable": 4194305,
      "context": 4194304,
      "rectangles": [{"x":20,"y":20,"width":200,"height":200}]
    }
  }
}
```

Listing 6.1: Sample request

The object contained in the `request` key is almost exactly the request sent by the X client to the backend application, translated in JSON, except from the op-code being stored in the wrapping object to make the request easier to handle. The rest of the message is extra meta information to help the browser handling requests efficiently.

The rendering implementation is based on KineticJS, and each window is treated as a different layer so that refreshing the display can be done without having to draw again the whole screen. This rendering strategy improves a lot performance as the area to redraw for each request is only the given part of the window and the rest of the display can stay intact.

6.2 Event handling

The next of this application is event handling, however, the implementation of this functionality not being done yet, we will here only give a general overview of the approach that will be taken to implement it.

The event that are to be handled by the X server are decided by the X client, and are different for each window.

The windows are created as KineticJS groups. The framework allows each node to have its own event logic, and groups being node, it is possible to assign all the events that JavaScript allow to a particular window. Therefore, when a window is set to react to some events, an event handler aware of the window resource id could be added, sending the event to the frontend application when triggered.

However, for events such as mouse motions, which trigger with a very high frequency, the events could be queued and sent when no event is triggered during an arbitrary period or when more than an arbitrary number of events have already been. This could help saving bandwidth and improving the general performances of the application.

Chapter 7

Discussion

In the current state of the system, real worlds example cannot be ran, and therefore a real evaluation of the application is difficult. We will therefore mainly discuss in a qualitative way to try to evaluate the system.

7.1 Performances

7.1.1 Used technologies

From a hardware level, the system performance should be good enough. We will here think the data transfer performances as well as the rendering performances.

Data transfer and bandwidth

Bandwidth has been increasing impressively during the last couple of years. About 10 years ago, 512kbits/s ADSL was quite common, and the downstream and upstream rate was very limited.

However, with the improvement of the communication protocols and the spread of fiber-optic communication, these rate has become higher and higher. Using fiber-optic communication, a server

Web browser rendering

Until HTML4, rendering something else than text in a web browser required the use of some other technologies, such as flash or yet Java applets.

From HTML5, Canvas has appeared in web browser and bitmap rendering can now be done natively. Furthermore, JavaScript implementations are becoming faster and faster, and recent JavaScript implementations are way faster than flash or Java applets[1].

In this system, the main purpose of the X server working is to be able to use a Linux distribution in a web browser, mainly for a normal desktop use and not to play 3D games in it. This should hardly require more than 10 frames per second, except maybe for videos. With the current performance of JavaScript and recent processors, such a frame rate should be easy enough to reach, and we therefore think that rendering performances should not be too much of a bottleneck for a main stream usage of the system.

7.1.2 System implementation

The system not yet at a very advanced implementation state, but there is however a point that needs to be fixed in order to improve the general performances.

In the actual implementation of the system, the server frontend application takes care of handling the client login, but also acts as a bridge between the backend application and the web browser. That means that all requests and replies pass through this application.

This is bad, and will therefore be fixed in the future releases, for two reasons.

- The requests and replies need to be transferred two times, and therefore have to be sent over three different channels to go from an X client to the web browser as shown in figure 3.1.
- If several users connect at the same time, the burden of the frontend application could become too big. This would slow down all the clients.

This could be greatly improved by having the web browser communicating directly with the backend application after the login phase.

For this improvement, we would simply need to have the frontend application sending the authentication token to the backend application after starting it. The web browser application would then been sent the hostname and eventually port to connect to. When connecting to the backend application which would check the received token and then finish setting up the connection.

7.2 Usability

As written in section 1.1, beginners often have a lot of trouble installing Linux. Actually, we think that this is not anymore a problem proper to Linux, but rather that the simple fact of installing an operating system on a machine is not as evident as it may appear for a non-experienced user.

With the system as it is, the user should be able to access the system without any need to install any new software, not even a virtualization software. The only software that the user may need to install could be a modern browser, which should be done much more easily that setting up a virtual environment. Furthermore, all the softwares needed by the user on the remote machine can be managed by an administrator and the user would therefore not have any need to take care of the server configuration by himself. This should make Linux environments much easier to access and use for backend users with no technical knowledge in computing.

Chapter 8

Related work

This project is not really inspired by or based on any existing project. However, there are some projects which have some similar functionalities and use cases. We will shortly present them and explain the main differences with this project.

8.1 RealVNC

RealVNC[1] is a remote access software that also allows clients to take control of a remote computer from their web browsers. There are however some major differences with this project.

RealVNC uses the Virtual Networking Computing (VNC) protocol to handle communication between the server and the client.

A big advantage of this method is that an existing implementation of the X server can be used, as the rendering can be done on the server side and the pixmap is passed through from the server to the client.

However, an inconvenient of this approach is that it can use a lot of bandwidth and though some optimization can be done, it remains way higher than a single JSON object used in this project.

Another difference with this project is that the user already needs to be logged in to access the remote computer and it can therefore not be used as a login manager, nor can be used efficiently for multiple users access.

8.2 WeirdX

WeirdX[2] is an X server implementation completely written in Java. It is basically quite different from this project as it is an X server and not a server/client system. However, by using Java Applets, it is possible to access a remote computer from a web browser.

That said, as the X server is completely running in the web browser and the clients are directly connecting to it, there are a lot of things that cannot be achieved. For example,

[1] `http://www.realvnc.com/`
[2] http://www.jcraft.com/weirdx/

a single instance will only be able to run for a single user, and it seems complicated to log users in and out through this system.

Chapter 9

Conclusion

Though a lot of work still needs to be done, we have managed to build a system to use a Linux/UNIX machine from a web browser using the X protocol. We managed to model and partly implement a system which could be self-sufficient for any graphical access to a machine running only X clients as graphical applications, which could remove a lot of dependencies and complexity in some cases.

From this experience, we could also show that new web technologies offer a lot of new possibilities, and we will try to continuing to prove it while continuing the project.

9.1 Discussion and future work

There are a lot of points that need further confirmation, and we will especially need to check performance in a real use-case, to see if the system could actually be used responding to real-world requirements. However, as discussed in section 7.1, we are expecting the performances to be enough for a daily life usage, and more than enough for educational purpose such as programming.

Some unsolved problems still remain, such as truetype fonts handling, or large data transfer. Truetype fonts could be rendered as drawings using Canvas [1] . Large data transfer could be done opening another websocket channel to avoid blocking the main communication channel especially when the connection is slow.

This project will continue to be developed, and we will try to reach a real-world usable implementation in the next six months.

[1] The Typeface project seems to be able to render truetype fonts using Canvas.

Appendix A

Actor model

In this chapter, we will give an overview of the actor model in general, and see how it can be used with Scala and the akka library.

A.1 Overview

First of all, we are going to briefly explain why and how can this model be useful, and then explain how it works.

A.1.1 Motivations

The first motivation for the actor model is too make multi-threading simpler and safer. To give an overview of the problem, in listing A.1 we are going to take a short example written in C++ using the pthread library for threading.

```
bool resource_manager::get_resource(int id, resource& r)
{
  pthread_mutex_lock(&this->_resource_lock);
  if(!this->has_resource(id)) {
    return false;
  }
  r = this->_resource[id];
  pthread_mutex_unlock(&this->_resource_lock);
  return true;
}

// in some other method
if(!manager->get_resource(wanted_id, wanted_resource)) {
  // if wanted id not available get fallback instead
  manager->get_resource(fallback_id, wanted_resource);
}
```

Listing A.1: Threading example in C++

In the above example, the application will deadlock as soon as the `wanted_id` is not available in the resource manager, as the lock is not released in that case.

Of course, in a simple example like above, any programmer having a little experience with threads would notice immediately that the lock is not released and it would probably would not take too long to debug anyway as the deadlock occurs each time the wanted id does not exist. However, everything is not always so simple, and as timing plays an important role when working with threads, deadlocks can be very difficult to debug.

An important motivation for using the actor model is to give an extra layer of abstraction to avoid this kind of problem.

A.1.2 Workflow with actors

When working with actors, the basic rule is that actor communicate with each others only using messages, and the actors' data should (almost) never be used and even less modified directly.

The main reason for this rule is that messages are treated sequentially by the actor, which is enough to avoid having to lock the data each time we try to use it, with the only condition being to respect the above rule.

We will give a short example of the above example in using the akka library in Scala.

```
class ResourceManager extends Actor {
  private var resources: Map[Int, Resource] = Map.empty
  def receive = {
    case ResourceRequest(resId) => resources.get(resId) match {
      case None => sender ! NotFound
      case Some(resource) => sender ! ResourceReply(resource)
    }
    ...
  }
  ...
}
```

Listing A.2: Threading using actors

Each time an actor receives a message, the `receive` method is executed, the common pattern is therefore to match the different messages that could possibly received and to handle it.

In the above example, when an other actor requests a resource, the `ResourceManager` actor checks if the resource exists, if it does it returns the resource to the sender wrapped in a `ResourceReply` object to help the receiver handling the response more easily, otherwise it returns a `NotFound` object.

The reason why no lock is needed here is, as explained above, that messages are treated sequentially and that actor resources are only accessed through messages, therefore two threads will never be accessing the same resource at the same time, which simplifies a lot synchronization.

A.2 Scala integration

The actor pattern integrates very well with Scala for different reasons that we will briefly discuss about.

A.2.1 Pattern matching and case classes

Scala allows very extensive pattern matching. As opposed to `switch` statements in a lot of imperative languages, in which only a few predefined types can be match, Scala allows, through `case class` and `case object` to match almost anything. A `case class` is an immutable class that have the necessary features to be matched. A `case object` is its singleton version.

This makes message handling when using actor very simple and efficient. To give a short example of it, let think of a chat client which can receive messages when a user connects, disconnects, or when a message is received. The case classes to use could be defined as in A.3 and the method handling could be implemented as in in A.4.

```
case class UserConnection(username: String)
case class UserDisconnection(username: String)
case class Message(username: String, message: String)
```

Listing A.3: Chat example definitions

```
class ChatActor extends Actor {
  private var users: Set[String] = Map.empty
  def receive = {
    case UserConnection(username) => {
      users += username
      chatView.putInfo(username + " has connected.")
    }
    case UserDisconnection(username) => {
      users -= username
      chatView.putInfo(username + " has disconnected.")
    }
    case Message(username: String, message: String) => {
      chatView.putMessage(username, message)
    }
  }
}
```

Listing A.4: Chat example implementation

This example if of course very minimalist but reflects quite well the simplicity of the model. Furthermore, the akka library integrates remote actors, which can be used to have the class handling a TCP connection. If the server sending the messages is using the same library and the same class definitions, the messages can be passed through TCP only by changing a configuration file.

Bibliography

[1] Jackson Dunstan. As3 vs. javascript performance. http://jacksondunstan.com/articles/2209.

[2] Martin Odersky. *Programming in Scala second edition.* artima, 2008.

[3] Kuo-Chung Tai Richard H. Carver. *Modern Multithreading: Implementing, Testing, and Debugging Multithreaded Java and C++/Pthreads/Win32 Programs.* John Wiley & Sons, 2005.

[4] Robert W.Scheifler. X window system protocol. http://www.x.org/docs/XProtocol/proto.pdf.