Towards a Java™-Based Enterprise Client for Small Devices

**Bill Bush, Bernard Horan, Vipul Gupta,
Phillip Yelland, and Patrick Chi**

Towards a Java™-Based Enterprise Client for Small Devices

Bill Bush, Bernard Horan,
Vipul Gupta, Phillip Yelland, and Patrick Chi

SMLI TR-2002-120 December 2002

Abstract:

The goal of the work reported here was to explore the use of the Java 2 Micro Edition (J2ME™) platform for applications connected to the enterprise, specifically focusing on Palm-based wireless applications. We found that the Java™ platform on the Palm is still maturing. The Palm itself has been carefully engineered to support small native applications, with a distinctive graphical user interface tuned for its display. Work remains to be done on the Palm to support more complex wireless applications and to make Java-based applications competitive. We also found that wireless enterprise applications in general are somewhat problematic, due to issues of network reliability, availability, bandwidth, and provisioning. Significantly, programming languages and their platforms are not the gating factors to large scale wireless deployment.

This work was performed in 2000 and 2001, before the current commercial deployment of Java-enabled mobile devices and faster wide-area wireless data services (such as GPRS). We hope to repeat our experiments using these technologies.

M/S MTV29-01
2600 Casey Avenue
Mountain View, CA 94043

email addresses:
bill.bush@sun.com
patchi@cal.berkeley.edu
vipul.gupta@sun.com
bernard.horan@sun.com
phillip.yelland@sun.com

© 2002 Sun Microsystems, Inc. All rights reserved. The SML Technical Report Series is published by Sun Microsystems Laboratories, of Sun Microsystems, Inc. Printed in U.S.A.

Unlimited copying without fee is permitted provided that the copies are not made nor distributed for direct commercial advantage, and credit to the source is given. Otherwise, no part of this work covered by copyright hereon may be reproduced in any form or by any means graphic, electronic, or mechanical, including photocopying, recording, taping, or storage in an information retrieval system, without the prior written permission of the copyright owner.

TRADEMARKS
Sun, Sun Microsystems, the Sun Logo, Java, J2ME, J2SE, PersonalJava, JDK, Java Card, Java Virtual Machine, JavaMail, Kjava, iPlanet, Solaris, Java Community Process, and Sun ONE are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

For information regarding the SML Technical Report Series, contact Jeanie Treichel, Editor-in-Chief <jeanie.treichel@eng.sun.com>. All technical reports are available online on our Website, http://research.sun.com/techrep/.

Towards a Java™–Based Enterprise Client for Small Devices

Bill Bush, Bernard Horan, Vipul Gupta, Phillip Yelland, and Patrick Chi

Sun Microsystems Laboratories
2600 Casey Avenue
Mountain View, CA 94043

December 2002

Introduction

The work described in this report is an outgrowth of an earlier effort at Sun™ Labs, the Spotless project [Spotless]. The original goal of that project was to build a small Java™ implementation for the Palm Pilot. The resulting virtual machine ultimately became the K Virtual Machine product [KVM]. Experimental work continued with Spotless, leading to an SSL–enabled version [KSSL] and a persistent version [P–Spot1], [P–Spot2].

The goal of the work reported here was to explore the use of Spotless–KVM as a client platform for applications connected to the enterprise, with a specific focus on Palm–based wireless applications. A number of subgoals were consequently pursued:

- building Java applications using the Palm platform as a wireless client in a multi–tier application architecture, to evaluate the feasibility of deploying such clients.
- developing and evaluating Java–based graphical user interfaces suitable for PDAs (rather than mobile phones), in conjunction with starting the PDA Profile effort [PDAP]; and
- implementing a general mechanism for calling Palm OS functions from KVM applications, in order to integrate the KVM more tightly with the Palm operating system, so that, for example, personal information manager (PIM) data could be accessed via Java applications.

This document thus describes and evaluates the client applications that were built and reviews the experience of building wireless Java applications for the Palm, discusses the user interface alternatives evaluated, and presents a general mechanism for calling Palm OS functions. It concludes with some observations about wireless clients in enterprise applications.

Corporate Email Access from Wireless Client Devices

Given the emergence of both wireless and Java technology on small devices, we wished to explore the development of Java–based client applications on such devices. In particular, we undertook the task of providing Sun employees with secure corporate email access via Palm–based devices. This section describes several approaches to the problem, and concludes with the lessons learned.

INTRODUCTION

According to most industry watchers, wireless devices will play a big role in the Internet's future evolution. However, people's opinions differ on how that role will evolve. Some believe that "wireless is different", existing Internet protocols are simply inadequate, and an entirely new protocol stack is needed. Others believe that suitable modifications to existing protocols will suffice. Unfortunately, quantifiable justifications for positions on either side of the debate are often missing.

Mobile data services such as Palm.net and WAP use a trusted gateway (or proxy–based) approach to securing communication. For example, when someone uses a Palm VII to connect to the secure web server of a content provider, SSL (in the form of HTTPS) is used only between the web server and the Palm.net gateway operated by Palm. Communication between the gateway and the Palm VII is performed via a proprietary wireless protocol with its own, different security. The gateway is in a position to see all communication in the clear as it is transformed from the wireless protocol to HTTPS and vice versa. In general, the security of the proprietary wireless protocol must be independently evaluated, and may not be as secure as HTTPS.

Thus there is increasing demand for end–to–end security between thin clients and secure servers. One possible means of meeting this demand is to use SSL from the client directly to the server. However, SSL support is not found (or possible) on all small devices. Alternatively, a secure connection can be established between a client and a server using encryption specialized for small devices. This, however, requires changing the server and perhaps the server architecture.

End–to–end security is required in the case of Sun's corporate email. This specifically means establishing a secure connection from the client through Sun's firewall to a server within Sun's Wide Area Network (SWAN).

We explored the problem of providing secure communication to an email server inside SWAN by experimenting with two architectures:

- a specialized encrypted connection architecture using the Palm VII and Certicom's encryption technology, and
- an SSL–based architecture using the Sun.Net Internet portal into SWAN and the Palm V running SSL.

For the second architecture, we developed three client applications:

- the Macchiato client (also known as the ToGo client),
- the Sun Labs SNMail client based on KJava, the original Spotless user interface, and
- the Sun Labs MIDP client.

We were assisted in the implementation of both architectures by the software development company Bonita Software. They also produced the client application for the Palm VII architecture, as well as the Macchiato client for the SSL architecture. The KJava and MIDP clients were both developed by one of this report's authors, Vipul Gupta.

The desired features in the email client were similar to those provided by most desktop–based email clients:

- settings to specify incoming and outgoing email servers (both POP and IMAP), user name, and password;
- a user interface to present the Enigma security challenge and capture the user's response (the Enigma token card is used to authenticate access to SWAN);
- an inbox to which new email is appended, with the ability to view email headers and fetch email bodies;
- an outbox for storing email messages yet to be sent;
- the ability to reply to a message and compose a new message; and
- the ability to delete a message from the client application and (in the case of IMAP mail servers) from the server.

The Palm VII / Certicom Architecture

This work was begun in the spring of 2000, soon after the Palm VII became widely available. It was a joint effort between Sun IT, iPlanet™, Java Software, Sun Labs, Bonita Software, and Certicom. The size of the collaboration reflected the difficulty of the undertaking. The solution required: a custom secure communication protocol (provided by Certicom); client and server encryption libraries (provided by Certicom); changes to the Java implementation on the Palm (made by Java Software and Sun Labs); a special security server to handle encryption and decryption of communication with the Palm VII (implemented by iPlanet); installation of that server in the Sun network's "demilitarized zone" (overseen by Sun IT); and email client and associated server software (developed by Bonita Software).

Figure 1 presents a diagram of the resulting architecture. The Palm client encrypts its messages (containing email text and control information), then packages and transmits them using standard Palm VII INet APIs. The Palm.net gateway sends these messages as HTTP transactions through the Sun.net gateway to a security server in Sun's DMZ. This server decrypts the messages and hands them to web server, which then sends email transactions to a back end email server. Email server responses flow symmetrically in the reverse direction. Note that:

- The PDA client includes the KVM plus Certicom encryption (using elliptic curve cryptography) plus the client application.
- The Palm VII communicates with the Palm.net gateway using Bell South's Mobitex pager network. The round trip time is ~ 20 seconds.
- The security server manages secure sessions with the Palm VII. It uses the same Certicom encryption technology as the Palm VII. The complete security protocol is provided in an appendix.

As can be seen from the diagram, the architecture is not restricted to email. However, due to time constraints, only email and LDAP applications were implemented. The user interface for the client was developed using the Spotless UI that was part of the original release of the KVM.

The project was successful as a prototype. All the components functioned and communicated as required, demonstrating the feasibility of the architecture and the viability of the Palm device as a platform for small, useful Java applications.

However, deployment with real users was hindered by several issues. First, the architecture required a special security server. Vetting and maintaining this server demanded a substantial commitment and investment on the part of SunIT. Second, a nonstandard KVM with proprietary libraries was also required, which in turn introduced maintenance and licensing issues. Third, the Palm VII service was not available worldwide. Fourth, the Palm VII's limited memory imposed serious constraints on the operation of the client in terms of features and email capacity (alleviated to some extent in the Palm VIIx and more fully in later KVM versions).

Thus, although the project demonstrated secure wireless email functionality, widespread deployment was prevented by pragmatic factors.

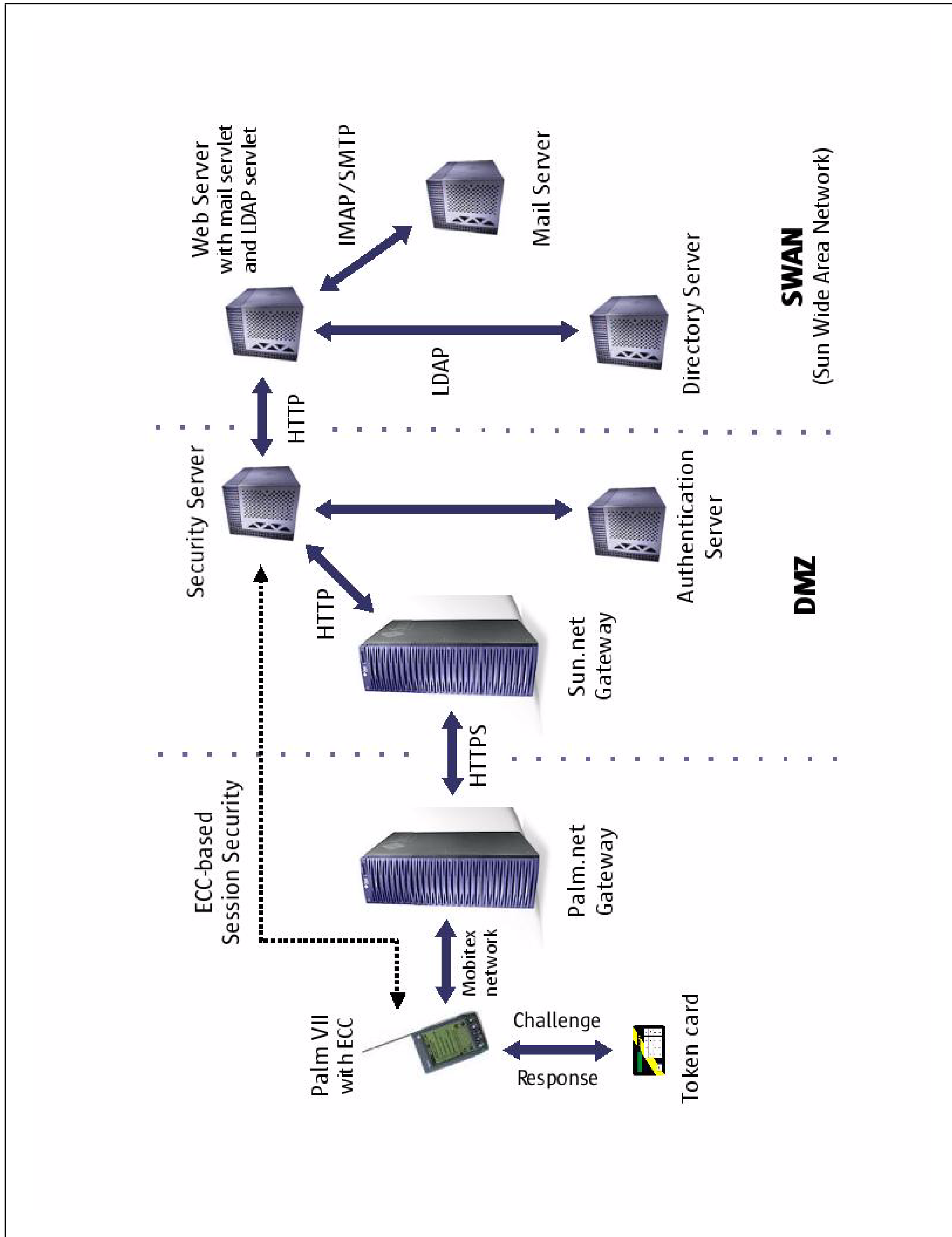

FIGURE 1. **Palm VII Architecture**

THE SSL ARCHITECTURE

Even though there were problems with the Palm VII project, the results were tempting enough that we undertook a second experiment.

SSL is the most commonly–used security protocol on the Internet, but is usually perceived as being too heavyweight for mobile devices. Nonetheless, one of this report's authors, Vipul Gupta, believed it possible to implement a Java–based SSL stack on Palm OS, specifically on a Palm Vx with OmniSky wireless IP connectivity. This effort was successful [KSSL].

We decided in our second experiment to use this KSSL base, and implement another set of clients and servers based upon it. As with the Palm VII project, our goal was, if possible, a wide scale deployment. We evaluated three clients, one derived from the Palm VII client, and two based on a proof–of–concept client written to test the SSL implementation.

Figure 2 provides an overview of the SSL architecture. The architecture applies to any KVM–enabled device with the computing power of the Palm and IP connectivity. Due to the use of standard SSL, the Sun.Net portal is used without modification. The architecture is not email–specific—each application employing it simply requires a client and a corresponding server inside SWAN that can take the HTTPS requests from the client, forward them on to the appropriate server (such as IMAP or LDAP) formatted properly for that server, and send back responses.

KSSL Implementation and Resource Requirements

SSL requires the underlying platform to provide a reliable, bi–directional byte–stream network service (typically TCP) and basic cryptographic algorithms for message hashing and encryption. Sun's J2ME™ platform supports TCP connections but does not include any cryptographic algorithms. We enhanced it by adding native C code for RSA, RC4, MD5 and SHA. We chose to expose the new functionality through a subset of the javacard.security.* and javacardx.crypto.* APIs rather than the java.security.* and javax.crypto.* APIs. This decision was rooted in the realization that several KVM devices will also be equipped with Java–based smartcards (GSM phones already have smartcards in them). On these devices, one can leverage the Java Card™ hardware for CPU intensive cryptographic transformations.

The SSL protocol code, including certificate parsing, is all written in Java. The KSSL API provides two public classes: SSLStreamConnection and Certificate and an interface called HandshakeListener for callbacks.

Cryptographic additions (RSA, RC4, MD5, SHA) increase the size of KVM.prc on the Palm from 278KB to 310KB. The SSL Java classes add an additional 32K to an SSL–enabled platform. Runtime memory requirements depend on the size of SSL records sent and received. A simple J2ME program can retrieve a small web page over HTTPS using around 55KB of heap memory on a Palm.

THE MACCHIATO APPLICATION

One KSSL–based client–server pair we tested was based on the Palm VII client and server code base. The Certicom security mechanism was removed from the client and the simpler KSSL mechanism put in its place. The security server was therefore no longer needed and was removed. The web server was modified to handle transactions forwarded directly from the Sun.net gateway. We named this client–server pair Macchiato (which means "marked" in Italian, and refers to espresso with a bit of milk or whipped cream). Both the Palm VII and Macchiato applications were developed by Bonita Software in collaboration with Sun. The remainder of this section describes the operation and appearance of the resulting client.

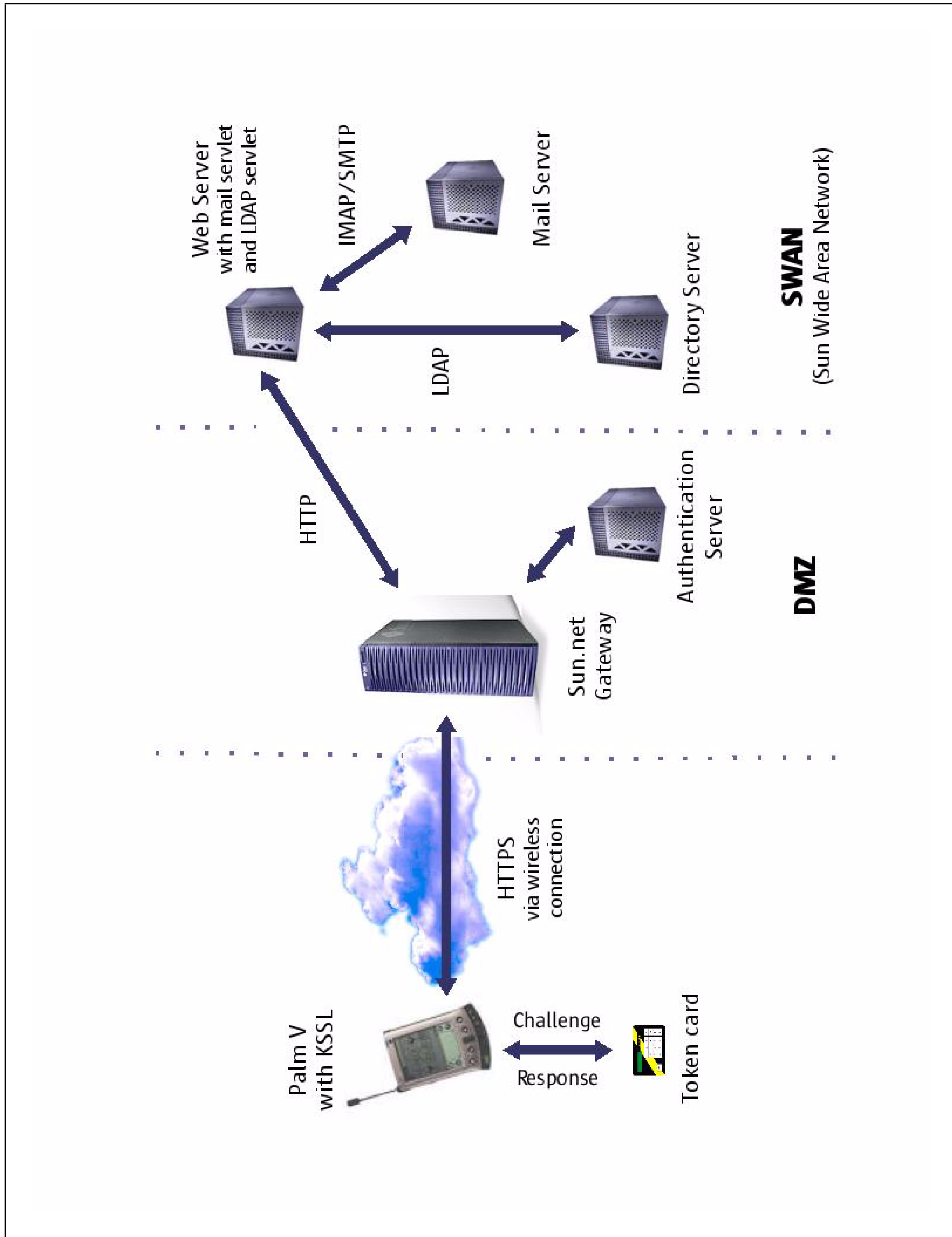

FIGURE 2. SSL Architecture

The Macchiato email client welcomed first time users with a "Settings" screen as shown in Figure 3. This screen captured values for fields such as email Server, UserID and Password required to connect to the email server and authenticate the user. These settings were stored permanently in the underlying Palm Database and were easily edited. These fields (required fields are indicated with *) were:

- *Type, the protocol used to connect to the email server (such as IMAP or POP3)
- *Incoming email server (such as ha1mpk-mail.eng.sun.com or pop3.mail.yahoo.com)
- Outgoing email server (example: smtp.mail.yahoo.com)
- *User
- *Password
- EmailID

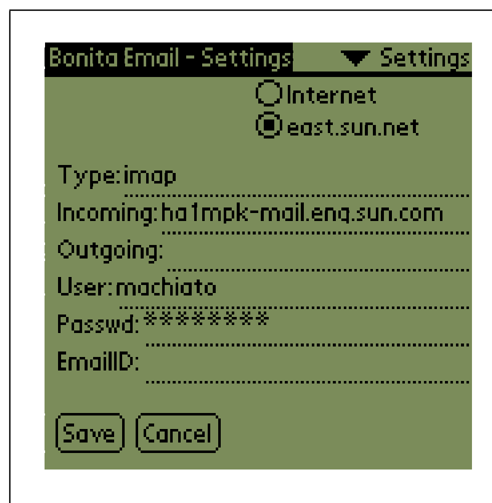

FIGURE 3. The Macchiato "Settings" screen

The Enigma Challenge–Response Screen (Figure 4) provided secure access to the Web Server (as illustrated in Figure 2) and the user's email server. This screen allowed users to authenticate with east.sun.net and secure a cookie to avoid authentication for further requests.

The Inbox, shown in Figure 5, provided the following features:

- Login and fetch most recent eight messages by clicking on **Refresh**.
- Traverse inbox back and forth by using **Previous** and **Next** buttons.
- Compose new email message.
- Select a message and view it.
- View status of Inbox (Update read and unread message counts).

All unsent messages were stored in the outbox (Figure 6), which provided the following features:

- Edit an unsent message.
- Delete an unsent message.
- Send a highlighted message.

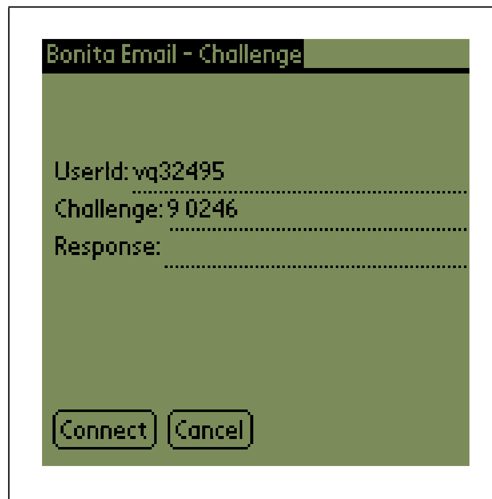

FIGURE 4. The Enigma Challenge–Response Screen

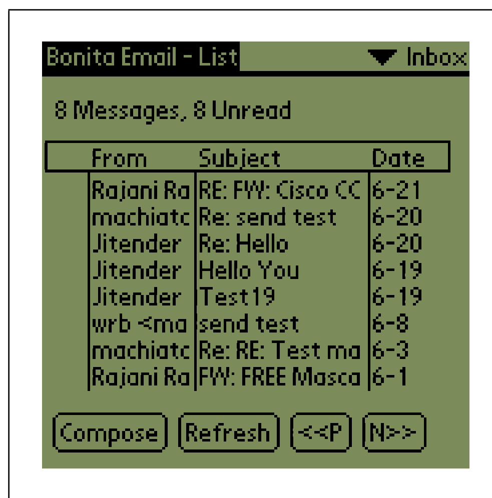

FIGURE 5. Inbox

- Compose a new message.

The email client allowed users to work in disconnected mode. A user could compose email messages and save them for future dispatching or whenever the network connection became available. When at the Draft screen (Figure 7), the user had access to the following features:

- Edit a message.
- Delete a message from the draft screen.
- Send a highlighted message.
- Compose a new message.

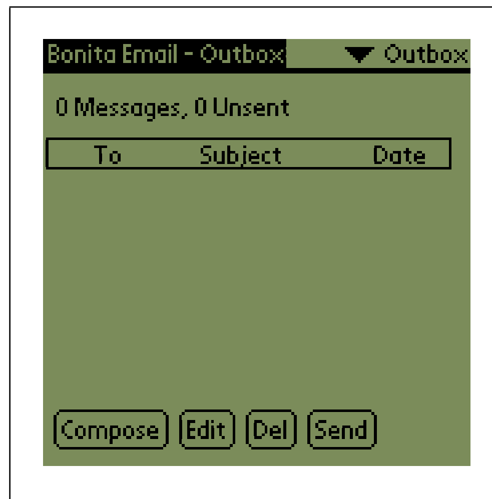

FIGURE 6. Outbox

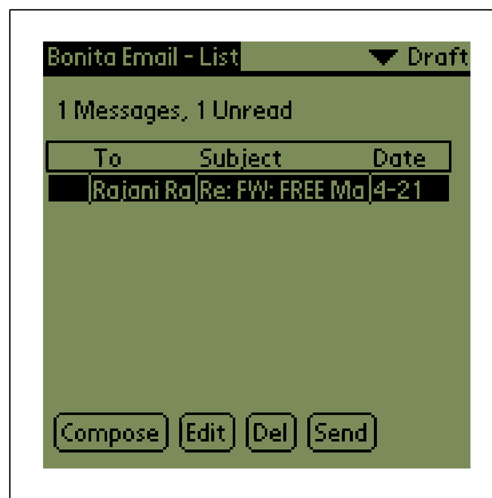

FIGURE 7. The Draft Screen

When composing a message, a screen popped up containing the 'To', 'From', 'Subject' and 'Body' fields (as shown in Figure 8). The 'From' field was pre–populated with the user's email id. In response to a **Reply** action, the 'To' and 'Subject' fields were also automatically populated from the opened message. The body of the opened message was not included in the reply.

The "view message" feature allowed a user to view a highlighted message from the Inbox. As shown in Figure 9, the ShowMessage screen displayed values for 'To', 'From', 'Subject' and a preview of the message body (first 60 characters). The ShowMessage screen also offered the following actions:

- **Full** (Full Body): This feature fetches and displays the full message body.
- **Del** (Delete): This feature deletes this message from the user's email server as well as from the Inbox.
- **Reply**: Reply to this message.

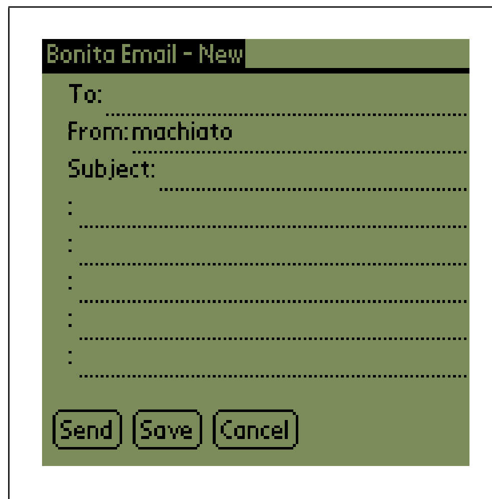

FIGURE 8. **Composing a new message**

- **Done**: Go back to Inbox.

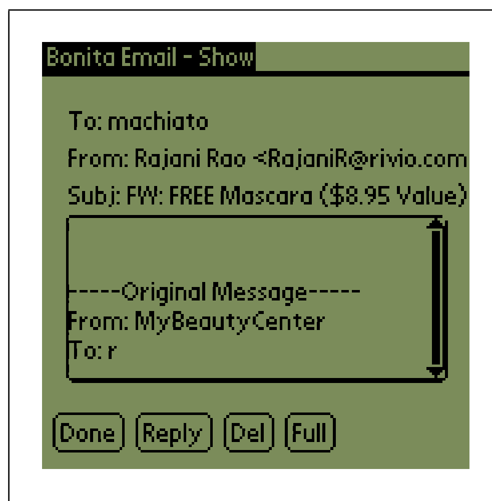

FIGURE 9. **The ShowMessage Screen**

The email client offered a drop down menu for easy navigation among folders, settings and other options.

THE SUN LABS SNMAIL APPLICATION

As KSSL was developed, a companion test email application was written, appropriate for demos but not for general use. For the work described here it was rewritten. A more complete server was implemented, and two new clients were written, the first using the KJava UI libraries and the second using the MIDP libraries. The first client was called SNMail, the second MIDlet mail (described below).

This work used the same KSSL architecture as the Macchiato application. The client communicated with the Sun.Net gateway (shown in Figure 2 on page 6) using HTTP (version 1.0 with cookies) over SSL (version 3.0) over TCP/IP. As such, the Palm handheld appeared no different than a remote laptop (albeit a slow one) to the Sun.Net infrastructure.

SNMail required a successful login with the Sun.Net authentication framework using a challenge–response exchange over SSL, as illustrated in Figure 10.

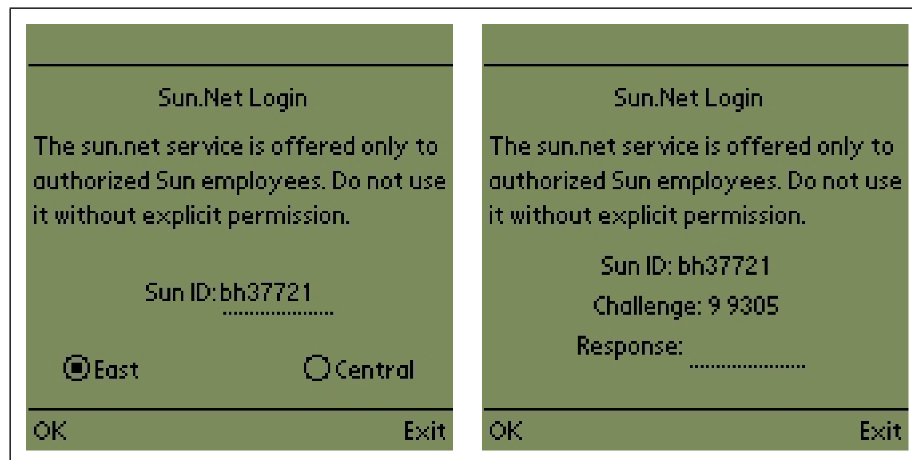

FIGURE 10. The Sun.Net Login and Challenge screens

After a successful Sun.Net login, the user was presented with a mail preferences screen (as illustrated in Figure 11) where the mail server, user name, and password were specified.

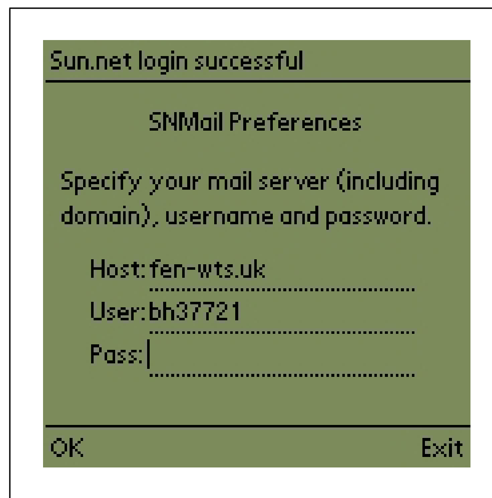

FIGURE 11. The email login screen

When the user selected **OK** on the mail preferences screen, an HTTP GET request containing the user's name, password, and server was sent to the mail servlet running inside SWAN (see Figure 2). The mail servlet used this information to

create a new mail session and then sent back headers from the most recent fifteen messages (this parameter was configurable in the servlet).

As shown in Figure 12, each message header included the message index, the subject, message flag if any (e.g. N for new, D for deleted, U for unread), sender, and month and date. Since each header took two lines, there were more headers than could fit on a single Palm screen. The user could use the scroll buttons to view off–screen headers. The widget used to display the text received from the servlet was developed especially for this application. Compared to the com.sun.kjava.ScrollTextBox class, it implemented a faster and smarter rendering engine and fixed a bug regarding how newline characters were displayed. However, it did not support manipulation of the scroll bar by tapping on the screen (only the scroll button was supported).

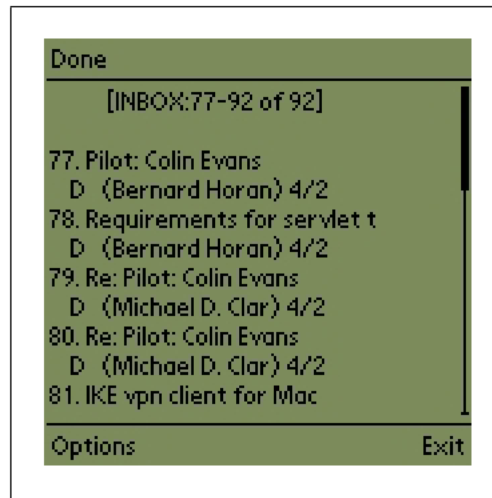

FIGURE 12. Email headers

Selecting **Exit** on the message headers screen terminated the mail session and automatically expunged any deleted mail. Selecting **Options** presented a list of actions available to the user, as shown in Figure 13. These included:

- Fetching a message by specifying its index (note that the index is the first piece of information included in a message header).

  A fetched message (as illustrated in Figure 14) displayed the sender's full name and address (only the name, possibly truncated, was shown in the header), date (including day, time and year), subject, the 'To' and 'Cc' fields, followed by the message body.
- Deleting one or more messages by specifying a single index or an index range (for example, *132–139*).
- Scrolling up or down the list of message headers. The user was not required to specify either the start or end index. For each mail session, the mail servlet maintained internal state such as the current message window (start and end indices), and the index of the current message. By maintaining this state in the servlet, it allowed us to create a very simple client that required less internal state of its own and thus reduced the memory requirements of the client.
- Getting the next chunk of the current message. In order to minimize unnecessary communication, long messages were sent to the user in chunks of (approximately) 1024 bytes. This allowed the user to preview a small portion of the message before deciding whether or not subsequent portions should be downloaded. The mail servlet included the line 'More…' at the end of each message chunk except the last.

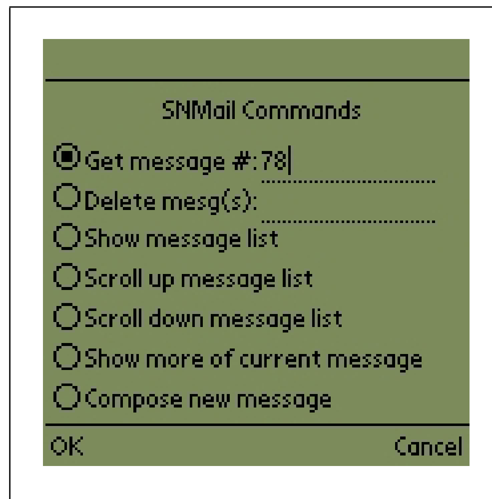

FIGURE 13. Email options

FIGURE 14. Viewing an email message

- Composing a new message. SNMail allowed users to compose and send short messages—those that would fit within two screens on the Palm. The user was required to use the scroll button to go back and forth between the two screens. SNMail did not support attachments, and the names of the email recipients were required to fit on one line.

The mail servlet used the JavaMail™ APIs to communicate with an IMAP or POP3 mail server. By specifying different parameters to either GET or POST requests on the mail servlet URL, SNMail performed various operations on the user's mailbox. The mail servlet was capable of generating XHTML–Basic (a constrained form of HTML), HDML (for WAP phones) and plain text, depending on the type of client accessing it. Client type determination was based on the use of certain heuristics that took into account the User–Agent header included in the HTTP request. SNMail was quite simplistic and was unable to render HTML, so the servlet only sent simple text in its HTTP responses.

We successfully used these applications on a Palm Vx connected to a wireless OmniSky modem and a Palm IIIc connected to a wireless Metricom modem. The OmniSky service used AT&T's CDPD service, which was available within the continental USA, whereas Metricom's Ricochet service was only available in the Bay Area, Washington DC, Los Angeles, and a few other major cities. Older Palm models such as the Palm III and Palm V were not supported because the heap space available to Kjava™ network applications on these devices is only around 55 Kilobytes. The SNMail application, however, require around 90KBytes. Any Palm device with 4MB or more memory running Palm OS3.5 provides a heap greater than 200KB, which was more than sufficient for our applications.

THE SUN LABS MIDP MAIL CLIENT

After SNMail was implemented, the MIDP UI libraries became available on the Palm, and the Kjava libraries were deprecated. The client was therefore reimplemented using MIDP, and KSSL was integrated into MIDP.

We created a set of MIDlets (Java applications written for MIDP) that allowed employee directory and calendar access in addition to email retrieval. The MIDlets required a Palm or Handspring PDA containing 8MB of memory and Palm OS 3.5 or later (such as the Palm Vx, Palm IIIc, or Visor Platinum). Approximately 900 KB of free memory was needed to install Palm MIDP and the application (the application was around 70KB).

After launching the client application, the user was presented with a list of MIDlets named (in order from top to bottom): Configure, Login, Nametool, Caltool, Mailtool and Logout (as shown in Figure 15). To launch any of these MIDlets, the user just tapped on the arrow button to the left of the MIDlet name.

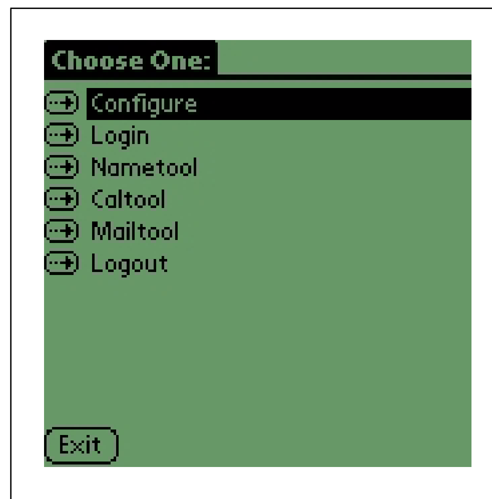

FIGURE 15. The List of MIDlets

The Configure MIDlet captured the user's configuration preferences. As shown in Figure 16, these included the user's Sun.Net user id, username (or email id) and password, plus the location of the user's email server.

The screen of the Login MIDlet (shown in Figure 17) displayed the user's Sun.Net ID as configured above. Tapping on **Ok** initiated an SSL connection with the preferred Sun.Net gateway. The first time the client communicated with an SSL server, it went through what is called a full SSL handshake. This took up to 10 seconds on the Palm. Subsequent communication with the gateway benefitted from SSL session reuse, which reduced the handshake delay to just a couple

FIGURE 16. The Configure MIDlet

of seconds (more detailed information is available in a Sun Labs technical report [KSSL]). The challenge sent by the Sun.Net gateway was displayed next. The user computed the appropriate response for this challenge using a token card and tapped on **Ok** to send the response. If there was a login error (authentication failure or time out), an alert box was displayed. Upon a successful login, the Login MIDlet terminated automatically.

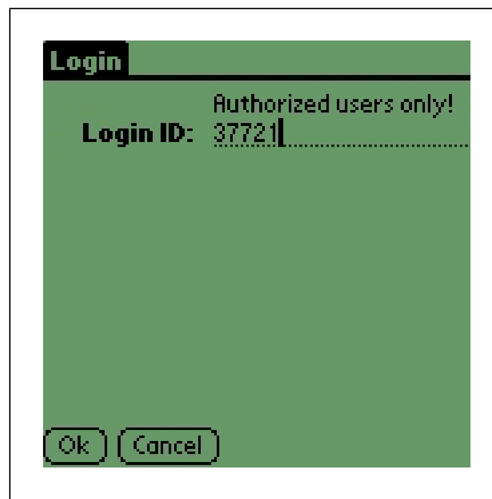

FIGURE 17. The MIDP Login MIDlet screen

The Mailtool MIDlet (illustrated in Figure 18) implemented the features for the email client. If the user had not specified a password in the initial configuration, the MIDlet prompted for it. Tapping the **Fetch** button retrieved the user's email.

The mail servlet responsible for generating content for this MIDlet returned 16 message headers at a time starting with the most recent, as shown in Figure 19. This allowed the servlet to respond with reasonable delay (15–20 seconds) irrespective of the size of the user's InBox. A brief body preview was included with each message as a means of reducing network communication. The **Next Hdrs** and **Prev Hdrs** commands could be used to navigate through the user's InBox

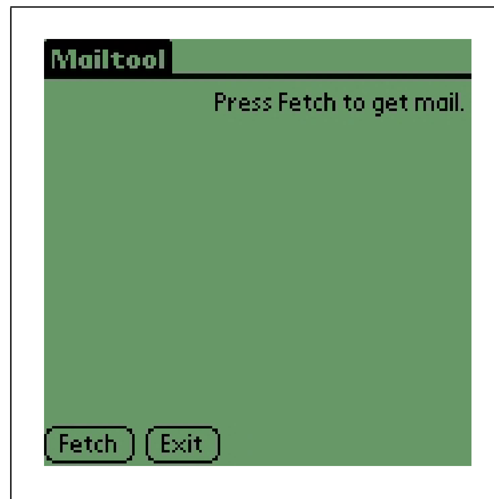

FIGURE 18. **The Opening Screen of the Mailtool MIDlet**

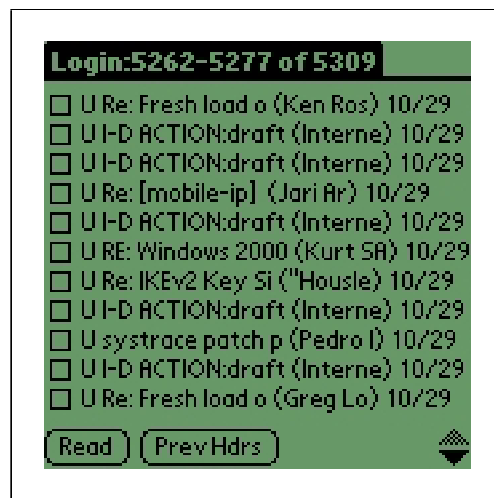

FIGURE 19. **The Mailtool screen showing email headers**

16 message headers at a time. The range of messages on display was shown in the top left of the screen. The complete list of commands allowed on the header list screen was accessed by tapping the **Menu** soft button on the Palm's silkscreen, under the **Actions** menu (as shown in Figure 20). One particularly useful feature was **Search** which provided searching of the user's InBox based on the date when the message was sent and/or key words in various attributes (such as sender or body). The **Delete** and **Undelete** options allowed the user to flag multiple messages as deleted or un–deleted on the mail server. Deleted messages were automatically expunged when the mailbox was closed using the **Exit** menu option.

To read a message, the user selected the corresponding header and tapped on **Read**. This displayed the first 100 or so characters of the message body (as shown in Figure 21). Based on this preview, the user could decide whether or not the rest of the message was worth fetching. Since the contents of the preview were sent along with the headers, tapping the **Read** button did not involve any network transaction and provided instantaneous access to the preview. The message preview screen and the list of possible actions is shown in Figure 21.

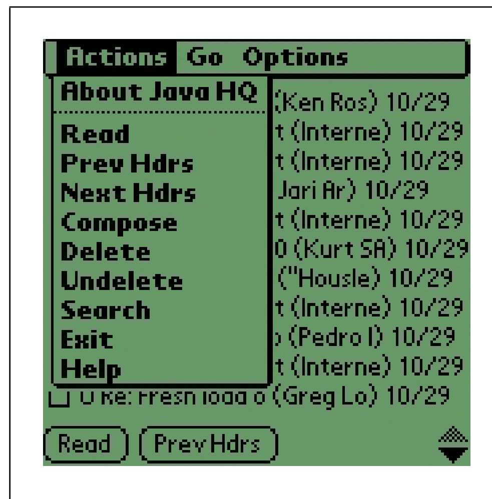

FIGURE 20. **The Action menu on the headers screen**

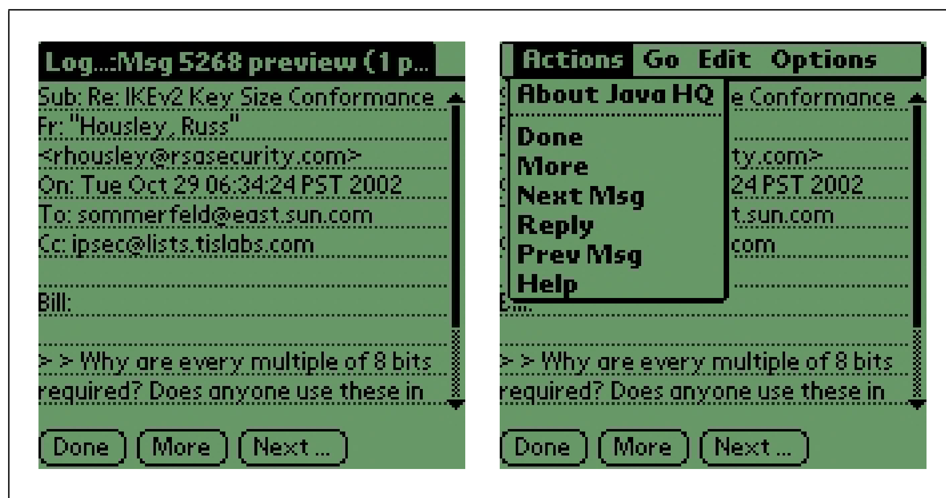

FIGURE 21. **Screens showing a message preview and available commands**

Each message was conceptually divided into a number of parts, each about 1KB in size. Tapping on the **More** button caused the next part of the message to be displayed (the part number being displayed was included in the text box label at the top of the screen as shown in Figure 22). Selecting the **Reply** button on this screen was similar to selecting the **Compose** menu option on the header list screen. Both of them brought up the “New Message” screen (shown in Figure 23); the subject and recipient fields were completed automatically if the user had selected **Reply**. Tapping **Done** on the “Message Preview” screen returned the user to the header list screen.

Selecting **Exit** on the header list screen closed the mailbox (this involved a network transaction) and terminated the MIDlet.

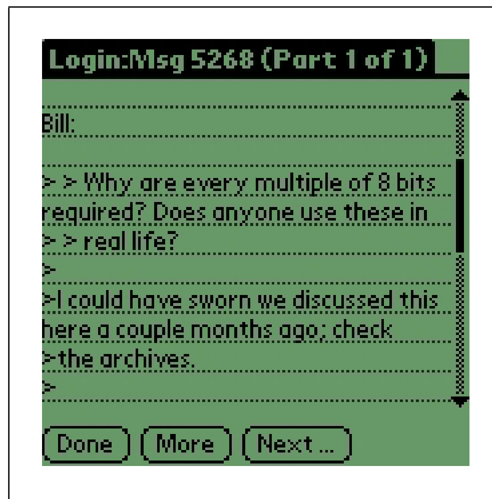

FIGURE 22. **The full message can be accessed in parts using the** More **command**

FIGURE 23. **Composing a new message**

Other MIDlets

The Nametool MIDlet could be used to search Sun's LDAP employee database by full name, last name, first name, user name, or phone number. The search was case insensitive and used partial matches.

Launching the Caltool MIDlet opened a screen that specified the user's calendar based on information previously configured. This information could be edited if the user wished to view someone else's calendar. The user was shown a list of scheduled appointments (or a message indicating no appointments were found). Note that this MIDlet could only display publicly viewable appointments.

User Interface Issues

A few user interface issues related to SSL were identified:

- Experimentation with the Kjava version of the mail application (SNMail) suggests that periodic feedback on the progress of a full SSL handshake helps the user in dealing with the associated delay. On platforms where this delay exceeds a few seconds (3–4), the MIDP implementation should provide such feedback without any special effort on part of the application.
- The SSL protocol relies on the successful validation and verification of the server's certificate. In certain instances, however, it is desirable to let the user decide whether a certificate related error (such as unrecognized issuer, mismatch between name in the certificate and the host name in the HTTPS URL) is serious enough to abort the connection attempt or benign enough to be ignored. This is also the approach adopted by popular browsers. Unfortunately, the small screen on most MIDP devices makes it cumbersome to display enough information about the error to allow a user to make an informed decision. This is an interesting example of how limited screen size can have potential security implications.
- Most popular browsers today provide a visible cue, typically a closed lock icon, to indicate if the current page was downloaded over a secure (HTTPS) connection. Unlike a browser that simply renders information downloaded over the network, a MIDP application often processes the downloaded information and may display it simultaneously (on the same screen) along with information generated locally or derived from an insecure connection. This makes the display of a lock icon on the screen almost meaningless. One might consider providing a special cue only while the secure connection is active (for example, a flashing lock icon) but some devices may allow both secure and insecure connections to proceed concurrently. At this point, it is not clear to the authors which approach is most appropriate. Whatever mechanism is implemented to notify users of secure communication, the platform must ensure that an application cannot fake that notification. For example, if the notification takes the form of a flashing icon, no application must be able to manipulate the area of the screen used by that icon. Given tight screen size limitations, a better idea might be to use a pop-up window of some sort with special window decorations that clearly identify it as having been created by the system rather than an application.

KSSL Application Results

All of the KSSL email applications worked, and had enough functionality to be useful. They were all more usable than the Palm VII Java–based email application. Nonetheless, we could not recommend any of them for wide scale deployment. The Java implementation on the Palm was too slow and quirky (compared to the native platform), user authentication was annoyingly difficult, wireless connectivity was unreliable, and reading and sending email was tedious.

The Macchiato Experience

The Macchiato application was tantalizingly easy to develop. The changes required to the Palm VII client and server were relatively simple and largely salutary, because they simplified the application. Nonetheless, the Macchiato client never achieved the reliability necessary to be deployed. The client was simply not designed with proper concern for either the limited memory available on the Palm (especially with respect to string handling) nor the unreliability of wireless connectivity and the resulting failures and exceptions. When it worked the client was somewhat compelling, allowing the user to tunnel into Sun and read corporate email in any setting, without the burden of a wire or even a laptop. But too often the client hung or died. At one Sun Labs Open House demonstration it hung roughly 29 times out of 30 attempts. Under continuous development it slowly improved, both in terms of reliability and features, but it never became production quality software. See the next section on building applications for some of the issues encountered in its implementation.

The SNMail Experience

Like Macchiato, SNMail was rather simple to implement. The basic ability to read email had been demonstrated in the original KSSL proof of concept. That basic client was primitive *in extremis*, but it was not difficult to design and implement a more advanced, usable client since the client was by nature small and the text–based client–server protocol simple. The SNMail application never suffered the reliability problems of Macchiato. It was as compelling as Macchiato without the problems. Development of it stopped, however, when work on the MIDlet client started.

The MIDlet Experience

The MIDlet applications were the most polished and usable of those tested for this report. The suite concept with login was especially valuable, creating a session in which various client activities could be performed. And like all these applications, under the right circumstances the user experience was compelling. Nonetheless, under the wrong circumstances, which occurred too often, the experience was quite frustrating. Specifically, the wrong circumstances involved: the slowness of the Java implementation, the clumsiness of Enigma authentication with Palm's graffiti, and the unreliability of OmniSky connectivity. Too often during authentication the connection would drop or the authentication would time out due to Java delays or problems with graffiti when entering the eight character Enigma challenge. In one key demo, four login attempts were made over a ten minute period, and only one succeeded. Techniques were available to streamline the login process, but they relied on Sun IT to produce certain keys, which were only sporadically supplied. As a result the MIDlets were a good proof of concept, but were not suitable for deployment.

The Native Palm VII and OmniSky Experiences

To explore wireless email outside the Java context, we experimented with the native Palm.net client for the Palm VII and the OmniSky client for the Palm Vx.

First, the native clients were much faster. All the Java clients were painful in comparison. The delays could have been caused by Java overhead or by the extra security provided by the Java clients. In any event, the native clients were clearly preferable.

Second, the native clients fit the Palm UI paradigm and were easier to use. The Kjava clients in comparison looked very primitive and the MIDlet client looked clumsy (using display space poorly and being different for no good reason). The MIDlet use of screen real estate, in particular, looked stupidly wasteful because most fields were indented almost halfway across the screen, as if some desktop metaphor were being applied to the Palm. In contrast, the native Palm applications looked artfully designed because, well, they were.

Third, although the native clients were not as unreliable as the Java clients (due to timeouts and implementation problems), the overall services were not especially reliable because of connectivity problems. Connections would come and go, especially with the OmniSky service. Not surprisingly, wireless reliability is to wired reliability as mobile phones are to wired phones. In fact, wireless data may be worse than mobile phone reliability because humans are pretty good at understanding unclear speech, whereas mobile devices are not so error resistant.

Fourth, coverage is still limited. One of us took the Palm VII on a cross country trip, with mixed results. Much of the western half of the US was silent. Coverage, like connection reliability, is not yet certain.

Fifth, the Palm device is usable as an auxiliary email client, but could not serve as a primary client because of screen and input limitations. It is simply not practical to read long messages on the Palm, let alone process attachments. And composing messages to send is tedious because of graffiti. Email on the Palm is better than the encrypted instant messaging lifestyle on mobile phones, but, certainly in an enterprise context, it is a supplemental platform.

Building Java Applications for Small Platforms

In this section we discuss some of the implementation issues encountered while working on the projects described above.

THE CONSTRAINTS OF A SMALL JAVA VIRTUAL MACHINE

The memory limitations of a small Java virtual machine force developers to exercise great control over the allocation and de–allocation of objects on the heap. (Legally we are required to point out that "the term 'Java virtual machine' means a virtual machine for the Java platform." Noted.). The original KVM did not include memory compaction, which led to severe memory fragmentation. This in turn had an impact on the performance of the VM. Consequently, it was necessary for Java application code to include frequent garbage collection (GC) calls. Even with the better GC implementation in newer versions of the KVM (such as that used for the MIDP platform), developers need to be careful in the management of memory.

For example, we discovered that the time taken to create a new String from a byte array can be reduced by a factor of nearly seven. This is achieved by first converting the byte array to a char array (by simply casting each byte to a char in a loop) and then by using the String constructor that takes a char array argument. However, this mechanism is safe only when dealing with ASCII strings and must not be used for UTF encoded strings. (It is unclear if string related enhancements in newer KVMs nullify the benefits of this trick.)

When designing our clients and servers we made several judicious decisions to reduce the internal state required on the client. Whenever possible that state was maintained by the server, for example in a servlet.

The performance of the virtual machine is somewhat dependent on the performance of the underlying platform. The Palm devices on which the client applications described above were deployed only had a 20 Mhz processor. In order to overcome the performance constraints of the virtual machine we adopted a couple of solutions.

First, if an operation is slow, find ways of reusing its results multiple times. For example, public key operations required by SSL can be slow, so we recommend amortizing the cost of these operations across multiple user transactions. In particular:

- reuse the effort of parsing and verifying a certificate by caching that certificate (and marking it as already verified).
- reuse the effort of creating master keys by implementing SSL's session reuse option.
- reuse the same TCP connection for multiple HTTP request–response transactions by implementing persistent HTTP. Unfortunately, this feature is not enabled in the Sun.Net gateway. Enabling this feature could have a significant positive impact on user experience because it reduces the number of times SSL handshakes must be performed.

Second, instrument the code to identify the right bottleneck. For example, when working with CLDC (with Kjava widgets) on the Palm, we discovered performance enhancing opportunities in unexpected places by carefully instrumenting the code. The impact of changes listed below was far greater than the potential benefit of reimplementing SSL (the commonly–perceived bottleneck) in C.

We discovered two ways to reduce the time taken to display text in a ScrollTextBox. We created a new widget as a substitute for `com.sun.kjava.ScrollTextBox`, which included the following two techniques. These techniques made a difference of 6–8 seconds in the user's perceived response time for each transaction involving a full screen update:

- the widget uses a less accurate but faster algorithm for computing line breaks.

- the widget starts displaying text as soon as it has computed enough line breaks to cover the widget's display area, namely the first screen. The original widget computes all the line breaks before displaying any text and this can add a good amount of dead time even when the displayed text has as few as one to two thousand characters.

In addition, it is important for developers to remember that the functionality offered by a small J2ME Java virtual machine is significantly less than its J2SE™ platform counterpart. We have observed Java programmers using the CLDC platform the way they would program on a desktop. Programming Java on the Palm is more like real–time programming than desktop Java programming.

Development and Testing

Just as designing Java software for a small device is similar to designing an embedded system, so developing and testing that software is similar to developing and testing an embedded system with emulation and remote debugging.

More specifically, a Palm–based MIDP application can be run on three platforms: Windows (which hosts the basic reference implementation, or RI), POSE (the Palm emulator, commonly run on Windows), or the actual Palm device. The Windows RI is convenient to use but is completely removed from the Palm platform. The Palm emulator uses the Palm OS and emulates some aspects of the hardware, although significantly not its speed. The actual Palm device is what users care about. Debugging is progressively harder, but more realistic, as one moves down this platform chain. In general, apparent correct operation on one platform is no guarantee of correct operation on a more constrained one.

It is important to ensure consistency between developer and user environments. One cannot expect good wireless applications if the developers' environment is different from that of the users. It is surprising how often something so obvious is overlooked. Writing an application that performs well on a real mobile device with a real wireless network requires a different mindset than writing an application for an emulator. This is especially true if (as is often the case) the emulator runs much faster and/or does not accurately model real constraints. For example, the emulator may offer a much larger heap or may run on an operating system with more liberal resource constraints (for example, Palm OS restricts the number of open sockets to four, whereas J2ME emulators running on Solaris™ or Windows have a much higher limit).

Because the J2ME platform is so different from its J2SE cousin, specialized tools are required for development, testing and deployment. For example, the J2ME Wireless Toolkit is a set of tools that provides application developers with the emulation environment to develop J2ME applications. It can be tightly integrated with third party IDEs such as Sun™ ONE Studio, providing a complete development environment with which developers can write, test and debug applications from start to finish.

In addition, development and testing are made more difficult because of wireless connectivity and security issues. Creating the connectivity and security test environments involves considerable effort and complexity, and may itself have errors.

Reliability

It is very important for usability that a wireless client application recover gracefully from the multitude of connectivity, security, and platform resource limit errors that can occur, even in a simple application. Failure to handle such errors properly commonly results in hanging, death, or bizarre state changes that are annoying to the user. Furthermore, these conditions can combine or cascade; a well–designed client should be designed around error recovery.

While using the Omnisky's CDPD network, we found that wireless network connectivity was intermittent and that an application might receive transient I/O Exceptions at unexpected moments. Applications should be written to anticipate such exceptions and deal with them gracefully whenever an attempt is made to use the network. This graceful handling may be as simple as displaying "Network error, please retry".

Due to the behavior of wireless networks, a naively–written application that runs flawlessly on an emulator (with a LAN or landline connection) can fail in mysterious ways on a real device with a wireless connection.

Given the current poor geographical coverage for wireless access, it is important to recognize that, unlike landline connectivity, there is no guarantee that a wireless client device will be continuously connected to a server. Indeed, developers should expect that a client will be only be intermittently connected to a server. Consequently, the architectural division of responsibility between a PDA client and a server may be different than that found in a desktop–based client and its server. For example, it may be necessary to store or process data on the client prior to synchronization with a server, and it may be necessary to handle synchronization failures gracefully, so as to enable the user to correct errors and retry.

The User Experience

Designing the user interface for a PDA client involves extra considerations compared to a desktop client: display real estate, text input, and in–progress feedback.

It is obvious that display real estate is at a premium. Wasted space is both obvious and annoying (see, for example, Figure 27 on page 30). On the other hand, a crowded display is also unpleasant. Palm application developers have been balancing these concerns since the Palm was first released, and the issue continues to be important.

Text input on a keyboard–less device is always a somewhat vexed issue. It is more vexing when text must be entered under severe time constraints, as was the case with the Enigma challenge. If at all possible, an application should be designed to avoid text entry, and should use some other mechanism. The Enigma challenge alone was enough to doom any Palm–based email client.

Finally, wireless transactions are often slow and unreliable. Without feedback, users assume the transaction has failed. Give periodic feedback to alleviate perceived delay. While performing any operation that may take more than a couple of seconds, it is important to update the user. For example, we found that providing a commentary on the progress of SSL handshakes improves the user experience tremendously. In this case, we didn't decrease the delay; we just made it clear to the user what was happening and reduced the apparent delay.

User Interfaces for PDAs with Non–Trivial Screens

In early 2000, Sun Labs began a collaboration with Palm Computing to develop a prototype Java UI based on the native Palm widgets. This prototype was used as the starting point for developing a new PDA UI standard, the PDA Profile. Part of this work involved evaluating three Palm–based Java UI toolkits, including the prototype:

- PAWT (the Palm AWT): this UI toolkit provides a strict subset of the Java Abstract Windowing Toolkit (AWT). It uses the native widgets on the Palm device to provide a pure Java UI that nonetheless reproduces the Palm look and feel. Dan Podwall of Palm designed and implemented PAWT.
- kAWT (the kjava AWT): this UI toolkit is similar to PAWT in providing an AWT subset, but uses non–native widgets (the kjava widgets that were part of the original Spotless implementation). In this sense it is closer in philosophy to the Swing UI implementation [SWING], [kAWT].

- MIDP: the UI toolkit component of this API was designed in the first instance for devices with small screens, specifically mobile phones. It is significantly different to both the AWT and the native Palm UI. Although aimed at phones, a MIDP implementation for Palm OS is available [MIDPalm].

In order to evaluate the different UI toolkits, we produced a Java implementation of the Palm MemoPad application using the UI toolkits described above. Each implementation was evaluated using several evaluation criteria, including: reliability, speed, ease of implementation, similarity in look and feel to the Palm MemoPad, and completeness of features.

Tests were conducted using the sixth candidate alpha release (a1c6) of the PAWT, version.9951 of the kAWT, version 1.0 of Sun's KVM, and MIDP for Palm OS version 1.0 early access 3. Rather than use a real PDA, the tests were performed on a Pentium 200Mhz PC running the Palm OS Emulator (POSE) version 3.0a6e4 installed with Palm OS 3.5. Note that applications have been known to run much faster on POSE than on real devices (by at least a factor of two).

The remainder of this section compares the user interfaces presented by our three implementations of MemoPad (one for each UI toolkit) and concludes with some general remarks regarding their suitability for PDAs.

There are no great differences between the APIs of kAWT and PAWT, which is unsurprising, as they both aim to provide AWT functionality. However, it should be noted that the kAWT differs more from the AWT than PAWT, which provides a strict subset. This reflects the more experimental nature of kAWT.

MIDP, on the other hand, uses a completely different UI model. To be as portable as possible it provides a very abstract UI, and as a result applications have very little control over look and feel. Appearance (placement, shape, color, and font), navigation, and scrolling are all beyond an application's control, and are different from other Java UI toolkits. MIDP includes an alternate low level pixel–based UI for games, but that UI does not support standard widgets.

SIZE

The MemoPad implementations developed using the PAWT and kAWT only operate on Palm devices containing a minimum of 2MB RAM. In contrast, the original MemoPad application can be run on any existing device that uses Palm OS.

Sun Microsystems indicates that MIDP for Palm applications should be run on devices with at least 4MB of memory and warns that applications may not run properly on devices with less memory. However, on POSE, the application ran on a 2 MB device without noticeable problems.

The size of each implementation is shown in Table 1. The "Application size" indicates the size of the file that contains the implementation of the MemoPad application, that is, the compiled Java classes along with any other resources necessary for it run on a Java virtual machine. The "Java Platform Size" indicates the size of the file containing the virtual machine and any other resources necessary for the virtual machine to run on the device. (Due to architectural differences, separate

sizes are not available for the MIDP implementation.) The size of the original native Palm MemoPad application (using all the GUI functionality built into Palm OS) is 22.7K.

	PAWT	kAWT	MIDP	Native Application
Application size	51K	50K	36K	2K
KVM size	385K	459K	–	–
KVMUtil size	144K	145K	–	–
Java Platform size	529K	604K	538K	–
Total size	580K	654K	574K	2K

TABLE 1: Size of implementation

SPEED

Three aspects of the speed of each implementation were measured:

- the time taken to start the VM (as indicated by the time taken for the splash screen to disappear);
- the time taken for the MemoPad application to start (as indicated by the appearance of a list of memos); and
- the time taken to close the application.

Table 2 below identifies the time taken (in seconds) to complete the activities described above. (Figures are not available for the MIDP implementation.)

	PAWT	kAWT	Native Application
Time taken to start the VM	16	12	–
Time taken to start the application	19	50	Minimal
Time taken to close application	Minimal	Minimal	Minimal

TABLE 2: Time taken to complete activities (in seconds)

The kAWT implementation is written in Java and thus contains no native code, relying instead on the original Spotless UI. Therefore, a kAWT application should theoretically be slower than one written using PAWT. In general, however, there is no noticeable speed difference between the two versions of the application. There are two exceptions: when the application is first launched (as shown in Table 2), and when text is being entered (in both cases kAWT is much slower).

The MIDP implementation of the MemoPad application takes a fairly long time to start up, but it is not much slower than any other Java implementation on Palm. Once the program is running, however, things run very quickly. MIDP is faster than either PAWT or kAWT.

Look and Feel

The look and feel of the kAWT is the kjava–Spotless look and feel, not that of native Palm applications. No attempt was made in the Spotless implementation to mimic the Palm look and feel.

The PAWT look and feel replicates that of the Palm. Except for startup speed, which is quite noticeable, the PAWT implementation of MemoPad appears the same as the original Palm application.

The MIDP for Palm uses native Palm widgets, thus preserving some consistency with native Palm applications. However, the UI model is fundamentally different. As a result, the overall appearance of the MemoPad application is different. See below for details on how MIDP makes it hard to duplicate the look and feel of an original Palm application.

Specific kAWT and PAWT Issues

Some specific problems were observed when using kAWT:

- Menus are not automatically pulled down when the user activates the menu bar.
- Menus can be pulled down from unusual places (such as the area next to **New** button).
- Instead of adjusting the horizontal length of a menu to fit the labels of the menu items, menus in kAWT acquire scrollbars if the labels run out of horizontal space.
- Instead of automatically adding ellipses for entries that are too long, List widgets use a horizontal scrollbar.
- The arrow for Choice widget is slightly off–center (too close to the label and too high).
- The cursor in a TextArea widget does not blink.

Other specific problems were observed when using both kAWT and PAWT:

- If the size and position of a widget are specified in the Java source code, the two UI toolkits will not necessarily produce the same layout.
- Both implementations show the outline of Lists and TextArea widgets even though their native counterparts have none.
- Choice widgets (popup triggers) in both APIs cannot be aligned on their right–hand sides.
- Neither information buttons or dialog widgets are available in either API.

The kAWT is missing some features:

- Class TextArea does not have any of the SCROLLBAR_constants that are needed to determine how many and which scrollbars should be included in an instance of TextArea.
- There is no way of adding a separator to a menu as Menu.addSeparator() is missing and using a hyphen as the name of a menu item fails to work.
- An instance of Component can only have one listener for each type of event.
- There is only one adaptor class available.
- Font manipulation cannot be performed: attempting to use a bold font simply displaces text without making it bold, and not all the standard Palm fonts are readily accessible.
- Instances of TextArea do not wrap their text automatically.
- Selector triggers are not available.

In contrast, the PAWT is relatively complete, except that class TextField isn't available. It would be a desirable feature to have but is not absolutely essential.

A number of bugs were observed in kAWT:

- Clicking in the empty space to the right of a Choice widget may pop it up if its bounds are extended too much. This is undesirable since the user normally wouldn't expect anything to pop up when the stylus is tapped in an empty space.
- There may be a problem with TextArea.setText(). The application is unable to open any memos created using the native Palm MemoPad application, but it was able to do so in a previous version of kAWT.
- Attempting to change the category of a memo will generate an OutOfMemoryError.
- An attempt to classify a memo as *private* freezes the device. A soft reset is required. Note: Under some circumstances a dialog is supposed to pop up after a memo has been classified as *private*, but this is not happening (though it works in the Palm AWT implementation of the MemoPad).
- Attempting to edit categories from the **Edit Details** dialog box appears to have no effect. An attempt to edit categories from anywhere else in the application causes the application to freeze, but the device remains responsive to the four soft buttons next to the pen input area and the four buttons physically built into the bottom of the device.
- Dialog boxes that are supposed to be popped up from the menu bar do not appear.
- The menu option **Delete memo...** in menu bar does not work.
- Class WindowListener does not work.

Some bugs were also encountered in PAWT:

- There may be a problem with KeyListener since the application is not responsive to key strokes when it is supposed to be.
- The choice widget in the **Edit Details** dialog box refuses to allow user to select a different item.
- The application behaves strangely if user tries to declare a memo as *private* or to change the sorting order of the memo list.
- Instances of class java.awt.List are not updated. Newly–created categories do not appear right away. If the user attempts to delete every single category or every single memo, the last item to be deleted does not disappear properly.

General kAWT and PAWT Observations

Given the similarity of intent of kAWT and PAWT, they can easily be compared. (MIDP is discussed in greater detail below.)

- The code for the implementation of the MemoPad application is nearly identical for both UI toolkits; only relatively minor changes were needed.
- OKDialog and OKCancelDialog are useful tools available from the PAWT in com.palm.awt.Toolkit, but they allow only a single line of text and do not have the appropriate icons. The kAWT uses the de.awt.OptionDialog class, which is supposedly similar to the Swing class JOptionPane, as a substitute.
- The PAWT's com.palm.os package has no kAWT equivalent. The classes provided in this package are essential since they provide an easy way of accessing the native Palm OS APIs. For example, cut, copy, and paste would be very difficult, if not impossible, to implement in kAWT since the com.palm.os.Field class is not available. Note: the com.palm.os package is not complete.
- Both implementations are experimental, which led them to being unstable when executed on the POSE emulator. Some tasks (especially the simpler ones) function as predicted, but other tasks lead to unexplained behavior (non–response, objects appearing out of nowhere, etc.), bizarre error messages, or freezes. Unpredicted behavior exhibited

by one API usually cannot be found on the other API. Note: freezes occur with greater frequency when the applications were run on real devices.

MIDP OBSERVATIONS

Someone who is used to AWT will be disappointed by MIDP's capabilities. Many of these flaws stem from the fact that MIDP was originally designed for mobile phones with small displays. These flaws, other quirks, and good features are recorded below (unless noted otherwise, assume that all classes mentioned come from the javax.microedition.lcdui package or a standard java.awt or java.awt.event package):

- MIDP has no classes for standard UI widgets such as buttons, drop down menus, or checkboxes. Instead, the existing classes for GUI objects are more abstract. Two of the more useful classes are the Command class, which represents "any user interface construct that has semantics for activating a single action" [MIDP] and acts as a button; and the ChoiceGroup class, which can represent any group of selectable widgets, such as a dropdown menu, a checkbox (multiple mode), and a set of radio buttons, depending on the specific device being used. Unfortunately, PDAs use a wider variety of UI widgets than mobile phones, and MIDP doesn't seem equipped to use all of the native GUI objects.
- The MIDP API provides *Alerts*, simple dialog boxes that merely present a short message, an appropriate icon, and a button for dismissing the alert. They are very convenient for presenting simple messages (errors, warnings, confirmations, etc.) that require no action on the part of the user and would be a welcome addition to any other Java UI. Unfortunately, MIDP does not provide any dialog boxes that are more sophisticated than alerts. The MIDP expert group believed that a windowing UI is inappropriate for the devices targeted by MIDP, and consequently MIDP for Palm does not have any built–in equivalent of the typical AWT Dialogs that can be set to any size and can hold as many UI widgets as necessary.
- The only list–like widget provided by MIDP is an instance of class List, and each list takes up the entire screen. It is possible to add instances of class Command to a List, but it is not possible to add instances of class Item, or its subclass ChoiceGroup. In essence, a MIDP list widget cannot coexist with anything other than a button. In addition, MIDP lists don't look anything like their native Palm counterparts, and instead of automatically providing ellipses for entries that are too long, a MIDP list simply displays the rest of the entry on another line, as one can see from Figure 24. The

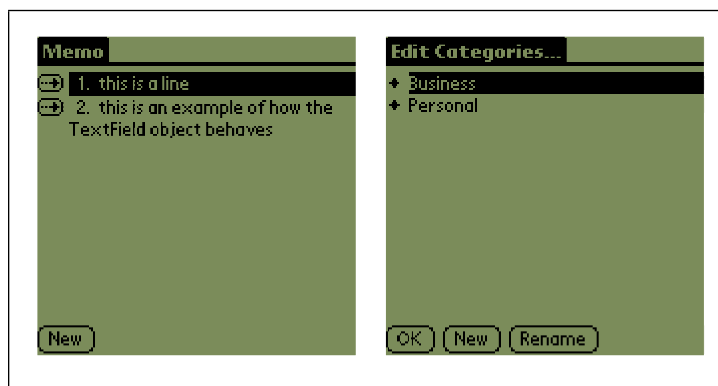

FIGURE 24. MIDP List widget

"exclusive–type" list does not appear like a native Palm list either. The drop–down menu of categories is missing because a ChoiceGroup widget cannot be added to a list widget. Also notice the appearance of this "implicit–type" list in Figure 24.

- A programmer has very little control over a MIDP application's menu bar. The Java Manager includes its own menu bar with any application it is running, and does not allow developers to create their own menu bars or modify the existing Java Manager menu bar. The Java Manager menu bar includes the Java Manager's own menu commands and is little use to the application being run (it is somewhat useful for a developer who is debugging an application, but almost useless to an ordinary user running the application). If there are any instances of Command (each instance is rendered as a button widget) associated with the visible screen, MIDP automatically places them on the menu bar; this is somewhat strange since all actions performed by the buttons can also be performed from the menu bar. The one neat feature of the Java Manager menu bar occurs when a TextField or TextBox is made visible on the screen: the Java Manager will automatically respond by providing an **Edit** menu with all the customary menu items (**copy**, **paste**, etc.), all of which function correctly (shown in Figure 25).

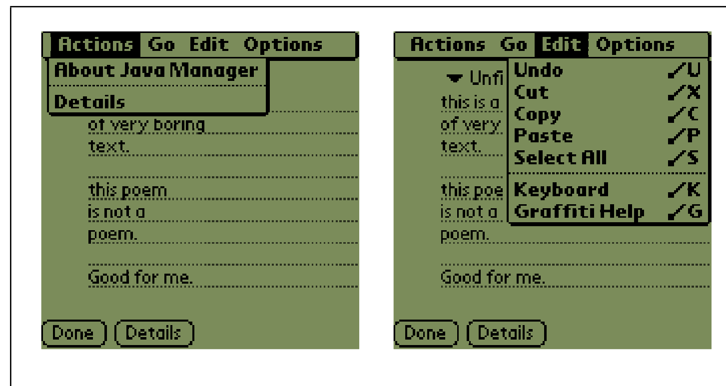

FIGURE 25. The Java Manager

- MIDP does not provide any way of controlling the layout of UI widgets. Layout is controlled by the MIDP implementation, which places buttons along the bottom of the screen and stacks all other objects vertically as they are added to the screen. This arrangement may be satisfactory for simple applications, but developers who wish to control the placement of their own widgets and/or use a more sophisticated layout arrangement will find no means of doing so. In addition, MIDP's overly simple layout arrangement may not make the best use of screen space (a lot of horizontal space may be left unused). MIDP might also choose to leave objects out of the screen, even if there is clearly enough screen space for the object. For example, in the leftmost screen shot of Figure 26, the **Delete** button (which is intended

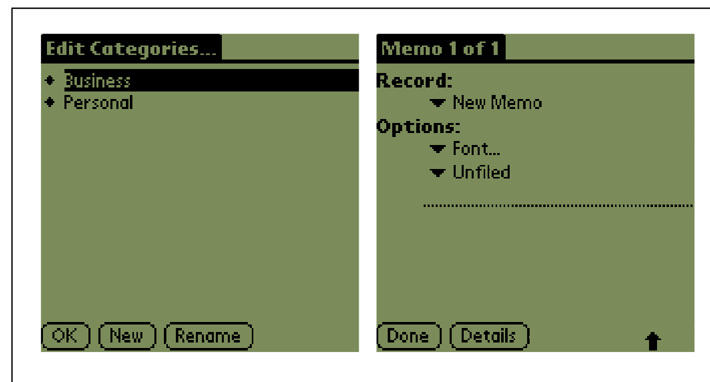

FIGURE 26. Control of UI Layout

to appear next to the **Rename** button) is not displayed; similarly the rightmost screen shot demonstrates that a menu bar cannot be created by using a row of instances of class ChoiceGroup, as there is no way to place those widgets horizontally.

- The label for a ChoiceGroup widget appears above and to the left of the widget. The label for a native Palm drop down menu would normally appear directly to the left of the widget.
- The ChoiceGroup class is missing several important functions, notably select(String), removeAll(String), and getSelectedItem(). It is not difficult for a developer to implement them, but it is inconvenient to do so.
- The TextField widget wraps text and scrolls properly (presumably the TextBox widget has similar features). Following a line break or period, the graffiti *shift* indicator (an arrow in the bottom right corner pointing upwards) is shown and the subsequent character is automatically capitalized; this is a feature of Palm OS that cannot yet be duplicated by any other Java UI. There are, however, some minor issues that do not affect the functionality of TextField widgets, but are aesthetically unpleasant. These include the fact that a TextField widget is placed at a peculiar location (the whole widget is indented) and that empty lines below the existing text are not shown. See Figure 27 for an example.

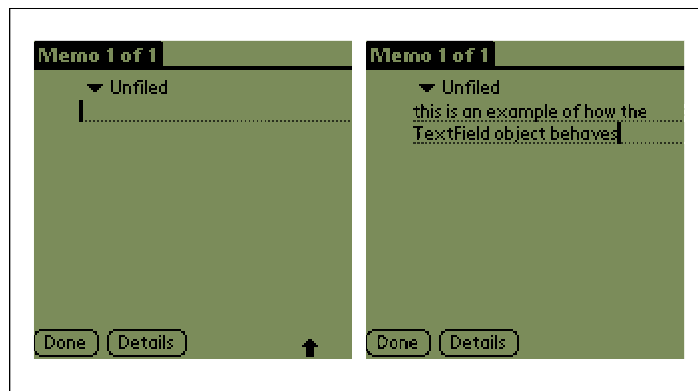

FIGURE 27. TextField widget

- There is no method available for setting the caret position for an instance of TextField or TextBox. (Note that a getCaretPosition() method does exist.)
- The MIDP event model is somewhat cleaner than the AWT event model that has been used since JDK™ 1.1. Instead of passing instances of EventObject (in practice, these are instances of class ActionEvent, ItemEvent, etc.) to listeners, MIDP passes the object that initiated that event; this makes for much neater code than in other Java UIs, where one would have to invoke multiple methods and perform several casts before one could do anything useful with an EventObject. MIDP also lacks an addXxxListener() method for objects, so the use of inner classes, which are visually messy for developers, is limited. Instead, it provides a setXxxListener() method for instances of Displayable, so that every instance of Display, Screen, Form, etc. can have at most one of each listener class (there are currently only two listener classes) and that listener will handle all events generated by the objects on the device's screen, a mechanism that can help programmers write better organized code.
- The two existing listener classes, ChoiceListener and ItemStateListener, may not be enough for a handheld device. Unlike a mobile phone, a PDA has a stylus with which the user can tap and write; in addition a PDA can also receive input from the on–screen virtual keyboard or even a physical keyboard. Any application that needs to take advantage of these features may require more listeners. The existing alternative is to use low–level event handling, but this is not very helpful; developers will be forced to use an instance of the less powerful Canvas class rather than class Screen, and since MIDP only provides constants for mobile phone keys (numbers, pound sign, fire button, etc.) many of a PDA's keys aren't mapped to key codes.

- An instance of class Displayable (the screen for a MIDP application) can perform low–level event handling *or* high–level event handling, but not both at the same time. One result is that font management is impossible for instances of class Screen.
- Due to the fundamental philosophical differences between MIDP and AWT, it is difficult to translate AWT–like code to MIDP (or vice versa).

There were also some problems with the non–UI functionality in MIDP:

- The MIDP record management system nicely provides meaningful exceptions that the Kjava Database class does not, but MIDP was unable to read or write the native MemoPad database.
- Calls to System.out.print*() do not produce on–screen output. Output is instead directed to a file, which can be viewed via a command from the Java Manager menu bar. This can be somewhat inconvenient for users (such as developers) who constantly need to look at the output. In addition, a lack of output may cause a user to believe that nothing is happening during the application's long start–up time.

UI Conclusions

Both AWT–based UI toolkits currently have numerous bugs, poor documentation, and similar levels of (in)stability. Improvements to kAWT in each successive version have been impressive, but at this time the PAWT is slightly better because it runs faster, takes up slightly less space, has more functionality, and can achieve an authentic look and feel.

MIDP for Palm is a relatively small–sized Java implementation for the Palm, runs at a tolerable speed, should not present users with any memory issues, and is well documented. Unfortunately, although it may be satisfactory for mobile phones and should allow Java programs originally written for mobile phones to run on PDAs, it does not take full advantage of a PDA's capabilities. For developers hoping to design applications specifically catering to PDAs, MIDP for Palm by itself falls short.

Extensible Access to OS Functionality

The Spotless platform had a very simple API library, restricted to basic graphics and a few native functions. In order to expand that library and make the rich set of Palm OS functions available to Java developers, we developed a prototype mechanism that provides extensible access to Palm OS. The kind of access that this prototype provides includes:

- access to the databases that underpin the PIM applications on the Palm, such as the calendar and the address book;
- serial connectivity not covered by the general connection framework of the J2ME platform, and
- other device–specific features such as screen brightness and audio volume.

The prototype was adapted from a proof–of–concept originally implemented for the J2SE platform (which was considerably more complex).

Access functionality is represented by a single Java class, OSAccess, which in turn relies upon a small number of primitives written in C and invoked as Java native functions. Static methods in OSAccess provide two broad types of functionality:

- "peek/poke" capability, allowing Strings and data of primitive numerical type (byte, short, int, long, float and double) to be stored at and/or retrieved from locations in memory; and

- the ability to invoke arbitrary Palm OS functions (implemented in Palm OS as system traps) with zero or more arguments.

The original J2SE OS access package consisted of fairly elaborate machinery mimicking C's type system and data manipulation facilities (including unsigned and pointer arithmetic) in Java. Resource limitations imposed by the Palm platform precluded a similar implementation. Instead, most C data types (including pointers and the primitive numerical types mentioned above, but excepting Strings) are represented in Java as ints. This places the burden of ensuring behavior congruent with C on the package user — for example, C pointer arithmetic is unsigned, whereas in the Java language, arithmetic on ints is signed.

The details of the peek/poke capabilities are fairly straightforward. Simple setByte and getByte methods are used. For example, to set the contents of location 0xFA to the byte value 2, the application invokes the following:

```
OSAccess.setByte(0xFA, 2);
```

Immediately thereafter the following yields the result 2:

```
OSAccess.getByte(0xFA);
```

Similar methods are provided for ints, floats, etc. All such methods are implemented as native Java methods, with fairly trivial C counterparts. For example, the C code corresponding to the setByte() method is the following:

```
void setByte()
{
    char *p = (char *)popStack();    // Retrieve the address from the top of the Java stack
    *p = (char) popStack();          // Store the next argument at the specified address
}
```

Palm OS system calls are invoked using the syscall method, which takes as its argument the number of the system call to be invoked. For example, number 0xA043 (referred to symbolically as *sysTrapDmNumDatabases* in the file systraps.h) returns the number of databases stored on a device. The following Java code makes this sysTrapDmNumDatabases call:

```
// get the total number of databases (0 is the card number)
// sysTrapDmNumDatabases = xA000 + 69 (see systraps.h)
int dbTotal = OSAccess.syscall(0xA043, 0);

// display the total
displayTotal(dbTotal);
```

Inside the OSAccess class, the method syscall(int trapid, int arg1) is merely a façade for a more general invocation method (similar convenience methods are provided for the cases of zero up to eight arguments):

```
public static syscall(int trapid, int arg1)
{
    int[] args = {arg1};
    syscall (trapid, args);
}
```

The general syscall(int, int[]) is a native method that relies upon a short machine-code sequence allocated and initialized in the C heap, illustrated in the following C code:

```
typedef struct TrapCall
{
    short i1;           // lea -args(a7), a7; -- skip past arguments
    short args1;        // -<argument count>
    short i2;           // trap #0xF; -- perform the syscall trap
    short id;           // <syscall id>
    short i3;           // lea args(a7), a7; -- pop the arguments
    short args2;        // <argument count>
    short i4;           // rts
} *TrapCallP;

…
    // Set up trap for outcalls
    TrapCall = (TrapCallP) heapStart;
    TrapCall->i1 = 0x4FEF;      // lea -args(a7),a7; -- skip past arguments
…
```

Three slots in the trap–call structure are modified for a particular trap call: two are fields of instructions that adjust the C stack pointer prior to and after a trap, and the third identifies the particular trap to be called. The trap call sequence is intended to be called as a zero–argument C function, with the arguments to the trap pushed onto the stack below the return address (the C stack grows from high memory addresses to lower ones). The trap–call sequence adjusts the stack pointer (since the trap call is presented as a zero–argument function to the C compiler, on entry to the call, the stack pointer is above the trap arguments), invokes the trap (placing the result in the D0 register), discards the arguments, and returns.

The native function C implementation of the Java method syscall(int, int[]) deals mainly with moving arguments onto the C stack in preparation for a call to the (suitably–modified) trap–call sequence:

```
void C_syscalln()
{
    ARRAY args = (ARRAY) topStack;  // Retrieve the Java array comprising the arguments
    unsigned trapid = sp[-1];       // Also the id of the system trap to be invoked
    int nargs = args->length;       // The number of arguments to the trap
    unsigned *argp = (unsigned *) (&args->iData) + nargs;
                                    // Compute a pointer to the last trap argument
    unsigned *csp;                  // Pointer to the top of the C stack
    unsigned i, rslt;

    // Patch the trap id and the size of the argument frame into the volatile trap sequence
    TrapCall->id = trapid;
    TrapCall->args1 = -(4 * nargs);
    TrapCall->args2 = 4 * nargs;

    // Locate the start of the next C stack frame
    csp = frameStart();

    // Arguments are passed from Java in first-to-last fashion, so shuffle them onto the stack
    // for the trap. Note that argp actually points to the last of the Java arguments.
    for (i = 0; i < nargs; i++) *--sp = *argp--;

    // Call the trap-call sequence as a regular zero-argument function. This will insert the return
    // address in the slot below the arguments. TrapCall will increment the stack pointer so as to
    // embrace the arguments pushed above and then execute the trap.
    rslt = ((unsigned (*)())TrapCall)();

    // Clean up the Java stack and return the result of the trap to Java.
    popStack();
    topStack = rslt;
}
```

To gain access to the C stack, the above code makes use of a brief assembly procedure:

```
// Return the address of the next frame on the C stack
asm static unsigned *frameStart()
{
    // An “asm” function is written in M68000 assembler and has no C function prolog
    unsigned frameptr;
                    // Allow a slot to store the frame pointer (cf. M68000 “LINK/UNLK” instructions)
    movea.l a7,a0;
                    // Move the value of the stack pointer into the result register
    rts
}
```

ISSUES

There are a number of respects in which the OS access package might be enhanced:

- Some minimal support for C data types (providing pointer arithmetic, for example) could be provided.
- The implementation is platform–specific, though every attempt has been made to minimize and isolate the platform dependencies. An attempt to reimplement it for another PDA platform (such as PocketPC) would help characterize the platform dependencies more clearly.
- The ability to synthesize C functions from Java methods would allow Java application programmers the opportunity to handle Palm OS callbacks; a preliminary implementation of such a facility (along the lines of the C Programming Object Kit of CINCOM’s VisualWorks Smalltalk system) has been completed.

Conclusions

SUMMARY

Secure corporate wireless email on the Palm is not yet a killer application. It can be useful at times, but is not yet a replacement for desktop email. This is so because:

- Wireless is not yet reliable (data connections fail with some frequency). This is exacerbated in Java applications by their slowness, which can cause timeouts.
- Wireless coverage is still limited.
- Authentication can be quite clumsy (especially with Sun's Enigma challenge and Palm's graffiti).
- The Palm's small screen limits its utility as an email reader.

The Java platform on the Palm is still maturing. The Palm itself has been carefully engineered to support small native applications, with a distinctive UI tuned for its display. In contrast to such applications:

- Java applications are slow, at least on start up.
- Applications based on MIDP, the only official Java UI now available on the Palm, do not have the Palm look and feel, are somewhat clumsy, and offer no compensating benefits.
- Java applications are not strikingly easier to develop than native C applications. The Palm is not an easy platform to program for in C, but the Java platform presents its own challenges. In fact, programmers familiar with J2SE will have to design their applications differently, using different libraries.
- Without special support, Java applications cannot take advantage of the functionality of the Palm OS platform, putting them at a disadvantage.

Nonetheless, recent developments make the platform more attractive. We hope to reevalute it when new versions of the Palm supporting the PDA Profile become available.

Applications based on Internet standards are easier to deploy than those based on proprietary technology. In particular, SSL proved to be much easier to use as an end–to–end security protocol than a proprietary one: implementing SSL on the Palm proved tractable, whereas the proprietary protocol required more specialized technology and more administrative overhead. The SSL implementation also challenges the more common but less secure proxy–based architectures.

Building wireless Java applications on the Palm is not like desktop Java programming, but more like embedded programming. Issues of machine resources, screen real estate, debugging, and wireless connectivity all require attention.

Note that the above remarks apply specifically to experience with the Palm platform of 2000–2001, centering on the Palm Vx and Palm VIIx, and thus are somewhat specific to that platform. Nonetheless, the observations involving PDA UIs, wireless clients, and programming for small devices are generally applicable. PDA devices, although increasingly powerful, are still largely defined by their display size. Wireless connectivity continues to develop but still is hampered by the problems encountered in this work.

THE EVOLVING JAVA PLATFORM FOR PDA DEVICES

In 1998, when the original Spotless work was done, "PDA" meant "Palm Pilot", which in turn meant a cluster of computing, connectivity, and UI characteristics. PDAs now are more diverse, and much more capable. As PDAs have evolved, so have the Java implementations for them.

The PDA Profile

The PDA Profile, Java Standardization Request 75 [PDAP], was undertaken to address some of the PDA UI issues discussed above. It began as a collaboration between Palm and Sun (Dan Podwall of Palm wrote the prototype implementation, the PAWT; one of this report's authors, Bill Bush, drafted the JSR and recruited the original expert group). The original goal of the profile was to define a Java UI appropriate for PDAs so that native PDA programmers and users would see a familiar look and feel, while Java programmers would develop to familiar APIs.

The profile defines a PDA as battery–powered device with a pointing device and a screen size between 128x128 and 240x320 pixels. At least 1 MB of memory (ROM plus RAM) must be available for the Java runtime and libraries; heap space can range from 64 KB to 64 MB. The PDAP UI is based on the AWT UI defined for the Personal Profile, removing: buffered images, alpha composite image manipulation, object cloning, serialization, data transfer APIs, and the J2SE security model. The PDAP also defines a PIM API, which includes access to an address book, a calendar, and a to do list, as well as general database access. Additionally, the profile defines optional serial port connectivity and file system connectivity for memory cards (such as compact flash cards and secure digital cards). Security is effected through control of: access to the AWT system event queue, listening to AWT events, reading and writing PIM data, and reading and writing data using the serial port and file system. Note that MIDP is a required base for the PDAP, meaning that MIDlets will run transparently on the PDAP.

Palm has led the work on the profile. The Java Community Process℠ public review stage for the profile was recently completed. Palm plans to release both a reference implementation and a commercial implementation for the Palm next year. The commercial Palm implementation is a joint effort of Palm and Insignia, and will be bundled with Palm OS.

Java Implementations

A wide variety of Java implementations are now available for PDA–class devices. Here is a partial list (in addition to Sun's offerings):

- Aplix: JBlend (http://www.aplixcorp.com/products/jblend.html)
  platforms: WinCE, ARM, consumer devices
  Java standards implemented: PersonalJava™, CDC, CLDC, MIDP
- Esmertec: Jbed Micro Edition CLDC, Jbed Profile for MID (http://www.esmertec.com/p_jbed_cldc_long.html)
  platforms: Palm OS, Linux, Nucleus; bare metal ARM, 68K, PowerPC, Coldfire
  Java standards implemented: CLDC, MIDP
- HP: Chai (http://www.hp.com/products1/embedded/)
  platform: Pocket PC
  Java standard implemented: MIDP
- IBM: WebSphere Studio Device Developer (J9) (http://www.embedded.oti.com/wdd/)
  platforms: Palm OS, Pocket PC (plus desktop OSes, real–time OSes)
  Java standards implemented: CDC, CLDC + MIDP (depending on the platform)
- Insignia: Jeode PDA VM (http://www.insignia.com/content/products/pda.shtml)
  platforms: WinCE, Linux
  Java standard implemented: PersonalJava 1.2

- Kada Systems: Kada Mobile Platform (http://www.kadasystems.com/resources/j2me.jsp)
  platforms: Palm OS, WinCE
  Java standards implemented: CDC + Foundation, CLDC + MIDP
- SavaJe: SavaJe OS (http://www.savaje.com/products/index.html)
  platform: bare metal ARM
  Java standards implemented: J2SE, PersonalJava, MIDP
- Symbian: Symbian OS 6.0 (http://www.symbian.com/technology/symbos-v6x-det.html)
  platform: bare metal ARM
  Java standards implemented: PersonalJava, CLDC + MIDP

This list demonstrates that a definitive PDA platform has yet to emerge, and that Java vendors are quite active in producing alternate implementations. It remains to be seen which device will dominate, and which associated Java standard (J2SE, PersonalJava, CDC, or CLDC) will prevail. This uncertainty is bad for developers and organizations deploying devices, because they must hedge their bets. On the other hand, there is a rich collection of alternatives to investigate and choose from, a collection that will doubtless evolve along with platforms and standards. Adding to the uncertainty is the appearance of Microsoft's .NET for the Pocket PC, which will undoubtedly appeal to enterprise customers familiar with Windows. As .NET matures, Java standards and implementations will have to do likewise to stay competitive.

The Future of Wireless Enterprise Applications

The problems we encountered deploying wireless applications are apparently not atypical, and are certainly not limited to Java–based applications. Various issues have been identified (see [Wireless1], [Wireless2], and [Wireless3]):

- Wireless deployments often involve several different vendors, potentially including: the maker of the client device, the maker of the client software platform, the writer of the client application, the wireless service provider, the writer of the server application, the provider of the security infrastructure, and the provider of the client application provisioning infrastructure. Coordinating the efforts of these vendors, while the technology is still being developed, is challenging.
- Problems with security, including establishing user and device identity, and guaranteeing end–to–end security, hinder deployments. Solutions are still custom–designed to specific deployment environments.
- Application management — application provisioning, device synchronization, and fault detection — is crucial to the success of large scale deployments, but remains largely ad hoc.
- Applications must be reengineered for wireless deployments. PDAs are not the desktop platforms that most mulitier applications were designed for. User interfaces must be redesigned, and applications must be refactored, moving computation from the less powerful PDA client to the server.
- Wireless coverage is simply not reliable or widespread enough.
- It is difficult to make the return–on–investment case needed for large scale deployments. The benefits of such deployments remain unclear and are hard to quantify, and costs are volatile. Corporations are wary of the wireless carriers' orientation toward consumers and their per-byte revenue models.

Some solutions to these problems are being developed.

- Corporations are being introduced to wireless clients via email deployments, which are relatively simple and generic. The BlackBerry device is being used as the client for corporate email service (using Outlook) hosted by both RIM and Cingular [Wireless4]. Additionally, both Cingular and Sprint PCS provide a tunnelling service from the client to the user's desktop, which can be done without the participation of the user's company [Wireless5] (this is typical of the PDA space, where individual users often initiate PDA use).

- Consultancy organizations are starting to provide wireless solution services. EDS is working with Sun and Nextel to provide custom mobile portal solutions [Wireless6]. HP is providing wireless solutions with test periods, so that their clients can evaluate wireless deployments without making major up front commitments [Wireless7]. IBM Global Services is expanding its efforts in this area [Wireless8].
- Partnerships between vendors are being formed to provide solutions. Microsoft and AT&T Wireless have agreed to cooperate on wireless enterprise client applications based on the Pocket PC [Wireless9]. The companies will start by offering email. Palm is working with IBM and BEA to provide back end support for Palm-based applications [Wireless10]. And as noted above, Sun and Nextel are working together along with EDS.
- CRM is becoming a focus of wireless enterprise client experimentation [Wireless11]. In particular, General Motors plans to provide devices to all its dealers with applications for tasks such as inventory tracking. This effort, however, is still in the bid stage, although it indicates a major commitment.
- Platform makers are improving developer support for building mobile applications [Wireless12]. Palm and Microsoft in particular are starting to provide frameworks and components for mobile enterprise applications, making it easier to integrate such applications with enterprise data.
- Service providers are attempting to improve wireless coverage by seamlessly linking 3G (WLAN) and 802.11x (LAN) networks, so that users can move transparently between these heterogeneous networks [Wireless13]. Both Lucent and IBM are developing these services.
- A solution that has recently appeared is the use of VPN clients on small devices, which is a higher level version of the SSL solution. See [Wireless14] for a list of PDA–based VPNs.
- Another solution, being tried at Sun with its “Edge” computing initiative, is to move some less sensitive services into the DMZ (making it rather thick). These services, specifically email, calendar, and LDAP, are not accessed with Enigma authentication, but simply via name and password.
- New 3G services aimed at the enterprise are being packaged as solutions [Wireless15]. Sprint’s new 3G PCS service includes access to corporate email, calendars, and contact information (which can be done via hosts behind the corporate firewall). Sprint is also teaming with system integrators (Accenture, IBM, HP, PwC Consulting, and Ingram Micro) to provide customized solutions for large customers, and is providing client hardware (a a Sprint–branded Handspring Treo).

Significantly, programming languages and their platforms are not the gating factors to large scale wireless deployment. Advancing the Java platform will not lead to massive wireless use in the enterprise.

To close on a humorous note, a recent article in *Business Week* (3 June 2002, page 51) on 3G in Europe described the current wireless experience in the context of 3G and mobile phones. Unlike standard mobile phones, with which users bring their own content (namely talking), 3G phones require compelling digital content, which has failed to appear: “In March, 2000, at the massive Cebit technology fair in Hanover, Germany, Alcatel officials displayed a traffic map of Paris that looked, on the tiny, black–and–white screen of a sample cell phone, like a pulsating amoeba.” One of the main anticipated uses of wireless is entertainment, in which category pulsating amoebas clearly fall.

References

[Spotless] *The Spotless System: Implementing a Java System for the Palm Connected Organizer*; Antero Taivalsaari, Bill Bush, Doug Simon; Sun Microsystems Laboratories Technical Report TR–99–73, February 1999. http://research.sun.com/research/techrep/1999/abstract-73.html

[P-Spot1] *Automatic Persistent Memory Management for the Spotless Virtual Machine on the Palm Connected Organizer*; Bernd Mathiske and Daniel Schneider; Sun Microsystems Laboratories Technical Report TR–2000–89, June 2000. http://research.sun.com/research/techrep/2000/abstract-89.html

[P–Spot2] *Automatic Persistent Memory Management for the Spotless Virtual Machine on the Palm Connected Organizer*; Daniel Schneider, Bernd Mathiske, Matthias Ernst, and Matthew Seidl; Java Virtual Machine Research and Technology Symposium, April 23–24, 2001. http://rwww.usenix.org/events/jvm01/schneider.html

[KSSL] *KSSL: Experiments in Wireless Internet Security*; Vipul Gupta and Sumit Gupta; Sun Microsystems Laboratories Technical Report TR–2001–103, November 2001. http://research.sun.com/research/techrep/2001/abstract-103.html

[KVM] http://java.sun.com/j2me/

[MIDP] http://java.sun.com/products/midp/

[CLDC] http://java.sun.com/products/cldc/

[SWING] http://developer.java.sun.com/Books/GJ22swing/

[kAWT] http://www.kawt.de

[MIDPalm] http://java.sun.com/products/midp4palm/index.html

[PDAP] http://www.jcp.org/jsr/detail/75.jsp

[Wireless1] “Small size, big demands”; Ephraim Schwartz; *InfoWorld*, 22 April 2002, pp. 51–52.

[Wireless2] “Wirelessly challenged enterprise apps”; Maggie Biggs; *InfoWorld*, 22 April 2002, p. 52.

[Wireless3] “Wireless users in the enterprise face hard choice”; Carmen Nobel; *eWeek*, 21 October 2002, p. 1.

[Wireless4] “Wireless carriers expand their BlackBerry services”; Carmen Nobel; *eWeek*, 20 May 2002.

[Wireless5] “Wireless carriers exploit firewall bypass”; Ephraim Schwartz; *InfoWorld*, 25 January 2002.

[Wireless6] “Nextel, EDS, Sun Offer Secure Wireless Solution to Extend Sun ONE Portal Capability to Nextel Java Technology-Enabled Devices”; *Yahoo Financial News*, 19 November 2002. http://biz.yahoo.com/bw/021119/192352_1.html

[Wireless7] “Information and commerce for a mobile world”; COMPAQ. http://nonstop.compaq.com/view.asp?IO=MOBLSOBR

[Wireless8] “Apps seek wireless route”; Ephraim Schwartz; *InfoWorld*, 5 August 2002, pp. 17–18.

[Wireless9] “Wireless move could fall short”; Carmen Nobel; *eWeek*, 5 August 2002, p. 14.

[Wireless10] “Palm reaches for enterprise”; Ephraim Schwartz; *InfoWorld*, 18 November 2002, p. 35-36.

[Wireless11] “Un–wired to the customer”; Jack McCarthy; *InfoWorld*, 29 April 2002, pp. 43–44.

[Wireless12] “Tuned up for Web services”; Ephraim Schwartz; *InfoWorld*, 24 June 2002, pp. 19–20.

[Wireless13] “Lucent, IBM services link WLANs, private nets”; Carmen Nobel; *eWeek*, 22 July 2002, p. 11.

[Wireless14] “VPNs trickle down to handheld computing”; *eWeek*, 29 July 2002, p. 37.

[Wireless15] “Backers of Sprint’s 3G net ready with devices, services”; Carmen Nobel; *eWeek*, 12 August 2002, pp. 9–10.

Acknowledgements

We would like to thank Dan Podwall and Philip Shoemaker of Palm for their efforts on the PAWT prototype, Julian Kupiec of iPlanet and Tim Dierks of Certicom for their work on the Palm VII prototype, Mario Wolczko of Sun Labs for his work on the MemoPad applications, Sumit Gupta of Sun Labs for his work on KSSL, Raj Singh and Jitender Aswani of Bonita Software for their work on the Bonita clients and servers, and Graham Poor and Margaret Mahoney of Bonita Software for their tireless dedication to small client computing.

Appendix: Palm VII End–To–End Security Protocol

INTRODUCTION

The protocol presented here provides end–to–end encryption and security for enterprise client applications running on Palm VII devices using their built–in Mobitex connections. Simple data encryption is not adequate. A session must be established and maintained between authenticated clients and servers in order to avoid such vulnerabilities as spoofing and man–in–the–middle attacks.

Messages are encrypted on the client (using elliptic curve cryptography), then transferred using the standard INet APIs on the Palm VII to Palm.net. Palm.net then converts these messages into Internet transactions, which terminate at a security server. This server decrypts the messages and passes them on to an application server. Optionally, the security server can send an encrypted response back to the client.

Each client has a pre–arranged public key for its security server. (To handle key freshening, the server can send down a new public key, signed with a key distribution key. Simple key formats or real certificates can be used.)

PROTOCOL OVERVIEW

The following definitions apply in this section:

E{K,M} denotes encryption with key **K** of message **M**.

D{K,M} denotes decryption of message **M** with key **K**.

MAC{K,M} denotes the messages authentication code with key **K** of message **M**.

| denotes concatenation.

ECDH{priv, pub} denotes an elliptic–curve Diffie–Hellman key agreement using private key **priv** and public key **pub**.

KD{V} denotes a key derivation algorithm which derives a key from a value **V**.

The server has a well–known elliptic curve public key, KS_{pub}. The private half of this key is KS_{priv}. Each message has a payload, P, composed of the message and other information (detailed below).

Each payload is encrypted for transmission to the server. To encrypt a payload, the client first generates an ephemeral elliptic curve key pair, KE_{pub} and KE_{priv}. It then calculates $K_m = ECDH\{KE_{priv}, KS_{pub}\}$. It then derives an encryption key $K_{me} = KD\{Km\}$. It encrypts the payload P into ciphertext C with $C = E\{K_{me}, P\}$ then sends:

KEpub | C

To decrypt this message the server calculates:

$K_m = ECDH\{KS_{priv}, KE_{pub}\}$

$K_{me} = KD\{K_m\}$

$P = D\{K_{me},C\}$

(K_m and K_{me} will be identical to the values of the same names generated on the client).

The server optionally creates a response to the client by formatting the response message into a payload P' of the same format as the payload received from the client, then calculating

$$K_{me} = KD\{K_m\}$$

$$C' = E\{K_{me}, P'\}$$

If the server sends a response to a message from the client, the client calculates

$$K_{me} = KD\{K_m\}$$

$$P' = D\{K_{me}, C'\}$$

and, after validating the payload format, retrieves the response message from P'.

Note that, if a response payload is sent from server to client, both client and server need to cache K_m until the response is processed.

Note: The key used to encrypt responses from server to client may be different than the K_{me} value used to encrypt payloads sent from client to server.

Payloads are formatted as follows. Each payload is composed of two parts: the payload body data (P_d), then a MAC on the payload body data. The key for the MAC varies with the payload type.

$$P = P_d \mid MAC\{K_{MAC}, P_d\}$$

A payload body data is composed of two components: first, an encoded payload record type (P_t), for which there are three possible values: 0, 1, and 2. A payload record follows, whose format depends on the payload record type.

A type 0 payload record has the following components:

- A message, M (this is the actual data being sent)

For a type 0 payload record, K_{MAC} is derived as follows:

$$K_{MAC} = KD\{Km \mid \text{"m"}\}$$

A type 1 payload record has the following components:

- A session ID, ID_{sess}
- A session key, K_{sess}
- A message, M

For a type 1 payload record, K_{MAC} is derived as follows:

$$K_{MAC} = KD\{K_m \mid \text{"m"}\}$$

A type 2 payload record has the following components:

- A session ID, ID_{sess}
- A record counter, N_{sess}
- A message, M

For a type 2 payload record, K_{MAC} is derived as follows:

$$K_{MAC} = KD\{Km \mid Ksess \mid "m"\}$$

where Ksess is the session key previously associated with the session ID IDsess with a previous type 1 payload record.

Thus, the legal payload variants are (in each of the following, the contents of P_d are enclosed in parentheses):

$$(0 \mid M) \mid MAC\{KD\{Km \mid "m"\},P_d\}$$

$$(1 \mid IDsess \mid Ksess \mid M) \mid MAC\{KD\{Km \mid "m"\},P_d\}$$

$$(2 \mid IDsess \mid Nsess \mid M) \mid MAC\{KD\{Km \mid Ksess \mid "m"\},P_d\}$$

All type 0 records are anonymous. (As such the number of type 0 records sent to establish a session should be minimized.) A type 1 record instantiates a session; when a type 1 record is received by the server, a session record is created in a database. The session record must store the session ID, the session key, and the current session record counter value. The record counter is initialized to 0. A type 2 record is associated with an existing session and requires that session's key to be authenticated. Each type 2 record which is received must have a session counter which is greater than the current value; when a type 2 record is received, its counter value is placed into the session record.

To link this to the logon protocol:

- The messages sent during logon handshaking are sent anonymously.
- The message in which the user is authenticated is sent as a type 1 message; the server must link this user to the session record created by this message.
- All subsequent authenticated messages are sent as type 2; the server must ensure that the userID encoded into the message matches the user for whom the session record was originally created.

About the Authors

Bill Bush is a senior staff engineer at Sun Microsystems Laboratories, where he has worked on various implementations of the Java programming language. His research interests include programming language design and implementation, program analysis, computer–aided design, computer architecture, software engineering, and organizational behavior, in which areas he has published papers. He has held various research positions at Harvard and UC Berkeley, where he received his Ph.D. He cofounded a venture–funded company that developed new program analysis techniques and that was acquired by Microsoft. See http://research.sun.com/people/wrb.

Bernard Horan is a senior staff engineer at Sun Microsystems Laboratories, where he has worked on interactive collaborative environments and Java introspection tools. He is currently working on projects that explore the use of metadata and ontologies. Bernard represents Sun in the W3C Web Ontology Working Group.

Vipul Gupta is a senior staff engineer at Sun Microsystems Laboratories, where his research interests include secure networking protocols and mobile computing. He has authored or co–authored over thirty technical publications in these and related areas. Prior to joining Sun he was an Assistant Professor at the State University of New York at Binghamton, where he taught courses in computer networking, parallel processing, and operating systems, and conducted research funded by the National Science Foundation and industry sponsors that included IBM and NEC. He has a Ph.D. in Computer Science from Rutgers University.

Phillip Yelland is a senior staff engineer at Sun Microsystems Laboratories. He has an M.A. and Ph.D. in Computer Science from the University of Cambridge in England, an M.B.A. from the University of California at Berkeley, and has worked in a wide variety of settings, ranging from academic research to software development management in high–tech startups. Throughout, his activities have reflected a preoccupation with the application of theoretical knowledge to the practical conduct of business. Currently, his research centers on the use of statistical techniques for supply–chain management.

Patrick Chi is an undergraduate at UC Berkeley. He evaluated the Palm–based Java user interfaces while a high school intern at Sun Labs.