Thin Client for Web Using Swing

Raffaello Giulietti and Sandro Pedrazzini

SUPSI, Dipartimento di Informatica ed Elettrotecnica,
Galleria 2, CH-6928 Manno, Switzerland

TINET SA
CH-6928 Manno, Switzerland

{lello,sandro}@idsia.ch

Abstract. In this article we will present how we use and exploit the Java Swing elements' functionality to realize a Web client/server architecture environment where the client represents a so called thin client, i.e. a reduced presentation layer which does not contain any application specific software (called business logic layer or simply domain layer) and the server represents the middle tier between (thin) client and data.

1 Introduction

1.1 Overview

After the years of terminal/mainframe solutions, the client/server architecture in general has become the architecture of choice for many applications because it has represented the opportunity to split the processing load and, through the clear hardware and software distinction between client and server, was able to raise the end-user expectations about interface.

The trend during the last years is towards different client/server forms: 3-tier and, more generally, N-tier architectures. The multi-tier model foresees a split between an interface layer (presentation layer) and a business logic layer (domain layer), creating the need for a middle-tier, responsible for the interaction between application and database, as shown in Fig.1.

The need of a 3-tier model has been accentuated by the strong use of the Web, which has led the software from departmental solutions to world-wide ones. In fact current implementations of Web based business applications are centered around the 3-tier model (even called 2.5-tier model, as documented in [1]). The application is organized in 3 layers. At the top is the presentation layer which runs on the Web client and is often implemented as an HTML forms based GUI with provisions like cookies

for emulating sessions. On the servers side, the middle and bottom layers implement the domain specific and the data persistency logic (data layer) respectively.

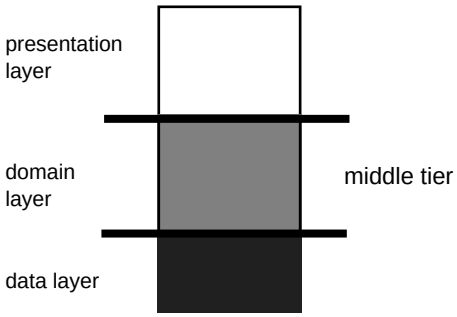

Fig. 1. 3-tier model

There is a natural tendency to come away from these often primitive HTML GUIs in favour to richer Java based presentation layers. However, Java applets, like any modern GUIs, can become quite complex. Complexity often resides in the strong interactions between widgets and between these and the domain layer. The usual approach of building the presentation layer completely in Java lessens the chance of applet reuse because of the strong coupling between the widgets and the presentation logic that glues them together.

1.2 Motivation

There are two main motivations for our work.
First, in general, adding the 3-tier further split between presentation and business logic also means a further step of complexity during the development of applications. So one goal is to propose a method that allows the development of Web applications in a transparent way, making use of Java Swing classes for the client side, without having to add further client specific local installations.

On the other hand, it can be fruitful to further split the presentation layer in more sublayers, shifting up the physical distinction between client and server, reducing the client responsibility on the application logic.

2 Architecture

At the top of the presentation layer there is the display sublayer. This is the less abstract component: it interacts directly with the user's IO hardware (frame buffers, pointing devices, keyboards and other IO devices) and presents itself as a more or less low level API which implements drawing primitives and fires low level IO events like mouse movements, key and button presses and so on.

Next comes the widgets sublayer which abstracts the display layer by offering itself as a collection of objects like buttons, text fields, menus together with their associated menu items, and the like. All widgets are designed to have a pleasant graphical look and most of them react on hardware events by forwarding more abstract events to the next sublayer: the presentation logic sublayer. Of course, such more abstract events usually carry more useful information: instead of forwarding a "left button pressed at (x, y) screen location" event to the presentation logic layer, the widgets layer translates it to a more abstract "primary button pressed on menu item 'Cut'" or even more abstractly as "invoke 'cut-selection' command".

The presentation logic sublayer is what glues the widgets together making them interact both with their peers as well as with the remote domain logic layer. It is responsible, for instance, for such behaviour like disabling the "page range" text fields in a print panel depending on the state of the "print all pages" radio button. Or, in a more complex multi-user application, it might have to update the items in a combo box whenever the state of some remote data, under control of the domain logic layer, has changed and the change needs to be notified to all current users of the application.

Both the display and the widgets sublayers provide highly reusable components. The presentation logic sublayer, on the other hand, rarely offers reusable components because most of its architecture defies a straightforward reification in terms of reusable objects.

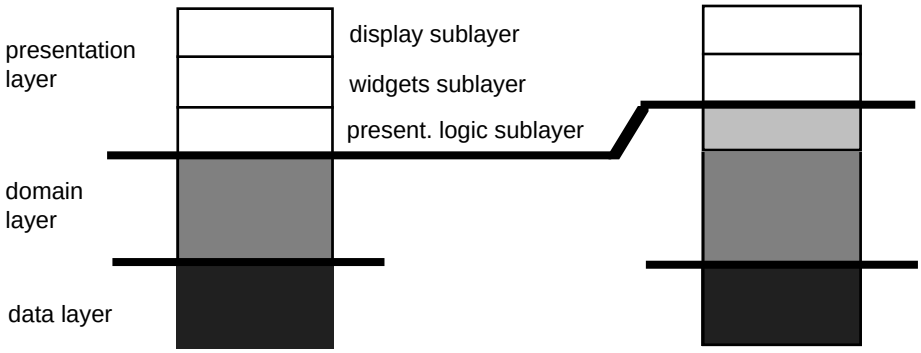

Fig. 2. Client/server distinction shift up between presentation and domain layer

Because of this asymmetry, it seems more natural to place the client/server interface between the widgets and the presentation logic sublayers rather than to put it between the presentation logic sublayer and the domain layers as implicitly suggested by the 3-tier model, as shown in Fig. 2. This approach becomes even more interesting with the introduction of the newer Swing based Java 2 platform. Once the new JRE has been downloaded and installed, a client machine provides a sophisticated platform of highly reusable widgets and supporting components. It is expected that newer versions of common web browser will include the new JRE so that in the near future every new networked client will be equipped with Swing and its relatives.

3 Realization

The basic ideas for the realization are: transparency of the distributions aspects during the development and exploitment of locally resident Java elements (Swing classes) during run-time.

The second point will reduce complexity, avoiding heavy specific installations and maintenance, reducing downloads of application specific classes. Because we want to make use of Swing classes, which are not built with a clear MVC distinction, the network interaction mechanism must be added at the Java event-handler listener level (delegation model). The interaction is built at socket level, in order to minimize the network traffic during run-time.

There is a white paper at IBM on a project called ULC which seems to split all interface instances between client and server, maintaining a client half and a server half. This approach assumes the rewrite, or at least the adaptation, of all native Swing interface objects.

3.1 Generic Applet

One interesting consequence of the proposed approach is that it becomes possible to write one generic applet that works for any compliant Web based business application. The applet simply implements the communication protocol between the widgets layer and the remote presentation logic layer. Initially, it notifies the presentation logic of its readiness. This, in turn, begins to set-up the GUI by means of a simple remote method invocation mechanism. Once this phase completes, the applet begins its role as an event dispatcher. It forwards widgets events to the presentation layer and synchronously waits for return values. Conversely it can accept asynchronous notification messages from the remote presentation logic and invoke widget updating methods as a consequence. The applet makes heavy use of the reflection capabilities built in the JDK to implement the remote method invocation mechanism. It bases its simple remote method invocation mechanism directly on sockets.

3.2 User Interface Description

An important aspect is the user-interface description made in the program.

Because we want to create the thin client layer, we cannot embed the use of Swing classes within the program itself. We must go through a protocol, which will represent and describe each Swing element, and eventually more, giving the opportunity to add new classes whenever needed.

This is the key point which will allow the use of Swing classes at client level, creating transparency with respect to the interaction and communication aspects. The use of the protocol is also the point which will allow the use of other programming languages, different than Java, for the realization of the domain layer. This will eventually allow the easy reuse of existing applications. Other approaches, like the current

Classic Blend [5], seem to strongly couple the language used in the client to a specific language in the server, or even to a specific implementation of it.

By placing the client/server interface between the widgets and the presentation logic layers, the proposed approach clearly reduces the network load with respect to X Window (as we will see in the evaluation) because only more abstract units of information interchange travel on the wire. Behaviour that is completely local to the widget itself, needs not to be forwarded to the presentation logic: compare the network load needed for a "tooltip" behaviour in Swing with an analogous implementation in X, for instance. Moreover, rendering of the widgets is completely local.

Does it generate more traffic with respect to the conventional 3-tier approach with its client/server interface between the presentation logic and the domain logic layers? Not necessarily. There are inter-widgets interactions that only need state information stored in the widgets alone which is readily available to the presentation logic layer: these would not require network traffic in a conventional 3-tier solution. However, smarter inter-widgets interactions need state information from the domain logic layer. Data validation, for instance, often requires deeper knowledge on the current state of the application which cannot be delegated to a presentation logic sublayer alone: by definition, the presentation logic sublayer needs only knowledge on the current state of the presentation layer, after all, not on the whole state of the application. In the end, it might even happen that keeping the presentation logic sublayer on the server side reduces the network traffic because there might be much more interaction with the domain logic layer than with the widgets sublayer.

3.3 Extending Swing

Provisions are included to extend the Swing set of widgets with new ones. This mechanism is not strictly necessary but can improve the network efficiency considerably. Consider, for instance, a specialized text field for input of dates which checks if the field string is a syntactically correct date in some locale. The check could be done by the presentation logic sublayer, but can typically be delegated completely to the widget itself, a more efficient and even higher reusable solution. As another example, an often found widget group consists of a text field and two small buttons on its side whose purpose is to increment or decrement the numeric value in the field. Clicking on the buttons would generate event messages directed to the presentation logic sublayer and, conversely, field update messages directed to the widgets layer. The widgets group can be implemented as a new standalone and highly reusable widget that need not forward its "internal" events to the presentation logic sublayer. A gain both in terms of network load and abstraction shift. The only difference with a built-in Swing widget is that the new one needs to be downloaded from the server.

3.4 Set-up

A less fundamental but sometimes practical provision is a mechanism to load a Java class for the set-up instead of relying on the presentation logic sublayer alone to con-

trol the build of the GUI layout. The set-up code has often a very streamlined structure: it might be conceivable to automatically synthesize a new Java class that implements this set-up, perhaps even on-the-fly, and to send it over the network. The other choice is to send a bunch of synchronous two-ways remote method invocation messages to build the layout directly from the server side. Evaluating which of the alternatives is better depends a lot on the complexity of the set-up code and its dependencies on the current state of the application, and needs to be done on a per case basis.

3.5 MVC Paradigm

In terms of the model-view-controller paradigm, the application as a whole distributes the view (widgets sublayer) remotely on the client side while retaining both the model (domain layer) and the controller (presentation logic sublayer) on the server side. It maintains a distributed view-controller pair for every client connected to the model. It implements the view as a generic empty Java applet which is dynamically populated with native Swing widgets already present locally and other reusable widgets downloaded from the network. This set-up phase is either driven remotely by the controller or locally by a class downloaded from the server and to which the controller delegates this limited role. The controller, being next to the model, can do a lot of smart processing in order to implement a sophisticated user interface without necessarily needing a high bandwidth with the view. Neither the controller nor the model need to be coded in Java. The controller, however, needs knowledge about the general Swing architecture and its widgets collection.

4 Comparison and Conclusions

The main goal of this work was to set-up a homogeneous environment to develop Web solutions. The developer only writes an application in a traditional way, using a single programming language, without having to deal with communication and synchronization problems, and ends up with a run-time solution with separated thin client part and business logic one.

HTML itself also represents a thin client solution. Its weak points are the lack of interaction, which must be filled with the use of other languages and protocols, such as Java (Applet and Servlet), JavaScript, Perl, CGI, etc. and the complete absence of transparency during development, where you are forced to use many languages and foresee each kind of network interaction and user-interface update.

A further known solution is represented by the X Window technology ([2]). This also consists of two distinct parts, the application and the GUI, and allows a transparent development. The X Window technology probably still represents the best example to consider and in many cases the tool of choice for intranet solutions. The X Window server (in the X Window terminology the user interacts with an X server while the remote application runs on the X client), represents the thinnest client at all, apart from the older text-only mainframe terminals.

There are even dedicated hardware implementations called X terminals. All they do is to communicate with the underlying IO hardware and to implement the low level X protocol. All drawing originates from the remote X client where the X application runs, and all inputs, whether discrete key presses or continuous mouse movements, go to the remote X client. This places a high load on the network but makes the system very flexible. Because the widgets are implemented remotely on the X clients and IO events are passed in raw form, the X application has full control on every detail. Whether this is desirable depends on the application, but this flexibility allows multiple toolkits and window managers to run on every X server. The X Window approach places the client/server interface between the display and the widgets sublayers. This generates a lot of network traffic making it reasonably deployable only on high bandwidth networks like LANs.

On the other side, Java applets often include a direct implementation of all of the presentation logic. They rely on Swing for the widgets and other supporting objects, and glue them together by means of ad-hoc Java objects. The applet, in turn, communicates with the remote domain logic layer by means of some standard protocol like CORBA ([4],[3]) or Java RMI, or by implementing some ad-hoc socket based communication mechanism or other. It should be clear that there is a strong dependency between the client applet and the domain logic layer: changes need to be maintained on both the client as well as on the server. This contrasts with the X Window approach where the decoupling between the X server and the X client is total.

The proposed approach stays inbetween. It commits to the JDK, in particular to Swing, on the client side because of the high reusability of its widgets and its wide deployment on a wide variety of devices, sending on the network only small higher level messages, required where application specific information are needed. It does not commit to the JDK, or even to Java, however, for the presentation logic, not to speak of the domain logic which is often a legacy system. The presentation logic is not deployed in the applet but is located on the server side.

References

1. Edwards Jeri, DeVoe Deborah: 3-Tier Client/Server At Work, John Wiley & Sons (1997)
2. O'Reilly Tim, Quercia Valerie, Lamb Linda: X Window System User's Guide, O'Reilly & Associates, Inc (1988)
3. Orfali Robert, Harkey Dan, Edwards Jeri : The Essential Distributed Objects Survival Guide, John Wiley & Sons (1996)
4. Siegel Jon: CORBA: Fundamentals and Programming, John Wiley & Sons (1996)
5. http://www.appliedreasoning.com/Products/Classic_Blend/ Classic_Blend_White_Paper/classic_blend_white_paper.html