Chapter

GUI framework communication via the WWW

Thomas Tilley, School of Information Technology, Griffith University, Australia 4215, T.Tilley@gu.edu.au

Peter Eklund, School of Information Technology, Griffith University, Australia 4215, P.Eklund@gu.edu.au

Abstract

This paper discusses experimental results and design issues arising from a project to implement a distributed, platform-independent GUI framework. Four different approaches for distributed-object communication are presented. Each approach is evaluated according to its execution time and engineering cost.

Keywords: World Wide Web, GUI, client/server, Java, CORBA

1 Introduction

Two main options exist for the implementation of an Internet front-end for applications with Graphical User Interfaces (GUIs). The first is to "hard-code" a remote interface specifically for the application and distribute it to client computers. The problem with this approach is the inflexibility of the resulting interface. If the application is modified then the GUI software needs to be changed and re-deployed to clients. World Wide Web based deployment options include: FTP or HTTP down-loading of the client software; using "push" technology to update clients; and distribution of the front-end as a Java applet.

An alternative approach that minimises the need for client re-distribution is to provide a GUI framework [1, 2]. Using the framework an application's interface can be built dynamically directly from the application server. If the appropriate set of GUI components are implemented on the client the remote GUI can provide the same “look and feel” as the program running locally on the application server. An additional advantage is that changes to the back-end (server-side) application do not necessitate modification to the front-end (client-side). Any re-arrangement of the application's user interface (UI) is reflected dynamically by the remote client.

2 Project Overview

The aim of the project was to provide a platform independent GUI capable of providing remote access to an application via the Internet while retaining the native “look and feel” of the application. This was achieved using a two-tier client/server approach that has parallels with the X-Windows System and “thin client” Network PC paradigms. The server-side target application was a Spatial Database Management System (SDBMS) providing real-time 3D data visualisation. The SDBMS was implemented in C++ on the Silicon Graphics IRIX platform.

Java was chosen to implement the front-end client because of the project’s platform-independence requirement. While Java is only as portable as the platforms that support it, this now represents a large number of operating-systems. The incorporation of the Java Virtual Machine (VM) into Web browsers also provides a convenient distribution mechanism for Java applets. It is important to note, however, that a Web browser serves only to deploy the client. As Nielsen notes it is unlikely that the Web will become a single, universal interface for all applications and services delivered by the Internet [3]. The aim of this work is not to massage an application's UI so that it can be accessed via a browser, rather it attempts to provide transparent remote access to an application independent of the client platform.

The Swing GUI toolkit from the Java Foundation Classes [4] was used to implement the GUI components, known as widgets, on the client. Swing was chosen for two reasons. Firstly, because Swing is implemented in Java it meets the platform-independence requirement. Secondly, it mirrors the functionality of the Qt GUI toolkit [5] used by the target application including support for the "Motif" look and feel.

3 Distributed-object Communication

“Distributed computing” typically refers to the transparent distribution of processes or data over an unspecified number of machines. In this paper the term “distributed” indicates the simple separation of an application and its user interface via the Internet. GUI building instructions from a server-side application are trapped and transported via the Internet to a remote client. The remote client then assembles the interface to produce a remote UI for the application. Events such as the user clicking on a button are returned to the server-side for processing.

Providing front-ends for legacy systems presents a number of engineering challenges. One of these challenges is how to best integrate components written in different languages. In this project the need for a Java client to interact with a server application written in C++ limited the available communication options. Four implementation techniques for inter-language communication were explored. The first was to communicate directly between Java and C++ using BSD style sockets.

Sockets provide a fast, low-level, file-like means of communication between processes or machines. This character based approach means that complex objects must be ``flattened" or serialised before they can be sent ``over the wire". The serialisation efficiency affects the length of the resulting object representation which is typically a string. The longer the string the longer it takes to transmit. There is also an associated engineering cost in terms of the design and implementation of a suitably complex and robust communication protocol [6]. Communication between different languages must provide compatibility and consistency between base types such as integers and strings.

One obvious way to solve the type compatibility problem is to use Java sockets for communication on both the client and the server. This was the second technique explored. While this technique increases the complexity of the server and appears to merely shift the location of the problem it actually facilitates a solution. Using the Java Native Interface (JNI) [7] Java programs can interact with C and C++ via shared libraries. JNI provides type mappings and methods for inter-operation between the languages. This resolves the compatibility problem at least for simple types. Java sockets can provide the client-server communication while JNI handles Java/C++ interaction within the server.

The third technique used CORBA - the Common Object Request Broker Architecture [8, 9]. As with the first socket option it facilitates "direct" communication between the Java client and the C++ server. CORBA is a high-level distributed object technology that largely abstracts over protocol and serialisation issues. It provides base types for inter-language operation as well as the ability to handle exceptions. The incorporation of CORBA object request brokers (ORBs) into Web browsers CORBA is an increasingly flexible deployment option.

The final implementation again required Java on both client and server. The JNI handled Java/C++ interaction while the client/server communication relied upon Java's Remote Method Invocation (RMI) [10]. RMI is a high-level, CORBA-like technology providing communication between Java VMs running on different hosts. While RMI does not provide the same range of services as CORBA it does provide simple and transparent integration into Java-code.

4 Protocol Evaluation

An experiment was devised to evaluate the suitability of the four techniques described above. The experiment is designed to simulate the transmission of "callbacks" between a remote GUI client and a server application. There are two main aims. The first is to measure the comparative performance in terms of string echo time for each of the communication techniques. The second aim is to gain insight into the engineering costs associated with implementing each method.

GUIs are assembled using sub-components called widgets. Widgets represent items such as buttons, sliders and menu items. In response to user actions widgets generate “callbacks”. For example a user clicking on a button generates a callback which then invokes the appropriate response in the application - perhaps to open a dialog box. A moderately complex GUI may incorporate hundreds of callbacks [11]. In a distributed GUI these callbacks are forwarded to the server.

Callbacks from some widgets can also contain parameters reflecting their current state. A slider's callback may for example pass an integer representing the current position of the slider. To simulate a typical single-parameter callback a randomly generated string was passed from a client to a server. This string was then echoed back to the client and the elapsed time was recorded. Simple implementations of the four techniques described in Section 3 were written and the experiment was conducted.

A program generated test files that contained 10, 20 or 40 strings. The strings were composed of random characters with lengths between 1 and 256 characters. Although the use of longer test strings would reduce the effect of marshalling on the results, shorter test strings more accurately reflected widget parameters.

Disk input/output latency was limited by pre-loading the test–data into a convenient data-structure before transmission. The JNI server implementations were also required to convert the test string from Java to C++ and then back to Java before echoing it to the client. Connection establishment times were ignored and network loading effects were reduced by running both client and server on the same host in loop-back mode. The server-side application was only designed to support a single-user at a time so testing multiple concurrent connections was not required.

Strings	C++ Sockets	Java Sockets	CORBA	RMI
10	8.1	17.3	32.5	32.8
20	12.5	35.7	55.9	61.6
40	27.3	59.9	113.4	117.4

Table 1: Average echo times (mS) for 10, 20 and 40 strings.

Table 1 presents the average echo times in milliseconds over 10 runs for files containing 10, 20 and 40 strings. The data appears to indicate an approximately linear increase in the average echo times. Some initial experiments with 10, 100 and 1000 strings also reinforce this observation.

The average echo times over 10 runs for each of the files containing 10 strings are represented in Figure 1. As expected the socket-based implementations provided the fastest times. The Java-based socket implementation is slower due to VM and JNI conversion overheads. CORBA was approximately 4 times slower than the C++ sockets while RMI provided the slowest times overall. These results may be influenced by the choice of a particular VM or CORBA implementation.

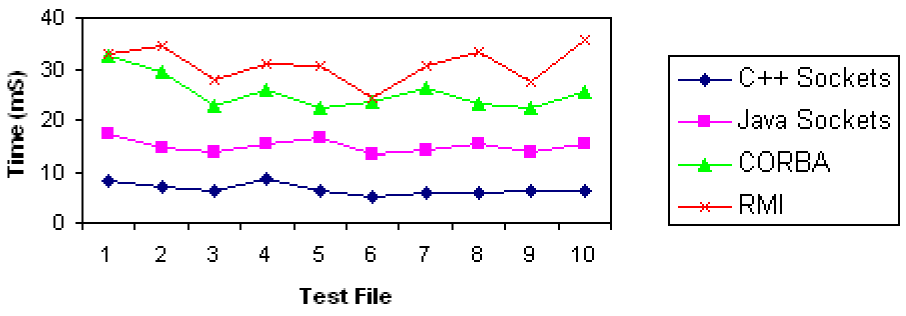

Figure 1: Average echo times of the four implementations.

While these results provide a simple performance comparison the other significant issue is the engineering cost of each technique. Although sockets provide the fastest implementations the engineering cost to develop a robust protocol for a modestly complex GUI under strict time constraints is too large. The high-level facilities provided by RMI and CORBA come at the expense of speed but they reduce some of the costs associated with implementing distributed systems [1].

Although the RMI method provided the slowest callback times it is simpler to use than CORBA and integrates transparently into Java code. RMI also automatically provides distributed garbage collection. This needs to be handled manually under CORBA and is an important consideration when dealing with C++. On the basis of its comparatively low engineering cost RMI was used to provide communication within the distributed GUI framework.

Remote visualisation typically requires significant volumes of data to be transmitted. While the GUI framework itself did not implement remote data visualisation the experimental results suggest that using RMI for visualisation communication would be inappropriate. Speed is essential and a less complex protocol is required [12]. With a simpler protocol requirement and the need for speed a sockets based implementation would be more suitable.

5 Integration

Transparently integrating a remote Java client with a C++ server presents a number of challenges. The use of RMI and JNI resolves most communication and inter-language problems but two main issues remain. Firstly, how can the distributed GUI framework be incorporated into the server without modifying the server-side application. Secondly, how can the front-end Java widgets be bound to their C++ counterparts so that assembly and callbacks operate correctly.

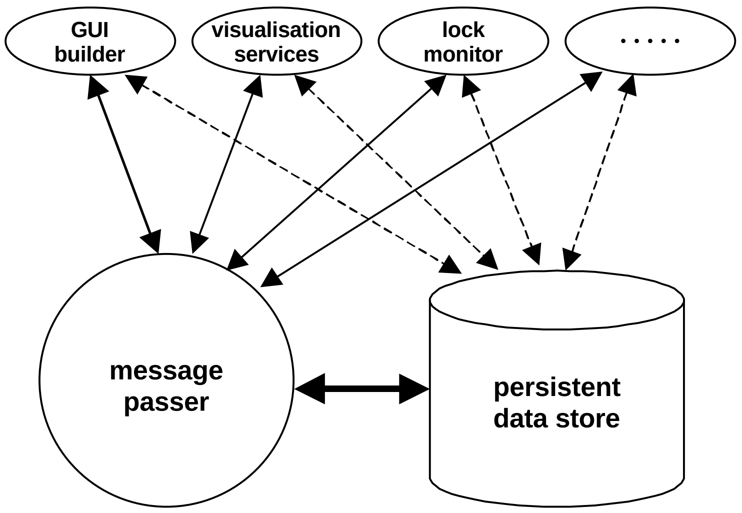

Figure 2: The server-side SDBMS architecture.

The architecture of the server-side application is presented in Figure 2. This architecture facilitates an elegant solution to the first problem. The SDBMS has a client/server architecture based around a central message passer and a persistent repository for spatial data. Components of the system are implemented as servers, each running as separate processes. These servers provided facilities such as GUI building, visualisation, triangulation, and data locking. The server labelled "**.....**" in the diagram represents the extensibility of this system. New facilities can be incorporated by implementing them as a server and then registering them with the central message passer.

The SDBMS's UI is built by forwarding component requests to the GUI builder. The GUI builder processes the requests and assembles the interface using the Qt GUI toolkit. By implementing a distributed GUI server with the same services as the original GUI builder these requests can be trapped and forwarded transparently to the distributed GUI framework. See Figure 3. Note that no modification to the existing application code is required. The distributed GUI server can be substituted for the original GUI builder at run-time.

As is the case with most distributed-object technology the distinction between client and server becomes blurred in this architecture. Both the distributed GUI framework and the distributed GUI server provide RMI services for each other: GUI building services on the “client”; and callback services on the “server”.

A solution to the widget binding problem makes use of this client/server, server/client architecture. When a request for a button is received from the message passer a "virtual" C++ button is created in the distributed GUI server. This C++ button then uses JNI to instantiate a real button on the front-end via an RMI call.

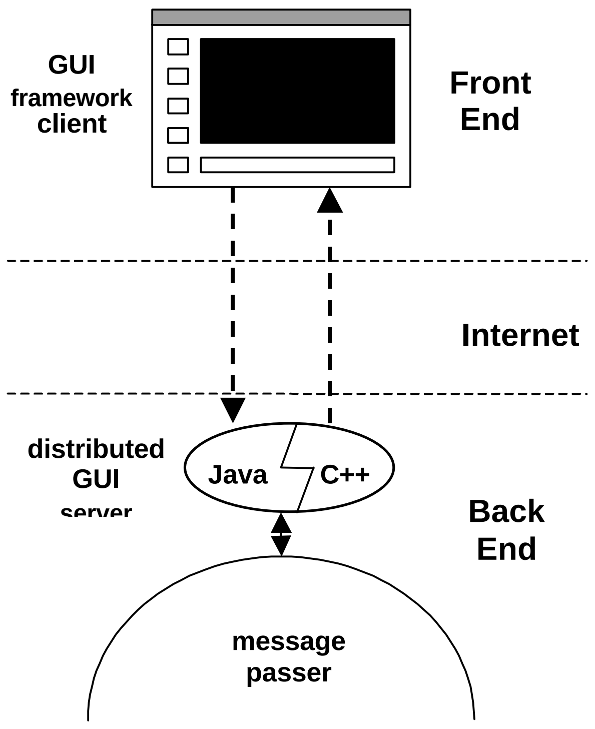

Figure 3: Communication within the distributed GUI framework architecture.

An integer representing a pointer to the C++ button is passed as a parameter to the Java button on the client. When a user clicks on the client-side Java button a server-side callback is invoked via RMI. The virtual button’s pointer value is passed back to the distributed GUI server as a callback parameter. Using JNI the integer value is turned back into a pointer to the C++ button in the distributed GUI server. The virtual button then sends a callback to the message passer which appears to have originated locally. This completes the loop which essentially implements the distributed callback technique described by Mowbray and Malveau [13]. As soon as the C++ callback has been sent to the message passer the RMI call from the GUI framework client returns.

6 Conclusion

There are a number of design decisions involved in the provision of an Internet front-end for an existing or legacy application. These include finding an appropriate deployment mechanism, minimising re-deployment, achieving platform independence, providing integration transparency and resolving language inter-operability problems. The distributed GUI framework presented in this paper attempts to address these problems.

With the inclusion of CORBA ORBs and Java VMs in Web browsers distributed object technologies are now widely accessible. The engineering advantages they provide, however, come at the cost of performance and a further divergence in browser compatibility. Sockets based implementations provide the best performance but the cost of developing complex protocols has to be addressed.

References

[1] D.C. Schmidt and M.E. Fayad, *Lessons learned building reusable OO frameworks for distributed software*, Communications of the ACM, 40(10), 85-87, 1997.

[2] D.C. Schmidt and M.E. Fayad, *Object-oriented application frameworks*, Communications of the ACM, 40(10), 32-38, 1997.

[3] J. Nielsen, *Does Internet = Web?*, Alertbox, September 20, 1998, http://www.useit.com/alertbox/980920.html

[4] Sun Microsystems, *Java Foundation Classes (JFC)*, http://www.java.sun.com/products/jfc/

[5] Troll Tech, *Qt Reference Documentation*, http://www.troll.no/qt/

[6] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall, *A note on distributed computing*, (Report No. SMLI TR-94-29), Sun Microsystems Laboratories, 1994, http://www.smli.com/techrep/1994/abstract-29.html

[7] Sun Microsystems, *Java Native interface (JNI)*, http://java.sun.com:80/products/jdk/1.1/docs/guide/jni/spec/jniTOC.doc.html

[8] S. Vinoski, *CORBA: Integrating diverse applications within distributed heterogeneous environments*, IEEE Communications Magazine, 35(2), 46-55, 1997.

[9] A. Vogell and K. Duddy, *Java programming with CORBA*, New York: John Wiley and Sons, 1997.

[10] Sun Microsystems, *Remote Method Invocation (RMI)*, http://java.sun.com:80/products/jdk/rmi/index.html

[11] B.A. Myers, *UIMSs, toolkits, interface builders*, Handbook of UI Design, 1996.

[12] B. Shneiderman, *Designing the user interface: Strategies for effective human-computer interaction (2nd ed.)*, Reading, Mass.: Addison-Wesley, 1992.

[13] T.J. Mowbray and R. Malveau, *CORBA design patterns*, New York: John Wiley and Sons, 1997.