- *Power management*. Palm OS has different operation modes (sleep, doze, running) to save power.
- *OS size*. Palm OS 3.5 is about 1.4 MB (including built-in applications) in size and needs up to 64 kb (including TCP/IP) runtime memory.
- *User interface*. The Palm application starter can be seen on the left side of Figure 3.20, and an example of a Palm application's look-and-feel is shown on the right side of Figure 3.20, with the built-in date book. The major user interface design principles of the Palm OS are: recognize only the Palm handwriting alphabet; one button access to applications; and minimize taps for most-used operations.
- *Memory management*. To keep the operating system small and fast, Palm OS does not separate applications from each other. This has implications for the system stability and security. If one application crashes, then the whole system crashes, so applications must be extremely well tested. It also means that each application can read and alter data contained within other applications.

The memory management divides the available memory into:

- *dynamic heap*: the dynamic heap is execution-based and clears on reset. Its size is between 64 and 256 kb, depending on the total memory size. It is used to store global variables, the stack, and

3.20

The Palm user interface

21

dynamically-allocated memory. It provides fast read and write access, but has no ability to protect itself against unauthorized writes;

- storage: this is designed to hold permanent data, such as databases, files, and application code, and is therefore not cleared on reset. The size is limited only by the available physical memory. It provides fast read access and slow write access, and write protection. Internally, all data are stored in databases with transaction support provided by the operating system. This guarantees the integrity of the data even in the event of a system crash.

Software development for Palm OS

Palm supports C and C++ for software development. The C support is quite exhaustive, whereas the C++ support is just at its beginning. Because the Palm OS is written in C, this is currently the best choice for developing performance-critical applications.

The free Palm development environment consists of a software development kit (SDK) based on GNU (GNUs, not UNIX) for Windows, Mac and Linux. There is also a commercial, integrated development environment available for the Palm: the Metrowerks CodeWarrior. This includes an editor, compiler, debugger, and visual graphical user interface constructor. Unfortunately, the shared library format between CodeWarrior and GNU is not compatible.

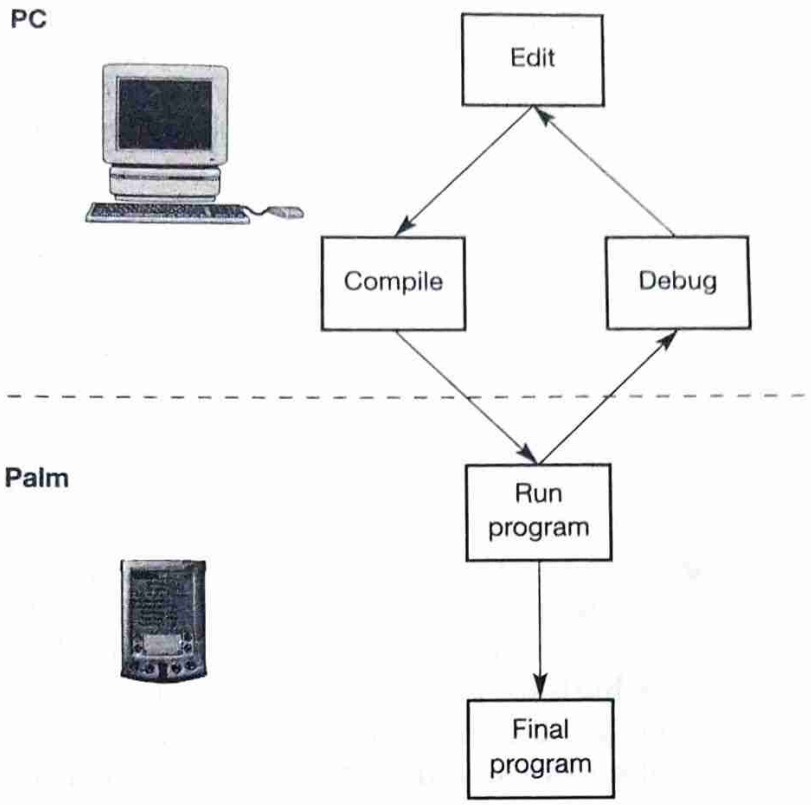
Palm supplies a Palm emulator, which emulates the Palm hardware on a PC. This means that Palm programs can be run and debugged on the emulator and then the same program can be downloaded to a real Palm device. Figure 3.21 shows the development cycle for Palm programs. On the PC, the Palm program is edited and then compiled. It can either be run in the Palm emulator on the PC or downloaded to a real Palm (the figure shows the latter). Afterwards, the program can be debugged remotely from PC. When a stable version is reached, this code is the final program. There is no need for any additional compilation steps, as in the EPOC development environment.

Application development in C is simple and fast for the Palm, because there is an extensive system library, and a lot of example programs are available. The support for C++ is minimal at the moment, but will hopefully be improved with the next major release.

Palm applications are synchronous and event-driven. They consist of the main event loop and the event handling. Events can be user interface actions (button pressed, handwriting input, etc.), system notifications (power management, global search, etc.), and application-specific events.

Several virtual machines (VMs) for the Palm are available from third parties, including the KVM (Java 2 Micro Edition) from Sun, the J9 from IBM, and the WabaVM from Wabasoft.

Figure 3.21



Palm development cycle

The Sun JVM implements the Java 2 Micro Edition standard from Sun, the J9 is performance and real-time optimized, and the WabaVM is targeted for small devices, with a subset of the Java language and Java bytecodes. However, all Java Virtual Mahcines (JVMs) struggle with the very tight resource constraints of the Palm. Because a JVM with class library takes 256–512 kb of storage, and quite a large amount of heap space, they are not very popular at the moment. This will probably change with the next generation of Palm devices, when the minimal memory configuration should be increased. For details about Java on pervasive devices, see Section 3.5.

3.4.2 EPOC

EPOC was originally created by Psion, but is now maintained by an offspring company called Symbian, which was founded by Psion, Motorola, Panasonic, Ericsson, and Nokia in 1998. The EPOC operating system was designed specifically for phones. There are two versions: EPOC16 for 16-bit processors and EPOC32 for 32-bit processors. The current version of EPOC is Release 5, which is available for NEC V30H (16-bit version) and ARM/StrongARM (32-bit version) processors.

EPOC supports Unicode, which is important for the Asian market. EPOC can display 256 colours. 23

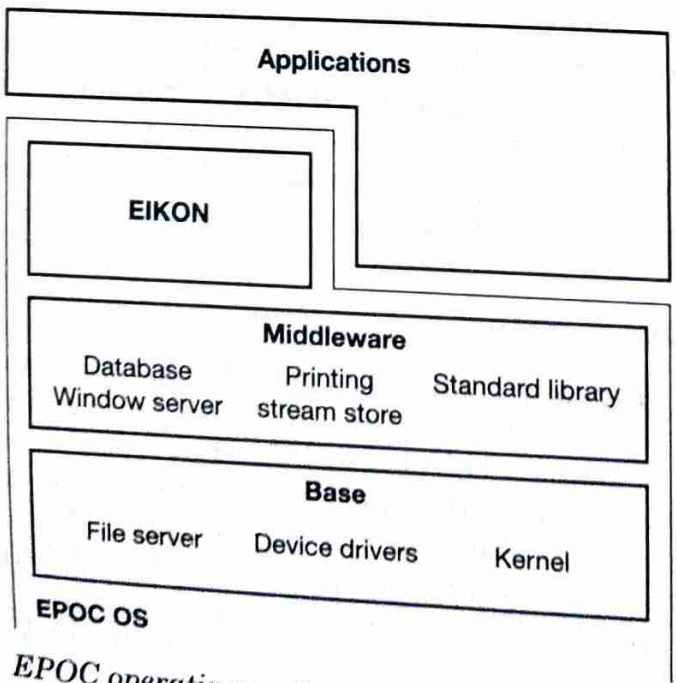
Core operating system functionality

In contrast to the Palm OS, which consists mainly of one task, EPOC is heavily multitasking. An overview of the EPOC operating system can be seen in Figure 3.22. The base layer provides the fundamental APIs; the middleware layer provides the graphics, data, and other components to support the graphical user interface and applications; EIKON is the system graphical user interface framework; and finally there are the applications.

The EPOC operating system consists of the following features:

- *User management*. Because EPOC devices are considered to be personal devices, EPOC is a single-user operating system.
- *Task management*. The real-time microkernel with low-interrupt and task-switching latency provides multitasking with a pre-emptive, priority-driven scheduler.
- *User interface*. The EPOC user interface supports display, keyboard, and sound. The user interface framework in EPOC is named EIKON and provides all the standard graphical user interface elements, such as buttons, dialogs, and menus. It is also responsible for handling the data and command input. Figure 3.23 shows the EPOC user interface of an Ericsson device with a map application.
- *Memory management*. EPOC has a memory management unit (MMU) concept to provide separate address spaces for each application. The EPOC operating system and development tools also provides a rich set of tools for checking out-of-memory errors and freeing up unused memory. These tools include design patterns, stack clean-up heap failure, and heap-checking tools.

Figure 3.22



EPOC operating system architecture

24

Figure 3.23

Ericsson EPOC handheld device

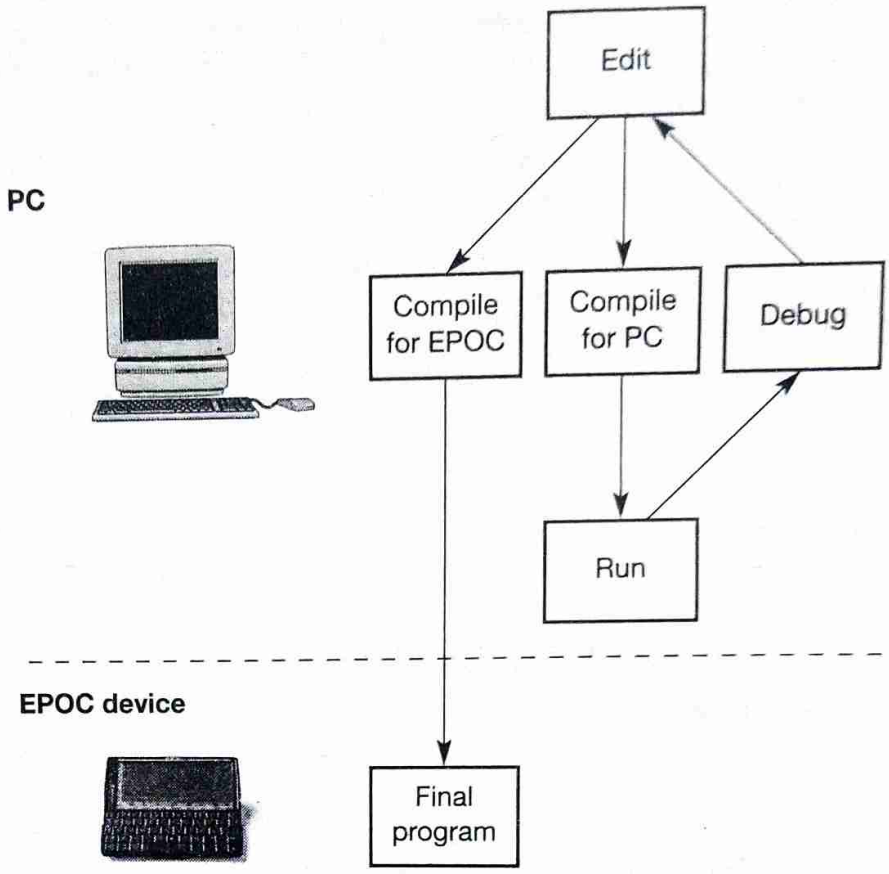
Courtesy of Ericsson

Software Development for EPOC

Programming languages supported by EPOC are C++, Java, and OPL, a proprietary BASIC-like language. C++ is the language of choice for system development and high-performance application programming. There are C++ development environments from GNU, for compiling the device code, and Microsoft Visual C++, for compiling for the emulator, available for EPOC. OPL has a long heritage in Symbian's EPOC and Psion's SIBO software, for rapid application development, and is specific for EPOC. Also included is a runtime environment for Java (JDK 1.1.4 in EPOC Release 5) from Sun with the Abstract Window Toolkit (AWT) graphical user interface classes. Symbian provides a simulator for testing and debugging EPOC programs. The development cycle for EPOC applications is shown in Figure 3.24. The EPOC program is edited on the PC and then compiled for the PC. After that, it can be run in the simulator on the PC and can be debugged. Finally, the program must be cross-compiled for the EPOC device and then loaded onto the EPOC device.

Normally, EPOC applications are developed in C++, because this is the best compromise between performance and development cycle time. OPL is an advanced scripting language that allows for rapid prototyping and is EPOC specific. OPL is an interpreted language and is therefore not suited for programs that require fast execution speed. As can be seen from Figure 3.25, Java is in the middle between C++ and OPL. In contrast to OPL, Java is not proprietary to EPOC, but it can also be used on other platforms.

Figure 3.24



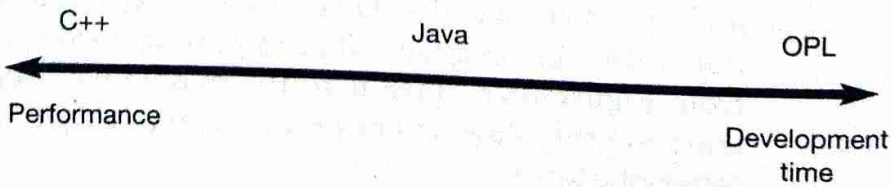
EPOC development cycle

EPOC applications are different from Palm OS applications because EPOC heavily relies on multitasking. Therefore, the application programmer can choose to program either synchronous applications (which would look like Palm OS applications) or asynchronous applications. In the asynchronous case, the application is waiting for messages from other processes (e.g. keyboard) and is not blocking the processor. This results in faster execution of parallel tasks, or power saving if no other task is running.

3.4.3 Windows CE

Windows CE is an embedded operating system developed by Microsoft. Previous versions (1.0 and 2.0) of the Windows CE user interface were similar to the Windows user interface. This meant clicking on about ten pop-up menus with a pen to enter a new date in the date book, which was

Figure 3.25



Program performance and development time

26

far too inconvenient and time-consuming for such a simple, frequently-used task. The current version (3.0, which is used in the Pocket PC) is now better optimized for ease of use. Windows CE is available for 32-bit CPUs such as ×86, SH3/4, ARM/StrongARM, PowerPC, and MIPS.

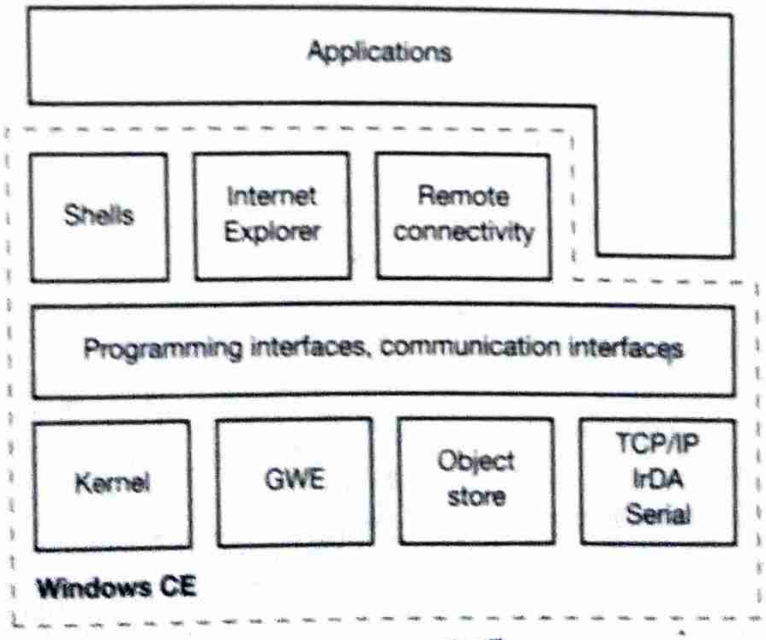
Windows CE 3.0 offers real-time support, a smart card subsystem for PC/SC compliant readers, is Unicode based, and supports grayscale and color graphics up to 32-bit depth.

Core operating system functionality

Windows CE is a modular operating system that can be configured by the device manufacturer. This is a result of the read-only memory (ROM)-based design of Windows CE, in contrast to more desktop-oriented, disk-based operating systems like Linux or BeOS; it can even be configured at runtime. This allows an operating system with only the essential parts to be created and saves precious space in the device. The different modules are shown in the Figure 3.26.

The kernel provides memory management, task scheduling, and interrupt handling. The graphics/window/event manager (GWE) integrates the user interface functions of graphical output and user input. The object store is the persistent memory of Windows CE and includes files, the registry, and a database. Finally, the communication interfaces include infrared communication via IrDA, TCP/IP, and serial drivers.

Figure 3.26



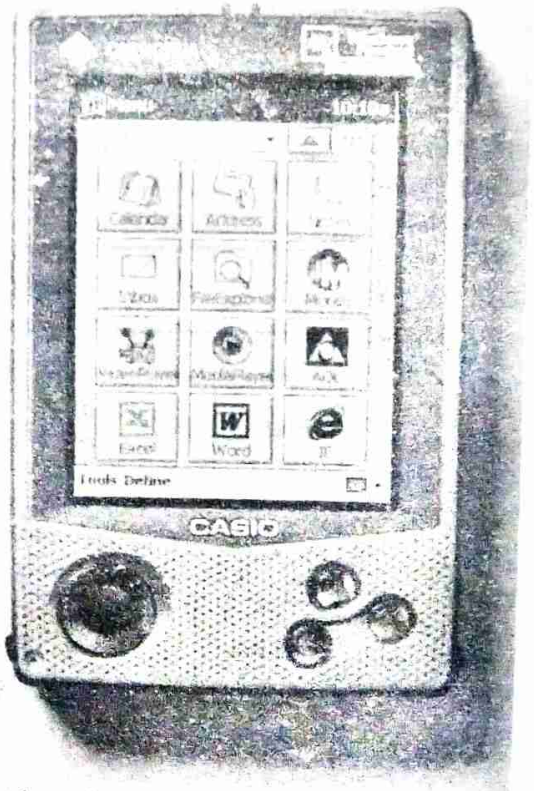
Windows CE architecture

27

Windows CE offers the following features:

- *User management*. Because Windows CE is designed for PDAs, it supports only one user.
- *Task management*. The task manager supports 32 simultaneous processes and an unlimited number of threads (limited only by the available physical memory).
- *Operating system size*. The Windows CE footprint can be as small as 400 kb for the kernel, up to 3 MB with all modules, and up to 8 MB including Pocket Word and Internet Explorer.
- *User interface*. Windows CE provides menu controls, dialog boxes, and icons, and supports sound. An example of the Windows CE user interface can be seen in Figure 3.27. The desktop shows icons like Word, for a reduced version of Microsoft's text-processing software Word, and the Microsoft media player for playing MP3 music files.
- *Memory management*. A protected virtual memory system that supports up to 32 MB memory per process protects applications against each other. There exists a special heap for the file system, registry, and object store that has a transaction service for ensuring data integrity. The object store can have a size up to 256 MB.

Figure 3.27



Windows CE user interface
Courtesy of Casio

27

- *Security*. Windows CE has support for cryptography with a cryptographic library (Cryptographic Application Programming Interface, CAPI) to securely store information in memory. The kernel-loader authentication program can use public-key signatures to prevent unauthorized applications from running. As an additional feature, it is possible to achieve more security for sensitive data by using the smart card interface of Windows CE. Private data of up to 64 kb can be stored securely and non-volatily on a smart card. Access to the data, however, will be slower because of the electrically erasable and programmable read-only memory (EEPROM) memory used instead of battery-backed RAM.

Software development for Windows CE

Since Windows CE is based on the Win32 API, it offers significant advantages in application software development. It is already known to a lot of developers, and there are professional development tools, such as Visual C++ or Visual Basic, available for this API. It also allows developers to reuse Win32 code from existing PC-based programs. Because Windows CE uses major parts of the Win32 API, the application programming is very similar to that on the PC.

There are JVMs from third parties available for the Windows CE, including the KVM (Java 2 Micro Edition) from Sun, the J9 from IBM, and Waba from Wabasoft.

3.4.4 QNX Neutrino

The first QNX operating system version was available in 1981, and the current QNX Neutrino has now reached the version 2.1. QNX is a real-time operating system consisting of a microkernel surrounded by a collection of optional processes (resource managers) that provide POSIX- and UNIX-compatible system services. Depending on whether resource manager processes are included at runtime, QNX ranges from ROM-based embedded systems up to a full-blown operating system with network support. This microkernel-based architecture is known as universal process model and allows for separation of processes in different address spaces. This is true not only for user processes, but also for the resource manager processes. That means that even if the file system driver or network driver crashes, then the system will still work, which leads to highly reliable and stable systems.

QNX is very well suited for set-top boxes or car devices, but there are also efforts to target the hand-held device space, e.g. to add power management support and a graphical user interface suitable for hand-held devices (Photon). It supports MIPS, Power PC, and ×86 processors.

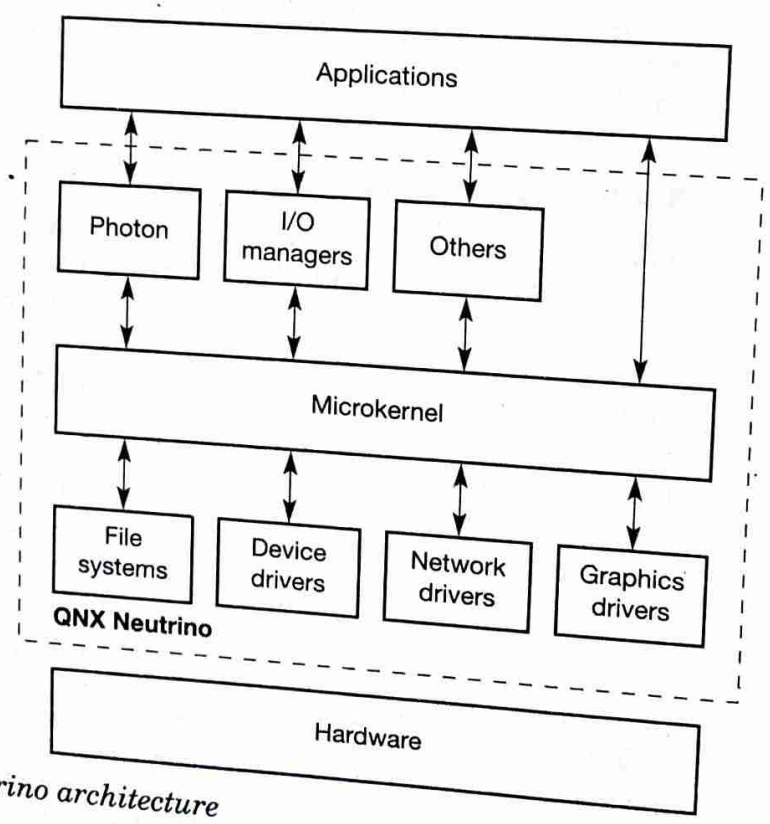
Core operating system functionality

The core operating system consists of a microkernel that has a size of only 12 kB. This kernel implements four services: interprocess communication (IPC), process scheduling, low-level network communication, and interrupt dispatching. The IPC is based on messages and can be given priorities.

In addition to the core microkernel, there are several resource managers that run as separate processes. Examples include a POSIX-compliant file system and device manager, network services (e.g. TCP/IP), graphical user interface, and power management. This architecture is called universal process model and is shown in Figure 3.28. A small set of core services reside in the kernel, while the remaining code runs in separate processes. This results in an extremely stable and flexible operating system. Malfunction in one part of the operating system (e.g. the device driver) does not cause the whole system to stop. In addition, it allows operating system services to be added or removed at a later date.

QNX Neutrino has the following features:

Figure 3.28



QNX Neutrino architecture

29

- *User management.* QNX supports only one user.
- *Task management.* QNX supports real multitasking. Through its modular organization, it allows for incremental linking and loading of components, resulting in an on-the-fly operating system configuration.
- *User interface.* X-Window-like concept, based on widgets to implement graphical user interface elements. Consists of a micro-graphical user interface, which can be scaled up or down dynamically.
- *Memory management.* The QNX memory management has an MMU concept for separation of address spaces of applications. The memory protection is used not only for applications, but also for operating system services running in different threads.
- *Additional features.* QNX is POSIX compliant, allowing existing POSIX-compliant code to be reused.

Software development for QNX

The QNX toolkit includes the GNU or Watcom C/C++ compiler, debugger, graphical user interface builder, and PhAB for creating rapid prototypes. In addition, there is a development environment available from Metrowerks.

Since the POSIX standard is C-based, the language of choice for programming fast and compact applications is C. Because most APIs are POSIX-compliant, reusing existing code and developing new code is easy and does not require extensive training. QNX is based heavily on multitasking, so applications should be programmed asynchronously in order to save battery power.

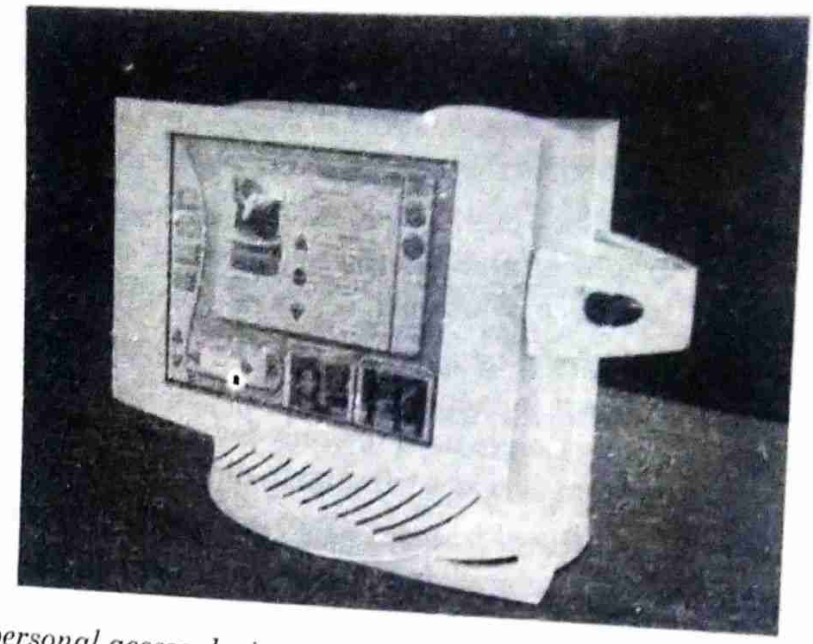
3.4.5 BeOS

BeOS was founded in 1985 with the goal to create a media operating system. Consequently, the current version (5) of BeOS is highly optimized for multimedia applications and is a good candidate for multimedia boxes. It offers multiprocessor support to integrate seamlessly sound and graphic processors, and a 64-bit file system for dealing with the huge amounts of data that modern multimedia applications deal with (e.g. the size of an uncompressed DVD is about 300 GB). BeOS is available for Intel and PowerPC processors.

Figure 3.29 shows how BeOS can be used as an operating system for an Internet personal access device, providing the multimedia foundation that such a device would need.

30

Figure 3.29



Internet personal access device running BeOS
Courtesy of Be Inc.

Core operating system functionality

To achieve the requirements for real-time multimedia and communication, BeOS features a very-fine-grained multiprocessor support included in its operating system core. The architecture is based on a symmetric multiprocessor model, allowing each processor to execute parts of the operating system and give each processor full access to all resources. In addition to that, it supports virtual memory and pre-emptive multitasking.

Another BeOS specialty is the so-called pervasive multithreading. This permits rapid switching between hundreds of small tasks. These tasks can be deployed rapidly on multiple processors, and reassigned when the system's processor load changes.

BeOS has the following features:

- *User management*. BeOS has multiuser support, just like standard UNIX operating systems.
- *Task management*. Pre-emptive multitasking with pervasive threads for fast context switching is implemented by BeOS. BeOS also supports and implements symmetric multiprocessing down to the operating system level.
- *Memory management*. The BeOS memory-management unit provides memory protection between applications and virtual memory support.
- *Additional features*. BeOS supports a 64-bit file system that allows the user to access up to 18 billion GB, in contrast to 32-bit file systems, which allow up to only 4GB.

Software development for BeOS

For the Intel version, a C/C++ GNU compiler and tools are available; for the PowerPC version, an integrated C/C++ development environment is offered by Metrowerks. BeOS is not well suited for devices with tight memory restrictions, but it does allow the standard Java environment from Sun to be used for software development.

Application development for BeOS is similar to that for a modern multitasking operating system like Linux. BeOS is POSIX-compliant, which makes it easy to reuse code from other POSIX platforms, such as Unix or QNX Neutrino. A rich library supporting multimedia applications offers 2D/3D graphics, OpenGL, and several kinds of media formats (e.g. MPEG, WAV, PNG, TIFF).

3.4.6 Embedded Linux

Embedded Linux is a stripped-down Linux operating system with some special support for pervasive devices. As processors and memory are getting cheaper and more powerful, stripped-down desktop operating systems are become increasingly attractive. For hand-held devices, most of the benefit of Moore's law is put not into performance and larger memory, but in cheaper chips and longer battery life.

Unfortunately, there is currently no standard for embedded Linux. However, there is a group called the Embedded Linux Consortium[6] that promotes embedded Linux systems. At the time of this writing, this consortium consisted of over 80 members. Embedded Linux versions are available for nearly all processors used in high-end embedded systems, including MIPS, ARM, Motorola, and Intel processors.

Core operating system functionality

The operating system functionality is the same as for standard Linux. The following features make Linux so well suited for the embedded market:

- *Configurable kernel.* Linux has a microkernel architecture like the QNX operating system. This allows easy configuration of the operating system core. Services and features can be compiled either into the kernel, or as a loadable module that can be loaded dynamically at runtime. This includes services and functions such as device drivers, file systems, and networking support.
- *Scalability.* Based on the modular kernel concept, Linux can be configured from sizes suitable for a watch up to multiprocessor server systems. Embedded Linux vendors offer configuration tools and utilities for tailoring Linux to the special needs of a specific pervasive device.

32

- *Networking*. Because most Linux systems are used as Web servers, the Linux TCP/IP stack is under constant scrutiny for security and optimized for speed. Drivers, utilities, clients, and servers are available for nearly every network function or protocol.

For comparison with the other operating systems described in this chapter, the embedded Linux features are summarized below:

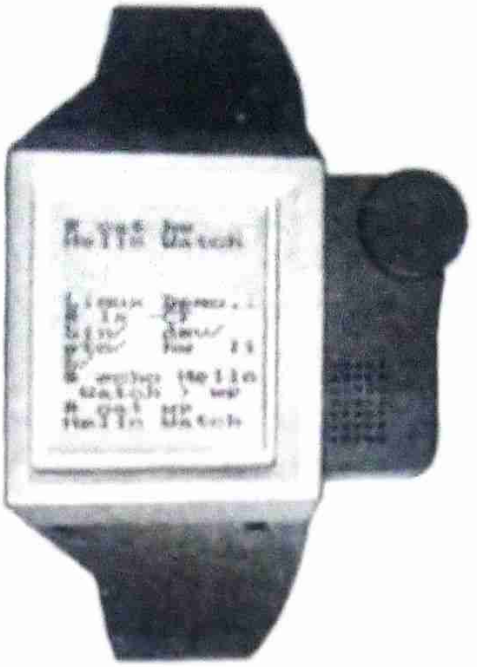
- *User management*. Embedded Linux offers multiuser support, just like standard Linux operating systems.
- *Task management*. Pre-emptive multitasking with optional real-time schedulers is implemented. There is also basic support for multiple processors.
- *Operating system size*. Depending on the configuration, the size of the kernel can range from 200 kB to several megabytes. However, a restriction for embedded devices is the fact that Linux needs quite a lot of runtime memory, typically 2–8 MB.
- *User interface*. Embedded Linux uses the x-Window system for the user interface. Some embedded Linux vendors provide stripped-down versions to save memory.
- *Memory management*. The Linux memory management supports MMUs to provide memory protection between applications and virtual memory for paging memory to hard disks.

Software development for embedded Linux

The same software development tools are available for embedded Linux as for a standard Linux system. Available programming languages include C, C++, and Java. Therefore, software development is fairly similar to that for desktop systems. However, the actual Linux devices can be quite different from normal desktop systems, as can be seen in Figure 3.30.

This figure shows an example of what small pervasive devices running embedded Linux can look like today. The watch is designed to communicate wirelessly with PCs, cell phones, and other wireless-enabled devices. It will therefore have the ability to view condensed email messages and receive pager-like messages. Also, personal information management (PIM) functionality, such as calendar, address book, and to-do list, can be accessed. Future enhancements will include a high-resolution display and applications that will allow the watch to be used as an access device for various Internet-based services, such as up-to-the-minute information about weather, traffic conditions, the stock market, sports results, and so on. The technical data of the Linux watch are: ARM7 processor, 8 MB Flash memory, 8 MB DRAM memory, infrared (IR) and RF wireless connectivity, 96 × 112 pixel touch screen, and a roller wheel.

Figure 3.30



Wristwatch running embedded Linux from IBM research

Courtesy of IBM

3.4.7 Summary

Just as pervasive computing devices cover a wide range in terms of size and functionality, so do the operating systems for these devices. The operating systems described above range from the small, lean, and minimalistic Palm OS to the multimedia, multiprocessor-supporting BeOS. This means that there will not be a single dominant operating system in pervasive computing, but rather a variety of them supporting different device categories. Table 3.3 gives an overview of the features of the different operating systems.

The Palm OS is intended for use in PDAs with limited memory and battery power. It is simple to use, with a user interface that is tailored for the special needs of PDAs, power efficient, and easy to program. The major disadvantage of the Palm OS is that there is no security. Every application can read and alter all data, which will become a problem when PDAs are used as personal security devices (PSDs), which will allow access to rooms and be used in e-commerce scenarios.

EPOC was developed for personal organizers and is now gaining market share in the mobile phone market. Its multitasking support is a perfect fit for mobile phones, which need to respond to user inputs and at the same time be able to receive data over the air.

Windows CE 3.0 has special features such as sound support and crypto support. The ability to customize Windows CE for special devices and only include the needed parts of the operating system is very attractive

Table 3.3 Features of the different operating systems

	Palm	EPOC	Windows CE	QNX	BeOS	Embedded Linux
Supported processors	Motorola Dragon Ball	NEC, ARM	×86, SH3/4, ARM, PowerPC, MIPS	×86, PowerPC, MIPS	×86, PowerPC	×86, PowerPC, MIPS, ARM
Operating system structure	Layered	Layered	Modular	Modular	Layered	Modular
Memory protection	No	Yes	Yes	Yes	Yes	Yes
Operating system size	Tiny	Small	Small	Medium	Large	Small–Large
Security	None	Low	High	Medium	High	High
Multi-tasking	No	Yes	Yes	Yes	Yes	Yes
Examples of pervasive devices	PDAs	Mobile phones	Pocket PCs	Car devices, Internet Appliances	Set-top boxes	Set-top boxes

for manufacturers. Like EPOC, Windows CE aims to reach the high-end PDA market, as the supported processors are all 32-bit processors, which need more battery power than the 16-bit Motorola processor of the Palm OS devices.

Neutrino from QNX is very well suited for car devices, as it is a real-time operating system and many car applications have hard, real-time requirements. An additional advantage is the possibility of replacing parts of the operating system while the system is running. This will be the most likely scenario for updating software in cars, with over-the-air connection to the car and the new software transmission modules. Other potential uses of Neutrino are in set-top boxes, Internet appliances (like the 3Com Audrey) and PDAs or pocket PCs, where QNX provides new features such as power saving technology and a Palm-like graphical user interface library.

BeOS is the perfect operating system for set-top boxes and other multimedia devices. As TVs become increasingly intelligent, switch to digital technologies (like MPEG), and interact with other devices (e.g. hi-fi system, room-control computer), the importance of multimedia operating systems will increase.

Embedded Linux is a Linux version configured for a special device. It has all the basic concepts of a standard Linux, including multi-user support, sound, high-end graphics drivers, and many features for ensuring high security. Embedded Linux has the advantage of having the same programming APIs as standard Linux. This means that several programs can run on an embedded Linux device with just a recompile, and developers do not need special training to write applications for this platform.

3.5 Java for pervasive devices

As we have seen already, pervasive devices have a great variety of operating systems. It would therefore be beneficial to have a common programming platform, such as Java. Since the Java Standard Edition requires significant computing resources, and resources are limited in pervasive computing devices, the challenge is to have a small, performing Java implementation for pervasive devices.

This section gives an overview of the different versions of Java, discusses the two relevant Java versions for pervasive devices (Java Micro Edition and Real-time Java), and describes three virtual machines available for pervasive devices (Sun's KVM, Waba, and IBM's VisualAge Micro Edition).

3.5.1 Java

The Java technology was created in 1991 at Sun as a programming tool in a small, closed-door project called the Green Project. The goal of the project was to look for the next wave in computing, which the team thought would be the convergence of consumer devices and computers. For this project, James Gosling created a new language that he called Oak, after the tree outside his window. As the plans for small computing devices were dropped, and the group focused on the Internet, the language was renamed Java and officially announced in 1995.

Because Java has its roots in the small-device area, it is no surprise that after the success of Java on clients with applets and then on servers with Java Enterprise Edition, Java has now finally reached mobile computing devices. The current Java versions are called Java 2 Platform and consist of:

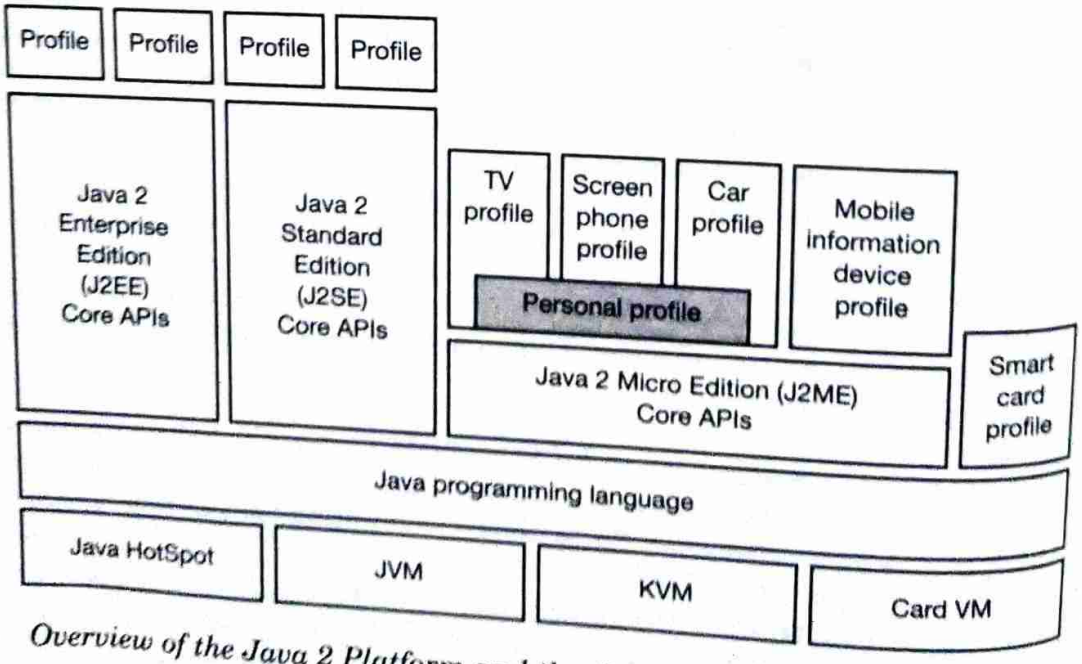
- *Java Micro Edition* (J2ME), which includes different virtual machines, core APIs and market-specific APIs defined in profiles. J2ME has been designed for pervasive computing devices and is explained in more detail below;
- *Java Standard Edition* (J2SE), aimed at the traditional PC. It has a richer set of APIs than J2ME, and is used in VMs that are optimized for performance and security but not size;

- *Java Enterprise Edition* (J2EE), created by Sun for server systems. It enhances the J2SE with APIs needed for server-based computing, including Java Beans, JSPs, database access, and more. For more details about J2EE see Chapter 10.

Figure 3.31 shows how the different Java versions relate to each other. The base layer consists of the JVMs that execute the Java bytecode. These JVMs range from the Card VM for smart cards, the KVM for pervasive devices, and the standard VM and Client HotSpot VM for desktop computing, up to the HotSpot VM for servers. The HotSpot VM is designed for high-performance applications, and therefore carries out intelligent precompiling of frequently-used code fragments. All VMs comply with the Java programming language definition, but the KVM and the Card VM only support subsets of this definition. On top of this layer are the different core APIs for the various Java editions. The topmost layer consists of the profiles that tailor the Java editions to special environments.

A profile is a collection of Java-based APIs that supplement a configuration to provide capabilities for a specific vertical market or a specific device type. A configuration defines the minimum Java technology libraries and virtual machine capabilities that an application developer or content provider can expect to be available on all devices.

Figure 3.31



Overview of the Java 2 Platform and the different Profiles

3.5.2 Java 2 Micro Edition

Java 2 Micro Edition is targeted for pervasive computing devices without real-time requirements. Typically these devices are characterized by:

- small amount of available memory (128–512 kB)
- limited energy (battery-operated)
- connected to a network
- restricted graphical display capabilities.

These devices range from smart cards to phones and set-top boxes. To cover this broad range of devices, there are several device configurations and technologies.

As explained above, a configuration defines the minimum Java technology libraries and VM capabilities that we expect to be available on all devices. Based on the Java 2 Micro Edition profile, the following configurations are available:

- *Connected device configuration* (CDC) is based on the PersonalJava technology and has the bytecodes and core APIs of standard Java. It is targeted at devices like screen phones and set-top boxes with more than 512 kB ROM, more than 256 kB RAM, and connection to a network. The main difference to standard Java libraries is the restricted user interface library.
- *Connected, limited device configuration* (CLDC) addresses the geographical user interface, data storage, messaging (e.g. email, SMS), security, and wireless networking for devices with 128–512 kB RAM, such as mobile phones and TV sets. Sun provides a special VM, called KVM, for this configuration that is optimized for memory-constrained environments (see also p. 75).

To enhance the use of Java beyond these two configurations, additional Java2ME technologies are defined:

- *Embedded Java* allows the Java platform to be configured. Unnecessary classes and VM features can be omitted in order to minimize resources and costs for embedded devices. Embedded Java uses the same bytecode as the standard Java edition.
- *Java Card* specifies a subset of Java with a special compressed bytecode format for very small devices, such as smart cards (e.g. SIM cards used in GSM mobile phones, credit cards).
- *Real-time Java* enhances Java with features needed for real-time applications (see Section 3.5.3).

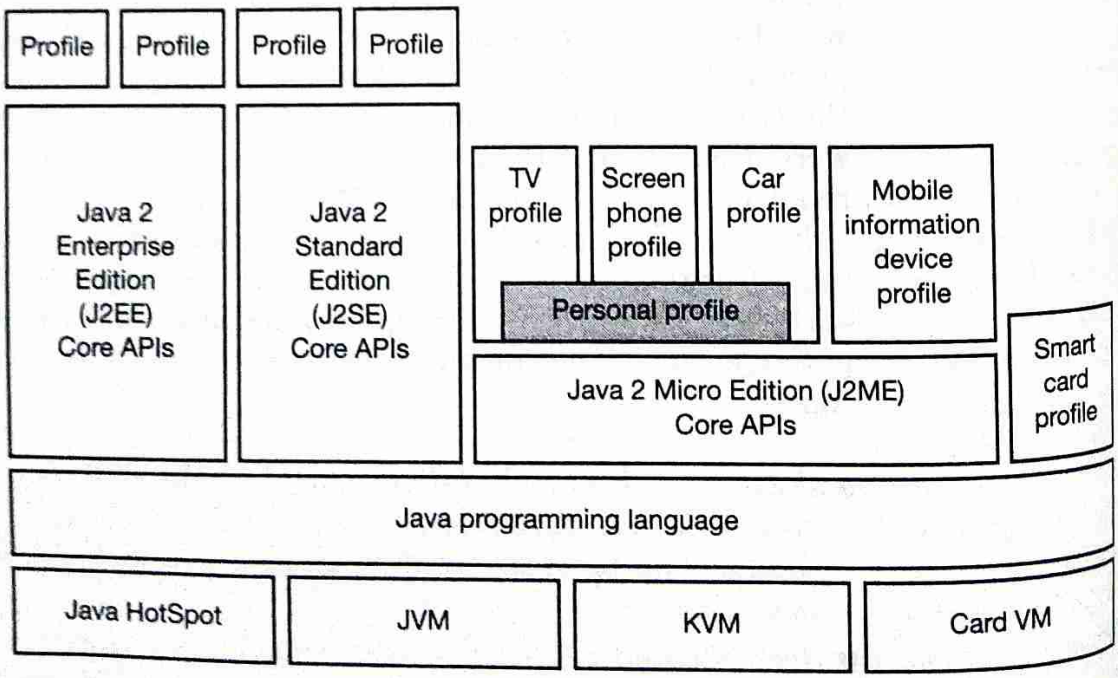
30'

- *Java Enterprise Edition* (J2EE), created by Sun for server systems. It enhances the J2SE with APIs needed for server-based computing, including Java Beans, JSPs, database access, and more. For more details about J2EE see Chapter 10.

Figure 3.31 shows how the different Java versions relate to each other. The base layer consists of the JVMs that execute the Java bytecode. These JVMs range from the Card VM for smart cards, the KVM for pervasive devices, and the standard VM and Client HotSpot VM for desktop computing, up to the HotSpot VM for servers. The HotSpot VM is designed for high-performance applications, and therefore carries out intelligent precompiling of frequently-used code fragments. All VMs comply with the Java programming language definition, but the KVM and the Card VM only support subsets of this definition. On top of this layer are the different core APIs for the various Java editions. The topmost layer consists of the profiles that tailor the Java editions to special environments.

A profile is a collection of Java-based APIs that supplement a configuration to provide capabilities for a specific vertical market or a specific device type. A configuration defines the minimum Java technology libraries and virtual machine capabilities that an application developer or content provider can expect to be available on all devices.

Figure 3.31



Overview of the Java 2 Platform and the different Profiles

3.5.2 Java 2 Micro Edition

Java 2 Micro Edition is targeted for pervasive computing devices without real-time requirements. Typically these devices are characterized by:

- small amount of available memory (128–512 kB)
- limited energy (battery-operated)
- connected to a network
- restricted graphical display capabilities.

These devices range from smart cards to phones and set-top boxes. To cover this broad range of devices, there are several device configurations and technologies.

As explained above, a configuration defines the minimum Java technology libraries and VM capabilities that we expect to be available on all devices. Based on the Java 2 Micro Edition profile, the following configurations are available:

- *Connected device configuration* (CDC) is based on the PersonalJava technology and has the bytecodes and core APIs of standard Java. It is targeted at devices like screen phones and set-top boxes with more than 512 kB ROM, more than 256 kB RAM, and connection to a network. The main difference to standard Java libraries is the restricted user interface library.
- *Connected, limited device configuration* (CLDC) addresses the geographical user interface, data storage, messaging (e.g. email, SMS), security, and wireless networking for devices with 128–512 kB RAM, such as mobile phones and TV sets. Sun provides a special VM, called KVM, for this configuration that is optimized for memory-constrained environments (see also p. 75).

To enhance the use of Java beyond these two configurations, additional Java2ME technologies are defined:

- *Embedded Java* allows the Java platform to be configured. Unnecessary classes and VM features can be omitted in order to minimize resources and costs for embedded devices. Embedded Java uses the same bytecode as the standard Java edition.
- *Java Card* specifies a subset of Java with a special compressed bytecode format for very small devices, such as smart cards (e.g. SIM cards used in GSM mobile phones, credit cards).
- *Real-time Java* enhances Java with features needed for real-time applications (see Section 3.5.3).

As can be seen, there are many varieties of Java to choose from. This helps reduce the resources needed for the Java implementations running on the different pervasive devices, as the Java functionality can be tailored to the specific needs of the device.

3.5.3 Real-time Java

Standard Java is not suitable for real-time applications, because it does not have a predictable runtime behaviour and does not allow reading and writing directly to the system memory. The use of algorithms can solve some of the shortcomings of the standard JVMs, such as garbage collection that leads to unpredictable execution times in the standard JVM. Others, such as direct access to memory, are not available in standard Java as the Java language definition prohibits this.

A collaboration of companies therefore created the real-time version of Java, Real-Time Specification for Java (RTSJ), which is still in its early phases. The draft specification is available, and a reference implementation will hopefully be available soon. This is the first effort under the formalized Java Community Process (JCP 1.0) for Java enhancements. Companies like Aonix, Cyberonics, IBM, Microware Systems, Nortel Networks, QNX, Rockwell-Collins, and Sun worked together to specify this standard.

Real-time Java will be backward compatible to standard Java, and will include all the features needed to build systems with hard real-time requirements. Here are some of the main features of RTSJ:

- *Predictable execution speed.* Non-predictable execution time, caused by time-slicing scheduling and occasional garbage collection, are the major reasons why standard Java is not very well suited for real-time applications. The first priority in designing RTSJ was to always have a predictable runtime behaviour.
- *Customizable schedulers.* RTSJ introduces the concept of a schedulable object, in addition to tasks and threads. Scheduling can therefore be executed on object level. In addition, RTSJ defines a predictable priority scheduler and allows for easy exchange of a scheduler, e.g. to include a custom scheduler.
- *Advanced memory management.* To support the needs of real-time systems with regard to memory management, RTSJ defines different memory types to allow short- and long-lived objects to exist outside the scope of garbage collection. To achieve this, a new definition of object lifetime is provided, and memory areas such as scoped memory (lifetime defined by syntactic scope), immortal memory (has a scope that is larger than the program and can be used to share data between real-time and non-real-time tasks; similar to shared memory on UNIX systems), and physical memory (see below) are introduced.

- *Access to physical memory.* Real-time systems often need fast access to sensors or actors that connect the real-time system to the physical world. To accomplish this efficiently, RTSJ has a mechanism for breaking the Java sandbox at defined places to have direct access to the machine memory. This gives Java programs the ability to read or write directly to the memory of sensors or actors.
- *Object and thread synchronization.* As real-time applications often consist of many threads that must get synchronized at some point in their execution, RTSJ defines synchronized wait-free read and write queues, and monitors for mutual exclusion synchronization, which avoids the priority inversion problem of the standard Java implementation.
- *Asynchronous event handling.* Many real-time systems are event-driven and therefore make extensive use of asynchronous events such as interrupts. RJTS supports asynchronous events and asynchronous transfer of control to other objects and threads.

With these features, Java is very likely to spread to pervasive computing devices with real-time requirements, for example car appliances.

3.5.4 Java virtual machines for pervasive devices

In this section, we describe some commercially available virtual machines for pervasive devices. First, we will cover Sun's KVM, which implements the J2ME standard. Then the Waba VM is explained, which implements a Java-like technology, but uses native device libraries to achieve smaller and faster programs. Finally, we show a fully integrated development environment for Java on pervasive devices, including a virtual machine, the IBM VisualAge Micro Edition.

Sun's KVM

Between the PersonalJava VM, which deals with devices that have more than 512 kB of memory, and the Java Card VM, which is aimed at smart cards with 32–64 kB memory, there was a gap. Extending the Java Card VM to bigger devices did not make much sense, because the Java Card technology has a restricted bytecode set and a special compressed class file format that prevents dynamic class loading. Since a whole new market of pervasive devices falls into this gap, Sun has filled it with the KVM, expecting devices to have more than 128 kB of memory.
Sun's KVM has the following features:

- *Full Java VM.* The KVM implements the full JVM specification, unlike the Java Card VM.

- *Small memory footprint.* The KVM was designed with the goal of a minimal footprint, not maximum performance. For example, the KVM on the Palm OS has a footprint of 50–70 kB and needs 128 kB at runtime.
- *16/32-bit CPU.* To be usable for small pervasive devices, the KVM was designed to run on 16-bit processors at 16 MHz. However, it also runs on 32-bit processors.
- *Reference platforms* for the KVM exist for Palm OS, Solaris, and Windows32.

Because the KVM is based on the J2ME configuration, it has some significant differences to J2SE. These deviations are caused by the limited device capabilities. The main deviations are:

- *different UI classes*: the KVM does not provide the AWT or Swing libraries, as most pervasive devices have very limited display capabilities and limited processor power. The KVM therefore uses special native I/O drivers;
- *restricted VM*: to make the VM as small and efficient as possible, the KVM does not implement some of the standard JVM features. Some features are optional, such as floats, multidimensional arrays, and class file verification, and others, such as remote method invocation (RMI), thread grouping, and reflection, are missing completely.
- *subset of J2SE libraries*: to reduce the size of the Java library for memory-constrained devices, only subsets of the J2SE libraries are supported. For example, java.net and java.io are implemented only partially.

A development kit for the KVM, including the KVM source code, is available from Sun. Porting the KVM to a new hardware or operating system is easy. At the moment, the KVM is running on more than 20 platforms, including Nokia and Sony phones, and Motorola devices, and it will soon be available on the EPOC operating system.

Waba

Waba[7] is a programming platform that includes a language, a VM, a class file format, and a set of base classes. However, Waba is designed such that developers can use Java development tools to write Waba programs, and Waba programs can run as Java applets or applications with no native code.

Waba is very similar to Java, but is not compatible with it. Just like Java programs, Waba programs are compiled into interpreted bytecode, and the code can run without any changes on Palm or Windows CE devices containing a Waba VM. The software to build and run Waba pro-

grams is licensed under the GNU license, just like Linux. The following is a summary of the description available from the Wabasoft website.

To develop programs in Waba, the Java SDK and the Waba SDK need to be installed on a workstation. The Waba SDK consists of reference documentation, a set of Java classes, and some programs that are used to bundle application classes into a format suitable for small devices. It contains the bridge classes that allow Waba programs to run under Java.

While the syntax and semantics of the Waba language are identical to Java, they are not fully compatible. However, the Java compiler from the Java SDK can be used to compile programs. Before the program can be run on the target device, the bytecodes have to be translated into the bytecodes understood by the Waba VM. A tool called exegen performs the translation, and generates an executable that starts a compiled Waba program.

The Waba foundation classes are different from the foundation classes used for developing a program with the Waba SDK. They interface directly with the VM, and call methods that are native to the device the VM is operating on. The class file format supported by a Waba VM is defined as a subset of the class file format supported by a JVM. The Waba class library is designed to be simple and easy to program. It contains all the basic classes needed to create programs, including I/O, networking, user-interface, and graphics classes. Access to Palm OS and Windows CE databases is possible by using the `Catalog` class. It allows writing single-record blob types to and from a database.

Waba programs are executed by the Waba VM. A Waba program consists of classes and a device operating system-specific launch program. The launch program starts the Waba VM and tells it where the class files for the Waba program are. It also tells the Waba VM how much memory to allocate for the program.

The Waba VM was designed to execute a subset of bytecode instructions defined in the JVM specification. Because the Waba class file format is defined as a subset of the Java class file format, developers can use Java development tools to create class files and bytecode that are compatible with Waba, and execute them under a Waba VM. However, Waba VMs do not support operations related to long and double data types, exceptions, or threads.

Waba VMs can be created that run in very small amounts of memory. For example, the complete Waba VM for the Palm OS platform is around 60 kB (around 30 kB for the executable and 30 kB for the full set of foundation classes). Native functions can be added directly into the Waba VM using the Waba VM source code. The Waba VM contains no Java classes and the `waba.lang` package is not part of the SDK. Nevertheless, developers are able to use the `java.lang String`, `StringBuffer` and `Object` classes. When the Waba VM encounters references to the `java.lang` classes, it maps them to their counterparts in the `waba.lang` package.

IBM's VisualAge Micro Edition

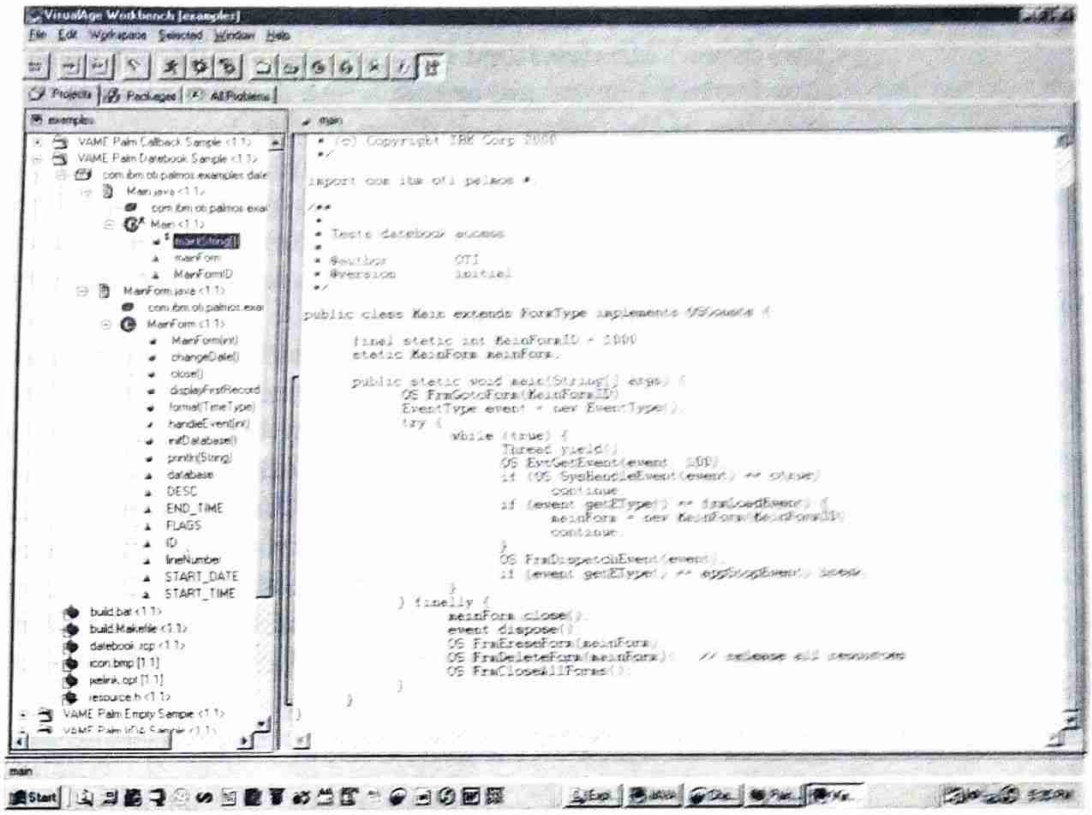
The VisualAge Micro Edition J9 VM[8] is the foundation of IBM's embedded systems solution. VisualAge Micro Edition comes with several class library versions, ranging from extremely small to fully featured. These version choices enable developers to minimize resource requirements by choosing the class library support that best fits their device. All the available configurations are subsets of the JDK 1.2 specification. Examples of the class libraries are:

- *jclXtr*: an extremely reduced function library. The library uses 92 kB of ROM Flash. It contains some APIs from the following packages: java.io, java.lang, java.net, java.util, and java.util.zip.
- *jclCore*: a small library with essential functionality. The library uses 344 kB of ROM/Flash. It contains most APIs from the following packages: java.io, java.lang, java.net, java.util, and java.util.zip.
- *jclGateway*: an extension of jclCore with URL and security functions. The library uses 563 kB of ROM/Flash. In addition to the packages used in jclCore, this package contains java.net.security for permission checking and also the java.lang.Runtime.exec methods.
- *jclMax*: the largest and most feature-rich library. The library uses 2479 kB of ROM/Flash and RAM. It contains most of the APIs from the packages java.io, java.lang, java.lang.ref, java.lang.reflect, java.math, java.net, java.text, java.util, java.util.jar, and java.util.zip. The following security packages are also included: java.security, java.security.acl, java.security.cert, java.security.interfaces, java.security.spec, and com.ibm.oti.security.provider.

VisualAge Micro Edition consists of an integrated development environment that supports JDK 1.2 compatible runtime environments and a set of platform-specific tools (Figure 3.32). The integrated development environment is available for Microsoft Windows and Red Hat Linux. It can be used to debug Java programs locally or remotely. The graphical debugger also supports the well-known operations like setting break points and stepping through the code, and allows viewing of the values of variables. It also allows the source code to be modified on the fly, but this feature is not supported on every target platform.

Supported target platforms are Windows, Linux, and many embedded environments, including Palm OS, Neutrino, embedded Linux and Windows CE. This enables, for example, the development of Java classes for the Palm OS platform with a Windows or Linux PC, and also allows running or debugging the software on the device immediately. In this debugging scenario, the J9 virtual machine running on the Palm device is connected to the PC via TCP/IP (Figure 3.33).

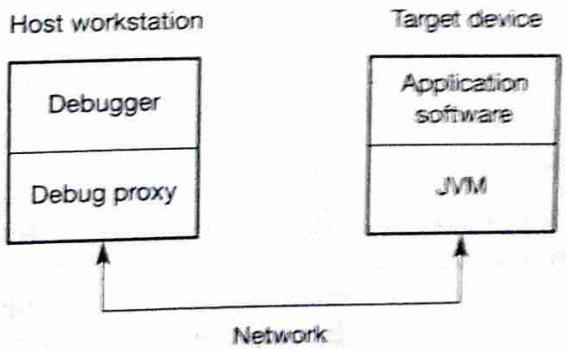
Figure 3.32



VisualAge integrated development environment

VisualAge Micro Edition supports code versioning and includes a repository server for team working. Also included is a smart linker, which creates a special prelinked class file archive where all unused methods are removed. This space optimization becomes all the more important, the less memory the target device offers. For including Java application in the device ROMs, ROM-able class archives can also be created.

Figure 3.33



Remote debugging connection

Comparison

The three VMS described here represent different approaches and therefore have different advantages and disadvantages. Table 3.4 gives an overview of the features of different VMs.

The Sun KVM is the reference implementation for the connected limited device configuration (CLDC) configuration and is available on many platforms. A disadvantage of this more generic approach is that the execution speed is slow and the VM needs large libraries, as the native libraries of the device are not used.

Waba is an open-source project that tries to overcome these drawbacks by having a compressed bytecode format and by using the native libraries. The Waba VM must therefore be rewritten for each new operating system.

The J9 from IBM follows a similar path, but is aimed at the professional market. It therefore delivers libraries that are compliant with the Java standards. It is also integrated in a visual development environment with remote debugging ability. In contrast to the KVM and the Waba VM, the source code of the J9 is not available; the J9 is not available free of charge, but bundled with the Java VisualAge Micro Edition development environment from IBM.

Table 3.4 Comparison of VMs for pervasive devices

	KVM	Waba	J9
Platforms	Palm, Windows CE, EPOC, Linux, Windows, Solaris	Palm, Windows CE	Palm, Windows CE, Neutrino, Linux, Windows
Performance	Low	High	High
Memory use	High	Low	Low
Libraries	Device-independent	Device-dependent	Device-dependent
Standards	Java2ME		Java, Java2ME
Source code available	Under Sun license	Open source	No
Development environment	Standard Java tools with KVM running on development platform	Standard Java tools with Waba VM running on development platform	Integrated development environment with remote debugging ability

4 Device connectivity

Pervasive computing devices do not develop their full potential unless they are connected to applications and services through the Internet. This chapter covers protocols for device-to-device and device-to-server interactions that are relevant in the pervasive computing domain, including wireless protocols, mobile phone technologies, Bluetooth, the Mobile IP, synchronization protocols such as SyncML, and transaction protocols and protocols enabling distributed services, such as Jini. In addition to those protocols, we investigate further algorithms and protocols that address security issues. Because system and device management will become a big challenge to support millions of devices, the last section will discuss device management in the pervasive space.

4.1 Protocols

Standardized protocols are basic prerequisites for meaningful use of pervasive computing devices. Most devices are not very powerful or useful when used stand-alone. They need to exchange data with other devices, for example via wireless protocols.

Wireless protocols will support IP in the near future, making Mobile IP (a specially tailored IP for the needs of mobile devices) very important. We introduce this protocol below and explain its relation to the IPv6 protocol.

Another important topic is the consistency of databases and their data (e.g. calendar entries) between a server and various pervasive computing devices. This problem is solved with synchronization (also called replication). The major protocols supporting this concept are discussed below.

As connected pervasive devices form a distributed network, distributed services and architectures such as Jini are gaining interest in the pervasive computing domain and are explained in this chapter.

To ensure the delivery of the data in an environment, where the de can be switched off or the connection can break down at any time, message and transaction protocols are used to maintain integrity. These protocols are explained in the section on message- and transaction-based protocols.

Today, the main focus is on connection protocols such as WAP and Bluetooth. The first step of a large-scale deployment is to get the devices connected. Soon, these protocols will be established, then data-related protocols will become more important. They will help to ensure security and guaranteed delivery of messages expected by device users. Only when this infrastructure is in place will e-commerce using mobile devices have a substantial basis.

4.1.1 Wireless protocols

Wireless protocols are the natural communication choices for small handheld devices such as PDAs and mobile phones. By definition, no cables are required in order to communicate with other devices.

Several wireless protocols already exist, and most are evolving rapidly. This section offers an overview of WAP and discusses Object Exchange (OBEX), IrDA, Bluetooth, and mobile phone technologies. The 802.11B protocol, a wireless local area network (LAN) protocol with high bandwidth (11 Mbps), will not be covered here. It is more suited for wireless connection of laptops to a LAN, than for pervasive computing devices. However, it may be used in a hybrid solution, where the pervasive device can make use of 802.11 inside an office and UTMS while outside.

WAP/WML

WAP is a technology designed to provide mobile terminal (i.e. mobile phones) users with rapid and efficient access to the Internet. WAP was conceived by Ericsson, Nokia, Motorola, and Openwave Systems, and is now driven by the WAP Forum industry association with over 200 members. WAP integrates telephony services with browser technology, and enables easy-to-use interactive Internet access from mobile handsets. Typical WAP applications include over-the-air e-commerce transactions, online banking, information provisioning, and messaging.

The WAP protocol is similar to HTTP, a high-level Internet communication protocol. WAP has been optimized, not only for use on the narrowband radio channels of second-generation digital wireless systems, but also for the limited display capabilities of today's mobile terminals. The wireless equivalent of HTML is WML, which defines a textual format and a compressed binary format (WBXML). Unfortunately, there is currently no end-to-end security available in WAP, but this will change in the future through a working group that has been set up to solve this issue.

WAP is still a very young standard, and at the moment there are discussions under way as to whether WML can be integrated into Extensible Hyper Text Markup Language (XHTML) to unify HTML and WML. Mobile access to the Internet will increase with new phone technologies and more user-friendly devices (e.g. PDA devices with an integrated phone). Therefore, WAP will become enormously important in the near

IrDA has the following characteristics:

- *Frequency band*. Infrared light is used as the physical transport medium.
- *Security*. Unlike Bluetooth, IrDA has no security concept, but relies on higher-level protocol security.
- *Transmitting capabilities*. Because IrDA is based on infrared light, it consists of point-to-point connections with a narrow angle (30-degree cone) between sender and receiver. IrDA is designed for short-distance communication (0–30 cm).
- *Bandwidth*. IrDA supports data rates up to 4 Mbps, with 16 Mbps under development.
- *Speech*. There is support for only one digital speech channel.
- *Cost*. IrDA senders and receivers are mass-produced, therefore they are very cheap ($1–2).

IrDA is perfectly suited for high-speed data connections (e.g. connecting a device to a wired network). To initiate a data exchange, it requires a device to be in direct line of sight to the other IrDA device (e.g. to exchange virtual business cards).

4.1.2 Mobile phone technologies

The low-level communication protocols of mobile phones are radio-based and suited for long-distance communication (up to about 100 km). However, the technologies used today have a very limited bandwidth, meaning that data exchange rates are very slow. This will change with the third-generation protocols, like UMTS. Table 4.1 provides an overview of the different mobile phone communication protocols and their properties.

The sections that follow explain cellular systems, and the foundation of the mobile phone technology, and then discuss the different mobile phone system generations in more detail.

Cellular systems for mobile communication

Digital mobile communication systems use electromagnetic waves at frequencies around 1 GHz. Small antennae in the range of a couple of centimeters can be used to send and receive signals from a mobile station (mobile phone). However, the reach of mobile stations is rather limited due to power constraints and high damping of signals in the gigahertz range. Therefore, transmitters (senders or base stations) must be placed on a grid at relatively small distances to cover an entire area. Every transmitter covers a cell, which may vary in size from a couple of hundred meters in a city to tens of kilometers in rural areas. Every cell can handle

Table 4.1 Overview of different generations of mobile phone communications

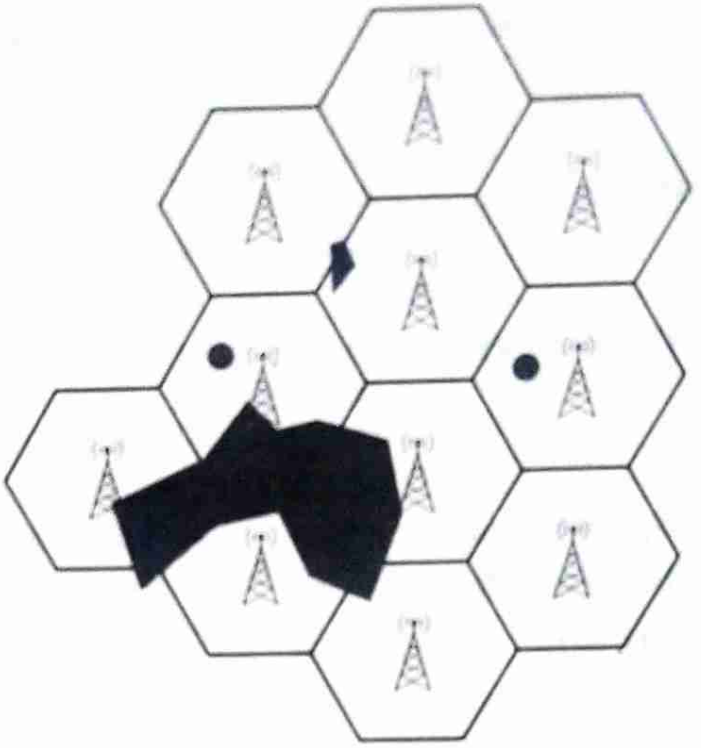
	1G	2G	2+G	3G
Protocol	AMPS, C-Net	GSM, TDMA, CDMA	GPRS, HSCSD, EDGE	UMTS, W-CDMA
Technology	Analog, circuit-switched	Digital, circuit-switched	Digital, circuit- or packet-switched	Digital, packet-switched
Speech quality	Poor	High	High	High
Bandwidth	Low	Low	Medium	High
Security	None	Depending on protocol, low to high	High	High

multiple channels. The limited reach of transmitters is certainly a disadvantage if mobile services must be delivered to a large area because many transmitters are required. However, the limited reach of senders allows reuse of scarce frequency bands between distant transmitters. Thus, a large number of mobile stations can be serviced.

Figure 4.2 shows that there might be dark spots where no mobile service is capable, even when senders are placed with care. Location and size of black spots may vary with weather conditions and mobile device characteristics. Frequently, the high investments for a complete coverage cannot be justified in rural areas, thus in many cases, mobile services are restricted to densely populated areas or the main commuter corridors. Mobile service providers typically publish maps where serviced areas are indicated. However, buildings and other objects may disrupt the signal.[4] Therefore, applications based on mobile systems must be designed to allow for non-availability of service or disruption of services. Typically this is achieved through a combination of online and offline operation.

Mobile stations are normally receiving signals from more than one station. Typically the station connects to the transmitter with the strongest signal. A moving station, e.g. a mobile phone in a car, may travel from one cell to another. The GSM system has been designed to allow movement of stations at a speed up to 150 km/h. Multiple base stations typically receive the signal of a transmitter. Signal delay is indicative for the distance between the transmitter and the mobile station. The station can be located through triangulation with accuracy of 30–150 m by combining

Figure 4.2



Cellular phone array

the information from three stations (this can also be achieved with two stations and a combination of angle of arrival (AOA) and time difference of arrival (TDOA) measurements). This allows the location of the mobile station to be detected, e.g. in case of an emergency call or to offer location-based services. Because such informative is sensitive and user-related, many countries have laws to prevent unauthorized use of these data (e.g. the Wireless Communications and Public Safety Act of 1999 in the USA).

First-Generation mobile systems

Usable wireless systems first came to market in the 1970s. In Germany the B-Net, which needed heavy transmitters and receivers, was successful as a car phone system, and in northern European countries the Nordic Mobile Telephone (NMT) system was launched. The successor of the B-Net, the C-Net, was in use in Germany until the end of 2000. In 1983, AMPS started the wireless phone market in North American countries.

All first-generation (1G) systems are analog, circuit-switched systems. Circuit-switched means that there is a dedicated point-to-point connection between the two ends of the call. As the systems were analog-based, they had poor speech quality and a low bandwidth for services like fax or data transmissions. Some analog systems are still in use (e.g. AMPS throughout the Americas), primarily because there is near 100% coverage for them. Analog systems have no built-in security for data transmission or user authentication.

Second-generation mobile systems

The second-generation (2G) mobile phone communication systems are digital, circuit-switched systems. The switch from analog to digital technology improved the speech quality enormously, and allowed for some additional services, such as encryption of the signal, authentication of the user, authorization, anonymity, and the ability to send short, alphanumeric messages. However, due to the circuit-switching technology, data transmission rates were still low (9.6–14.4 kbit/s).

Because there were a lot of incompatible analog mobile phone standards in Europe, the European Union in 1982 decided to develop a common standard, Groupe Spécial Mobile (GSM). GSM was finally standardized in 1991 and renamed Global System for Mobile communications. It works at 900 MHz, offers full international roaming, automatic location services, signal encryption, user authentication and authorization, anonymity, SMS, fax, and data service. SMS is becoming the most popular service of GSM, with about 12 billion SMS messages sent per month worldwide (data from October 2000). The GSM-900 standard is adopted worldwide in over 160 countries, and is the most popular mobile phone system, with more than 400 million users worldwide (including GSM-1800 and GSM-1900 at year end 2000). This results in a market share of about 60%, followed by the analog AMPS with a share of about 20%. To get a higher user density per cell in cities, GSM-1800 was developed. This works like GSM-900, but in the 1800-MHz frequency band, and offers better speech quality.

The USA also switched to digital systems, but largely stayed with their 850-MHz frequency range from AMPS. Because there were no standardization movements as in Europe, the result was three incompatible systems: the old analog AMPS, and the new digital TDMA and CDMA systems. In contrast to GSM, TDMA and CDMA do not support encryption, authentication, or SMS. With the user density in big cities becoming a problem, CDMA and TDMA switched to 1900 MHz. There is now a US GSM version in place operating in the same 1900-MHz frequency band, therefore called GSM-1900. The reason for all these different frequencies is the lack of standardization. UMTS will be the first telephone standard that also defines worldwide valid frequencies.

Table 4.2 shows the development of the wireless phone market. The first three rows show the most popular digital systems: the GSM system has by far the most subscribers. The figure also displays the trend towards digital systems, as the number of digital system subscribers has grown by more than 25% in six months, and the number of analog system users has declined by 10% in the same time.

Two-and-a-half-generation mobile systems

Between the 2G and 3G mobile phone systems lies an intermediate step, generation 2+G. In contrast to a generation 3G system, a generation 2+G

Table 4.2 Worldwide wireless phone subscribers (millions)

	April 2000	October 2000
GSM	304	397
CDMA	62	76
US TDMA	44	56
Total digital	457	579
Total analog	83	75
Total wireless	540	654

Source: EMC World Cellular Database

system requires only minor hardware and software changes by the phone companies. In Europe, 2+G systems like GPRS and High-speed Circuit-Switched Data (HSCSD) are already in operation. However, users need new mobile phones in order to use these services.

The easiest way to achieve a higher bandwidth than GSM is by aggregating multiple GSM channels, for example in the HSCSD technology. In HSDSC, multiple basic traffic channels are bundled to achieve data rates of up to 57.6 kbps. Different numbers of channels can be assigned to upstream or downstream data transmission. HSCSD can provide a fixed bit rate in transparent mode, or a variable bit rate in non-transparent mode. The advantage of HSCSD is that it is still compatible with GSM technology and therefore requires no major changes or investments by the service provides. Static assignment of channels is not well suited for the typical requirements of Internet applications, where a small burst of data is sent from the client to the server, followed by a large burst of data as response. Therefore, most GSM-based systems will not implement HSCSD, but will move directly on to the next generation of packet-oriented data transmission.

GPRS has been standardized by the European Telecommunications Standardization Institute (ETSI) as part of the GSM Phase 2+ development. It gives GPRS providers a smooth transition to the third-generation (3G) networks, as network architecture, services, and business models are similar. It represents the first implementation of packet switching within GSM. Rather than sending a continuous stream of data over a permanent connection, as in circuit-switched systems, packet switching utilizes the network only when there are data to be sent. This results in a more efficient use of the network, as one physical connection can be shared. The transmission rates range from 20 kbps to 171 kbps. The drawback of packet switching is that there is no guaranteed band-

width, which can be important for live video streams. However, technologies are available, such as ATM, where guaranteed bandwidth can be reserved even with packet-switched networks.

By breaking the transmitted data into packets instead of sending them as a continuous stream, mobile devices remain connected 'virtually' to the server, using airtime only when data are actually being sent. GPRS thereby optimizes airtime use (and possibly associated connection costs) as well as power consumption. This will simplify access to the Internet, making it cheaper and therefore more attractive.

The packet-based GPRS protocol is ideally suited for burst-data applications, such as email or Internet access, and also enables virtually permanent connections to data sources, allowing information to arrive automatically rather than being sought (push model instead of a pull model). Another benefit of the packed-based approach is that being connected to the Mobile Internet does not interfere with receiving a phone call, because the data session may be suspended while the call is answered. Furthermore, it enables broadcast, multicast, or unicast message services that cannot be achieved using standard circuit-switched networks.

Third-generation mobile systems

3G mobile phone systems will bring packet switching and high data rates in the range of several Mbps. This will be a breakthrough for mobile computing, making high data rates and full Internet access using the standard Internet TCP/IP protocol possible for pervasive devices. It will also support bandwidth-hungry multimedia applications, such as full-motion video and video conferencing.

Requiring new hardware for transmission, it will be some time before the 3G networks offer good coverage. 3G networks are expected to be in operation in Japan by 2001, in Europe and Asia/Pacific by 2002, and in the USA by 2003–4.

The UMTS is part of IMT-2000, the 3G wireless services specification of the International Telecommunication Union (ITU). The UMTS Forum was established to focus on spectrum availability, licensing issues and long-term market surveys for third 3G. The UMTS frequencies in the USA are still used by TV stations that have contracts running until 2003–4 with an option to extend it. There are discussions and negotiations about how to make UMTS happen as soon as possible. It is expected that worldwide coverage will be achieved by the year 2005.

UMTS is based on wide-band CDMA (W-CDMA), which was originated by Japan's NTT DoCoMo and is now adopted for 3G use by ETSI. In addition to speech and data, W-CDMA also supports high-speed multimedia services, such as video films, video conferencing, and Internet access, and offers data throughput up to 2 Mbps. Table 4.3 shows how the data transmission rate influences the availability of offered services. For reference, the wired-line ISDN is inserted.

Table 4.3 Delivery time of different services for different phone technologies

Service	ISDN	2G (GSM)	2+G (GPRS)	3G (UMTS)
Email file (10 kB)	1 s	8 s	0.7 s	0.04 s
Web page (9 kB)	1 s	9 s	0.8 s	0.04 s
Text file (40 kB)	5 s	33 s	3 s	0.2 s
Large report (2 MB)	2 min	28 min	2 min	7 s
Video clip (4 MB)	4 min	48 min	4 min	14 s
TV Quality Movie (6 GB)	104 h	1100 h	52 h	~5 h

Source: UMTS Forum (2000) Report 11

Since the patent issues about CDMA seem to be resolved, W-CDMA will soon get a wide availability because the GSM community, which supplies the dominant technology today, supports it. Among the supporters of W-CDMA are NTT DoCoMo, Ericsson, Nokia, and Qualcomm. UMTS and W-CDMA are important for future mobile devices because they offer the high data rates needed for accessing the Internet via mobile devices.

4.1.3 Mobile Internet Protocol

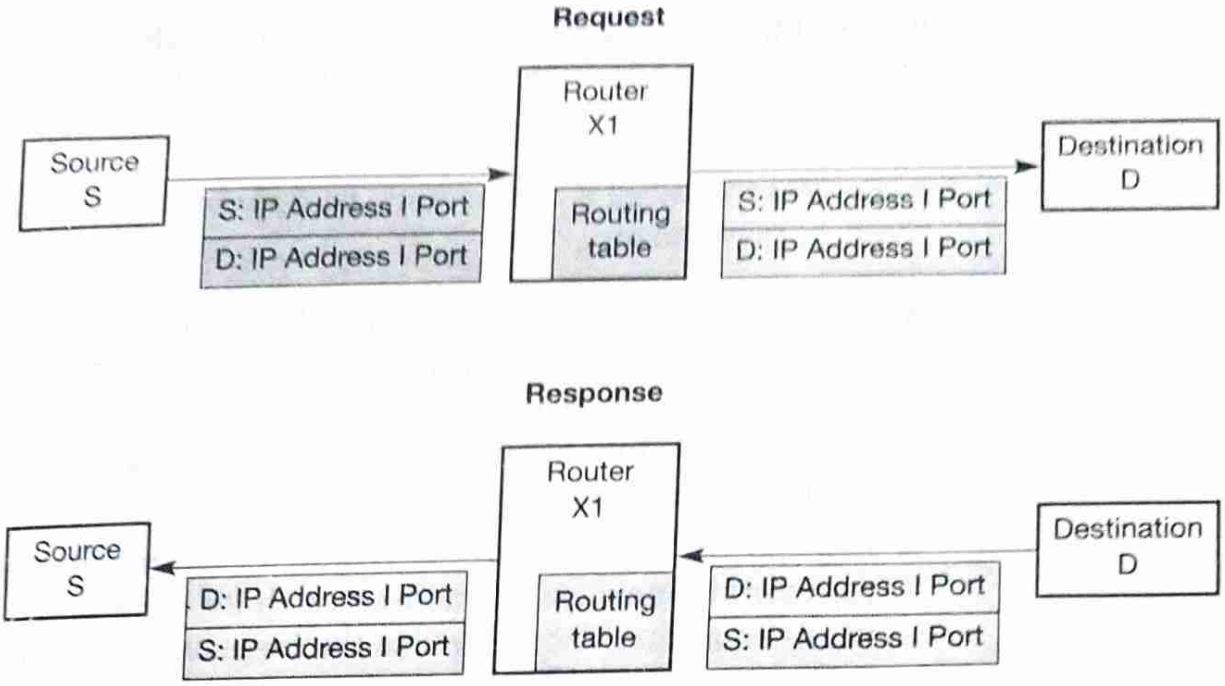
As soon as high-speed mobile connection is available, the next step is to use this connection from a mobile device to access the Internet. This section explains the standard IP and the Mobile Internet Protocol (Mobile IP). Finally, the new Internet Protocol v6 and its effects on the Mobile IP are discussed.

Standard Internet Protocol and mobile devices

At the moment, the standard protocol used in the Internet is IP version 4 (IPv4). IPv4 defines nodes that have a unique and fixed address consisting of a quadruplet. The Internet addressing scheme is similar to that of the telephone. In a telephone network, the addressing scheme consists of a number for the country, a number for the city, and then a number for the individual telephone connection. IP addressing also has a prefix that determines the subnet the address is part of. This prefix can be followed by a group and subgroup prefix before the address number for the individual device.

The messages that are sent through the Internet are called packages. They incorporate the sender's address and port number, and the recipient's address and port number. A scenario for sending a package over IP can be seen in Figure 4.3.

Figure 4.3



Sending a package using the standard IP

The delivery of packages via intermediate hosts is called routing. The routers take the destination address, mask out the low-order bits, and then look up the resulting network address in their routing table. Thus, the IP address typically carries with it information that specifies where it belongs in the IP topology. The routing table contains the next routing host for each network address, and tells the routing host to which host the received package has to be forwarded. There can be many routing hosts between the sender and the receiver of a package.

IPv4 was designed for PCs that do not move around, and it works quite well for this purpose. When using IP with mobile devices, the problem is maintaining an existing transport layer connection with the device. Preferably the device should keep its IP address and port number fixed. As the mobile device moves, it reaches new points of attachment with a different IP address range. The device can now get a new IP address from this point of attachment, but then the connection to its old address will be broken. Mobile IP, provides a way to solve this problem and how to enable mobile devices to communicate over IPv4.

Mobile Internet Protocol

The Mobile IP was created to overcome the problems of IPv4 for mobile devices. In 1996, the Mobile IP working group submitted an IPv4 Mobile Host Protocol to the Internet Engineering Steering Committee (IESC).

To overcome the address problems of IPv4 for mobile nodes (devices) as described above, Mobile IP uses two IP addresses: a fixed home address and a care-of address that changes at each new point of attachment. The

home address is attached to a home agent, so whenever the mobile node is not attached to the home network, it gets all the packages intended for the mobile node and forwards them to the mobile node's current point of attachment.

For establishing a connection to a mobile node, the following steps must be performed:

- *Discover the care-of address*. The Mobile IP discovery is built on top of a standard protocol, the router advertisement. The router advertisements already existing in the IP standard are extended to carry information about the care-of addresses. This enables the routers to update their routing tables.
- *Register the care-of address*. Whenever the mobile node moves, it registers its new care-of address with its home agent. The mobile node needs to authenticate itself to the home agent to prevent unauthorized nodes from registering at the home agent. The authentication is performed with a digital signature.
- *Tunnel the care-of address*. The home agent modifies the received packages so that the care-of address appears as the destination IP address. This is called redirection or tunnelling. When a packet arrives at the care-of address, the reverse transformation is applied. This results in a packet that once again has the mobile node's home address as the destination IP address. Figure 4.4 shows this tunnelling process for the request. As the sender address is unchanged, the reply from the mobile node goes directly to the sender, not via the home agent. This results in asymmetric routing – triangle routing – and can cause routing inefficiencies.

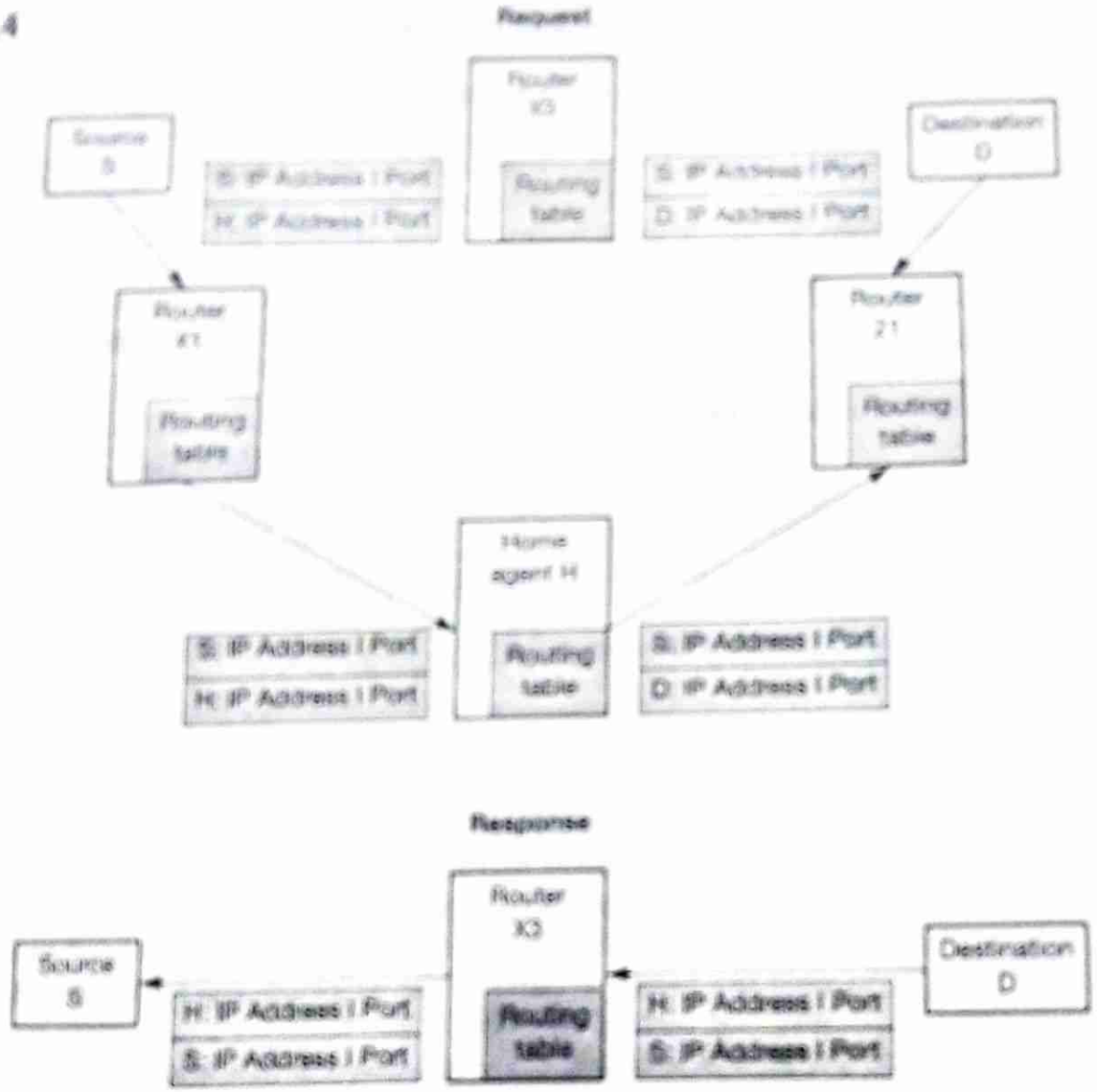
Mobile IP is still an ongoing effort and under discussion. One of the open points is the firewall concept. Firewalls block incoming packages that do not meet specific criteria. One of these criteria is that no incoming packet can have an internal IP address as source. But with the Mobile IP protocol, this is exactly the case if a mobile node outside its home network wants to send a packet to its home network.

Changes to Internet Protocol Version 6

IPv6 will include many features for mobility support that are currently missing in IPv4. IPv6 features relevant for Mobile IP are stateless address autoconfiguration and neighbor discovery. With these two features, a mobile node can create or obtain a topologically correct care-of address for the current point of attachment, without the need for a foreign agent to provide the mobile node with such address.

A big difference between IPv4 and IPv6 is that IPv6 expects all nodes to implement strong encryption and authentication features. Therefore,

Figure 4.4



Sending a package via Mobile IP

Mobile IP for IPv6 can use the IPv6 security features, and need not provide its own security mechanism.

Although IPv6 will support mobility to a greater degree than IPv4, it will still need Mobile IP to make mobility transparent to applications and higher-level protocols such as TCP. Therefore, the Mobile IP working group will submit an IPv6 Mobile IP protocol to the IESG for standardization.

4.1.4 Synchronization and replication protocols

Synchronization, also called replication in the database area, keeps data consistent between different devices. An example of synchronization is having an electronic calendar on the PC and on the PDA. Changes in the

Table 4.4 Different synchronization solutions

	Clients	Server	Used for
AvantGo	Palm OS, Windows CE	AvantGo Server	Retrieving content channels (e.g. Web pages)
Mobile Connect	Palm OS, Windows CE, EPOC	Lotus Notes, MS Exchange, RDBS	PIM applications, databases
IntelliSync	Palm OS, Windows CE	Lotus Notes, MS Outlook, etc.	PIM applications
TrueSync	Mobile phones (Ericsson, Motorola, Nokia), Palm OS, Windows CE	Lotus Notes, MS Outlook, Sidekick, etc.	PIM applications

PC calendar should be reflected in the PDA calendar, and vice versa. Another scenario is software update. When a lot of devices are out in the field, there comes a time when it is necessary to update parts of the software in these devices. Synchronization can be used to efficiently update only the required parts of the software and take account of the dependencies (e.g. library versions).

At the moment there are many synchronization products available for different devices and different needs. Some of them are listed in Table 4.4 (AvantGo is from AvantGo Inc., Mobile Connect is an IBM product, IntelliSync is provided by Puma Technologies, and TrueSync is from Starfish Technologies).

Unfortunately, all solutions have different synchronization protocols, and it is therefore not easy to synchronize two clients with the same software to the same server (e.g. a PDA and a mobile phone). The goal of the SyncML standard is to solve this problem. SyncML is explained in more detail below.

Synchronization principles

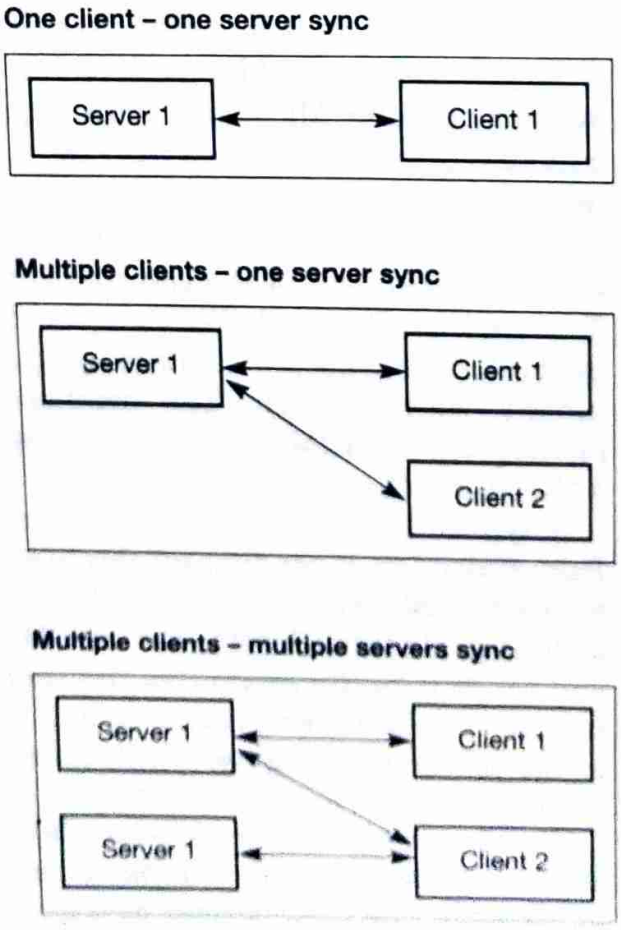
To synchronize data, such as calendar entries or address books, between several pervasive computing devices, there are two strategies: device–server synchronization; and device–device synchronization. As can be seen in Figure 4.5, this can result in complex scenarios, where different device clients can even synchronize with different servers. When perform-

ing device-to-device synchronization, one device needs to act as a server. To handle out-of-sync situations, a special protocol is needed.

Synchronization protocols typically consist of the following steps:

1. *Presynchronization.* To prepare the actual synchronization, some action must be taken before this can happen. These actions fall into the following groups: authentication, authorization, and determination of device capabilities. Authentication ensures that the server is who it claims to be, and that the client is who it claims to be. Authorization checks whether the client is allowed to perform the requested action (e.g. delete, update, or create new entries). Finally, the server determines the device capabilities (e.g. maximum buffer size) to optimize the data flow to the device.
2. *Synchronization.* This is the part in which the synchronization data are exchanged. Between two synchronization partners, all local IDs of data entries are mapped to global IDs known to both partners. Every partner therefore has a mapping table to map local to global IDs. Only the updated, new, or deleted entries are exchanged. If both partners

Figure 4.5



Various synchronization scenarios

update the same data entry, there will be a conflict. This update conflict can be resolved in different ways: attempt to merge the updates, duplicate the entries, let one entry win over the other, or do nothing and report the conflict so that the user can solve it.

3. *Post-synchronization*. At this point all the clean-up tasks are performed, such as updating the mapping tables and reporting unresolved conflicts.

In the following section, we present an example of a synchronization protocol, SyncML.

SyncML

In late 1999, Ericsson, IBM, Lotus, Motorola, Nokia, Palm, Psion, and Starfish founded the SyncML initiative. After the official launch in February 2000, the initiative consists of more than 600 organizations supporting the SyncML standard (as of May 2001).

The vision of the SyncML members is to enable ubiquitous data access from any device to any networked data. To achieve this goal, SyncML released documents describing the protocol and transport bindings, source code of a reference implementation (called SyncML Framework), and source code to demonstrate the use of SyncML.

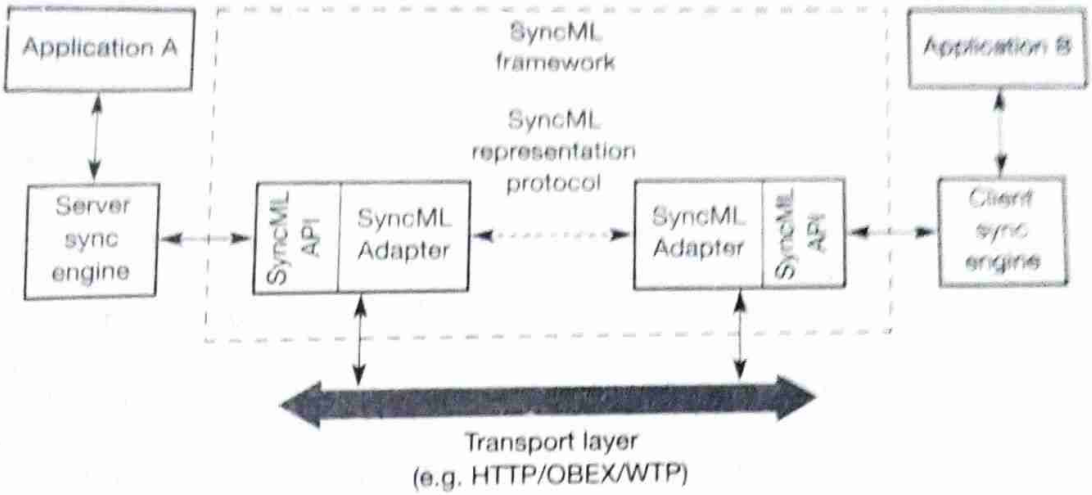
In contrast to the Microsoft Mobile Information Server, which tries to synchronize only Microsoft Office data to different client devices, SyncML supports standard formats for PIM (xCard and xCalendar) and for database synchronization (XRelational).

The SyncML protocol is based on Extensible Markup Language (XML) and is independent of the transport protocol. HTTP, OBEX, and Wireless Transaction Protocol (WTP) are defined as transport bindings for a SyncML-compliant device. Each synchronization message is an XML document, which is sent. The recipient can identify the format of the message by the MIME type of that message, e.g. clear-text XML or binary XML (WBXML).

A typical SyncML system is shown in Figure 4.6. Application A (e.g. Palm Calendar) is sending synchronization data via its client sync engine to the SyncML Framework. The SyncML Framework translates the API calls (e.g. Update, Create) and the data into a valid SyncML document and sends it to the server. On the server side, the SyncML Framework receives the document, parses it, and then sends the command and data to the server sync engine, which then talks to application B (e.g. Lotus Notes Calendar).

The SyncML Framework is available for Windows, Linux, Palm OS, and EPOC. It aims to give device manufactures a jump-start in integrating SyncML into their devices, as it gives the Sync Engine an API for easy use, handling all the XML encoding and the transport protocol-related

Figure 4.6



SyncML Framework

issues. SyncML-compliant devices and synchronization products are now available from Nokia, Ericsson, and Motorola.

4.1.5 Distributed services

Many pervasive computing scenarios consist of networked components that build a distributed system, like mobile phones or PDAs with wireless connectivity. With this distribution and connectivity new challenges arise. Not only are distributed and networked systems gradually becoming more difficult to build than stand-alone systems, but also the complexity is growing exponentially with the number of interacting systems. In addition, networks are not reliable, secure, static, or deterministic. There are not only gradual differences to stand-alone systems, but completely different architectures are required to deal with these issues. For example, the non-deterministic behaviour of a network leads to the problem of deciding whether a component has failed, or whether it is just very slow in processing the request. Distributed services must take all this into account and find a solution for these problems.

While pervasive computing devices get wireless connectivity to build a distributed system, distributed services will become increasingly important. Two examples for distributed infrastructures are presented below: Sun's Jini and Microsoft's Universal Plug and Play.

Jini

Jini, defined by Sun Microsystems, provides a technology for distributed Java software systems.[5] Jini was originally designed for software and hardware, and was targeted for the small office/home office market with a focus on instantaneous networking. However, Jini has developed far beyond the use of consumer devices that can be plugged into a network,

just as appliances are plugged into a power supply. Jini is now used in the enterprise space, where it provides a service-oriented infrastructure, self-healing networks, and federation of services for an easy legacy system integration.

Jini requires that all participants have a Java VM running on their system. This is a more restricted approach than, for example, that of CORBA or DCOM, where data types of different languages are mapped. However, this approach has the advantage that there is no problem with primitive data types (e.g. the number of bytes for an integer value in C depends on the hardware platform). Because all systems require the Java VM, it is also easy to distribute data as well as code. Downloading code has one drawback: it is difficult to ensure that the downloaded code has the same quality as the code already running on that device and does not make the device unusable.

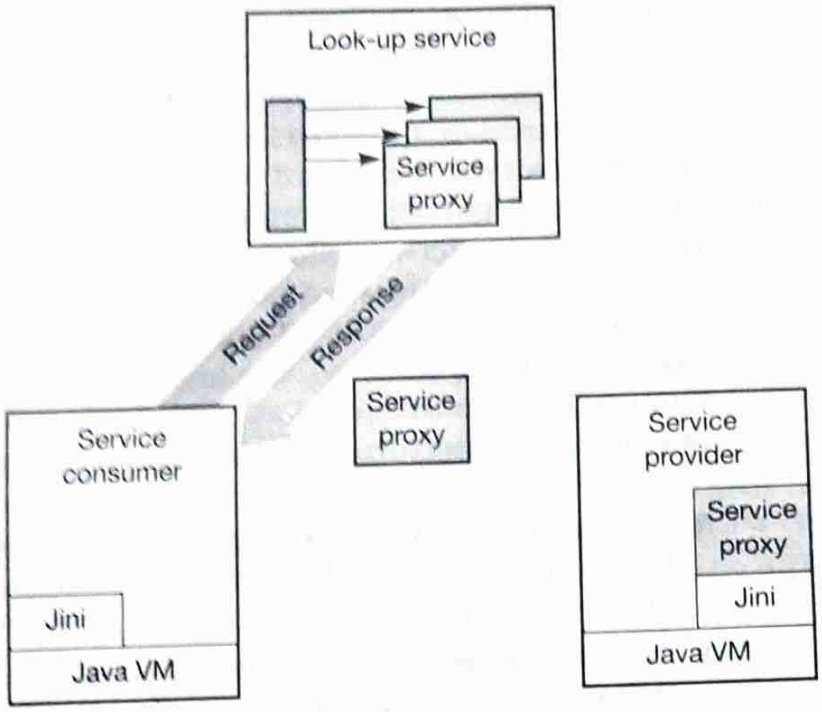
The main weakness of Jini is that currently it is not secure. Jini uses the Java RMI architecture, which still needs some improvement in security. In addition, Jini needs services for authentication and authorization. Hopefully this will be included in a future version of Jini.

As mentioned, Jini uses Java RMI as a communication layer between the different Jini participants. Some very useful services to enable spontaneous, self-healing distributed systems are built on top of RMI. They are based on five key concepts:

- *Discovery:* the service that finds other members in the Jini community and joins them, enabling spontaneous networking capability.
- *Look-up:* a directory that understands the Java-type hierarchy. This means that a look-up search is not simply a text string search, but is based on the object type and even considers the inheritance hierarchy.
- *Leasing:* ensures that the Jini network is stable and self-healing. Resources or services are only leased, meaning they have an expiry time stamp. After that time, they must be requested again and it would become obvious if the service or the providing server was no longer available.
- *Remote events:* allow Jini services to send notification to participants in a Jini community.
- *Transactions:* similar to database transactions. They allow a program to perform a series of computations and reach a consistent state after either all steps are completed successfully, or none of them has been completed. The transactions solve part of the problem of the non-deterministic network behavior mentioned before.

A typical Jini scenario is shown in Figures 4.7 and 4.8. The service consumer first contacts the look-up service and requests a service, e.g. printing (the request in Figure 4.7). The look-up service searches its list

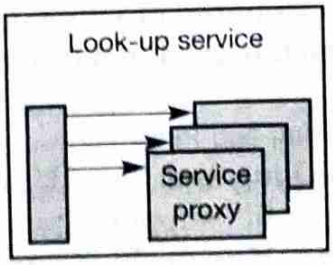
Figure 4.7



Jini scenario for accessing a remote service: first step

of registered services and returns the code to talk to the requested service (proxy service), e.g. printer driver for a specific printer (the response in Figure 4.8).

Figure 4.8



Jini scenario for accessing a remote service: second step

The service consumer can now execute this piece of Java code and therefore use the service of the service provider directly, e.g. send objects to the printer driver, which translates them into the printer protocol (the service use in Figure 4.8). After the look-up service provides the code to talk to the requested service provider, it is no longer involved. This prohibits communication bottlenecks that would occur if the look-up service received all requests first to forward them to the appropriate service provider.

As already mentioned, a prerequisite for Jini is the Java RMI functionality. Unfortunately, a lot of pervasive computing devices do not support Java or RMI yet, and are therefore not able to participate in the Jini community. To solve this problem, Sun developed a proxy concept called Jini Surrogate Architecture. This architecture consist of two parts:

- *Surrogate host:* a full Java 2 system with Jini support. In addition, it knows how to talk to its clients (e.g. WAP, X10, OSGi, USB). It offers discovery services and can store the proxy code for the client.
- *Surrogate client:* the first kind of client does not have a JVM and therefore cannot execute the proxy code. This means that the host must execute the proxy code. The other kind of client has a limited JVM, which allows the proxy code to be executed, but does not have the RMI capability (e.g. J2ME KVM). Executing the proxy code on the client takes workload from the server without requiring the client to implement the RMI package.

Jini allows spontaneous networking and is very attractive for pervasive computing devices that are not attached permanently to a specific network. An example of using Jini in the pervasive computing space is the OSGi, which is used in automotive systems (see Chapter 2).

Universal Plug and Play

Universal Plug and Play (UPnP) was created by Microsoft and currently has a lot of supporters (Compaq, Hewlett-Packard, IBM, Intel, Matsushita, Mitsubishi, Siemens, Sony, and others). It has the same goals as Jini: to provide spontaneous networking and device interoperability. However, UPnP chooses a different approach. Instead of building the distributed infrastructure service on Java, UPnP uses HTTP and XML, which are both open standards. As a consequence, UPnP can distribute data but not code over the network. This can be an advantage (more secure) or a disadvantage (less functionality).

UPnP allows for zero-configuration networking and relies heavily on the Dynamic Host Configuration Protocol (DHCP) for devices in order to dynamically join a network. DHCP is used for retrieving an IP address and to get the DNS server location. It also relies on Auto IP, an enhancement to DHCP to claim IP addresses in the absence of DHCP servers that is based on generating randomly an address in a reserved address space.

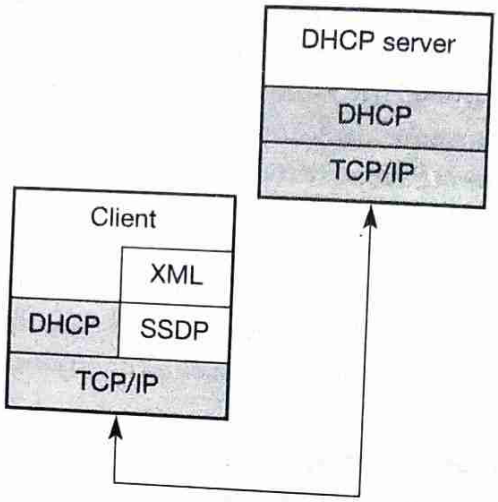
UPnP does not depend on a specific transport protocol, and it can be used with wireless transport protocols. It does not define any security protocols and therefore relies on the protocols used below UPnP or the security mechanism implemented by applications using UPnP.

The services offered by UPnP are very similar to those offered by Jini. Instead of using a programming language like Java to implement these services, UPnP builds on XML standards, such as the Simple Service Discovery Protocol (SSDP) for service discovery, the General Event Notification Architecture (GENA) for events, and SOAP for remote procedure calls. These services are explained in more detail below.

- *IP addressing*. A device that will join an UPnP network must get an IP address. This can be achieved by two different methods in UPnP. The first is to use DHCP, meaning the device must have a DHCP client. The device sends out a request to be answered by a DHCP server. If no DHCP server answers the request in a reasonable time, then the device uses Auto IP to generate an IP address from a set of reserved IP addresses.
- *Discovery*. The discovery mechanism allows devices to advertise their services to control points, and to use new control points to search for devices of interest. The messages sent in both cases are XML documents based on the SSDP protocol.
- *Description*. After a device has announced itself to a control point, it needs to tell the control point what kind of service it is offering. This is called a description in UPnP and is represented by an XML document that the control point requests from the device.
- *Control*. After the control point has the description of the device and therefore knows the provided services, it can send requests to the device. UPnP calls these requests 'control messages'. They are represented by XML documents based on the SOAP standard, which allows remote function calls.
- *Events*. Control points can subscribe to events on devices. Whenever an event occurs, the device will send a notification to all subscribed control points. The mechanism for subscribing and sending the events is taken from the GENA protocol, that also uses XML documents for communication between event generator and event receiver.
- *Presentation*. Devices can specify a presentation URL under which the device can be configured remotely via a browser.

A UPnP scenario can be seen in Figures 4.9 and 4.10. The device (client) uses DHCP to get an IP address from the DHCP server (Figure 4.9). After it has received a valid IP address, it can participate in the UPnP network. Therefore, it starts the discovery mechanism that is

Figure 4.9

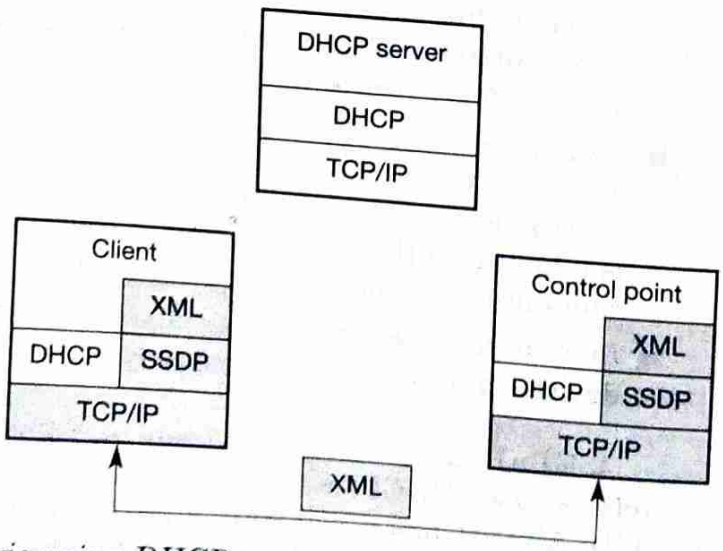


UPnP scenario using DHCP: first step

based on SSDP and advertises its services via an XML document to a listening control point (Figure 4.10).

Now the control point knows the device name, a universal device ID and a URL where it can get the device description. The next step for the control point is to request the device description. The device sends back an XML document containing vendor-specific data, the offered services, a control URL, an event URL, and a URL for presentation (e.g. when a device consists of several subdevices). After that, the control point is able to use the services of that device.

Figure 4.10



UPnP scenario using DHCP: second step

As UPnP requires mainly IP and XML, it does not need much space on a device and is therefore well suited for pervasive devices. The example given on the UPnP website is a digital camera using UPnP to offer remote-control services.

4.1.6 Message- and transaction-based protocols

Message- and transaction-based protocols ensure delivery and atomic operations in the pervasive computing space, where communication breakdowns, or devices being switched on and off while attempting to transmit data, are likely scenarios. This becomes extremely important for e-commerce applications. If something is bought over the Net via a mobile device, it is essential that the order is received, but only once.

The following sections explain the different technologies available, and then give an overview of available products.

Messaging and transaction technology

Guaranteed message delivery and atomic operations are two different technologies. Nevertheless, it does make sense to put both in the same section, as normally we would want ensured delivery for atomic operations. Databases on mobile devices therefore often use messaging systems as their transport layer.

One issue that is slowing down the fast spreading of message systems and transactional databases is the lack of standards. The solution to this problem can possibly be achieved on a higher level using SOAP. SOAP provides a mechanism for exchanging structured and typed information between peers in a decentralized, distributed environment using XML. SOAP is a new proposal to the ITEF and is supported by many companies.[6] For more information about SOAP see Chapter 9.

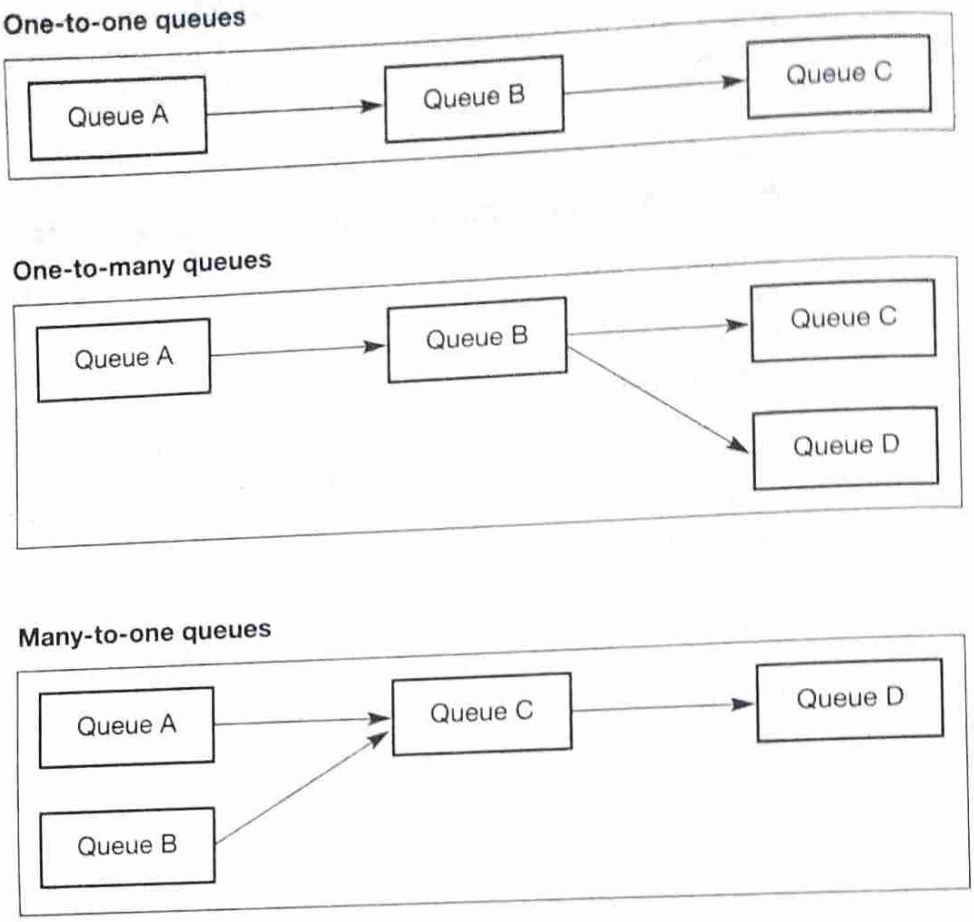
Message queuing solves the problem of assured delivery of a message. This is achieved via a queuing system, where the application puts the message in the queue and the queuing system ensures the delivery.

Queuing systems also allow asynchronous delivery of messages. This is of interest for devices that are not always connected to a network (e.g. PDAs). As soon as there is a connection to the network, the queuing system delivers the message to the recipient. It can be specified how long the message may wait in the queue before it becomes obsolete (e.g. ordering stock).

Figure 4.11 shows some typical messaging scenarios. Complete topologies can be built with queues. Besides the trivial one-to-one queues, the configuration can include distributing one message to several queues (e.g. sending to multiple devices), or collecting messages from different queues into one queue (e.g. forwarding email from different accounts to one device).

Transactional databases are the natural choice for grouping messages into an atomic block (also called transaction). Transactions are important

Figure 4.11



Messaging scenarios

in the e-commerce area to ensure either that no action is performed or that all actions are performed (e.g. one action could be a payment and the other an order).

Transactional databases are very common on server systems, but not many pervasive computing devices have a built-in transactional database with rollback and standard SQL as query language. This is because until recently databases were very big and needed a lot of runtime memory. This is changing now, and more and more database vendors are providing smaller versions that are suitable for pervasive computing devices, interoperating with the databases on the host side. Therefore, transactional databases with an SQL interface will soon be integrated into pervasive computing devices.

Messaging and transaction products

While there are several messaging systems available on the server side, e.g. the free SwiftMQ, which is based on Java Message Services, there are not many products that implement message or transaction support for small devices. The major products are ExpressQ from Broadbeam

Corporation (formerly Nettech Systems), MQ Series Everyplace (MQe) from IBM, Microsoft's MSMQ, and iBus//Mobile from SoftWired. Table 4.5 shows the various message queuing systems and their availability for client operating systems. The footprint of the major products is between 64 and 150 kB.

The situation for databases is similar to that of the message queuing systems. Currently, few relational databases support more than one client device platform. Table 4.6 details the most popular two: IBM's DB2 Everyplace (DB2e) and Oracle's Oracle Lite.

Both databases also have support for database synchronization between pervasive computing client devices and servers.

4.2 Security

In this section, we give an introduction to security in pervasive computing, focusing on the server-side aspects of pervasive computing applications. We start with an overview of the basic security concepts, including identification, authentication, authorization, transaction authorization, and non-repudiation, and give an outline of how they can be implemented. Then we give an overview of the most relevant cryptographic algorithms that are commonly used to realize these concepts as a

Table 4.5 Message queuing systems

	Palm OS	Windows CE	EPOC	Embedded Linux
ExpressQ	✓	✓		
MQe	✓	✓	✓	
MSMQ		✓		
iBus/Mobile	✓	✓	✓	✓

Table 4.6 Relational databases for pervasive computing devices

	Supported platforms	Database footprint
DB2e	EPOC, Linux, embedded Linux, Neutrino, Palm OS, Windows CE	150 KB
Oracle Lite	EPOC32, Palm OS, Windows CE	50–750 KB (depending on configuration)

basis for secure pervasive computing applications. We present standard cryptographic protocols for authentication and secure transmission of data, and give some examples of how these protocols are used in pervasive computing applications.

4.2.1 Security concepts

Before discussing security of pervasive computing applications, it is important to understand the basic security concepts. Below, we explain briefly the concepts of identification, authentication, authorization, and non-repudiation that will be used in the remainder of this chapter.

Identification

There are various methods for identifying users accessing a server using pervasive computing devices. The most universal method is using a user identification that the user has to enter or that is stored in the device. Another method that may be employed if the user has a mobile phone is using the telephone number. If the user has a certificate, the certificate's unique identifier can be used. Because the user may access a server using different identifiers, depending on the device used, these identifiers have to be mapped to canonical user IDs.

Authentication

Authentication means proving that somebody actually is who he or she says they are. Authentication may be performed in various ways, depending on the capabilities of the device used, resulting in different levels of assurance of the user's identity. The most universal authentication method is authentication by user identification and password. After a secure connection between the client and the server has been established, for example using Secure Sockets Layer (SSL) from a PC or Wireless transport layer security (WTLS) from a WAP phone, the application prompts the user for the identification and a password using an appropriate form. The user identification and the password, or a hash of the password, are transmitted to the server and verified to authenticate or reject the user.

A more secure method that may be used with PCs, or soon with WAP phones with a wireless identification module (WIM), is authentication using a smart card. The importance of smart cards is increasing, especially in Europe and Asia, as they are used as SIMs in all GSM phones, and more and more smart-card-based payment systems and public-key infrastructures (PKIs) emerge. Here, an authentication protocol is executed between the smart card on the client side and the authentication software on the server side, e.g. by an applet or a browser itself. Usually, the server gives a random challenge to the client, who provides it to the smart card and

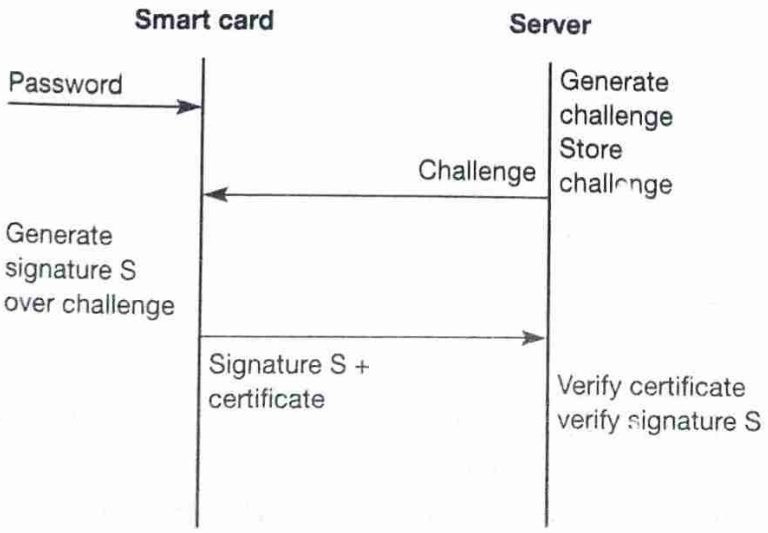
requests a signature over the card identification and the challenge. Often, a password provided by the user has to be given to the smart card before a signature can be obtained to ensure that someone who finds or steals the card cannot abuse it. Finally, the card identification with the signature is sent back to the server for verification (Figure 4.12).

Authorization

A common authorization scheme is to define principals and associate permissions that these principals posses. Permissions can be used to invoke particular methods of particular objects. Such schemes are appropriate when there is only one sort of client and one sort of authentication protocol that may be used by these clients. Because pervasive computing applications may be accessible from various clients using different authentication methods, authorization to perform an action may depend not only on the principal, but also on the device and authentication method he or she currently uses. Table 4.7 gives an example.

User A has a PC and a WAP phone. A can access his or her account information and transfer money from the PC after authenticating themselves using a smart card. Access to the account information from the PC is also possible after authentication with user identification and password. From the WAP phone, the user can only view the account information after authentication with user identification and password. User B has only a normal telephone, thus there is no entry besides the voice entries, which specify that user identification and password are required for this user to view account information or transfer money. User C has a PDA and can view account information and transfer money authenticated by user identification and password.

Figure 4.12



Example of an authentication protocol

Table 4.7 Examples of authorization

User/role	Device	Authentication mechanism	Permissions (application/function)
User A	PC	Smart card 1024-bit signature	Home banking/ view account
	PC	Smart card 1024-bit signature	Home banking/ transfer amount
	PC	User ID/password	Home banking/ view account
	WAP phone	User ID/Password	Home Banking/ view account
User B	Voice	User ID/password	Home banking/ view account
	Voice	User ID/password	Home banking/ transfer amount
User C	PDA	User ID/password	Home banking/ view account
	PDA	User ID/password	Home banking/ transfer amount

Transaction authorization

Some applications allow users to initiate transactions that are very sensitive, e.g. money transfers in home-banking applications or placing orders in brokerage applications. To achieve an appropriate level of security for these transactions, the user usually has to authorize each individual transaction. The technical means for transaction authorization must assure that only the legitimate user is able to authorize transactions. Two commonly used solutions for this problem are digital signatures endorsed by a password and transaction authorization numbers (TANs).

Digital signatures endorsed by a password

To enable transaction authorization through digital signatures endorsed by a password, the user is equipped with a token that has the ability to store securely a key, and to generate digital signatures using that key. The token has to be set up in a way that allows generation of a signature only if the user provided a password before. When a user initiates a sensitive transaction, the server requests the user to generate a digital signature over a challenge and the transaction data using the token. The user enters the password for the token and the token generates the required

signature. The signature is passed to the server, which checks it and executes the transaction only if the signature is correct. Thus, authorizing a transaction is only possible if one possesses the token and knows the password for the token. A person who steals the token will not be able to use it because he or she will not know the required password.

Transaction authorization numbers
TANs are secret numbers delivered to the legitimate user in blocks. The organization that sends the TANs to the user makes sure that the users are aware of the importance of the TANs: the user may have to acknowledge formally that they have received a TAN block, and they may have to sign a statement that they will keep the TAN block secret and never reveal it to anyone else. When a user initiates a sensitive transaction, the server requests the user to enter the next valid TAN (each TAN may be used only once). The user enters the TAN and it is sent to the server. The server checks whether the TAN matches the number it expected. Only if this is the case does it execute the transaction.

Non-repudiation
Non-repudiation means that a user cannot falsely deny later that he or she authorized a transaction. To ensure non-repudiation, transaction authorization must result in data – like a digital signature, for example – that can be used later to prove that the transaction was authorized by the user.

4.2.2 Device security

The security of pervasive computing devices varies considerably. Some devices are considered very secure and are even used for financial transactions while others provide only a very low level of security. The security of a particular device depends on many parameters. Several devices run unchangeable software, while others allow loading arbitrary software – potentially Trojan horses – into the device. A number of devices have no memory protection and thus do not isolate data of an application from other applications, which potentially might be dangerous. Some devices can only support signatures and encryption with small key lengths that are not considered secure, while others have enough computing power to support key lengths that provide a very high level of security. A few devices have secure hardware modules built in to store private keys, while others only have one memory shared between applications.

When implementing distributed pervasive computing applications that need to be secure, it is very important to be aware of the level of security that the client devices support.