Developing Java-AWT Thin-Client Applications for Limited Devices

Gerardo Canfora, Giuseppe Di Santo, Eugenio Zimeo
g{gerardo.canfora, giuseppe.disanto, zimeo}@unisannio.it
RCOST – Research Centre On Software Technology – University of Sannio, Benevento, Italy

Abstract

The rapid spread of personal wireless devices (PWDs) has raised the need for specific software development methods and tools due to the restrictions placed by the mobile nature of these devices. The paper presents Thin-Client aPPlications for limiTed dEvices (TCPTE), a framework that supports the development of thin-client applications for mobile devices. By using TCPTE, Java applications can be executed on a server and their AWT interfaces displayed on a remote client. The approach combines the advantages of thin-client computing with the richness of client-server graphical interfaces and allows programmers to easily develop applications for PWDs by adopting the same processes and tools typically employed for desktop applications. The paper describes TCPTE features, the architecture, and some performance evaluations under different working conditions.

Keywords: Java, Software architectures, Mobile computing.

1. Introduction

The appearance and rapid spread of personal wireless devices (PWDs), such as personal digital assistants (PDA), smart phones and 3G cellular phones, have raised the need for specific software development methods and tools. Typically, applications for these devices are either web or Java based [1], even if other platforms, such as Microsoft .NET Compact Framework (http://msdn.microsoft.com/mobility/prodtechinfo/devtools/netcf/), Symbian SDK (http://www.symbian.com/developer/sdkas.asp), or Palm OS Developer Suite (http://www.palmos.com/dev/dl/dl_tools/dl_pods) are becoming widespread. In any case, key problems of software development for PWDs are related to the heterogeneity of terminals, to the limitations of processing power, storage capacity and display size and resolution, and to low bandwidth and high latency of wireless networks.

- Issue 1: Significant restrictions on software development for mobile devices derive from their heterogeneity, which makes the development of portable applications very complex. In many cases, applications are developed in multiple versions that use proprietary libraries and ad-hoc configuration code. As a consequence, content and service providers have to supply services and applications in different forms, depending on the end-user’s hardware, with a consequent growth of development and maintenance costs and complexity.

- Issue 2: Limited processing power and storage capacity affect the possibility of performing complex computations and working with large datasets. More importantly, they prevent mobile devices from accessing to a wide range of enterprise technologies and resources.
• **Issue 3**: The available **bandwidth** plays an important role for network oriented applications, nevertheless bandwidth limitations are often negligible in comparison with the high **latencies** that characterize wireless networks (GPRS, Bluetooth, WiFi).

• **Issue 4**: A considerable limitation of mobile devices is the **reduced size and resolution** of graphic displays. Some devices, such as the pocket-PCs, have a resolution of 240 x 320 pixels, many palmtops have a display resolution of 160 x 160 pixels, whereas mobile phones present screen sizes and resolutions even smaller.

The heterogeneity (**issue 1**) could be tackled with tools for designing new graphical user interfaces (GUIs) through a platform independent model. Typically these tools [2] generate an XML description of the UI that can be successively transformed into different platform specific GUIs by means of XSLT transformations, customized on the basis of device profiles. However, for existing GUI-based applications (e.g. Java AWT applications) the problem is the reverse engineering from the original platform dependent GUI to its abstract model.

A promising approach to execute resource demanding (**issue 2**), desktop applications on devices with limited resources is the application decomposition in two groups of components to transparently distribute between a PWD and one or more servers accessed through a network. Although many existing programming models and middleware platforms support the development of distributed applications in client/server environments, they often require non trivial re-factoring of desktop applications, which becomes more complex in environments characterized by limited resources.

To address the problem, software architectures based on the thin-client model represent a viable solution since they are able to transform a monolithic application into a client/server one, with a limited burden for the programmer. Such architectures belong to two main classes: “distributed presentation” and “remote presentation”. With the former class only a thin part of the presentation layer runs on the client, whereas the remaining part as well as the application and data layers run on the server. With the latter class, the client handles more of the presentation logic so determining less round trips with the server.

Many protocols for implementing distributed presentation have been released in recent years. Among these, *X Window System* [3], *Microsoft Remote Desktop Protocol* (RDP) [4], *Citrix Independent Computing Architecture* (ICA) [5] and *Virtual Network Computing* (VNC) [6] represent the most known examples. All these protocols are characterized by a significant network traffic due to the necessity of communicating keystrokes, mouse movements or bitmap alterations.

On the other hand, even if the remote presentation well suited both to wireless communication, due to the limited number of interactions between client and server, and to reduced screen size of PWDs, due to the capability of mapping logical UI components into different native ones, the approach has only emerged recently mainly because of the spread of Web browsers. Its lower popularity is related to the necessity of developing client applications specifically designed for this architecture. However, academic and industrial researchers are defining frameworks for easily porting existing Java AWT in thin-client environments based on remote presentation. *Ultra Light Client* (ULC) [7] is a presentation engine written in Java, which transparently handles the distribution of an application through the use of half-objects: hardware-dependent objects on the client side and faceless proxies based on an API similar to the one provided by Sun on the server side, which communicate through the ULC infrastructure. Classic *Blend for Java* ([http://www.appliedreasoning.com/](http://www.appliedreasoning.com/)) allows both HTML and Java-based widgets to be used on the client side of a Web application. To achieve this objective, each widget is transparently distributed to the client by using the HTTP protocol and communicates with the server through a proxy. Despite of the transparency for object distribution, a property file on
the server is required in order to connect widgets to the application logic on the server. Thin-Client Framework (TCF) (http://www.alphaworks.ibm.com/tech/tcf) is both a framework and an approach based on the Model View Controller architectural pattern, for designing client/server applications, which can be also thin-client applications. Differently from ULC and Classic Blend, TCF is a set of interfaces and abstract classes that have to be implemented for developing a client/server specific application. Remote Abstract Window Toolkit (RAWT) (http://www.alphaworks.ibm.com/tech/remoteawtforjava/) is an implementation of AWT and Swing that allows the remote presentation of all Java applications using 1.1.6, 1.1.7, 1.2.2 and 1.3.1 versions of Sun JRE. Like ULC, RAWT uses the half-object design pattern. Moreover, differently from all the cited frameworks, RAWT does not require modifications to existing widget-based Java applications for using them as thin-client applications. On the other hand, it requires to adopt the same JRE version on both server and client side.

All the discussed frameworks were developed for desktop computers and so do not easily support the development of thin-client applications on limited devices due to the use of unsupported communication mechanisms (issue 3) and graphical components on PWDs. The paper proposes Thin-Client aPplications for limiTed dEvices (TCPTE) (available at http://tcpte.rcost.unisannio.it/), a framework that supports the development of thin-client Java-AWT applications for PWDs. TCPTE was designed with the aim of supporting easy migration of Java desktop applications towards devices with limited resources through (1) the adoption of the thin-client model in order to reduce computation on client-side (issue 2); (2) an implementation of the thin-client model based on the remote presentation architecture optimized for wireless communication (issue 3); (3) the adaptation of the user interface to heterogeneous devices thanks to the decoupling of the server-side and client-side portions of the graphical framework through a specific design pattern and a communication protocol (issue 1). The current release of TCPTE (1.0) does not tackle the problem of reduced screen size and resolution (issue 4). Moreover, it refers to AWT instead of Swing, because AWT constitutes a common support on which Swing is based. It is simple to extend the approach to Swing components.

2. Application development, deployment and execution

Typical applications for the TCPTE approach are management tools for databases and financial investments, remote control of home devices, remote work for business agents, video games characterized by limited interactivity and intensive computations (e.g. chess). All these applications may be logically divided in two macro-components, namely user interface and application logic. To transparently support such a division in a distributed client/server environment, TCPTE offers a limited version of the java.awt package (in the following called AWTServer) that contains most of the graphical components of the homonymous Sun’s package preserving the same naming and interfaces but with a limited set of methods for some classes. The implementation of this package interposes a middleware layer between the abstract components of a UI and its graphical representation, thus moving this on the client device. The division is performed at run time and is transparent to developers and end-users.

The whole process of development, deployment and execution is summarized in figure 1. In the first phase (fig. 1-a and 1-b) existing Java AWT applications may be reused or new ones developed and compiled independently of the device on which they will be displayed. For complex GUIs, a testing with AWTServer is necessary to verify the need for a re-factoring through the adaptation of the application to the new environment. In the deployment phase, the application and its XML description are to be moved on a TCPTE server (fig. 1-c). Here, the application is dynamically linked to AWTServer. Finally, in the execution phase, using a client written ad hoc for the specific device, the user connects to the server (fig. 1-d) that in turn sends to the client the list of the deployed applications (fig. 1-e). The user selects an
application and asks the server for the execution (fig. 1-f). The server creates a new process for the execution of the requested application and its UI is incrementally shown onto the client (fig. 1-g).

3. Architecture and design

The TCPTE architecture is shown in figure 2-a. The top layer on the server side represents the application that has to be run on the server and displayed onto a remote device. The application layer uses the components defined by the AWTServer layer. Starting from a method call to an AWTServer component, the object communication layer allows for invoking the method on the corresponding AWTClient component on the device. The current implementation of this components uses personal Java and J2ME. However, .NET for Windows CE platform or C++ for Palm-OS platform can be adopted thanks to a platform independent object communication protocol.

---

1 We tested the framework with J2ME with MIDP 1.0 and CLDC 1.0. However, the framework should work even with MIDP 2.0 and CLDC 1.1
3.1 TCPTE AWT layer

In Java AWT, an application that has to change the UI state invokes one or more methods on the graphical components, which trigger the invocation of the appropriate methods on the related peers that show the data onto the screen by invoking specific native methods. On the contrary, an input event sent to a UI graphical component is propagated through the correspondent logical component to the application logic by means of listeners.

In the thin-client architecture, the separation between application and presentation logics raises the necessity of two address spaces in which the two logics can execute independently. By adopting the remote presentation approach, an important task is to individuate the best way to distribute application components between the two address spaces, so as to allow the user to interact with the application through a separated device and to leave the application unchanged.

In TCPTE, the internal implementation of Java AWT was modified, and the Half-Object design pattern [8] was used to distribute the graphical components between client and server. By using the Half-Object pattern, each component needs to be split in two objects: one half in the application address space (AAS), the other half in the user interface address space (UIAS). The two half-objects communicate with the protocol offered by the TCPTE Object Communication layer.

Figure 2-b shows the deployment of a subset of the TCPTE class hierarchy. In the AAS the classes related to the application logic are deployed, whereas in the UIAS we distinguish: (1) the half-object interfaces (HOIs), which are the interfaces for the half-objects, (2) the platform-specific half-object implementations (PSHI)s that are the platform dependent implementations of the HOIs, and (3) the platform-dependent GUI components (PDC) that are the GUI objects available on the UIAS. Each logical component has a remote reference to an HOI implementation; when the application invokes a method on a logical component the corresponding method is invoked on the PSHI that implements the HOI. In the UIAS, each method invocation on the PSHI is mapped onto a correspondent method invocation on the PDC. Platform-specific PSHIs are created by using the abstract factory pattern implemented by a specific toolkit.

3.2 Object Communication and Session Layer

The communication layer deals with ad hoc serialization/deserialization techniques of objects transferred over the network, which is necessary to avoid Java RTTI (Run-Time Type Information) used by the default implementation of the serialization in J2SE and J2EE platforms and often not implemented in J2ME platforms. This layer makes TCPTE independent of the platform as far as the communication mechanisms are concerned.

In order to improve the performances of the communication layer, the objects transferred between client and server are serialized in a binary format whereas communication between objects should be based on asynchronous interactions (without blocking the caller) built on a lightweight transport protocol in order to tolerate the high latencies of wireless networks.

Even if asynchrony would require a specific middleware for the interaction between client and server, it is easily pursuable for void method invocations by avoiding the reply for the acknowledgement (one-way invocations). In this case, to ensure both the “at most once” invocation semantic (instead of “maybe” semantic provided by some object oriented middleware for void invocations) and consistency when several one-way invocations are performed, some properties from the underlying transport protocol are needed: reliability and FIFO delivery order, respectively. Moreover, flow control is important to reduce the number of losses and consequently of retransmissions, which are very
costly in wireless networks. So, even if, the UDP protocol would be interesting for improving the reactivity of the system, TCP represents the best choice (among the existing and diffused protocols) in that it natively provides the desired properties and mechanisms. However, it is worth to note that some optimizations of TCP that are oriented to stream communication must be released for interactive communication in order to improve the reactivity. In particular, the Nagle algorithm should be disabled in order to immediately send segments when they reach TCP sending buffers².

Starting from these design considerations, TCPTE object communication layer implements a one-way scheme for void method invocations and a synchronous two-way scheme for non-void method invocations, since in this case a specific middleware would be necessary to ensure asynchrony. Unfortunately, a simple substitution of TCPTE RMI middleware with an existing asynchronous one (for example the one proposed in [9]) is not immediate when PWDs are used, since restrictions posed by some client platforms prevent these middleware, which are typically designed for desktop computers, to be used on PWDs without significant modifications.

The session layer deals with connection setup and maintenance between client and server, since for several reasons, such as out of range in wireless communication, the connectivity between client and server could be temporary lost. In TCPTE 1.0, when a connection error occurs, the client freezes the UI and tries to connect to the server.

4. TCPTE evaluation from developer perspective

The main benefit related to the use of TCPTE is the possibility of developing applications independent of the heterogeneity of PWDs. Currently, the framework allows for a seamless distribution of an application between a server and heterogeneous clients, but it is not able to automatically adapt UIs to devices with limited screen size. Therefore, to assess the benefits and the limitations of TCPTE, we give an evaluation of the framework by taking into consideration the porting of two applications initially developed for PCs and characterized by UIs with different complexity. The first one was αEngine (see Figure 3), an open source workflow engine able to execute distributed processes modeled with XPDL both in local area networks and in the Web. The second one was Bloof (see also [10]), a Java library for analytical processing of version control data. Bloof goal is helping developers, project managers and software engineering researchers in comprehending the evolution of a software system.

![Figure 3. WfMC compliant αEngine architecture and remote monitoring with a PWD](image)

² Telnet is the most famous example of interactive and request/reply application built atop TCP by disabling the Nagle algorithm. HTTP is another example in which TCP is optimized with persistent connections and pipeline instead of using a more natural protocol such as UDP for request/reply interactions.
Table 1 shows the effort required from a single developer for porting the two applications onto PWDs. Porting *α*Engine took only few hours because of the simplicity of its UI. The UI layout was slightly modified to adapt the visualization of some graphical components to devices with a limited screen size.

<table>
<thead>
<tr>
<th>Application</th>
<th>Gui/Application analysis</th>
<th>Code analysis for Component</th>
<th>Code Analysis for Layout</th>
<th>Layout impl.</th>
<th>Component adaptation</th>
<th>Testing</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>α</em>Engine</td>
<td>2 hours</td>
<td>0</td>
<td>0.5 hours</td>
<td>0.5 hours</td>
<td>0</td>
<td>1 hour</td>
<td>4 hours</td>
</tr>
<tr>
<td>Bloof</td>
<td>1.5 days</td>
<td>1 day</td>
<td>0.5 day</td>
<td>1 day</td>
<td>1 day</td>
<td>1 day</td>
<td>6 days</td>
</tr>
</tbody>
</table>

Table 1. Comparison between *α*Engine and Bloof porting times

On the contrary, porting Bloof onto PWDs required more efforts (six days of one developer) since its UI, named Bloof-Browser, is too large and complex to be completely shown on a PWD without changes. In particular, an analysis of the application structure and the understanding of graphical interface characteristics was needed to adjust the UI to a 160 x 160 pixel display resolution. Finally, the source code was modified to convert the Swing application into a TCPTE AWT application. The reorganization of the UI entailed a redefinition of the layout to adapt it to reduced size displays. To this end, the toolbar was removed and menus were updated to offer all the functionalities of the original application. Moreover right-button clicks were replaced with specific buttons and the main three areas of the UI were handled through a JTabbedPane.

5. TCPTE evaluation from user perspective

To assess the effectiveness and the efficiency of the proposed approach and to validate the design choices discussed in § 3, several performance measurements were conducted to compare TCPTE with existing technologies. Among these, VNC was chosen for three reasons: (1) a version for personal devices exists, (2) it represents an example of distributed presentation approach, (3) inside its class it performs better than other frameworks [11].

All the measurements were performed by using the testbed depicted in figure 4-a. A Pocket PC was wired connected with a measurement station (MS) through a USB cable, whereas this station was connected with a server through wired and wireless channels ranging from Ethernet to GPRS.

A first analysis aimed at establishing how TCPTE and VNC depend on the network bandwidth. A network shaper was installed on the MS in order to throttle the available bandwidth in the range from 14 Kbps to 1 Mbps. For this experiment, a fast Ethernet network was installed between the MS and the server. We noted that both TCPTE and VNC are dependent on the bandwidth only for small values of it. When the available bandwidth grows over 33.6 Kbps, the performances of both the technologies remain unchanged. As consequence of this result, we decided to perform other experiments varying the network latencies by changing the network between the MS and the server. It is worth to note that the results obtained with the first experiment validated the testbed for all the other measurements. In fact, even if the USB connection between the PDA and the MS could be a bottleneck when a network with high bandwidth is used to connect the MS with the server, the obtained results show that the bandwidth becomes a real bottleneck only for values under 33.6 Kbps which is lower than the bandwidth of the USB connection.

Therefore, a second experiment aimed at comparing TCPTE and VNC with three different applications in several scenarios characterized by different wired and wireless networks, fast Ethernet, WiFi (11 Mbps), Bluetooth and GPRS, whose latencies measured by using the ping command are reported in figure 4-b. For Bluetooth, we used BNEP [12] in order to emulate an Ethernet network atop the L2CAP layer provided by the Bluetooth network architecture [13].

*Developing Java-AWT Thin-Client Applications for Limited Devices – Draft version.*
applications (available from the TCPTE web site) used for the experiment were: ColorApps to measure the reactivity; aEngine to measure the visualization latency with small-medium size UIs; Bloof, to measure the bootstrap and visualization latency with medium-large size UIs. The measure of the reactivity is the average of the sample times between a request from the PDA to the reception of the response that changes the color of the UI background. The measure of visualization latency is the average of the sample times from the command request for the visualization of the workflow diagram to the last acknowledgement sent by the client after the diagram visualization. The measure of the bootstrap and visualization time is the average of the sample times from the request for the execution of the application and the last acknowledgement sent by the client after the visualization of the UI. The variance measured in every experiment was very low.

Figure 4-c compares the response times of VNC with two different configurations of TCPTE, a first one based on a TCP layer adopting the Nagle algorithm and a second one that disables the algorithm. With the first configuration,
TCPTE performances are comparable with those offered by VNC for Ethernet and WiFi networks, whereas with higher latency networks TCPTE response times are better.

By disabling the Nagle algorithm, TCPTE response times are much lower due to the elimination of delays imposed by the delayed acknowledgement (about 200 ms for each acknowledgement) mechanism of TCP. Figure 4-d and 4-e show an analogous behavior for the bootstrap and visualization of both medium and high size UIs. In particular, with Bloof, VNC requires about 2.3 s to show the UI whereas TCPTE shows the UI in 1.2 s over a Bluetooth network. However, the diagram in fig. 4-f highlights a significant difference in term of data exchanged between client and server, while the performance ensured by TCPTE is not so exciting. This is due to the fact that VNC is based on a raster approach in which the UI image is transferred through a continuous data flow from the server to the client whereas TCPTE is a component based approach that for each component requires an interaction between client and server. If this interaction is synchronous (non-void methods), the framework suffers of network latencies, as discussed in § 3. So, to evaluate the best performance that could be obtained by using an asynchronous RMI middleware, we explicitly transformed, at application level, each two-way method call in an asynchronous one-way invocation with a call-back to return the value. In such a way, while VNC performances remain unchanged, figure 4-e shows clearly that asynchrony ensures excellent performances to TCPTE.

Moreover, Figures 4-c,d,e show that on networks characterized by high latency (like GPRS), the adoption of the Nagle algorithm improves the performances of TCPTE.

To conclude our analysis from user perspective, we evaluated the usability of TCPTE comparing it with VNC and in general with other existing solutions for PWDs based on distributed presentation. By following this approach, each action on the UI throws an interaction between client and server. Some actions, such as scrolling a text area or changing a panel in a tabbed pane, require a sensible delay to visualize information. In general, this delay is acceptable if the client is connected to the server through a fast network, but it becomes unacceptable when a slow network is used. With very slow networks, such as GPRS, the distributed presentation approach raises significant delays also for a click on a menu. On the contrary, with the remote presentation approach, scrolling a text area, changing a panel or a menu do not require any interaction with the server since the information needed for updating the UI is already present on the client. Client-server interactions occur only when the user fires an event that is to be processed by the application logic.

6. Concluding remarks and future work

The paper presented TCPTE, a framework to develop Java AWT Thin-Client applications for limited devices masking their heterogeneity and exploiting the advantages of a traditional client-server application. A performance evaluation showed that the TCPTE approach is promising for wireless environments since the performances obtained are significantly better than those presented by VNC. Moreover, we can conclude that TCPTE appears more usable than VNC due to its high reactivity and the possibility of adopting known and optimized components for PWDs. Moreover, the small traffic produced by TCPTE (see fig. 4-f) makes it suitable for shared wireless channels in large open spaces.

To understand the future impact of such a technology, it is important to analyze how the wireless world will evolve in the next years. In fact, if the trend will be to increase the available bandwidth, reducing connection costs, and leave devices very simple, then solution to offer services through thin-clients will be winning. Vice versa, if the devices will become more powerful then, probably, the client-server model will predominate. TCPTE allows for a reduction of the risks, as it supports the development of applications that can run in both ways.
As a consequence of the growth of communication bandwidth, most content providers are offering multimedia services, sending to wireless users images, and real time video and audio streams. According to this trend, TCPTe AWTServer and AWTClient will be enhanced with multimedia components to play video and audio on mobile devices.

The session layer will be extended in order to support: (1) application wake-up and push messages to notify events from the TCPTe server, such as applications availability; (2) session handoff to allow for starting an application on a device, freezing it at any instant, and resuming it on another device.

As resulted from the evaluation, at the communication layer the framework can be enhanced by implementing a middleware for asynchronous remote method invocations in order to tolerate high latencies of some networks through the overlapping of computation and communication. In this middleware both void and non-void methods will be implemented through asynchronous, non-blocking communication. For non-void methods, a transparent synchronization mechanism based on future variables will be implemented. Moreover, to reduce the response times, a mechanism that enables or disables the Nagle algorithm on the basis of the value of the network latency will be implemented on top of the transport layer.

Acknowledgements

We would like to thank Giancarlo Tretola, Angelo Scarnata, and Nicola Barbiero, Carlo Carrabs, Alessandro Craviolatti, Roberto De Pascalis, Franca Perrina, for their contribution in the development phases of TCPTe and aEngine, respectively, and Gabriele Venturi for the interesting discussions that animated this work.

References
