HELSINKI UNIVERSITY OF TECHNOLOGY
Department of Electrical and Communications Engineering
Laboratory of Acoustics and Audio Signal Processing

Eduardo García Barrachina

Interactive On-line Testing of Java and Audio Support on Web Browsers

Master's Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Technology.

Espoo, Nov 25, 2003

Supervisor:	Professor Matti Karjalainen
Instructors:	Martti Rahkila

HELSINKI UNIVERSITY OF TECHNOLOGY	ABSTRACT OF THE MASTER'S THESIS

Author:	Eduardo García Barrachina		
Name of the thesis:	Interactive On-line Testing of Java and Audio Support on Web Browsers		
Date:	Nov 25, 2003	**Number of pages:**	83
Department:	Electrical and Communications Engineering		
Professorship:	S-89		
Supervisor:	Prof. Matti Karjalainen		
Instructors:	Martti Rahkila M.Sc. (Tech.)		

This Master's Thesis deals with on-line testing of web browser capabilities. In particular, we are interested in determining whether the Java platform is present on a given host machine and if certain sound samples can be played through the use of Java applets. To achieve these goals a testing application has been developed.

The tests try to determine whether the browser accessing the application has any kind of Java support. If this is the case, some sound samples are played through the use of Java applets to find out the audio capabilities of the system. Help concerning the enabling of the Java platform and the correct audio (both hardware and software) settings is made available for the user.

The application tests for different Java graphical interfaces such as AWT and Swing and tries to play different audio formats such as *au*, *wav* or *aiff* with the use of Java applets.

The thesis evaluates the results obtained from the testing of the application done by the people from the Laboratory of Acoustics and Audio Signal Processing (Helsinki University of Technology). Conclusions concerning different Java platforms and browsers are drawn as well as ways to compile the applets for maximum compatibility.

The testing application has been made available for the Internet community at the following web site: http://www.acoustics.hut.fi/demos/javatest

Keywords: Java, Java applets, web applications, browsers, audio testing.

Acknowledgements

This Master's thesis has been written during the winter of 2002 and spring, summer and autumn of 2003. During this period I have been an attending the Helsinki University of Technology as an exchange student following a Socrates/Erasmus exchange programme between my home university (Technical University of Madrid - UPM) and the Helsinki University of Technology (HUT).

I want to thank my supervisor Professor Matti Karjalainen for giving me the opportunity to realize this project in Finland and specially my instructor Martti Rahkila for his thoughtful guidance throughout the whole thesis and for his constant advice not only in the academical aspects but also concerning my stay in Finland.

I would also like to thank Jesús Soler Ascaso, Ricardo Tormo Más, Pilar Esteban Pascual and Juan Gutiérrez Plaza for their help and support throughout the writing of the thesis.

My gratitude also goes to the whole *Erasmus community* which made me feel like being at home during my stay in Finland and my family for their constant support and caring.

Finally, I would like to thank Markku Hartikainen for making me take the final step towards my stay in Finland. Thank you all.

Otaniemi, November 25, 2003

Eduardo García Barrachina

Contents

Abbreviations

JDK	Java Development Kit
JRE	Java Runtime Environment
SDK	Software Development Kit
JMF	Java Media Framework
JFC	Java Foundation Classes
AWT	Abstract Windows Toolkit
API	Application Programmer Interface
GUI	Graphical User Interface
CGI	Common Gateway Interface
HTML	Hypertext Markup Language
DHTML	Dynamic Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
URL	Uniform Resource Locator
MSIE	Microsoft Internet Explorer
NS	Netscape
JVM	Java Virtual Machine
J2SE	Java 2 Standard Edition
IP	Internet Protocol
TCP	Transport Control Protocol
W3C	World Wide Web Consortium
RFC	Request for Comments
CSS	Cascading Style Sheets
DOM	Document Object Model
XML	Extensible Markup Language
AOL	America Online
OS	Operating System
MIDI	Musical Instruments Digital Interface
CD	Compact Disc

List of Figures

List of Tables

Chapter 1

Introduction

This project deals with on-line testing of web browser capabilities. There is a multitude of systems accessing the Internet nowadays. Different configurations and different settings on each of the systems make each computing device connected to the Internet a new challenge for on-line testing and development of web services. This project focuses on the testing of web browsers and their capability to display Java applets and reproduce sound samples made available through them.

1.1 The World Wide Web

The Internet

The Internet is a network of networks. It is a world wide computer network that interconnects millions of computing devices throughout the world. This collection of networks is basically formed by servers, routers and host machines. The servers have the information, the hosts access that information and the routers lead the data from one machine to the other. All the computing devices attached to the Internet must know how to communicate with each other. They must *speak* the same language, they must follow certain rules or protocols. The IP and TCP protocols (namely TCP/IP [Com00a]) form the technical basis that allows all the information on the Internet to flow from one computing device to another.

From a service oriented point of view, the Internet is a network that allows distributed applications running on different systems to exchange data with each other. Electronic mail and the world wide web (WWW) are, by far, the most popular applications used nowadays on the Internet.

Client-Server Model

Figure 1.1 shows a diagram with client and server machines connected to the Internet. This is the basic scenario where the WWW application runs. The WWW follows a client-server model, that is, a client or host machine requests some data from the server and the server sends the data back to the host. The data requested is generally a web page that is written in a special language that an application running on the host machine can display. The application that requests the page is a web browser and is specially designed to display HTML documents. The HyperText Markup Language (HTML) is the language with which the web pages on the Internet are written.

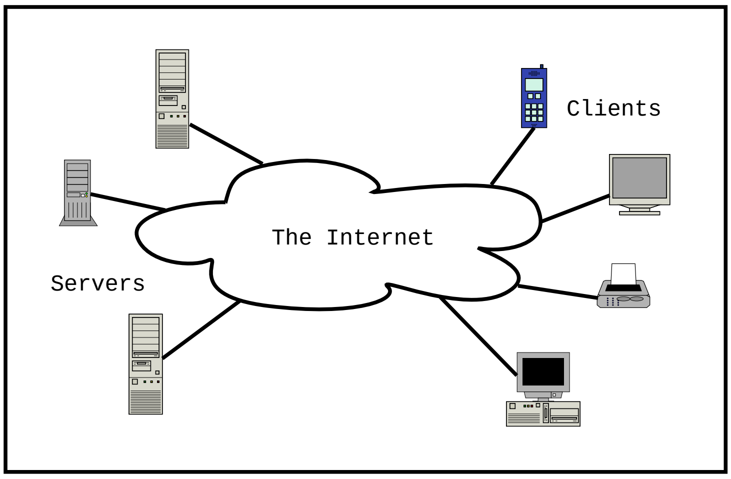

Figure 1.1: World Wide Web: Client-Server model

Although all the machines connected to the Internet have to provide the TCP/IP protocol stack in order to control the sending and receiving of data, the distributed applications running on a higher level of abstraction[1] also need a way of communicating with each other. Therefore, they also have a specific protocol that every application must follow in order to implement the service. For web applications the protocol used is the Hypertext Transfer Protocol (HTTP) [BLFF96] [FGM^{+}99].

Interactivity

At first the Internet was full of static web pages with no graphics that made the world wide web application simple but suitable for the needs of the early stages. With time, the Web

[1]TCP/IP is in charge of the transportation of the data to its correct destination regardless of its content. Higher level protocols are concerned about the actual data sent.

evolved and graphics and sounds were included, and dynamic web sites and applications created.

The HTML specification is not powerful enough to support dynamic web pages and applications and therefore new technologies have been demanded. In order to create these dynamic web sites, it is necessary to combine different programming technologies. We can combine the interactiveness that the dynamic web pages offer on the client side with active pages on the server side. Server side technologies do not depend on the browser being used and can make common tasks, such as accessing databases, simple for the developer to use. Client side and server side technologies are combined to create professional web applications nowadays [BAAG99].

On the server side, technologies such as CGI (Common Gateway Interface), PHP (Hypertext Preprocessor), ASP (Active Server Pages) or JSP (Java Server Pages) allow the developer to create active web pages. In some cases, most of the interaction is achieved through HTML forms which are read by the *server side part* of the application, generating the corresponding response based on the examined form.

Server side technologies do not depend on the software installed on the client machine. The reason why web developers would like to keep all their code running on the server side is because they have much more control over how their applications are running. None of their code is being executed elsewhere but on the server. Sometimes this is simply not possible and client side technologies have to be used to perform certain tasks that cannot be done on the server side. Whenever client side technologies are used, there is no way of knowing if the host machine will have the required software installed to perform the desired task or not.

When executing client side technologies, the computational load is on the host machine. This frees the server from executing heavy applications but adds some delay on the client. With technologies such as JavaScript, the script instructions have to be interpreted in real-time causing certain delay, and with Java applets, the loading of the runtime environment will also slow down the loading of the site. These two are the client side technologies which have been used in this work, but others such as DHTML (Dynamic HTML), ActiveX or Flash also allow dynamic content to be ran on the client side.

Client side technologies have security restrictions. Since they are running on the client machine, it is necessary to have some security measures to stop programs written with these technologies from accessing files or resources which they are not meant to. As an example, Java applets are not allowed to read or write files from the client file system and are also forbidden to make network connections to any machine other than the one that sent the applet in the first place [Micc]. The level of security can, nevertheless, be altered in some cases.

The drawback to client side technologies is that there is a need for extra software to be present on the host machine beforehand. This additional software is in charge of interpreting the code written for it; if the software is not there the code cannot be run. Browsers detect this code and act consequently.

Generally browsers launch an external viewer which is capable of interpreting the client side technology instructions. This external viewer is normally available through the use of a *plug-in*. *Plug-ins* are small programs that can understand certain types of code or files. They give some extra functionality to the browser that it didn't originally have.

1.2 JavaScript

JavaScript has the basic characteristics of an object oriented language but does not have all the complex structures that object oriented languages have. [2] It is an interpreted language and its scripts are inserted among the HTML code of the web pages.

JavaScript is a client side technology which allows the user to interact with HTML pages. Being a client side technology, the reactions to the user's actions can be performed without sending any additional data to the server and thus providing dynamic interaction. It also allows the browser to react to certain events that the user may perform such as mouse clicks or form submission checks. Scripts written in this language depend upon the browsers capability to interpret them.

1.3 Java

Java is a full featured, object-oriented programming language developed by Sun Microsystems. Among the different types of programs that can be written in Java, in this work we are mostly interested in Java applets. These programs are small applications that are shipped with the web page requested by the browser and run on the client's machine. To do so, they require a Java interpreter called a Java Virtual Machine (JVM). This JVM is specific to the

[2]It cannot, for example, defi ne new classes nor has the inheritance property.

operating system running it, and therefore, there is a JVM coded for each platform. This guarantees that one same applet can run on any computer regardless of the operating system given that the adequate virtual machine is installed.

As shown on figure 1.2, when a Java applet reaches the browser, the latter will try to load the JVM so that it can play the applet. Some browsers have the JVM embedded into them, but others don't. Most of them now use *plug-ins* that allow the browser to play Java applets. Even though the browser has a JVM embedded into it, if a plug-in is installed, the applet may be played by the *plug-in* depending on a series of parameters.

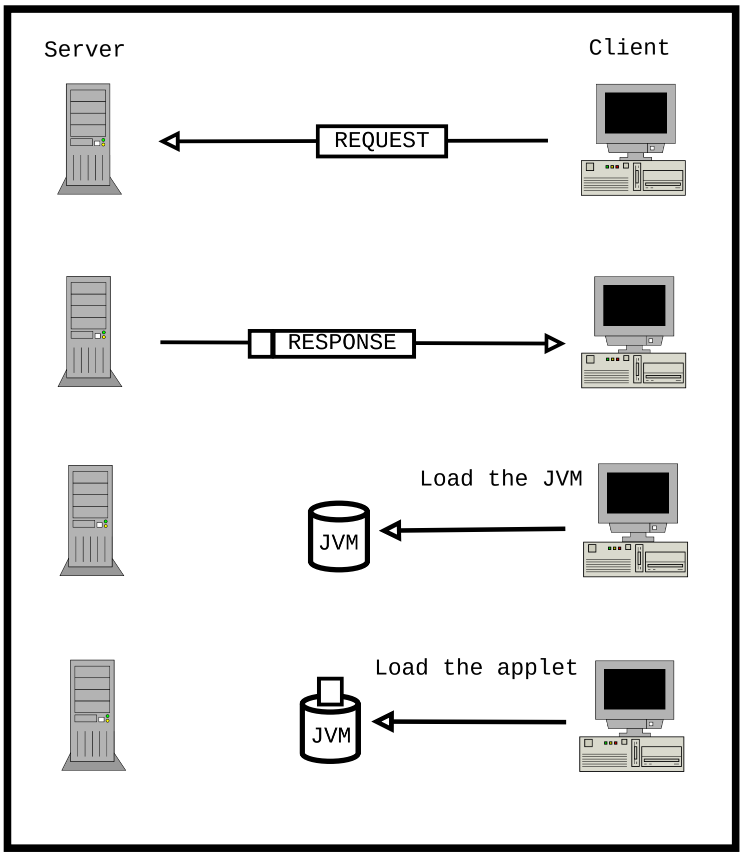

Figure 1.2: Loading of Java applets

The original JVM was developed by Sun Microsystems, but other companies have also developed their own JVM. Even though all the JVM should be the same in terms of functionality and applets should behave in the same manner regardless of JVM, experience shows that this is not so. Depending on the browser your system uses, you may have a specific JVM embedded into it. Nevertheless, in most of the cases, you can download another JVM and configure the browser to use that one instead.

1.4 Browsers

There are many types of browsers available on the Web nowadays. Web browsers present the information requested on the systems' display. Different companies present their own browsers and release new versions regularly to support the latests features. There are newer and older versions of all the browsers being used on the Internet nowadays, and they all behave in slightly different manners. Furthermore, there are browsers specifically written for the different platforms available and every browser can be configured to behave in a different way. User settings and personalization are available for every browser and this makes it even harder to determine their capabilities at first sight since key features can be easily disabled by the user. This results in a great variety of systems accessing the Web.

1.5 The Testing Application

Given these three axes (browsers, platforms and JVMs), the number of different kind of systems and different versions that may approach a web based application is considerably large. The tweaking of the browsers' user preferences make the testing even more necessary since same browsers, on a same platform and using the same JVM may behave differently because of different user preferences.

In this project we have developed a tool that will determine the audio capabilities of a client browser through Java applets, regardless of the platform, JVM or browser type it is executed on. To do so, we perform a series of simple tests focused on the Java support given by the browser. With technologies like Java, it is simple to play different kind of audio samples on the client machine using applets, and test for sound availability. The testing starts looking for some kind of Java support on the browser and then moves on to the audio testing. Results are kept on a file that keeps track of the user's replies concerning the sounds produced by his system.

The audio testing is achieved by playing different sound samples which vary in frequency, sampling rate, coding techniques, amplitude and number of channels. The aim is to determine the audio limits on the client machine.

Should the sound problems not arise through the Java support, but through hardware or other type of software settings, the application also includes basic tutorial help that will guide the user into the correct hardware and software settings and will perform a series of tests to find out the reason why sound is not being produced if this was to be the case.

The application may need to be run several times in order to achieve its goals. A browser upgrade or the downloading and installation of a brand new Java platform, may force the user to run the application more than once.

Finally, a human readable file with the results is generated and given to the user. The server keeps track of this information plus other debugging logs such as the path followed by the user, user agent of the browser, or the times of access to the application. No personal identifying information (including IP addresses) is kept.

1.6 Motivation of the Testing Application

Web application developers and designers need their software to work correctly on every system. If some part of their application were to be run on the client side, they would have to rely on the client machine to display their work in a correct manner. Since this is not reliable, most of the application will be kept on the server side where the developer is sure it will run smoothly. Unfortunately it may not be possible to keep the whole application on the server side and, therefore, some client side programming will have to be done. In order to make sure that the client will display the information correctly (or even display it at all), we need some kind of performance proof that will assure that the accessing machine can handle the application in the way it is meant to be.

Due to the fact that many possible different systems may be accessing the web and running the same application, it is very hard to ensure that it will work on all of them. Besides, if the application produces a crash of some kind or hangs the system up, that user or potential customer, will never access the application again. This risk must be diminished to the largest possible extent and thus the testing becomes essential.

If a developer (or a company) can get to know how a client machine is going to behave when it faces Java applets, it may be decided to code an application using applets instead of using other technologies that would ensure stability but maybe not the interactiveness the developer wanted to achieve in the first place. The web developer will not approach the user's system in a blind way, but will have useful information beforehand.

This on-line testing application will give the web developer proof of the capabilities of the machine accessing the service. Further more, the web developer may choose to send different kind of information to different systems in order to make the most of the enhanced features a client machine may happen to have. In order to do so, there is a need to know what the remote computing device will support and what it will not.

1.7 Goals of the Application

The main goal of this project is to create an on-line application that will test the audio capabilities in a system through the use of Java applets and at the same time determine the Java capabilities of the browser. This application will be made available from the servers in the Laboratory of Acoustics and Audio Signal Processing belonging to the Helsinki University of Technology in Finland for public use.

After the testing has been run, we should be able to know the systems' audio properties and how it will respond to applications that use audio through Java applets. The client machine should also be able to reproduce sounds and if sounds weren't to be available, know the reasons and give suggestions on how to overcome the problem.

The application will try to serve as a filter that will determine whether a client machine is capable of reproducing sound samples (and to what extent) or not.

The user will be informed about his/her system in a human readable way. It is intended that the user learns about his/her system and understands all the steps that were taken throughout the testing. This will result in a better understanding of the system and some basic knowledge on browser settings and audio hardware and software configuration on the user part.

Experience shows that the user does not really know what software or hardware his computer system has installed and if he/she does, he/she may not know what it is able to do. This application will try to serve as an educational guide to basic hardware and software audio setting principles and also to give some useful information to the user about the computer system being tested.

All the testing being done is focused under an educational scope. In no way does this application try to spy on the client machine nor try to gain the information for purposes other than the here stated. The user is always aware of the tests that are being performed on his machine and *why* they are being performed. The user is always given the option to end the testing explicitly and the information gathered up to that point is then shown.

Chapter 2

World Wide Web Browsers

2.1 Different Browsers

A web Browser is a computer program that allows you to view web pages. It can interpret the HTML language and display it on the screen in a way it considers to be correct. Web browsers let you *browse* the Web. A definition for a *browser* can be found in www.webopedia.com [web02a]:

Short for web browser, a software application used to locate and display web pages. The two most popular browsers are Netscape Navigator and Microsoft Internet Explorer. Both of these are graphical browsers, which means that they can display graphics as well as text. In addition, most modern browsers can present multimedia information, including sound and video, though they require plug-ins for some formats.

Variety

There are many different browsers *browsing* the Web nowadays. Some of them, such as Apple's Safari are specific to only one platform, but most of them (Opera, Netscape, Mozilla) are cross-platform meaning they have a corresponding version for different operating systems. Not only is there a large amount of different browsers for different platforms available, but also different versions of them are browsing the Web. Besides, new browsers are being released all the time and becoming a part of the browser market. Web browsers may differ slightly from one release to another or may be totally different. All these browsers are potential visitors of our application, and it should work for all of them.

Mozilla

Among the different browsers that may access the testing application it is important to note Mozilla based browsers. Mozilla is an open-source web browser which is developed by an organization named mozilla.org [Moza] When Netscape decided to make the source code for its browser freely available to the public, the group responsible for releasing the code became mozilla.org and since then, they have been working on different open-source projects (the most important one being the Mozilla browser). Since Mozilla's code is open-source there are many web browsers today based on the Mozilla engine (Netscape is the most well known one, but there are plenty of others as well). Many of the browsers accessing the application will identify themselves as being a Mozilla browser although they may not be so (explanation follows), and keeping track of them may be a difficult task.

Robots

We accessed some server logs to have an idea of the different browsers that were to be expected for the application. The logs from the Laboratory of Acoustics and Audio Signal Processing (Helsinki University of Technology) showed that only a small collection of browsers was being used to access the server. Being a server located within a University domain, we thought that the list may not be representative of the whole web community and therefore searched for other sources.

Nevertheless, it was interesting to see the huge amount of different web robots or spiders that accessed the server. These web robots are launched by the most popular search engines to find new sites for their growing databases. Some of them even presented more requests than other not so popular browsers such as Opera or Mozilla.

These robots were undoubtedly going to access the application. The measures taken to stop them from surfing the pages was to introduce some *META* tags [W3C99] at the beginning of the template documents in the hope that they would stop there. Besides, at some point in the application you cannot continue unless you are redirected to another page by a Java applet. There is no way robots can understand Java applets, and would therefore stop in the early stages of the application.

2.2 Browser Detection

2.2.1 History: The "Browser War"

The first non-text based browser that was developed, was released at the University of Illinois in 1993. It was called Mosaic [fSA97]. The leader of the "Mosaic" project later founded Mosaic Communications which soon became Netscape Communications. In October 12th 1994 the first version of Netscape Navigator was released (Beta 1 version) and two months later the final release was made available for download [qin]. During those two months, Netscape's browser market share rose up to 70 per cent [Nie95].

In august 1995 Microsoft launched Internet Explorer to challenge Netscape's dominant position in web browsing software. Three months later, the Internet Engineering Task Force's (IETF's) HTML Working Group set the standard for core HTML features by developing HTML 2.0. [BLC95] but this did not work. The attempt made by the IETF to standardize website design was in vain. HTML browser manufactures like Netscape and Microsoft chose which parts of the standard they would follow and which ones they would ignore. They even made their own parts which, in some cases, eventually became parts of the standards.

The battle between the two mayor browser vendors had begun. Each of them added new features to the HTML code their browsers supported in order to provide the richest content to their users and to invite web developers to support those new features on their web pages. This led to a time battle where both competitors would rush their releases in order to keep their positioning on the market.

Developers were forced, from the beginning, to write their code bearing in mind the variety of browsers, some of which supported the latest HTML version and some which did not. Web designers had to either use the lowest common denominator of HTML when creating their pages (they had to ensure the document displayed correctly on every browser) or use server side browser detection techniques in order to send different information to the user depending on their browser.

The rushing in the development of the new releases caused standards to be incompletely implemented on the browsers. Each one supported the standards in a different way and therefore behaved differently when facing a web site. Browser detection can be crucial for a web service if the content on the server is browser specific, since a wrong detected browser may get the wrong information if it gets any information at all[1].

The "browser war" was finally declared over by the end of 1999 when Microsoft had around 2/3 of the market share [Leg99]. Nowadays its share has risen over 90% [One03] with Explorer 6.0 being the browser that is most popular on the Internet. Companies that offer these figures base their results on the parsing of the user-agent string, and as we found out from our own experience, and will explain in the following section, browsers can be disguised.

2.2.2 User Agent Strings

Evolution

Browser detection is one of the most common problems found in web developing. Developers have figured out different techniques and scripts for detecting certain types of browsers but none of them seem to be definitive. Since at first, different browsers presented specific features that tried to catch developers and user's attention, it was simple to test for something unique which defined a particular browser. This seemed to work smoothly when only two browsers were on the scenario, but as soon as new web browsers appeared this identifying scheme did no longer work.

The parsing of the user-agent string is what is now used to detect the browser. Browsers send more information to the server than just the page they are requesting. They identify themselves by sending a user-agent string along with the request. The user-agent string sent by a Mozilla browser running on a Linux machine can be seen in table 2.1

Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.0.2) Gecko/20030708

Table 2.1: Example of the user-agent string sent by a Mozilla browser

RFC 1945 [BLFF96] on HTTP/1.0 states that:

The User-Agent request-header field contains information about the user agent originating the request.

Just by parsing the user agent string we should be able to determine the type of browser. Unfortunately, there is no standard format for the user agent string. According to RFC

[1]Many of the problems reported in the press regarding Netscape's Gecko inability to display content were directly related to inadequate browser detection strategies [Cla02]

1945 [BLFF96]:

Although it is not required, user agents should include this field with requests. The field can contain multiple product tokens (Section 3.7) and comments identifying the agent and any subproducts which form a significant part of the user agent. By convention, the product tokens are listed in order of their significance for identifying the application.

During the "browser war", detecting the browser that was accessing a web site correctly was a key problem. Browsers started using the Mozilla token in their user agent strings in order to access Netscape-only web sites even if they were not Mozilla based browsers. This has been left unchanged, and nowadays it is difficult to find a user agent string which does not start with the Mozilla token.

User agent strings can be easily modified. Advanced users of the web can create their own web browsers and make them send whatever information to the server they want. Some browsers, such as Konqueror, allow the user to choose from some standard user agent strings available, and some Opera versions will default to Internet Explorer in the string if the site is not configured to identify Opera. This is why statistics have to be taken with precaution. There is no way Explorer's dominance can be argued on, but the low percentage of other browsers on the market share seems to be too low.

Concerning our application, new browsers will appear in the future and will have unpredictable user agent strings. The application is ready to handle the most common tokens in the user string concerning the browsers that are available today. However, future browsers will send new tokens that the application does not recognize, and therefore it will detect them as *Unknown* browsers.

Motivation

When we decided to detect the browser that was accessing the application, two main objectives were being followed. The first one was to filter some browsers that were known not to have Java support at all. If the browser belonged to this list the user would be informed and asked to use another browser instead. Browsers that do not support Java are text based browsers and browsers which were developed prior to the existence of the Java technology. Not many of these old browsers are still accessing the Web, but still we decided to look for them before taking the audio tests.

The second goal was to see if a link could be established between a particular browser and the Java capabilities it presented. Naturally, this turned out to be unsuccessful. There is no way of knowing if the applets will load just by parsing the user agent string, so no relation could be established. What really determines the Java capabilities, is the version and vendor of the virtual machine that is present on the system, and even this is not accurate in all the cases as the results will show.

Procedure

A large collection of user agent strings were obtained from the Web [cyS]. The idea was to check the user agent string from the browser accessing the application against the list of non-Java-supporting browsers. If it was found to belong to this list the user would be prompted to use another browser and else wise the test would be taken. It was found out that the algorithm was suitable but the initial condition had to be changed. One same browser has different user agent strings depending on the language it is in, meaning we would have had to check for all the language possibilities concerning one unique string.

Instead, what was done was to group the non compatible Java browsers and study their user agent strings in the search of a common pattern that would distinguish them from the valid ones. These were to be the initial conditions to the algorithm. Table 2.2 shows the browsers that the application detects as non valid Java browsers. The tokens' column shows the keywords that the application searches for to decide if the browser is not valid. If any of these tokens are found, the user is asked to use another browser.

Gathering this information was not an easy task since most of the browsers now *do* support Java and are able to load Java applets. The earlier versions of the browsers did not support Java, but no, or little, information about them can still be found on the Web. Besides, they are no longer available for downloading, so we cannot perform our tests on them and have to rely on the information given by the browsers' companies or by other Internet users.

The browsers which were thoroughly tested during the development of the application were Mozilla 1.0.1 and 1.0.2, Netscape 4.79 and 4.80, Opera 6.1, Galeon 1.2.6 and Konqueror 3.0.5a-0.73.2. All of them running on a Linux machine.

Browser	Version	Platform	Tokens
Internet Explorer	2.0 (or below)	Windows Macintosh PowerPC Macintosh 68000	MSIE 2. Microsoft Internet Explorer
Netscape Navigator	3.0 (or below)	Windows 3.x	Mozilla/3. & Win16
Netscape Navigator	2.0 (or below)	Windows Linux	Mozilla/2. & 16bit Mozilla/1.
Netscape Navigator	2.0 (or below)	Macintosh PowerPC Macintosh 68000	Mozilla/2. & Mac
Opera	3.x (or below)	Windows	Opera/3.
Others			lynx libwww aolbrowser1 WebTV1.0

Table 2.2: Common tokens in the user-agent strings of non-Java supported web browsers.

2.3 The Future of Web Browsers

2.3.1 The Browser Market

Internet Explorer

As figure 2.1 shows, the market is completely dominated by Microsoft's Internet Explorer. OneStat.com, as of July 2003 situates Microsoft's Internet Explorer with 93.5% of the market share. All the other browsers share 6.5% of the market pie (web robots are not included).

Explorer is available for Windows platforms and Apple environments. However, on June 2003, Microsoft officially dropped the development of the Explorer browser for the Apple Macintosh platform [Dal03]. Microsoft is stopping the development of Internet Explorer in part because of Apple's new web browser, Safari. Apple has an unlimited access to the operating system that no other developer has and can therefore tweak its browser to achieve best performance on Apple's machines[2] [Dal03]. This makes Explorer a platform specific browser again.

Netscape

America Online, the world's largest Internet service provider [JC98] bought Netscape Communications in 1998. Netscape was losing its relevant position in the browser market share to Microsoft's Internet Explorer and this acquisition was seen as an effort to make the

[2]According to OneStat.com [One03] the Safari market share has risen 0.4% from February to July 2003.

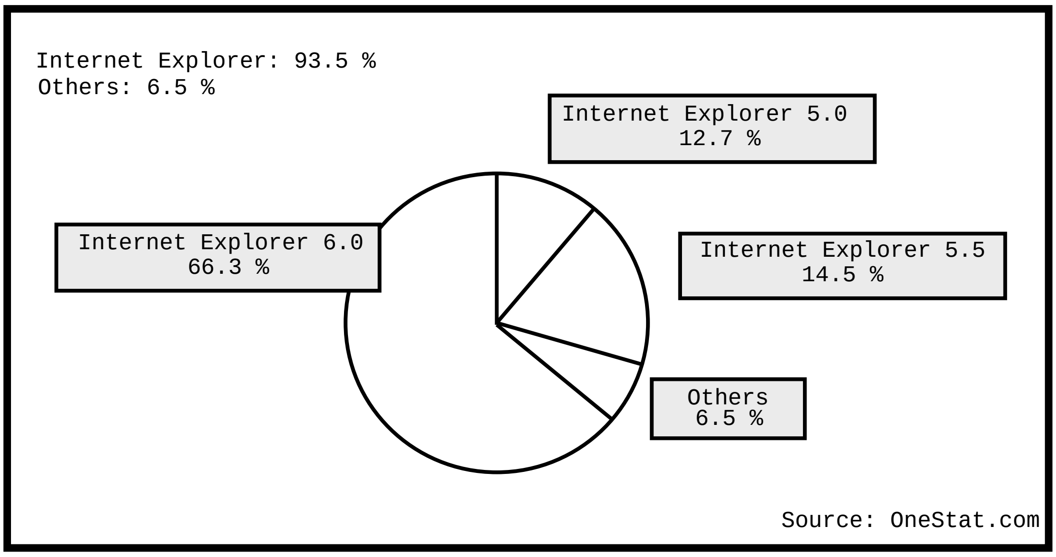

Figure 2.1: Browser market share (July 2003)

Netscape browser competitive again. But Netscape was unable to fight back.

Although AOL has not officially announced that they have stopped supporting the development of the Netscape browser, the company seems not to be putting any more efforts into it [Lan03]: it has announced economical help for the Mozilla Foundation and has signed an agreement with Microsoft to exclusively offer Internet Explorer to their clients for the next seven years. AOL has also reduced Netscape staff by 10% and analysts agree that version 7 of Netscape could be the last one that is ever released [Bec03].

On July 2003 the Mozilla site published the following news [Org03]:

RIP Netscape

On Tuesday, AOL Time Warner closed the Netscape browser division and laid off or reassigned most of the development team. AOL has agreed to donate Mozilla-related trademarks, the mozilla.org domain name and equipment (such as the mozilla.org servers) to the Mozilla Foundation. Some staff will be kept on for a couple of months to help with the transition.

While some Netscape developers will continue to contribute to Mozilla as volunteers, the team as a whole will be missed. Netscape made commercial browser development a *reality and later showed the world that a major proprietary product could be successfully turned into an open-source project. Messages from ex-Netscape employees can be found at ex-mozilla.org.*

As stated before, AOL has not officially announced they are stopping the development of new Netscape versions.

Mozilla

Mozilla was not originally intended to be an end-user product; it was meant to be a technology product. As open source code, it would be available to anyone who wanted to use the technology. Mozilla never advertised the product nor marketed it in any way; AOL's Netscape was its major client. With the creation of the Mozilla Foundation, Mozilla now intends to enter the browser market, competing with Internet Explorer, Opera, and others [Org03].

OneStat.com states that there has been a rise in the usage of Mozilla browsers of 0.4% from Frebuary to July 2003

2.3.2 Standards

Another important aspect of the future of browsers are standards. Designing and building following the standards make web sites accessible to more people and more types of Internet devices. Furthermore, sites developed using the standards will continue to function correctly as browsers evolve and new Internet devices come to the market.

During the fight for browser market share between Netscape and Microsoft, the HTML standard was not followed by either of the vendors. Developers had to decide which browser to code their pages for if they could not afford to have different versions of every web page on their site. These browser-specific sites do not longer work on the new browsers, although they did on the old ones. Many of these non compliant browsers are still used for surfing the Net.

In response to these problems, the Web Standards Project (WaSP) [WaS] was formed in 1998. Their goal was to encourage the companies that developed the browsers to follow the web standards. Starting from 2000 the major manufacturers of web browsers promised to follow many of the standards that were promoted by WaSP.

It is beneficial for the Internet to use standard compliant browsers since it is the only way to display web pages as they were designed to be seen. The latest standard on web publishing is XHTML 1.0 specification issued as a W3C Recommendation by the 26 January 2000 (revised 1 August 2002)[W3C02], although other recommendations such as CSS, ECMAScript and DOM are also relevant when developing web sites or web applications.

Netscape claims that it follows the standards as completely as no other browser does[3] since it was designed from scratch to be compliant with the W3C HTML, W3C CSS, W3C XML, W3C DOM and the ECMAScript standards. [Dev].

The web pages of the application are all HTML 4.01 Transitional compliant, so browsers that follow the standards should be able to display the information correctly.

[3]Still it does support other features outside the standards that are still commonly used on the Internet.

Chapter 3

Java

Java is an object oriented programming language created by Sun Microsystems. One of the main characteristics Java presents is that once it has been compiled and the *human readable* code has been transformed into bytecodes by the appropriate compiler, the resulting code is platform independent. In order to interpret the bytecodes a virtual machine is needed. This software interprets the code and displays the information as it understands it. The virtual machine has to be platform dependent.

Several kinds of Java programs can be developed: applications, applets, servlets or JavaBeans are some of the possible programs that can be created with Java. All of them are used for different purposes and all of them have different characteristics. In this work we are most interested in Java applets.

Unlike applications which are regular stand-alone programs which are executed from the command line using the *java* command, applets are written to be downloaded from the Web and executed on a web browser. Applets are small applications which the browser downloads when it reaches certain HTML code present on the web page. The browser then loads the JVM and executes the applet.

Applications have less security restrictions than applets. Applications run under a *controlled* environment whereas applets can be downloaded from an untrusted site and therefore need to be controlled in some way. As an example, applications have access to the file system of the machine they are being executed on whereas applets do not. [Micd][1]

Servlets are Java programs which are executed on the server side and have similarly

[1]A special type of Java applets called *signed applets* (introduced in JDK 1.1) have less security restrictions

functions to other programs coded using other server side technologies [Micf]. JavaBean components are reusable software components written in Java that can be used to create your own applications [Mich]. None of these two Java programs are of interest in this work.

3.1 Java Releases

There have been many Java versions since the first Java Development Kit release on January 1996. Sun's JDK offered the tools for developing applets and applications: it had the core packages, a Java compiler, a debugger, some other related tools and a java launcher which could execute the Java code.

Other software that Sun released were Java's Runtime Environments (JRE). The Java Runtime Environment provides the libraries, Java virtual machine, and other components necessary for users to run applets and applications written in the Java programming language. It does not contain tools and utilities such as compilers or debuggers for developing applets and applications.

At first Sun distributed their Development Kits separately from their Runtime Environments, but starting from Java 2 Standard Edition, they were both bundled together in a unique package. Developers need to download the whole package since they need the complier, debuggers and a virtual machine where to test their applications, but end users need only to have the Runtime Environment. End users are just interested in running applets and applications developed by others they're not interested in developing any applications themselves. Besides, the JRE is smaller and therefore easier to download.

Java Development Kits ranging from 1.0.2 (this was the first Java release that was available for downloading) to 1.1.x keep their names as so, but starting from JDK 1.2, all the Java releases were called Java 2 and had different version numbers depending on the Development Kit. At the time this thesis was being written, Sun's latest Java release was version 1.4.1 (*Hopper*), available since September 2002, but just before the ending, its follow on update, 1.4.2 (*Mantis*), was released in June 2003. On Sun's site you can even find information about Java 2 Platform, Standard Edition 1.5 namely called *Tiger* which is will be available as a beta version by the end of 2003 as was stated at the at the 2003 JavaOne Conference. [Aus03]

3.1.1 Java Plug-ins

A Java plug-in is a product created by Sun that is able to run Java applets using Sun's virtual machine. On Sun's developers site we can find this definition of a plug-in [Mic02b]:
Java Plug-in extends the functionality of a web browser, allowing applets or Java Beans to be run under Sun's Java 2 runtime environment (JRE) rather than the Java runtime environment that comes with the web browser. Java Plug-in is part of Sun's JRE and is installed with it when the JRE is installed on a computer. It works with both Netscape and Internet Explorer.

Some browsers have their own Java virtual machine. Some have the possibility to use another virtual machine through the use of a plug-in to play the Java applets, others have the Java code embedded into the browser. Non-Sun Java virtual machines are not totally compatible with Sun's and due to old agreements, have not been updated since JDK 1.1 [Micg]. This was a great problem in the developing of our application since we used applets throughout it and we had to make sure they worked correctly regardless of the virtual machine be it new or old.

Important dates in the life of the Java platform are shown on the time line in table 3.1. [Mic00] [Har02a] [Har02b] [

Every new release of the Java platform included new APIs and fixed old bugs. It's evolution has made Java a powerful tool.

3.2 Graphics in the Java Platform

Two major graphical user interfaces (GUI) have been developed for the Java platform: AWT and Swing. We test for both of them in our application.

3.2.1 Abstract Windows Toolkit (AWT)

The Abstract Window Toolkit (AWT) is made up of a large collection of classes which are used to build graphical user interfaces in Java. The java.awt package contains the basic Application Program Interface (API)[2] that allow Java applications to interact with the window

[2]Webmonkey's [Web03] API definition: *Abbreviation of application program interface, a set of routines, protocols, and tools for building software applications. A good API makes it easier to develop a program by providing all the building blocks. A programmer puts the blocks together.*

Date	Event
May 23, 1995	Java technology launched
January 23, 1996	JDK^{TM} 1.0.2 software release day
December 9, 1996	JDK 1.1 beta software released
February 18, 1997	JDK 1.1 release ships
April 2, 1997	Java Foundation Classes technology to be included in next revision of Java platform
August 29, 1997	Java^{TM} Media Framework Specifi cation
March, 1998	JFC/"Project Swing" ships
April 20, 1998	Java Plug-in ^{TM} product ships
December 8, 1998	Java 2 platform ships; flexible licensing terms announced
August 25, 1999	J2SE version 1.3 beta software released
September 30, 1999	J2EE beta software released
December 8, 1999	J2EE platform ships
December 8, 1999	J2SE platform on Linux ships
May 8, 2000	J2SE v. 1.3 platform released
May 17, 2000	J2SE v 1.3 platform gains industry support from Apple with Mac OS X
February 13, 2002	J2SE v 1.4.0 Standard Edition released.
September 7, 2002	J2SE v 1.4.1 Standard Edition released.
March 10, 2003	Apple updates the Java version to 1.4.1 for the Mac OS X
June 27, 2003	J2SE v 1.4.2 Standard Edition released.

Table 3.1: Milestones in the evolution of the Java programming language

system on each platform. This package is part of the core of the Java platform: no Java release is missing this package.

AWT was the original API made available with Java 1.0.2. All the graphical interfaces developed by programmers used the tools provided by this API. In our application, the first applets are coded using classes which exclusively belong to this graphical toolkit. We were trying to guarantee that at least these applets would load on every virtual machine that tried to interpret them.

Being a simple API, it was soon upgraded and enhanced by more powerful APIs.

3.2.2 Java Foundation Classes & Swing

The Java Foundation Classes (JFC) were first announced at the 1997 JavaOne developer conference. The JFC are composed of a group of features which help developers build graphically advanced GUI's for Java applications. They are a superset that contain AWT, but extend the toolkit by adding many components and services. The JFC contain the following features:

1. The Swing Components.
2. Pluggable Look and Feel.
3. Accessibility API.
4. Java2D API.
5. Drag and Drop Support.

Swing was the project code name for the lightweight[3] components in the JFC. Swing improved most of the classes AWT had, and introduced some new graphical items and properties. Swing allows the developer to specify the look and feel of the applications being created whereas with AWT the look and feel was always platform dependent. This is the reason why the Swing applet in the application looks the same in all the machines tested, while the AWT applet depends on the platform.

The other features contained on JFC are not relevant to the application developed and are therefore left uncommented.

Table 3.2 shows the plug-in needed to interpret applications and applets developed using different Java platform configurations [Mica]:

In our application we test for Swing support. After testing for AWT we request the user machine to play a Swing applet. Not all the Java versions understand Swing, so special attention has to be payed at this point. If the client machine fails to load a Swing applet, the application must still keep on running since an old version of Java may be present, and still some of the audio files should be able to load successfully.

The first JFC release was made as an extension to Java's Development Kit 1.1 (JDK 1.1) and was shipped as JFC 1.1. This first JFC 1.1 contained the Swing 1.0.3 API. At this time Java 2 had still not been developed and this was the only way to have Swing support. When

[3]Lightweight components are those that can be executed without any native operating system tools

[4]Some developers call this Swing version 1.3 since it fi xes some bugs, adds performance enhancements, and has new API additions, but Sun does not refer to it as Swing 1.3. Instead it considers it bundled with the SDK

[5]Similar thing happens with the Swing components present in SDK 1.4

Java Platform	JFC/Swing version	Java Plug-in
JDK 1.0.2	*none*	*none*
JDK 1.1	JFC 1.1 & Swing 1.0.3	1.1.1
JDK 1.1	JFC 1.1 & Swing 1.1	1.1.2
JDK 1.1	JFC 1.1 & Swing 1.1.1	1.1.3
J2SE 1.2	JFC 1.1 & Swing 1.1 bundled with SDK 1.2	1.2
J2SE 1.2.1	JFC 1.1 & Swing 1.1 bundled with SDK 1.2.1	1.2
J2SE 1.2.2	JFC 1.1 & Swing 1.1.1 bundled with SDK 1.2.2	1.2.2
J2SE 1.3	[4]Bundled with SDK 1.3 (fi xed bugs, new features)	1.3
J2SE 1.4	[5]Bundled with SDK 1.4 (new bugs fi xed, new components)	1.4

Table 3.2: Plug-ins which provide applet support for different Java configurations

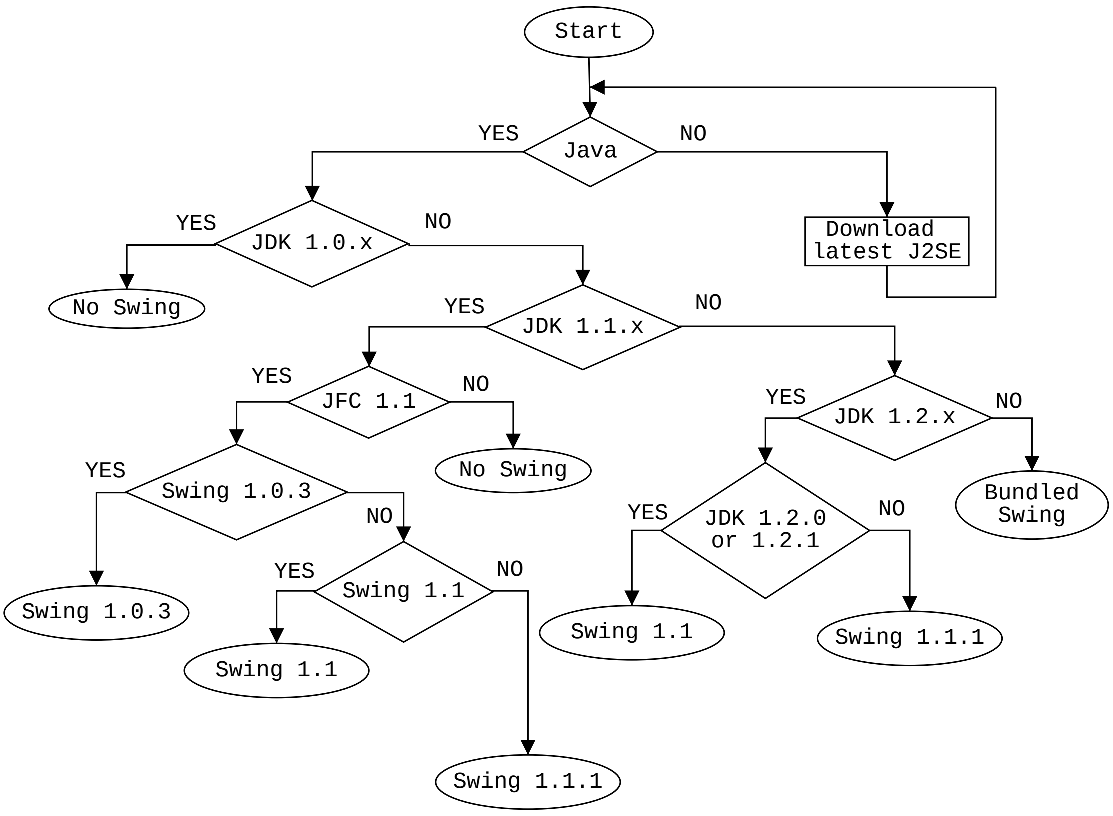

Figure 3.1: Swing availability through different Java configurations

Java 2 was finally released, the JFC were made a core part of it and were shipped together with the Development Kit. Nevertheless, Swing versions 1.1 and 1.1.1 could still be obtained as an extension to JDK 1.1. With Java 2 Platform (Standard Edition) version 1.3 no more extensions for Swing support on JDK 1.1 were released. On figure 3.1 we can see the different possible configurations that give Swing support to the Java platform.

The first version of Swing that was released (Swing 1.0.3) used the instruction showed in table 3.3 to load the classes inside the Swing package:

import com.sun.java.swing.*;

Table 3.3: Instruction used to load the Swing package in Swing version 1.0.3

The next Swing version that was released (Swing 1.1) changed the way to load the package to what table 3.4 shows.

import javax.swing.*;

Table 3.4: Instruction to load the Swing package in Swing version 1.1

Although in our application we tend to use the earliest versions of the Java platform to compile the applets (since this solution offered the best results in the loading of applets on every tested browser), we could not compile our Swing applet with the very first version of the Java compiler that understood Swing. If we had done so, the applet would have only loaded on machines using plug-in version 1.1.1 (or virtual machines from 1.1.2 to 1.1.8 given they had JFC 1.1 and Swing release version 1.0.3.). This would have meant that the great majority of browsers would have been unable to load the Swing applet even though they were perfectly capable of doing so since plug-in 1.1.1 is really outdated.

The solution adopted was to compile Swing to work with plug-ins newer than version 1.1.1, that is, using the second of the instructions to load the Swing package.

3.3 Sound in the Java Platform

Sound was very limited in the first Java releases, but more and different sound API's have been made available for Java, and now, applets can play a great variety of different audio formats.

3.3.1 Sound with Java.Applet

In the first Java releases, the only sound format that was supported by Java was the 8 bit, mono, μ-law, 8000 Hz, one channel, Sun *.au* files. This was made possible through some classes available in the Java.applet package. The first audio samples presented in the application have this format. This would guarantee that if the client machine had the ability to play sounds and had Java enabled, at least the basic sound applets would be correctly loaded and heard. The Java.applet classes were available since JDK 1.0.2.

3.3.2 Java Media Framework (JMF)

Sun's Java Media Framework API [Mice] consists of a series of tools that make synchronization and control of audio and video (and other time-based data) possible with Java applications. This allows real time data (streaming audio and video for example) to be added to applets and Java applications.

On its first release on August 1997, JMF 1.0 only presented the tools for the playback of time-based data. That meant synchronization, control, processing and presentation of streaming and stored time-based media, such as audio, video and MIDI. Java had the tools to become a time-based media player.[6]

JMF 2.0 went further and enabled media capture, streaming, transcoding, a pluggable codec architecture and more control options over the media being transmitted.

JMF version 2.0 and above, uses the Java Sound API for sound rendering and therefore there is a need for Java Sound to be present on the system. JMF is distributed with Java's SDK since Java 2 version 1.3, but an earlier version of JMF was released to be used with JDK 1.1.x in a similar way that the JFC package was. This made Java Sound available for the first Java releases.

The number of supported media formats by the latest JMF version can be found in Sun's web site. [Mic03]

[6]Studies concerning the versatility of a Java media player can be found in [Nyk99]

Capturing Audio through Java Applets

Starting from JMF 2.0, audio and video capture is made available for Java applications. Nevertheless, by default (for security reasons), JMF does not allow data capture from an applet the same way that the virtual machine does not let you read or write files on the host that's executing it. Such actions are considered to be of high risk, and are protected by the Java security manager. If the user wants to enable these features, it is necessary to turn them on explicitly by using a policy file. [Mic98]

A simpler way to enable audio and video capture is through JMFRegistry [Mici]. JMF keeps a registry of different settings which the user can modify by executing the JMFRegistry program. By enabling the *Allow capture from Applets*, and if the browser's security permits loading some native libraries, then capturing data from an applet is made available to the user.

Capturing audio will, for example, enable programmers to develop telephony or audio conferencing tools among other applications.

3.3.3 Java Sound

Java Sound is another sound API made available for Java. It is a part of all Java 2 Platforms although the full featured API is only available since version 1.3 and above. As mentioned before, it is also available as a part of JMF 2.x for use on JDK 1.x platforms.

Java Sound provides a high quality 64 channel audio rendering and MIDI sound synthesis engine which adds high quality audio to Java applications allowing the developer to play many different types of audio clips. The following formats are supported:

1. Audio file formats: AIFF, AU and WAV
2. Music file formats: MIDI Type 0, MIDI Type 1, and Rich Music Format (RMF)
3. Sound formats: 8- and 16-bit audio data, in mono and stereo, with sample rates from 8 kHz to 48 kHz
4. Linear, a-law, and μ-law encoded data in any of the supported audio file formats
5. MIDI wavetable synthesis and sequencing in software, and access to hardware MIDI devices

6. An all-software mixer that can mix and render up to 64 total channels of digital audio and synthesized MIDI music

Java Sound's engine will default to 16 bit stereo with a sample rate of 22 Khz, but if hardware does not support these settings it will automatically use 8 bit or mono output.

In order to function in a correct manner, Java Sound needs a soundbank. Soundbanks are necessary for the correct operation of the internal software synthesizer that is shipped with Java Sound[7]. Soundbanks contain a set of instruments that can be loaded into a synthesizer [Mic02a] for the synthesizer to play.

Java Sound applets require an updated JVM to play them. It wouldn't be surprising to find systems that cannot play these applets since version 1.3 of the JVM is needed for full support. Versions 4.x of Netscape's Navigator and versions 4.x and 5.x of Internet Explorer will not be able to play them if no other JVM is available[8]. Therefore, Sun's plug-in 1.3 or above is needed to play Java Sound applets on these browsers[9]. Another option is to have JMF installed on the system which allows Java Sound to be played on 1.1.x virtual machines.

Tritonus

Tritonus is an independent, open source implementation of the Java Sound API [Trib]. The Java Sound API specification is the same, both for Java Sound and Tritonus; they are simply different implementations of the same API.

There are areas of the implementation, however, where they take different approaches. Sun's implementation has a built-in software synthesizer and will not work as smoothly with hardware synthesizers whereas Tritonus, on the other hand, works with hardware synthesizers much better than it does with software ones [].

Tritonus offers a series of plug-ins that increase the number of different sound formats that can be used by Java applications. Although Tritonus is the only full Java Sound implementation for JDK 1.2.x, many of these plug-ins will only work on versions 1.3 or above of

[7]In the Windows platform, no synthesizer is shipped with the JRE, so it must be additionally downloaded.

[8]Netscape has version 1.1.5 virtual machine and Explorer has version 1.1.4.

[9]Versions of Netscape prior to 4.8 do not even have the possibility to use another virtual machine other than its own embedded machine.

the Java platform. The most relevant plug-ins for our topic are as follows: [Tria]

1. Ogg Vorbis Decoder. Enables decoding of Ogg Vorbis bitstreams and is available for all platforms
2. CDDA extraction. Allows reading, ripping and extracting digital audio from an audio CD and is available only for the Linux platform.
3. mp3 Decoder. It lets you play mp3 encoded files. Available for all platforms.
4. mp3 Encoder: Allows mp3 encoding with Java Sound. Only available on Linux and Windows platforms.

Figure 3.2 shows the possible Java configurations that allow the user to have full Java Sound availability.

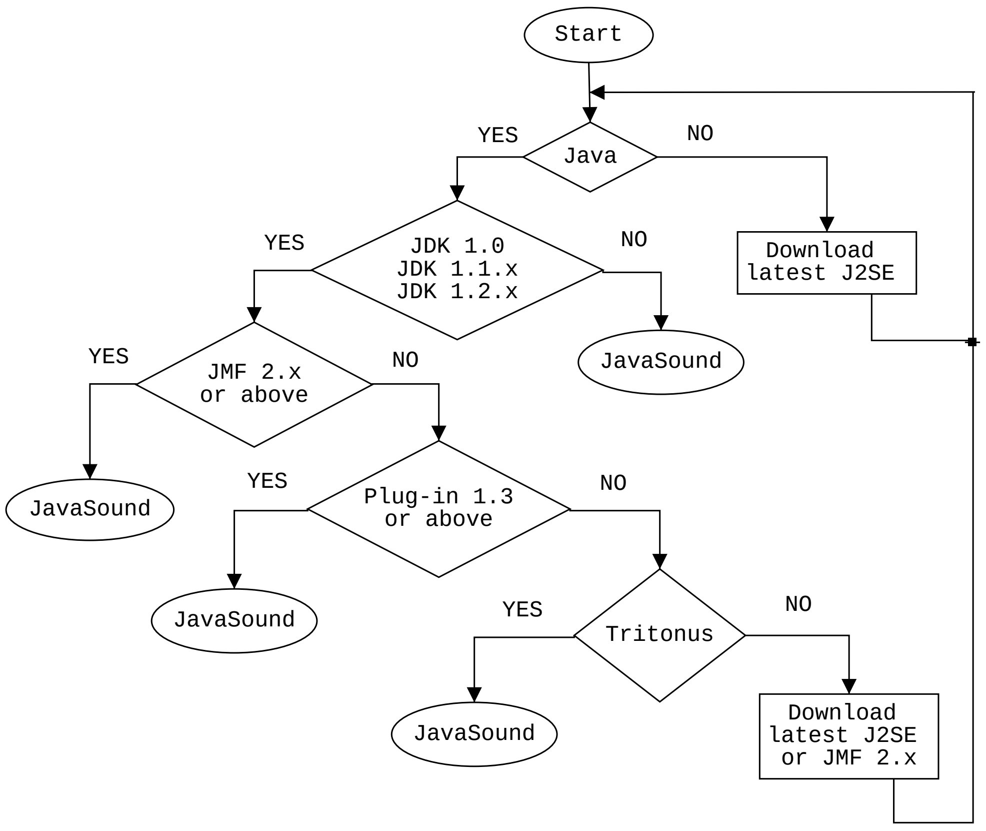

Figure 3.2: Full Java Sound availability through different Java configurations

3.3.4 JSyn

Jsyn is a Java API for synthesizing audio. This means that Java developers have another set of classes and methods to use when wanting to add sound to their programs.
Unlike all the other API's presented so far (Tritonus is an implementation of an API, not an API itself), JSyn does not belong to Sun Microsystems. It is an API created by Soft-Synth [Sof], and therefore you need to install an extra plug-in to play these type of applets, since the API is by default not supported by Sun's virtual machines.

Jsyn can be used to generate music, sound effects or even audio environments. To do so, JSyn has a free Software Development Kit which can be downloaded from their site. Among its features, JSyn presents a library of oscillators, filters, noise generators and effects, audio input support for voice recording and processing, combined audio sample playback with other synthesis and processing units and is available for Windows, Apple and Linux machines through the use of plug-ins.

Chapter 4

Perl, CGI & HTTP

The Common Gateway Interface (CGI) is an interface specification that allows web servers to transfer information with special programs called CGI scripts [web02b]. These programs, which can be written in different languages, have to conform to the CGI specification and allow the sending and receiving of data from and to the server.

A plain HTML document is a static resource that is sent from the server to the browser. If another browser client requests the document it will receive exactly the same HTML file. A CGI program, on the other hand, generates the HTML document dynamically considering certain parameters obtained from the user's request and the environment where the CGI script is being ran on. Since it is executed in real-time, it can output dynamic information.

When a web server receives a request for a static HTML file, the server looks for the page on its file system and serves it back to the client. When a web server receives a request for a CGI script, the server executes the script, and it is the script which generates the HTML document and sends it back to the server which then forwards it to the client. Figure 4.1 shows this behaviour.

Although CGI scripts can be written in many programming languages, Perl has become by far the most widely used. There are various reasons for using Perl in our application, but the main ones are its easiness of use and the huge amount of Perl modules available for very specific tasks. For example, modules such as CGI:Application or HTML::Template which are focused on specific issues that concern our application, are used, making it easier for us to achieve our goals. Apart from that, Perl is easily portable and is available on many platforms, which is also a point to take into consideration if we want to make the code available on different servers with different operating systems.

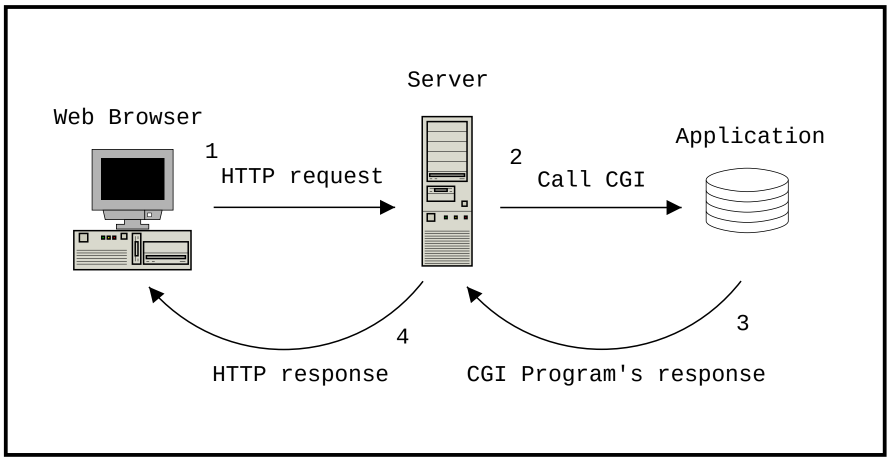

Figure 4.1: How a CGI application is executed

4.1 Perl modules

Due to the characteristics of the testing application developed (basically that it is web oriented), we needed to use certain tools that made things easier for us. CGI::Application is an Object-Oriented Perl module which is used to build reusable web based applications and is built on standard non-proprietary technologies such as CGI or the CGI.pm module. The functioning of the application is entirely based on this module.

Being a web application, a great amount of HTML coding had to be done since the interaction with the user is achieved through web pages. In order to separate the actual coding of the application from the HTML code that is to be presented on the browser, it was chosen to use another Perl module, HTML::Template. HTML::Template allows external HTML files to be created for each screen in our application. These templates contain pure HTML code except for a small additional syntax for including variables that are set by calling particular methods from CGI::Application as will be later explained.

The CGI.pm module has become the standard tool for the creation of CGI scripts with Perl. It provides a very simple object oriented interface for the most common tasks in CGI programming. In particular, dealing with parameters and query strings is made simple and straight forward. Our application requires information to be sent back and forth to the server so dealing with input and output in an easy way was required. Originally, we only needed the user agent string and the user's answers concerning audio performance on the machine being tested, but many other parameters (mainly control parameters) turned out to be necessary for the correct execution of the testing application.

4.1.1 CGI::Application

CGI::Application is a Perl module based on the idea that any web based application is made up of, or can be organized as, a series of states or run-modes. These different states are controlled by a single parameter called run-mode parameter. Modifying the value of that parameter will take the application to a different state, making it possible to jump from one mode to another simply by modifying one parameter.

When the application reaches certain state through the value of the run mode parameter, the subroutine which corresponds to that state is executed. It is the duty of the subroutine to perform the corresponding actions and send the result to the server which will then forward the generated HTML file back to the browser. In the testing application each state corresponds to a different HTML file. We will therefore have a series of HTML files linked together by a mode parameter whose value will determine the HTML file that is being displayed on the browser. The mode parameter is changed through the user's interaction with the application. Depending on the user's answers concerning the audio and Java testing, the application will follow a certain route of run modes and test for different things. Different web pages will thus be displayed.

CGI::Application only uses two files to control the whole application, a file called application module and a file called instance script.

Instance Script

The instance script is what is actually called by the web server. It is a very small, simple file which simply creates an instance of the web application when a user accesses the program, and calls the run() method (thus the name instance script) to start it *running*. By inheriting from CGI::Application we inherit some of these built-in methods. The new() method and the run() method are needed for the starting off the application.

1. new(): This method is the constructor for the CGI::Application module. It creates the application object. Inside the new method we can insert some parameters (such as the path where the templates are to be found) so that the object created has some special features.

2. run(): The run method sends parameters to the created object for it to execute certain subroutines in the application module. It first tries to determine the state the application is in by looking at the run mode parameter. If no run mode parameter is specified, then the application defaults to the value of start_mode(). Once we know where to start, the run() method determines which subroutine to execute by looking at the mapping tables located in the run_mode() method, inside the setup() method of the application module.

This way a new application object is generated with every call to the instance script, but a different mode is executed each time, depending on the value of the run mode parameter. This means that each time the user sends an answer back to the server, the reply contains the new value of the run mode parameter (among other parameters) and the application is driven somewhere else.

Application Module

The application module contains all the code specific to the application's functionality. In other words, it contains the whole application. All the possible run modes are configured in this module as subroutines. When the instance scripts sends the order to run, one of the multiple run modes is selected and that subroutine is executed.

The application module is a subclass of the CGI::Application module. Being so, it inherits all of the methods available in the parent module, some of which are intended to be overridden. The most important method to override is the setup() method, and should be used to define the following properties:

1. mode_param(): This is the name given to the parameter that will control the state the application is in. If none is given, 'rm' is the default.
2. start_mode(): This method specifies the starting mode of the application. When no mode_param() is given, the application starts from the start_mode() mode value.
3. run_modes(): This method maps the different available modes with the subroutine that is to be executed when the application reaches that mode. If a mode that is not available is demanded, no run mode is executed. Instead, a default error run mode called 'AUTOLOAD' is loaded.

When any of the run modes is executed, the result of the execution is sent to a template file which is expecting them. When the template file receives these parameters, it becomes a valid HTML file and it is sent to the browser for it to display the new created page. All the process, starting from the CGI call, is explained in a graphical way on figure 4.2.

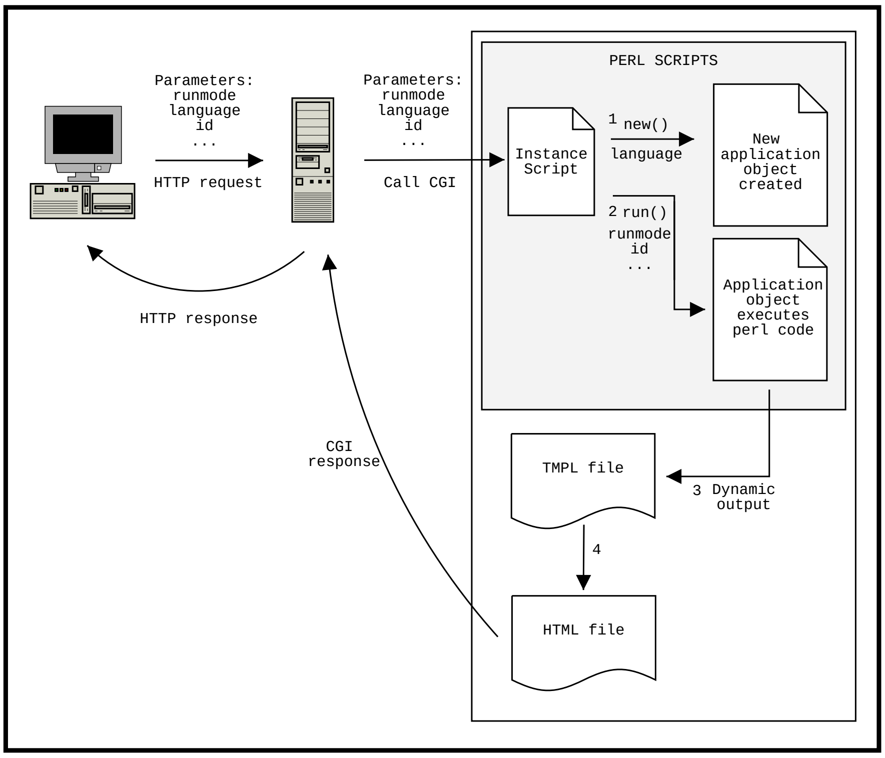

Figure 4.2: CGI::Application: Instance Script & Application Module

The application module can be placed outside the server's document root, making it unreachable via the web server. This fact increases security. The only condition to the location of this file is that it must be somewhere in the Perl library search path.

An extract of the application module can be found in appendix B

4.1.2 HTML::Template

HTML::Template is a Perl module that deals with HTML and Perl. This module allows us to create HTML files which contain special tags that give us extra functionality. These tags contain names of variables which, at first, do not have any value. When each run mode is executed, the variables are given their values and the HTML code is generated. This way, the HTML file, or *template file* (since at first it is a template waiting to be updated with the corresponding information) is dynamically created. The new tags are not regular HTML tags, but are made available through the HTML::Template module.

On the application module we set the values for the variables. Typically *if-else* statements precede the assignment of these variables so different paths can be taken depending mostly on previous answers given by the user:

```
$tmpl->param(change_mode => ’rm_parameter’);
if (parameter eq value1){
    $tmpl->param(new_runmode => ’runmode2’);
} else {
    $tmpl->param(new_runmode => ’runmode3’);
}
```

On the template files we have the variables inside the new special tags with no values. (Code taken from an HTML form):

```
<INPUT TYPE="hidden" NAME="<TMPL_VAR NAME="change_mode">"
       value="<TMPL_VAR NAME="new_runmode">">
```

When the final HTML file is generated, the code that is sent to the browser is plain HTML:

```
<INPUT TYPE="hidden" NAME="rm_parameter" value="runmode2">
```

The code in the template file is just an extension of the HTML code, plain HTML plus some extra tags. Being so, the coding of the application and the presentation (user interface) can be completely separated which is desirable for maintenance and clearness of the code.

4.1.3 CGI.pm

As stated before, CGI.pm is the standard tool for creating CGI scripts in Perl. Our application receives key parameters through POST and GET methods on the server side. Using CGI.pm, these parameters are easily accessed and can be parsed in a very simple manner.

When the server receives an HTTP petition requesting *JavaTestingApplic.cgi*, it also receives some parameters through the query string. These parameters are crucial for the correct functioning of the application. Information such as the next run mode the application has to display or the language in which the information is to be presented is sent through the query string. It was therefore desired to use a suitable tool for reading the parameters.

Example code of reading parameters from the query string: (The parameters read are *id* and *ownmd*. The URL is also registered.)

```
sub upgrade{
    my $self = shift;
    my $q = $self->query();
    my $url=$q->url;
    my $id=$q->param('id');
    my $prevmd=$q->param('ownmd');
```

But not only do we have to read information that is sent to the server, the CGI script has also got to generate the HTML file that the server has to send to the browser. CGI.pm made this information flow easy and straight forward.

4.2 HTTP Methods

A web browser has several ways to request a web page from a server. The most common ways are via the GET and POST methods, but others such as HEAD, PUT or DELETE are also available but many servers have not implemented them for CGI scripts.

Depending on the method chosen to *speak* with the server, the information passed between client and server is sent or received in different ways. Both methods request a page from a server and both methods can send data to a server (POST always does, GET is able to), but it is done in different ways.

4.2.1 GET Method

As its name suggests, the GET method is used when a browser wants to *get* a web page from a server. The web browser sends a request with a GET method to the server which then serves the file back to the client. The GET method does not only allow the browser to ask for web pages, it can also send data to the server. The way to do so is by adding the information to the end of the URL. The information sent is seen on the navigation bar of the web browser and has the structure that is presented in table 4.1

http://www.site.com/applic.cgi?parameter1=value1&parameter2=value2&parameter3=value3

Table 4.1: An example of the format of a query string

The big drawback to this way of sending data to the server is basically that since the information is attached to the URL, the data sent is visible to the user on the navigation bar. In this way, the information can be seen and users have the possibility to change it, disturbing the correct functioning of the application.

This is a huge drawback in terms of our application because session control is achieved through the information sent to the CGI script, and if that information can be easily corrupted, the whole session control could be lost[1]. Although we would like to have all of the information sent to the server unseen and unavailable to the user, it will be necessary to use the GET method in some cases as will be explained in further chapters.

4.2.2 POST Method

The POST method is mainly used to send information to the server. Unlike the GET method, this time, a connection to the server is opened and data is sent through this connection. This way, the data sent is not directly shown to the user[2]and it cannot be altered in an easy way. Besides, no limit is imposed on the number of characters sent. The information sent through a POST connection has the same structure as the one sent by the GET method requests; pairs of parameters and values separated by ampersands (&).

Our aim was to keep the session control of the application through parameters sent to the CGI Script via the POST method. This way we could keep the user away from the temptation of freely changing state, but in some cases the application could not move on with the exclusive use of POST requests, so we had to use GET in some cases.

[1]Some web servers will also limit the maximum length of the line they will accept as part of a GET request (typically 255 or 1024 bytes [Har00]) but this does not affect us since we will send small amounts of data.

[2]The information sent to the server can be seen as parameter-value pairs simply by looking at the HTML source code, but it is not as easily available as with the GET method requests.

Chapter 5

Testing and Usability

5.1 Testing Procedures

The objective of software testing is to detect all the possible malfunctions that a program may have before making it available. Unfortunately, it is impossible to find all the possible problems by testing. If we could test an application with all the possible input parameters, then we would know how the software behaves in every possible case. This is not possible on a general basis, and therefore we need to plan some testing strategies that will guarantee, to some extent, that the application is ready for deployment.

Testing procedures can be divided into two main categories, black box testing and white box testing [Ngu01]. However, a new strategy called grey box testing can also be taken into consideration, specially in web based applications like the one being developed here.

5.1.1 Black Box Testing

Black box testing tries to determine if the application does what it is meant to do, if it works as expected from a user's point of view. This type of testing is mainly based on input-output pairs of values and tries to see if the application behaves how it should given different types of input.

Black box testing must deal with the widest scope of possible data input. The application should not have any errors when the expected input values are given, but also with the not expected ones, giving an error message or behaving as established by the programmer. In this sense, it is recommended to divide all the possible inputs into different categories, and test at least with one of the inputs from each category. Special attention has to be payed with the frontier values that specify the different categories, since experience shows problems often arise here.

The inputs expected in our application are just the user’s answers (controlled mouse clicks), and some information from the browser running the software. The information obtained from the user clicking on the different buttons will present no surprise since we have foreseen the possibility of having such replies, but the information sent from the browser has to be tested thoroughly.

The application expects to receive some kind of user-agent string format that it understands, but since the user-agent string can be almost anything and user-agent strings will probably change through time[1], the application must be prepared to act accordingly to what it may receive. We must test for blank user-agent strings, numbers instead of letters, non valid strings, extremely long strings and, of course, expected user-agent strings in order to have a reasonable collection of possible inputs. In every one of the the testing cases we must make sure the application works as expected.

5.1.2 White Box Testing

Whereas in black box testing, the only matter we were interested in was that the program worked correctly in a functional way, without knowing anything about what was happening on the *inside* of the software, with white box testing, we want to deal with the program’s source code. White box testing tries to determine how many of the lines of source code are executed, if they are executed at all, and if so, if there are any errors when they execute. White box testing deals with testing from the developer’s point of view.

Sometimes, some lines of code are only executed if the input is a very specific one and will never be executed on a normal basis. It is the task of the programmer to prepare a set of tests that will execute all the lines of the source code to see if there are any errors when the program loads them into memory. If the case happens to be that some parts of the program are never executed, no matter the input received, then, probably those lines of code could be done without.

It is desirable to reach a high percentage of coverage of the source code executed before white box testing is considered to be done.

[1]See chapter 2 for further details

5.1.3 Grey Box Testing

Grey box testing is mixture of black and white box testing. Nguyen's [Ngu01] definition is as follows:

Grey box testing consists of methods and tools derived from the knowledge of the application internals and the environment with which it interacts, that can be applied in black box testing to enhance testing productivity, bug finding, and bug analyzing efficiency.

Grey box testing tries to combine both black and white testing strategies, but also introduces a new concept; the environment. Web applications are not like stand alone applications. In web applications, as well as having to interact with the client machine there are other variables that can affect the way the application behaves. The application will be available from a web server which will have to be accessed from a remote machine through a crowded network of interconnected computing devices. There will be an Internet connection speed that will limit the throughput of the data sent, there are some protocols speaking to each other on both sides of the connection, there is a client software that the user has to have in order to run the application, dynamic delays, firewalls, proxy servers and other items that make Internet applications different form stand alone applications.

Grey box testing benefits form the knowledge of how the connectivity works. It can be considered as part of the internals of the application itself. If we know how the *inside* of the application works we can prepare tests to see if there are any errors when the environmental variables change. This kind of testing allows us to find other type of errors that would be difficult to find with black and withe box testing.

5.2 Testing the Application

Knowing how the internals of the application work, it was simple to spot where the critical points were bound to appear. The parsing of the user-agent string and the points where different technologies (JavaScript and Java) were tested were candidates to present the most critical points in the application. In the end, the loading of the first applet turned out to be the most problematic spot, since it generally determined the browser's behaviour through the rest of the application.

The application was tested by the people in the Laboratory of Acoustics and Audio Signal Processing (Helsinki University of Technology). The results and the settings of the tests can be found in chapter 8: Conclusions and Future Work.

User-Agent String

When parsing the user-agent string, we were looking for a known browser, its version and the platform under it was running. The browsers that the application *understands* are: Microsoft Internet Explorer, new versions of Netscape (older versions did not specify they were Netscape browsers, they just had the *Mozilla* token and a distinction with the Mozilla browser is not made in this application), the Opera browser, Mozilla, Safari, Galeon and Konqueror.

All of these browsers ran the user-agent test successfully and therefore were correctly detected. Different versions of some of them were also tested and the results were always satisfactory.

Browsers with user-agent strings that were not expected were also tested. Even blank user-agent strings were included in the tests. In this case the result was always *unknown* for the browser, version and platform.

JavaScript and Java

Testing for JavaScript was another of the points that was thought to be critical. The testing phase showed that JavaScript was always enabled on the browsers, except when disabled deliberately to test the application. Not even one of all the testing results (around 40 different browsers) had JavaScript disabled. In order to see how the application reacted when there was no JavaScript we had to manually disable it and run the tests.

All the possible paths that the application may follow combining the enabling and disabling of Java and JavaScript had to be tested. Four cases had to be tested: Java enabled with JavaScript enabled and disabled, and Java disabled with JavaScript enabled and disabled. We also changed the different values during execution time to see how the application reacted.

It turned out that we had to clean the code in the detecting of Java and JavaScript because there were too many run modes and some of the code was repeated. The structure of the application can be seen on appendix A and it shows the great number of run modes that surround the Java detecting web page.

Applets

The loading of the first applet is the most critical point of the application from the results' point of view. If the applet does not load, the user cannot go any further and none of the audio tests can be performed. If the redirection applet works, then the rest of the applets work (except maybe for the Swing applet) even though the sounds may not be heard.

The redirection applet does also show the Java version and some conclusions can be drawn from this piece of information.

During the development of the application we did not pay much attention to the size of the audio samples played through the applets. Since the samples were stored on the same machine as the application there was no delay when the applets played them.

When testing the application from a machine that is not located on the university networks, the delay becomes really important. Even the speech samples played which are relatively small (around 80 KB) take some time to download. During this time the applet does not fully load and the user may feel confused.

Music samples are much bigger and therefore information concerning average download times is supplied. The downloading does not start automatically, therefore the applet can load before the user chooses to play the music sample.

5.3 Testing Usability

Since the application is a continuous flow of web pages and will be available for the Internet community, it is necessary to keep in mind some usability concepts. Given that our application serves educational purposes, not all of the usability concepts can be followed (we are not trying to sell anything nor want to attract users to our web site), but the basic ones can still be applied. In the words of Jakob Nielsen: [Nie00] *Usability rules the web*

Before starting the design of the interface we had to answer some basic questions about who will be using the system and what we would like to accomplish with it.

5.3.1 Potential Users

At first, the idea was to create an application that would serve the Laboratory of Acoustics and Audio Signal Processing (Helsinki University of Technology) to determine whether the browsers in the student's machines were able to play sounds. This would determine whether their computer would be able to use applet-based course material on the web. The project evolved into an educational testing application which not only serves this first idea, but also explains how all the sound related hardware and software elements in a computer should be configured and placed. The scope of the potential audience was therefore broadened so that the application could serve anyone, not just technical students from the Helsinki University of Technology, to determine and learn about their own machine's browser and audio setup.

Being so, the language and mood used throughout the application had to be changed slightly since it was no longer focused on users with general knowledge on computer systems, but was focused on any user that wanted to learn something about his or her system, be an expert or not. The level of subject expertise was set to 'novice'. This change would also affect the distribution of different browsers used for accessing the application.

5.3.2 Meeting the Application's Goals

The purpose of the application was to determine the capability to play sounds through Java applets that a particular browser had in the users' machine. Since the application was thought to serve not only technical students, but inexperienced users too, each step in the testing had to be explained thoroughly and in a simple manner.

This naturally evolved into an educational application where basic audio principles were explained, and basic setup and configuration schemes were taught. In the end the application's goal is to show some of the system properties of the users' machine (especially browser, Java version and audio related issues) and explain how the tests are working, what they are doing, and why we perform such tests, always keeping in mind an inexperienced user and trying to make explanations as simple and clear as possible. The output from the application should be the user's system properties, and some gained computer knowledge for the user.

5.3.3 Page Design and Content

The design of the whole testing application is plain and simple; we do not use fancy graphics (actually we do not use graphics at all[2]), we just present a testing application which loads

[2]The application had a picture on the initial page but it was removed to maintain the whole look and feel

quickly and easily determines browser audio capabilities on the client side.

Page layout

In this sense the page layout is really simple. A centered heading indicates what the application is doing in that run mode. There is some text which later explains more in detail what is happening or is about to happen. Since users do not tend to read the whole of the text, no matter how long or short it is [Nie00] the conclusions are presented on the first lines.

To help the user follow what the application is doing without having to read much, there is a centered box in grey on each page which is clearly outstanding and explains in three or four lines what is happening. By reading this box in the middle of the page the application can be easily followed. The text was kept simple and the sentences short.

The testing results obtained from the testing phase which took place at the laboratory indicated there was too much text on the pages. A great percentage of the users complained about there being too much text to read. The approach that was followed was to reduce to the maximum possible extent the amount of text that appeared on the pages. But, if there was any kind of problem with the browser and some help pages were needed, these would have all the text necessary even if the page ended up having lots of text. If this was the case, the page was splitted into two.

This gave the application a consistent look and feel. None of the users complained about the user interface which they found easy to use and simple. Once they had gone through the first three run modes, all the rest of the pages were similar.

Speed in loading

Speed in the loading of the pages was one aspect were we thoroughly focused on. Very short pages were created. The only drawback to the speed of the loading was found when using Java applets. There was a considerable delay which we could not speed up. The Java runtime environment takes some time to load, and the applets need to download as well as the audio samples. Nevertheless the applets were kept as small as possible and frequently reused so that once loaded, they needed not to be reloaded again (changing the *param* tag of the applet would change the way the applet worked, but no reloading was needed.).

As stated before, the delay experienced in the downloading of the samples was none at the server, but considerable when leaving the university network.

User interface

Another aspect in the design of the pages that was considered vital was the user interface. The user interface is the window through which the user communicates with the application, therefore, special attention has to be payed to it. Our goal was to make the interface transparent to the users so that they could focus on the testing without even noticing anything strange about the way they moved from page to page.

In this sense two simple buttons are always placed at the bottom of the page[3]. The one on the left is the one the user is expected to press if there are no problems; that is, if JavaScript and Java are enabled, and if the applets load and the audio samples are heard.

The one on the right can abort the application, force it to continue without JavaScript or indicate that an audio sample has not been heard among other options. When this button is pressed the application has found something that may not be working as expected.

The text inside the buttons was made as informative as possible.

Some of the run modes in the application have more than two options the user can choose from. At first all the possible paths the page offered where placed as buttons on the pages. This gave them an untidy feeling and inconsistent look, but allowed to have buttons submitting information using the POST method and therefore sending it unseen. It was decided that keeping the same layout on the pages and not confusing the user with new and misplaced buttons was more important than sending the information hidden. Therefore, two buttons were kept and plain HTML links (which use the GET method) were used for the extra options.

Scrolling

The information presented on each page was tried to be kept inside the limits of a 800x600 pixel screen. However, depending on the size of the text of the browser and the amount of text on some pages, this was not possible. We were aiming to have an application where the user did not have to scroll down the page to find all the information on it since the usability is reduced, but having that screen size, some run modes needed to be scrolled down to reach the buttons.

[3]Only on the upgrade and help pages may the user find only one button which redirects the application to the previous run mode.

Horizontal scrolling was avoided by placing all the text inside a table with 60 % of the width of the browser screen size.

Chapter 6

The Structure of the Application

6.1 Language Support

The structure of the language support implemented in the application allows any language to be added to it. It was primarily considered that the application should have English, Finnish and Spanish language support when it was released.

Each time the script is called and an application object is created, it is done so with a path that indicates where to locate the file that is to be loaded. This path is specified with each request, and it selects a directory on the servers file system. The application module then loads the template file (the name of the file is the same name for all the languages), and the file is fetched from the directory specified on the path variable. This way, just by sending the path on each of the requests, the language support is achieved.

The template files are translated into all three languages. When it comes to the applets, the same applet is used regardless of the language. What changes is the parameters that the template sends to the applet. Having the text that appears on the applets as parameters, will let us reuse the applets.

6.2 Session Control

In web based applications, session control is an important issue since there may be more than one user accessing the system at the same time. In our application we also need session control to know which information belongs to which browser.

UNIQUE_ID

The idea is to assign a unique code to each user at the beginning of the application and keep that code in every request that is later made. The Apache server uses the mod_unique_id module to generate unique identifications. These codes are based on a quadruple which mixes IP addresses, processes identification numbers, time and counters to get a random 112 bit number. This is then coded using a closed alphabet made of capital and small letters, numbers, @ and -. It is passed to the CGI as an environmental variable called *UNIQUE_ID*

Hidden fields

The most common ways to achieve session control is through cookies [KM97], hidden fields, or through identifiers on the query string. We use the later two.

It was desirable to only use the hidden fields method which passes the user identification code from one run mode to another without it being shown to the user. Since our application moved from run mode to run mode through the clicking of form buttons this was easily obtained. An example of the hidden fields on a form button is shown (some of the template variables can be also seen):

```
<FORM METHOD="POST" ACTION="<TMPL_VAR NAME="SELF_URL">">
      <INPUT TYPE="hidden" NAME="rm" value="abort">
      <INPUT TYPE="hidden" NAME="id" value="P7PpCYL...">
      <INPUT TYPE="hidden" NAME="ownmd" value="brwsrok">
      <INPUT TYPE="hidden" NAME="lang" VALUE="en">
      <INPUT TYPE="submit" VALUE="Finished Testing">
</FORM>
```

In this case, the hidden parameters that are submitted with the button are: the new run mode where the application will be led if it is pressed, the identification (session control) code, the run mode where the application comes from and the language parameter.

Whenever it was possible, this was the way to move from run mode to run mode and to keep the session control. When the hidden fields could not be used because no POST method could be used, the identification code was sent as part of the query string.

6.2.1 POST problem

When parameters are sent from one run mode to the next one, it is desirable to use the POST HTTP method. This method keeps the information *hidden* from the user. If we press the previous button example code, the application will be redirected to a new run mode without any information appearing on the browsers navigation bar. Should we have done it with a GET method, all the query string would have been seen.

Posting data to the server through plain HTML forms using *<INPUT TYPE="hidden"...>* worked throughout most of the application. However there were times where we wanted to POST information to the server from a Java applet and not directly from a HTML form. The following sequence and figure 6.1 illustrate what happened when an applet tried to POST information to the server.

1. The browser calls the Java applet when getting to the HTML tag <APPLET>
2. The applet initializes and loads itself and waits for the user to interact with it.
3. User interaction forces the applet to establish a connection to the server and send the data corresponding with the previous interaction.
4. The server to which the data was sent calls the CGI script (giving it the information sent) and the script generates the next HTML file on the fly.
5. The HTML file is given to the server which forwards the file back through the connection established to the applet.
6. The response (in the form of an HTML file) can only be seen in the JVM Console.

The applet opens the connection and sends the data, the server receives the data and calls the CGI script. The script then creates the page and sends it back to the server which then, replies through the same opened connection. The information reaches the applet. If nothing else is done, the user will not be able to see anything on the browser. It is the applet who has the information, not the browser. It should be the the duty of the applet to print the HTML file on the browser for the user to see. Java applets can't do this. They have special methods to print on the screen, but they are not browsers and cannot act as a browser and understand the HTML code that has just been sent to them. The applet could show the document it had just received on the screen, but all the HTML tags would be seen and none of the HTML functionality would be preserved since it is the browser who interprets all the tags and can create the appropriate links and display the document conveniently, not the applet.

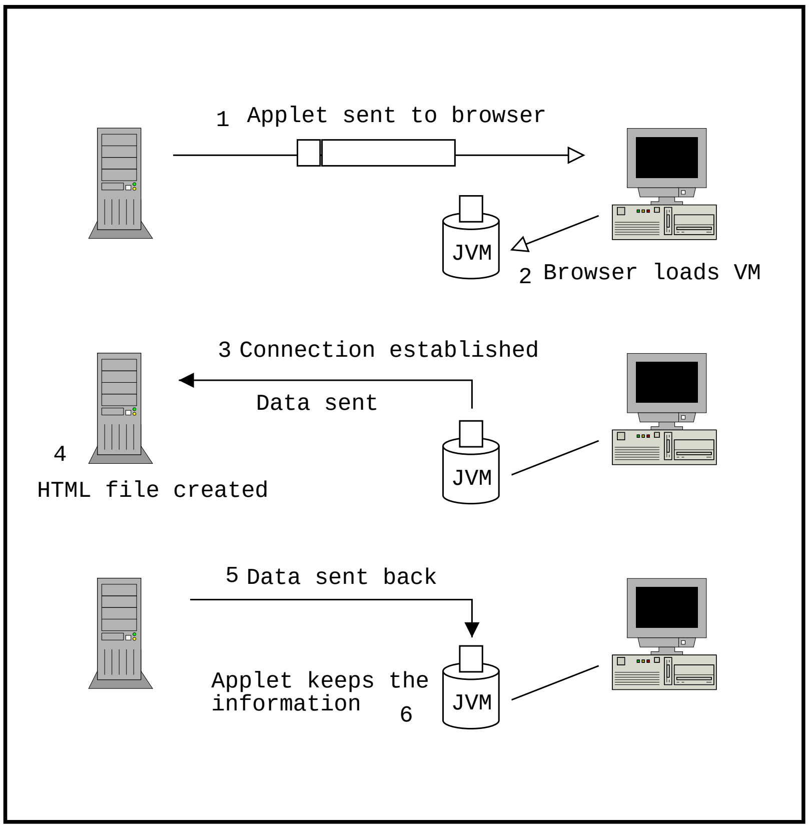

Figure 6.1: Using the *post* HTTP method from an applet

After investigating this issue it was discovered that it could only be done with a browser that was coded in Java which most of the users don't seem to have. Thus, the only way to get the application to move along to the next stage was to get the applet to display a new page in the browser. This could only be done through the *showDocument()* Java method. Unfortunately, this method sends a GET request to the server and not a POST. We had to build the URL that the *showDocument()* method required as a parameter with a query string showing the run mode that was next desired. The information was seen on the browser's navigation bar.

The benefit of using the POST method to send the information through an applet is that it is hidden from the user and cannot even be seen by looking at the source code of the HTML file. The information sent is compiled inside a *.class* file on the server. The disadvantage of using applets to send the data is that after being sent, we cannot display the servers response on the browser, and we need to use the *showDocument()* method to move on to the next step (HTML file) in the testing application. Using a GET request forces us to present the state parameters of the application on a query string that is perfectly visible to the user of the system.

6.3 JavaScript

JavaScript was not intended to be used in our application at first. Using JavaScript forced the user to have a browser that could support JavaScript and besides that, have it enabled. Since we were already testing for Java, we didn't want to test for anything else.

Web browsers can crash when attempting to load a page that contains Java code. We wanted to avoid this possibility whatever it took. After some research and some testing it was found that the only reliable way to know if a Java applet will load on a browser is to execute the code inside the HTML file and hope it loads. But, before attempting to do so there are a few tests we can perform to minimize the possibility of a browser crash. Using JavaScript to determine whether Java is installed is one of those tests.

There are some Java methods that will get information about the Java version installed in the system, but using Java to determine whether Java is present or not, forces us to load an applet and hope that the browser does not crash. Therefore, it was decided to use JavaScript. There is a JavaScript method called *navigator.javaEnabled* that will determine if there is a version of Java running on the browser. The advantage of using JavaScript is that JavaScript does not crash the browser if it is not enabled or not present, it simply does not execute the code, so it is a safer option to run this test first.

The problem to this approach is that JavaScript's answer is not conclusive. If it is found that Java is enabled, we still cannot assume that it is working correctly and loading an applet can still crash the browser.

On the other hand if JavaScript indicates that Java is not present then it may mean it is simply disabled (but is installed on the user's system) or that it is not installed at all. These seemed to be the only two possible options. However results showed that the JavaScript detection was not always trustworthy due to a bug in Mozilla 1.4 on Mac OS X[1]. The JavaScript script would say that Java is always enabled even though it is not.

The third and last possibility is that the browser does not support JavaScript at all or

[1]Bugzilla Bug number 97613 [Mozb]

that the user does not want to enable it. If this is so, the browser will just ignore the code concerning the script and an option to continue without JavaScript will be shown. The application can continue working without JavaScript. The only difference that a user may find if JavaScript is nor enabled is when getting to the music sample testing. Since these tests ask the user to select which of the different samples can be heard (there are more than one sample per page), some checkboxes are made available to explicitly select the audible ones. JavaScript comes handy in validating the answer returned by the user. We check that at least one of the checkboxes is checked when the user confirms he or she has been able to hear any of the samples.

If the scripting language is enabled on the browser, it redirects the application to a new run mode through the GET method and thus, important information is seen on the navigation window.

6.4 User File

Starting from the first run mode that is visited by the user, an information file about the session and the browser running the application is kept. This user file is unique[2] and is created the moment the user selects a language to run the application in.

In this file we store relevant information about the system accessing the application. First of all the user-agent as a whole is kept. It is later parsed to obtain the browser, its version and the platform. Whenever important information is obtained, the file is updated. On this file we will write whether JavaScript and Java are enabled, if the redirection, AWT and Swing applets have loaded, the Java platform version and all the results from the audio samples.

Apart from the results that the application is following to obtain, we also update the file whenever a new run mode is reached. This way we can keep track of the path that users follow throughout the application. Information on the way the users behave (for example when some technology is not enabled) and how long they take on each screen is recorded. This is by no means aimed at storing personal information of individuals (IP addresses, for example, were kept during the testing phase, but are no longer stored), but it serves as a debugging tool concerning the usability of the application. If users seem to be lost whenever something is not working in their system, we can see it by the way they navigate through

[2]As unique as the UNIQUE_ID parameter from the Apache server can be

the application and the time it takes them to move from one screen to the other. This has helped us to figure out (among other things) that the text must be very clear in the case when something has gone wrong, and that whenever things are working out correctly the user does not read all the text.

If the security of the application is tried to be compromized by, for example, the user changing parameters, all these attempts will be stored in this user file for further examination.

6.5 Run Modes

As mentioned in previous chapters, the application is made up of an instance script, an application module and some HTML templates. The instance script creates an instance of the application object depending on the parameters it receives from the HTTP request. One of those parameters is the run mode parameter which selects the subroutine in the application module to be run and the output is sent to the corresponding HTML template so that it becomes an HTML file and can be correctly displayed by browsers.

The Perl code corresponding to the run mode is first executed and then the resulting output is what is sent as parameters to the template. Each run mode maps to an HTML template with the same name (except that the template has the *.tmpl* extension).

The application has 59 HTML templates, 58 valid run modes and the error default mode which is loaded when an invalid run mode is read from the run mode parameter[3]. Each HTML template has some parent run modes and can lead the application to other templates depending on the option the user selects. The different templates and a brief description of their functionality can be found in appendix A.

6.6 Security

In order to achieve maximum security in the application developed, we have followed good web programming practices and tried to eliminate potential security violating situations, especially error situations. However, being an application that is going to be publicly available on the Internet, we must be extremely careful. Therefore, the information received

[3]Typically when the user changes the information on the browser's navigation bar when GET requests are sent

from the browser has to be thoroughly checked before it can be written onto the file since this information can, on a general basis, never be trusted.

Whenever an application writes or reads from a server there is a potential security problem. Malicious code may be stored or read from the server files causing unpredictable damage. It is necessary to check what is being written and read from the server files, and where it is trying to be written and read from. If the user can write unexpected code into an existing or non-existing file, there is a security problem in our application.

What we are always writing to the files are pairs of parameters and values such as *Java=Enabled* for example.

6.6.1 Accessing the files

Before the file is written on, an initializing subroutine is run. This creates the file with a 19 character long identification code. After the file has been initialized, it can be written on. In order to update the user file with new data we need to have an auxiliary file that holds temporary information. This file is also initialized in the same way as the user file and has the same identification code.

Whenever the application needs to write on a file, it sends the user file, the auxiliary file and the data to the writing subroutine as parameters. The writing subroutine verifies the following checkpoints:

1. Check that both the user file and the auxiliary file have the same identification code.
2. Check that the code is 19 characters long
3. Check that the characters belong to the correct set of characters (i.e. capital and small letters, numbers, @ and -)
4. Check that both the files already exist before opening them

If all these conditions are matched, the application writes onto the file. With this security measures, we are trying to avoid new files being created by users.

6.6.2 Values of the parameters

The file where we keep the information relative to the session is what has to be protected. On it we write the path that the user is following plus the various parameters and values that the application is testing for.

The information that is written to the file has to be checked so that the parameters only take the possible values they are supposed to take. In this sense the different parameters that the application expects to receive are packed in groups. Different groups of parameters will only accept certain values. For example, a parameter such as *Javacript* can only accept the values *enabled* or *disabled*. If any other value is tried to be given to this parameter the application does not write to the file and the error page is shown to the user

Some parameters have values that cannot be put into a set. The user-agent string for example is something that does not have a range of values that we can expect, therefore in this case, the application will not check the content of the user-agent string. Similarly, the dates and times of the run modes being accessed is something which is not controlled either and do not have a set of valid values.

The Java version number is not checked either. It may seem logical to check the value of this parameter since this one is shown in the browser's navigation bar and can be easily changed by the user. Besides, the Java version should be made up of numbers so it would be easy to check and parse. But, being a value that is not assigned by us (it is directly obtained from an applet), it is difficult to know what to expect. Our Java version already had an underscore on its version number, and other releases may end up having letters or symbols, so this did not seem so obvious in the end.

The Java version value was decided not to be checked at all since another security measure was being taken. The Java version is obtained through an applet (it cannot be obtained other that with a Java function). This applet redirects the application to the *gwelcome* run mode and in the redirection process, it has the version number attached to the query string as the value of the *jver* parameter. Before the HTML file corresponding to the *gwelcome* run mode is created and shown to the user, the version number is read and written on the file. It comes directly from the applet, it cannot be changed before that. Once it is written on the file, no other value is accepted. It remains unaltered.

Another parameter that does not have a set of valid values is the *Forbidden runmode* parameter. This is written onto the file whenever a non-valid run mode is tried to be accessed, so we don't know beforehand which run modes will be made up by the user who may try to find security holes in the application.

6.6.3 Parameters

Not only the values of the parameters that are written on the files have to be controlled. The parameters themselves have to be checked. If a parameter that is not expected is tried to be written, the application will detect it and will not be written on the file.

If the files that are tried to be accessed exist beforehand and the parameters and their values are valid, then, all the security locks are opened and the file is written on.

Chapter 7

Problems Encountered During the Testing

7.1 Applets, Virtual Machines and Plug-ins

The testing application was developed under Linux and basically tested on Mozilla 1.0.1 (later upgraded to 1.0.2) and Netscape 4.8 browsers. For the full testing of the application, Windows and Apple machines were also used. It was desirable to test the application on the browser which most of the users were using and make sure it could run.

When tested on the Windows platform, it was found out that the applets would not run on MSIE. The problem went on even further; they did not run on NS 4.76 either nor on Opera 7.11 on a Windows ME (Windows Millenium) machine. The only browsers that understood the applets were Mozilla 1.4 and Netscape 7.01. The problem was found to be a virtual machine problem.

The documents created for the application use the <APPLET> tag to specify the information about the applet and the <PARAM> tag to store additional run time information. These tags are read by the browser which then attempts to load the applet. If no further information is supplied, it is the browser's virtual machine (if it indeed has one available) the one which attempts to load the applet.

We have already noted the fact that JVM are not the same. Microsoft and Netscape developed their own JVM's and since they do not exactly behave the same way as Sun's, some incompatibilities appear. Neither of the JVM from Microsoft or Netscape were able to load any of the applets (Netscape 7.01 and Mozilla 1.4 worked fine though) and this was because they used their own JVM and not Sun's. The applets used in the application are kept as simple as possible since the testing for Java on the client machine must start from Sun's first Java releases. This fact shows that it was not that the applets used were too sophisticated for the non-Sun JVM to run them, but that it was their incompatibility that made them fail.

The application wouldn't run on more than 90%[1] of all the browsers accessing the web due to differences between Microsoft, Netscape and Sun's virtual machines even if the most simple applets were being used.

Fortunately a workaround for this problem was found. The idea was to force all the browsers to use the same JVM and have the applets run on it smoothly. The common denominator was Sun's Java virtual machine which we thought all the browsers could launch. The solution was to tell the browsers not to use their JVM but use Sun's instead. This would assure that all the browsers treated Java applets in the same way and the probability of having errors during the loading would be therefore diminished. If Sun's plug-in was not installed on the system, the user would be prompted to do so.

The way to tell the browsers not to use their own JVM and use Sun's plug-in instead is through the use of new HTML tags, but here a new problem arose; there were specific tags for MSIE and NS, but what about the rest of the browsers? and are those tags W3C HTML compliant?

Microsoft's Internet Explorer

For MSIE the tag that launches Sun's plug-in is the <OBJECT> tag[2]. If the applet is loaded using the <APPLET> tag, MSIE will use its own virtual machine.

Inside the <OBJECT> tag you had to specify what kind of object you were trying to execute, and which program should try to to so. The *classid* parameter in the <OBJECT> tag is used for this purpose and it uniquely identifies the Java plug-in. It should be used on every HTML file that launches an applet. When MSIE finds this tag on the web page, it will try to load the Java plug-in into the browser. The <OBJECT> tag allows information to be sent to the applet at run time in a similar way that the <PARAM> tags do to the <APPLET> tags, so the language support could still be available with this new format.

[1]See chapter 2: World Wide Web Browsers

[2]This new tag will take over the <APPLET> tag which has been deprecated in the HTML 4.0 standard in deference to the <OBJECT> tag which can do the same as the original <APPLET> tag but serves a much more general purpose.

When the original <APPLET> tags were converted to the new <OBJECT> tags, MSIE 6.0 on a Windows Millenium machine executed the testing application successfully.

Netscape Communication's Navigator

Netscape 7.01 could load the applets using the <APPLET> tag, but version 4.76 of Netscape's Communicator couldn't[3]. On the Linux machine, Communicator 4.8 also worked fine with the <APPLET> tags. It seemed version 4.76 was the problem. Netscape's Release Notes for Communicator 4.76 [Com00b] didn't show anything revealing, but looking at the Release Notes for 4.8 [Com02], we found this information:

Java Plug-in Support. A new Java plug-in preference option allows users to take advantage of the latest Sun Java plug-in (available for free at http://java.sun.com). When enabled, the Sun Java plug-in delivers an improved user experience for Java-enabled sites and completely replaces Netscape Java.

Netscape Communicator 4.8 is capable of using Sun's plug-in instead than using its own JVM if selected on the browser. On the same section on Communicator's 4.76 Release Notes [Com00b] nothing was stated in this sense. That explained why the <APPLET> tags worked on the 4.8 version and they didn't on the 4.76 version, regardless of the machine they were being executed on. Netscape Communicator 4.8 could use Sun's plug-in and not Netscape's virtual machine if selected on the user settings, but version 4.6 was forced to use Netscape's virtual machine because it didn't have any plug-in support. In fact, it was found out that the plug-in from Sun could be used starting from version 4.78 of Netscape Communicator, as stated on its release notes [Com01b][4].

To get the applets running on Communicator 4.76 we must use a new tag (the same way we used the <OBJECT> tag with MSIE). This new tag is <EMBED>. A parameter called *attribute type* in the <EMBED> tag is used to identify the type of the Java executable (Java Bean or applet). When the browser finds this attribute in the <EMBED> tag it will try to load the Java plug-in into the browser. Netscape Communicator 4.76 was able to successfully load the applets after the <EMBED> tags had been correctly set up.

[3]Note the different names of the browsers; the earlier versions are called Netscape Communicator and the newer ones (starting from 6.0) are just Netscape

[4]Version 4.77 Release Notes [Com01a], do not say anything about the capability of using Sun's plug-in, so the fi rst version to use the plug-in is 4.78.

Other Browsers

The problem now was to care about the rest of the browsers. We were able to have both <OBJECT> and <EMBED> tags in one same HTML file and hide the code concerning MSIE from NS and vice versa by using tags that only one or the other browser would understand, but what if another browser took the tests? Even though the browser market share showed that it was only 4% of all the browsers [One03], it was considered to be important enough not to discriminate less popular browsers. If a browser that wasn't MSIE or NS executed the testing application it would find new tags that it may not undestand, so this was a problem we had to deal with.

One of the possible solutions was to identify the browser through JavaScript and then decide which parts of the HTML code were to be shown to it. Should the browser not be MSIE or NS, then the <APPLET> tag should be shown. The drawbacks to this approach were:

1. JavaScript had to be enabled on the browser.
2. We had to use non W3C conformant HTML (we have to use tags that only MSIE understands but not NS and vice versa).
3. We had to rely on a proper identification of the browser. The identification of the browser had to be done through the user-agent string, with all the inconveniences mentioned in the previous chapters.

Another possibility was to have different HTML files for the different browsers, and ship the corresponding one to the client depending on the browser detected. This second option did not avoid any of the inconveniences found before, and meant three more files per applet page in our filesystem.

At first, the first solution was adopted being the one which guaranteed the broadest browser coverage and simplicity. It would ensure that browsers that do not support Java plug-in or browsers that do not support JavaScript could handle the applet using their own default Java virtual machine.

Sun's plug-in 1.4.1 and above supports the <APPLET> tag for launching applets[5]. That

[5]On MSIE 4.0 or higher and Netscape 6.0 or higher on Windows 95, 98, ME, NT 4.0, 2000, XP, UNIX and Linux.

is, if specified on the browser that Sun's plug-in is to be used, when the browser reaches the <APPLET> tag, the plug-in, (and not the own browser's JVM) will be loaded and the applet will be run on it without the need of <OBJECT> or <EMBED> tags. Since we cannot rely on the user having one of Sun's latest plug-ins, we have to stick with all three of the tags on the same document: <APPLET>, <OBJECT> and <EMBED>.

7.1.1 Plug-in parameters

When launching an application through the <OBJECT> tag, the actual *object* that is being sent has to be specified by some kind of code. Reading this code, the browser understands that it has to try to load a certain application that will know how to display the object that is being sent to it. In order to launch Sun's Java plug-in, we need to specify its particular code.

There are two codes that load Sun's plug-in on the browser if using the <OBJECT> tag; one specifies *dynamic* versioning and the other one specifies *static* versioning. The versioning refers to the Java version with which the applet was compiled. If the code specifies dynamic versioning, the applet will look for for a plug-in that has the same or newer version than the Java release that compiled it. If the plug-in version is not the same or newer, then the user is prompted to download an updated version of it. In static versioning, the plug-in that is required by the applet is exactly the same one as the one which complied it, and the user will be prompted to download it, even though he or she may have a newer version of that same plug-in.

Our applets were coded with Java 2 Runtime Environment, Standard Edition (build 1.4.1_01-b01). Java supplies a tool called *HTMLConverter* that converts the <APPLET> tags to <OBJECT> and <EMBED> tags automatically. This tool directly created an HTML file that stated that the user should have this same Java version to load the applets (static versioning). If we used dynamic versioning, the applet would work only on machines with this plug-in or newer, and if we used the static one, they would only work with the same exact plug-in. Either ways it seemed the used had to probably update his system in order so see the applets. This was against the usability of the testing application. If the user had to download, install and probably reboot the system in order to run our application, many of them would probably not do so.

Since the applets played were very simple and used Java features that were available from version 1.1, it was decided to modify the automatically created HTML code so that it showed that the version which compiled the applet was 1.1. If we then used dynamic versioning, our applets would work on any plug-in that was version 1.1 or above (that is, on any plug-in). If the user happened not to have any Java plug-in installed, then he or she was prompted, not to download version 1.1 which would be obsolete, but the most recent version available from Sun's website[6]. This guaranteed that the minimum number of downloads had to be made, and only the users who did not have any plug-in at all were affected by the downloading.

7.1.2 Deprecated Tags and New Plug-ins

The <APPLET> tag has been deprecated in the HTML 4.0 standard [W3C98] in favour of the <OBJECT> tag[7]. The <APPLET> tag has certain limitations; it is unable to include new and future media types and can only be used to play Java applets. On the other hand the <OBJECT> tag is an all-purpose solution to generic *object* inclusion in web pages. It can handle any *object* that developers want to place in HTML files, ranging from images to any type of (not just Java) applets.

Even though it is deprecated, starting from Java 1.4, the plug-in supports the <APPLET> tag for launching applets so that other tags like <OBJECT> and <EMBED> need not to be used. This is confusing. New technology starts supporting a deprecated item and does not encourage developers to use its non deprecated alternative. This just shows how difficult it is for everyone to meet the standards.

7.1.3 Solutions that Work

It was strange to find out that on Sun's Java site, tutorials on writing applets and making them available on the Web used the <APPLET> tag and did not make emphasis on the fact that the most popular browsers needed another tag to work correctly. It was also strange to see how not even the most basic applets would load when using Microsoft's or Netscape's own virtual machines. Could the implementations be so different that not even the simplest of applets could be run without the aid of Sun's plug-in?

After testing different solutions (including the browser detection, which we did not like)

[6]The version the applet was compiled with and the Java version to download (if necessary) are different parameters, so they can have different values

[7]W3C defi nition of deprecated: *Deprecated: A deprecated element or attribute is one that has been outdated by newer constructs. Deprecated elements are defi ned in the reference manual in appropriate locations, but are clearly marked as deprecated. Deprecated elements may become obsolete in future versions of HTML.*

that combined different HTML tags in the same document, it was found out that some combinations suited some browsers but did not help others[8] loading the applets. Table 7.1 and table 7.2 were created in this sense, and they show the different incompatibilities that were found both on a Linux and a Windows machine.

	Linux Platform		
	Netscape 4.8		Mozilla 1.0.1
	JVM 1.1.5	Plug-in 1.4.1	Plug-in 1.4.1
<APPLET>			
Redirection	Error:	OK	OK
AWT	bad	OK	OK
Swing	major	OK	OK
Button	version	OK	OK
Sun's *.au* audio	number	OK	OK
<OBJECT> & <EMBED>			
Redirection	No redirection	No redirection	OK
AWT	OK	OK	OK
Swing	OK	OK	OK
Button	OK	OK	OK
Sun's *.au* audio	No sound	No sound	OK
<ALL PLATFORMS>			
Redirection	Error:	OK	OK
AWT	bad	OK	OK
Swing	major	OK	OK
Button	version	OK	OK
Sun's *.au* audio	number	OK	OK

Table 7.1: Testing different HTML tags to launch Java applets on different browsers on a Linux platform.

Research through Sun's Java site concerning bugs was made. One of the biggest problems encountered was the loading of new pages using Java's *showDocument()* method regardless of the browser used. In our testing we found out that, on some occasions, all the other applets would work, except for the one which used *showDocument()* to redirect the user to another part of the testing application. If the redirection applet worked, all the other applets worked. This was definitely the most critical part of the application. Sun argued this was not a plug-in bug, but that it was due to third party software that somehow interacted in the

[8]Not to mention the fact that the number of browsers and platforms tested was not at all exhaustive and probably the more we tested the bigger the incompatibilities we were bound to fi nd

	Windows ME				
	Explorer 6.0	Netscape 7.01	Opera 7.11	Netscape 4.76	Mozilla 1.4
	JVM 1.1.4	Plug-in 1.3.1	Embedded	JVM 1.1.5	Plug-in 1.3.1
<APPLET>					
Redirection	Error:	OK	Crash	Error:	OK
AWT	Class	OK	OK	bad	OK
Swing	not	OK	OK	major	OK
Button	found	OK	No redir	version	OK
Sun's *.au* audio	error	OK	OK	number	OK
<OBJECT>					
& <EMBED>					
Redirection	OK	OK	Crash	No redir	OK
AWT	OK	OK	OK	OK	OK
Swing	OK	OK	OK	OK	OK
Button	OK	OK	No redir	OK	OK
Sun's *.au* audio	OK	OK	OK	No sound	OK
<ALL					
PLATFORMS>					
Redirection	Done	OK	Crash	Error:	OK
AWT	but	OK	OK	bad	OK
Swing	with	OK	OK	major	OK
Button	errors	OK	No redir	version	OK
Sun's *.au* audio	on page	OK	OK	number	OK

Table 7.2: Testing different HTML tags to launch Java applets on different browsers on a Windows platform.

loading of the new page [Micb]. No clear answers were given in this sense and users which had encountered this problem ended up upgrading their browsers, so the issue was never cleared.

The feeling we had was that there wasn't a clear and simple solution that would work for the vast majority of browsers. Developing web pages with lots of *if* and *else* conditions, depending on the browser, seemed the only solution, but we wanted our application to run on as many browsers as possible, and if this was necessary it would have been done so.

The final solution was found while looking for some information on the *showDocument()* method. The example page found used *showDocument()* and called the applet through the <APPLET> tag. It was strange to find out that that example applet worked on every browser, both on the Windows and the Linux machines (except for the Opera browsers) when our applet did not.

The applets had been coded with one of the first releases of Java and they worked all the way. All the applets we had developed had been compiled into class files using Java 2 Platform Standard Edition (J2SE v 1.4.1_01). Maybe that was the reason why they didn't work. We downloaded and compiled our applets with Java Development Kit (JDK), v 1.0.2 on a Windows machine, since there was no Linux version in the early stages of the Java language, and found out they worked perfectly well with the original <APPLET> tag (no <OBJECT> or <EMBED> tags needed) on all of the browsers[9]. During compilation time, no *deprecated* warnings were received, because of course, the original methods were not deprecated on the first Java releases.

Obviously when we tried to compile the Swing applet with JDK 1.0.2 it did not work. We downloaded the next Java release available 1.1.6_009 and compiled all the applets with it. Some features were already deprecated! The Swing applet did still not compile because no JFC were downloaded. When we downloaded the first Swing components, we even had to use the name of the original package for the Swing components. We had to change *javax.swing.** for *com.sun.java.swing.** because that was the original name, and the compiler did not understand *javax.swing.**. This was found out when we referred to the API.

The applets that tested for Swing had to be compiled with a newer version of the Java releases, since the Swing technology was not available at first. We managed to compile the applet but we were unable to make it work. It was because of the changing of the name of the package from *com.sun.java.swing.** to *javax.swing.**. The new plug-ins did not understand that package and so, didn't display the applet.

We finally compiled the Swing applet to work with Java plug-in 1.1.1 or above[10] and it worked on all the browsers except for Netscape 4.6 because its JVM does not support Swing.

By compiling all our applets with the first Java release available for download, we managed to keep a simple structure in our web pages while it guaranteed the broadest possible browser coverage we could afford. The Opera browser was still not loading the redirection applet.

[9]The redirection applet still did not work on the Opera browsers.

[10]See Chapter 3: Java

Chapter 8

Conclusions and Future Work

8.1 Testing phase results and conclusions

Before making the application available for the Internet community, we evaluated its behaviour with a small number of users. These users were the people at the Acoustics and Audio Signal Processing Laboratory at the Helsinki University of Technology. For this testing phase, all the applets were compiled with Java 1.0.2, except for the AWT applet which was deliberately compiled with version 1.4.1_01 and, of course, the Swing applet which needed a newer version since its methods were not supported by the first Java releases. The AWT applet was compiled in this way to see how the browsers coped with its loading.

We sorted the test results according to Browser, Platform and JVM used. The results can be seen in tables 8.1 and 8.2. Some behaviour patterns were easily spotted:

8.1.1 Browsers

Opera

Opera does not have a proper Java support, neither for Windows, Linux, nor Apple machines. Opera has Java embedded into the browser code, but results are not the expected. Five users tested the application with this browser. Not even one of them could hear all the sound samples, in fact Java applets only loaded once in all the five tests, and even then, the music samples were not to be heard. The platforms under which Opera was tested were Windows, Linux and Apple Macintosh, so all major platforms were covered; the plug-in we were not able to find since only once did the redirection applet work, providing us with the Java platform version. Five samples is not really enough to make any definite statement about the Java support the Opera browser presents, but it gives an idea.

Netscape

The other browsers seemed to depend more upon the plug-in they were using. Old versions (4.x) of Netscape still use Netscape's virtual machine if Sun Microsystem's plug-in is not selected through the user settings or if is not installed at all. This caused that some of the testers could only listen to the basic .au format (8 bit, μ law, 8000 Hz, mono) that Sun made available with its first Java platform release. But this was not true always; some testers could not load any of the sound samples at all despite they had some kind of Java support working on their browsers.

Mozilla

Mozilla was the browser that most coherently behaved throughout the testing since it does not have a built in Java virtual machine nor has Java embedded into the browser. In the cases that there was no Java plug-in installed, the application did not run at all. If a plug-in was there, all the Mozilla testers, could listen to the samples. The plug-ins that were tested were versions 1.3.1 or above.

Internet Explorer

Explorer's 6.0 version was tested by nine users; seven of them were using a Windows platform and two of them had an Apple platform. This browser has a built in Java virtual machine only if an additional service pack provided by Microsoft was downloaded. It was odd to see that this browser, on a Windows platform, would stick to its own virtual machine except when plug-in version 1.4.1_01 or above was installed. With its own virtual machine (version 1.1.4), only the basic .au file was heard, and the applets compiled with Java platforms 1.2 (Swing applet) or 1.4.1_01 (AWT applet) did not work at all. Only the applets compiled with version 1.0.2 of the Java platform could be loaded with Microsoft's Java virtual machine. The browser did not aim to use the plug-in from Sun at all and so missed all the sound samples except for the basic one.

The Explorer browsers on an Apple machine that were tested were versions 5.22 and 5.23. On this platform the browser used the 1.3.1 Java plug-in implementation for Apple's Mac OS X and all the samples could be heard. It was odd to see how the AWT applet loaded while the Swing applet didn't even if the AWT applet was compiled with a newer

Java version than the Swing one. Up to now, only Apple's Safari browser can run Java applets under J2SE 1.4.1 if 1.4.1 is installed. All the other browsers are currently developed to specifically use Apple's 1.3.1 implementation.

Safari

Apple's Safari browser was tested with the latest Java plug-in released for this platform (version 1.4.1_01) and worked all the way. Only three users took this test.

8.1.2 Platform

There does not seem to be a clear relationship between the platform selected and the Java applet performance of the testing application. Only on the Apple platform there seems to be some kind of pattern, but is is only due to the fact that all the testers were using Apple's Mac OS X which was originally shipped with Apple's 1.3.1 version of the Java plug-in and therefore all the systems have it installed. Besides, plug-in version 1.3.1 is expected to be able to play all the samples. This version of the plug-in worked well on a general basis and the sound samples could be heard. Unluckily our testers did not have older versions of this platform available, so performance there could not be tested. It is true to say that all the OS X users could load and hear all the samples except for the Opera browser users; this browser behaved in a strange manner.

The Windows and Linux platforms are more dependent on the plug-in used by the browser. They do not show a clear pattern since they both have unsuccessful test results in approximately half of the samples collected.

8.1.3 Plug-In

This is where the most clear pattern is found: if the plug-in is 1.3.1 or above, then the samples can be heard. If the plug-in is below 1.3.1, then probably an own Java virtual machine is being used, and both Microsoft and Netscape's coped really badly with the applets and the sound samples. The Java version in this case was shown to be 1.1.5 for Netscape (versions 4.8 and below) and 1.1.4 for Explorer (version 6.0).

There were some exceptions to this rule. The latest Opera browser (7.11) running on a Windows machine claimed to be using plug-in 1.4.1_01; the speech samples could be heard, but not the music. Netscape 6.2.1 on a Windows machine was unable to play all the samples but the original *.au* sample with a 1.3.1 plug-in (only one case in both examples). The browsers using their own virtual machines, could at the most only play the basic *.au* sample, if they did at all.

8.1.4 Further considerations

Java 1.0.2

The problems encountered before this testing phase took place, clearly showed that in order to get the largest number of browsers to load the applets, these had to be compiled with the earliest version of the Java platform. This version was 1.0.2 which was available from Sun Microsystem's site. This result is clearly stressed if we look at the results from the testing phase.

On table 8.1 we can observe how on some occasions Sun's *.au* applet loads when the AWT applet has not loaded. The graphical interface that the *.au* applet has is entirely made out of AWT components. If a basic AWT applet does not load, the following applets that use these components should not load either.

The AWT applet does not load because it has been compiled with version 1.4.1_01 of the Java platform. The *.au* applet loads because it has been compiled with version 1.0.2. Should the AWT have been compiled with the first Java version, it would have loaded. The compiling was done this way deliberately so that this behaviour could be seen. On the final version of the application all the applets, except for the Swing applet, have been compiled with version 1.0.2 of the Java platform.

MIDI samples and Opera browser

Special attention had to be payed to the MIDI sample and the Opera browser. It was odd to find out that on more than one machine, all the samples but the MIDI file could be played. The Java Sound package is able to play WAV, AIFF, AU and MIDI samples and it is shipped with a sound synthesizer except on the version for the Windows platform. By playing all the other samples we can be sure that Java Sound is working correctly, but the MIDI sample somehow was not heard. The reason for this behaviour showed up to be that another application had somehow taken all the priviledges over MIDI files and would not let any other application play them.

The Opera browser turned out to be the least Java compatible browser of the whole set. Only once, out of the five testers who tried it, were we able to even load the first redirection applet and obtain the Java version being used. In all the other cases, either Java was not enabled, or what is worse, it was enabled but could not even load the first applet which all others could.

Survey: Usability

Apart from the technical conclusions, a survey about the application was also carried out. We were looking for feedback about usability issues, usefulness of the application and problems encountered in general. The basic problems that we spotted out were the following:

Too much text to read: Sometimes the user felt that there was too much to read despite the box in the middle of the application page held all the information in three or four lines. The impression we got is that users just wanted to know if their Java platform was up to date, and did not care much whether they had just loaded a Swing applet or they had listened to a 16 bit sample, so the explanatory text at some times felt unnecessary, they just wanted to click onto the next page.

The order of the application seemed logical and the navigation easy: These were usability issues that we had been really concerned about. We wanted a really simple user interface and navigation scheme plus a consistent look and feel throughout the application. Users were quite satisfied with this issue.

Testers of the application seemed to be lost if something went wrong and easily abandoned when facing some problems. Only once out of twenty testing results did the user finish all the testing despite he could not hear the samples at all. The impression we got was that either the testers gave up and thought that their systems did not have the Java support and abandoned or that the helping pages were not clear enough. Only one user upgraded his system when he couldn’t hear the samples, the others just gave up or didn’t know what to do.

The tests helped us understand the importance of clear and simple help pages. When the application tells a user that there is something that is not working in her or his machine, the last thing the user expects is to find a useless help page. At this point users do read all the text displayed on their screens. The help pages that were available during the testing phase turned out not to be helping at all. We were forced to rewrite the text and change the structure of most of them to make them simpler and clearer. The pages with lots of text were divided into two (one having more detailed information) to make them easier to read.

Other issues

These tests were performed by the people at the Laboratory of Acoustics and Audio Signal Processing. In this scenario, not even one of them had to check if their loudspeakers were working correctly or if they were plugged into the correct connector of their sound card, so some parts of the application were not visited at all. The testers' level of expertise concerning computer usage and sound was considered high, and the equipment used to take the tests quite modern. An important fact to point out is that the network connection was exceptionally fast, and no delay in the downloading of the files was observed. This is something which will most probably cause problems to users outside the closed Laboratory conditions.

This is a very close scenario indeed: user expertise is high, equipment is modern and network connection extremely fast. Nevertheless some conclusions can be obtained since it represents the profile of university students interested in audio and signal processing with high technology equipment. This was, at first, the potential users we were focusing on.

One of the tests revealed an interesting (and probably confusing) issue. Web browsers may have Java's checkbox marked as enabled on the user settings even if the actual platform is not installed at all. The application's JavaScript test does say that Java is not enabled (that works fine) even though the checkbox may be selected. If the application (correctly) detects that Java is not available, it will show the user some help pages on how to enable the Java platform. But then, the user will open the browsers preferences menu and find that the *Java Enabled* checkbox is already marked despite being told that Java is not enabled. This will confuse the user.

8.1.5 Final Conclusions

After analyzing the testing results and after working with the Java platform throughout the thesis, these were my final conclusions:

Java applets are a mature technology, but unfortunately they cannot be completely trusted to work as expected in practice.

When the browser starts to load the JVM for the first time, there is a considerable delay before the applets can be used. This process slows down the loading of the web page.

The real problem is that companies have not always followed the standards. This is the reason why there are different virtual machines, different versions of them and different browsers interpreting the code in different ways.

Compared to Java applications, applets are restricted with their functionality on the client machine. Some of these restrictions are related to security, some of them to the interoperability between the browser and applets. However, other client-side technologies such as Javascript have similar or other limitations as well.

Macromedia's Flash technology was focused on presentations at first, but it has now concentrated on web applications and is a realistic competitor to Java applets. Similarly to the Java plug-in, Flash applications require an appropriate plug-in to be used together with the web browser. The applications created with Flash give more reliable playback rates between different machines and browsers than applets [Dra02] although Macromedia has also got version incompatibilities similar to those of Java applets.

2001 press release from Macromedia [Mac01] claims to have support on more than 97 & of the desktops connected to the Internet, so availability is guaranteed. This can also be noticed on the Web, where Flash applications seem to be more popular nowadays than Java applets.

The major drawback to Macromedia's technology is the fact that it is proprietary. Both the runtime tools (Flash plug-in) and authoring tools have been provided by Macromedia. However, the Flash format is XML-based and the specification partly open, which may lead to a wider selection of tools for Flash development.

Future of Java applets

The future of Java applets is uncertain. Netscape dropped the development of its virtual machine since Netscape version 4.8 and what is more important (because of the browser market state) Microsoft is dropping its support for the Java virtual machine in September 2004[1]. Microsoft is moving onto other technologies such as *.NET* while Sun is fighting for Microsoft's platforms to be distributed with their runtime environment. The situation is certainly uncertain. If the browser which has most of the market share does not support Java applets, this client side part of the whole Java programming language may become obsolete or outdated.

[1]Originally this dead line was set to January 2004

The mobile devices market may be the new scenario where Java applets can have an opportunity to become popular. If the different vendors follow the standards and are all using the same virtual machines, applications will be able to be loaded on them. Having the virtual machines embedded into the system (hardware or software wise) may speed up the loading of the applets and an overall improved performance will be achieved.

8.2 Future work

The application only tests for basic sound reproduction. Only four different audio formats are tested (*au*, *MIDI*, *wav* and *aiff*) and none of the most popular ones on the Net are on that list. Other sound packages for Java are prepared to be included for testing on the application. Testing for JSyn and the JMF was scheduled to be included on the application at first, but were left out due to timing limitations. With JSyn, the user can play sounds from oscillators, filters or noise generators among others. JMF, on the other hand, would allow for media streaming with Java, both audio and video. These and other Java packages could be tested for by simply adding new applets to the application and trying to load them on the accessing system.

Browser	Version	Platform	JVM	AWT	Swing	Sun's *.au*	WAV	AIFF	*.au* 16 bits	MIDI
Explorer	6.0	Windows	1.1.4	NO	NO	Loaded	NO	NO	NO	NO
Explorer	6.0	Windows NT/2000	1.1.4	NO	NO					
Explorer	6.0	Windows	1.1.4	NO	NO	Loaded	NO	NO	NO	NO
Netscape	4.76	Windows	1.1.5	NO	NO	Loaded	NO	NO	NO	NO
Netscape	4.8	Linux	1.1.5	NO	NO	NO	NO	NO	NO	NO
Netscape	4.8	Linux	1.1.5							
Mozilla	1.4	Windows	1.3.1	Loaded	Loaded	Loaded	Loaded	Loaded	Loaded	NO
Netscape	7.01	Windows	1.3.1	Loaded	Loaded	Loaded	Loaded	Loaded	Loaded	NO
Netscape	6.21	Windows NT/2000	1.3.1	Loaded	Loaded	Loaded				
Explorer	5.22	Macintosh	1.3.1	Loaded	NO	Loaded	Loaded	Loaded	Loaded	Loaded
Mozilla	1.4	Windows	1.3.1	Loaded	Loaded	NO				
Mozilla	1.4	Macintosh	1.3.1	Loaded	Loaded	Loaded	Loaded	Loaded	Loaded	Loaded
Mozilla	1.4	Macintosh	1.3.1	Loaded	Loaded	Loaded	Loaded	Loaded	Loaded	Loaded
Mozilla	1.4	Macintosh	1.3.1	Loaded	Loaded	Loaded	Loaded	Loaded	Loaded	Loaded
Explorer	5.23	Macintosh	1.3.1	Loaded	NO	Loaded	Loaded	Loaded	Loaded	Loaded

Table 8.1: Part 1 of the results obtained from the testing done by the people at the Laboratory of Acoustics and Audio Signal Processing. Table sorted by JVM (only JVM 1.1.4, 1.1.5 and 1.3.1 are shown on this table)

Browser	Version	Platform	JVM	AWT	Swing	Sun's *.au*	WAV	AIFF	*.au* 16 bits	MIDI
Explorer	6.0	Windows	1.4.1_01	Loaded	Loaded	Loaded	Loaded	Loaded	Loaded	Loaded
Opera	7.11	Windows	1.4.1_01	Loaded	Loaded	Loaded	NO	NO	NO	NO
Mozilla	1.3.1	Windows	1.4.1_01	Loaded	Loaded	Loaded	Loaded	Loaded	Loaded	Loaded
Netscape	4.8	Linux	1.4.1_01	Loaded	Loaded	Loaded	Loaded	Loaded	Loaded	Loaded
Konqueror	3.0	Linux	1.4.1_01	Loaded	Loaded	NO	NO	NO	NO	NO
Galeon	1.2.11	Linux	1.4.1_01	Loaded	Loaded	Loaded	Loaded	Loaded	Loaded	Loaded
Mozilla	1.0.2	Linux	1.4.1_01	Loaded	Loaded	Loaded	Loaded	Loaded	Loaded	Loaded
Explorer	6.0	Windows NT/2000	1.4.1_01	Loaded	Loaded	Loaded	Loaded	Loaded	Loaded	Loaded
Safari	1.0	Macintosh	1.4.1_01	Loaded	Loaded	Loaded	Loaded	Loaded	Loaded	Loaded
Mozilla	1.3.1	Linux	1.4.1_01	Loaded	Loaded	Loaded	Loaded	Loaded	Loaded	Loaded
Safari	1.0	Macintosh	1.4.1_01	Loaded	Loaded	Loaded	Loaded	Loaded	Loaded	Loaded
Safari	1.0	Macintosh	1.4.1_01	Loaded	Loaded	Loaded	Loaded	Loaded	Loaded	Loaded
Safari	1.0	Macintosh	1.4.1_01	Loaded	Loaded	Loaded	Loaded	Loaded	Loaded	Loaded
Explorer	5.01	Windows 98	1.4.1_02	Loaded	Loaded	Loaded	Loaded	Loaded	Loaded	Loaded
Mozilla	1.3	Linux	1.4.1_02	Loaded	Loaded	Loaded	Loaded	Loaded	Loaded	Loaded

Table 8.2: Part 2 of the results obtained from the testing done by the people at the Laboratory of Acoustics and Audio Signal Processing. Table sorted by JVM (only JVM 1.4.1_01 and 1.4.1_02 are shown on this table)

Bibliography

[Aus03] Calvin Austin. A Roadmap for Java 2 Platform, Standard Edition (j2se) 1.4.2 and 1.5, June 2003. http://developer.java.sun.com/developer/technicalArticles/RoadMaps/J2SE_1.5/j2se_1_5.html.

[BAAG99] Jesús Bobadilla, Alejandro Alcocer, Santiago Alonso, and Abraham Gutiérrez. *HTML Dinámico, ASP y Javascript a través de ejemplos*, volume 1. Ra-Ma, 1 edition, 1999.

[Bec03] David Becker. Is this the end of Netscape?, May 2003. http://news.com.com/2100-1032_3-1011356.html.

[BLC95] T. Berners-Lee and D. Connolly. Hypertext Markup Language - 2.0. RFC 1866, November 1995. http://www.rfc-editor.org/rfc/rfc1866.txt.

[BLFF96] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext Transfer Protocol – http/1.0. RFC 1945, May 1996. ftp://ftp.rfc-editor.org/in-notes/rfc1945.txt.

[Cla02] Bob Clary. Browser Detection and Cross Browser Support. Technical report, July 2002. http://devedge.netscape.com/viewsource/2002/browser-detection/ (revised 10 feb 2003).

[Com00a] Douglas E. Comer. *Internetworking with TCP/IP Principles, Protocols and Architectures*, volume 1. Prentice Hall, 4 edition, 2000.

[Com00b] Netscape Communications. Communicator 4.76 Release Notes, November 2000. http://wp.netscape.com/eng/mozilla/4.7/relnotes/windows-4.76.html#java.

[Com01a] Netscape Communications. Communicator 4.77 Release Notes, March 2001. http://wp.netscape.com/eng/mozilla/4.7/relnotes/windows-4.77.html#java.

[Com01b] Netscape Communications. Communicator 4.78 Release Notes, August 2001. http://wp.netscape.com/eng/mozilla/4.7/relnotes/windows-4.78.html#java.

[Com02] Netscape Communications. Communicator 4.8 Release Notes, August 2002. http://wp.netscape.com/eng/mozilla/4.8/relnotes/windows-4.8.html#java.

[cyS] Inc. cyScape. Browserhawk. http://www.cyscape.com/browscap/.

[Dal03] Jim Dalrymple. Microsoft drops development of Internet Explorer for Mac, June 2003. http://maccentral.macworld.com/news/2003/06/13/explorer/.

[Dev] Netscape Communications Devedge. Netscape Gecko Central. http://devedge.netscape.com/central/gecko/.

[Dra02] Max Drayman. Java vs Flash, September 2002. http://www.winneronline.com/articles/september2002/javavsflash.htm.

[FGM$^+$99] R. Fielding, J. Gettys, J. Mogul, H.Frystyk, L. Masinter, and T. Berners-Lee. Hypertext Transfer Protocol – http/1.1. RFC 2616, June 1999. ftp://ftp.rfc-editor.org/in-notes/rfc2616.txt.

[fSA97] National Center for Supercomputing Applications. NCSA Mosaic Home Page, January 1997. http://archive.ncsa.uiuc.edu/SDG/Software/Mosaic/NCSAMosaicHome.html.

[Har00] Elliotte Rusty Harold. *Java Network Programming*, volume 1. O'Reilly, 2 edition, 2000.

[Har02a] Eliotte Rusty Harold. February 2002 Java News, February 2002. http://www.cafeaulait.org/2002february.html.

[Har02b] Eliotte Rusty Harold. September 2002 Java News, September 2002. http://www.cafeaulait.org/2002september.html.

[Har03] Eliotte Rusty Harold. March 2003 Java News, March 2003. http://www.cafeaulait.org/2003march.html.

[JC98] Sandeep Junnarkar and Tim Clark. AOL buys Netscape for $4.2 billion, November 1998. http://news.com.com/2100-1023-218360.html.

[KM97] D. Kristol and L. Montulli. HTTP State Management Mechanism. RFC 2109, February 1997. ftp://ftp.rfc-editor.org/in-notes/rfc2109.txt.

[Lan03] Sheri R. Lanza. AOL to Drop Netscape, August 2003. http://www.infotoday.com/newsbreaks/nb030804-1.shtml.

[Leg99] David Legard. Zona declares Microsoft winner in browser war, November 1999. http://www.cnn.com/TECH/computing/9911/10/microsoft.wins.idg/.

[Mac01] Inc. Macromedia. Industry Leaders Support Macromedia Flash Player on Windows-CE Based Devices, September 2001. 2001 Press Releases: http://www.macromedia.com/macromedia/proom/pr/2001/fp5_support.html.

[Mica] Sun Microsystems. About the JFC and Swing. http://java.sun.com/docs/books/tutorial/uiswing/start/swingIntro.html.

[Micb] Sun Microsystems. Bug id 4296836. http://java.sun.com/developer/bugParade/bugs/4296836.html.

[Micc] Sun Microsystems. Frequently Asked Questions - Java Security. http://java.sun.com/sfaq/#prevent.

[Micd] Sun Microsystems. Java 2 Security Architecture. http://java.sun.com/sfaq/j2se/1.4.2/docs/guide/security/spec/security-spec.doc1.html#18313.

[Mice] Sun Microsystems. Java Media Framework API. http://java.sun.com/products/java-media/jmf.

[Micf] Sun Microsystems. Java Servlet Technology. http://java.sun.com/products/servlet.

[Micg] Sun Microsystems. Java Upgrade Guide: Migrating From the Microsoft VM for Java to the Sun JRE. `http://java.sun.com/j2se/1.4.2/docs/guide/deployment/deployment-guide/upgrade-guide/`.

[Mich] Sun Microsystems. Javabeans FAQ General Questions. `http://java.sun.com/products/javabeans/faq/faq.general.html#Q2`.

[Mici] Sun Microsystems. JMFRegistry User Guide. `http://java.sun.com/products/java-media/jmf/2.1.1/jmfregistry/jmfregistry.html`.

[Mic98] Sun Microsystems. Default Policy Implementation and Policy File Syntax, October 1998. `http://java.sun.com/products/jdk/1.2/docs/guide/security/PolicyFiles.html`.

[Mic00] Sun Microsystems. The Java Platform: Five Years in Review, 2000. `http://java.sun.com/features/2000/06/time-line.html`.

[Mic02a] Sun Microsystems. javax.sound.midi Interface Soundbank, 2002. `http://java.sun.com/j2se/1.4.1/docs/api/javax/sound/midi/Soundbank.html`.

[Mic02b] Sun Microsystems. Overview–What Is Java Plug-in? What Does It Support?, 2002. Developer Guide: `http://java.sun.com/j2se/1.4.1/docs/guide/plugin/developer_guide/overview.html`.

[Mic03] Sun Microsystems. Supported Media Formats and Capture Devices, May 2003. `http://java.sun.com/products/java-media/jmf/2.1.1/formats.html`.

[Mon02] Dan Moniz. Java 1.4.0 released, February 2002. `http://wmf.editthispage.com/discuss/msgReader$7032?mode=day`.

[Moza] Mozilla.org. Mozilla.org. `http://www.mozilla.org`.

[Mozb] Mozilla.org. Mozilla.org. `http://bugzilla.mozilla.org/show_bug.cgi?id=97613`.

[Ngu01] Hung Q. Nguyen. *Testing Applications on the Web*, volume 1. Wiley computer publishing, 1 edition, 2001.

[Nie95] Jakob Nielsen. Features for the Next Generation of Web Browsers, July 1995. `http://www.useit.com/alertbox/9507.html`.

[Nie00] Jakob Nielsen. *Designing Web Usability*, volume 1. New Riders Publishing, 1 edition, 2000.

[Nyk99] Sebastian Nykopp. A Java-based Presentation System for Synchronized Multimedia. Master's thesis, Helsinki University of Technology, August 1999. Available from `http://www.acoustics.hut.fi/publications/files/theses/nykopp_mst.pdf`.

[One03] Onestat.com. Microsoft's IE 6 global usage share continues to rise according to OneStat.com, July 2003. `http://www.onestat.com/html/aboutus_pressbox23.html`.

[Org03] The Mozilla Organization. RIP Netscape, July 2003. `http://www.mozilla.org/status/2003-07-18.html`.

[qin] qindex.info. Qindex.info, Developer's Quick Index. `http://www.qindex.info/home/miscellaneous_tips/browserHistory.asp`.

[Sof] SoftSynth. Softsynth. `http://www.softsynth.com`.

[Tria] Tritonus.org. Plug-ins. `http://tritonus.org/plugins.html`.

[Trib] Tritonus.org. Tritonus: Open Source Java Sound. `http://tritonus.org`.

[W3C98] W3C. Objects, Images and Applets, April 1998. W3C Recommendation: `http://www.w3.org/TR/html4/struct/objects.html`.

[W3C99] W3C. Appendix B: Performance, Implementation, and Design Notes, December 1999. `http://www.w3c.org/TR/REC-html40/appendix/notes.html#recs`.

[W3C02] W3C. W3C XHTML 1.0 The Extensible HyperText Markup Language (Second Edition), August 2002. W3C Recommendation: `http://www.w3.org/TR/xhtml1/`.

[WaS] WaSP. WaSP: Fighting for Standards. `http://www.webstandards.org/about/`.

[web02a] Webopedia.com, January 2002. `http://www.webopedia.com/TERM/b/browser.html`.

[web02b] Webopedia.com, April 2002. `http://www.webopedia.com/TERM/C/CGI.html`.

[Web03] Webopedia.com, May 2003. http://www.webopedia.com/TERM/A/API.html.

Appendix A

Application Run Modes

Figure A.1 shows the run modes that are present in the first stages of the application. During this phase the application tries to determine the properties of the client machine accessing the system. It is interesting to see how most of the coding of the application is located around the run modes which test for Java for the first time. *Javachk* and *enabnok* are the run modes which offer the most possible paths to follow, and therefore, their coding was the most complicated. This was found out to be one of the most critical points of the application.

Figure A.2 shows the interaction between the run modes that build the last stages of the application, where the actual audio testing takes place. We can see how the structure is less complicated than in the first stages although the number of run modes is similar. Once the Java platform is available on the system and it is enabled on the browser, the tests are much simpler. Speech tests and music tests take two different *branches* of the diagram, and the help pages take the third.

A brief explanation of every run mode follows:

Language: language.tmpl

This is the first run mode executed in the application. It is stated as so in the start_mode() method of the application module. Since at first there is no run mode, the application defaults to what the start_mode() method reads. The HTML template shown with this first run mode, allows the user to select the language the application will be running on. This file, unlike the rest of the templates, is on the same path as the application module. This is so because at first the language is not known, so the language parameter is not yet set and the file cannot belong in any of the different language directories. After the selection of the language parameter is made, the application will directly look for the next template in the

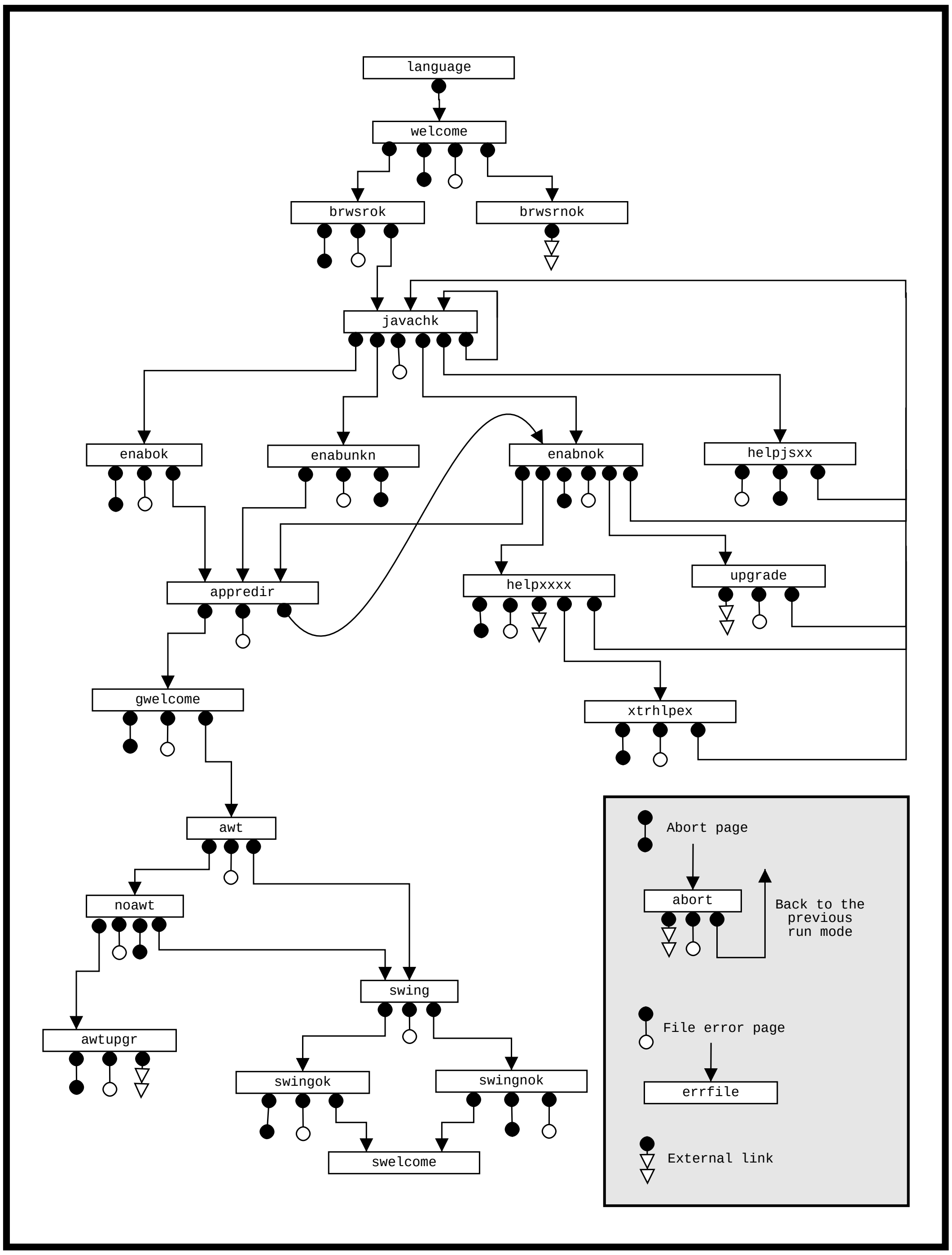

Figure A.1: First run modes of the testing application

directory which the language parameter has selected.

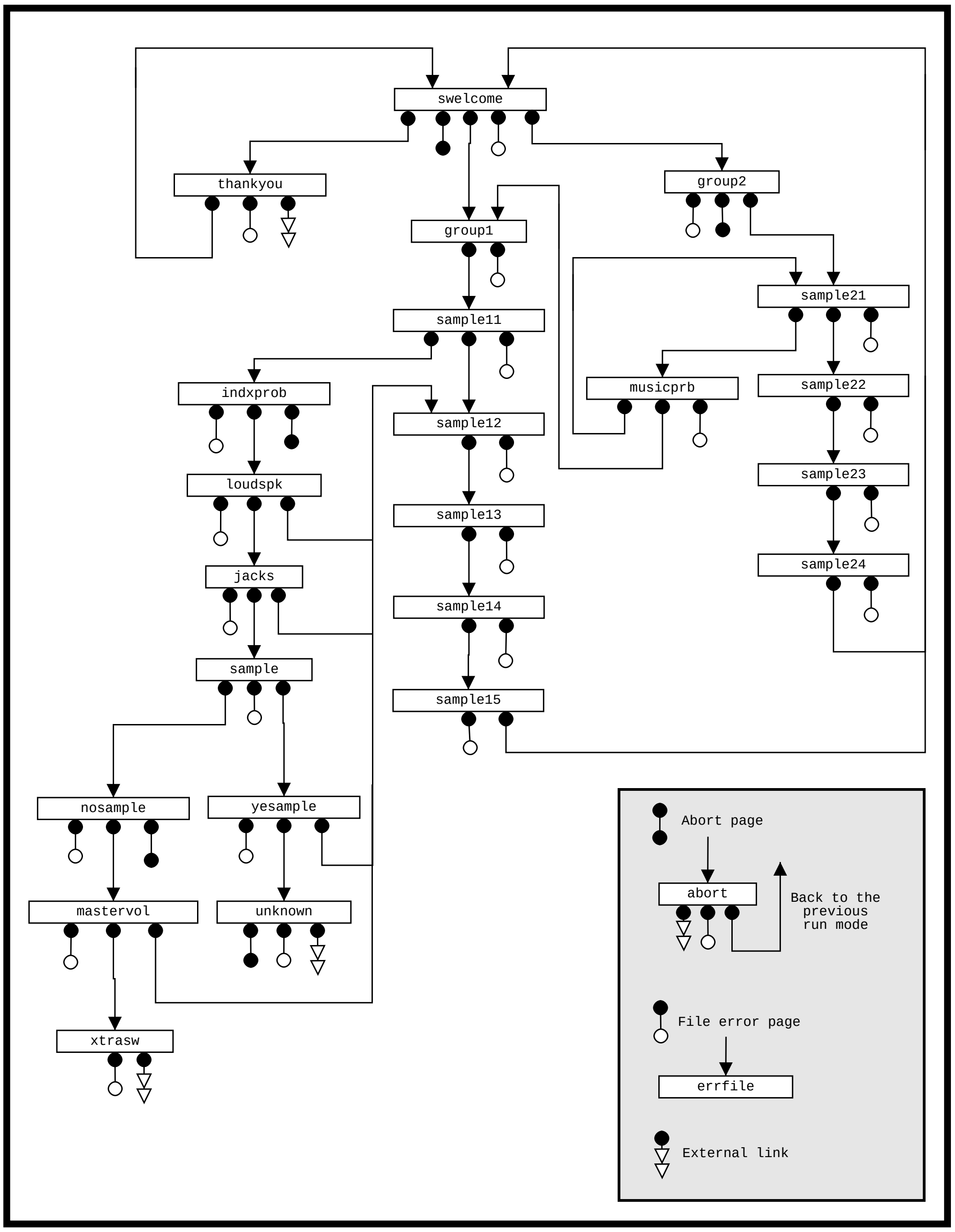

Figure A.2: Last run modes of the testing application

Welcome: welcome.tmpl

A brief explanation of what the application does is presented to the user. The user has the option to abort at this point or to continue with the application. At this point the user-agent string of the browser is being examined. If the browser is found to be non-java compatible, the button which allows to continue with the application will directly take the user to a run mode which recommends the changing of the browser. If the user-agent string is found to belong to a valid Java browser, then it is parsed in the search of a known browser, its version and the platform. This parsed information is written on the user file. If the user-agent string is unknown, it is also considered a valid Java browser.

Browser not OK: brwsrnok.tmpl

This run mode is executed if the user-agent corresponds to a very old browser which did not support the Java platform, or to a text based browsers[1]. The user is informed about this and is given different downloading options to get a new browser.

Browser OK: brwsrok.tmpl

The browser seems to have a valid user-agent string the user is informed about it. The kind of browser, version and platform is also shown. The user is about to start testing for Java support.

Java check: javachk.tmpl

This is one of the most complicated run modes of the application. This template checks for the presence of the Java platform with the use of Javascript. Loading Java applets can cause the browser to crash. Testing for Java through Javascript will not crash the browser whatever the result. Unfortunately testing through Javascript requires Javascript to be present and enabled on the system.

If Javascript is enabled on the browser, then the check for Java can be made. Javascript will redirect the browser to the corresponding run mode. If Javascript is not enabled, no redirection is performed and the user is given some options. Some help in the enabling of Javascript on the browser is offered and the possibility to continue without Javascript is also made available. The application can run perfectly without any Javascript support. Its functionality remains intact although some further HTML form checking is performed using Javascript but does not alter the application's results in any case.

If the user enables Javascript and selects the button which informs the application that Javascript is now enabled, then the application is sent to this run mode again. This time the redirection should work since Javascript is said to be now enabled. Should it not be

[1]Only the most popular text-based browsers are filtered: Lynx and Links.

enabled, the same options as before remain available. If on the third attempt the system does not have Javascript enabled despite the user saying so, the button is made unavailable and the only possibility is to continue without Javascript.

The help pages that can be accessed from this page show how to enable Javascript on different browsers. The type of browser is read from the user file and thus, a different link is made available for every browser. There is a general help page for browsers that the application has been unable to detect.

Help on enabling Javascript: helpjsmo.tmpl, helpjsne.tmpl, helpjsex.tmpl, helpjssa.tmpl, helpjsop.tmpl, helpjsun.tmpl

These web pages show how to enable Javascript for Mozilla, Netscape, Explorer, Safari and Opera browsers. There is also generic help for unknown browsers. Basically it shows the route to be followed through the browser menus in order to find the box to enable Javascript.

Java is enabled: enabok.tmpl

Javascript has found Java to be enabled on the system. The user is informed that his or her system has the Java platform installed and it is enabled on the browser being studied. It seems that Java applets will load, but unfortunately there is no other definite way to know if a browser will load an applet than by trying to do so. We are about to load the first applet.

Java is not enabled: enabnok.tmpl

Javascript has detected that the Java platform is not enabled. Three options are shown to the user: help on enabling Java for the browser (again, the browser parameter is read from the file and the appropriate link is selected), a browser and Java platform upgrade page is available and the chance to load the Java applet despite Javascript's advice saying it is not enabled[2] is the other possible option.

If the user selects the help pages and enables Java on the browser, when the *Java is now enabled* button is pressed, the run mode that is called again is *javachk*. This time, the system will be (normally) redirected to *brwsrok*. If the user attempts to enable Java more than three times without succeeding, the button becomes unavailable.

[2]This option was at fi rst not even considered but the testing of the application on different browsers prove it necessary; there was a Mozilla bug on the Apple Mac OS X version

Selecting the option to run the applet despite having no Java support will attempt to load the redirection applet on the user's system.

Help for Java: helpmozi.tmpl, helpsafa.tmpl, helpoper.tmpl, helpkonq.tmpl, helpmsie.tmpl

Show how to enable the java platform on Mozilla, Safari, Opera, Konqueror and Explorer browsers. There is also generic help for unknown browsers. It basically shows the route to be followed through the browser menus in order to find the box to enable Java. During the testing period it was found out that the *Java Enabled* box may be checked even though there is no Java platform present on the system at all. Fortunately Javascript does not rely on this information when deciding if Java is enabled or not.

Java platform unknown: enabunkn.tmpl

We reach this state if the user decides to go on through the application without Javascript. No information is known about the presence of the Java platform since it has not been tested for. This template file only gives information to the user about the applet that is about to be loaded.

Redirection applet: appredir.tmpl

At this point the first one of the applets is loaded. The applet does not have any graphical output, it just redirects the application to a new state. If the applet fails to load, and a browser crash does not occur, the user has the chance to go back to the *enabnok* run mode and try to enable Java again through the help pages.

This redirection, as the Javascript redirects did, also uses the GET method.

If the redirection works, the application is driven to the *gwelcome* run mode.

Graphical welcome: gwelcome.tmpl

Having reached this point, we can assure that Java applets can be loaded on the user's machine. An applet has just redirected the application to this run mode. The user is informed about this fact.

During the redirection procedure, the Java version is found. It is not searched for before this point since the Java version can only be obtained through a Java function and we had no proof of Java being available until now. Results showed that the Java version is the most determining parameter to look for to see whether the applets will load or not.

A brief explanation on the applets that follow is also given to the user at this point.

AWT applets: awt.tmpl

The first of the graphical aspects of the Java platform that is tested is its support for AWT components. A basic AWT applet is loaded on this page. The text that is to be shown on the applet is passed to it as a parameter thus allowing to use the same applet for different languages just by changing the message.

If the applet is not loaded, there seems to be a contradiction in the application. There is no way the user can be redirected to *gwelcome.tmpl* other than by a working Java applet. This means the Java platform is working correctly. All the Java platforms have the AWT graphical classes because AWT was released with the first Java release and if the redirection applet worked, the AWT should work fine as well.

No AWT applet: noawt.tmpl

Should there be some problems in the loading of the AWT applet, the user is given the chance to upgrade the system and is suggested not to continue before the AWT applet works. We do not know why the applet has not loaded given the redirection applet has. Despite being risky, the user is allowed to continue with the application even if the AWT applet did not load.

Swing applet: swing.tmpl

The AWT applet loaded correctly and now we test for Swing support with a very simple Swing applet. This time the system may not be able to load the applet given everything else is working fine. Swing was not available since the first Java releases and some old Java platforms do not support it[3].

The Swing applet also has the text passed to it as a parameter so that language support is achieved easily. Unlike the AWT applet, the Swing applet will look the same on every browser (if it is correctly loaded). One of the benefits of the Swing classes over the AWT ones is that they do not use any of the native graphical resources from the platform they are being executed on. Being so, the Swing applet will look identical on every browser

[3]See chapter 3: Java

whereas the AWT applets depend on the platform where they are being executed and may look different (text size specially).

Swing not OK:swingnok.tmpl

The Swing applet has not loaded. Some of the system's properties are shown to the user at this point. The graphical testing has finished and the user gets a summary of the properties of the system so far. The sound testing is about to begin.

Swing is OK: swingok.tmpl

The system has been able to load Swing applets and the user is informed about this. The properties gathered up to this point are shown on screen for the user to read. The sound testing is about to begin.

Sound welcome page: swelcome.tmpl

This page is the starting point for the audio testing. It loads an AWT applet that gives access to both the speech samples and the music samples. When either of the sample sets have been listened to (or attempted) the buttons change to a *Done! Repeat?* status. When both sets of samples have been tested, the text on the page changes telling the user the testing is over.

Group One, speech samples: group1.tmpl

Just a simple explanation of the samples that are about to be heard. Sample rates, number of channels, coding techniques, and quantification information of the samples is presented before listening to them.

Speech samples: sample11.tmpl, sample12.tmpl, sample13.tmpl, sample14.tmpl, sample15.tmpl

These files try to play a speech sample through a Java applet on the user's machine. They load the same applet but refer to different sound files depending on parameters passed to the applet. The same thing happens to the text, so the text displayed is different on each case. The audio file is automatically loaded, but a button is also available for the user to hear the sample again.

If the sound sample is not heard and the user says so by pressing the corresponding button, it is written as so on the user file except when it happens on *sample11.tmpl*. If the first sample is not heard, it may not be that the applet has not been able to load correctly, but it may mean that sound is not available on the computer at all.

The first sample that we attempt to play has a special format. When Sun released the first Java version, it only had support for one type of sound file. Our first sample has this same format, so every single Java platform available should be able to play it. This is the reason why if no sound is heard, the application inquires the user about their hardware and software configuration. The application does not move on to the second sample unless the first sample has been heard (if the first sample is not heard the following wont be either.).

The other samples in this set have similar properties as the first one, but differ in some aspects, therefore it is not expected that every Java platform is able to play them. If the user states that no sound has been heard on speech samples from 2 to 5, we have to assume that the system can play sounds, but the Java version is unable to play these new sound formats.

After the last speech sample has been visited, the application returns to the *swelcome.tmpl* file.

Index of possible problems: indxprob.tmpl

This page informs the user about the possible things that may have gone wrong: sound is not heard and it may be a hardware or software problem. Missing loudspeakers (or headphones) may be the hardware reason or the master control volume may be the software reason. The user is informed about possible solutions.

Checking the loudspeakers: loudspk.tmpl

The user is told how to correctly setup a pair of loudspeakers. Basically checking they are plugged to the main power supply and that their level is turned up high enough for the user to hear sounds. The applet is made available to see if after checking the loudspeakers, the sample can be heard.

Checking connections: jacks.tmpl

It is not that unusual to find systems where the problem is that the speakers or headphones are not plugged into the correct connector. Information on where to plug the speakers or headphones is given. The applet can also be tested here.

Trying to play a sound sample outside an applet: sample.tmpl

If the system is still unable to reproduce sounds, it may be a sound card problem. In order to discard this possible malfunction, a sound sample is made available for downloading outside a Java applet. If the user can play this sample, then their sound system is correctly set up although they do not hear the samples played through the applets. Should the user not be able to listen to this sample either, some further checking will have to be made.

No sample heard: nosample.tmpl

If the downloaded file is not heard then either there is a problem with the sound card and its configuration or it may be that the main volume control level is too low for any sound to be heard.

The master volume: mastervol.tmpl

The user is told to raise the level of the main volume control of the system. This page has a link to some tips on where to find the main master volume controls on different platforms. It tells the user which control bar to check for.

The sample was heard: yesample.tmpl

If the sample could be heard, a conflict between applications using the same resources may be happening. The system is perfectly capable of playing sounds but it will only play them outside Java applets. We ask the user to close all the applications that may be using the sound resources and try to listen to the sample through the applet again.

Java sample is still not heard: unknown.tmpl

At this point we do not know what may be going wrong and we inform the user about it. The system is able to load applets correctly and sound is available if it is not played through applets. We do not know why this happens. Our suggestion is a system upgrade.

Group two: group2.tmpl

The same way we informed the user about the speech samples about to be heard, we do the same thing with the music samples. The different sound formats are given and some of their particularities too.

Music samples: sample21.tmpl sample22.tmpl, sample23.tmpl, sample24.tmpl

Different sound formats are tested and within the same sound format, special characteristics. Mono and stereo samples are presented with the *aiff* file for the user to appreciate the difference between using one or two channels. Testing for the left speaker, right speaker and both speakers is done with the *wav* file. The 16 bit *au* sample presents different sample rates where sound quality can be compared. Finally we test a MIDI file against the same sample in CD quality for the user to appreciate the difference.

These samples are huge. We cannot use popular compressor utilities widely used on the Web nowadays because we would need the user to have special Java packages which to reproduce those files with. The user is therefore warned beforehand that the samples are big and an estimate of the time they will take to download considering different connection speeds is also given.

The Java runtime needed to play these sound files needs to be relatively up to date and therefore it is not unusual to find systems that will be unable to play them. However, before deciding that this is the case we warn the user that maybe the file has not had enough time to download. When the user is sure that the file has had enough time to download and still no sound can be heard then a negative answer is taken as a valid one.

These samples need the Java sound package to be installed on the system[4].

No music sample has been heard: musicprb.tmpl

This run mode can only be executed once. If the user does not hear the first music sample, we want to make sure it is because of an old Java platform and not because of a slow connection. The user is given the chance to go to the speech samples (*group1.tmpl*) to see if his or her system can reproduce sound, or to wait for the file to download completely. The second time the user answers that no sound can be heard, this run mode is not reached, instead *sample22.tmpl* is loaded and the user file is updated with a negative answer concerning the first sound sample.

After the last sample has been heard (*sample24.tmpl*), the system is taken to the *swelcome.tmpl*.

[4]See chapter 3: Java for more details.

Quit the application: abort.tmpl

In many of the web pages of the application, the user has the chance to abandon the testing application. What happens then is that the user file is read and a table with all the results available up to that point is presented to the user.

Non valid run mode: autoload.tmpl

Since some of our pages are redirected to new run modes through GET requests, the parameters that are sent from one run mode to the other can be seen in the browser's navigation bar. The run mode parameter is visible to the user and if it is changed (intentionally or unintentionally) a security problem could arise. If the run mode that is read does not belong to any of the run modes in the application module, then the *autoload.tmpl* file is shown.

It informs the user that a non valid run mode has been tried to be reached. This attempt to load a mode which does not exist run mode is naturally written on the user file.

Error in the reading or writing: errfile.tmpl

The user file created to store the information relevant to every different session is kept in the server. Writing and reading from the server has some security risks which are discussed on the *security* section.

Whenever there is a security problem like trying to write to a file that has not been initialized by the application previously, or whenever the file cannot be written to the disk for whatever reason, all the links in the page that is being visited are *locked*. What we mean by this is that all the links on the current page will point at the *errfile* run mode. The user is informed that there has been a problem in the reading or writing of files in the server and that the application cannot continue any further.

Appendix B

Application Module

An extract from the setup() method overriding in the application:

```
sub setup{
    my $self = shift;
    $self->start_mode("language");
    $self->mode_param($runmodeparam);

#all subroutines map to files with the same name (extension .tmpl)
    $self->run_modes(
     "AUTOLOAD" => "autoload",
     "language"=>"language",
     "welcome"=>"welcome",
     "source"=>"source",
     "brwsrok"=>"brwsrok",
     (...)
     "mastervol"=>"mastervol",
     "mixer"=>"mixer",
     "thankyou"=>"thankyou",
     "musicprb"=>"musicprb",
     "abort"=>"abort",
     );
}
```

Example of one of the subroutines in the application:

```
sub brwsrok{
    my $self = shift;
    my $q = $self->query();
    my $url=$q->url;
    my $timestamp=‘date‘;
    my $id=$q->param("id");
    my $fileerror="FALSE";
    my $detectedbrowser="unknown";
    my $version="unknown";
    my $os="unknown";
    my $ownmd="brwsrok";

    my $tmpl=$self->load_tmpl("brwsrok.tmpl");
    $tmpl->param(SELF_URL=>$url);
    $tmpl->param(chgmode => $runmodeparam);

    $detectedbrowser=&searchfile("Browser","$logdir/user.$id");
    chomp($detectedbrowser);
    if ($detectedbrowser=~ /UserFile/) {
      $fileerror="TRUE";
      my $errresult= &writefile("$logdir/errorlog.$id",
      "$logdir/aux.$id","id",$id,"Writefile search error
      message while searching for Browser",
      $detectedbrowser,"Runmode",$ownmd." - ".$timestamp);
    }

    $os=&searchfile("Platform","$logdir/user.$id");
    chomp($os);
    if ($os=~ /UserFile/) {
      $fileerror="TRUE";
      my $errresult= &writefile("$logdir/errorlog.$id",
      "$logdir/aux.$id","id",$id,"Writefile search error
      message  while searching for Platform",$os,"Runmode",
      $ownmd." - ".$timestamp);
    }

    $version=&searchfile("Version","$logdir/user.$id");
    chomp($version);
    if ($version=~ /UserFile/) {
      $fileerror="TRUE";
      my $errresult= &writefile("$logdir/errorlog.$id",
      "$logdir/aux.$id","id",$id,"Writefile search error
      message  while searching for Version",$version,"Runmode",
      $ownmd." - ".$timestamp);
    }

    my $writeresult = &writefile("$logdir/user.$id",
    "$logdir/aux.$id","Runmode",$ownmd." - ".$timestamp);
    if ($writeresult) {
      $fileerror="TRUE";
      my $errresult= &writefile("$logdir/errorlog.$id",
      "$logdir/aux.$id","id",$id,"Writefile routine error
      message",$writeresult,"Runmode",$ownmd." - ".$timestamp);
    }

    $tmpl->param(version => $version);
    $tmpl->param(browser => $detectedbrowser);
    $tmpl->param(platform => $os);
    $tmpl->param(start => "javachk");
    $tmpl->param(quit => "abort");
    $tmpl->param(id => $id);
    $tmpl->param(ownmd => $ownmd);

    if ($fileerror eq "TRUE"){
      $tmpl->param(start => "errfile");
      $tmpl->param(quit => "errfile");
    }

    my $output="";
    $output.=$tmpl->output;
    return $output;
}
```