Swing Based Remote GUI Emulation

Thomas Tilley and Peter Eklund

School of Information Technology, Griffith University, Australia 4215

{T.Tilley,P.Eklund}@gu.edu.au

This paper describes the implementation of a remote, Swing based GUI framework for a C++ application. Using a client-server approach the framework dynamically builds a platform-independent, remote GUI that retains the application's original "look and feel".

Introduction

The provision of Graphical User Interface (GUI) front-ends for legacy applications presents a number of engineering challenges. These can include language inter-operability problems and locating suitable hooks into the original application. These issues are further complicated for Internet front-ends which allow users to access an application remotely. This paper describes a project which addresses these issues for a chosen target application.

The project's aim was to provide a platform-independent GUI framework [1, 2] that dynamically builds an application's interface on a remote client via the Internet. Based on a two-tier client-server model this dynamic approach retains an applications original "look and feel" while minimising the need for future client re-deployment. Any modifications to the server-side application's user interface are dynamically reflected by the client.

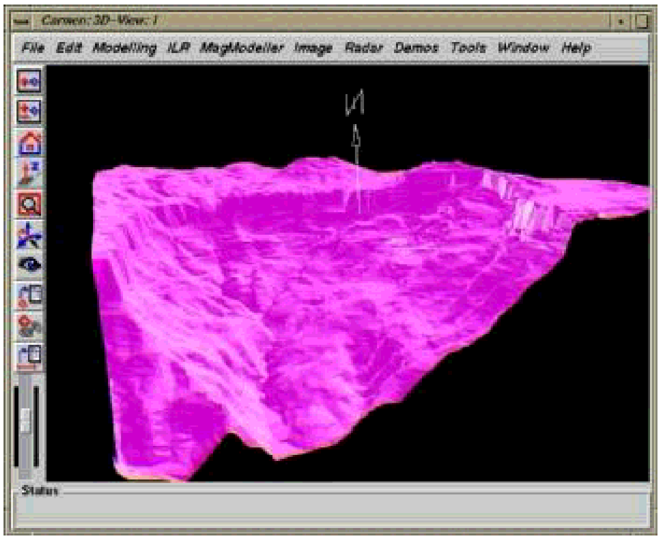

Figure 1: Visualisation of a land-basin in 3D.

A Spatial Database Management System (SDBMS) under development by Maptek [3] was the target application for the project. Providing real-time visualisation of data it is used in the defence, mining and urban planning domains. Written entirely in C++ for the SGI IRIX platform the SDBMS's interface is built using the Qt GUI toolkit [4]. Figure 1 presents a screenshot of the system visualising a land-basin in three dimensions.

System Architecture

Although Maptek's SDBMS was under active development at the time of this project's implementation it could be considered as a legacy system requiring an Internet front-end. Figure 2 depicts the architecture of the SDBMS which is based around a persistent data store and a central message passer. System components are implemented as servers which register the services they provide with the central message passer. For example, the "GUI builder" server depicted in Figure 2 receives GUI building requests from server applications via the message passer. These are then rendered locally using Qt and widget callbacks are returned to the application via the message passer. This architectural approach provides easy system extensibility. New applications can be implemented as servers and their services registered with the message passer at run-time.

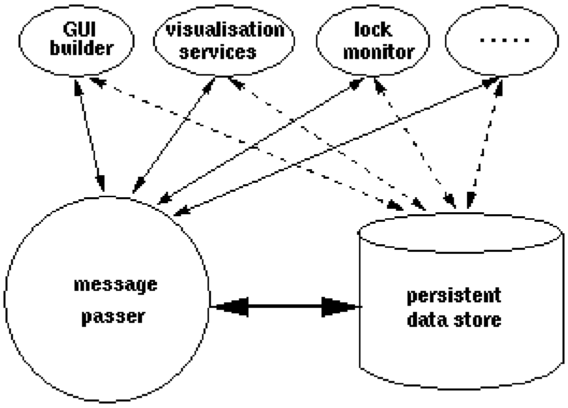

Figure 2: The server-side SDBMS architecture.

The project's platform independence requirement made Java an obvious choice for the remote framework-client implementation. Swing components from the Java Foundation Classes [5] were used to implement the client-side GUI. Swing provides a set of widgets equivalent to those available in the Qt toolkit. Using a tweaked version of the pluggable "Motif" look and Feel, it is possible to recreate the SDBMS GUI in a cross platform manner. The Java code is implemented using version 1.1.6 of the Java Development Kit for the IRIX platform and Swing version 1.0.2. The test machine was an SGI Origin 200 running under IRIX 6.4.

Client-server communication in the project was implemented using Java's Remote Method Invocation (RMI) [6]. Schmidt and Fayad [1] warn programmers not to rely too heavily on distributed object technologies, however, tight project constraints limited the time available to develop a suitable robust solution involving a complex protocol [7]. A number of distributed-object technologies were evaluated and RMI was chosen because of its comparatively low engineering cost [8]. Once an RMI reference to a remote Java Virtual Machine (VM) has been obtained the incorporation of remote calls in Java code is transparent.

RMI is a Java-to-Java distributed object technology so it was necessary to integrate Java with C++ on the server-side. This inter-operability was achieved using the Java Native Interface (JNI) [9]. JNI allows Java to communicate with C or C++ via shared libraries. The overall operation of the GUI framework system can now be described.

Remote GUI Construction

Figure 3 represents the remote-GUI creation process. First, the client creates an RMI reference to a host running a remote GUI server (1). The server then creates an RMI reference back to the client's host (2). This allows the client and the server to invoke methods on each other. As with most distributed object technologies, the distinction between "client" and "server" becomes blurred. The GUI framework client provides methods for widget and GUI construction while the remote GUI server provides connection management and callback services. Three separate socket connections are used to implement each RMI reference so this equates to six sockets per client. While this represents a potential scalability issue with multiple clients, the

SDBMS itself only supported single users at the time of the project's implementation. The remote GUI server starts the SDBMS and the RMI reference to the framework-client is then passed from Java into C++ via JNI (3). Within the remote GUI server C++ "proxy" widgets are created in response to GUI requests received via the message passer. These proxy-widgets embody the state of individual widgets on the server-side. The RMI reference to the client is passed into this proxy-widget hierarchy via their class constructors (4). Each server-side widget can now invoke its client-side Swing counterpart "directly" via RMI to render the remote interface. Although the invocations actually come via the server-side VM they conceptually come directly from C++ (5).

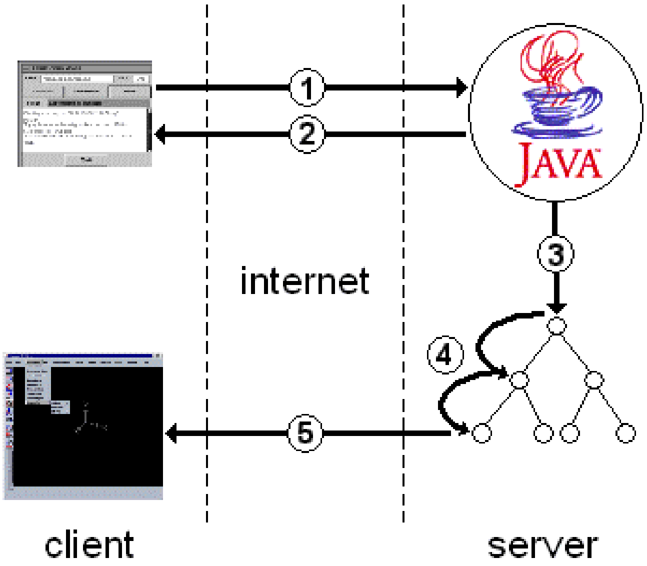

Figure 3: The remote-GUI creation process.

Passing RMI references into C++ is potentially dangerous because the references may become invalid if the VM unloads the corresponding Java class. To avoid this problem the method which starts the SDBMS takes the RMI client reference as a parameter. This method is called from a new thread which does not return until the user exits the SDBMS. This ensures that the class will not be unloaded until the user has finished with the application.

Widget Binding

Maintaining referential integrity between an individual proxy-widget and its Swing counterpart was accomplished using a hash-table. Framework-client method invocations required a proxy-widget's constructor value to be passed as an integer parameter. The constructor value is then used as a key into a hash-table containing the actual Swing widget. This one-way binding technique allows references to be maintained after the method returns and it also facilitates the implementation of callbacks.

Callbacks were implemented using a similar constructor value technique which is represented in Figure 4. For example, when a user clicks on a Swing menu-item (1) it invokes a method in the remote VM (2). These callback methods take the value of the corresponding proxy-widget as an integer parameter. This parameter value is passed via JNI into C++ where it is cast back into a valid proxy-widget reference (3). A callback is invoked on the appropriate proxy-menu-item and at this point the RMI method returns. This callback handling technique implements the distributed callbacks described by Mowbray and Malveau [10]. The proxy-menu-item then forwards a callback to the message passer which appears to have originated locally (4). Additional parameters that convey the state of a particular type of widget can also be included in the RMI callbacks. Sliders and scrollbars, for example, typically require an integer parameter to convey their current position.

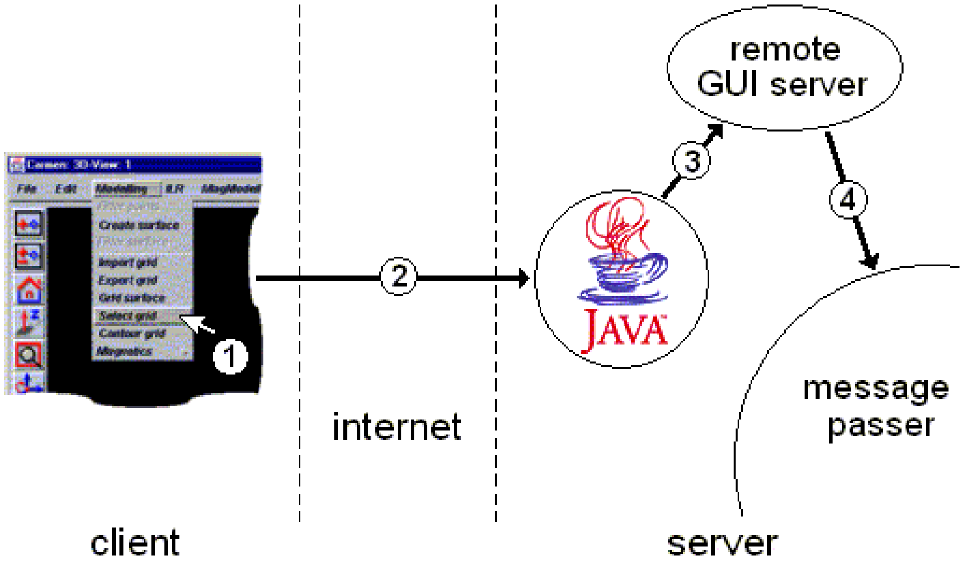

Figure 4: Handling distributed callbacks.

Passing C++ pointer values around presents an obvious security problem. Security issues were not considered during this initial implementation but could be addressed by using RMI over secure sockets.

Using the widget invocation and callback techniques described here are relatively expensive. Their performance depends upon a number of factors including the efficiency of object serialisation and the communication latency. One approach to reduce latency is to implement "setters" and "getters" assymetrically. "Setters" are methods which change or set widget state while "getters" return the current state. Invocations of "setters" are forwarded to the remote framework-client so that it reflects the desired state but "getters" need only return the state from the local proxy-widgets. This eliminates both the cost of an RMI call and any additional overhead imposed by JNI.

System Integration

One of the main implementation difficulties with the project was finding the appropriate level of abstraction for the proxy-widgets. The SDBMS's original GUI builder wraps the Qt toolkit and presents programmers with an Application Program Interface (API) for a set of idealised "virtual" widgets. This abstracts over the actual widgets used and facilitates the substitution of different GUI toolkits. Within the remote GUI server too much abstraction made it difficult to access the required widget functionality. Alternatively if the level of abstraction was too low then the system was tied too tightly to Qt.

The problem of finding appropriate hooks into the original SDBMS application is elegantly solved by exploiting the architecture represented by Figure 2. If the remote GUI server implements the same services as the original GUI builder then it can be substituted for the "real" GUI builder at run-time. All requests for GUI services can be trapped and forwarded to the framework-client for rendering. Similarly all callbacks are returned via the substitute server and appear to have originated locally. This allows the remote GUI framework to be transparently integrated into the SDBMS without modifying the original application code.

This transparency, however, relies heavily on the architecture of the target application. Adapting the GUI framework for alternative applications may require modification to the application?s code or to the remote GUI server. Suitable hooks would need to be located for trapping GUI building requests and also for returning callbacks. While no client-side modification would be required for Qt based applications or those built using similar Motif-like toolkits, the system would be unable to render GUIs that used custom widgets or a different look and feel. These requirements mean that the remote GUI framework is not a generic solution for all legacy applications.

A generic alternative that does not require application modification is the screen redraw or "screen scraping" approach employed by thin clients such as the Virtual Network Computer (VNC) [11] or systems based on Citrix?s Independent Computing Architecture (ICA) [12]. Rather than sending widget-requests to a client, thin clients typically transmit screen updates, which are then rendered remotely. Instead of forwarding callbacks to the server, all keystrokes and mouse activity are returned back to the original application. The clients are stateless and simply redraw the GUI as it would appear locally. Java clients can be deployed as applets within browsers or as applications and servers are available for a range of platforms.

The disadvantage of the thin client approach is that it consumes more bandwidth and has a higher latency than the remote GUI framework approach. Screen updates are more frequent than widget requests and typically contain more data. Callbacks within the remote GUI framework are also only invoked in response to significant user action. In thin client systems every keystroke or mouse movement results in traffic to the server and a screen update in response.

Conclusion

This paper has described the implementation of a platform-independent remote GUI framework for a SDBMS written in C++ and a number of integration issues were discussed. A screenshot of the remote GUI framework running on the Windows NT platform is presented in Figure 5.

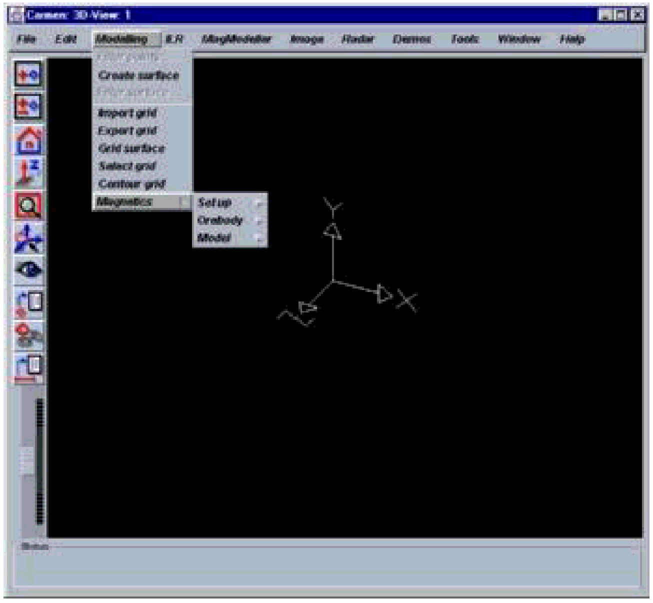

Figure 5: GUI-framework screenshot on the Windows NT platform.

The Remote Abstract Window Toolkit (AWT) from IBM’s alphaWorks [13] provides similar functionality to the remote GUI framework described in this paper. Based on a client-server model it allows a Java application to display its GUI remotely. Although it can be used without modifying the target code it is restricted to Java applications only. A further restriction is the fact that it can only render AWT components. Faithful emulation of the chosen SDBMS? GUI required Swing widgets.

Although the framework presented here could be re-used for other target applications with a Motif look and feel, the remote GUI server would require modification and appropriate hooks into the application would also need to be discovered. The server-side integration transparency achieved in this implementation is dependent upon the architecture of the chosen target application and a "screen scraping" approach may be more appropriate for other legacy systems.

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] Maptek Pty Ltd, Maptek Online vulcan, http://maptek.com.au/vulcan/vulcan.html

[4] Troll Tech, Qt Reference Documentation, http://www.troll.no/qt/
[5] Sun Microsystems, Java Foundation Classes (JFC), http://www.java.sun.com/products/jfc/
[6] Sun Microsystems, Remote Method Invocation (RMI), http://www.java.sun.com/products/jdk/rmi/
[7] 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.sun.com/research/techrep/1994/abstract-29.html
[8] T.Tilley, S. Pollitt, and P. Eklund, Evaluating Distributed Graphical User Interface Communication, Proceedings of AusWeb99, the Fifth Australian World Wide Web Conference, Southern Cross University Press, Lismore, 1999.
[9] Sun Microsystems, Java Native interface (JNI), http://www.java.sun.com/products/jdk/1.1/docs/guide/jni/spec/jniTOC.doc.html
[10] T.J. Mowbray and R. Malveau, CORBA design patterns, New York: John Wiley and Sons, 1997.
[11] AT&T Laboratories, VNC - Virtual Network Computing from AT&T Laboratories Cambridge, http://www.uk.research.att.com/vnc/
[12] Citrix, Citrix ICA Technology, http://www.citrix.com/products/ica.asp
[13] IBM alphaWorks, Remote AWT for Java, March 20th 1998, http://www.alphaworks.ibm.com/tech/remoteAWT/