An Instant Messaging Framework for Flexible Interaction with Rich Clients

Matthias Book, Volker Gruhn
Chair of Applied Telematics/e-Business, Dept. of Computer Science, University of Leipzig
Klostergasse 3, 04109 Leipzig, Germany; Tel. +49-341-97-32337, Fax +49-341-97-32339
{book, gruhn}@ebus.informatik.uni-leipzig.de

Abstract

Today, we are seeing an increasing number of software applications that users want to use anywhere, anytime. Such mobile applications often deliver their user interfaces (UIs) to client devices over the World Wide Web. However, web-based UIs cannot provide the same level of usability as window-based UIs on mobile devices with their small screens and occasional network dropouts. To address this challenge, we present a UI framework that combines the usability of a full-featured UI with the flexibility of a thin presentation logic: We exchange interface specifications and events between the application logic on the server and a generic UI rendering engine on the client device using an instant messaging infrastructure. The paper gives an overview of the framework architecture and the features of the communication protocol, and discusses performance measurements obtained on a public network.

1. Introduction

Many information systems that are intended for mobile access are implemented as thin client applications today for reasons such as reduced infrastructure cost, reduced deployment and maintenance effort, and increased user flexibility. Often, these thin client applications take the form of web-based applications whose hypertext user interface (UI) can be easily rendered by web browsers, which are ubiquitous tools on most client devices by now, while all application logic remains on a central server [1].

However, web-based UIs still cannot offer the same level of usability that window-based applications do, since they do not provide intrinsic support for complex widgets (e.g. tree views, type-ahead combo boxes etc.) or interaction patterns (e.g. direct manipulation of objects, multiple window panes, context menus etc.) [8]. Besides these limitations of HTML, the underlying request-response communication pattern imposed by HTTP restricts web-based UIs on mobile devices further: Not only must any interaction be initiated by the user (since the server cannot update the UI on its own by "pushing" data to the client [13]), but the combination of content and markup produces a high communications overhead. Recently, a combination of web technologies collectively termed Ajax (Asynchronous JavaScript and XML) [2] is being touted as the solution to these problems. The approach uses JavaScript to manipulate one web page's Document Object Model (DOM) at run-time according to XML-based update instructions requested from the server. However, while Ajax provides a smoother user experience, largely due to the elimination of page jumps, it still relies on the same web technologies and therefore requires quite a bit of communications overhead to present a more sophisticated UI. This overhead can be a critical factor on mobile networks, which are often characterized by low bandwidth and high transmission costs.

Due to their limited usability and high overhead, we consider web-based UIs sub-optimal for mobile devices. The use of mobile devices' native UIs would reduce these disadvantages, however, they derive most of their expressive power from the close coupling between application and presentation logic, which seems to preclude a thin client approach where the application logic resides on a remote server. In order to retain the usability and efficiency of devices' native UIs without giving up the flexibility of a thin client architecture, this paper presents a user interface framework that combines their benefits. Our framework enables developers to build thin client applications with window-based UIs, where clients communicate with the server using an instant messaging (IM) protocol. By exchanging user interface descriptions and events over public IM networks, clients and servers can communicate very flexibly, and the use of a powerful user interface description language allows developers to let sophisticated GUIs be rendered on mobile clients.

In the following sections, we present the architecture and protocol of our IM framework (Sect. 2), as well as the results of initial performance measurements (Sect. 3). We conclude with an overview of related work (Sect. 4) and an outlook on future research opportunities (Sect. 5).

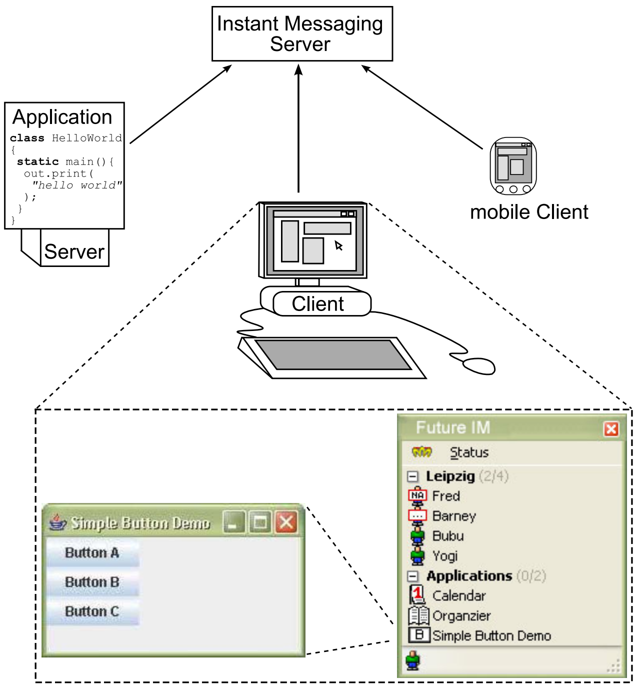
2. Instant Messaging Infrastructure

Request-response communication as implemented in HTTP is a one-sided, synchronous pull mechanism: Only the client can initiate communication and needs to wait for a response before it can proceed with the next interaction. For highly interactive thin client UIs, a more flexible communication scheme would be more desirable. Ideally, both the client and the server should be able to transmit (i.e. push) messages anytime on their own initiative, and without having to wait for a response (i.e. asynchronously). These messages should require low bandwidth in order to save costs on mobile networks, and be delivered quickly in order to avoid the sluggish response often experienced in web-based applications.

Existing IM protocols such as the Extensible Messaging and Presence Protocol (XMPP) [11] fulfill these requirements: Typically employed for chat networks, they are tailored to the asynchronous, nearly instantaneous transmission of small messages between peers, and thus provide an ideal medium for communicating UI events (i.e. user gestures and interface reactions) between a thin client and its application server.

In order to keep the client as thin as possible, we do not want to deploy any application-specific code on the client, but only require the presence of an IM client for handling communications, and a generic GUI engine for rendering the interface. Much like a web browser interpreting and rendering HTML code, this GUI engine interprets and renders a window-based interface described in the XML User Interface Language (XUL) [6] or a comparable format.

Figure 1 gives a coarse overview of this instant messaging infrastructure for thin client applications: The mobile or stationary client devices are running an IM client that shows the typical "buddy list" of other users on the IM network. In addition to showing online users, however, it also shows the applications that are currently accessible via the IM network. When the user selects an application from the list, the GUI engine will download and render the respective UI description markup. While the user works with the application, all client-server interactions are transmitted as instant messages to the application server, which is connected to the central IM server just like any other peer. Since both thin clients and application servers are equal peers on the IM network, they can communicate freely over the IM infrastructure and use its special features, e.g. peers' presence information (see Sect. 2.1.3).



```
class HelloWorld
{
 static main(){
  out.print(
   "hello world"
  );
 }
}
```

Figure 1. IM-based thin client infrastructure

2.1. Framework Features

To support the development of applications that can be accessed by IM-based thin clients, as described in the previous section, we developed a framework that encapsulates the technical aspects of the IM-based communication and provides high-level services like session management to the application. Our framework implementation is independent of a concrete user interface description language and IM transport protocol, but application developers will typically want to use the popular XUL language and the standardized XMPP transport protocol.

In order to reduce mobile bandwidth usage, the framework transmits the complete UI description (i.e. the specification of all windows and their widgets) to each client only once. Since the UI structure typically remains static over time, the client can cache this description and re-use it the next time the user works with the same application, without having to receive it again. For devices without sufficient cache memory, or complex applications whose complete UI description is too large to download all at once, it is also conceivable to transmit only parts of the interface initially and download other chunks of the UI description dynamically if the user enters other areas of the application.

After the UI description has been transmitted, the GUI engine can render the interface and let the user work with the application. From now on, only user and application events need to be transmitted over the IM network anymore. Each event is encapsulated in a single instant message that contains all required parameters (e.g. type of the user gesture, entered or selected data, etc.), and is sent to the application server through the IM network. If necessary, the application server may then react by sending back an instant message that contains instructions for updating the view rendered on the client. The application server may also send such an event on his own initiative (e.g. triggered by the expiry of a timer, the completion of a time-intensive transaction, the fulfillment of a certain condition etc.) to modify the UI without previous user interaction. The application's instant messages do not contain low-level drawing commands (as in e.g. the X Window System [12]), but rather instructions for modifying the DOM of the UI description (as in the Ajax approach [2]). Based on the updated UI description, the GUI engine can then display or hide certain windows, enable or gray out certain widgets, fill content areas with application data etc.

In the following subsections, we will present core features of the framework that reflect the special characteristics and challenges that need to be addressed in an IM-based thin client infrastructure, in contrast to e.g. a web-based infrastructure.

2.1.1. Single Sign On. A major advantage of an IM-based vs. a web-based thin client infrastructure is that all users must authenticate themselves to the IM server in order to use the IM network, so each user is uniquely identifiable. Consequently, IM-based applications do not need to implement their own authentication mechanisms, but can rely on the IM server to take care of checking the users' credentials (which is also more convenient for users, who need to log in only once to use all applications). An application may only require additional authentication if it does not trust the IM server's authentication process. Of course, even with successful authentication, all applications still need to check the user's authorization for accessing certain application features. For additional security, XMPP provides mechanisms for end-to-end signing and encryption [10], which allow clients and applications to sign and/or encrypt instant messages using a public key infrastructure.

2.1.2. Session Management. Since all users on the network must log in to the IM server, there are no anonymous messages – each instant message is unambiguously associated with a certain sender and receiver. Therefore, IM-based applications do not have to implement the cumbersome session identification and expiry mechanisms known from web-based applications (such as passing a user ID along with every request in order to associate it with previous requests), but can simply associate all application data with users by their name.

In addition to employing user names for easy session management, our framework contains additional session management logic to distinguish "sub-sessions" that occur when the same user simultaneously works with several instances of the same application, i.e. when he has opened several windows of the same application (a situation that is virtually impossible to detect or handle in web-based applications, where users may clone browser windows).

The framework can also handle persistent sessions: In contrast to the X Window System [12], where the application running on the server is terminated when the user closes the client window or the connection to the server is lost, our framework will preserve the user's application data and UI state on the server, so users can close the thin client window and open it again at a later time in order to resume working where they left off. This way, users can suspend work with the thin client application (e.g. upon entering a zone without mobile network coverage) without loss of data.

2.1.3. Presence Information. An additional feature of our thin client framework that is unique to IM networks is the use of so-called presence information (PI) to convey status information about clients and applications. In peer-to-peer chat applications, the PI typically conveys user information such as "online", "offline", "away", etc., but it can also be used to provide application-specific status information.

Using the PI mechanism, application servers can publish information about their program version, server load or maintenance cycles. Transparently for the users, the client can then update its UI description if an application's PI indicates that a new version has been deployed. Or, if multiple instances of the same application are available on the IM network, clients can connect to the application instance that publishes the lowest server load in its PI, or switch to a different instance if the currently used instance indicates in its PI that it will be going down for maintenance.

Clients could also use the presence information to publish information about the quality of the mobile network connection they are currently experiencing, which applications might react to by adapting the volume or priority of transmitted messages. Using positioning information derived from GPS or cellular network topology data (if available), mobile devices could also publish their current geographical coordinates in the PI to enable applications to provide them with location-based content and services [7].

2.2. Thin Client Protocol

In order to enable thin clients and application servers to communicate with each other over the IM network, our framework uses an XML-based communication protocol whose commands and notifications are transmitted in instant messages. One might argue that an XML-based protocol introduces quite a bit of communication overhead, but the messages are typically very small, and the extensibility of XML allows flexible communication between clients and servers that do not all implement the same protocol.

The protocol elements we defined are wrapped into XMPP messages using a separate namespace, which is the default way of deploying extensions to the XMPP protocol. This way, clients that are unable to process these extensions will simply ignore them. Our approach could be implemented using other IM protocols as well, but unless those protocols used a similar extension mechanism, the commands would have to be incorporated into the body of the sent messages, making them visible to all clients.

For the sake of brevity, we will not go into the details of the various messages defined by our protocol, but just give an overview of the message classes here. They include:

- system and control messages, used for error handling, timer synchronization, and several other framework functions that do not affect the client or the application directly.
- session management messages, used to establish, resume or close sessions.
- UI description and definition messages, used to describe the user interface provided by the application.
- event messages, generated by user interactions or by triggers in the application logic. They are handled with highest priority.
- data transfer messages, used to transfer general data such as multimedia content or complete UI descriptions through the framework. They are handled with lowest priority.

2.3. Framework Architecture

Our framework comprises a set of components that implement the IM-based communication protocol outlined in Sect. 2.2. The framework serves as an interface between the IM transport protocol and the application or client logic, as shown in Fig. 2.

The framework is connected to the transport layer using a suitable third-party API, such as the open-source Smack API for the XMPP protocol [3] (to access a different API, only the Transport interface of the framework needs to be adapted). Since our framework only serves as a middleware for the presentation logic, it does not require exclusive access to the transport API. Rather, if the application or client need to communicate other domain-specific data over the IM network, they can access the transport API directly to do this, without having to go through the framework.

In order to allow thin clients' UIs to remain responsive while data is sent to the application server, all incoming and outgoing traffic is processed asynchronously. To ensure that the UI responds quickly to user interactions that require communication with the application server, all traffic is prioritized according to the message type: Events generated by user interactions need to be processed as soon as possible,

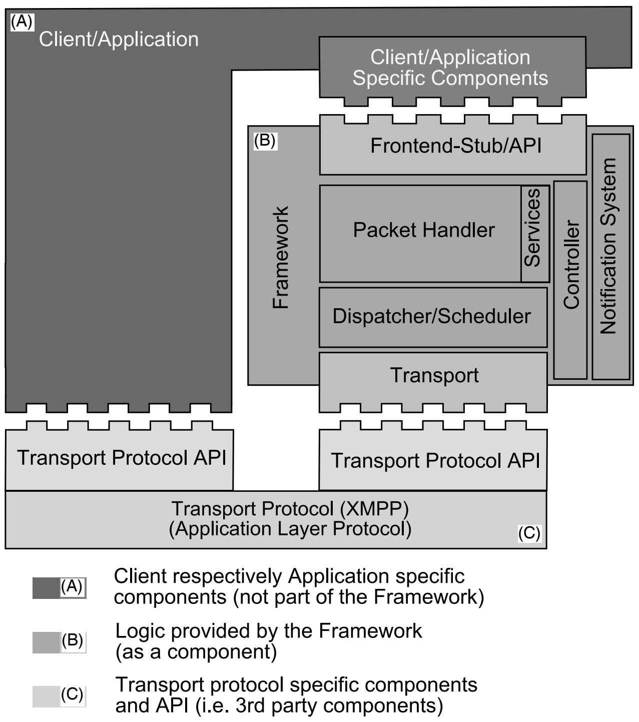

(A)
Client/Application
Client/Application Specific Components
(B)
Frontend-Stub/API
Framework
Packet Handler
Services
Controller
Notification System
Dispatcher/Scheduler
Transport
Transport Protocol API
Transport Protocol API
Transport Protocol (XMPP)
(Application Layer Protocol)
(C)
(A) Client respectively Application specific components (not part of the Framework)
(B) Logic provided by the Framework (as a component)
(C) Transport protocol specific components and API (i.e. 3rd party components)

Figure 2. Framework architecture

while data transfers or control messages can be transmitted with lower priority. To handle the messages according to their priority, the Dispatcher/Scheduler component identifies the type of each incoming message and passes it to the according Packet Handler (for UI, data, and control messages, respectively). All three packet handlers run concurrently, but with different priorities. They extract the content from each message and pass it to the Front-End interface.

While the packet processing component is the same for clients and application servers, the front-end works differently depending on whether the framework is deployed in a client or application server context. On the client side, it serves as an API for the GUI rendering engine, while on the server side, it serves as an API for the application logic. The packet handlers are managed by a central Controller and supported by Service components that handle tasks such as timer synchronization, presence data etc. The framework's components can exchange asynchronous messages internally using the Notification System.

3. Performance Measurement

Response time contributes significantly to the overall usability of any application, and can become a decisive factor in thin client applications that frequently communicate with the server in order to react to user interactions. To measure the response times that can be achieved with our IM framework, we implemented a simple prototype application that contains only trivial application logic, but employs the core communication features of our framework.

After initializing the session, the prototype application displays three buttons that change their color when the user clicks on them. Every time a user clicks on a button, the client sends an event message to the server, which sends an UI update message with the new color back to the client. While this behavior looks like a request-response cycle, it is implemented using asynchronous instant messages, so the interface is not blocked during the very brief cycle times. In an actual application, we are obviously not restricted to the request-response pattern – the application may not have to answer to every user event, or send update messages on its own initiative. In this prototype, however, the request-response pattern enables us to easily measure the round-trip communication time by taking the time between the transmission of the user event message and the receipt of the UI update message.

In order to identify critical sections in the infrastructure, we took timings at different steps in the communications pipeline (Table 1). The values shown in the table are the averaged results of ten round-trip cycles on a 2x 1 GHz double processor machine running Linux (Kernel 2.4). To facilitate easy comparison between the timings on the client and the application server side, both the application and the thin client were deployed on the same host. To obtain realistic measurements of the public IM infrastructure, we sent all messages through the public Jabber network using the public server `jabber.ccc.de`. Since our focus was on testing the latency impact of our IM framework and the public IM infrastructure, rather than a particular mobile network's characteristics (which vary widely with the type of mobile network and environmental conditions anyway), we ran our experiment on a 100 MBit/s LAN connected to the Internet. Our measurements thus indicate the lower bound to be expected for latency times on a mobile network.

As the table shows, our framework itself handles messages quite efficiently (steps 3 and 6), however, the processing of messages by the Smack API, which encapsulates the XMPP functionality, took a much longer time (steps 2 and 5) – actually, longer in total than the delay introduced by transmitting the messages over the public network (steps 1 and 4). Even then, the total response time for this configuration averages about 163 ms, which means that the delay is noticed by the user, but not experienced as a disturbing interruption [14]. With suitable optimization of the Smack API, we hope to reduce the overall response time further.

4. Related Work

Providing a complex GUI on a thin client has been a challenge for many years. Among the first proposed solutions was the X Window System [12], a network-transparent window manager following the client-server paradigm. An X server consists of a device-independent layer, which provides an interface for graphical operations to the application, and a device-specific layer, which is adapted to the characteristics of a particular operating system and graphics hardware. The X protocol defines a number of messages that can be exchanged between client and server to initiate operations such as opening a window. In comparison to our approach, the X protocol offers more graphical flexibility since it works on a lower level, however, it also places a higher implementation effort on the application developer since more features need to be implemented manually. The low-level implementation also requires more constant and high-bandwidth network connectivity than our framework does, where the user can perform basic interactions with the user interface (e.g. sizing windows, filling form fields) without having to communicate with the server.

	Step	**Time [ms]**
1	Client-server message transmission	27.7
2	Server-side Smack API processing	52.6
3	Server-side IM framework processing	2.7
4	Server-client message transmission	31.2
5	Client-side Smack API processing	29.1
6	Client-side IM framework processing	20.0
	Total response time	163.3

Table 1. Average processing and transmission times on client and application server

Originally derived from the X Window System, Virtual Network Computing (VNC) [9] is another client-server approach to presenting a GUI on remote thin clients. Since VNC works on the frame buffer level, it can be used to transmit any graphical interface to a graphics-enabled client and communicate keystrokes and mouse gestures back to the server. Even though the VNC protocol implements a number of compression algorithms, the transmission of pure pixel data is quite data-intensive, so a high-bandwidth connection is necessary for fluent work.

Due to their strong server dependence and high bandwidth requirements, the low-level solutions are unsuitable for use on mobile networks. In order to move further away from this low interface level, the client needs to take up more rendering and execution responsibilities: The Remote Java Foundation Classes (RJFC) approach [4] is an extension to the existing JFC model, where a client builds the user interface from local JFC instances, which are proxies for the actual JFC implementations on the server. All communication between the interface and application logic is then performed by Remote Method Invocation (RMI), which keeps the bandwidth requirements low. However, the UI cannot be adapted as flexibly to different client devices' capabilities as with the user interface description language employed in our approach, and firewalls typically preclude RMI communication over wide-area networks.

In the area of web-based solutions, the Ajax approach already mentioned in the introduction [2] and Macromedia's Flex framework [5] both strive to enhance the user experience beyond simple page-to-page navigation. While Ajax manipulates web pages' DOM in order to let the interface react to user interaction, Flex allows developers to specify the user interface in the XML-based description language MXML, which is compiled into Shockwave applications on the server that are transmitted to the client. This way, Flex offers developers more freedom in designing the user interface than Ajax can. However, both approaches rely on HTTP as the underlying protocol and thus are also limited to its request-response communication scheme, which does not allow the server to initiate interactions on its own.

5. Conclusion

In this paper, we presented the architecture of a framework for developing thin client applications that employ a public instant messaging infrastructure to transmit user interface descriptions and user events between the server-side application logic and a generic GUI rendering engine (comparable to a browser) on the client. The advantage of this approach is the device-independent specification of the user interface using a language like XUL, and the flexible communication model in which client and server are equal peers who can initiate transmissions individually and asynchronously. Due to its low bandwidth requirements and robustness against temporary losses of connection, this approach is especially suitable for thin client applications on mobile networks.

In our ongoing work, we are focusing on improving the performance of the framework by incorporating mechanisms for measuring the quality of the connection between the application server and different clients, and adapting the message prioritization and scheduling accordingly. We are also working on improvements to the framework API that will allow better integration with client-side GUI renderers and the server-side application logic. This way, we are striving to gain more insights into the impact of the framework on the application development process, the suitability of IM communication for complex GUI interactions, as well as the response times and communication costs incurred on mobile networks. From these factors, we can then deduce the suitability of the IM-based approach for developing mobile applications with rich user interfaces that are more sophisticated than those that can be realized in a web browser.

Acknowledgments

The authors wish to thank Gerald Mücke for implementing the IM framework prototype and carrying out the performance measurements. The Chair of Applied Telematics/e-Business is endowed by Deutsche Telekom AG.

References

[1] M. Gaedke, M. Beigl, H.-W. Gellersen, and C. Segor. Web content delivery to heterogeneous mobile platforms. In *ER Workshops*, pages 205–217, 1998.

[2] J. Garrett. Ajax: A new approach to web applications. www.adaptivepath.com/publications/essays/archives/000385.php, Feb 2005.

[3] Jive Software. Smack API. www.jivesoftware.org/smack.

[4] S. Lok, S. Feiner, W. Chiong, and Y. Hirsch. A graphical user interface toolkit approach to thin-client computing. In *Proceedings of the 11th Intl Conf on the World Wide Web (WWW 2002)*, pages 718–725. ACM Press, 2002.

[5] Macromedia Inc. Macromedia flex: The presentation tier solution for delivering enterprise rich internet applications. www.macromedia.com/software/flex/whitepapers/pdf/flex15_tech_wp.pdf, 2004.

[6] Mozilla.org. XUL. developer.mozilla.org/en/docs/XUL.

[7] A. J. H. Peddemors, M. M. Lankhorst, and J. de Heer. Presence, location, and instant messaging in a context-aware application framework. In *MDM '03: Proceedings of the 4th International Conference on Mobile Data Management*, pages 325–330. Springer-Verlag, 2003.

[8] J. Rice, A. Farquhar, P. Piernot, and T. Gruber. Using the web instead of a window system. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI '96)*, pages 103–110, 1996.

[9] T. Richardson, Q. Stafford-Fraser, K. Wood, and A. Hopper. Virtual network computing. *IEEE Internet Computing*, 2(1):33–38, 1999.

[10] P. Saint-Andre. End-to-end signing and object encryption for the extensible messaging and presence protocol (XMPP). www.ietf.org/rfc/rfc3923.txt, Oct. 2004.

[11] P. Saint-Andre. Extensible messaging and presence protocol (XMPP): Core. www.ietf.org/rfc/rfc3920.txt, Oct. 2004.

[12] X.Org Foundation. X window system. www.x.org.

[13] W. Zhao, D. Kearney, and G. Gioiosa. Architectures for web based applications. In *4th Australasian Workshop on Software and Systems Architectures (AWSA 2002)*, 2002.

[14] H.-J. Zuberbühler. Wahrnehmung von Verzögerung in netzvermittelter Kommunikation. e-collection.ethbib.ethz.ch/show?type=bericht&nr=301, November 2002.