XML11 – An Abstract Windowing Protocol

Arno Puder

San Francisco State University
Computer Science Department
1600 Holloway Avenue
San Francisco, CA 94132
`arno@sfsu.edu`

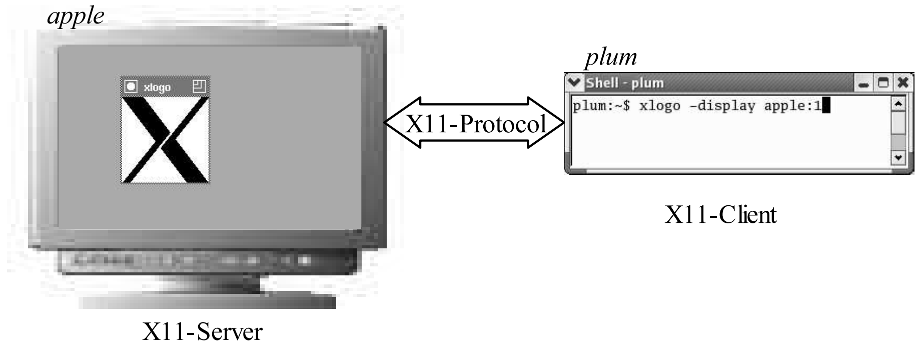
Abstract

This paper describes XML11, an abstract windowing protocol inspired by the X11-protocol develop by MIT. XML11 is an XML-based protocol that allows asynchronous UI updates of widgets to an end-device. To overcome high-latency connections, XML11 allows migration of application logic to the end-device. Implicit middleware enables transparent interaction between the end-device and the server. The middleware is implicit, because the programmer is unaware of the distribution. The prototype implementation of XML11 runs in any standard web browser without Java capabilities on the client-side and replaces AWT/Swing on the server-side. This also allows us to expose legacy AWT/Swing applications as web applications.

1 Motivation

The X11-protocol was developed by MIT in 1984 (see [14]). The X11-protocol distinguishes between a client and the X11-server (see Figure 1). The X11-server runs on a host with a graphical display. One or more clients that run on different machines can render their output on the remote X11-server by way of the X11-protocol. The X11-protocol is characterized by pixels and rectangles; the X11-server has no knowledge of widgets such as buttons or list-boxes. These have to be drawn manually by a client. This explains the many different looks-and-feels of X11-applications. One consequence of this is relatively high protocol overhead. The X11-protocol works best on high-bandwidth, low-latency communication networks.

The X11-protocol is therefore not suitable for wide area networks with high-latency connections. The World-Wide Web (WWW) has established itself successfully in this domain. One can draw a comparison between the WWW and the X11-protocol: a standard web browsers serves a similar purpose as the



```
plum:~$ xlogo -display apple:1
```

Fig. 1. The X11-protocol.

X11-server in the way that both act are generic interfaces to a user. However, a web browser has knowledge of basic widgets such as labels and button and a browser is usually also capable of executing code. Initially the WWW was document centric and was only meant to serve documents in a distributed environment. With the rise of the WWW, new standards such as HTML-Forms and Common Gateway Interface (CGI) have extended this document-centric model to allow operational interaction between client and server. Parallel to the web technologies, new kind of end-devices such as PDAs or mobile phones have entered the scene that serve as generic clients to remote services. The document-centric protocols of the world-wide web are even less suitable for these devices.

This paper proposes a new protocol that we call XML11. It is inspired by the X11-protocol, but based on XML to define the PDUs (Protocol Data Unit). XML11 makes use of advanced capabilities found in typical end-devices such as the ability to render widgets and execute application logic. In Section 2 we describe the XML11 framework and present a detailed overview of the XML11-protocol. Section 3 describes our prototype implementation of XML11 that allows legacy Java/Swing applications to be controlled from a non-Java aware web browser. In Section 4 we discuss related work and the paper concludes with an outlook in Section 5.

2 Framework

We describe our framework in several steps. Section 2.1 introduces the design goals of XML11. Section 2.2 describes the XML11 protocol. Section 2.3 explains the code migration framework embedded in XML11 and finally in Section 2.4 we discuss the idea of implicit middleware as part of the code migration framework.

2.1 Design goals

Before introducing the design goals for XML11, we first make some assumptions of the underlying technology. These assumptions are derived from today's technology in use. We make the following assumptions regarding the capabilities of an end-device: first we assume that the end-device has its own windowing system that knows about widgets such as buttons, labels, or checkboxes. Those widgets are rendered using the native look-and-feel of the end-device. Furthermore we assume that the end-device provides an execution platform that is capable of running business logic. The execution platform is expected to be Turing-complete, but we do not impose any specific programming language.

Two possible end-devices that fit those assumptions are standard web browsers or PDAs. Both are capable of rendering a user interface and both allow the execution of business logic. In case of the web browser, widgets can be rendered using standard HTML elements. Literally all browsers offer a Turing-complete execution platform either through JavaScript, Java, or ActiveScript. All browsers support a common subset of JavaScript.

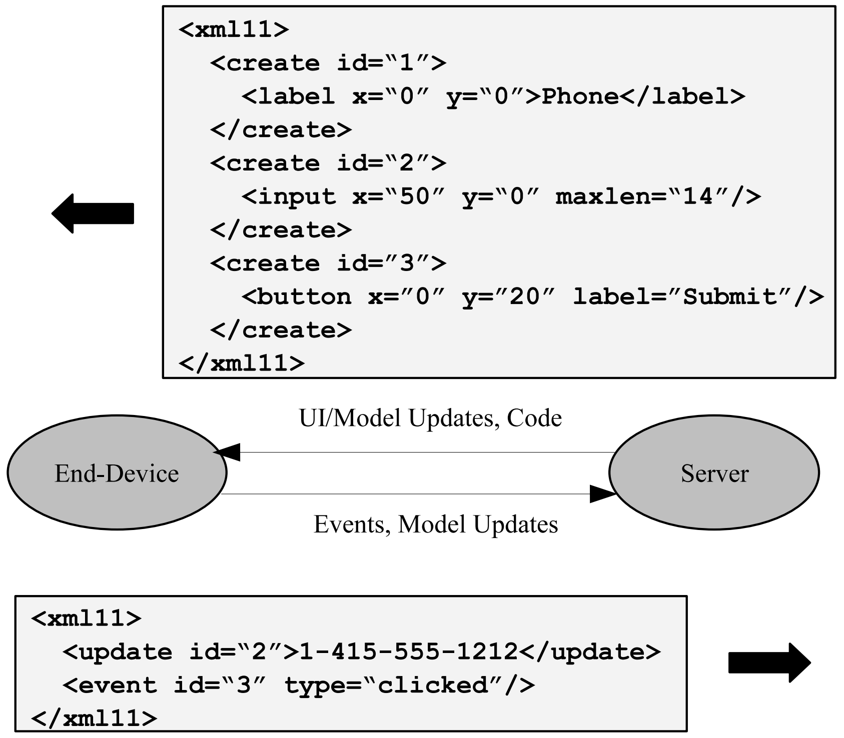
Another assumption we make is that the end-device is connected to the server by a low-latency, medium-bandwidth network connection. Based on those assumptions, XML11 has the following design goals:

- Device independence: XML11 should not make any specific assumption about the end-device except those outlined above.
- Code migration: To support high-latency connections, XML11 supports code migration of business logic to the end-device.
- Operational interaction: The migrated code can do remote object invocations to the server side.

2.2 XML11 Protocol

The core of our framework is the XML11 protocol. Its name is inspired by the X11 protocol because it serves a similar purpose. Unlike the X11 protocol, XML11 is not a binary protocol, but an XML-based protocol. This means that all PDUs are expressed as XML documents. In terms of a protocol, we will adopt the following terminology: the *server* hosts the application whose UI is rendered remotely. The *end-device* is the client where the user interface is rendered and that interacts with the remote server.

XML11 is an asynchronous, event based protocol. The server and the end-device send PDUs in response to certain events. On the server side a typical



```
<xml11>
  <create id=“1”>
    <label x=“0” y=“0”>Phone</label>
  </create>
  <create id=“2”>
    <input x=“50” y=“0” maxlen=“14”/>
  </create>
  <create id=”3”>
    <button x=”0” y=”20” label=”Submit”/>
  </create>
</xml11>
```

```
<xml11>
  <update id=“2”>1-415-555-1212</update>
  <event id=“3” type=“clicked”/>
</xml11>
```

Fig. 2. XML11 Protocol Overview.

event is the creation or update of a widget. On the end-device an event is raised whenever the user interacts with the UI, such as pressing a button. Figure 2 gives a conceptual overview of the XML11 protocol. The server sends a PDU to the end-device requesting it to draw a label, an input field and a button. Each of these widgets have additional attributes that describe features of the particular widget (such as position).

The end-device, receiving this PDU, will render the UI accordingly. The user can now interact with the UI and fill in the input field. When pressing the button, the end-device sends a PDU back to the server. The PDU, as shown in Figure 2, contains a model update that informs the server of the value the user has provided in the input field. Furthermore, the PDU contains the type of event; in this case a button-click. The server would queue the event with the application. During the processing of the event, the application will send itself further updates in form of XML11-PDUs to the end-device.

Conceptually, this is similar to the HTTP/HTML protocol of the WWW. However, there are some important differences: in the WWW, only complete HTML-pages are transported to the browser. In XML11, individual widgets are sent to the end-device. HTTP is also a strict client/server protocol where the web browser always initiates the interaction. In XML11, the server can also send PDUs independently to the end-device. In this regard, XML11 is more similar to the X11-protocol. However, while the X11 protocol is about pixels and lines, XML11 knows about widgets. Table 1 gives an overview of

XML11 Element	**Description**
`<property>`	Properties of the end-device.
`<button>`, `<label>`, `<input>`	Standard GUI widgets. Based on XUL.
`<create>`, `<new-value>`, `<destroy>`	Create, update and destroy widgets.
`<event>`	UI events raised by client and sent to server.
`<update>`	Model updates sent by client and server.
`<code>`	Code migration framework.

Table 1
XML11 Protocol Elements

the various XML11 PDUs.

The end-device communicates its capabilities (such as screen resolution, supported fonts and widgets) to the server via the `<property>` tag. The XML-tags used to describe widgets are based on XUL (see [19]). XUL is an XML-based user interface description language developed by the Mozilla project. Mozilla-based browsers use XUL to describe their own modal dialogs. In XML11, widgets described through XUL can be created and destroyed individually by appropriate tags `<create>` and `<destroy>`. The XML11 protocol further defines tags to relay events (such as mouse clicks) from the end-device to the server, as well as tags that allow code migration.

2.3 Code Migration

Our framework includes code migration to overcome network latencies by moving some of the application logic to the client. Table 1 in the previous section already introduced the `<code>` element of the XML11 protocol, which will be explained in more detail in this section. If the server wants to migrate some application logic to the client, it includes the implementation between the `<code>` element. Any programming language would be a suitable candidate for the implementation, but since we make no assumption on what languages are supported by the client, we decided for an XML-based programming language. The benefit of using XML is that is integrates nicely with the rest of the XML11 protocol.

Several XML-based programming languages exist (see [3,10,11,6,9]), however those languages are impractical for day-to-day programming. First and foremost programmers have to learn a new programming language. Someone using this approach would need to master a new programming language for which no tools (such as smart editors or syntax checkers) exist. Another problem results in the fact that XML tends to be very "verbose". By this we mean that it takes on the average more lines of code to express an algorithm in XML than in other high-level languages. This is because of the rigid syntax that XML imposes on the structure of a document. For these reasons we took a different approach.

The XML-based programming language that is used to migrate code is based on the instruction set of the Java Virtual Machine. The instruction set, also referred to as *byte code*, resembles the machine code of other hardware architectures. It is interesting to note that Sun Microsystems as the inventor of Java never standardized an assembly language of the their own byte code. Several assemblers were developed, but they had to invent their own syntax. The most commonly used assembly syntax stems from the Jasmin project (see [13]).

Our XML-based programming language is based on the Java byte code. This effectively defines an assembly language for the Java virtual machine whose syntax is based on XML. Since we model the syntax very closely to the Java byte code instructions, there is a direct bidirectional mapping between Java class files and our XML-based programming language. Since the XML-based programming language is closely related to the byte code of the Java virtual machine, we call this language XMLVM.

XMLVM is just an intermediate representation. When migrating code to the end-device, the XMLVM program has to be translated to the target programming language. To which programming language the XMLVM will be translated depends on the capabilities of the end-device. One way to translate XMLVM to another programming language is via an XSL-stylesheet. XSL is a technology whereby an XML document can be translated to some other representation. The translation rules are expressed themselves in XML (see [16]).

The following example illustrates this. The XMLVM instruction `<iadd>` pops two integers from the top of the stack and then pushes the sum onto the stack. If the end-device is a standard web browser and we further assume that the only supported language inside the browser is JavaScript, we need to translate `<iadd>` to JavaScript. The following XSL-excerpt describes a transformation rule, that will generate some JavaScript code whenever the `<iadd>` instruction is encountered in XMLVM:

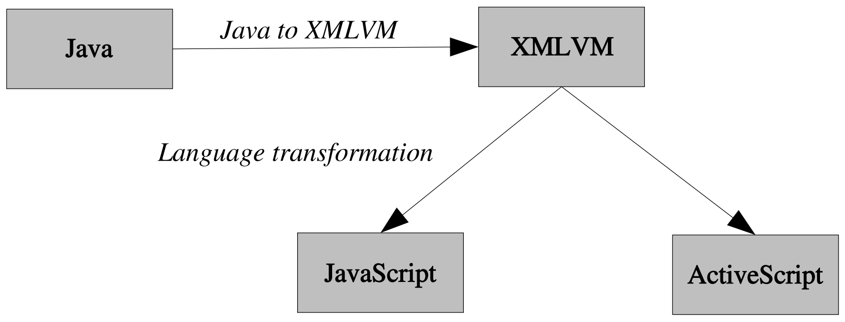


Fig. 3. XMLVM.

```
<xsl:template match="iadd">
  <xsl:text>
    __op2 = __stack[--__sp]; // Pop operand 1
    __op1 = __stack[--__sp]; // Pop operand 2
    __stack[__sp++] = __op1 + __op2; // Push sum
  </xsl:text>
</xsl:template>
```

The variables shown in this example (e.g., `__op1`, `__sp`) are declared as part of the XSL-translation for every method. They are used to mimic the stack machine. Every XMLVM instruction is mapped to JavaScript via its own XSL-template. The set of all XSL-templates is referred to as *language transformation* in Figure 3. Different XSL-templates can be created for different programming languages. While this is a very simple mapping, it is easy to implement and usually efficient enough for code running on the end-device. The implementation section will provide more details on the translation process.

One question we have not yet addressed is which portions of the application is migrated to the client. This is not a trivial task and depends on many different factors. A likely candidate for functionality to be migrated to the end-device is user input validation which helps to overcome communication latencies. On the other hand, code that requires resources only located on the server-side cannot be migrated. For our prototype implementation, we have chosen a pragmatic approach whereby an external configuration file determines which classes to migrate to the end-device.

2.4 Implicit Middleware

Generally it is not possible to migrate the complete business logic to the end-device. For one, the size of the application might be too big to allow to migrate the complete code to the end-device. Another reason could be that the application uses fixed resources (such as databases or special purpose hardware) that cannot be migrated. For all practical purposes, only part of the business logic will be migrated to the end-device. The implication is that this might necessitate remote object invocations. This will invariably happen when an object residing of the end-device makes an invocation to an object located on the server-side.

Remote invocation can easily be handled by a middleware. Using middleware ensures the notion of transparency where local invocations cannot be distinguished from remote invocations. This transparency is typically achieved by proxy-objects because the proxy offers the same interface as the remote object. The proxy marshals actual parameters and forwards them to the remote object. Since XML11 is an XML-based protocol, we have decided to use SOAP messages (see [18]) for remote method invocations. Whenever remote invocations are necessary, SOAP requests and replies are exchanged to pass the actual parameters.

The novel idea in our XML11 framework is how the proxies are generated. Web Services allow the description of object interfaces through the Web Services Definition Language (WSDL) (see [17]). WSDL is itself an XML document that describes the specifics of an object interface. Based on the WSDL specification, a compiler can generate the proxy objects. In XML11 we bypass the generation of WSDL. The signatures of remote operations are derived from the XMLVM version of the remote class. To illustrate this, consider the following simple Java-class:

```
public class Calculator {
    public int add(int x, int y)
    {
        return x + y;
    }
}
```

In the following we assume that class `Calculator` is not migrated to the end-device, so that any part of the migrated business logic using this class will need a proxy in order to access remote instances of Calculator. Even though `Calculator` is not migrated to the end-device, we can translate it to XMLVM:

```
<method name="add" isPublic="true" stack="2" locals="3">
  <signature>
    <return type="int" />
    <parameter type="int" />
    <parameter type="int" />
  </signature>
  <code>
    <var name="this" id="0" type="Calculator" />
    <var name="arg0" id="1" type="int" />
    <var name="arg1" id="2" type="int" />
    <iload type="int" index="1" />
```

```
      <iload type="int" index="2" />
      <iadd />
      <ireturn />
    </code>
  </method>
```

This code shows how the two actual parameters are first pushed onto the stack using <`iload`> and then how the sum is computed and returned using <`iadd`> followed by <`ireturn`>. Also note that the XMLVM version of class `Calculator` contains the complete signature of method `add`. A tool constructs the proxy for class `Calculator` by inspecting this signature and then generating the marshaling code for method `add`. The marshaling code is generated in XMLVM and replaces the actual implementation of method `add` of the <`code`>-tag. By generating the mashaling code in XMLVM, we can easily translate the implementation of the proxy to the target language supported by the end-device using the same XSL-translations used for code migration. The reason we call this approach implicit middleware is because the use of middleware is completely transparent for the application programmer.

3 Prototype Implementation

We have implemented a prototype of XML11 based on the framework introduced in the previous section. Figure 4 shows the architecture of our implementation. For the end-device we use a regular web browser such as Firefox or IE. However, we do not require or assume the availability of a Java VM inside the browser. The client-side of the XML11 protocol is implemented in JavaScript which is commonly available in all browsers. We use the JSolait JavaScript library (see [7]) to transport the XML11 PDUs via HTTP-POST requests. To implement the event-driven model described in the previous section, we use deferred HTTP-POST responses. A response to the HTTP-POST is deferred until the server wants to send an XML11 PDU to the end-device.

On the server-side, the application is embedded inside the application server Tomcat (see [1]). The server-side implementation of the XML11 protocol is done in Java. We have chosen to implement the server-side protocol of XML11 as AWT (Abstract Window Toolkit) peer components by implementing the abstract base class `java.awt.Toolkit`. This approach guarantees that the AWT-API is unchanged which means any AWT-application can render its user interface via XML11 inside a standard web browser.

We implemented the XMLVM programming language using the Byte Code Engineering Library (BCEL, see [5]). BCEL allows to analyze the contents of a Java class-file, which makes it easy to generate XMLVM out of any class-file.

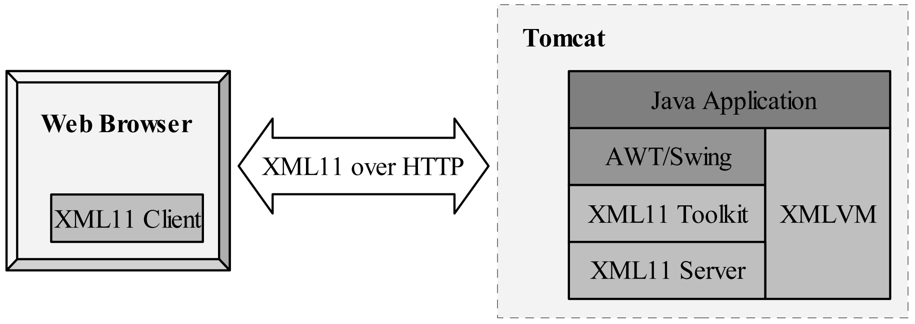


Fig. 4. Reference implementation.

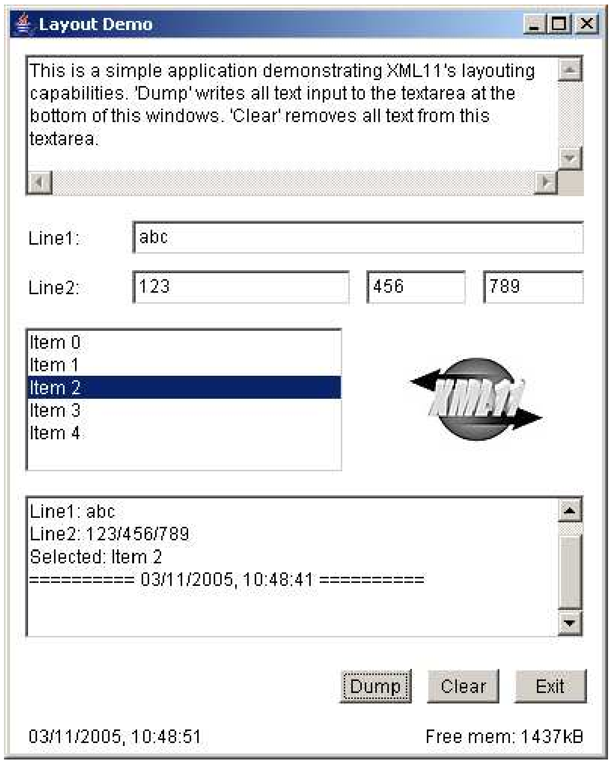
We use XSL to translate the resulting XMLVM to the target language, which in our case is JavaScript.

A complete scenario using XML11 looks as follows: the user types in a URL into the browser. Based on the URL, the browser issues an HTTP-POST to the application server. The parameters (which application to run and which parts of the application to migrate to the browser) are encoded in the URL. The application server launches the application and responds to the browser with a bootstrap library written in JavaScript. This JavaScript code implements the client-side of the XML11 protocol. As the application runs within the application server, widgets are created via XML11's own AWT toolkit. This results in appropriate XML11 PDU to be sent to the client. These asynchronous updates that are sent from the server to the browser are transported via responses to an HTTP-POST.

As a proof-of-concept we have implemented a moderately complex application that uses some common widgets. Figure 5 shows a screenshot of the application running as a Java desktop application. The application uses widgets `Label`, `Button`, `List`, `TextField`, and `TextArea`. It also uses a `Panel` with a custom `paint`-method for the image shown in the screenshot. Furthermore, the application uses the following layout managers: `GridBagLayout`, `BorderLayout`, and `FlowLayout`. At the bottom of the window, the current date and time are updated every second. The application consists of 650 lines of Java code.

Using XML11, this application can be run without modifications (or even the need to re-compile the application) as a web-application. The application is rendered inside a browser with exactly the same look-and-feel as the Java desktop application shown in Figure 5. The asynchronous updates of the date and time (at the bottom of the screenshot) are handled by fine-granular updates. I.e., it is not necessary to re-load the complete HTML page whenever the time and date on the bottom of the window are updated. XML11 is implemented on the client-side using only HTML, DOM, and JavaScript. A Java-plugin is not required.

In order to compare XML11 with the X11 protocol, we have conducted some



This is a simple application demonstrating XML11's layouting capabilities. 'Dump' writes all text input to the textarea at the bottom of this windows. 'Clear' removes all text from this textarea.

Line1: abc

Line2: 123 456 789

Item 0
Item 1
Item 2
Item 3
Item 4

```
Line1: abc
Line2: 123/456/789
Selected: Item 2
========== 03/11/2005, 10:48:41 ==========
```

03/11/2005, 10:48:51 Free mem: 1437kB

Fig. 5. Example application.

benchmarks by measuring the size of the PDUs going back and forth between end-device and the server. We used the application shown in Figure 5 in three different setups:

(1) as a native X11-application.
(2) as an XML11 application without code migration.
(3) as an XML11 application with code migration.

For each of the three setups we measured the network traffic for different stages during the life-time of the application:

Library: how much library code is transferred. This does not apply to X11. For XML11, the library code includes any initial startup libraries implemented in JavaScript.

Code: the overhead introduced by code migration. The application described earlier translates to 2.600 lines of JavaScript code.

Initial screen: how much traffic is generated in order to display the initial screen. This includes all the widgets as well as the image that is part of our demo application.

Asynchronous update: how much traffic is generated to update the date and time in the application. We measured the traffic for exactly one update.

User interaction: we measure the traffic generated by a fixed pattern of user interaction. This includes some input to the various widgets as well as

	X11	XML11	
		W/O Migration	With Migration
Library	0	92.660	92.660
Code	0	0	51.656
Initial screen	203.904	15.016	15.016
Async. update	15.804	1.964	1.964
User interaction	25.036	3.510	0
Total	244.744	113.150	161.296

Table 2
PDU sizes in Bytes.

clicking one of the buttons.

The results of our measurements are shown in Table 2. We used `xmon` to measure the traffic generated by the X11-protocol (see [12]) and `tcpmon` from the Apache Axis project to measure the HTTP traffic (see [2]). As can be seen in Table 2, XML11 significantly reduces the amount of traffic over the network. For larger applications the library overhead for XML11 will become less significant making XML11 even more efficient than X11.

Table 2 only shows the amount of traffic generated by our demo application in different scenarios. We do not show the number of PDUs that were exchanged. In particular the row labelled "User interaction" is very different for the various setups. For the X11-protocol, virtually every single keystroke results in a PDU-exchange with the server. The values shown for XML11 only stem from one single HTTP interaction, which underlines the fact that XML11 is better suited for high-latency connections than X11.

4 Related Work

Several projects – commercial and Open Source – exist that aim at providing an easy migration path for legacy Java applications to web applications. WebCream is a commercial product by a company called CreamTec (see [4]). They have specialized in providing AWT and Swing replacements that render the interface of the Java application inside of a web browser. WebCream makes use of proprietary features of Microsoft's Internet Explorer and therefore only runs inside this browser.

Two Open Source projects, both hosted at SourceForge, follow the same idea of exposing Java desktop applications as web applications. The first one is called WebOnSwing (see [15]). Unlike WebCream, this project is not tailored for a particular browser. One feature offered by WebOnSwing are templates that allow to change the look-and-feel of the application that is rendered inside the browser. Another project with similar features, but not quite as mature, is SwingWeb (see [8]).

There are several differences between these projects and XML11. First and foremost, XML11 defines a technology-independent protocol. Although our prototype also uses Java as a foundation, we are not limited to Java. The XML11 protocol also decouples client and server side technologies and allows us to support different end-devices. The XML11 protocol also supports asynchronous updates which none of the projects discussed in this section supports. This means that for WebCream, WebOnSwing, and SwingWeb, every change on the user interface requires these technologies to re-load a new HTML page. None of these project supports code migration and implicit middleware.

5 Conclusions and Outlook

XML11 defines a client/server protocol similar in spirit to the X11-protocol developed by MIT. The difference is that XML11 is at a higher level of abstraction, making use of advanced capabilities often found in end-devices. XML11 includes a code migration framework that allows the migration of business logic to the end-device in order to compensate for network communication latencies. Our prototype implementation of XML11 embeds the server-side of the XML11 protocol in Java's AWT/Swing. The client side implementation of XML11 runs inside any standard web browser using JavaScript.

XML11 combines many different techniques from other areas of research. It uses middleware to allow transparent interaction between the end-device and the server. The middleware is integrated implicitly, which means that the application programmer using XML11 effectively is not aware middleware is used. XML11 also makes use of cross-language translation to implement a code migration framework. This is achieved by creating an assembly language for Java byte code which we call XMLVM. The UI is described through XUL, but unlike XUL, XML11 allows for dynamic modifications of the UI.

We plan to apply our framework for different domains. For one, we plan to implement a PocketPC version of the XML11 client-side protocol. Furthermore, we will investigate XML11 server-side implementations for different platforms such as .NET. It will be interesting to see the compatibility of XMLVM with .NET's Intermediate Language.

References

[1] Apache Foundation. *Jakarta - Tomcat*. http://jakarta.apache.org/tomcat/.

[2] Apache Foundation. *WebServices Axis*. http://ws.apache.org/axis/.

[3] G.J. Badros. A markup language for java source code. May 2000. http://www.cs.washington.edu/research/constraints/web/badros-javaml-www9.pdf.

[4] CreamTec, LLC. *WebCream*. http://www.creamtec.com/webcream/.

[5] Markus Dahm. Byte code engineering. Java Informations Tage, pages 267–277, 1999.

[6] W. Emmerich, C. Mascolo, and A. Finkelstein. Implementing incremental code migration with xml. pages 397–406. In M. Jazayeri and A. Wolf, editors, Proc. 22nd Int. Conf. on Software Engineering (ICSE2000) Limerick, Ireland, ACM Press., June 2000. http://www.cs.ucl.ac.uk/staff/W.Emmerich/publications/ICSE2000/MobXML/mobxml.pdf.

[7] Jan-Klaas Kollhof. *JSolait - A JavaScript library*. http://www.jsolait.net/.

[8] Tiong Hiang Lee. *SwingWeb*. http://swingweb.sourceforge.net/swingweb/.

[9] Jonathan I. Maletic, Michael L. Collard, and Andrian Marcus. Source code files as structured documents. pages 289–292, June 2002. http://www.sdml.info/papers/iwpc02.pdf.

[10] E. Mamas and K. Kontogiannis. Towards portable source code representation using xml. pages 172–182. Proceedings of the Seventh Working Conference on Reverse Engineering, IEEE Computer Society Press, Brisbane Australia, November 2000.

[11] G. McArthur, J. Mylopoulos, and S.K.K. Ng. An extensible tool for source code representation using xml. pages 199–208. Proceedings of the Ninth Working Conference on Reverse Engineering, IEEE Computer Society, Richmond, Virginia, USA, October 2002.

[12] Greg McFarlane. *Xmon - X Protocol Monitor*. http://sourceforge.net/projects/xmon/.

[13] Jonathan Meyer. An assembler for the java virtual machine. 1996. http://jasmin.sourceforge.net/.

[14] The Open Group. *X Window System (X11R6) Protocol*, 1999.

[15] Fernando Petrola. *WebOnSwing*. http://webonswing.sourceforge.net/xoops/.

[16] W3C. *XSL Transformations*, 1999. http://www.w3.org/TR/xslt.

[17] W3C. *Web Services Description Language (WSDL) Version 1.1*, 2001. http://www.w3.org/TR/wsdl.

[18] W3C. *Simple Object Access Protocol (SOAP) Version 1.2*, 2003. http://www.w3.org/TR/soap.

[19] XULPlanet. *XUL*. http://www.xulplanet.com/.