Quality of Service for Remote UI

A thesis submitted to the University of Utrecht
for the degree of Master of Science
in the Department of Computer Science

August 28, 2005

By
Gertjan van Montfoort

Thesis number: INF/SCR-04-97

Abstract

Within the area of consumer electronics there is need for remoting a user interface from one device to an other. The solution must be brand independent, work on resource constrained devices and be responsive to the user in a wireless (possibly low bandwidth) environment. This combination is an interesting problem to solve.

Within this thesis project we did research in the area of remote user interfaces and present test results of potential remote user interface descriptions. Also we describe optimizations for remote user interfaces like an optimization that lowers the need of communications between the server and the client saving bandwidth and processing resources. Also we introduce a new encoding generalized for XML that not only reduces the file size but also saves processing resources at client side. Furthermore we give advice on how animations can be implemented optimally within remote user interfaces.

The results presented in this document can be used to create a new standard for remote user interfaces.

Acknowledgements

First of all I like to thank Philips Research that offered this thesis project and especially Walter Dees that was my internal supervisor within Philips. He has a lot of knowledge on the subject and he helped me a lot.

Also I like to thank Doaitse Swiersta for being my supervisor from the University of Utrecht. He was always able to light the subject from a different angle bringing forth new ideas. Also he suggested me to read "The elements of style".

Strunk and White I like to thank for writing the book The elements of style [SW79]. It improved my english and made my thesis better readable.

Bart Golsteijn I like to thank for being my roommate at Philips Research and for the nice discussions that we had at the lunch break.

My roommates in my student house (Hoogstraat 12 in Eindhoven) I like to thank for the nice time that I had in Eindhoven.

Last but certainly not least I like to thank my parents for always supporting me with good advice and a lot of trust.

Contents

1 Introduction 1
- 1.1 Some use cases . . . 2
- 1.2 What do we need for a remote user interface to work properly? . 3
- 1.3 Requirements . . . 4
- 1.4 State of the art techniques for Remote UI . . . 4
  - 1.4.1 UPnP Remote UI . . . 4
  - 1.4.2 XHTML . . . 6
  - 1.4.3 SVG . . . 7
  - 1.4.4 RDP . . . 7
  - 1.4.5 UI Fragments . . . 8
  - 1.4.6 VNC . . . 10
- 1.5 Background info . . . 11
- 1.6 Abbreviations . . . 11

2 Testing existing protocols 13
- 2.1 What to test . . . 13
- 2.2 Test case . . . 14
- 2.3 Existing markup content formats to test . . . 15
  - 2.3.1 XHTML . . . 15
  - 2.3.2 SVG . . . 18
  - 2.3.3 VNC . . . 19
- 2.4 Test results . . . 20
  - 2.4.1 Client performance/resource consumption . . . 20
  - 2.4.2 Server performance/resource consumption . . . 24
  - 2.4.3 Bandwidth usage . . . 28
- 2.5 Conclusions . . . 30

3 Next steps 31
- 3.1 Identified problems bandwidth consumption . . . 31
- 3.2 Identified problems at the server side . . . 32
- 3.3 What to do next . . . 32

4 Optimization 1: Client eventing 33
- 4.1 Introduction . . . 33
- 4.2 At the client side . . . 34
- 4.3 The proposed structure in detail . . . 34
  - 4.3.1 sendAction element . . . 34
  - 4.3.2 The argument element . . . 37

4.3.3 DOM extension 37
4.4 Examples 38
4.5 Sending the HTTP request 39
4.6 Test results 41
4.7 Client eventing conclusion 43

5 Optimization 2: SAX Encoding 47
5.1 Introduction 47
5.2 Encoding of SAX functions 48
5.3 Identifiers 49
5.4 START flags 51
5.4.1 STARTELEMENT 51
5.4.2 STARTPREFIXMAPPING 53
5.4.3 STARTCHARACTERS 53
5.4.4 STARTPI 53
5.5 Encoding string values (encstring) 53
5.6 Whitespace handling 54
5.7 Testing 54
5.7.1 Test local decode performance 55
5.7.2 Test remote decode performance 56
5.8 SAX Encoding conclusion 58

6 Optimization 3: Reduce animation resources 59
6.1 Testing results 60
6.2 Animation conclusion 60

7 Possible other optimizations 63
7.1 SVG widgets 63
7.2 Combining VNC with SVG 64

8 Conclusion 65

A Testing procedures 69
A.1 Test environment 69
A.2 Software used for testing 70
A.3 Testing procedures 70
A.3.1 How are the measurements created and started 70
A.3.2 Running the tests 71
A.3.3 Transforming the test results 72

B SAX Encoding example 77

C SVGForms abstract 83

List of Figures

1.1 Device hierarchy for multi-level stylesheets 9
1.2 Adaptive flow of VNC 10

2.1 Use case example 14
2.2 Processing time XHTML clients 22
2.3 XHTML scripting example 23
2.4 SVG declarative animation example 24
2.5 Processing time SVG and VNC clients 25
2.6 Processing time used at the server side 27
2.7 Bandwidth used 29

4.1 Client eventing example 34
4.2 sendAction element DTD 35
4.3 Syntax of clock values 37
4.4 DTD argument element 37
4.5 Examples of client events 39
4.6 Example of client event for HTTP message 40
4.7 Bandwidth measurements 42
4.8 Client processing resources 44
4.9 Server processing resources 45

5.1 BNF for SAX Encoding 50
5.2 Example how scripting can be more compactly stored 54
5.3 Client processor utilization when sending/decoding batikce.svg 100+ times 57

6.1 SVG declarative animation example 59
6.2 Animation reduction if we choose for animation skipping 61

B.1 Example to encode with SAX Encoding 77
B.2 Calls on ContentHandler when parsing example in figure B.1 . . 78

Chapter 1

Introduction

First we give a brief introduction of what exactly a remote user interface is. What we can do with it and what is needed for a remote user interface.

So what is a remote user interface exactly? A user interface is usually shown locally on the device that is running the application. In the consumer electronics ,however , there is a growing opportunity and need for remoting a UI. With remoting we mean that the user interface is shown on a physically seperated device other than the device that is running the application. The communication between the user interface and the application is done through a remote network.

Why do we need to remote a UI? A good example for explaining remote user interfaces is the remote control for a television. This remote control is only created to give input to the television. Also other devices have remote controls like a stereo, VCR, DVD and maybe even a central heating. Is it not handy if the same remote control can be used for all these devices? The problem is that all these remote controls have their specific functionality described by device specific buttons. We can create a union of all these controls on one remote control, but this results in a remote control with a lot of different buttons that is not very clear to operate: the number of buttons on the remote control can easily confuse the user.

A different solution is to create a remote control with a local display that for each device shows a different user interface. This user interface is downloaded from the device that needs to be controlled. The remote control then can bind its local generalized buttons or touch screen to the functionality of the device specific user interface. So with an advanced remote control that can display remote user interfaces we can switch TV channels, turn on the VCR, dim the lights and turn on the heater without switching remote controls or leaving the couch.

But do we really need a specialized remote control with a display to control all these devices? A mobile phone, PC or PDA can also be used to display and control a remote user interface. The ideal situation would be that any electronic device can be controlled using a display device that in the current situation is the most convenient to use.

In the next chapters we use client for the device that displays the remote user interface and server for the device that is controlled by the remote user interface.

1.1 Some use cases

The following use cases gives a better view of the possible applications of remote user interfaces. These use cases can be generalized to include other types of devices such as portable devices and media players.

Use case 1: Laura is playing Tetris at home on her mobile phone. The display, however, of her mobile phone is very small. The option ”display on TV” on her mobile phone would be very nice in this case. This can be achieved by a remote user interface, the mobile phone acts in this case as a remote UI server and the TV as a remote UI client. Also remote UI can be used to edit an SMS or MMS message on the TV so that the user can see the total message at once.

Added value

- Make use of larger screens for mobile devices
- More uses for a TV

Use case 2: Bas is 15 years old and wants to buy a MP3 player so that he can listen to his favorite songs. He is searching for an MP3 player that allows him to see which song is currently playing and a list where he can select songs to play. By integrating a remote user interface server in an mp3 player Bas can use any displaying device in range. So Bas can use for example his mobile phone to look at which song is currently playing and select a new song to play out of the list while the MP3 player is still in his pocket.

Added values:

- The display can be wirelessly connected to a mp3 player resulting in that a user can keep the mp3 player in its pocket while choosing a song
- Production costs of mobile devices get lower.

Use case 3: Mike has a lot of work to finish for his boss and has to work overtime resulting in that he is going to miss his favorite show on the television. By using the remote user interface of his VCR on his mobile phone or PC he can program the video recorder from his desk at the office so that he can later on watch his recorded favorite show at home. In this case the video recorder is the UI server and the mobile phone/PC is the UI client. For this use case Mike also needs a proxy server in his home that can be used to connect to by using a WAN.

Added values

- Remote access to devices in home

There are also a lot of other use cases but these above give a general idea about what can be achieved with remote user interfaces. Notice that a mobile phone is in use case 1 a server and in use case 2 and 3 a client.

1.2 What do we need for a remote user interface to work properly?

It is not very convenient for a user to use a cable to connect a mobile device to a server. So the first thing we need is a wireless LAN to allow a device to communicate without using physical wires. This wireless LAN has to be supported by both the server and the client.

To communicate over a wireless LAN we need a standard protocol to transfer the data. Within the wireless LAN there are already good standards that can be used, and these standards are defined in the IEEE 802.11 [IEE] (Institute of Electrical and Electronics Engineers) specifications.

The IEEE 802.11 standards, however, only describe protocols for transferring data, not protocols to find and describe other wireless devices, so-called device discovery protocols. We need a protocol that can give a list with available devices in the wireless LAN and we need to know what services these devices can provide. Also we need to know the location/unique name of the device. A

good example of what can happen if a device does not have a unique name is a user trying to switch channels on the TV in the living room while in fact the user is switching channels in the bedroom. When a user can choose a device by name, for example "TV living room", then the user can really target the correct device.

There are multiple protocols available that offer device discovery services. Known protocols are BlueTooth, SLP (Service Location Protocol) and UPnP (Universal Plug and Play). Philips prefers UPnP.

Also we still need some default remote UI protocol to remote the user interface from the server to the client. With a remote UI protocol we mean a technique to display and use a user interface on a remote device. Various remote UI protocols are already defined, however, there is currently not a default that works between different vendors and that meets our requirements as defined in section 1.3.

1.3 Requirements

A list of requirement where our solution must deal with

- Resource constrained clients
- Resource constrained servers
- Wireless LAN (variable/low bandwidth)
- User interface must stay as responsive as possible; ideally it must respond like a local user interface
- User interface must be remotable

1.4 State of the art techniques for Remote UI

1.4.1 UPnP Remote UI

UPnP [Mic00] is a service discovery technology that is managed by a forum consisting of a large group of companies. The goal of the UPnP Forum is to allow devices to connect seamlessly and to simplify the implementation of networks at home and in corporate environments.

Within the UPnP technology the following components are described

- Device: a specific type of device that offers zero or more services to the network.
- Service: a service on a device that other (remote) applications can use.

Also the term control point is often used. A control point is used to find a device or service of interest. The control point is implemented at the device that takes the initiative to search for other devices that it can use.

UPnP is implemented by an architecture consisting of 6 layers. Below we give a brief description of these layers

- Addressing Layer: is responsible for getting an IP-address for the current device
- Discovery: after a unique IP address is assigned to the device it multicasts a discovery message over the network by sending a HTTP notify message. Control Points can listen on the network for these multicasts, when interested in that particular device the control point can store a reference, that is stored in the multicast message, to the device. Also control points can sent a broadcast to search for devices.
- Description: the discovery message of the device contains an URL to a device description. With this URL the control point can request information about the device, like what kind of services the device provides. With this information a service can be used.
- Control: actions that services provide can be triggered by sending SOAP messages over HTTP to the control URL.
- Eventing: UPnP also supports eventing. An application can subscribe to a service that supports eventing. When the state changes the subscribed application gets notified.
- Presentation: Optionally a presentation can be stored in the device. This presentation can give an overview of the device and its services by using HTML.

UPnP is a technique that allows discovery and communication between devices. But how must a client know which methods it needs to call. Take as example a device that provides printing services to the network. How must this device be used to print a document? By calling the print function? But which arguments does it want? To solve this problem default interfaces are defined. The Printer Device standard [Mic02] is a good example that defines the methods and variables that a default printer needs to have. By using this default interface all client devices can use a printer service without worrying about the type/brand of the printer.

A large problem of using these standard interfaces is that devices don't have the possibility to offer their own special features. They are only limited to the default methods within the default interface. The company that created the printer has spent time and money in creating a special feature that later on gets ignored because a default interface is used by UPnP. Remote user interfaces can give a good solution for this problem; printers can then offer their unique features to a user by showing their own user interface on a client device so that the client can choose to use printer specific functionalities.

UPnP also defines standard interfaces for discovery and usage of Remote UI [Mic04]. When implemented, UPnP Remote UI can be used for discovery of UPnP Remote UI enabled devices. Furthermore, default functions are added to find a match between the different remote UI protocols that the devices support. Also functionality is added to connect two devices. No required content formats, however, are defined within the standard. This means that if a server device supports UPnP Remote UI this does not mean that all UPnP Remote UI enabled devices can be connected. A match of user interface content formats is not always available because there is no required content format defined.

Implementing the remote UI client and server interfaces results in a remote UI discovery system; not a standard for describing remote UI's.

1.4.2 XHTML

XHTML [W3C00b] is a standard that is developed by W3C. XHTML is an XML version of HTML that solves the problems with inconsistencies within the HTML format. Also XML is more extensible than HTML. By making the HTML specification conform to XML we get an easy extensible specification that for example also can contain XForms and SVG fragments. Because HTML already looks very much like a XML structure XHTML is still compatible with HTML 4 browsers. Here we give an example of HTML syntax.

<B><I>Bold italic text</B></I>

XML is more restrictive: the example given here above is not allowed. This results in better-formatted documents that can more easily be interpreted, resulting in more consistent browsers. XHTML is mainly used for formatting and displaying hypermedia but can also be used to create some basic user interfaces.

An advantage of XHTML is that there are already a lot of browsers on the market. A disadvantage is that these browsers are still not very consistent on how to show an interface. The reason is that there are a lot of extensions possible like CSS (Cascading Style Sheets) version 1, 2 and 3 [W3Ca] or DOM (Document Object Level) versions 1, 2 and 3 [W3Cb]. Some mobile browsers implement only CSS 1 and others CSS 1 and 2. If functionality of CSS 2 is used it is not shown on a browser that does not support CSS 2. With DOM this is even a larger problem because if a remote user interface uses DOM level 2, but the client does not support this, then this results in errors displayed at client side or improper behaviour .

Also there are differences between interpretations of the JavaScript language. A good example is within Javascript in communication between frames; in the example below we show 4 ways on how this can be described.

1. window.subsection.callMethod();
2. window.frames("subsection").callMethod();
3. window.frames(1).callMethod();
4. window.frames[1].callMethod();

All statements describe the same action but only the first and fourth statement are understood by a Netscape/Mozilla kind of browser while Internet Explorer and Opera only understand the second and third statement. We did not find a union that all brands of browsers understand: we had to write exceptions for different kind of browsers.

XHTML is only the user interface description language, to transport the user interface to the client and communicate from the client to the server HTTP [oai96c] is used.

1.4.3 SVG

SVG is developed by W3C (World Wide Web Consortium) and is a technique to describe a user interface in vector graphics comparable to Macromedia Flash [Mac].

SVG is an open standard and there are a lot of client applications available that can display SVG documents. Adobe for example created a free SVG plugin for Internet Explorer. Also there are open source clients available from which the source code can be adapted. SVG uses XML which makes the file format less compact than the Flash binary files. On the other hand SVG is more extensible because it is an open standard.

Currently on mobile platforms more and more SVG enabled devices come on the market which make implementing a standard in SVG more easy and acceptable. Thats why SVG can also be a good option for Remote UI because a lot of resource constraint devices already have proven that they can run a SVG viewer. Furthermore there are projects going on that optimize the SVG performance and resource consumption for mobile devices by using hardware solutions. A good example is OpenVG [KHR] which is currently being developed and this specification can be seen like the OpenGL acceleration specialized for SVG.

For SVG there exist three different profiles:

- SVG Tiny [W3C03]: is a SVG profile especially for the mobile phone; it is limited in graphical possibilities which saves device resources but still is able to show a SVG Tiny websites.

- SVG Basic [W3C03]: is developed for PDA's that normally have more resources than a mobile client, but still constrained in graphical possibilities.

- SVG [Lis03]: is the full specification without any resource sparing restrictions.

Currently a SVG 1.2 specification [W3C05a] is being created in which the old basic and tiny specifications are replaced by one mobile specification.

1.4.4 RDP

RDP (Remote desktop protocol) is used by Microsoft terminal services to distribute a user interface to a client device. This is done by transferring the WIN32 display commands directly to the client which effectively limits the use of RDP to only WIN32 API compliant clients. This restriction is in our case a large disadvantage because interoperability is needed between clients to be platform independent: meaning that a wrapper facade needs to be implemented that maps the original graphic API to the WIN32 API. This mapping is not really feasible because the WIN32 API is very large.

RDP sends all key presses and mouse events directly to the remote application on the server, the server in response sends back new WIN32 API commands that change the remote UI on the client.

RDP is based on the ITU-T T.128 [ITU97] recommendation.

1.4.5 UI Fragments

User interface fragments [eWD] is a technique developed by Philips that is tailor made for Remote UI's. UI fragments defines an abstract widget XML structure comparable to XForms [DKMR03]. This structure gives clients the possibility to keep its own look and feel while rendering the user interface (using native widgets). The name "user interface fragments" is chosen because it can send screen updates in fragments. In XHTML, SVG, Flash or XForms after submission of a request the server typically returns a complete new page. This is not always necessary especially when monitoring the progress of a playing DVD. It is only required to set the position of the progress slider instead of updating the total screen each time the progress goes one second further. This can be done by sending a fragment to the page that only updates a part of the screen which results in less bandwidth needed.

Also incremental updates can result in incremental rendering sparring resources on the client.

Within UI Fragments also Multi-Level Stylesheets [Dee04] is explained. The problem that this feature solves is that when a user interface becomes more abstract the user interfaces that are generated may become less attractive. By using a stylesheet tree structure like displayed in figure 1.1 it is possible to specify styles for capabilities of devices. Within the root of the tree the styles are specified that apply to all devices. Childs of the root styles can be created that only apply to a group of devices that have a specified device capability (e.g. landscape layout).

Traversing down the tree and taking the styles that correspond to the capabilities of the current device determine the resulting style for that device. If a new device connects to the remote UI that uses a resolution that is not specified it can still use the styles from the tree until the aspect ratio node.

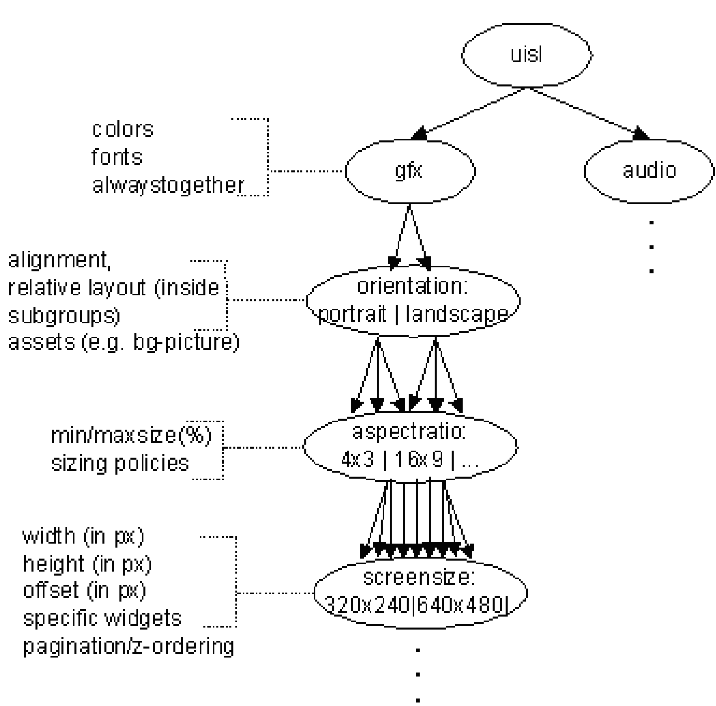

Figure 1.1: Device hierarchy for multi-level stylesheets

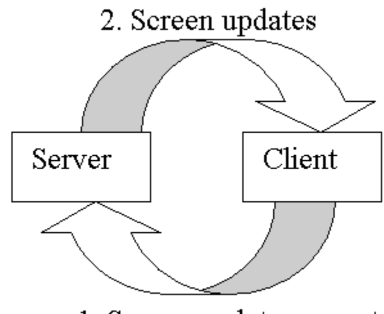

Figure 1.2: Adaptive flow of VNC

1.4.6 VNC

VNC stands for Virtual Network Computing and is based on the RFB (Remote FrameBuffer) protocol [Ric05]. Within RFB the screen data is generated at the server side and sent over the network in a raw image format. The client renders the sent screen data directly on its screen. This is only done when the screen changes, so if no changes on the screen occur, no image data is sent. Also the server stores a client framebuffer that calculates the difference when a change occurs: only the changed parts of the screen are sent (not the full screen) thus saving bandwidth. Calculating changes, however, takes a lot of resources at the server side.

The client of VNC is kept very simple; resulting in a lot of implementations for many platforms. Furthermore the resources are automatically balanced on the server, client and bandwidth because it uses a client initiated roundtrip mechanism as described in figure 1.2. This roundtrip mechanism works as follows. The client initiates a request to the server for new screen updates. If the server has screen updates it sends these back. The client draws these screen updates on its screen, when finished the client sends a new request to the server for new screen updates. This architecture results in a synchronous flow that automatically balances the resources. If there is a slow network and the user drags a window the window will make large jumps and in between no other screens are generated; this is called skipping frames. Skipping frames can be annoying to users because animations can get hiccups. If the network gets faster more requests will come from the client resulting in more screen updates to be sent by the server. This roundtrip mechanism results in an adaptive architecture that reacts on bandwidth, client resources and server resources.

VNC also has support for compression which saves bandwidth but takes extra resources.

A similar protocol , developed by Intel, with the same behavior as VNC is XRT2 (The Extended PC Remoting Technology Protocol 2) [WEJ+02].

1.5 Background info

Currently Philips, Samsung, Intel and other companies are working together to create a standard for Remote UI. This thesis project can be useful input for the further development of a standard because it contains measurements of resource consumption for some of the different Remote UI description standards. Also we try to give a clear image why SVG is in our opinion the best choice for a standard in Remote UI.

1.6 Abbreviations

API	Application Program Interface
CSS	Cascading Style Sheets
DOM	Document Object Level
DTD	Document Type Definition
HTTP	Hypertext Transfer Protocol
IE	Internet Explorer
IEEE	Institute of Electrical and Electronics Engineers
LAN	Local Area Network
PDA	Personal Digital Assistant
RFB	Remote FrameBuffer
SAX	Simple API for XML
SMIL	Synchronized Multimedia Integration Language
SLP	Service Location Protocol
SOAP	Simple Object Access Protocol
SVG	Scalable Vector Graphics
UI	User Interface
URL	Uniform Resource Locator
VNC	Virtual Network Computing
WAN	Wide Area Network
XHTML	XML-compliant Hypertext mark-up language
XML	Extensible Markup Language
XRT2	The Extended PC Remoting Technology Protocol 2

Chapter 2

Testing existing protocols

To get an indication of what is a good protocol that can efficiently show a remote user interface on a client device without consuming a lot of resources we started with testing. Within this testing phase we selected three markup languages/protocols for remote UI that we see as possible starting points respectively, XHTML, SVG and VNC.

We started with testing because we don't have a clear view on how the resource consumption is balanced between these three protocols. If we use VNC to transfer a user interface to a different device how much bandwidth is consumed and how is this related to the other protocols? Also resource consumption at the server and the client side was not very clear: does SVG use less processor resources on the client than XHTML?

This chapter describes the test results; for a more detailed description on how these tests are done see appendix A at page 69

2.1 What to test

The following values are measured in our tests.

- Bandwidth
- Memory
- CPU percentage

Testing is done on a wired network; we are interested in how much bandwidth is required when transferring a user interface. Later on this can be calculated within the different IEEE 802.11 wireless specifications as explained in [Pla03] to see what the response times are in an optimal wireless environment.

To see how we actually measured bandwidth/memory and CPU percentage we included a detailed description in our appendix, see chapter A on page 69.

Figure 2.1: Use case example

2.2 Test case

For testing we describe a media browser/player that suggests that it can play audio and video files.

In figure 2.1 we show the first opening screen that we will get when starting up the remote user interface. We have two options in the combobox in the upper right corner, "Now playing", and "Media Library". The center and the right screen display two black areas. The left black area contains the media library. The right black area contains the playlist. When the media library is selected it is possible to select a song and add it to the playlist.

Why a media browser/player as a test use case?

- It contains eventing to the server application in the form of play, pause, and add to playlist actions.
- It contains server eventing to the client; the current play status of a music file is displayed in the lower left corner and must be updated from the server side.
- It is a good example of an application that contains frequently occurring functionalities within wireless remote user interface applications, like play and progress indicators, control buttons, and status information.
- It contains different widgets like a combobox, a slider, buttons and labels

The application on the server side that gives input to the user interface is kept as simple as possible. No real audio playing is done; only timing and display

information is generated by the server application. The UI that is generated for each of the tested protocols must resemble each other as close as possible to get a comparable test case.

The sequence of user actions on the UI that we are going to test is described here.

1. Startup the remote UI
2. Browse through the media library. While browsing large animations will be displayed that move a big part of the screen from the right to the left.
3. Add one song to the playlist
4. Play the song
5. Wait until the song finished playing. In the mean time the slider and the counter in the lower right corner are updated by server events.
6. The user, after adding a new song to the playlist, drags the slider to the end position. We use here a greedy type of slider which means that for every pixel movement of the slider an event is sent to the server. Also lazy sliders exist that only send a notification to the server when the user stops dragging the slider. We are only interested in greedy sliders because these generate a lot of network traffic. After a server event occurred the counter in the lower right corner gets updated.
7. On the end the user closes the remote user interface.

All this is done on a screen size of 640x480. For making the test results comparable with each other we used a tool called WinMacro [Kum]. WinMacro enables the recording and replaying of user events like mousemovements and keypresses. By using WinMacro exactly the same user actions can be replayed for each test making the results more comparable.

2.3 Existing markup content formats to test

This section describes how we have implemented the test cases and what problems we faced when creating the test case. For a detailed description of these content formats see section 1.4 on page 4.

2.3.1 XHTML

XHTML by itself is just a document markup language and inefficient for remote UI because for example fragmental updates can not be described; we need the DOM model for that. Also absolute positioning was in some cases needed; in XHTML you need CSS level 2 for absolute positioning. So we needed extensions for XHTML.

So from now on we will use the term XHTML to indicate the follwing terminoligies.

- XHTML 1.0 specification [W3C00a]
- ECMAScript language specification, 3rd Ed [oai0]
- Cascading Style Sheets (CSS) level 1 [W3Ca]
- Cascading Style Sheets (CSS) level 2 [W3Ca]
- Document Object Model (DOM) level 1 [W3Cb]
- Document Object Model (DOM) level 2 Core [W3Cb]
- Document Object Model (DOM) level 2 Events [W3Cb]
- HTTP 1.1 [oai96c]
- TCP/IP

Clients that we have tested with

- Internet Explorer 6.0
- Opera 7.54u2
- Netscape 7.2
- Mozilla Firefox 1.0

Building the test case

Differences of interpretation between different browsers As already described in section 1.4.2 it is difficult to create a remote user interface in XHTML that looks and operates the same in different brands of browsers.

Differences in browsers Spacing There are differences between browsers in how spacing is managed in frames and tables. In Internet Explorer there is space reserved for a scroll bar in the right of the screen, this is not the case in other browsers like for example Opera. This inconsistency is solved by specifying the parameter scrolling="no". In tables also inconsistencies are found between browsers. If the valign property is used in Internet Explorer to vertical align content in a cell the content is vertical aligned to the bottom of the cell. In Opera however there is still a lot of space left between the items in the cell and the bottom of the cell. Until now we did not find out how to remove this extra space in Opera; we simply did not use valign anymore which solved our problem.

Scripting JavaScript is interpreted differently in different browsers as already explained in section 1.4.2. Also browsers tend to implement their own specific functions that only work when using the specific browser brand which gives interoperability problems when the site is run on other browsers. We don't use these functions and only use the functions declared in the original specifications.

Adaptability to screen resize XHTML is adaptable to layout changes. It does not scale its user interface but scales on empty space that is free in the user interface. This scaling behavior however has to be created by the developer. Also XHTML often uses scrollbars if something does not fit in the available space. Because scrolling, however, has huge usability problems scrolling has to be avoided, therefore it is needed to create user interfaces for different devices.

CSS level 2 If a designer wants to animate objects that are moving over the screen on the client side it is required to have the CSS level 2 property position:absolute. This behavior is almost not possible in CSS level 1. Also we needed the cursor attribute that is defined in CSS 2 to change the cursor. The cursor attribute can help with giving feedback to the user by e.g. changing the mouse pointer to notify the user that he or she can click on the element. So CSS level 2 is a requirement that we need for implementing our test case.

DOM level 0: setInterval For defining the client side animations we also need functions like setInterval or setTimout. These methods are implemented in a lot of browsers, but are not defined in any specification of HTML, XHTML or CSS. Only DOM level 0 refers to these methods but this is not really a specification. DOM level 0 contains a number of methods that are defined in most browsers. Without these methods it is not possible to animate an element over time; this is required because Philips wants to provide nice animated user interfaces with their products.

DOM level 2 event model In some situations we also need to know the exact position of the mouse (e.g. for sliders and drag drop behavior). This is possible with the event object that is introduced in DOM level 2.

DOM implementation in Internet Explorer 6.0 Internet Explorer 6.0 is not a full implementation of DOM level 2. Below we mention some of the differences between the implementation of Internet Explorer and the DOM level 2 specification that we encountered while creating the test case.

- When dynamically generating a table by the document.createElement statement Internet Explorer requires us to create a tbody element. In the HTML 4.01 this is not required for creating a table and also other browsers do not require this.
- The AddEventListener of DOM level 2 is not implemented. There is an Internet Explorer specific method, however, called attachEvent that provides almost the same functionality.
- The event object of DOM level 2 does not have a target property in Internet Explorer. There is however a property called srcElement that contains the target of the event.

For these points it was needed to write Internet Explorer exceptions. All other tested browsers just worked as expected when following the DOM specifications.

2.3.2 SVG

For SVG we create two test cases, one for SVG Basic that only uses gif and jpg images for its user interface and a SVG test case that replaces some images by SVG shapes and gradient effects. We also needed extensions for SVG; we needed the DOM model for describing fragmental updates.

From now on we will use the term SVG to indicate the following terminologies

- SVG Basic 1.1 [W3C03]
- ECMAScript language specification, 3rd Ed [oai0]
- Document Object Model (DOM) level 1 [W3Cb]
- Document Object Model (DOM) level 2 Core [W3Cb]
- Document Object Model (DOM) level 2 Events [W3Cb]
- HTTP 1.1 [oai96c]
- TCP/IP

We tested the following client with the SVG Basic implementation

- Adobe SVG viewer 3.0

Currently we do not have a good dedicated SVG browser to test with; all good dedicated SVG browsers for resource constrained devices are currently commercial products, that where hard to get our hands on because of license issues, and mostly only support SVG Tiny 1.1 which is insuficient as discussed in section 2.3.2 on page 18. SVG Tiny 1.2 is much better usable for remote UI because it supports scripting and gradients, however, SVG Tiny 1.2, when writing this document, was still a working draft.

Building the test case

Positioning the graphical objects within SVG is simple because SVG uses absolute positioning. In XHTML tables and transparent images are often required to position elements or texts.

Also animations are more easily described in SVG than in XHTML; in SVG it is possible to describe animations in a declarative way which is more compact. Also declarative animations are easier to optimize at the client side than scripting. Declarative animations are described in section 2.4.1.

SVG Tiny We first tried to create a test case in SVG Tiny. We concluded however that SVG Tiny is simply too tiny to use for generating user interfaces. Within SVG we need scripting to allow server eventing which is a requirement for remote user interfaces; scripting is not included in SVG Tiny. Furthermore we need scripting for fragmental updating so that our test cases become comparable to the XHTML and VNC test cases. If we do not implement fragmental updates by using DOM level 1 scripts [W3Cb] then we are restricted to full

screen updates whenever something changes on the screen. Full screen updates are however not done in VNC; VNC only sends the changed fragment of the screen. To make SVG comparable to VNC we need the same kind of behavior, so fragmental updates.

Shortcomings of SVG Tiny

- Server eventing is not possible, needs atbest some form of scripting.
- Server requests are only possible by using the <A> element. The <A> element is used for defining hyperlinks, it is unsuitable, for example sending a play request to the server; this will result in a total page reload.
- Style elements are only available in SVG Basic, not in SVG Tiny. It is however still possible to use the style attribute, but it is more elegant to separate styling from the layout.

SVG Basic In the SVG Basic implementation we managed to get some sort of the same behavior of fragmental updates and server eventing as in XHTML by using scripting, DOM level 1, and the getURL function. The getURL function is not in any specification, however, many SVG viewers have implemented the getURL function. By using the getURL function we have an SVG implementation that is comparable in functionality to XHTML and VNC.

Shortcomings of SVG Basic

- Programming fragments by making use of DOM makes code not very clear. When we for example want to describe a rectangle that we need to add to our SVG description we need around 15 lines of ECMAScript code.
- We miss a function like getURL in the default specification. If a SVG viewer is fully implemented following the SVG specification it still does not guarantee that our remote user interface will work correctly.

2.3.3 VNC

VNC is interesting for testing to see the resources needed for a framebuffer level protocol. We are interested in how resource consumption relates to protocols like SVG and XHTML. Does VNC need twice the bandwidth of SVG or is the bandwidth needed multiplied by 10?

VNC can be described just like XHTML and SVG as a content format, however, VNC does not have an own user interface description language. For VNC we still need an extra application that renders a user interface. For testing VNC we used the XHTML test case to render the user interface. Furthermore VNC supports compression techniques like, RRE (Rise and Run-length Encoding), CoRRE (Compact RRE), Hextile (variant of CoRRE), and ZRLE (Zlib Run-Length Encoding). For a more detailed description of these compression techniques see [Ric05].

We tested the following client

- RealVNC client version 4 [Ric05]

At the server side we tested the following servers

- RealVNC server version 4 [Ric05]

2.4 Test results

Within this chapter we will first presents and explain the client test results , after that we will present the server test results, and in the end we will present the bandwidth measurements. To see how testing is actually done and how the test environment was set up see appendix A on page 69.

2.4.1 Client performance/resource consumption

We first describe some observations that were visually noticeable while running the tests.

Visual observations

IE (Internet Explorer) behaves mostly like intended, only when using the slider the response was not good; the screen blinks and the granularity of the slider is not good. Also IE (but also other clients) closes sockets in the middle of HTTP requests without finishing the total request: resulting in incomplete HTTP requests that arrive at the server. Closing sockets in the middle of the request is allowed within the HTTP 1.1 specification [oai96c] and a server must ignore these requests. Closing sockets can be seen, however, as an optimization; only the current position will be sent, older requests will be canceled when not finished. Closing a socket on a subsequent action is not always wanted; e.g. in editing a playlist it is undesirable that adding a song to the playlist fails if a second song is added while the request for the first song is still not finished. There is an option missing that allows to set the behavior of how to deal with subsequent action requests. Other tested clients just send and finish all actions (all slider updates), this also gives problems; with the greedy slider this can result in a not very responsive slider. If we drag a slider from position 0 to 100 than 100 client events (100 round trips to the server) need to be processed before the slider reaches its final destination which results in a very slowly performing slider. This was especially noticeable in the SVG Adobe plugin.

XHTML clients displayed the large animation fluently. The animation is described as an animation that should take 1,02 seconds, it however takes XHTML clients around 4 seconds. SVG clients handled this large animation fluently in 1 second. Within VNC large animations perform very poorly, within a large animation the screen only showed a few updates resulting in a flickering remote user interface that is not really presentable to a user. In fact the total performance of the user interface in VNC was very bad; a lot of hiccups in animations, even the small ones.

Processing resources XHTML

In figure 2.2 on page 22 the processing times of the different XHTML clients are displayed. On the Y axis the % processing time is shown and the X axis shows the timeline of the test. Within this section we describe the results of the different actions that are done over time.

- From 0:00:02 till 0:00:09 is used for starting up client process and rendering the remote UI. In the diagram IE uses the least resources but takes the same time to start up as others. FireFox and Netscape use double the processing resources in comparison to IE.

- From 0:00:11 till 0:00:40 small and large animations are rendered for selecting the media browser and navigating the menu. From 0:00:20 till 0:00:26 we described an animated area that moves 255 pixels and does this by moving 1 pixel at a time every 4ms. This means that the animation should be fully executed in 255 * 4 = 1020 ms = 1,02 s but it takes XHTML clients around 4s. A solution can be to raise the steps of the test case from 1 pixel to 4 pixels resulting in a faster animation, however, this is not comparable to the animation that SVG provides.

  Almost all XHTML browsers (except IE) use 100% processing power to render the animation on a Intel Pentium 4 1.7 GHz processor, which is not executable on almost any mobile device which is not acceptable.

- From 0:00:40 till 0:00:54 is used for adding a song to the playlist and playing the song. All clients perform the same, they don't use much processing power for this testing phase and we think that this is acceptable.

- From 0:00:54 till 0:01:04 is used to add one song and drag the greedy slider. Here IE has the worst processor resource consumption and compared with Opera takes more than triple of the processing time. The results of IE are in this case not really acceptable. The remote UI in IE also flickered and gives a bad visual impression. The other XHTML browser displayed the slider as intended. Opera came out as best in this part of the test (Opera used the least resources).

- From 0:01:09 till 0:01:13 is used to close the client. Netscape and FireFox take more processing resources than Opera and IE.

When studying the processor resource consumption the conclusion can be made that IE uses the least processor resources. However we suspect that the graphical processor on the video card is used. We also tested IE on PC's with not very good video cards and we concluded out of these test results that IE uses around the same amount of resources as other XHTML clients. So within the current test results IE uses processing resources that the other clients do not use. This makes IE not really comparable to the others and can give totally different results when used on a mobile device without a dedicated graphical processor. Currently we take the Opera client as best client for XHTML for mobile devices when looking only at the processor consumption of the client.

Processing resources SVG and VNC

Figure 2.5 on page 25 shows the results of executing the same test case on SVG and VNC clients. For SVG we did two tests. One test that used the same images as the XHTML test and one test that replaced some of the images by using SVG vector grapics description with some gradient effects which resulted in our case in the same user interface. Also we included the client resources of Opera so that these results are visually comparable to the XHTML results.

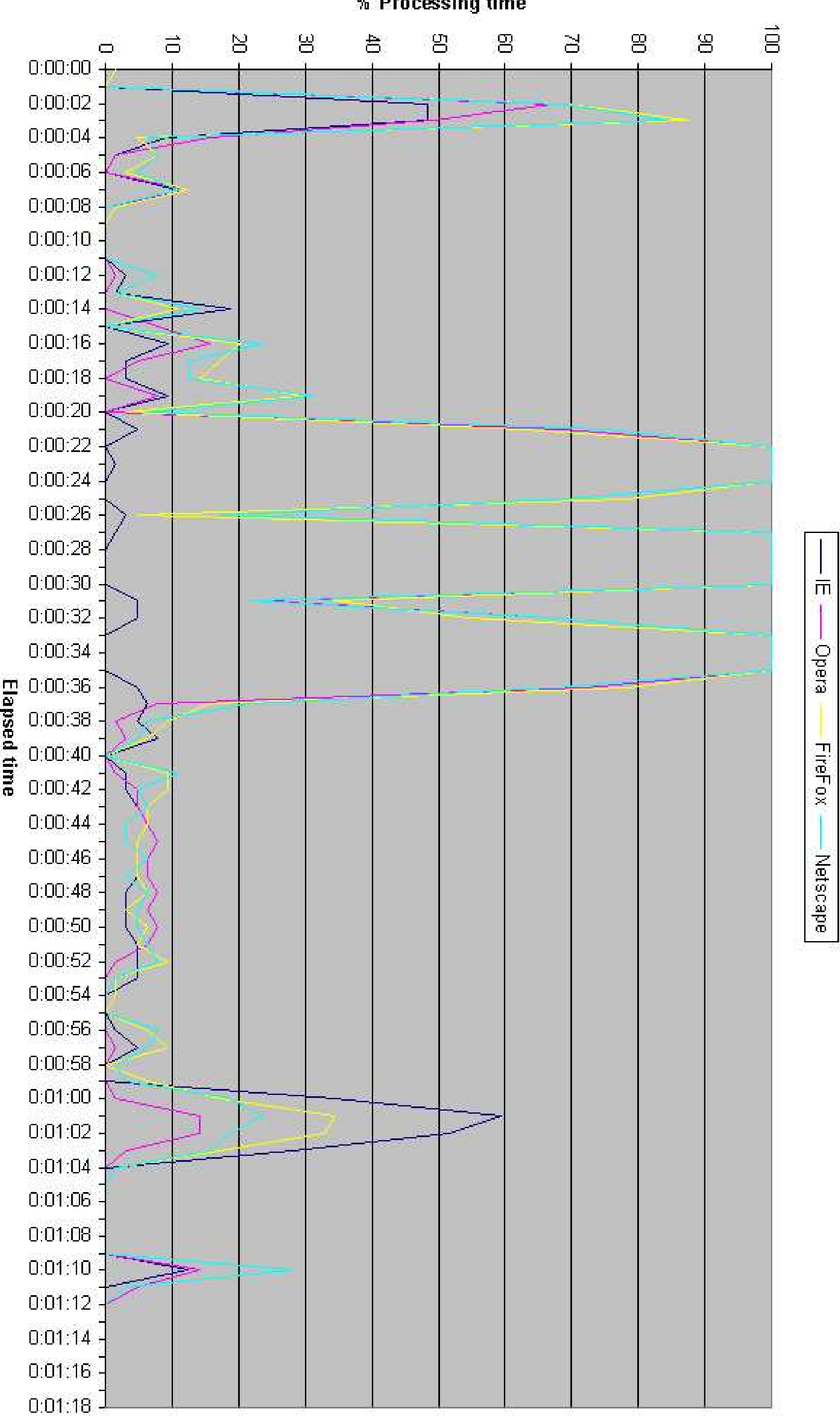

Figure 2.2: Processing time XHTML clients

```
<script type="text/javascript">
    var currentPos=0;
    var timerInterval;
    function highliteFolder(evt) {
        timerInterval = window.setInterval("moveHighlite()",555);
    }

    function moveHighlite() {
        currentPos = currentPos + 5;

        if(currentPos >90)
            window.clearInterval(timerInterval);

        document.getElementById("rect").style.top=currentPos + 'px';
    }
<\script>
<img id="rect" width="10" height="10" src="rect.gif" onclick="highliteFolder()"
style="position:absolute" \>
```

Figure 2.3: XHTML scripting example

If we look at the two SVG test cases then we don't see a large difference in the processing resources consumed. The rendered version seems to consume the same amount of processing resources as the non rendered test, something which we did not expect. We expected that the processing costs for rendering an image out of an image encoded description costs fewer resources than rendering images out of a declarative SVG description of an image. This however also depends on the complexity of the image, if an image is very complex then rendering this image using declarative vector graphics costs a lot of effort and probably will cost much more processing resources then rendering a bitmap.

SVG compared to Opera consumes fewer processing power at the client side. Starting up the client takes fewer processing resources; which is strange because also IE has to start up; the SVG test is done with the Adobe plugin for IE. Also there is a large difference when executing animations. The visual result is comparable but it is difficult to say if SVG also moves pixel by pixel or takes larger steps. Within SVG we used a declarative animation that uses a linear calculation for calculating an animation of 1 second from x coordinate 255 to x coordinate 0.

That resources needed for large animations in SVG are lower than in XHTML is probably because we use declarative animations. Declarative animations allow clients to choose how they animate an object based on the resources that are currently available. To clarify this statement figure 2.3 shows an scripting example of an animation in XHTML and 2.4 shows a declarative animation example. Not only is the declarative description much more compact, but it can also be much more optimized based on the client resources as described in chapter 6 on page 59.

If we look at the processor resource consumption of VNC in figure 2.5 we see

```
<rect x=”45” width=”10” height=”10” fill=”red”>
    <animate attributeName=”y” from=”0” to=”90” dur=”10s”\>
<\rect>
```

Figure 2.4: SVG declarative animation example

that it is very low. A VNC client only needs to draw image data on the screen, nothing more. The graphical result, however, was very poor and all animations only show a few frames: the reason is a lack of server and network resources, see section 2.4.2 on page 24.

Memory consumption

Within appendix A on page 69 we included a detailed description of how the memory was measured and it also contains charts of the results.

VNC uses the least memory at the client side (around 3,5MB) and IE (around 9,5MB) is in between VNC and the rest (around 15MB). Probably the memory consumption can be much lower if we use a client dedicated for mobile devices. We don't think that memory is really a problem.

Client resource consumption conclusion

Although the resource consumption of VNC did not use a lot of processor and memory resources we do not think VNC came best out of the test with respect to client performance. Our main goal is to provide a nicely performing remote user interface. Something VNC lacked because animations where very slow and annoying; the responsiveness of the user interface was very slow. The reason why VNC was very slow is explained in the next section.

We think SVG came out as best on the client side because it provided a nice looking and performing user interface and took less processing resources than XHTML. The memory consumption, however, of SVG is too high. This is, however, maybe much better in a dedicated SVG browser.

Describing a user interface in XHTML resulted in a slightly less smootly animating user interface and uses more resources in comparison to SVG. A good feature of XHTML, which is not available in SVG, are widgets.

2.4.2 Server performance/resource consumption

In this section we present and explain the test results at the server side.

Processing resources

Figure 2.6 shows the processing resource consumption at the server side. In this figure you see a lot of similarities between the test cases. This is because the same server software is used for the tests (ie a handmade Java webserver). The only difference is around 0:01:06; SVG uses here more processing resources than the XHTML variants. The SVG Adobe plugin handles all HTTP requests and does not ignore the results or closes the socket when a new server request

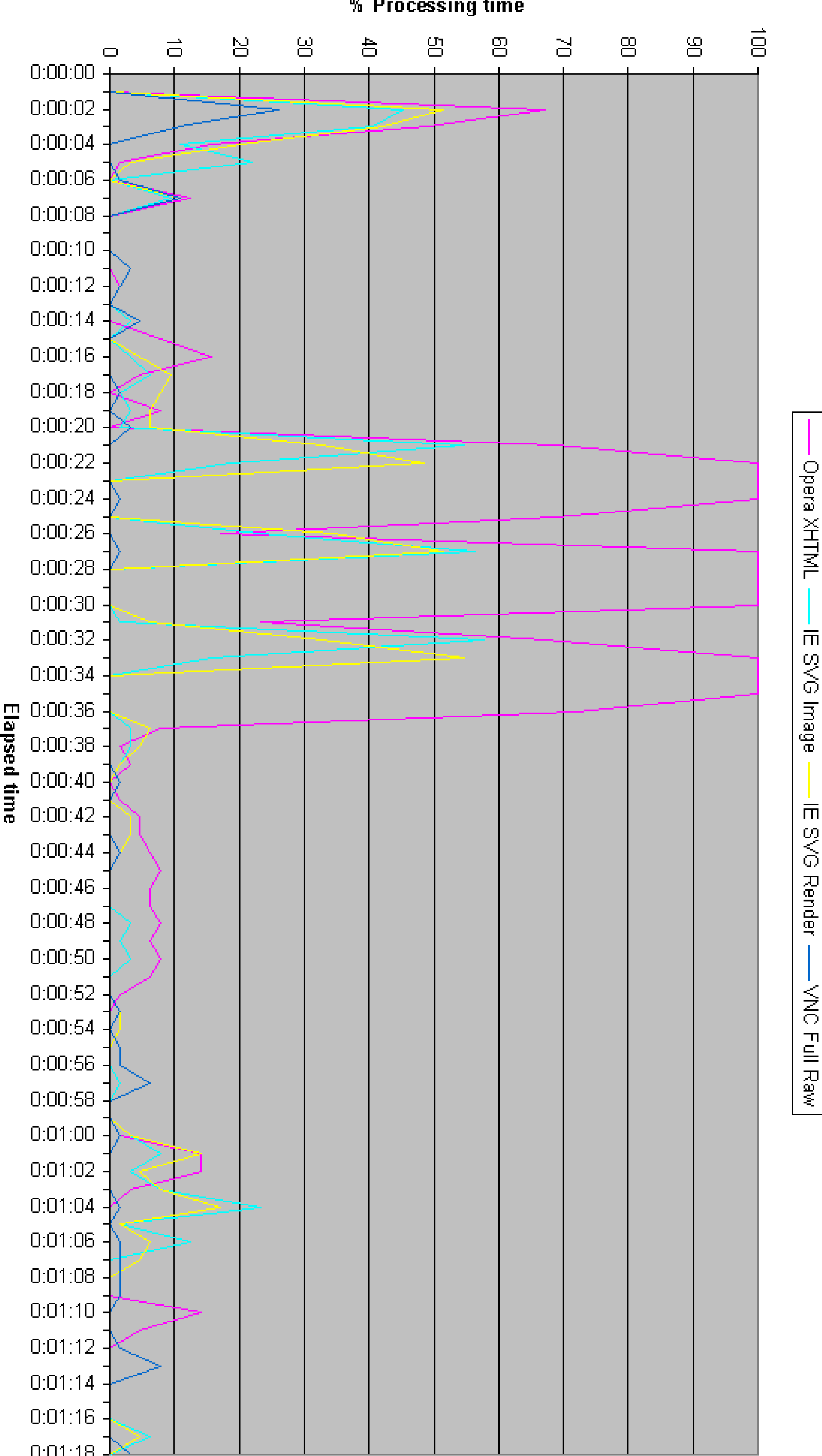

Figure 2.5: Processing time SVG and VNC clients

comes in: this results in that SVG takes more time and resources for operating the slider.

The only test case that differs from the others at the server side is VNC. All processing resources are needed at the server side to run the test; we can however not blame it all on VNC. For testing VNC at the server side we also need a viewer that renders the user interface so that VNC can generate an image out of it and send this image to the client. But still a user interface rendered at the server side is always needed when using VNC for remote UI: VNC cannot render a user interface by itself so we also need to include the processing resources needed for rendering a user interface. For this test we used the FireFox browser at the server side. It would be nice to isolate the resource consumption of VNC. Isolating the resource consumption, however, is in the current situation not really possible because VNC takes 100% of the processing power. So if we blindly remove the processing consumption of FireFox (figure 2.2 on page 22) out of the results of VNC we can conclude that animating large images costs VNC 0% of extra processing power which is of course not correct. The test case for VNC is, however, not the most optimal. We used a VNC server that by placing hooks in the operating system can detect screen changes. A VNC client that directly hooks within the FireFox code and renders on the VNC framebuffer instead of the screen can lower the resouces at server side.

For the rest we think that the processing resource consumption at the server side is not very disturbing when looking at XHTML and SVG, even without any optimizations. The current processing resource consumption of VNC at the server side is however not acceptable. VNC needs to do almost all work on server side. When multiple clients, with different screen resolutions, connect to the server multiple user interfaces have to be fully rendered: resulting in a lot of resources consumed what means that we need a very heavy server side.

Memory consumption

The VNC test case took the most memory, around 21MB in total, the FireFox client used 18MB and VNC 3MB. VNC needs to keep a framebuffer in memory which in our case will be

$$640 \times 480 \times 4 = 1.228.800$$

bytes (4 bytes for color) which is not needed when using SVG/XHTML. The other test cases took around 5MB of memory. They use the same server software, that is written in Java. Java runs within a virtual machine which is not optimal. Probably the memory consumption will be much lower if a programming language like C++ is used.

We think that the memory consumption for XHTML and SVG is not a bottleneck for remote user interfaces especially when implemented by an optimized webserver that probably uses less memory.

Server resource conclusion

Only VNC has high service resource requirements. XHTML and SVG do not require a lot of server resources and if one only looks at the server resources XHTML and SVG are almost equal. We assume that the resource consumption

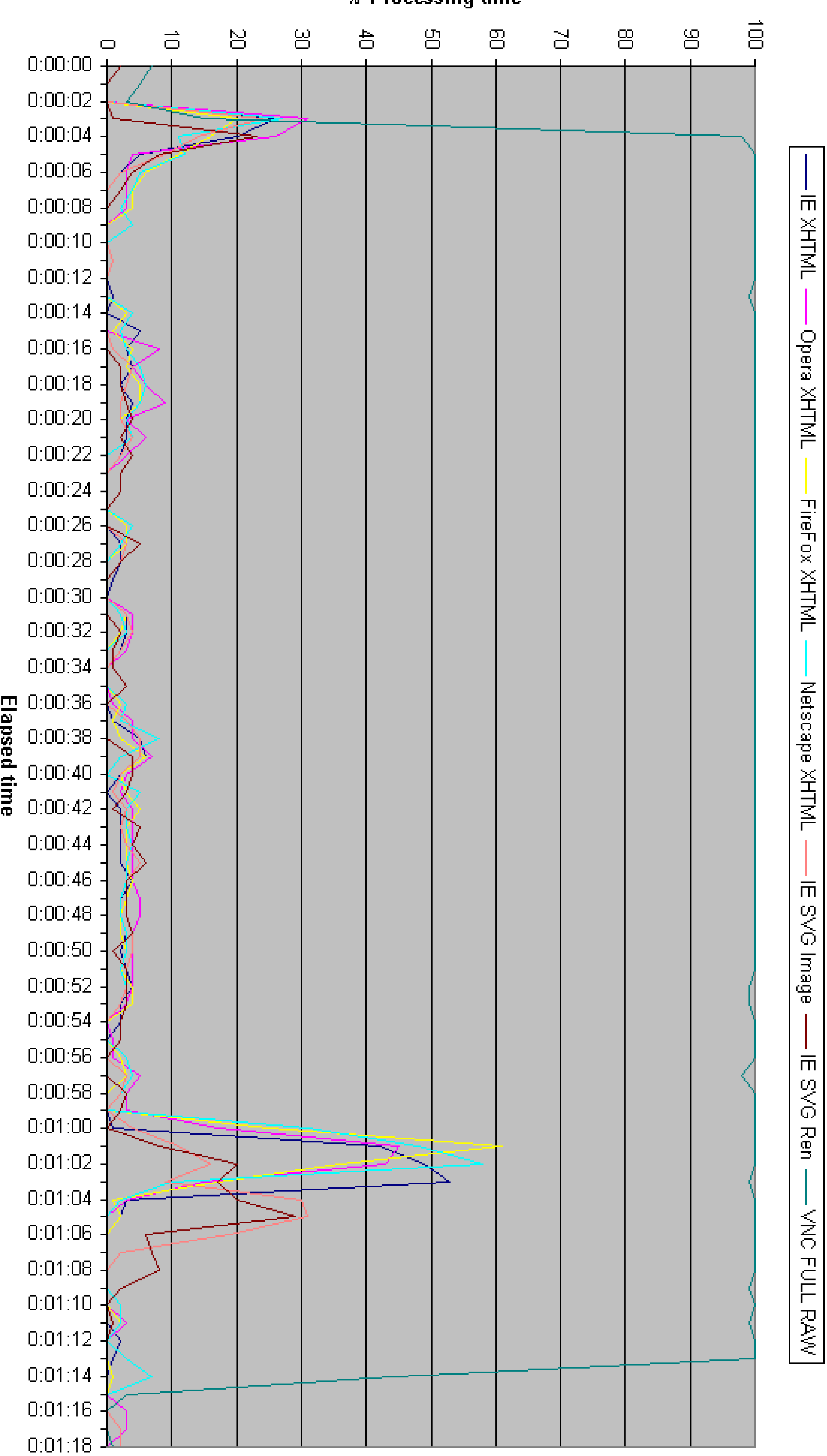

Figure 2.6: Processing time used at the server side

at the server side for XHTML and SVG is executable on resource-constrained devices, even without any optimizations at the server side.

2.4.3 Bandwidth usage

In figure 2.7 we show the bandwidth consumption of the different clients/protocols. The bandwidth consumption of VNC shows a large difference with other protocols, it was even not possible to show the full bandwidth line of VNC because then the other bandwidth lines would be almost flat (VNC has peeks of 1455372 bytes $\simeq$ 1,3 MB within one second). Of course this can be much better with optimizations like compression, but compression also adds extra processor resources at the server and the client side. At the server side this is a large problem because VNC already requires a lot of processor resources. Also reducing the amount of colors can solve the problem. We also tested with 256 colors, but this is just too low for displaying a nice user interface so also not an option for solving the bandwidth problem. 16 bits colors can be a good choice, the implementation that we used however did not provide this color mode. Still with 16 bits colors we get peeks of 1,3 / 2 = 0,65MB what is much more than the other test cases.

Pos.	Client name	Total bytes	Average bytes/s	Max bytes/s
1:	IE SVG Render	191.496	2.775,3	23.978
2:	IE SVG Image	223.563	3.240,0	40.394
3:	Opera XHTML	239.715	3.474,1	44.144
4:	IE XHTML	251.698	3.647,8	32.172
5:	Netscape	266.202	3.858,0	58.044
6:	FireFox XHTML	271.610	3.936,4	54.519
7:	VNC	3.593.154	52.074,7	1.455.372

Table 2.1: Bandwidth results

Figure 2.1 shows a table with bandwidth measurements. IE SVG Render comes best out of the bandwidth results; this test case needs fewer image files because some are described in SVG basic shapes and gradient effects. SVG and XHTML, however, do not really differ that much. VNC however uses too much bandwidth and will lead to performance problems on wireless networks. In our test cases we also have situations where a remote user interface has to be shown while watching, for example, a streaming movie what will give problems when using VNC. The differences between the XHTML clients is the result of how client events are handled; some clients close the socket earlier when using the greedy slider than others resulting in a difference in bandwidth consumption.

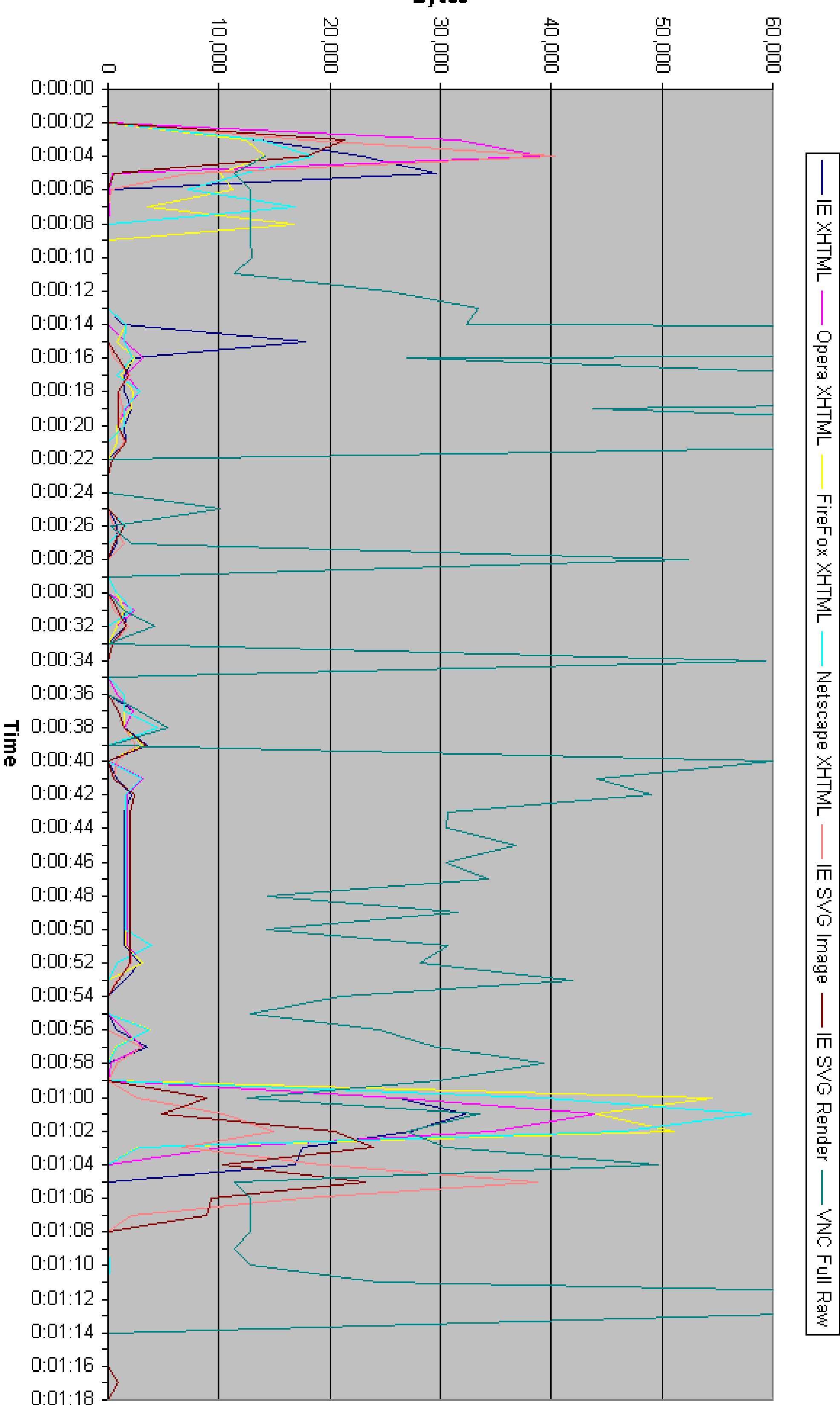

Figure 2.7: Bandwidth used

2.5 Conclusions

The test results that we presented here show that for remote user interfaces VNC is not very suitable. The main bottleneck for VNC is the server processing resources as indicated in the previous section. One of the requirements is that the solution must be able to run on resource constrained servers, this is not able with VNC. We think VNC is out of balance to be a suitable candidate for a remote user interface; a very heavy server is required, and a very thin client can be used. In our use cases we also have thin servers with low bandwidth, a use case that cannot be solved with VNC.

XHTML is in resource consumption almost comparable to SVG. Bandwidth needed for XHTML is a little higher, but not significant. Resources needed at the server side are equal. At the client side, however, there are some differences.

Advantages of SVG in comparison to XHTML

- Resource consumption of SVG is significantly less than XHTML when executing animations, probably because of declarative animations.
- Gives more freedom in specifying the user interface because it is more graphical based, XHTML is more text base (Hyper**Text** Markup language). XHTML gives a restricted feeling when creating a user interface because all kind of backdoors need to be used to achieve the intended results.
- Less scripting is required because more can be done declarative (saves processing resources client).
- Graphics can be described in SVG syntax and fewer images are needed , which saves bandwidth

Advantages of XHTML in comparison to SVG

- XHTML supports widgets, see page **??** for advantages.

Our conclusion based on testing is that SVG is the best choice to start from for making a new remote user interface standard protocol.

We, however, see some key problems with resource consumption within SVG that can be solved to provide a better quality of service to the user. In the next sections we explain the key areas that give problems on resource constrained devices.

Chapter 3

Next steps

The processor consumption on the client side is high. There are however already many devices with an SVG implementation; at the moment of writing this document 52 mobile phones are on the market that contain a SVG Tiny implementation (brands that produce phones that support SVG are Motorola, NEC, Nokia, Panasonic, Sagem, Sanyo, Sharp, Siemens and Sony) showing that SVG can run on resource constrained devices.

In combination with remote UI we still see some key problems that need optimization so that remote UI can perform better.

At the client side we identified the following problems

- At startup we see a peek of processing resources needed to load the remote UI description, this results in that clients have to wait longer on resource constrained devices before the remote UI is started.
- Large animations consume a lot of resources: this can result in bad performing animations on resource constrained clients
- Greedy sliders are not very responsive; they are only set to the correct position when the user drops the slider.

Also we described that SVG Tiny is simply too tiny for Remote UI. We need SVG Basic for defining remote UI which takes some extra resources at the client side.

3.1 Identified problems bandwidth consumption

Bandwidth consumption for remote UI is not very high. We however also have to deal with situations where almost no bandwidth is available. If we define a media browser user interface and within this interface we are watching a streaming movie we do not want to lower the quality of the movie because the user interface consumes bandwidth. Our goal is to make the bandwidth consumption as low as possible. Reducing resource consumption at the client side is however more important.

Identified problems for bandwidth consumption

- At startup we have a peek in our bandwidth measurement that is the result of the user interface definition that is sent over.
- When using a greedy slider we have a large peek that is the result of a lot of messages that are sent to notify the server that the current play position needs to be changed.

3.2 Identified problems at the server side

At the server side the resource consumption is not very high compared to the client side. We, however, still can try to reduce this. Also notice that these tests are done on a PC that has a lot of processing resources; the target devices like music players, DVD players, mobile phones, etc, do not have a processor comparable to that used in a PC.

Identified problems server side

- At startup we see a small peek that is the result of sending the remote user interface definition to the client
- When using the greedy slider we see a peek that is the result of handling a lot of requests for setting the current play position.

3.3 What to do next

We take SVG as starting position and in the next chapters we will present some optimizations that can be applied to SVG to make it more suitable for describing remote user interfaces. These optimizations can, however, also be applied to other XML based remote user interface protocols like for example XHTML.

Chapter 4

Optimization 1: Client eventing

4.1 Introduction

The first optimization technique is what we call client eventing. Client eventing is a technique that saves bandwidth and client/server processing resources. This is achieved by specifying the events that need to be sent to the server in more detailThe client can use this extra information to handle the event more optimally.

An example; a user interface contains a slider that notifies for each movement the server. If there is a high latency on the network it can take some time to send a message to the server. If we , however, move the slider 5 places to the left 5 messages to the server are sent. In the current situation we are only interested in that the last message is fully handled by the server and all previously sent messages can be canceled. Client eventing gives the ability to specify this kind of behavior.

Canceling messages that are currently being sent to the server is not always a good solution; sometimes it is needed to send all messages. An example is the functionality of adding songs to a playlist. If a user adds two songs to the playlist and the first message to the server is still running when the second song is added it is not correct if the first message is canceled.

Furthermore client eventing gives queuing support for messages; it is possible to specify that messages to the server need to be queued for 1 second before sent: this results in less network congestion.

We specified client eventing as a namespace extension in XML that can be used within for example XHTML or SVG.

Also we included our test results on an implementation of client eventing in this chapter and compared it to applications that do not use client eventing (see section 4.6).

```
<cle:sendAction begin="ellipse.click" name="Play"
        xmlns:cle="http://www.philips.com/remoteui/clientEventing">
    <cle:argument name="InstanceID" value="1"/>
    <cle:argument name="Speed" value="1"/>
</cle:sendAction>
```

Figure 4.1: Client eventing example

4.2 At the client side

The sendAction element is within client eventing the most important element because it is used to declare client events. When triggered sendAction sends a HTTP POST request to a webserver describing the client event that was triggered.

The XML structure in figure 4.1 describes an client event, the sendAction element marks the start of a client event declaration. The begin attribute within the sendAction element describes when the client event needs to be sent. In the above example we have a client event that is sent when a user clicks on an element that has as id attribute with the value "ellipse". The name of the event is stored within the name argument of the sendAction element which in the current example is the value Play. As childs of the sendAction element we can add optional elements with as element name "argument" and these elements contain values that are also sent to the server when the client event is triggered. The exact format of the client event that is sent to the server is described in section 4.5.

4.3 The proposed structure in detail

Here we present our prefered declarative XML structure for implementing client eventing. Client eventing, however, can also be implemented by, for example, adding optional arguments to the getURL statement.

4.3.1 sendAction element

This is the main element for defining a client side action. Figure 4.2 shows the DTD that describes the structure and the possible arguments that the sendAction element can have.

The sendAction arguments

- begin: CDATA | "indefinite"

  The begin attribute specifies when the action needs to be sent to the user. This can be on a mouseclick or on a mouseover event as long as it is a known DOM type event (see DOM level 2 events specification [W3Cb]). This DOM type can be preceded with an identifier for an element that will be the eventTarget. The "." symbol will be used

```
<!ELEMENT sendAction (argument*) >
<!ATTLIST sendAction
begin (CDATA |”indefinite”) ”indefinite”
onbegin CDATA #IMPLIED
onend CDATA #IMPLIED
disabled (”true” |”false”) ”false”
caching (”true” |”false”) ”false”
name CDATA #REQUIRED
onNextAction (”sendOne” |”sendAll”) ”sendAll”
queueTime CDATA ”0s”
displayCancelButton (CDATA — ”never” — ”always”) ”5s”
timeout (CDATA |”never”) ”30s”
timeoutAction (CDATA |”cancelRequest” |”alertUser”) ”alertUser”
controlPage CDATA ”clientControl”
>
```

Figure 4.2: sendAction element DTD

as separator between the event target and the event type (e.g. button.click; button is here the event target where to listen on, click will be the event type to listen for). The default value is ”indefinite” which means that the action can only be triggered by scripting.

- onbegin: CDATA
  Defines an optional pre-processing function that needs to be executed before an action is sent (e.g. disable the button so that the user can not click on it for a second time). The pre-processing function is the last function that can change the client action before it is sent.

- onend: CDATA
  An optional argument that can contain a post-processing function that gets called after the action is sent and processed by the server. As argument to this function a data structure is passed that can be used to read the response from the server. This data structure contains data.succeed which is a boolean that describes if the action succeeded and data.content which contains the result string that is sent back by the server in the HTTP body.

- disabled: ”true” | ”false”
  This optional argument can be handy to temporally disable the client events. The default value is ”false”

- cacheResponse: ”true” | ”false”
  Depending on the argument values certain responses from the server can be cached resulting in fewer requests that have to be sent to the server if multiple times the same information is requested. Caching is turned off as default but can be handy in some situations. Also in scripting the cache can be cleared by calling clearCache on the element.

- name: CDATA
  A required field that specifies the action name that needs to be sent to the server. The value of this field is used within the HTTP message sent to the server as described in section 4.5.

- onNextAction: "sendOne" | "sendAll"
  If an action is sent multiple times it can happen that a previous action with the same name is not finished yet. Within the onNextAction argument it is possible to specify the action that must be taken when a previous action is not finished yet. The default value is "sendAll" which means that all actions will be sent. The other option is "sendOne" that depending on the argument values makes a decision if the previous message is being canceled or the new message is being ignored. If the argument values of the client events are the same then the new message can be ignored, if the argument values of the message are different then the message that is currently being sent is canceled and the new message is sent.

- queueTime: CDATA
  This value makes it possible to specify a time value that indirectly specifies the maximum amount of actions that are sent. The only syntax that is accepted for this value is represented in figure 4.3. The implied value for this argument is "0s" which means that messages are immediately sent. When a different value is specified like "1s" then the sent action element only sends every second at most one message to the server. So when no action is sent for the last second and a new action needs to be sent then this action is sent right away. If after this action a new action needs to be sent then this action gets queued until one second is passed.

- displayCancelButton: CDATA | "never"
  This attribute specifies if the user needs the ability to cancel the action. This must not be implemented as a message box but shown to the user in a more gentle way, for example somewhere in the progress bar. Only clock-values (see figure 4.3 for syntax) are allowed in the CDATA. The implied value for this attribute is "5s" which delays the notification with cancel button for 5 seconds before the user sees it. If the value is "never" the cancel option is never shown to the user.

- timeout: CDATA | "never"
  Specifies how much time to wait before the action times out. The implied value for this argument is "30s" and CDATA can only hold clock-values like specified in figure 4.3. If "never" is specified the action never times out (see also timeoutAction).

- timeoutAction: CDATA | "cancelRequest" | "alertUser"
  Defines the action that needs to be taken when a timeout occurs. The default value is "alertUser" which means that the user gets an alert by means of a message box that the action is timed out. A different option is "cancelRequest" that ends the request without informing the user. The third option is to specify a function defined in scripting that gets called when an action times out.

```
Clock-val  ::= Timecount (Metric)?
Metric     ::= "h" |"min" |"s" |"ms"
Timecount::= DIGIT+
DIGIT      ::= [0-9]
```

Figure 4.3: Syntax of clock values

```
<!ELEMENT argument PCDATA>
<!ATTLIST argument
name CDATA #IMPLIED
value CDATA #REQUIRED
>
```

Figure 4.4: DTD argument element

- controlPage: CDATA
  Specifies to which server page the action is sent, when not specified this argument defaults to "clientControl". See section 4.5 for more information about this argument.

4.3.2 The argument element

The argument element is stored as a child of an sendAction element and contains the argument values that must be sent to the server. It is not required to include an argument in an action; it is optional. Figure 4.4 shows the DTD definition for an argument element.

The arguments

- name: CDATA
  A value that contains the name of the argument. This value is only used as documentation.

- value: CDATA
  The value of the argument; this is a required argument. Within the value also a scripting function can be stored that gets called before the client event is sent and the returning value is sent within the client event.

4.3.3 DOM extension

clearCache(): This function is added to clear the values stored in the cache from scripting. It gives more freedom to the programmer to select the exact moment that a cache has to be cleared.

4.4 Examples

Figure 4.5 shows some examples of the client events that can be declared within the client eventing structure.

Here the explanation for each example in figure 4.5

Example 1. Shows an implementation of a client action that is sent to the server when the play button is clicked. The onNextAction is set to "sendOne" because if the user clicks the play button multiple times the play action has to be sent only once to the server.

Example 2. Shows the client event for a pause event. The pause event uses the same event name as the play event resulting in that they share the same queue for there messages however they have different argument values. So if the play action is currently being sent, and the pause button is clicked, then the play action is canceled. If the pause action is clicked multiple times then only the first pause action is sent to the client, the other actions are ignored.

Example 3. Shows an example of a slider event. The begin attribute is set to indefinite because the client event is initiated by client scripting; it needs to be sent when the slider is dragged which is difficult to describe with the current DOM level 2 events [W3Cb]. We need client side scripting to trigger the action. Furthermore we also defined the onend attribute that contains a client side script that is called when the event finishes. This function is needed to set the position of the slider. The onNextAction is set to sendOne because we are only interested in the most recent position. If a new event is fired with a different position argument value (that is calculated by a scripting function) then the previous action is canceled. If the position argument is the same as the previous action then the new action is ignored.

Example 4. Shows an example of caching, we represent here an event that is fired to fetch artist information. If an user navigates multiple times over the same artist then it is not necessary to fetch the same information again and again from the server. This information can already be cached at client side and when the event is sent the information that is cached can directly be given to the getArtistsHandle client side scripting function without sending a request to the server. Setting the cacheresponse attribute to true enables caching.

Example 5. The final example shows a server eventing routine that is declared in terms of client eventing. Server eventing are events that are initiated at server side, an example, sending notifications to the client that the current playing position of a song is changed. Server eventing is currently often declared within HTTP as a client event that keeps blocking on the server until the server has an event for the client. After the server event is handled on the client side a new client event is sent to the server that keeps blocking until the next server event. This results in a server side eventing mechanism described by client eventing. To describe this kind of behavior we first declare an id on the clientevent so that we can use this

```
<!– Example 1: play event –>
<cle:sendAction begin=”play.click” name=”Play” onNextAction=”sendOne”>
    <cle:argument name=”InstanceID” value=”1” />
    <cle:argument name=”Speed” value=”1” />
</cle:sendAction>

<!– Example 2: pause event –>
<cle:sendAction begin=”pauze.click” name=”Play” onNextAction=”sendOne”>
    <cle:argument name=”InstanceID” value=”1” />
    <cle:argument name=”Speed” value=”0” />
</cle:sendAction>

<!– Example 3: a slider control event –>
<cle:sendAction begin=”indefinite” name=”SetPosition” onend=”setSlider”
        onNextAction=”sendOne” queueTime=”100ms”/>
    <cle:argument name=”Position” value=”getNewPlayerPosition()”/>
</cle:sendAction>

<!– Example 4: gets info of current folder –>
<cle:sendAction begin=”artists.mouseover” name=”GetArtistsInfo”
        onend=”getArtistsHandle” onNextAction=”sendOne” cacheResponse=”true”>
    <cle:argument name=”ArtistID” value=”getArtistIndx()”/>
</cle:sendAction>

<!– Example 5: server eventing –>
<cle:sendAction id=”serverEventing” begin=”document.load;serverEventing.endEvent”
    name=”ServerEvents” onend=”handleServerEvent” timeout=”never” />
```

Figure 4.5: Examples of client events

id to reference itself. In the begin attribute we select two events where the event must start on. These are the load event of the document that occurs when the user interface is loaded, and its own endEvent that sends a new request to the server when a server event is handled. The timeout attribute value is set to never so that the client event never timesout for waiting on a server event. The onend argument contains a handle to a client side script that can react on the server event.

4.5 Sending the HTTP request

To use client eventing, client side software must be adapted and also there must be a standard on how these events are sent to the server. This requires, if it is going to be a specification, that there must be a default way of how these messages are sent over the network to the server. Therefore we specify here below how we transferred these messages from the client to the server. An other option to sent client events to the server could, however, be SOAP.

```
<cle:sendAction begin=”play.click” name=”Play”
          xmlns:cle=”http://www.philips.com/remoteui/clientEventing”>
     <cle:argument name=”InstanceID” value=”2” />
     <cle:argument name=”Speed” value=”1” />
</cle:sendAction>
```

Figure 4.6: Example of client event for HTTP message

When the begin event is fired a HTTP POST request is sent to the server. This HTTP POST request uses the value in the controlPage argument to determine where the post request must go to. If the controlPage argument is not specified this defaults to ”clientControl”. So if the remote UI description was loaded from address http://www.philips.com/mediabrowser/remoteui.svg then the client events will be sent to http://www.philips.com/mediabrowser/clientControl if the clientControl argument is not set.

The HTTP body of the POST element will be structured in the following way, Name&firstArgument&secondArgument&etc. An example: figure 4.6 shows a client event; the HTTP body that will be sent when this event is sent is the following

$$Play\&2\&1$$

The order of arguments in the HTTP post message must have the same order as specified within the sendAction element. Because the server already knows the order and the number of arguments for a specific client event no argument name has to be sent.

If succeeded the response of the server will be the HTTP message 200 OK. Within the HTTP body extra information can be added which is evaluated by a scripting function that is mentioned in the onend argument. When no information is sent back but the action succeeded the response is 204 (No content) which means that the client action is processed and no extra information is sent back. If a function is specified in the onend argument then an empty string is passed as content value.

If an error occurs at the server side the response will be HTTP response 500 (Internal Server Error) and error information will be sent in the body. If a function is specified in the onend argument then the data structure passed to this function will have a success value that is false and the content contains the body of the HTTP response containing the error message from the server.

If the client sends an invalid action or an action with wrong arguments the server will respond with a 400 (bad request) and in the body of this HTTP request a reason will be sent that describes why the request is a bad request. If a function is specified in the onend argument then the data structure that is passed to this function will have as success value false and as content value the body of the response of the HTTP request.

4.6 Test results

In chapter 2 we showed tests that we have done on user interface descriptions/protocols like XHTML, SVG and VNC. In this section we add to these results test results of an open source SVG client with as extension client eventing in the way that we described it in the previous sections. As SVG client we used Batik 1.6 [apa] because it is the most complete open source client currently available. We also looked at the mozilla SVG client [moz], this client was still in beta phase, and we missed some essential SVG functionality that we needed for remote user interfaces. The graphical performance of this client is however much better compared to Batik. Batik 1.6 also needed an extension because it does not support animations, but creating only an animation extension was less work than the extensions needed for Mozilla.

In figure 4.7 we show the bandwidth consumption of some clients without client eventing compared to a Batik implementation with support for client eventing. Strange is that Batik requests at startup some files multiple times (peeks seen between the 0:00:09 and 0:00:15) which gives some extra bandwidth overhead of around 18.000 bytes; we do not know the reason for this.

The peeks between 0:01:00 and 0:01:09 are the client events sent by a greedy slider. Especially in this peek you see that client eventing saves a lot of bandwidth, still we have a greedy slider that updates the slider position at server side while the user is dragging the slider. Also we think that caching of responses based on the argument values can save a lot of bandwidth. In our test case we did however not record a lot of user actions of situations where the same information is requested multiple times so this is not really visible in this test.

Client name	Total bytes	Average bytes/s	Max bytes/s
VNC	3.593.154	52.074,7	1.455.372
IE XHTML	251.698	3.647,8	32.172
Opera XHTML	239.715	3.474,1	44.144
FireFox XHTML	271.610	3.936,4	54.519
Netscape XHTML	266.202	3.858,0	58.044
IE SVG Image	223.563	3.240,0	40.394
IE SVG Render	191.496	2.775,3	23.978
Batik SVG Render	186.966	2.596,8	36.059
Batik SVG CE (Render)	127.547	1.771,5	20.100

Table 4.1: Table of bandwidth results

In table 4.1 we show how much bandwidth is used exactly in our tests. This table shows that client eventing saves 32% of bandwidth in our test case.

$$\frac{185.955 - 127.547}{185.955} * 100 \approx 32\%$$

This percentage can even be higher for other clients because Batik already handled the greedy slider correctly by canceling previously sent events. If we fore instance can extend the Adobe SVG plugin with client eventing then this saves around 43% of bandwidth.

$$\frac{191.496 - (127.547 - 18.000)}{191.496} * 100 \approx 43\%$$

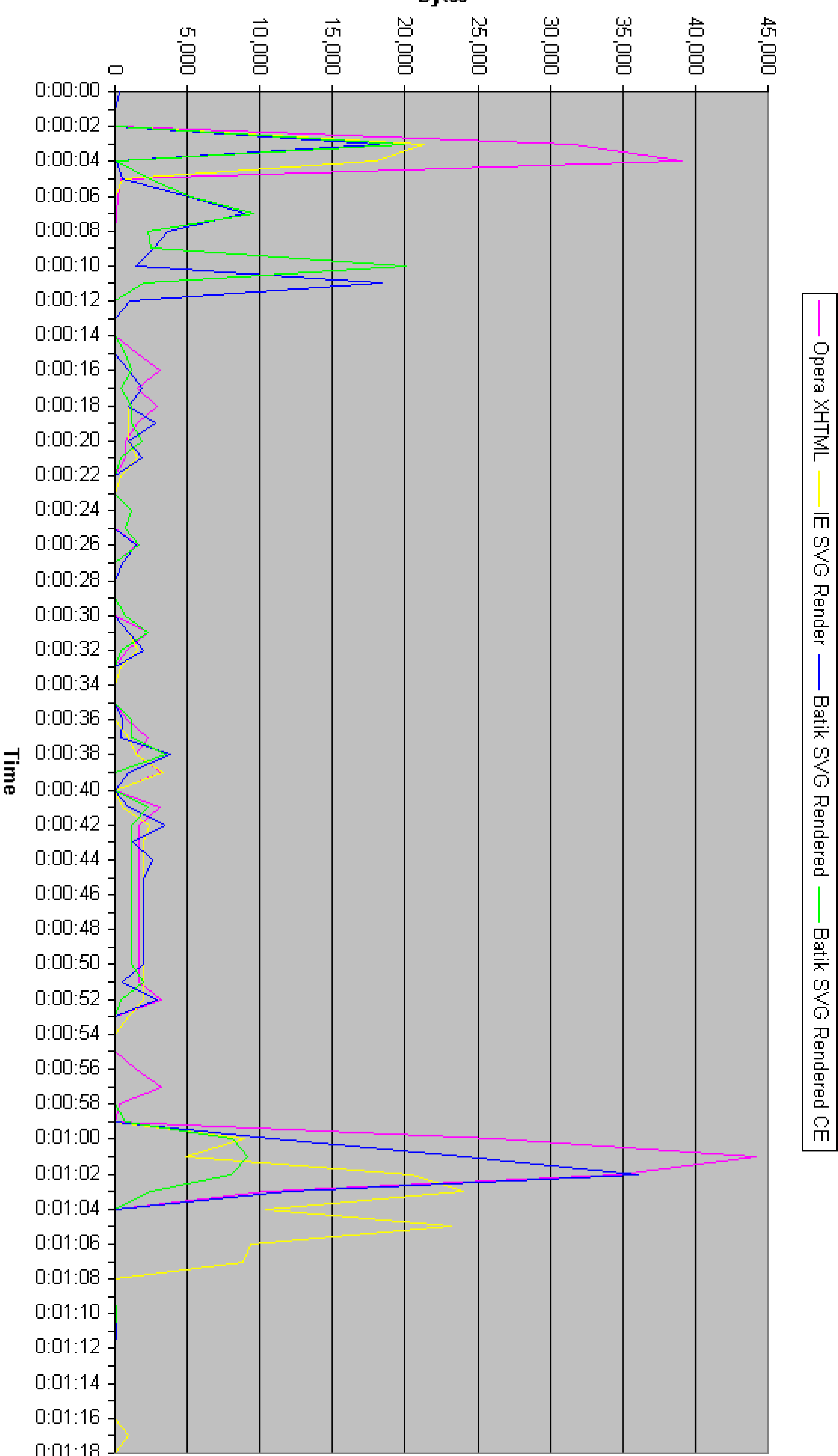

Figure 4.7: Bandwidth measurements

(18000 bytes can be removed out of the Batik SVG CE (Render) results because these were downloaded twice which is notneeded).

Besides bandwidth (and memory which we do not show here because client eventing does not have significant impact on memory) we also measured % processing time used at the server and the client side. Figure 4.8 shows the % processing time used at client side between the different test cases. Batik 1.6 is a SVG client without animation support, we had to implement this ourselves. You can see in figure 4.8, between 0:00:09 and 0:0039, that Batik has large problems with rendering animations and this costs a lot of processing time. The first part 0:00:21 are small animations defined within scripting, the second part are large animations defined in a declarative way. This extra resource consumption is however not interesting for us because we know that animations can be handled better (see for example the yellow line that represents the IE SVG Adobe plugin in figure 4.8). Also client eventing is not aimed at lowering graphical resource consumption but is aimed at lowering resource consumption needed for sending and receiving messages. A lot of messages are sent (in the old way) in between 0:01:00 and 0:01:09. If you look at the difference in processing power needed between the client with client eventing and the other clients without client eventing one sees that client eventing can save significant resources at client side.

Also at the server side we see a difference in resource consumption as shown in figure 4.9. Because fewer requests have to be handled, resources are saved at the server side. The timeline between 0:01:00 and 0:01:09 shows that the client eventing enabled clients take less processing power than the normal clients.

4.7 Client eventing conclusion

We demonstrated here that client eventing is a valuable addition to XML languages that describe user interfaces. Client eventing saves bandwidth, and processing resources. Furthermore client eventing also results in better responding user interfaces, our greedy slider in the example is more responsive in the client eventing enabled test; fewer events are sent to the server and more events actually finish the total procedure without being canceled. The slider is, however, probably still not as responsive as we would like it to be because a roundtrip over the network is needed after moving the slider.

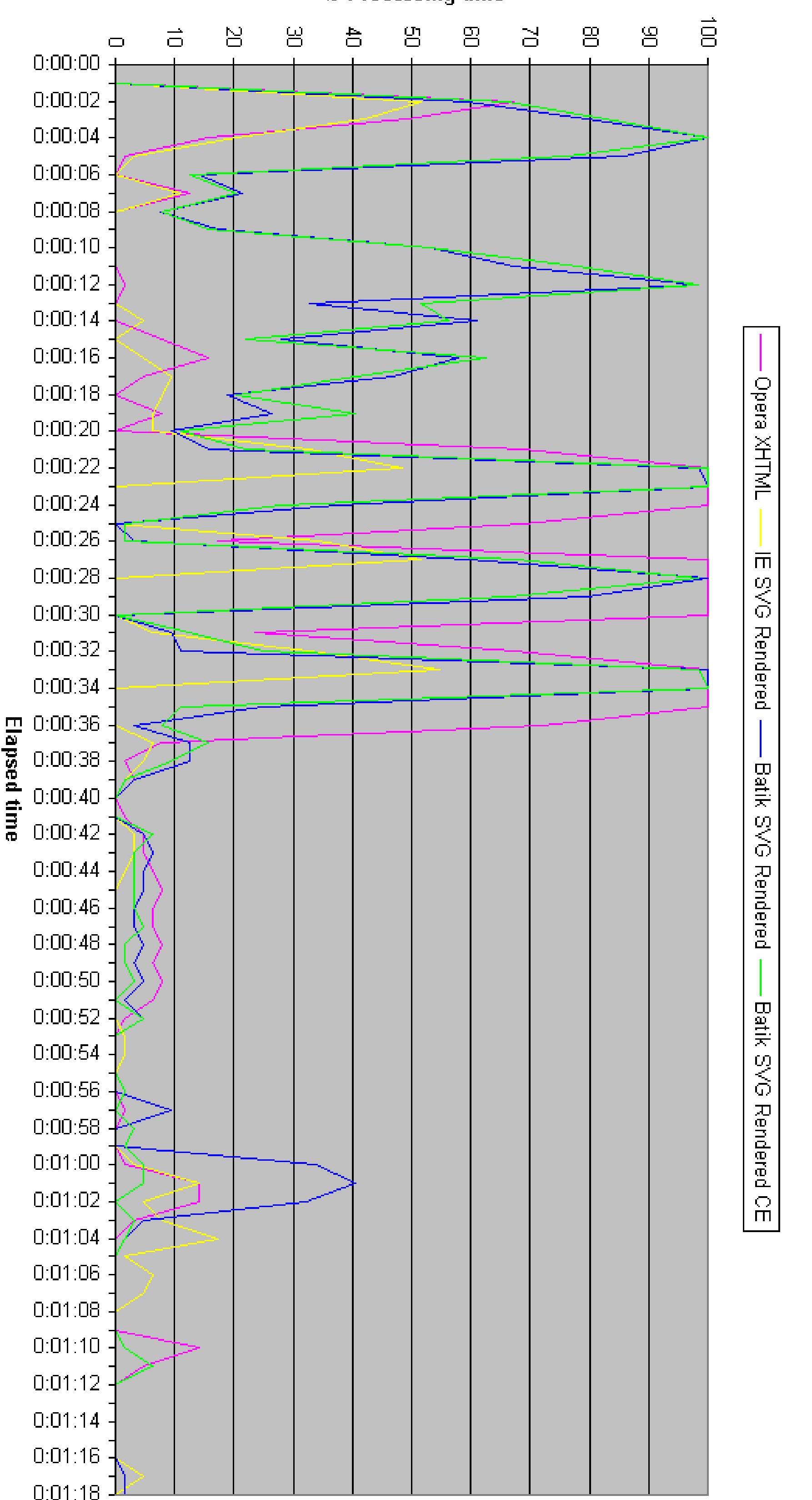

Figure 4.8: Client processing resources

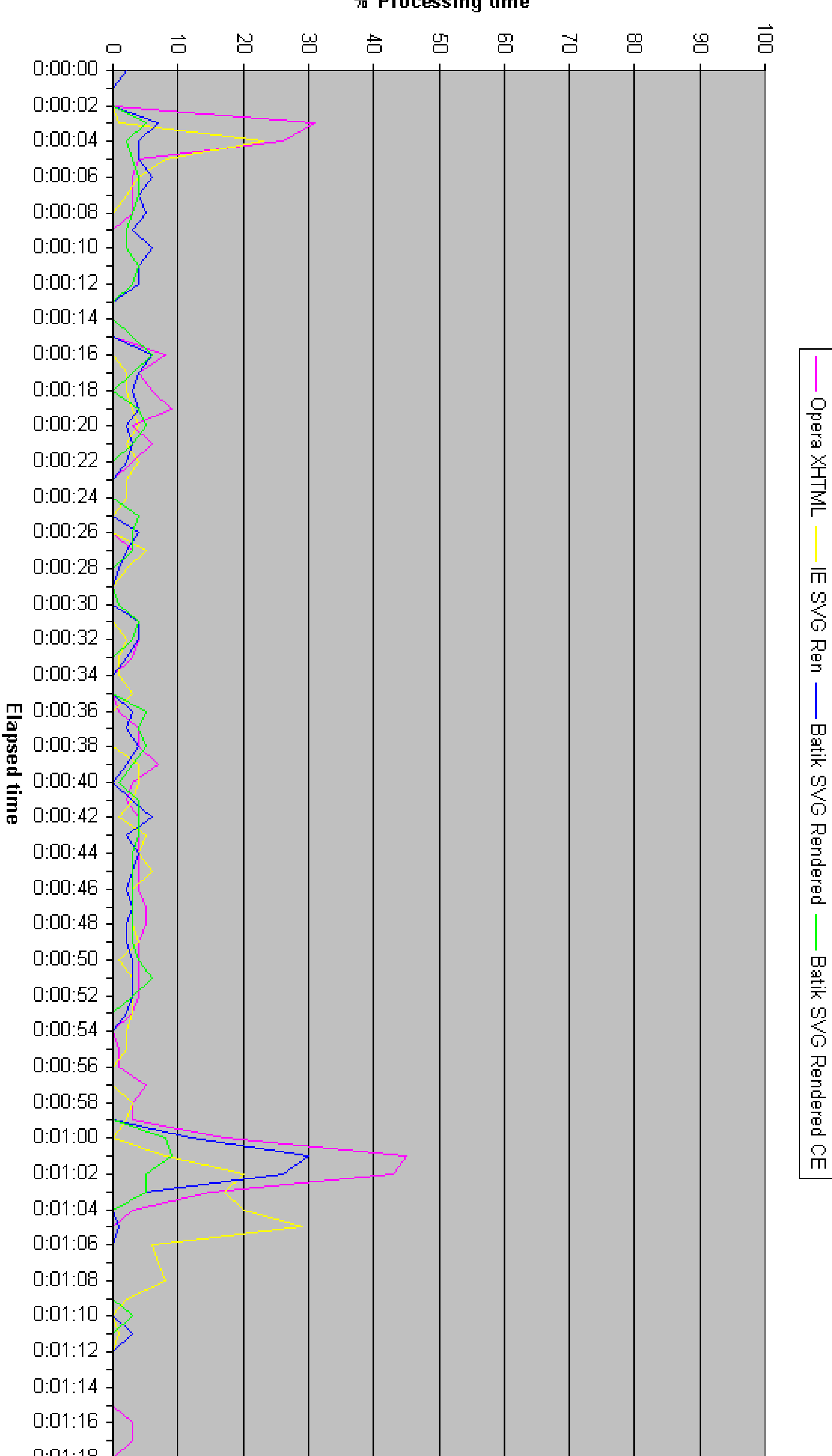

Figure 4.9: Server processing resources

Chapter 5

Optimization 2: SAX Encoding

5.1 Introduction

SAX Encoding is an encoding style based on SAX parsing. What we try to achieve with this encoding is to specify a lightweight encoding for XML that saves bandwidth and processing resources (client/server side). Furthermore the new encoding must be streamable. Streaming is a technique for transferring data such that it can be processed as a steady and continuous stream. With streaming, the client can start displaying the data before the entire file has been transmitted. Streaming technologies are becoming increasingly important with the growth of the Internet because most users do not have fast enough access to download large multimedia files quickly.

But why do we need a new specification/encoding. Can't we not simply use an already existing compression standard like gzip [oai96b], xmill [LS00] or WBXML [W3C05b]? We think that gzip and xmill compression take simply to much resources from resource constrained devices to be really a good option.

WBXML can be a good solution however WBXML has some major points that are not satisfying and can be done better.

Disadvantages of WBXML

- Is not a fixed standard. Software has to be adapted each time a new codepage is added. Codepages are predefined token sets for a particular namespace that can be used to replace frequently used words by a number sparring bandwidth
- Does not support streaming
- Only 255 tokens are available for codepages
- Is not general enough to support all kind of XML conform documents
- See [Kha99] chapter 2.6 for a more elaborate description of unsatisfying points

SAX Encoding has the following advantages

- Streaming
- Encodes a document in one pass
- Low resource consumption at the server and the client side
- Average of 44% compression without any string encoding (so only XML structure is stored on an different way)
- Generalized for all kinds of XML structure
- Does not have internal codepages like WBXML that need software updates
- Unlimited tokens can be used
- Easy to deploy; replaces existing SAX parsers

SAX is a common interface implemented for many different XML parsers, just as ODBC is a common interface implemented for many different relational databases. A SAX parser takes an XML document as input and while processing the document from start till end the SAX parser sends out notifications that can more easily be interpreted by a program.

With SAX encoding we move the SAX parser from the client side to the server side. We encode the SAX messages generated by the SAX parser at the server side. While running the SAX parser we send these messages, instead of XML, in a stream to the client. At the client side we simulate the SAX parser by invoking the methods directly on the default SAX ContentHandler interface, the client does not see the difference between a SAX parser and SAX decoder. By using SAX encoding we can more efficiently encode XML document and save resources at the client side: the new SAX encoded document is already preprocessed and stored in a format more easily translatable to SAX messages. The client does not have to be adapted; only a different SAX (decoder) parser needs to be used. When using dynamically generated documents at server side the server side needs to be adapted because it also needs a SAX parser (was normally done at the client side). If we, however, use static XML files then we can also store the pre encoded version of the XML file at the server side. This has as advantage that the server does not have to encode the static XML each time a client requests the file but can just send the SAX encoded file without doing anything extra. The server side in this case does not need a SAX parser.

5.2 Encoding of SAX functions

Table 5.1 contains the methods defined in the ContentHandler interface of SAX and this interface can be seen as the connection between the SAX parser and the application.

Method name and parameters	**Description**
characters(char[] ch, int start, int length)	Receive notification of character data.
endDocument()	Receive notification of the end of a document.
endElement(java.lang.String namespaceURI, java.lang.String localName, java.lang.String qName)	Receive notification of the end of an element.
endPrefixMapping(java.lang.String prefix)	End the scope of a prefix-URI mapping.
ignorableWhitespace(char[] ch, int start, int length)	Receive notification of ignorable whitespace in element content.
processingInstruction(java.lang.String target,java.lang.String data)	Receive notification of a processing instruction.
setDocumentLocator(Locator locator)	Receive an object for locating the origin of SAX document events.
skippedEntity(java.lang.String name)	Receive notification of a skipped entity.
startDocument()	Receive notification of the beginning of a document.
startElement(java.lang.String namespaceURI, java.lang.String localName, java.lang.String qName, Attributes atts)	Receive notification of the beginning of an element.
startPrefixMapping(java.lang.String prefix, java.lang.String uri)	Begin the scope of a prefix-URI Namespace mapping.

Table 5.1: SAX ContentHandler interface

Not all of the methods of the ContentHandler need to be sent to the client side. The only methods we are interested in for encoding are, startDocument, startPrefixMapping, startElement, endElement, characters and processingInstruction. The rest of the methods are lossy information and do not need to be encoded.

In figure 5.1 on page 50 we show the BNF format for encoding the SAX messages. The BNF-like description used in figure 5.1 use conventions established in [Cro82], except that the "|" character is used to designate alternatives and capitalized words indicate single-byte tokens, which are defined later. Briefly, "(" and ")" are used to group elements, optional elements are enclosed in "[" and "]". Elements may be preceded with * to specify zero or more repetitions of the following element.

In appendix B on page 77 we have included an example of SAX Encoding that can be used to clarify the SAX Encoding.

5.3 Identifiers

Within SAX encoding identifiers are used to identify for example an element that has already been used before. By only sending the identifier we do not need to encode the total name anymore which saves a lot of bits. These identifiers are used to identify namespaces, elements, and attributes that where already included before.

Identifiers within SAX Encoding do not have a fixed size but are variable in size. Depending on the amount of different identifiers the size of the identifier

```
startDocument  = xmlversion charset (describe these)

startElement   = STARTELEMENT [elementName] [attributes]
attributes     = ATTRIBUTE [attrName] attrValue lastAttrBit [attributes]
elementName    = encstring
attrName       = encstring
attrValue      = encstring

endElement     = END

startPrefixMapping    = STARTPREFIXMAPPING [prefix] uri
prefix         = encstring
uri            = encstring

characters     = STARTCHARACTERS encstring

processingInstruction = STARTPI target data
target         = encstring
data           = encstring
```

Figure 5.1: BNF for SAX Encoding

is calculated. E.g. if we store 5 different element names then we need 3 bits to identify one out of this set so 3 bits are used as element identifier. The size of an identifier can grow, an example, the start of an XML document may only need 2 bits to identify an element, in the end of the document for the same element the identifier size can be 12 bits because a lot of different elements are found within the document (between 2048 and 4096).

Namespace identifiers are used to identify the namespace where an attribute or element belongs to. Also the namespace identifier is a reference to the prefix that is used for identifying the namespace which can be used to generate the qualified name.

The size of the namespace identifier is like all identifiers variable and depends on the amount of namespaces declared; if there are 3 namespaces declared then we need 2 bits as namespace identifier (with 2 bits we can make at most $2^2 = 4$ unique combinations).

The namespace identifiers are kept in sync at the server and the client side by storing the uri and prefix from the startPrefixMapping messages in an array. To identify to which namespace an element or attribute belongs to the position within the namespace array is encoded as an identifier. The decoder then decodes the namespace by looking in its own namespace array and using the namespace that is on the position that is described by the namespace identifier. The client knows the size of the identifier because it can just look at the size of its own namespace array. Also the order is kept the same, the first prefix mapping is stored at position 0 the second at 1, etc.

Elemenent identifiers are always preceded by a namespace identifier. Depending on the amount of elements in a particular namespace the size of the element identifier is determined. So elements are stored in an array that is unique to the namespace: this results in smaller identifiers that only store the element names for that particular namespace.

The array with element names stores only new element names and works in the same way as the namespace identifier. If the element name is not known yet then it is added to the element array within its namespace. If the element is already known then an identifier is stored that points to the position of the element within the element array.

Attribute identifiers are stored for each element name separately. These attribute identifiers point to an attribute name and namespace that is already used once for the specific element type. The attribute identifier works in the same way as the element identifier.

5.4 START flags

The start flags in the BNF displayed in figure 5.1 are flags used to indicate which kind of SAX message is described in the coming message part. What is generale within these START flags is that the first part of the flag is used to identify the method. See figure 5.2 for the bit codes that are used for representing a unique START flag.

Bitcode	Starttoken name
00	STARTELEMENT
01	ENDELEMENT
10	STARTCHARACTERS
110	STARTPREFIX
1110	STARTPI
1111	reserved for extensions

Table 5.2: START flags to identify the type of the function

The START flag is followed by some specific flags that are described in the next sub sections.

5.4.1 STARTELEMENT

The encoding of a startElement function call on the content handler is as follows.

startElement = STARTELEMENT [elementName] [attributes]

The STARTELEMENT flag is used to encode the startElement function in the content handler interface that is shown in figure 5.1. Table 5.3 shows the bits used within the STARTELEMENT tag.

Bit position	**Description**
1st and 2nd	Identification bits (00)
3th	Attributes flag
4th	Known element flag
5th	Default namespace
from 6th	namespace/element identifier (variable size)

Table 5.3: STARTELEMENT structure

The first two bits are used to identify the STARTELEMENT. If the 3th bit is set then the STARTELEMENT flag is followed by a list of arguments as described in the next paragraph. If the element is not yet known then the 4th bit is set to 0 and the STARTELEMENT token is followed by an element name, this element name is encoded following the encstring encoding described in section 5.5. If the 5th bit is set to 1 then the default namespace can be used (the default namespace is the namespace declared within the document as xmlns=”...”), if the 5th bit is set to 0 then the 6th bit will contain the start of the namespace identifier as explained on page 50.

Attributes are stored after the STARTELEMENT if the attribute flag is set. Attributes are encoded in the following way.

attributes = ATTRIBUTE [attrName] attrValue lastAttrBit [attributes]

Attributes are stored for each element type uniquely. This saves bandwidth because identifiers stay smaller. The first bit of the attribute flag stores if the attribute is already known within the identifier table of the element. If the attribute is known then the identifier is stored within the ATTRIBUTE flag starting on the 2nd bit position. If the attribute is a not yet known attribute then the 2nd bit contains information about if the attribute belongs to the default namespace. If this is not the case then the 3rd bit contains the start of the namespace identifier followed by the attribute identifier. Figure 5.4 shows an overview of the bit encoding for the ATTRIBUTETAG.

Bit position	**Description**
1st	Known attribute flag
2th	Attribute identifier (if known attribute)
2th	Default namespace followed by namespace identifier (if not known attribute)

Table 5.4: ATTRIBUTETAG structure

If the attribute name is not known then it is added after the ATTRIBUTE flag in the encstring encoding described in section 5.5. The value is also stored in the encstring encoding and stored after the attribute name, or if it is an already known attribute the attribute value is stored after the attribute identifier.

At the end of a description of an attribute one bit is used to indicate that this is the last attribute within the description or that there are more attributes following. If set to 1 then this is the last attribute, if set to 0 then there is an

other attribute following.

5.4.2 STARTPREFIXMAPPING

The STARTPREFIXMAPPING is used to identify that the startPrefixMapping function is called on the contentHandler. A short description of the structure that is used for the startPrefixMapping is as follows.

startPrefixMapping = STARTPREFIXMAPPING [prefix] uri

The first three bits of the STARTPREFIXMAPPING are used to identify the STARTPREFIXMAPPING and has the fixed value of 110. The 4th bit contains the default namespace flag that when set to 1 sets the namespace as default, if set to 0 also an prefix is encoded.

5.4.3 STARTCHARACTERS

The STARTCHARACTERS is used to flag that the characters function is called by the SAX parser. The first two bits are used to identify a characters message. After that the character string is encoded.

5.4.4 STARTPI

processingInstruction = STARTPI target [data]

The STARTPI is used to flag that at the server side the processingInstruction is called on the SAX ContentHandler interface. The first 4 bits are used to identify the STARTPI flag and contains the bit value 1110. The 5th bit is used to flag if the processing instruction contains a data section, if so the data is encoded in the end as an encstring. A dedicate string token table is used for processing instructions that is shared for all processing instructions.

Bit position	**Description**
1st till 4th	Identification bits (1110)
5th	Contains data

Table 5.5: STARTPI structure

5.5 Encoding string values (encstring)

Currently we do not use any optimizations on strings that are stored within SAX Encoding. We directly write the character data within the SAX Encoding without doing any transformations. Transformations on character data cost extra processing resources to encode and to decode, currently we only want to see what is the minimal processing consumption needed to transfer SAX messages to the client.

```
function helloYou()
{
      alert("Hello you");
      alert("Hello you 2");
}

This could also be more compactly encoded like

function helloYou(){alert("Hello you");alert("Hello you 2");}
```

Figure 5.2: Example how scripting can be more compactly stored

5.6 Whitespace handling

Within XML document white spaces (e.g. spaces, enters, tabs) are often used in elements to layout the XML so that it is more readable for an user. If we need to use XML for an automated system these white spaces are not needed and therefore we do not need to encode them. A lot of document types however contain elements where the white spaces within the content must be preserved.

To make our encoder aware of these elements that should preserve white spaces the encoder needs some extra information. This extra information can already be hardcoded for some file types; for SVG we only need to preserve white spaces within the text element and the script element. The rest of the white spaces can be ignored resulting in fewer character function calls that need to be encoded. Also DTD's can be used to indicate if white spacing must be preserved or can be ignored, within DTD's it is possible to set the xml:space attribute to 'preserve' indicating that whitespace must be encoded for this element/attribute. If this attribute is set to default then the SAX parser will call the function ignorableWhitespace instead of characters. Within SAXEncoding we do not encode ingorableWhitespace so these spaces will not be encoded. If an writer still insists on preserving white spaces also the ignorableWhitespace can still be encoded within a file as a character call resulting that it will be shown on the decoding side.

In some cases white spaces within elements on some places can be removed. For example within a script element that uses ECMAScripting (can be used in XHTML and SVG) a lot of white spacing can be removed. An example is given in figure 5.2, this results in a more compact encoding because fewer white spaces need to be encoded.

5.7 Testing

We have implemented SAX Encoding and tested it on 6 different sizes of XML files that where stored on a computer within Philips. Figure 5.6 shows the files and where we found them.

File name	File size	File path
mscorlib.xml	4.904.380	C:\WINDOWS\Microsoft.NET\Framework\v1.1.43
system.drawing.xml	1.258.141	C:\WINDOWS\Microsoft.NET\Framework\v1.1.43
powerplant.xml	469.097	C:\moz\mozilla\lib\mac\PowerPlant
plugin.xml	112.646	C:\java\eclipse\plugins\org.eclipse.debug.ui_3.1.0
batikce.svg	17.655	C:\Experiments\Tests\SVG
feature.xml	919	C:\java\eclipse\features\org.eclipse.sdk_3.1.0

Table 5.6: Files used for testing

5.7.1 Test local decode performance

The main advantages that we try to achieve with SAX encoding is an encoding of XML that saves bandwidth, and client resources. To see if SAXEncoding is a better way of encoding XML documents, that can preform better on the client side, we implemented SAXEncoding.

Within this test we stored the encoded files in memory and see how long it takes to decode the XML file and parse itlocally. Within the decoding process no IO is needed so based on the time used for decoding the file we can draw conclusions on how much more efficient a new encoding is at the client side compared to deflate [oai96a] and just normal XML files. In table 5.7 we first show the size of the encoded files that need to be decoded

File name	File size	SAXEncoding [1]	Deflate
mscorlib.xml	4.904.380	3.177.204 (35%)	424.183 (91%)
system.drawing.xml	1.258.141	830.408 (34%)	155.965 (88%)
powerplant.xml	469.097	88.366 (81%)	21.770 (95%)
plugin.xml	112.646	60.549 (46%)	10.518 (91%)
batikce.svg	17.655	12.415 (30%)	4.594 (73%)
feature.xml	919	559 (39%)	307 (67%)
Average size reduction		**44%**	**84%**

Table 5.7: File sizes after compressing

Table 5.7 shows that by encoding XML in a different way without any efficient encoding of string values within the document we already save an average of 44% in file size. Fewer bytes need to be sent to the client and fewer bytes need to be parsed by the client side.

If you look at the compression ratio of powerplant.xml it is even without any string encoding 81%. The outlining for this file is done with a lot of spaces (normally tabs are used), these spaces are not encoded within SAXEncoding which already results in a significant size reduction. Also not a lot of different element names are used so that the identifiers are kept small resulting in a very compact format for describing XML files.

Within this test we measured the local decoding time. We store the encoded file in memory so that we do not need any IO while decoding the file. This IO can have impact on our test results and gives probably not good comparable test results.

[1] Without encoding strings

In table 5.8 we show the test results of our local tests.

File name	Normal time	Deflate time	SAX decoding time
mscorlib.xml	526ms	579ms (+10%)	565ms (+ 7%)
system.drawing.xml	276ms	303ms (+ 9%)	207ms (-25%)
powerplant.xml	302ms	297ms (- 2%)	76ms (-75%)
plugin.xml	155ms	159ms (+ 3%)	71ms (-54%)
batikce.svg	84ms	85ms (+ 1%)	35ms (-58%)
feature.xml	33ms	34ms (+ 3%)	22ms (-33%)
Average time reduction		**+ 4%**	**-40%**

Table 5.8: Local file decoding times

Within these test results we see that in 5 of the 6 test files SAX encoding performs significantly better compared to when using a normal SAX parser. The only test case that did not perform better is the test case with mscorlib.xml, we don't have a clear explanation yet why this is the case. The file does not contain a lot of different XML structures and contains a lot of insignificant whitespaces. We however think that within this file a lot of character data is swapped from normal byte encoding (every byte contains one char) to binary encoding (one char can be spread over 2 bytes) which results in a slight performance loss, but this is still a guess.

We however can see in table 5.8 that SAX decoding can save around 40% of processing resources at the client side. We used quite a heavy machine for testing (see the client machine specs in appendix A), if we hover apply the same SAX Encoding to a client with less processing resources this can save significantly in time.

5.7.2 Test remote decode performance

Our second test measures the performance of decoding a file in combination with sending the decoded file over the network. We test this by storing the file in memory at the server side and running the following sequence.

1. starttime = currenttime
2. Send a HTTP request to the server for the XML file in a certain encoding format
3. While downloading the file to the client decode the XML file and run the SAX parser
4. Wait until the total document is parsed and the endDocument method on the contentHandler is called
5. testtime = currenttime - starttime

All this is done on the same situation as within the test environment described in appendix A. The only thing that is different in our test environment is that we reduced the network speed to 10Mbps half duplex. Table 5.9 shows the average of a test that was run 100 times for each file.

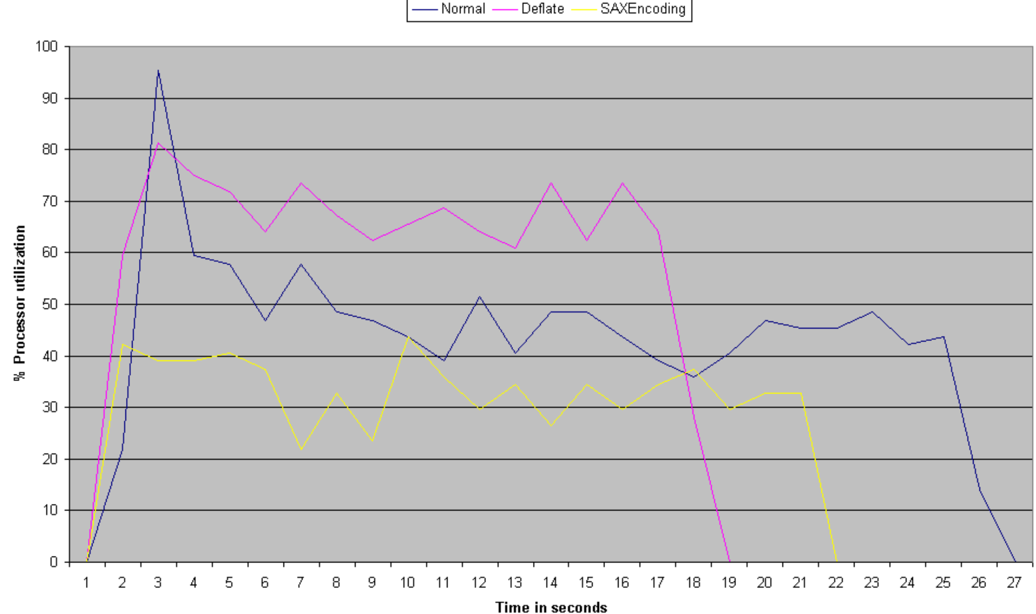

Figure 5.3: Client processor utilization when sending/decoding batikce.svg 100+ times

File name	Normal time	Deflate time	SAX decoding time[2]
mscorlib.xml	4480ms	435ms (90%)	2923ms (35%)
system.drawing.xml	1145ms	160ms (86%)	758ms (34%)
powerplant.xml	426ms	59ms (86%)	85ms (80%)
plugin.xml	105ms	23ms (78%)	61ms (42%)
batikce.svg	22ms	15ms (32%)	19ms (14%)
feature.xml	10ms	12ms (-16%)	8ms (20%)
	Average time reduction	**74,3%**	**37,5%**

Table 5.9: Remote file decoding times

As you can see high compression of files wins in response time, however this number is also based on processing resources at the client side. Our goal is to get an encoding that does not need a lot of resources at the client side and still performs rather good. If we look at figure 5.3 we see that SAXEncoding without any encoding of string values on average uses less resources then deflate or normal SAX parsing.

[2]Without encoding strings

5.8 SAX Encoding conclusion

Within figure 5.3 we see that SAX encoding does the same amount of work in less time and even uses fewer processing resources. Furthermore SAX Encoding gives us a more compact way of storing XML data which saves us on average around 44% of bandwidth.

SAX encoding is not only intended to be used for encoding remote user interfaces but is general enough to be used for any XML data.

Chapter 6

Optimization 3: Reduce animation resources

As shown within figure 2.5 on page 25 showing animations on a client can take a lot of processing resources. If an animation that needs a lot of resources is shown on a resource constrained device this can result in an animation that takes much more time than was original intended. Also a large animation can slow down other tasks that an user interface needs to perform, like react on user actions.

We rather see animations as best effort tasks with a low priority that can skip frames if needed. This skipping frames can be easily implemented if declarative animations are used (is available in SVG). If we have an animation description like in figure 6.1 the client software can decide how much frames are shown at the client. If we however use an animation that is described in scripting this will be much more difficult. The programmer of the user interface then needs to describe how much frames are shown for the animation and this is often fixed in code.

When using declarative animations we can choose one of the following approaches.

- Full animations: If a lot of resources are free we can show a fluent animation that shows around 50 animation frames a second.

- Skipping frames: If the client is limited in resources we can show fewer animation frames depending on the amount of resources. This can be done by using a linear algorithm for calculating animations that based on the current time can calculate the current position. For figure 6.1

```
<rect x=”45” y=”10” width=”10” height=”10” fill=”red”>
    <animate attributeName=”y” from=”10” to=”90” dur=”10s”\>
<\rect>
```

Figure 6.1: SVG declarative animation example

the algorithm is

$$f(t) = 10 + 80 * t/10$$

where t is the time in seconds. This is an approach already available in SVG and described within the SLIM 2 specification [Wor01].

- Skip animation: If the client does not have resources free to generate an animation, skip the animation by moving the animated object directly to the end position and raise the endevent of the animation: so we ignore the duration of the animation. If chosen for this approach all animations need to be skipped at that specific time. It is not correct if the animation that moves the background is skipped while an animation that at the same time moves the text is animated; this will give strange effects.

- Reduce animation frames based on real display size: If a user interface is created on a resolution of 640x480 but it is scaled to a display of 320x240 fewer movements need to be calculated because the amount of pixels that need to be moved over is reduced (e.g. if the animation on 640x480 is done over 250 pixels then the same animation on 320x240 only has to move over 125 pixels so we need at most 125 animation frames).

Of course this can also be described within JavaScript. This takes, however, for every animation a lot of extra code which results in extra bandwidth consumed, more parsing/interpreting of JavaScript and more work for developers.

6.1 Testing results

To see how the animations perform if the skip animation approach is chosen we have included here some test results. These test results are measured in the way described in appendix A on page 69 and shown in figure 6.2.

The blue line in figure 6.2 shows the normal processor % utilization without any optimizations. The red line shows the processor % utilization with skipping animations. As you can see in figure 6.2 this saves a lot of resources between the 21 and 37 seconds.

6.2 Animation conclusion

We think that using skipping animations is a valuable addition to remote user interfaces. Even on devices that have a lot of resources free for rendering animations this can still be a nice option to provide to the user; the user interface gets much more responsive without any animations.

Skipping frames or even skipping animations can reduce the resources needed for an animation significantly resulting in a more responsive UI on resource constrained devices. We think that large animations can be really a large burden for resource constrained devices.

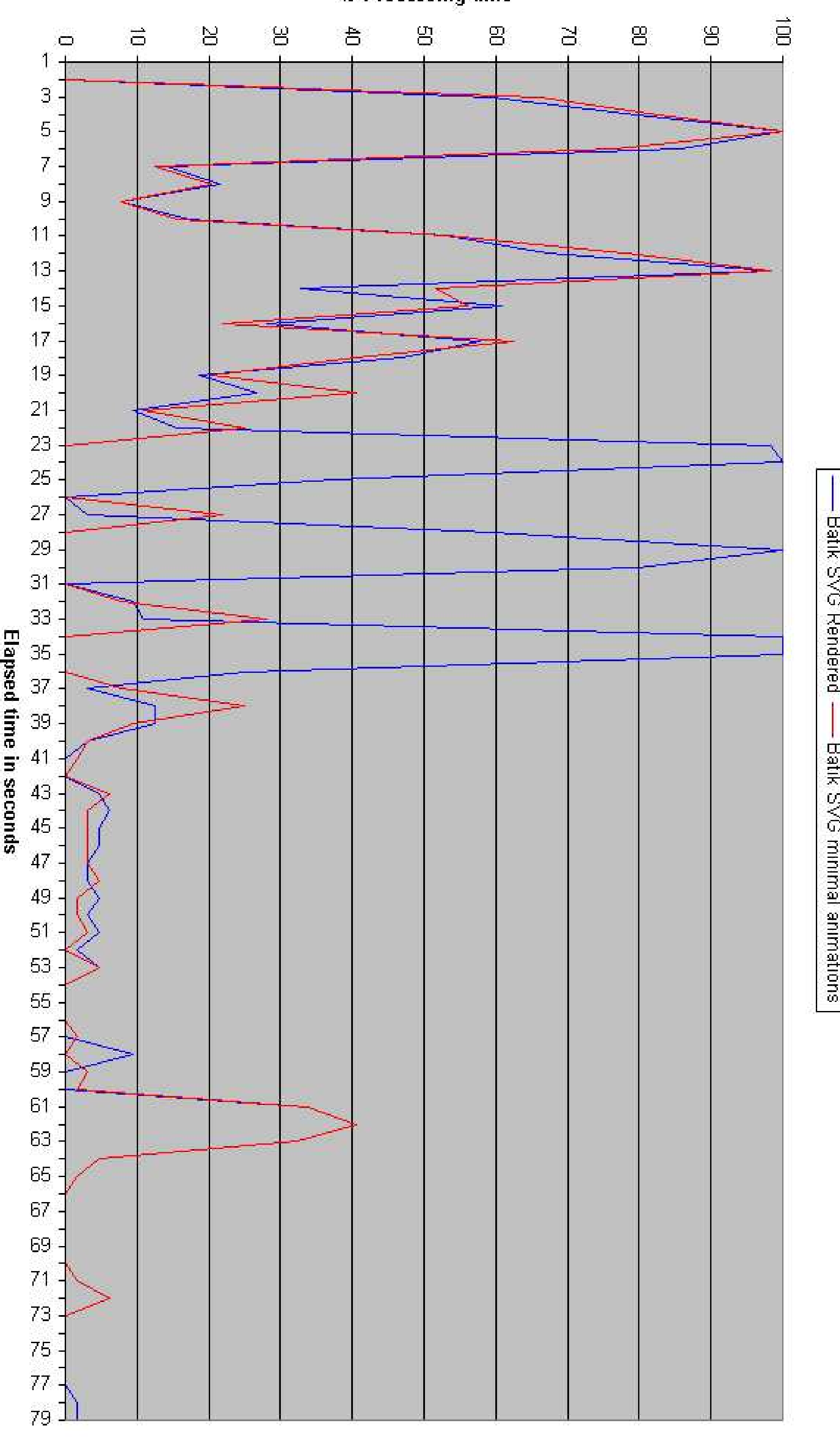

Figure 6.2: Animation reduction if we choose for animation skipping

Chapter 7

Possible other optimizations

Within this chapter we give an overview of possible other research topics to even more reduce the resources needed for remote UI or to add extra functionality.

7.1 SVG widgets

Something missing within SVG is the notion of a widget.

Widgets have the following advantages

- Can give better performance results
  - Fewer scripts needed to define widget operations saving bandwidth and client resources
  - Clients can use its own device optimized widget sets that can perform better and width lower resource consumption than widgets described in SVG
- Give the user a familiar feeling over different user interfaces because the user knows it local widget set
- Reuse widget implementations over multiple document

A good starting point for this research can be the new SXBL specification (SVG's XML Binding Language) [Wor04]. SXBL can be used to bind widget definition to abstract widget declarations that are stored within the SVG remote user interface definition. This solution will however probably loose the performance advantage that you get when using a widget set optimized for the local device: by describing widgets in a general, not device/operating specific, way we can not get better performance results, it performs the same as describing widgets in normal SVG..

Other research in this area is for example SVGForms, we however could not lay our hands yet on a paper that describes this technique because it was not yet published. We however found the abstract, see appendix C on page 83 that looks very interesting.

7.2 Combining VNC with SVG

The main problem when using VNC is the resource consumption at the server side. These resources at the server side are needed for storing and performing calculations on a framebuffer. A possible solution to replace the framebuffer at the server side is by using a SVG tree instead of a framebuffer. Within SVG an absolute coordinate system is used which gives the possibility to only render a small part of the total SVG screen with minimal knowledge of the rest of the SVG structure. Also within SVG animations can be described in a declarative way as described in chapter 6. By using declarative animations in combination with an absolute coordinate system an application at the server side can easily calculate which part of the screen is animated. By using the copyrect command, defined in VNC, a server can sent to the client a command to copy a rectangle from one position in the screen to an other resulting in that no screen data has to be sent over.

Using a SVG tree at the server side instead of a client framebuffer can save a lot of resources. Not only memory but the calculations done on the server side probably are less because a user interface is only partially rendered in memory. Also using the copyrect method gets a lot easier by using declarative animations. A framebuffer can probably fully be replaced by a in memory SVG tree.

Also the user interface only needs to be partially rendered when a request of screen changes comes in from the client. So when a request comes from client side the SVG tree is searched for changes. If there are changes these are described by copyRect or rendered image data and sent back to the client.

Furthermore we also have a standard user interface description language at the server side that gives us two possibilities to describe a user interface on the client. If the client has enough resources to render a full SVG user interface it can simply download the SVG file, if not it can use VNC to render the user interface at the server side.

Chapter 8

Conclusion

From testing we concluded that choosing SVG as default remote user interface description language saves processing resource and bandwidth when compared to other content formats. Furthermore we think that SVG is more suitable for describing remote user interfaces than XHTML. XHTML is more a language for laying out texts with some nice images just as the name suggests (TML stands for Text Markup Language). We see user interfaces as more graphical descriptions that do not contain large chunks of text but more consist of graphical objects that respond on user actions. SVG is more graphical based so in our eyes more suitable for remote UI.

By implementing client eventing within SVG/XHTML browsers we save around 43% of bandwidth and get a better responding remote user interface because events sent from the client are now better handled. Also client eventing saves resources at the client and the server side. We even save work for a remote user interface developer; by specifying the client event in a declarative way saves a lot of scripting for the developer and eliminates the need for the none standard getURL function that otherwise needs to be used.

SAX encoding is a new way of encoding XML structured documents that saves 40% of processing time at the client side compared to normal SAX XML parsers and also saves around 44% of bandwidth. Implementing SAX encoding within already existing clients is quite simple because you only have to replace the existing SAX parser by a new SAX decoder.

Declarative animations allow clients to render animations on a way that is most convenient for the current client device based on the available resources. If a client has a lot of processing resources free we can render the total animation fluently. If the client has almost no processing resources we can simply skip the animation and still have a working user interface. Declarative animations are already defined in SVG, for remote user interfaces this can be a very good optimization for XHTML.

If these optimizations are combined within one client we get a remote user interface description that saves around 50% bandwidth. We also reduced processing resources at the client side significantly, and even save resources at the server side. We think that an combination of SVG, client eventing, and UPnP can be a good standard for remote UI. This standard can show a remote UI on the combination of a resource constrained server, low bandwidth, and a resource constrained client.

Bibliography

[apa] Batik svg toolkit.

[CC04] Jack Chaney and John Card. A user interface for home networks using web-based protocols. CEA-2027, January 2004.

[Cro82] D. H. Crocker. Standard of the format of ARPA internet text messages. Technical Report RFC 822, August 1982.

[Dee04] Walter Dees. Handling device diversity through multi-level stylesheets, February 03 2004.

[DKMR03] Micah Dubinko, Leigh L. Klotz, Roland Merrick, and T. V. Raman. XForms 1.0. World Wide Web Consortium, Recommendation REC-xforms-20031014, October 2003.

[Eth] Ethereal. A network protocol analyzer. http://www.ethereal.com/.

[eWD] R. Verhoeven en W. Dees. Defining services for mobile terminals using remote user interfaces.

[IEE] Ieee 802.11, the working group setting the standards for wireless lans.

[ITU97] ITU. Itu-t recommendation t.128share - application sharing. International Telecommunication Union, 1997.

[Kha99] Rohit Khare. W* effect considered harmful. April 1999.

[KHR] Openvg - the standard for vector graphics acceleration.

[Kum] S. Senthil Kumar. Winmacro.

[Lis03] See List. Scalable vector graphics (SVG) 1.1 specification. Technical report, January 14 2003.

[LS00] Hartmut Liefke and Dan Suciu. XMill: An efficient compressor for XML data. SIGMOD Record: Proc. ACM SIGMOD Int. Conf. Management of Data, 29(2):153–164, 16–18 May 2000.

[Mac] Macromedia. Macromedia flash (swf) file format specification version 7.

[Mic00] Microsoft Corporation. Universal Plug and Play Device Architecture, June 2000. statut : version 1.0 , http://www.upnp.org/download/UPnPDA10_20000613.htm.

[Mic02] Microsoft Corporation. Printer Device V 1.0 and Printer Basic Service V 1.0, July 2002.

[Mic04] Microsoft Corporation. Remote UI Client and Server V 1.0, August 2004.

[moz] Mozilla svg project.

[oai0] ECMAScript language specification, March 31 0.

[oai96a] DEFLATE compressed data format specification version 1.3, August 06 1996.

[oai96b] GZIP file format specification version 4.3, June 14 1996.

[oai96c] Hypertext transfer protocol - HTTP/1.1, June 03 1996.

[Pla03] Timmons C. Player. Wlan benchmarking mixes old with new using wired-network-device methodologies. August 2003.

[Ric05] Tristan Richardson. The RFB Protocol. RealVNC Ltd, March 2005.

[SW79] William Strunk, Jr. and E. B. White. The Elements of Style. Macmillan Publishing Company, New York, 3rd edition, 1979. 0-02-418200-1 (pbk); PE 1408.S772.

[W3Ca] World Wide Web Consortium. Cascading Style Sheets.

[W3Cb] World Wide Web Consortium. Document Object Model (DOM) Technical Reports.

[W3C00a] World Wide Web Consortium. XHTML 1.0, January 2000. statut : W3C Recommandation , http://www.w3.org/TR/xhtml1/.

[W3C00b] World Wide Web Consortium. XHTML 1.0: The Extensible HyperText Markup Language, January 2000. statut : W3C Recommandation , http://www.w3.org/TR/xhtml1/.

[W3C03] World Wide Web Consortium. Mobile SVG Profiles: SVG Tiny and SVG Basic, January 2003. statut : W3C Recommandation , http://www.w3.org/TR/SVGMobile/.

[W3C05a] World Wide Web Consortium. Scalable Vector Graphics (SVG) Full 1.2 Specification, April 2005. statut : W3C Working Draft , http://www.w3.org/TR/SVG12/.

[W3C05b] World Wide Web Consortium. WBXML Specification, April 2005.

[WEJ+02] Mark R. Walker, Jim Edwards, Michael Jeronimo, John G. Ritchie, and Ylian Saint-Hilaire. Remote I/O: Freeing the experience from the platform with UPnP* architecture. Intel Technology Journal, 6(4):30–36, November 2002.

[Wor01] World Wide Web Consortium. SMIL 2.0 Specification, January 2001.

[Wor04] World Wide Web Consortium. SVG's XML Binding Language (sXBL), September 2004.

Appendix A

Testing procedures

A.1 Test environment

Here a description of the equipment we used for doing the tests.

The client

- Intel Pentium 4 1.7 Ghz
- 512 MB of RAM
- 3Com 3C920 Integrated Fast Ethernet Controller
- Windows XP professional SP-1
- 16MB ATI Rage 128 Ultra (video card)

The server

- Intel Pentium 3 500 Mhz
- 256 MB of RAM
- Windows 2000 professional SP-4
- 3Com 3C918 Integrated Fast Ethernet Controller
- 3D Rage Pro AGP 2X (video card)

The network

- Wired 100Mbit full duplex (with a cross cable) for batik test case
- Wired 10Mbit half duplex (with a cross cable) for encoding test case

A.2 Software used for testing

- Ethereal 0.10.9 [Eth]: For monitoring the bandwidth consumption and view the packages that are send from and to the server.
- Performance viewer: For monitoring CPU and memory consumption of an application at the server and client side.
- WinMacro v1.2 [Kum]: This tool enables us to record actions like mouse-movements and keypresses done by the user. After recorded one session we can replay the actions taken by the user so that with each test the user actions are done in the same way with the same timeline so the results are more easily comparable.

A.3 Testing procedures

A.3.1 How are the measurements created and started

On the server and client side measurements are done with a counter log in the performance monitor. This counter log is created as follows.

1. Startup the performance monitor
2. Expand "Performance Logs and Alerts" in the left pane
3. Select "Counter logs"
4. Right click the right pane and select "New log settings"
5. Enter "RemoteUI measurements" as log name
6. Add the following counters by using the "Add counters" button
    - Memory\% Committed Bytes In Use
    - Memory\Available Bytes
    - Memory\Committed Bytes
    - Memory\Page Faults/sec
    - Memory\Pages/sec
    - Paging File(\??\C:\pagefile.sys)\% Usage
    - PhysicalDisk(_Total)\% Disk Time
    - Process(_Total)\Virtual Bytes
    - Processor(_Total)\% Processor Time
    - Processor(_Total)\Interrupts/sec
7. Set the sample data interval to 1 second
8. On the "Log files" tab, select "Text file (comma delimited)" as log file type
9. On the "Schedule" tab, modify the start log settings to start the log manually

The performance log is created at the server side and the client. On the client side the performance monitor is started automatically by the WinMacro script (actions for turning on and off the performance monitor are recorded). Because we recorded the action of turning on and off the performance monitor the monitor results on the client start exactly on the same second in the timeline of testing. On the server the performance monitor is turned on by hand but these test results can later on be automatically adapted by removing the extra-recorded results from the start so that the server side results start exactly on the same moment as the client.

Also at server side Ethereal is used to monitor the bandwidth consumption. Ethereal bandwidth monitoring is started as follows.

1. Startup Ethereal
2. From the menu select "Capture", from the submenu select "Start"
3. In the window that opens add "host 169.254.85.77" as "Capture Filter:" what results in that all messages from/to host 169.254.85.77 are captured; 169.254.85.77 is the client pc.
4. Select "OK": this starts capturing network packages.

Ethereal can be started before the tests begin; Ethereal starts its timeline when the first package is captured what is the first request to the server by the client.

A.3.2 Running the tests

The following actions are performed before the tests so that we get clean test results

- Disks at client and server are defragmentated
- Redundant processes are shutdown like the virus scanner, SMS (System Management Server), Task schedulers, etc.
- Cache of client is cleared (our own webserver at server side currently has no cache)

Startup the tests

1. Startup WinMacro at client side and select the WinMacro script that is recorded for the test as selected file.
2. Startup performance monitor on the client side
3. Start Ethereal at server side to monitor bandwidth
4. Startup and start performance monitor at the server side
5. Start playback of the recorded script in WinMacro at the client side (first recorded action is starting up the performance monitor)
6. Wait until WinMacro script is finished

7. Stop Ethereal and the performance monitor at the server side (the performance monitor at the client side is started and stopped by the WinMacro script)

A.3.3 Transforming the test results

After a test we have three files, the server side performance monitor log, the client side performance monitor log and the Ethereal bandwidth capture log. We now transform these logs so that we can compare the results in Excel charts.

Transforming the client side performance monitor log The client side performance log is a comma-separated file that can be opened in Excel. First step that we need to perform is to modify the date/time. The performance monitor stores this in one field as 02/21/2005 14:08:22.084. For our analysis we are only interested in the time. To get the time in one column without the date the following action needs to be done.

1. Select the column that stores the time in Excel.
2. Select from the menu "Data", from the submenu "Text to columns"
3. A wizard opens, in the first screen select "Fixed width" and click "Next >"
4. Click again "Next >"
5. Fill in the destination field the following value "=$L:$L" and select "Finish"
6. Right click column "M" (column that stores time values)
7. Select out of the drop down menu "Format cells"
8. Select as category "Time" and as type "37:30:55"
9. Click "Ok" Also we need to remove the second row (first row with test results) out of the results; this row contains startup resources needed for starting up the performance monitor, we are not interested in these numbers.

After these actions open a new Excel file and copy the modified results to a new excel sheet.

We also like to know how much of memory is consumed at the client and server side. Memory is however a difficult thing to measure/calculate. In this test we use the available bytes counter value from the performance viewer. But we don't want to see the available bytes but rather the used bytes. To get the amount of used bytes we first need to calculate the differences in available bytes over the test. We calculate it as follows.

1. On the first not used column we place the following formula "=MAX(F2:F80)-F2" in the second cell from the top (the first is the title). F2 is the column that contains the measured available bytes.
2. The formula in the second sell of the column we copy to the rest of the column.

Transforming the server side performance monitor log The server side performance monitor log has the same format and structure as the client performance monitor log. The same steps need to be done as described for the client log.

Transforming Ethereal bandwidth capture log When Ethereal is stopped the captured packages are not yet stored in a file. So first save the Ethereal log by clicking the save button. The log is saved in an Ethereal format, however in Ethereal it is also possible to export captured logs as Plain text files as described here below that we can use to generate charts in Excel.

1. In Ethereal select "File" from the menu, from the submenu select "Export", from the subsubmenu select "as 'Plain Text' file".
2. Give a file name to save the results in with the extension ".txt"
3. Deselect in the "Packet Format" the "Packet details:" selection box
4. Click "OK"

Ethereal stores the captured data in the exported plain text file as shown in table A.1.

No.	Time	Source	Destination	Protocol	Packet length	Info
243	5.007586	169.254.85.78	169.254.85.77	TCP	54	8080 >2194 [FIN, ACK] Seq...
244	5.007700	169.254.85.77	169.254.85.78	TCP	60	2194 >8080 [ACK] Seq=409 ...
245	5.007799	169.254.85.77	169.254.85.78	TCP	60	2194 >8080 [FIN, ACK] Seq...
246	5.007860	169.254.85.78	169.254.85.77	TCP	54	8080 >2194 [ACK] Seq=1520...
247	11.07838	169.254.85.77	169.254.85.78	TCP	62	2195 >8080 [SYN] Seq=0 Ac...
248	11.07850	169.254.85.78	169.254.85.77	TCP	62	8080 >2195 [SYN, ACK] Seq...
249	11.07866	169.254.85.77	169.254.85.78	TCP	60	2195 >8080 [ACK] Seq=1 Ac...

Table A.1: Example of ethereal log

For analyzing the bandwidth we are interested in how much data is send within each second. The data that we are interested in the above example should look like table A.2 so that we can generate a chart in excel that can be compared to CPU and memory consumption.

Time	Bytes send/s
5	228
6	0
7	0
8	0
9	0
10	0
11	184

Table A.2: Example of wanted log

For generating these results we used Access. The following steps describe how we used Access to get these results.

1. Startup Access and create a Blank Access Database
2. Select out of the menu ”File” out of the submenu ”Get external data” and out of the subsubmenu ”Import”
3. Select the Ethereal plain text file and click ”Import”
4. Select ”Fixed width” and click ”Next >”
5. Check in this screen that each column is separated by a line. If a column contains multiple lines remove them, click ”Next >”
6. Click again ”Next >”
7. Select for each field ”Do not import field (skip)”except for the time and the packet length field, click ”Next >”
8. Click again ”Next >”
9. Give as name for the table ”Ethereal results” and click ”Finish”

We now imported the Ethereal bandwidth data into access. The following action is to create a second table that we also can reuse for other tests. This table has to look like table A.3 and is named ”Clean”.

ID	Time	Packet length
29445	0	0
29446	1	0
29447	2	0
29448	3	0
29449	4	0
...	...	...

Table A.3: Clean table

It should have at least 130 records and in these records time has to be numbered from 0 till 129 and the packet length must be 0. This table is used by the following query that we execute on the access database.

INSERT INTO [Ethereal results] ([Time], [Packet length]) SELECT Clean.Time, Clean.[Packet length] FROM Clean;

This query inserts a zero value of the packet length for each second. We need this zero value because otherwise the following query only shows the seconds when data is send, but we will miss values for seconds when no data is send.

SELECT TimeSerial(0,0,Fix([Time])) AS Seconds, Sum([Packet length]) AS SumOfPacketLength FROM [Ethereal results] GROUP BY Fix([Time]);

The results of the above query we can copy to the excel workbook in a new sheet that also contains the sheets with server and client resources used. Out of these results we can generate data charts and information that contain the test results of resources consumed for each second.

Within this Excel workbook we also store other test results in separate sheets so that we can compare tests of other clients/protocols with each other.

Appendix B

SAX Encoding example

To show how the SAX encoding works we are going to encode a XML file that is shown in figure B.1. When we parse this example in a normal SAX parser the sequence in figure B.2 shows the calls that will be done on a ContentHandler interface. These ContentHandler messages are encoded within the SAX encoding as described within chapter 5 and exactly how is shown in figure B.1.

The small example shown within figure B.1 contains 344 bytes (characters). By encoding this example with SAX encoding we reduce the size of this document to a size of 166 bytes what means that we already have compressed the example with

$$100 - \frac{166}{344} * 100 \approx 52\%$$

resulting in a smaller file format that needs less bandwidth to be transferred. The great thing about this encoding format is that decoding the SAX encoding costs less resources at client side then running a SAX parser on the normal XML file.

```
<svg id=”svgdoc” xmlns=”http://www.w3.org/2000/svg”
              xmlns:xlink=”http://www.w3.org/1999/xlink” >
    <g>
        <!–Some SVG file –>
        <rect x=”30” y=”30” width=”40” height=”40” fill=”green”/>
        <g>
        <rect x=”40” y=”40” width=”40” height=”40” fill=”red” xlink:href=”test2.svg”/>
        </g>
    </g>
    <text x=”40” y=”80” >Hello to you</text >
</svg>
```

Figure B.1: Example to encode with SAX Encoding

```
1: startDocument()
2: startPrefixMapping("","http://www.w3.org/2000/svg")
3: startPrefixMapping("xlink","http://www.w3.org/1999/xlink")
4: startElement("http://www.w3.org/2000/svg","svg","svg","arguments")
5: characters("
")
6: startElement("http://www.w3.org/2000/svg","g","g","arguments")
7: characters("
")
8: characters("
")
9: startElement("http://www.w3.org/2000/svg","rect","rect","arguments")
10: endElement("http://www.w3.org/2000/svg","rect","rect")
11: characters("
")
12: startElement("http://www.w3.org/2000/svg","g","g","arguments")
13: characters("
")
14: startElement("http://www.w3.org/2000/svg","rect","rect","arguments")
15: endElement("http://www.w3.org/2000/svg","rect","rect")
16: characters("
")
17: endElement("http://www.w3.org/2000/svg","g","g")
18: characters("
")
19: endElement("http://www.w3.org/2000/svg","g","g")
20: characters("
")
21: startElement("http://www.w3.org/2000/svg","text","text","arguments")
22: characters("Hello to you")
23: endElement("http://www.w3.org/2000/svg","text","text")
24: characters("
")
25: endElement("http://www.w3.org/2000/svg","svg","svg")
26: endPrefixMapping()
27: endPrefixMapping("xlink")
28: endDocument()
```

Figure B.2: Calls on ContentHandler when parsing example in figure B.1

Table B.1: SAX Encoding example of figure B.1

Binary code	**Text value/Description**
1: startDocument()	
00000001	SAX encoding version 1
01101010	UTF-8 IANA charset encoding
2:startPrefixMapping("","http://www.w3.org/2000/svg")	
110	Start prefix mapping flag
1	Default namespace
	http://www.w3.org/2000/svg (StringEnc of uri)
3: startPrefixMapping("xlink","http://www.w3.org/1999/xlink")	
110	Start prefix mapping flag
0	Not default namespace
	xlink (StringEnc of prefix)
	http://www.w3.org/1999/xlink (StringEnc of uri)
4: startElement("http://www.w3.org/2000/svg","svg","svg","arguments")	
00	Start element flag
1	Element contains attributes
0	Element is a not known element
1	Element is in default namespace
	svg (StringEnc of Element name)
0	Not a known attribute
1	Is default namespaces
	id (StringEnc of attribute name)
	svgdoc (StringEnc of attribute value)
1	Last attribute flag
5: characters(" ") (ignore)	
6: startElement("http://www.w3.org/2000/svg","g","g","arguments")	
00	Start element flag
0	Element not contains attributes
0	Element is a not known element
1	Element is in default namespace
	g (StringEnc of element name)
7: characters(" ") (ignore)	
8: characters(" ") (ignore)	
9: startElement("http://www.w3.org/2000/svg","rect","rect","arguments")	
00	Start element flag
1	Element contains attributes
0	Element is a not known element
1	Element is in default namespace
	rect (StringEnc of element name)
0	Not a known attribute
1	Is default namespaces
	x (StringEnc of attribute name)
	30 (StringEnc of attribute value)
0	More attribute are coming

Binary code	Text value/Description
0	Not a known attribute
1	Is default namespaces
	y (StringEnc of attribute name)
	30 (StringEnc of attribute value)
0	More attribute are coming
0	Not a known attribute
1	Is default namespaces
	width (StringEnc of attribute name)
	40 (StringEnc of attribute value)
0	More attribute are coming
0	Not a known attribute
1	Is default namespaces
	height (StringEnc of attribute name)
	40 (StringEnc of attribute value)
0	More attribute are coming
0	Not a known attribute
1	Is default namespaces
	fill (StringEnc of attribute name)
	green (StringEnc of attribute value
1	Last attribute
10: endElement(”http://www.w3.org/2000/svg”,”rect”,”rect”)	
01	End element (rect)
11: characters(” ”) (ignore)	
12: startElement(”http://www.w3.org/2000/svg”,”g”,”g”,”arguments”)	
00	Start element flag
0	Element not contains attributes
1	Element is a known element
1	Element is in default namespace
01	Element identifier for g
13: characters(” ”) (ignore)	
14: startElement(”http://www.w3.org/2000/svg”,”rect”,”rect”,”arguments”)	
00	Start element flag
1	Element contains attributes
1	Element is a known element
1	Element is in default namespace
10	rect element identifier
1	A known attribute
000	Attribute identifier of x
	40 (StringEnc of attribute value)
0	More attribute are coming
1	A known attribute
001	Attribute identifier of y
	40 (StringEnc of attribute value)
0	More attribute are coming

Binary code	Text value/Description
1	A known attribute
010	Attribute identifier of width
	40
0	More attribute are coming
1	A known attribute
011	Attribute identifier height
	40
0	More attribute are coming
1	A known attribute
100	Attribute identifier fill
	red
0	More attribute are coming
0	Not a known attribute
0	Not default namespaces
1	namespace identifier for xlink
	href
	test2.svg
1	Last attribute
15: endElement(”http://www.w3.org/2000/svg”,”rect”,”rect”)	
01	End element (rect)
16: characters(” ”) (ignore)	
17: endElement(”http://www.w3.org/2000/svg”,”g”,”g”)	
01	End element (rect)
18: characters(” ”) (ignore)	
19: endElement(”http://www.w3.org/2000/svg”,”g”,”g”)	
01	End element (rect)
20: characters(” ”) (ignore)	
21: startElement(”http://www.w3.org/2000/svg”,”text”,”text”,”arguments”)	
00	Start element flag
1	Element contains attributes
0	Element is a not known element
1	Element is in default namespace
	text
0	Not a known attribute
1	Is default namespaces
	x
	40
0	Last attribute
0	Not a known attribute
1	Is default namespaces
	y
	80
1	Last attribute
22: characters(”Hello to you”)	

Binary code	**Text value/Description**
10	Start characters flag
	Hello to you
23: endElement("http://www.w3.org/2000/svg","text","text")	
01	End element (rect)
24: characters(" ") (ignore)	
25: endElement("http://www.w3.org/2000/svg","svg","svg")	
01	End element (rect)
26: endPrefixMapping() (ignore)	
27: endPrefixMapping("xlink") (ignore)	
28: endDocument() (close file socket/stream)	

Appendix C

SVGForms abstract

- Author: Jan-Klaas Kollhof
- Published at: http://www.svgopen.org/2005

Forms have always played a big part in the world wide web for transmitting data, entered by the user, to a server for processing. With SVG becoming more and more a format not only to describe vector graphics but to build web applications with, developers will need widgets a user can use to input data. There are a number of existing libraries to build rich user interfaces in SVG. SVGForms does not try to replace these but rather provide a light weight widget library for forms.

The integration of sXBL(SVG's XML Binding Language) in the 1.2 specification gives SVG a great tool to create reusable components. SVGForms will use sXBL to describe a basic widget set very similar to HTML forms. Besides the visible widgets it will provide the non visible components for transmitting the forms data to the server and handling of the data returned.

The paper will give an introduction to the SVGForms widget library by using real world examples. It will show how easily it can be used from within an SVG document and how the developer can integrate it with existing server side tools(ASP, JSP, PHP, ..., CGI scripts). Furthermore the paper will discuss the similarities and differences to HTML forms as well as security concerns and limitations.