Introduction

The rapid technological evolution of wireless devices, such as smart cards, laptop computers, mobile phones and personal digital assistants, that we have witnessed in recent years, enables a new generation of applications and systems; however, these present new challenging problems to researchers and designers, such as variable network topology (due to moving devices and frequent disconnections), scarcity of network and hardware resources.

The complexity and heterogeneity of a distributed system is usually made transparent to application programmers exploiting middleware technologies [Emm00], which raise the level of abstraction, hiding the complexity that is inherent in a distributed system. In the last decade, a very large number of successful middleware systems have been proposed; however, they have been developed for wired networks and may unsuitable for a mobile setting. In fact, these are generally based on design paradigms that rely on fixed networks. Furthermore, some systems targeting fixed network setting have been adapted to a mobile environment. Recently, many research efforts have focused on designing completely new and innovative middleware systems that are capable of supporting the requirements imposed by the mobile scenario.

We focus on the problem of data sharing in a mobile context, where hosts may be unreachable or disconnected from the network. Caching is usually used in distributed systems in order to increase availability and to improve system performances. In a mobile scenario, it is essential, since only using local replicas it is possible to access and modify data, while a host is unreachable or disconnected.

Distributed cached data are not necessarily consistent, since users may modify them, when they are disconnected. Therefore, applications have to deal with the problem of data re-synchronization (also called *data reconciliation*) after a reconnection. Existing systems do not provide effective mechanisms for inconsistency management; the most common behaviour is to choose a copy

considering a predefined priority or using timestamping without implementing particular resolution policy. Furthermore, they usually apply the principle of transparency and prevent the middleware to exploit knowledge that only applications have in order to solve conflicts between replicas that are stored in different hosts.

XMIDDLE [MasCZE02] is a data sharing middleware for an ad-hoc setting that has been developed at University College London. It is an innovative platform that hides the complexity due to this highly dynamic mobile scenario, performing an automatic reconciliation process of data replicas, also exploiting application-specific knowledge in order to deal with possible conflicts.

The goal of this thesis is to contribute to the design and implementation of some aspects of this system related to data replication and synchronization. Essentially, the problem that I addressed is the distributed reconciliation process of data replicas.

The data structures to be stored and manipulated by our system are trees, a classic but, at the same time, very expressive data structure. Therefore, XMIDDLE is able to support the development of applications that need to manage complex and structured information; it is worth noticing that this would be more difficult exploiting other paradigms like tuple space based systems that can manage only flat structures.

We use the eXtended Markup Language (XML) [BraPS00] in order to represent trees and exploit XML related technologies, such as DOM [W3C02] and XML Schema [Fal01], to manipulate data. This approach also permits the coordination between mobile hosts at different levels of granularity, since it is possible to share branches of tree structures that are stored in mobile hosts.

In this thesis we also present an original approach to inconsistency management in XML data structures that exploits the expressiveness of XML Schema in order to represent information semantics for conflict resolution.

This thesis is organized as follows. In Chapter 1 we describe the fundamental issues and challenges in mobile computing design and the technologies that enable this new scenario. The state of art of mobile middleware systems is

analysed in Chapter 2 in order to understand the possible solutions and to try to define general models of this class of systems.
In Chapter 3 we introduce XMIDDLE, focusing on its architecture and the main ideas that are behind it; the aspects related to data replication and synchronization are presented in Chapter 4. The implementation of our middleware is described in Chapter 5, analysing the original solutions that we have adopted. In Chapter 6 we focus on two possible target domains and we describe the applications that we have developed for these contexts. The results of the testing phase of XMIDDLE prototype are presented in the last chapter, where we also discuss and evaluate some aspects of our system comparing it with other existing middleware platforms.

This thesis has been developed at the Department of Computer Science, University College London, UK, with a sixth months visit period as a member of the Software Systems Engineering Group, working in the research area of middleware systems for mobile computing under the supervision of Dr. Cecilia Mascolo.

Chapter 1
Mobile systems

In recent years, we have witnessed outstanding advances in wireless technology and a growing success of mobile devices, such as laptop computers, mobile phones, personal digital assistants, smartcards and the like. These are usually equipped with various networking capabilities such as IrDA [IrDA02], 802.11 [IEEE02], Bluetooth [Blu02] and similar technologies. This new scenario has engendered a new paradigm of computing, called mobile computing. In this chapter we describe the challenging issues of this research area and the network technologies that are related to it; finally, we will introduce general definitions and conceptual paradigms.

1.1 The challenges of mobile computing

From a technical point of view, the challenges raised by the new generation of mobile devices are not trivial [For94] [Sat96a]. First of all, mobile computers require wireless networks that are generally of lower quality than wired ones, because of limited bandwidths, higher error rates and more frequent disconnections [Kle96a].

In contrast to fixed computers, which are connected to a single network, mobile computers have to deal with heterogeneous network connections: in different places they may experience diverse network qualities: for example, a meeting room may have better wireless equipment than a hallway.

Although mobile devices are getting rapidly more powerful, their computational and memory capabilities are still limited, even if we can point out that the performances of a recent PDA and a desktop PC of early 1990s are comparable. In recent years a large number of ad-hoc solutions has been developed; all leading manufacturers have been designed low-power processors to address this application field. In fact, mobile devices are also sensitive to power consumption

owing to the use of batteries; therefore applications must also be aware of the cost of the device activity and manage energy expenditure, for example switching off components (such as screen light) after a specific idle time.

Security is an important issue in any distributed systems, even more so we consider a wireless environment, when the medium is more accessible. To prevent data tampering as well as eavesdropping, it is important to ensure that connections are always private and secure. For example, technologies such as IPSec-based VPNs and personal firewalls can supplement traditional techniques to provide excellent data protection [Int01]. Furthermore, portable computers are more vulnerable to damage and loss; a possible solution is to exploit replication to synchronize and copy data in a secure media (i.e., a centralised data repository).

Mobile devices may be also limited in terms of the input and output capabilities; in fact, a portable computer requires a limited dimension and a constrained size user interface. In this field many researchers are investigating possible solutions: nowadays, the most promising technologies are touch sensitive display and voice recognition. With respect to pointing devices, if the mouse is the standard pointing device for desktop computers, it is unsuitable for mobile devices: therefore, for example, the main PDAs use pens and touch displays (with graphical interfaces for applications) for the interaction with the user.

1.2 Mobile wireless devices and operating systems

A mobile device is characterised by the possibility of moving and attaching to provide services theoretically anywhere. The portability, together with their ability to connect to networks in different places, makes mobile computing possible.

There is no precise classification of these devices, by size, by shape, weight or computing power. The following list gives some example of mobile and wireless devices graded by increasing performance (CPU, memory, display, etc.):

- Sensors
- Embedded controllers
- Pagers
- Mobile phones

- Personal Digital Assistants
- Pocket Computers
- Notebooks/Laptops

A Personal Digital Assistant (PDA) is a handheld device with a small portrait-oriented screen that combines computing and networking capabilities. A typical PDA can function, for example, as a cellular phone, a fax sender and a personal organizer. Unlike portable computers, most PDAs are pen-based, using a stylus rather than a keyboard for input; for this reason they may also incorporate handwriting recognition technologies.

There are three leading operating systems for mobile devices: PalmOS [Palm02], PocketPC [Mic02a] and Symbian [Sym02]. PalmOs is a proprietary operating system that has been designed to work with Palm products; these are mainly PDAs that were conceived as a complementary device to a personal computer and so they need a minimal number of functionalities. Only one application can run at any time and all ones are single-threaded; moreover Palm OS was designed with the assumption that applications will be event-driven so all programs contain an event loop. Built-in functions allow programmers to utilise the infrared port to transmit information to other devices and to synchronize with them. Palm OS does not use a conventional directory file system; all data are held in database records. The version 5 of this operating system introduces also some security APIs for 128-bit RC4, SHA1 and RSA cryptographic algorithms, as well as SSL 3.0/TLS 1.0 services.

Pocket PC is a Microsoft operating system and was formerly known as Windows CE 3.0; its essential features are similar to other Windows products (adapted to mobile devices). It introduces a Common Execution Format to allow developers to ship the same version of their applications to every type devices running Pocket PC on different processors. It also provides a large number of functionalities to interact with other Windows-based machines, for example for data synchronization [Mic02b].

Symbian is a consortium formed by Ericsson, Matsushita, Motorola, Nokia and Psion. Symbian's EPOC operating system has been designed to exploit the potential of mobile devices better and to minimise their drawbacks related to the scarcity of resources. Multi-tasking is implemented through event-driven

messaging rather than with multi-threading; it has a POSIX-compliant interface and an integrated Java Virtual Machine. With respect to networking capabilities, it provides all standard communication protocols, such as TCP, IP version 4, IP version 6, WAP, IrDA and USB. EPOC also addresses some requirements related to the new generation of mobile phones; for example it supports text messaging, multimedia messaging (using MMS) and picture messaging (with EMS). Moreover, it provides a SynchML client for data synchronization; we will describe SynchML initiative later in paragraph 4.1.2.4. Programmers can essentially use C++ or Java to develop applications. We can also underline that Java has a central role in Symbian OS; in fact it provides a full PersonalJava implementation with all mandatory features and most optional features defined in the PersonalJava Application Environment 1.1 specification.

1.3 Wireless networks technologies

In order to understand the characteristics of the application domains of our middleware and the related technologies, we now give a general description of Wireless Local Area Network and a brief overview of 802.11 and Bluetooth standards.

1.3.1 Wireless Local Area Networks

Wireless Local Area Networks (WLAN) are essentially the extensions of today's fixed local area networks into the wireless domain.

The main goals of WLANs are to replace cabling and at the same time to introduce a higher flexibility for communication among mobile users; they are typically restricted in their diameter to buildings, campuses, rooms, etc.

We can find some advantages introduced by WLANs; firstly, they provide flexibility: in fact within radio coverage nodes can communicate without restriction; they are a solution in an environment, for example, where wiring is impossible or difficult. Furthermore, we can point out that a wired network requires an initial precise design of infrastructure; in a wireless environment, as long as devices follow the same standard, they can communicate without previous

planning. Finally, wireless networks are more reliable in case of disasters, for example earthquakes.

On the other hand, WLANs also exhibit disadvantages in relation to Quality of Service issues: in fact, they offer lower quality than their wired counterparts, mainly because of the lower bandwidth due to limitations in radio transmission and higher error rates.

1.3.2 Infrastructure-based and mobile ad-hoc networks

Wireless networks can be divided into two main groups, infrastructure-based and mobile ad-hoc networks, according to their topology [Sch00].

With respect to the first ones, the infrastructure not only provides access to other networks, but also other relevant functionalities such as forwarding functions or medium access control. In this kind of network, communication typically does not take place among wireless nodes, but only between these and the so-called access point. In fact, the design of infrastructure-based wireless network relies on the fact that the most part of functionalities are concentrated on access point (Figure 1.1); therefore, clients can be simple devices. This structure is similar to switched Ethernet or other star-based networks, where a central element (for example a switch) controls network flows. This type of networks can use different medium access mechanisms with or without collision, which may occur if there is no coordination between entities. However, if only the access point controls medium access, no collisions are possible; this may be useful in order to guarantee a certain quality of service, such as minimum bandwidth.

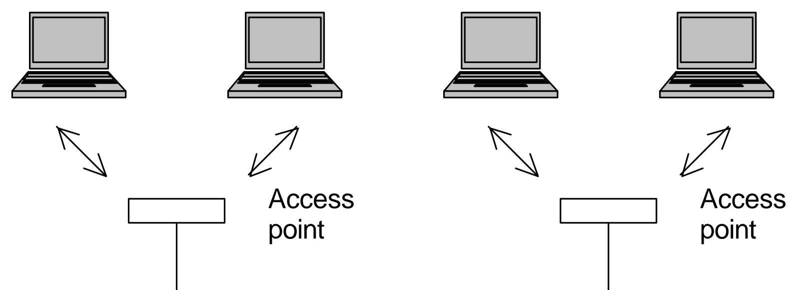

Figure 1.1 Example of infrastructure-based wireless networks

Infrastructure-based networks have a limited flexibility and reliability, because they are based on a single point; for example they cannot be used in disaster scenarios, where no infrastructure is left. Typical examples of this kind of networks are cellular-based mobile telecommunication systems.

The term mobile ad-hoc networks (MANET) [Mac98] [Per01] describes distributed, mobile, wireless networks that operate without the benefit of an existing infrastructure except for the nodes themselves; hosts can communicate with other ones if they are within each other's radio range or if other nodes can forward the message; they also can be connected at the edges to the fixed Internet (see Figure 1.2).

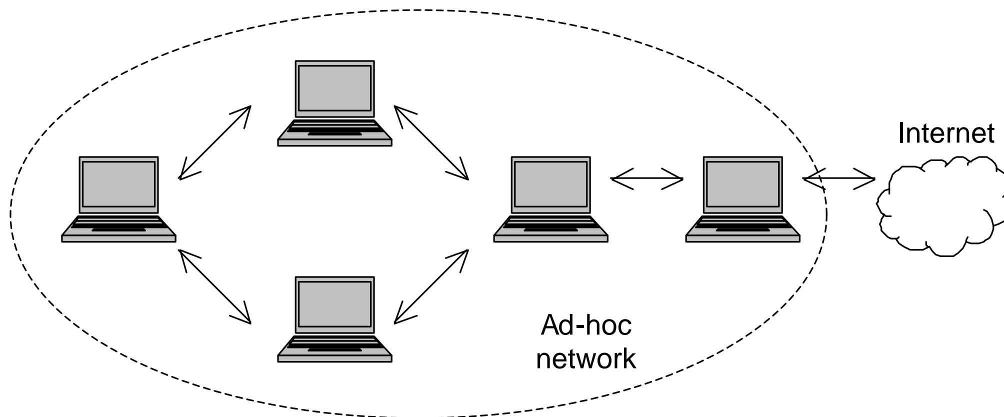

Figure 1.2 A mobile ad-hoc network connected to the Internet

The first researches in this field were sponsored by U.S. Department of Defense (DoD); in fact the original motivations for MANET can be found in the military need for battlefield survivability, where war fighters must be able to move freely without any of restrictions imposed by wired communications devices. An additional motivation is that soldiers cannot rely on access to fixed communications infrastructure in battlefield conditions.

We can find a large number of application fields of this kind of networks: probably the most interesting examples are mobile conferencing or collaborative works (i.e., sharing documents during a meeting), home networking and emergency services.

In ad-hoc networks, the complexity of each node is higher because each host has to implement, for instance, medium access control, routing and quality of service mechanisms.
In this kind of networks one of more interesting and challenging aspects is routing [IETF02]. While in wireless network with an infrastructure support a base station can reach all mobile nodes, this is not only always the case in ad-hoc network; in fact destination node might be out of range of a source node transmitting packets and so routing is necessary.
Traditional algorithms do not work at all in this scenario, since they have not been designed to address the aspects related to highly dynamic topologies or asymmetric and redundant links. Therefore extensions of existing algorithms (using hierarchical approach) or absolutely new ones have to be applied. For example, distance vector algorithms, such as RIP, converge too slowly or link state algorithms, such as OSPF, fail completely, since it is based on the exchange of the entire network topology. Moreover, ad-hoc networks face additional problems due to hardware limitations; for example, periodic updates wastes battery power and bandwidth. We do not discuss routing algorithms for ad-hoc networks in depth, because these topics are beyond the scope of this thesis.
The current implementation of the XMIDDLE prototype is designed to address a single-hop scenario, without considering routing among mobile hosts (in other words, a multi-hop scenario).
Basically, the two basic variants of wireless networks do not always comes in their pure form; for example there are networks that rely on access points and infrastructure for essential services (e.g., management functions or access authentication), but also allow for direct communications between the wireless nodes.
In the following paragraphs we analyse two key-technologies for communication among mobile hosts in a wireless LAN context, IEEE 802.11 and Bluetooth standards, in order to understand the technological scenario of our middleware system; the first one is designed for infrastructure-based networks that additionally support ad-hoc networking, while the second addresses ad-hoc network environments.

1.3.3 IEEE 802.11

The IEEE 802.11 [IEEE02] specifications are standards that specify a communication interface between a wireless client and a base station or access point, as well as among wireless clients. The 802.11 standards can be compared to the IEEE 802.3 standard for Ethernet for wired LANs. The IEEE 802.11 specifications address both the Physical (PHY) and Media Access Control (MAC) layers and are tailored to resolve compatibility issues between manufacturers of Wireless LAN equipment. Wireless networks can exhibit two different basic system architectures, as we have discussed before, infrastructure or ad-hoc. The Figure 1.3 shows the components of an infrastructure-based IEEE 801.11 WLAN.

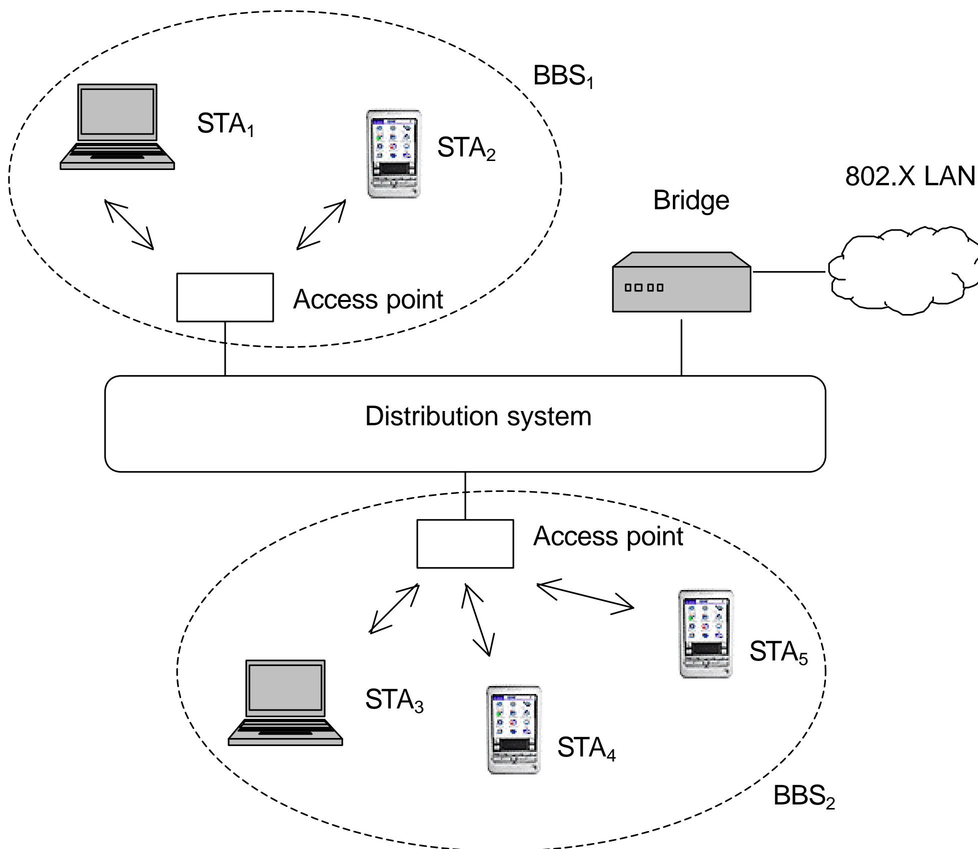

Figure 1.3 Architecture of an infrastructure-based IEEE 802.11 WLAN

Several nodes, called stations (STA_i), are connected to access points (APs). Stations are terminals with access mechanisms to the wireless medium and radio contact to the AP. The stations and the AP that are within the same radio coverage

form a basic service set (BSS_i). The figure shows two BSSs (BSS_1 and BSS_2) that are connected via the AP to form a single network, called extended service set (ESS): this is a typical example of extension of wireless coverage area. Furthermore, as we can see in the illustration, it is possible to connect this wireless network to other LANs, via the APs by the use of a so-called portal, that is an internetworking unit.

Stations can select an access point and associate with it; a distribution system service also provides data transfer between APs (roaming service). Furthermore, APs provide synchronization within a BSS, support power management and can control medium access to support time-bounded services.

In addition to infrastructure-based networks, IEEE 802.11 allows the creation of ad-hoc networks between stations; this scenario is described in Figure 1.4.

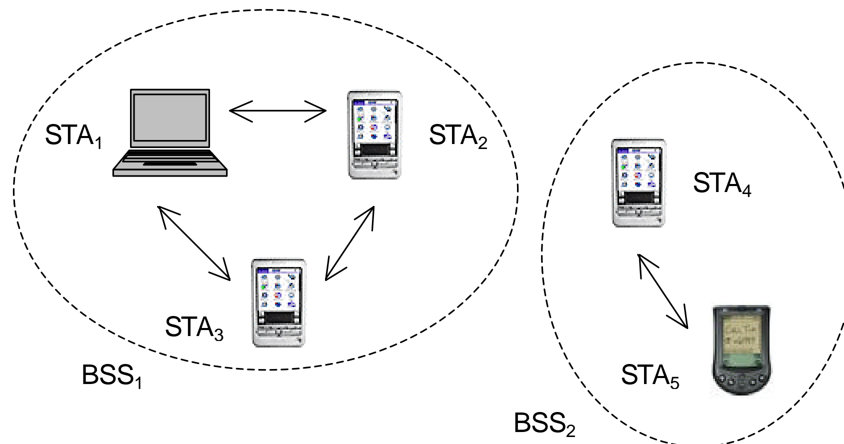

Figure 1.4 Architecture of IEEE 802.11 ad-hoc WLANs

In this case, a BSS comprises a group of stations using the same radio frequency within the same coverage area. Stations STA_1, STA_2 and STA_3 are in BSS_1, while STA_4 and STA_5 are in BSS_2: for example, this means that STA_3 can communicate directly with STA_2 but not with STA_5. Different BSSs can either be formed by staying in distant locations or by using different carrier frequencies; in the second case the BSSs could overlap physically.

We analyse briefly some details about the physical and medium access layer. The first one is subdivided into the physical layer convergence protocol (PLCP) and the physical medium dependent sublayer (PMD) (see Figure 1.5).

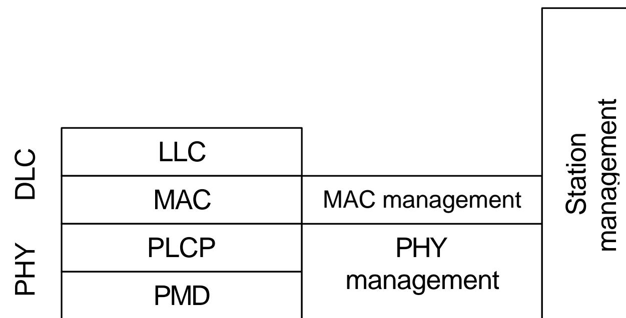

Figure 1.5 Detailed IEEE 802.11 protocol architecture and management

The PLCP sublayer provides a carrier sense signal, called clear channel assessment (CCA) and provides a common PHY service point (SAP) independent of the transmission technology; the PMD sublayer handles modulation and encoding/decoding of signals. IEEE 802.11 supports three different physical layers: one based on infrared and two on radio transmission (mainly in the ISM band at 2.4 GHz, which is available worldwide). The basic tasks of the MAC layer comprise medium access, fragmentation of user data and encryption; it also addresses the issues related to the association and re-association of a station to an access point and the roaming between different access points. Furthermore, it controls authentication mechanisms, encryption, synchronization of a station with an access point, and power management (to save battery power). We do not analyse these issues in detail, because they are beyond the scope of this thesis; however we underline that designers and developers must be aware of the services provided by these underlying technologies (for instance, with respect to security issues).

It is worth noticing that the network layer (for example IP) looks the same for a wireless and a wired LAN, because the upper part of the data link control layer, the logical link control (LLC), covers the differences of the medium access control layers needed for the different media. Figure 1.6 shows a typical example of a common internetworking scenario, a wireless LAN connected to a wired LAN via a bridge.

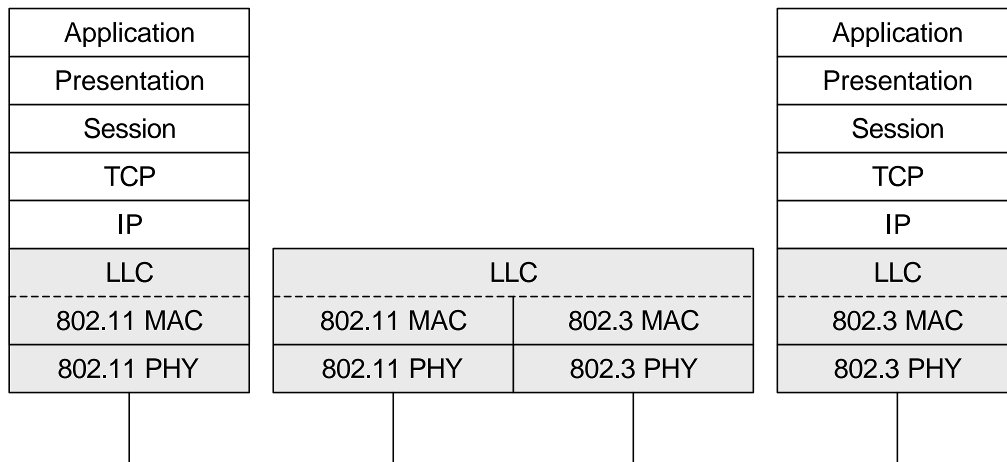

Figure 1.6 Example of IEEE 802.11 protocol architecture and bridging, using TCP/IP

1.3.4 Bluetooth

Bluetooth [Blu02] is a technology specification for low-cost, short-range radio links between mobile devices. It has a solid industry support, with more than 2,000 companies in the Bluetooth Special Interest Group (including, for instance, Ericsson, Nokia, Motorola, Toshiba, IBM, and Microsoft). It is named after Harald Blåtand ("Bluetooth"), King of Denmark from 940 to 981, who unified Denmark and Norway.

Bluetooth operates in the worldwide 2.4 GHz unlicensed band: because of the unrestricted access to this frequency range, Bluetooth devices are exposed to a high level of interference from unknown proprietary products and in particular other WLANs devices (for example belonging to 802.11 group). A frequency-hopping/time-division duplex scheme is used for transmission with a fast hopping rate of 1600 hops per second. The time between two hops is called a slot, which is an interval of 625 µs, thus, each slot uses a different frequency; on average, the frequency hopping sequence uses each hop carrier with equal probability. All devices using the same hopping sequence with the same phase form a Bluetooth *piconet*.

With transmitting power up to 100 mW, Bluetooth have a range of up 10 m (or even up 100 m with special transceivers). Relying on battery power, a device

cannot be in active transmit mode all the time; Bluetooth specifications defines several low-power states for the device (for example, if it is currently not participating in a piconet, it is in *standby* mode).

As concerns the medium access, one Bluetooth unit acts as the master of the piconet, whereas the other units acts as slaves; up to seven slaves can be active in the piconet. In addition, many more slaves can remain locked to the master in a so-called *parked state*. These parked slaves cannot be active on the channel, but remain synchronized to the master. The master controls both for active and parked slaves, the channel access.

Multiple piconets with overlapping coverage areas form a scatternet. Each piconet can only have a single master. However, slaves can participate in different piconets on a time-division multiplex basis. Moreover, a master in one piconet can be a slave in another piconet.

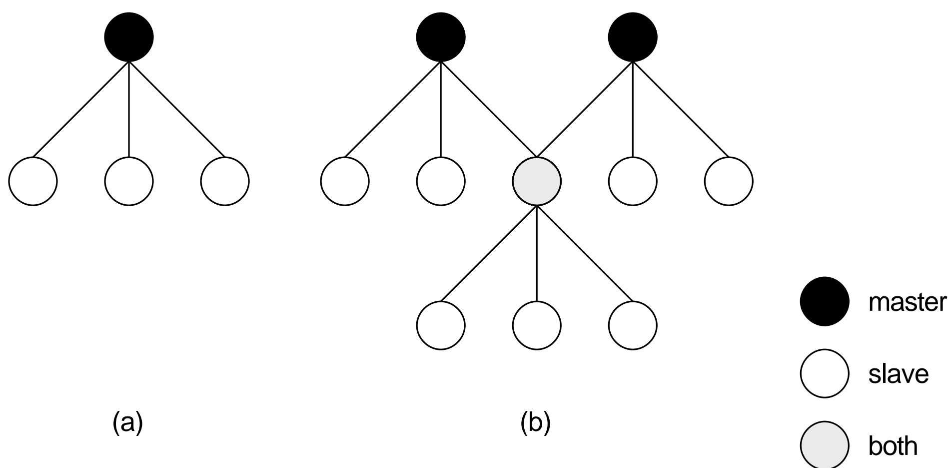

Figure 1.7 (a) Example of Bluetooth piconet topology.
(b) Example of BlueTooth scatternet topology

In the example in Figure 1.7, the scatternet consists of three piconets, where you can see a node (the grey one) that acts contemporaneously as master and slave.

A Bluetooth system is managed by a link manager which handles the authentication parameters, carries out link setup and release, configure links etc.; it uses the services provided by the link controller that provides the basic service of sending and receiving data and requesting names of other devices.

Furthermore, it sets up connections, triggers authentication and negotiates link modes, which could be data or voice.

As concerns security, the main features include a challenge response routine for authentication, a stream cipher for encryption and a session key generation; each connection may require an one-way, two-way or no authentication using the challenge-response routine; the possible key lengths are 0, 40 or 64 bits.

In march 2002 the Standards Board of the Institute of Electrical and Electronics Engineers Standards Association (IEEE-SA) has approved the IEEE Standard 802.15.1 ("Wireless MAC and PHY Specifications for Wireless Personal Area Networks (WPAN)" [IEEE02], which is adapted from portions of the Bluetooth wireless specification.

1.4 Mobile distributed systems

The technologies that we have described in the previous paragraphs have a great influence on the design of present distributed systems. In fact, they enable new forms of computation that cannot be described using the traditional paradigms; now we provide a classification of existing distributed systems in relation to these emerging technologies.

1.4.1 A classification of distributed systems

First of all, we can give a general definition of distributed system as a collection of independent entities located in different places that cooperate to some global result. This definition is suitable for both fixed and mobile distributed systems.

The traditional systems are a collection of fixed hosts, permanently connected to the network via stable links, executing in static context; in other words locations almost never change, hosts can be added, deleted or moved, but the frequency at which this happens is not relevant. Services may change as well, but the discovery of available services is usually and easily performed by the registration of service providers in a well-known location server.

With respect to mobile ones, we can operate a distinction between *nomadic* and *ad-hoc* mobile distributed systems. A *nomadic* system [Kle95] [Kle96b] is usually composed of a set of mobile devices and a core infrastructure; they move from

location to location, while maintaining a connection to the fixed network. The load of computation is mainly carried on the backbone; services are mainly provided by core network to mobile clients. A particular characteristic of this type of systems is that disconnections are also allowed and mechanisms for transparent reconnection and synchronization are provided.

Ad-hoc mobile distributed systems consist of a set of mobile hosts, connected to the network through links that are usually characterised by high bandwidth variability. The execution context is also extremely dynamic: hosts may come and leave very rapidly and in an unpredictable way. They differ from traditional and nomadic essentially in that there is not the presence of a fixed infrastructure.

In a mobile scenario service lookup is more complex, especially in the case of ad-hoc environment: broadcasting and multicasting are the usual ways of implementing service advertisement, even if we have consider power expenditure during send and receive operations and we have to try to avoid the problem of possible congestion of the network caused by message flooding.

The main non-functional requirements of a traditional distributed system are scalability, openness, heterogeneity, fault-tolerance, resource sharing and security. In a nomadic distributed system we have to deal with the same issues, but, for example, heterogeneity is complicated by the fact that different links, fixed and wireless, are present at the same time. Ad-hoc distributed systems are more problematic; for example, providing scalability is very difficult (i.e. routing is very complex in ad-hoc networks with a large number of hosts because of the dimension of routing tables). Moreover, for this class of systems, given the highly dynamic structure of the network, disconnections have to be considered the norm rather than exception.

1.4.2 Physical and logical mobility

It is worth noticing that the classification given in the previous paragraph is only referred to the possibility that a host running an application is able to move, but we have also to consider the case of fixed hosts with a support for mobile code. In fact, in general, it is possible to define two forms of mobility: the first one is physical mobility that is related to the movement of the hosts within the physical

space; the second one is logical mobility that allows the code and possibly also the state of an executing program to be migrated in part or as a whole at run-time.
We use the term mobile computing middleware to refer to the class of systems that exploit physical mobility and mobile code middleware to the class of systems that support code mobility. However, one of the current research issues is the definition of a unifying model of treatment of both types of mobility.

Chapter 2
Mobile middleware

In this chapter we will analyse mobile middleware systems, discussing the issues that they address, the existing solutions and the current research directions, in order to define the conceptual background of this research area.

2.1 Middleware systems

Middleware [Ber96] [CamCK99] [Bak02] is a class of software technologies designed to manage the complexity and the heterogeneity inherent in a distributed system. In other words, its main goal is to enable communication between distributed components; in order to achieve this, it provides application developers with a higher level of abstraction built using the primitives of the network operating system. In fact, a fundamental aspect of middleware is the provision of location transparency and independence from the details of communication protocols, operating systems and computer hardware. If it is very hard to use a computer without an operating system, programming a distributed system is in general much more difficult without middleware, especially if we have to deal with heterogeneity.

The term middleware first appeared in the late 1980s to describe network connection management software, but it did not come into widespread use until the mid 1990s due to the increasing success of network technologies. By that time middleware has evolved into a much richer set of paradigms and services; in fact, at the beginning, the term was associated mainly with distributed relational database systems. However, concepts similar to today's middleware previously went under the names of network operating systems, distributed operating systems and distributed computing environments.

Cronus [MacTB82] was probably the major first distributed object middleware system; it was designed essentially as a base for the development of large-scale

distributed heterogeneous applications. Although the architecture of the system is object-oriented, this is largely hidden from application developers; most of the details of the implementation of distributed applications are provided by a combination of code automatically generated from an interface specification, library routines, and system components.

Remote procedure calls (RPC) abstraction [BirN84] can also view as an example of middleware service, since they extends local procedure calls hiding location. Early RPC systems that achieved wide use include, for example, those by Sun in its Open Network Computing (ONC) [IETF01].

During the past years, middleware technologies have been successfully used in industry. Some examples are object-oriented technologies like OMG CORBA [Pop98], Microsoft COM [Rog97] and Sun Java RMI [PitM01] or message-oriented technologies like IBM MQSeries [WacAS99].

Although these systems are very successful in fixed environments, they might not to be suitable in a mobile setting for their specific characteristics, which we have described in the first chapter. Academic and industrial researchers have been and are working to design middleware targeting the mobile setting; in this chapter, we present a reference model in order to analyse some different existing solutions.

2.2 A characterisation of middleware

It is possible to classify middleware systems on the basis of *computational load* they require to execute, the *communication paradigms* they support, and the kind of *context representation* they provide to applications [MasCE02]. We can consider the summarizing scheme in Figure 2.1.

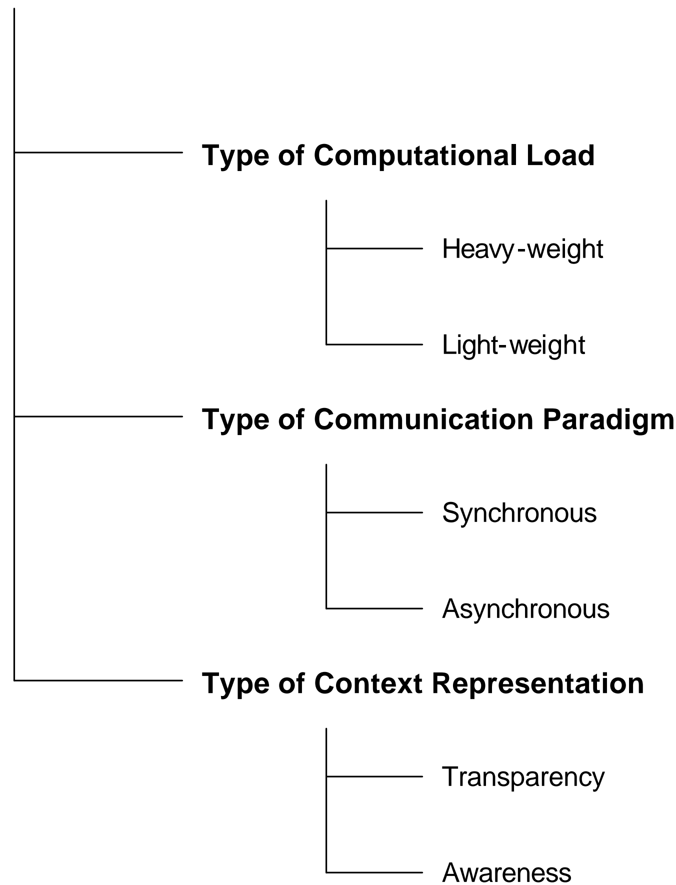

Figure 2.1 A possible characterization of middleware systems

Now we detail this possible classification, analysing each characteristic showed in Figure 2.1.

2.2.1 Type of computational load

The computational load depends on the set of system requirements; since the main purpose of any middleware is to facilitate communication, we can distinguish systems considering their reliability. In fact, for example, it is more expensive to guarantee that a request will be executed *exactly once*, instead of providing only *best-effort* reliability.

We may consider replication as another illuminating example. It is widely used in order to achieve fault tolerance and to improve scalability, but it requires a lot of resources and it generally introduces remarkable overhead; depending on how

consistent the replicas are, the computational load varies accordingly. For example some applications do not require a strict consistency of distributed replicas or this cannot be guaranteed (i.e., mobile applications with the possibility of disconnected modification of data), whereas others need a support for transactions (i.e., financial applications).

We usually refer to a system that requires a large amount of resources (e.g., CPU, main memory, network bandwidth, etc.) with the term *heavyweight* and we use the term *lightweight* to define one that presents the opposite behaviour. In general, different computational loads are associated to different qualities of service; for this reason we have to choose the most suitable system, evaluating this trade-off and considering the requirements of our application scenario. In fact quality of service is a fundamental issue in relation to today's new applications (e.g., video on demand or real-time financial trading) that are often based on middleware systems.

2.2.2 Type of communication paradigm

Middleware systems usually support two types of communication paradigm, *synchronous* and *asynchronous* [Emm00]. With the first term, we refer to systems where the client is blocked while the server executes the requested operation, while with the second we identify the system where the clients show the opposite behaviour. The asynchronous paradigm is usually preferred in order to increase system performance, especially in the case of single-threaded server.

It is possible to relax these definitions considering other cases; for example, in the *oneway* model requests return control to the client immediately and so the client and server do not synchronize. This pattern can be used when the client does not need to wait the completion of its request, since its state does not depend on this.

Another example may be deferred synchronous paradigm; with this term we refer to the systems where requests give control back to the client as soon as the middleware has accepted them and clients poll for results later. Therefore, this pattern can be used in the cases clients depend on the result of requested operations. A remarkable drawback of this approach is that clients have the responsibility of synchronizing with the server to obtain the result, since this may not be available when it is requested by a polling operation.

Instead, according to the asynchronous model, clients regain control as soon as the middleware has accepted their requests; a service is performed a server and, when this has finished, it calls an operation of the client to transfer the result (*callback*).

2.2.3 Type of context representation

Another important parameter related to the execution context can be identified in order to define a classification of middleware systems. In fact, it is worth noticing that middleware can interact with the underlying network operating system and collect information about, for instance, bandwidth availability, communication latency, actual host location (in the case of mobile devices), available remote services and so on. Information about the context may be available (we refer to this case with the term *awareness*) or not (we refer to this case with the term *transparency*) to the above application layer.

In the first case, information about the execution context (or a part of it) is available to running applications that, thus, are able to take decisions exploiting this knowledge. In the second case, context information is exclusively used by the middleware and not provided to the above applications; for example, if the middleware may discover network congestion, it can redirect access data requests to another portion of the distributed system. Considering the same example, using a context-awareness based approach, applications can decide to redirect request, because they are *aware* of the network congestion.

Due to the complexity introduced by the mechanisms that are necessary to provide context-awareness, middleware for distributed systems usually follow transparent strategies. In a fixed system this is possible because the lack of information is generally compensated by the abundance of resources. In a mobile scenario context-awareness plays a fundamental role and it is a key characteristic of some existing systems that we will analyse later in this chapter.

2.3 Middleware for fixed distributed systems

We now discuss some key aspects of middleware for fixed distributed systems with respect to the conceptual model presented below. Our aim is to understand how the scenario (fixed or mobile) influences design issues.

Wired distributed systems are generally designed and run on powerful machines characterized by a large and constant availability of resources; therefore, it is worthwhile exploiting these (e.g., fast processors, storage capacity, broad bandwidth, etc.) in order to deliver the best quality of service to applications; furthermore, it is also possible to fulfil other requirements like fault tolerance or resource sharing easily.

Fixed distributed systems are often permanently connected to the network through stable links; this means that the sender of a request and its receiver (in other words, the component that asks a service and the component that deliver it) are usually connected at the same time and for the whole interaction; therefore the typical form of communication is synchronous. In this scenario, the case of a disconnected host is considered an exception and middleware may also deal with this kind of event simply reporting a system fault.

We can also find systems that use asynchronous mechanisms, for example the so-called message oriented middleware; it is worth underlining that this form of communication has recently been introduced in CORBA specifications as well.

The execution context of a fixed distributed system is generally static; in fact the locations of hosts and the network topology seldom change. This fact enables, for example, complex mechanisms like replication in order to increase fault tolerance. The middleware can transparently decide to create replicas of data and to redirect access requests in case of a network or a host failure. Therefore, programmers do not have to deal with this kind of problems and they can concentrate, instead, on the real problems of the application that they are developing. In some cases, middleware is also able to re-synchronize inconsistent data replicas that are stored in different hosts. This process can be performed in a transparent way, hiding the complexity of this operation from the user but in this case it not possible to exploit application-specific knowledge in order to deal with conflicts. This is an effective approach, but it is lacking in expressiveness; for this reason some systems can also provide mechanisms in order to influence and modify this process according to application requirements; in other words, these middleware platforms exploit transparency and at the same time provide mechanisms in order to specify policies.

We can summarize these design aspects related to middleware for fixed distributed system by the following table:

Design aspect	Solution	Motivation
Computational load	Heavy computational load	Presence of resource-rich fixed devices
Communication paradigm	Synchronous communication (with the exception of message-oriented middleware)	Permanent connection
Context representation	Transparency	Static context Availability of resources

Table 2.1 Design aspects of middleware for fixed distributed systems

Now we consider three groups of systems for a fixed scenario: object-oriented, message-oriented and transaction-oriented middleware; their differences can be explained by their initial goals. Other categories can be identified, like middleware for scientific computing, but we do not consider them because they are not related to the mobile setting. Furthermore, this classification is not rigid; there is in fact a trend of merging these categories together (i.e., CORBA Object Transaction Service). We provide this description to analyse the impact of these traditional types of middleware in mobile scenarios.

2.3.1 Object-oriented and component middleware

Object-oriented middleware supports communication between distributed objects, in other words a client object can request the execution of an operation from a server object that may reside on another host. It evolved more or less directly from the idea of remote procedure calls. Products in this category include implementations of OMG CORBA [Pop98] (like IONA's Orbix [IONA02]), Microsoft COM [Rog97], Java/RMI [PitM01] and Enterprise JavaBeans [Sun02a].

Despite the great success of these technologies in building fixed distributed systems, their applicability to a mobile setting is restricted because of the heavy computational load required by these systems and the principle of transparency

that has driven their design and that prevent any forms of awareness. However some experimental systems about porting of CORBA to mobile settings has been developed, for example ALICE [HaaCC00] that we will analyse later in this chapter.

2.3.2 Message-oriented middleware

Message-oriented middleware supports the communication between distributed systems by exchange of messages; the products in this category include, for example, IBM's MQSeries [WacAS99] and Sun's Java Message Service [Sun02b].

It is usually used when reliable, asynchronous communication is the dominant form of distributed system interaction. Client components use these systems to send a message in order to request the execution of a service to a server component; the content of the message includes service parameters; another message is sent to the client from the server to transmit the result of the service. Exploiting this paradigm, it is possible to de-couple clients and servers in order to design more scalable systems.

Message-oriented middleware can also support multicasting, in other words it can distribute the same message to multiple receivers in a way that is transparent to client; by these primitives the implementation of event notification and public/subscribe architectures is made easier.

Fault-tolerance is achieved by implementing message queues that store messages temporarily on persistent storage: the sender writes the message into the queue and if the receiver is not available due to a failure, the message is kept until the receiver is ready to read it.

Messaging systems are classified into different models; the most common are publish/subscribe, point-to-point, request-reply. Publish/subscribe messaging is usually used when multiple applications needs to receive the same message. It is a distributed event-based paradigm: an object that generates events publishes the type of events that it will make available for observation; objects that want to receive notifications from an object that has published its events subscribe to the types of events that are of interest to them. When a publisher generates an event, subscribers that expressed an interest in that type of event will receive

notifications. When a process needs to send a message to another process, point-to point messaging can be used; a host may only send, only receive or send and receive messages. Request-reply messaging is used when an application sends a message and expects to receive a confirmation; this is the standard synchronous object-messaging format; it is often defined as a subset of one of the other two models.

These middleware systems require resource-rich devices, especially in terms of amount of memory in which to store persistent queues of messages received but not already processes: however it is possible to consider this model in a mobile setting; in fact some adaptations of Java Message Service for mobile devices have recently been developed (see paragraph 2.4.1.6). Furthermore, context-awareness is usually not provided by this class of systems.

2.3.3 Transaction-oriented middleware

Transaction-oriented middleware is often used with distributed database applications; it supports transactions using the two-phase commit protocol [Gra78]. The products in this category include, for instance, IBM's CICS [IBM02a] and BEA's Tuxedo [Hal96] these systems are essentially based on the aforementioned protocol.

Many transaction-oriented middleware systems use distributed transactions to maintain replicated databases on different servers; updates are synchronized and load balancing is used to direct queries to the least loaded replica. Nowadays support for transaction is also provided in a large number of object-oriented middleware systems (for example, the Open Group have adopted a standard for distributed transactions called Open DTP [Open02]).

This type of middleware supports both synchronous and asynchronous communication across heterogeneous hosts achieving high reliability; however, guaranteeing the atomicity property introduces considerable overhead.

Transaction-oriented middleware is not suitable for mobile systems because of its characteristics; for example, as we have discussed previously, transactions introduce an overhead that is not tolerable for mobile devices with scarce resources.

2.4 Middleware for mobile distributed systems

The existing nomadic and ad-hoc middleware differ in various aspects, however we can point out some common choices made by the designers of these systems.

Due to resource limitations of devices, heavyweight middleware systems optimised for powerful machines are usually not suitable for a mobile scenario. The assumption of keeping replicas always synchronized is usually relaxed; in other words, in order to reduce the overhead due to inconsistency management, we might allow the existence of diverging replicas, which eventually will be reconciled.

With respect to communication models, we can consider the case of a mobile device that connects to a network to access data; during the delivery of this service a disconnection may happen, for example, because the user has entered an area with no radio coverage. Therefore, we can understand that an asynchronous form of communication is necessary. For example, it might be possible for a client to ask a service, disconnect from the network and collect the result in a subsequent reconnection.

Furthermore, mobile connectivity is highly variable in performance and reliability: as we have discussed before, for example, inside some building a mobile user may benefit from large bandwidth, but outdoors it may have an unreliable wireless connection. The solution according to many researchers is adaptation and context-awareness [Sat96b].

We can summarize these design aspects related to middleware for mobile distributed system by the following table:

Design aspect	**Solution**	**Motivation**
Computational load	Light computational load	Mobile devices (characterised by scarce resources)
Communication paradigm	Asynchronous communication	Intermittent connection
Context representation	Awareness	Extremely dynamic context

Table 2.2 Design aspects of middleware for ad-hoc and nomadic distributed systems

2.4.1 Traditional middleware applied in a mobile environment

2.4.1.1 Overview

It is possible to find different examples of traditional middleware systems used in the context of mobile computing. We will analyse a small number of cases of adaptation of object-oriented and message-oriented middleware to mobile context. The main problem with the most part of object-oriented systems is that they are based on synchronous communication primitives that are not suitable for mobile applications. Moreover the computational load of these systems is generally high and it is very difficult to respect the principle of transparency.

Object-oriented middleware has been adapted to mobile settings mainly to allow interoperability between mobile and fixed hosts. Many researchers have been investigated possible adaptations of IIOP (Internet Inter-ORB Protocol), an essential part of CORBA specifications, to a mobile environment. IOOP is the implementation of GIOP (General Inter-ORB Protocol) for transport with TCP; it standardizes how TCP connections are established. In general, the GIOP defines the contents and the structure of a number of message types for ORB interoperability; it also assumes the availability of a connection-oriented protocol (the different instantiations of GIOP adapt these to particular protocols).

We now describe some solutions, starting from the examples of adaptation of traditional object-oriented middleware such as CORBA or Java Messaging Service to mobile platforms and also considering systems targeting completely ad-hoc scenarios.

2.4.1.2 The ALICE project

ALICE (Architecture for Location-Independent Computing Environments) [HaaCC99] [HaaCC00] is an experimental system developed at Trinity College, Dublin that has been designed to offer general support for distributed object-oriented applications operating in mobile environments. The main component of the architecture is the *mobility gateway*, an advanced proxy mechanism operating on the border between a mobile device and its point of connection to the Internet;

it performs dynamic address translation, client redirection and connection tunnelling, allowing standard clients and servers that run on fixed Internet hosts to interact seamlessly with clients and servers on mobile hosts.
The first version of ALICE was specific to CORBA; it allowed CORBA objects running on mobile devices to interact transparently with objects hosted by off-the-shelf CORBA implementations; another important characteristic was the possibility for server objects to reside on mobile hosts without relying on a centralised location register to keep track of their locations. The second version maintains these features and adds support for disconnected operations in the form of caching of server functionalities on the client side. It also supports other distribution infrastructures, such as SOAP, DCOM and Java RMI.

2.4.1.3 The DOLMEN project

The Dolmen project [Raa97] examines CORBA-based object communication in the context of wireless networks and host mobility. It uses the interoperability mechanisms introduced in the CORBA 2.0 specification to support host mobility in a transparent way. It applies the concept of interoperability bridges, described in the CORBA 2.0 architecture; the Fixed Distributed Processing Environment Bridge (FDBR) serves as an access point for mobile hosts. A Mobile Distributed Processing Environment Bridge (MDBR) connects the local ORB of a mobile host to the fixed network. The MDBR and FDBR perform also location management functions and handover, enabling host mobility. A Light Weight Inter-ORB Protocol (LW-IOP) between the FDBR and MDBR was defined to enhance reliability and performance of object communication in the mobile environment, since it defines efficient message formats and a compressed data representation.

2.4.1.4 Mobiware

Mobiware [AngCKL98] is a middleware system developed at Columbia University; the aim of this project is to provide interfaces and algorithms for enabling adaptive mobile multimedia applications. It is built on CORBA and Java distributed object technology; it runs on three types of host: mobile devices, wireless access points and mobile-capable routers.

In Mobiware, mobile devices are seen as terminal nodes of the network connected to access points with roaming capability; the other fundamental part is the core composed of programmable routers and switches. Furthermore, it is worth noticing that Mobiware abstracts hardware devices and represents them as CORBA distributed objects.

Another important characteristic of this system is the presence of mechanisms in order to guarantee a definite quality of service. In fact adaptive techniques present an effective approach to deal with quality of service variability in wireless and mobile networking environments. Mobile applications need to be capable of responding to these variations: Mobiware provides a set of adaptive quality of service API based on a group of controllers (i.e., transport controller). Mobile applications use this API at the transport layer; two parameters are used to describe quality requirements, a utility function and an adaptation policy. The first one captures the adaptability to available bandwidth in terms of a utility curve; the second one describes the adaptive nature of mobile applications in terms of a set of policies (i.e., fast, smooth, after handoff, never), which represent how they are able to respond to instantaneous changes in bandwidth availability.

After the description of its aims, now we can briefly analyse the architecture of Mobiware. First of all, we can point out the separation between mobile signalling and adaptation management. At the transport layer, an active wireless transport supports the end-to-end transmission of audio, video and real-time data services based on an-adaptive QoS paradigm. The active wireless transport is object-based and it is built on a set of Java classes; the transport service binds active and static transport objects at mobile devices and access points in order to provide end-to-end adaptation. For example, static transport objects include rate control, flow control, resource control and buffer managements objects. These are loaded into the mobile device as part of the transport service creation process, while active transport objects can be dynamically dispatched to mobile devices and access points to support value-added QoS services.

At the network layer, a support is provided for the introduction of new mobile adaptive-QOS services; it is composed of a set of CORBA objects and adaptation proxies that operates at the mobile device, access points and at mobile capable routers.

We have described Mobiware since it represents a state-of-art research prototype; in fact, nowadays, it is more and more clear that middleware for mobile devices should not ignore the problem of the definition of the execution context; moreover, it provides adaptation that is probably the key point of the design of applications in a mobile environment.

2.4.1.5 Rover Toolkit

Another example of attempt to deal with the problem of frequent and intermittent disconnections is Rover Toolkit [JosTK97] that extends the traditional RPC paradigm with the introduction of enhanced buffering capabilities.
It is mainly based on two concepts, Relocatable Dynamic Objects (RDO) and Queued Remote Procedure Call (QRPC). An RDO is an object with a well-defined interface that can be dynamically loaded into a client computer from a server computer (or vice versa). QRPC is a communications abstraction that permits applications to continue to make non-blocking Remote Procedure Calls (RPCs) even when a host is disconnected. Interrupted client requests and server responses are exchanged after network reconnection.
The Rover toolkit permits disconnected hosts to update shared objects; consistency is provided by locking objects or by using application-specific algorithms. Every object has a *home server*: a mobile host can import objects and export updated versions to their home servers, where update conflicts are detected during the reconciliation process. Because Rover can employ type-specific concurrency control, some conflicts are resolved automatically; it also provides functionality for application dependent reconciliation.
In order to support these functionalities, the Rover runtime system consists of four key components in the client-side and in the server-side: the *Access Manager*, the *Object Cache* (only in the client-side), the *Operation Log* and the *Network Scheduler*. The first is responsible for all interactions among server-side and client-side applications; it serves requests for objects (RDOs), logs modifications to them, controls network access and manages the Object Cache of the client, where imported objects are locally stored. If these are modified, the changes are inserted into the Operation Log, which is transferred to the server-side by the Network Scheduler.

In anticipation of disconnections, useful RDOs are imported from the server; applications can provide prioritized prefetch lists based upon high-level user policies. For example, Rover Exmh, the TCL/TK based e-mail browser ported in the Rover framework, automatically prefetches the user's inbox folder and any recently received messages; it also imports folders that users explicitly select or frequently visit.

We can find a similar approach in Mobile DCE [SchBBK95] that exploits RPC mechanisms and it is based on the Distributed Computing Environment from the Open Software Foundation.

2.4.1.6 Java Messaging Service

Java Messaging Service [Sun02b] is a Java API that allows applications to create, send, receive, and read messages. As we have discussed before, messaging enables distributed *loosely coupled* communication; moreover, other important features of this technology are asynchronicity and reliability. The default semantic is at-most once, even if it is possible to use lower levels of reliability for applications that can afford to miss messages or to receive duplicate messages.

When Java Messaging Service was introduced in 1998, its most important purpose was to allow Java applications to access existing messaging-oriented middleware systems, such as MQSeries from IBM. Since its release, many vendors have adopted and implemented the Java Messaging Service API; now it is an integral part of the J2EE platform.

Now we analyse the architecture of a typical Java Messaging Service application. The central element is the *JMS provider*, which is a messaging system (a third-part component) that implements Java Messaging Service interfaces; JMS clients are the components, written in Java, that produce and consume messages, which are objects that are used to store information to transmit. There are five types of messages: bytes messages (streams of uninterpreted bytes), map messages (sets of name/value pairs), stream messages (streams of Java primitive types), text messages (streams of text) and object messages (serialized Java objects). We can also underline that it is possible to use XML [BraPS00], the de-facto standard for data exchanging, exploiting text messages. The specification describes other

components, but the discussion about these is not relevant for the scope of this thesis.

The JMS specification defines two different messaging paradigms: *point-to-point* and *publish-subscribe*. These two paradigms (also called *domains*) represent the most common models of messaging; as we have discussed before, both domains can be characterised by the type of destination for message delivery and the pattern of interaction between destination and client.

A point-to-point application is based on the concepts of message queues, senders and receivers. Each message is addressed to a specific *queue*, from which receiving clients extract messages. Queues retain all messages sent to them until they are consumed or they expire. The next figure shows the architecture of a point-to-point application.

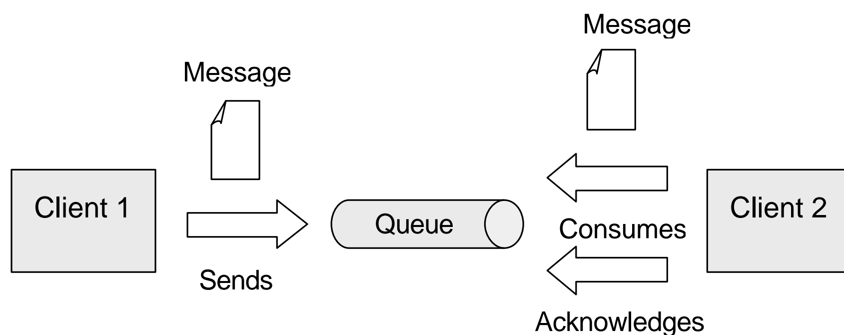

Figure 2.2 Point-to-point messaging in Java Messaging Service

We can underline other important concepts related to point-to-point paradigm: each message has only one consumer that can fetch it; the receiver may be active or not, when the client sends the message; furthermore, the receiver acknowledges the successful processing of a delivered message.

In a publish-subscribe application, clients address messages to a *topic*. Topics take care of distributing messages from the publisher to the subscribers; they retain messages only as long as they take to distribute them to current subscribers. Moreover, we can underline that each message may have multiple consumers. In Figure 2.3 we show the typical architecture of a publish-subscribe Java Messaging Service application.

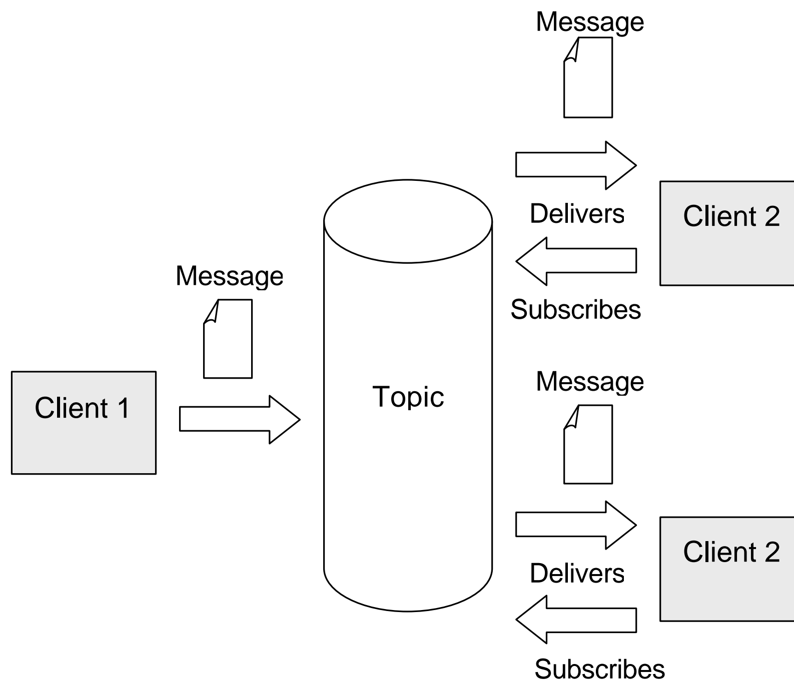

Figure 2.3 Point-to-point messaging in Java Messaging Service

Java Messaging Server can be efficaciously used in a mobile environment, because it supports point-to-point and publish/subscribe models that are typical mobile computing paradigms and it also offers an asynchronous communication mechanism that allows disconnected operations. However, the aim of Java Messaging Service is mainly to allow communication and, for this reason, some important aspects of mobile computing design, such as adaptation and data sharing, are not addressed.

2.4.2 Context-awareness based middleware

2.4.2.1 Overview

Mobile systems usually execute in an extremely dynamic context; for example bandwidth may not be stable, location may change, connection with a service provider may be lost. There is generally not static knowledge about all possible execution contexts.

In order to address this problem, context-awareness based middleware systems provide a support to applications that, by the interaction with underlying layer, become *aware* of their execution environment. A current issue in mobile computing research is how to enable this flow of information between middleware and applications. One of the most promising proposals is to exploit reflection [CapBMEG02]; it is a technique that first emerged in programming languages research and now it is applied to middleware systems in order to provide introspection, adaptation and reconfigurability. Applications can acquire information about execution context by the middleware and at the same type can modify their default behaviour (for example, they can decide a particular replication policy). Thus, it is possible to design a middleware core with a small number of essential functionalities; therefore this type of systems is also suitable for mobile environments with scarcity of resources. It is worth noting that nowadays is widely used in the design of operating systems and distributed systems; for example there are some interesting examples of the application of this concept in traditional middleware systems, such as DynamicTAO [Kon00] or OpenORB [Exo02].

2.4.2.2 Odissey

Odyssey [Sat96b] [Nob00] is a system designed and implemented at Carnegie Mellon University by Satyanarayanan and his group; it is a descendant of Coda (that we will describe in the paragraph 2.4.6.2) and the second version of Andrew File System (AFS) [Kaz88] and inherits much of their design philosophy. Scalability is the central focus of AFS, Coda adds the goals of high availability and application-transparent adaptation to support network or server failures and mobile computing, whereas Odyssey was essentially designed to support mobile information access.

Applications reside on mobile hosts and access/update data stored on remote hosts; there is the assumption that servers are more capable and trustworthy than clients; they can also specify a particular policy of adaptation, for example related to the access to multimedia data. In fact, according to the authors of this project, adaptation is fundamental because of the characteristics of mobile systems (scarcity of resource, variable connectivity, etc.); however, they define their

approach *application-aware adaptation* [SatNKP94]. The client is responsible for making adaptation decisions: operating systems can determine resource availability and provide ad-hoc mechanisms, however the application is the only entity that can properly decide how to adapt to a given situation. However, it is worth noticing that this is not the only solution to the problem of adaptation: for example, according to *application-transparent adaptation* approach, the operating system on the mobile node is wholly responsible for making adaptation decisions for applications.

Odissey can be thought as a set of operating system extensions supporting adaptive applications. The architecture of the client is shown in Figure 2.4.

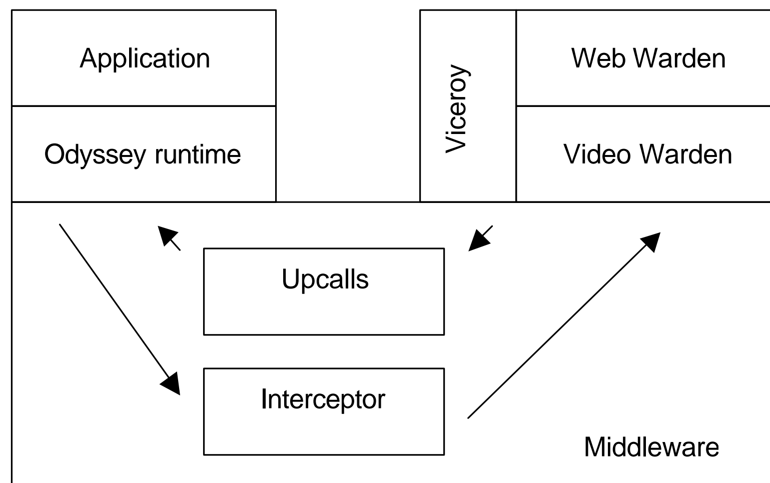

Figure 2.4 Odyssey client architecture

The client is composed of a Viceroy component, which represents a single point of resource control in the system, and Warden components, which implement the access method on objects of a certain type (for example MPEG files). A middleware module called Interceptor redirects file systems operations to the corresponding Wardens.

Applications are also able to specify which changes in resource availability are important to them; this is an optimisation to avoid a significant overhead related to this kind of notifications; in fact applications do not generally need to be informed of every small modification. Odyssey introduces a new system call, the

resource request, in order to express a window of tolerance. This request is passed to the Viceroy that will notify the application if an estimate of resource availability lies outside of its declared window trough an *upcall*. This carries with it the new value so that the application might properly react.

Data is stored in a shared, hierarchical name space composed of *tomes* that are conceptually similar to volumes in AFS and Coda. Each tome type identifies a unique combination of characteristics and policies such as naming, consistency, forms of graceful degradation and caching strategies. A *codex* distinguishes each type tome; examples might be “Unix”, “MPEGVideo”, “SQL” and so on. Therefore data are typed by its location in the name space; for example, video files must be in a video tome. Every codex is associated to a Warden, as we have discussed before.

Odissey has been designed to operate in a mobile environment, even if it presents some limitations. Firstly, the size of data that are transferred across mobile hosts may be too large for a mobile environment (for example, MPEG files), in presence of devices with scarce resources and expensive (and generally low-quality) connections. Secondly, the architecture is composed of thin clients and servers with different capabilities, so it is not suitable for an ad-hoc scenario. Finally, files are handled as simple byte-streams so there is a lack of semantics that does not permit an easy development of conflict detection and reconciliation at application level.

2.4.3 Location-awareness based middleware

2.4.3.1 Overview

Location is probably the most interesting aspect of the research about context-awareness, because there are a lot of possible related applications. For example global positioning systems are becoming popular devices in many cars; for instance, automobiles that move in different directions on the same road could share traffic information. Another interesting application is the possibility of accessing local tourist information by a portable device (e.g., a PDA).

To enhance the development of location-based services, middleware systems have been designed with integrated positioning technologies. We now analyse Nexus, an interesting example of middleware that provides a support to build location-aware applications.

2.4.3.2 Nexus

Nexus [FriKV00] is a research prototype developed at University of Stuttgart; one of the most interesting aspects of this system is the management of spatial models that represent physical world. A mobile user can move in a particular place (for example a square) and access resources (for example addresses of restaurants in the neighbourhood) in a defined area of visibility. The authors of this project called local information sources Virtual Information Towers: Figure 2.5 illustrates these concepts.

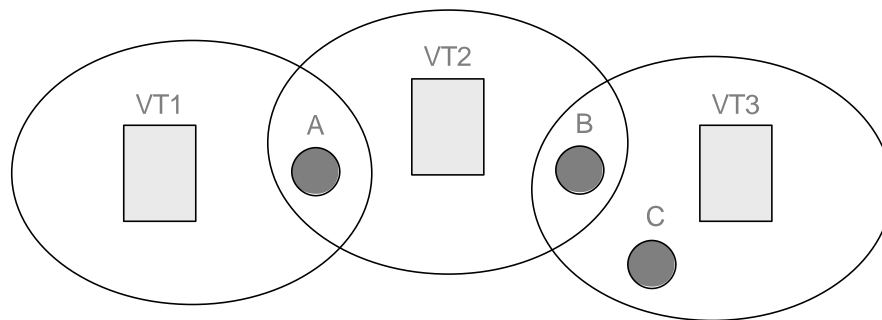

Figure 2.5 Virtual Information Towers in Nexus and their different areas of visibility

In the example that is shown in the illustration, mobile user A in this position can access information by connecting to Virtual Information Towers 1 and 2, while user B can retrieve it from Virtual Towers 2 and 3; in this configuration user C can access only information stored in Virtual Tower 3.

The architecture of this system is composed of Nexus stations (i.e., handheld computers or wearable devices), which additionally dispose of positioning and wireless communication facilities. To determine location in outdoor areas, the Differential Global Positioning System (DGPS) is used; indoor positioning is evidently based on different methods (mainly infrared signals, radio networks, and indirect techniques like image interpretation). A user interface component is

present on each mobile device; it contains basic functionalities to interact with the Nexus infrastructure and it supports adaptation mechanisms to devices with different levels of capabilities. The user component communicates with a distributed data management system that provides the access to information according to host position.

We can also point out that Nexus does not provide data sharing among the hosts; moreover, there is an infrastructure that manages information and provides it to mobile hosts and so ad-hoc network communication is not allowed.

2.4.4 Mobile code middleware

Even if XMIDDLE is a system designed for physical mobility, we now analyse in brief the essential concepts and features of mobile code middleware, since it is one of the most promising and interesting current research area. Moreover, in some recent systems, such as Lime [PicMR99], we can find an abstract model that includes both physical and logical mobility. In other platforms designed for logical mobility the support for mobile computing applications is also provided, usually as an extension of the core of the middleware. We refer to code mobility as the possibility to ship (and execute) a unit of code over the network; in other words, the binding between the components (code, objects, etc.) and the hosts where they are executing is not static. This approach can also be applied in a mobile setting; in fact, it addresses the requirements that derives from this extremely dynamic computational environment: for example, a mobile device may not have the known-how to interact with a server in a new location in order to obtain a particular service and, exploiting this approach, it can download the code that is necessary to perform its request directly from the server or from another host.

Essentially, we can find four general paradigms used in systems that exploit code mobility (according the classification proposed in [FugPV98]):

- *Client-Server interactions*: the request of a client triggers the execution of a unit of code in a server, which returns the result to the client.
- *Remote evaluation*: according to this approach, a host can send code to another one, have it executed and retrieve the result later.

- *Code on demand*: according to this paradigm, a host can request a unit of code from another one and, afterwards, execute it.
- *Mobile agents:* this approach differs from the previous, since it involves the movement of a whole computational component to a remote site, along with its code, its state and the resources that are required to perform the service [Pic01].

Furthermore, it is worth noticing that Java, TCL/Tk, Python and other interpreted or scripting language provide mechanisms enabling code mobility. Java is particularly well suited for mobile agents, because of its dynamic class loading, multithreading support, object serialization and reflection. Furthermore, another important aspect to be underlined is the wide adoption of Java and the consequent availability of systems provided with a Java Virtual Machine.

The interest of academic and industrial researchers in this field has produced a considerable number of platforms that support code mobility; for example, we may cite Aglets [LanO98], Ara [PeiS97], Voyager [Rec02] and SOMA [BelCS99] (furthermore, it is worth noting that this platform also provides a support for mobile computing applications [BelCS01]).

We do not analyse any system in detail, because it is beyond the scope of this thesis; however, later, we will discuss some features of Lime concerning logical mobility and, in the next chapters, we will present some issues of the design of XMIDDLE related to code mobility.

2.4.5 Tuple space based middleware

2.4.5.1 Overview

In a mobile setting, the characteristics of wireless transmission media (e.g. variable bandwidth) and frequent disconnections favour a decoupled style of computation. For this reason, in the last years, a new class of middleware systems has been developed exploiting a communication model that was initially proposed by David Gelertner in 1985 in Linda [Gel85] [AhuCG86] [CarG89], a coordination language for concurrent programming. This relies on an asynchronous communication mechanism based on a shared, associatively

addressed collection of *tuples* called tuple space. A *tuple* consists of a sequence of one or more typed data fields (for example `<"UCL", "London">` or `<"DEIS", "Bologna">`). The principal characteristics of this repository are persistency and globality: in fact every process can have access to the tuple space (hence the globality property) and at the same time the shared space exists independently from the existence of the processes (hence the persistency property).

Linda provides a very simple interface to the shared space; there are only three possible input/output operations. Tuples are added to a tuple space by performing an `out(t)` operation in it; after execution the tuple `t` is available on it; this update is performed atomically. Tuples are removed by executing `in(p)`. Since tuples are anonymous, their selection takes place through pattern matching on the tuple contents. The argument `p` is often called a *template* and its fields may contain either *actuals* or *formals*. The first are values, whereas the second are like "wild cards" and are matched against actuals during tuple selection. For example in `<"name", ?integer>` the first parameter is an actual, while the second is a formal. In case of multiple matching tuples, one is returned in a non-deterministic way. The `in` operation is blocking, in other words it suspends a process until a matching tuple appears in the tuple space; in case of multiple processes blocked to withdraw a tuple, when a process writes it, only one process, selected non-deterministically, will receive it. Another reading operation is `rd(p)` that proceeds identically to `in`, except for the fact that the tuple is copied rather than withdrawn from the tuple space; it is also blocking, similarly to `in`.

Therefore, the Linda coordination language relies on a generative communication model by which computational components can be flexibly coupled together. It is said to be generative, because generated tuples are explicitly withdrawn; they have independent existence in the tuple space and equally accessible to all processes within a space, but are bound to none. Communication in Linda is decoupled in time, since senders and receivers do not need to be present at the same time in order to exchange information and in space, since an associative addressing scheme for tuples rather than a physical one is used and so a globally shared data space is provided to all processes, regardless of machine and platform boundaries.

Thus, tuple space based middleware systems may be used in order to share information (for this reason, we analyse them in depth) or to coordinate a distributed computation.
In recent years, also due to the increasing interest in mobile computing, a significant number of systems have been developed; we consider four examples: Lime, TSpaces, JavaSpaces and L^2imbo.

2.4.5.2 Lime

Lime [PicMR99] is a Java-based middleware that provides a coordination layer based on tuple spaces; it tries to support specifically the complexity of the ad-hoc mobile environment, exploiting Linda-like operations for the communications among mobile hosts.
In Lime there is not the notion of global tuple space; this is distributed across mobile components of the systems; when they are in reach (for instance, mobile agents are on the same host or the communication between different mobile hosts that contains hosts is possible), the sets that are contained in the tuple spaces of the individual mobile components are *transiently* and *transparently* shared, forming a so-called *federated tuple space*.
The only entities that are active are agents: each mobile agent is associated to an *Interface Tuple Space (ITS)*, which is transferred when it moves from a place to another one. Agents may have multiple ITSs and also private ones that are not subject to sharing.
In Lime the notion of shared persistent and global space is represented by two different tuple spaces:

- host-level transiently shared tuple spaces (which are created on a host by the merging of the Lime tuple spaces of the mobile units that are present in this node);
- federated transiently shared tuple spaces (which are created across hosts by the merging of the host-level tuple spaces).

The process is dynamic; a reconfiguration takes place when an agent arrives or a host gets connected (*tuple space engagement*) and when an agent leaves or a host gets disconnected (*tuple space disengagement*). We illustrate this concept in Figure 2.6.

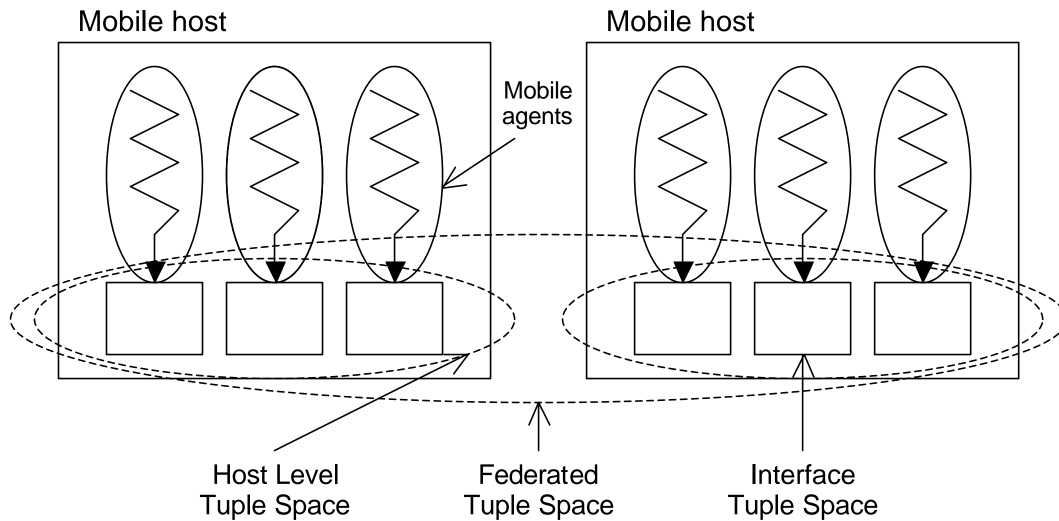

Figure 2.6 Lime model

We can underline that Lime applies the concept of transiently shared tuple space regardless of its nature (physical or logical). The authors of Lime introduce an extended definition of connectivity: mobile hosts are connected if they can communicate; this is not only related to physical conditions (for example hosts in the same radio coverage), but also to security or quality of service considerations, that can be represented by appropriate policies. On the other hand, mobile agents are connected if they are co-located or if they reside on hosts that are connected.
Since mobile environments are extremely dynamic, the reactivity to events is fundamental; therefore, Lime also extends tuple spaces adding the notion of reaction to changes. Events can be related to the physical environment (for example, bandwidth variations or disconnections) or to applications (for example, arrival or departure of a mobile agents); according to the Linda model, they are represented by tuples. The `in` operation provides a basic mechanism for dealing with events: their occurrence can be modelled by the appearance of a corresponding tuple in the space. However, this solution has some disadvantages; for example from a semantic point of view, there is not guarantee that the event is certainly caught by a process A interested in it, because, for example, another process B can be interested in it and perform the `in` operation before A. For this reason a more complex scheme is used in Lime; the authors introduce the notion

of reactive statement having the form `T.reactsTo(s, p)`, where `s` is a code fragment that is to be executed when a tuple matching the pattern `p` is found in the tuple space `T`. The execution of this operation registers the reaction with T; it is also possible the complementary de-registration. This event management strategy shows some complex aspects that we do not discuss, because they are beyond the scope of this thesis.

Moreover, in order to provide context awareness, information about the system configuration is made available through a read-only transiently shared tuple space called LimeSystem. Another important aspect to point out is that in Lime there is not the notion of adaptation.

2.4.5.3 TSpaces

TSpaces [WicMSF98] [IBM02b] is a network middleware system for ubiquitous computing developed at IBM Almaden Laboratory, enabling communication, computation and data management between hand-held devices. It exploits both tuple spaces concepts and database technologies; the implementation of TSpaces is in Java, so it is also portable and platform-independent.

TSpaces provides several types of services, such as:

- communication services
- database services
- URL-based file transfer services
- event notification services

The architecture of TSpaces is composed of clients and servers: a tuple space exists only on a TSpaces server that, in other hand, may host several spaces. We can distinguish two layers in a TSpaces server; the bottom layer comprises the basic tuple management and this is where tuple sets are stored and updated, while the top layer comprises the operator component, which is responsible for operator registration and handling.

There are some differences between TSpaces and the other tuple based systems: first of all, the behaviour of the middleware is dynamically modifiable, since new operators can be defined and new datatypes can be added; secondly this platform is able to provide functionalities similar to relational database systems. In fact, one of the differentiating features of TSpaces is that it builds an index on each

named field in a tuple. This feature enables clients to request tuples based solely on values in fields with a given name, regardless of the structure of the rest of the tuple or even the position of the named field. For example, an index query of the form ("Bologna", 123) will select all tuples of any format containing a field named "Bologna" with the integer value 123. It is also possible to specify a range of values to be found in the index.

Furthermore, TSpaces uses a terminology that is different from Gelernter's works, since in this case the `out` operation is called `write`, while the corresponding of the `in` operation is `take`. Some examples of interactions in TSpaces are showed in Figure 2.7.

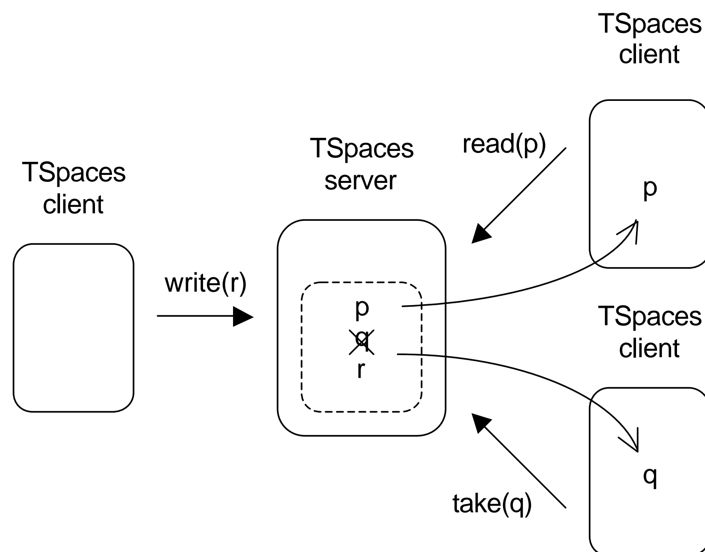

Figure 2.7 Examples of interactions in TSpaces

TSpaces also provides an access control mechanism; the architecture is similar to AFS, with a hierarchical approach. Each tuple space defines a group-based set of access control permissions that governs the creation and deletion of children tuple spaces, reading and writing of tuples and event registration.

In a server there are two system tuple spaces: the first, Galaxy, contains tuples that describe its contents; the second, Admin, stores access control permissions for each tuple. The communication between the client and the server is non blocking; if a client thread issues a blocking request, it is suspended after sending the request and is awakened when the response arrives; in this way, multiple threads

running on the same Java Virtual Machine can share a TCP/IP connection to a server. Another interesting characteristics are the presence of a HTTP server interface for debugging and maintenance purposes and a support for large objects through URL reference.

This middleware system addresses the requirements related to a mobile setting, but it is unsuitable for an ad-hoc environment, because there is a sharp distinction between clients and servers; in fact, while the first can be small devices like PDAs, the second requires higher computing capabilities, because they must be able to execute transactional operations and advanced queries.

2.4.5.4 JavaSpaces

JavaSpaces technology [Sun02c] is a lightweight infrastructure for dynamic communication, coordination and sharing of objects between Java-based components (clients and servers), exploiting a typical tuple space based architecture.

The Java programming languages has a remote method invocation system called RMI, layered on the Java platform's object serialization mechanism; it permits to marshal parameters of remote methods into a format that can be shipped across the network and unmarshalled in a remote server running a Java Virtual Machine. JavaSpace is essentially based on this technology and on an extensive use of object serialization. We can also point out that it is built upon the so-called Jini technology [Sun02d] that we will describe in brief later.

The JavaSpaces system design is strongly influenced by Linda in the sense that there is a virtual space for exchanging information (tasks, requests, etc) between providers (servers) and requesters (clients), but we can point out some relevant differences. First of all, while Linda system does not use rich typing, in JavaSpaces tuple entries are typed as Java objects, so they have methods (in other words behaviour) associated with them. As another consequence, JavaSpaces services allow matching of subtypes, so a template that matches for a certain type can also return a subtype.

There are four main types of operation:

- `write`, which adds a given entry (according Linda terminology, a tuple);

- `read`, which extracts an entry from a JavaSpaces service that matches the given template without removing it; if a match is found, a copy of the matching entry is returned; if there are no matching objects, then `read` may wait a user-specified amount of time until a matching entry arrives in the space.
- `take`, which extracts an entry from a JavaSpaces service that matches the given template, removing it from the space;
- `notify`, which asks a JavaSpace service to notify the object whenever entries that match the given template are added to the space.

By means of the `notify` operation, it is possible to use JavaSpaces to design an event-based system; in particular, using this primitive, a client can request to be notified when a specific tuple (representing an event) is written to the JavaSpace, according to the classic publish/subscribe model. In Figure 2.8 we show a possible example of events-based coordination using JavaSpaces. In this example T identify a template and C is a tuple that matches it.

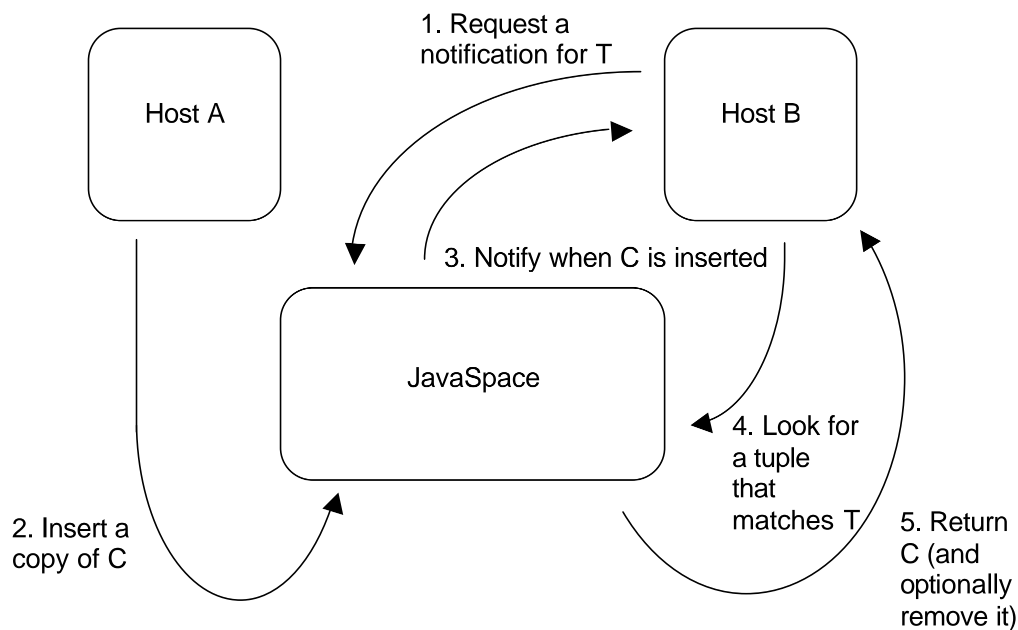

Figure 2.8 Example of event-based coordination using JavaSpaces

JavaSpaces also supports a transactional semantic, providing the classic ACID properties (atomicity, consistency, isolation and durability). As discussed for

TSpaces, for similar reasons, JavaSpaces is better suited for a ubiquitous environment than for an ad-hoc one. To conclude, we can sum the most remarkable differences between TSpaces and JavaSpaces in the following table:

TSpaces	JavaSpaces
Simple types and objects as tuple fields	Only serializable objects allowed
No replication, one server per TSpace	Servers can be replicated across nodes
Access Control Lists on users and groups	Protective partitioning using multiple TSpaces
Event Register invoked on all events	Notify() is only invoked for committed writes
Database indexing and range queries	Database facilities are not available
Downloadable operators	Fixed set of operators

Table 2.3 Comparison between TSpaces and JavaSpaces

2.4.5.5 L^{2}imbo

L^{2}imbo [DavFWB98] project, developed at the University of Lancaster (UK), is a middleware platform based on Linda model; however it includes significant extensions to support operations in mobile environments. It specifically addresses the quality of service issue.

Multiple tuple spaces may be specialised to meet application level requirements (for security, consistency or performance); furthermore, it is also possible to define an explicit tuple type hierarchy. Tuples may include QoS attributes with delivery deadlines, which indicates how long they should be available in the space.

The architecture of L^{2}imbo is composed of tuple spaces (all interactions between entities are based on these) and a certain number of agents that implement system application and computation and can migrate as necessary. In fact mobility of agents is naturally supported by the tuple space model: they simply stop interacting with the tuple space, move to another place and then restart their interactions.

The most significant types of agents that are defined in L^{2}imbo are three:

- *bridging agents*, which provide the means of linking arbitrary tuple spaces and controlling the propagation of tuples between the spaces;

- *QoS monitoring agents*, which control some key aspects of the system; for instance, *connectivity monitors* watch over the characteristics of the underlying communications infrastructure (i.e., throughput between the hosts), *power monitors* review the availability and consumption of power on a particular host (i.e., to enable applications to utilise hardware power functionalities) and *cost monitors* determine the cost associated with the current communications links between the hosts.
- *type management agents*, which provide facilities to the tuple space protocol to find sub-types that match supplied criteria and, at the same time, are the authority through which the sub-type relationship tuples generated by local applications are ratified for validity.

L^2imbo supports some mechanisms for achieving adaptation; the most important is that of filtering agents, which are a special form of bridging agents. They perform transformations on the tuples to map between different levels of QoS; for example they can be used to translate between different media formats or to reduce the overall bandwidth requirements from the source to the target tuple spaces.

For its characteristics (primarily for the effective support for disconnected operations) L^2imbo is well suitable for high mobile environments.

2.4.6 Data-sharing oriented middleware

2.4.6.1 Overview

The principal aim of data-sharing middleware system is to provide a support to access and modify data in a mobile environment, where disconnections are frequent and we have to deal with the problem of inconsistencies of different cached replicas. This has been a typical issue in distributed system research since its origin; a significant number of systems targeting this problem have been developed and in recent years some interesting prototypes for mobile setting have been designed.

The first works related to this area belong to file systems research field with the introduction of the problem of disconnected operations. In a fixed network,

computers could virtually always communicate; if a file server is unavailable or the network is inaccessible, individual workstations could be unusable until the problem is resolved; there is evidently some exceptions to this, for example some systems support partitioning and replication of essential services. In a mobile setting disconnected operations are very frequent. One of the first solutions to this problem was provided by Kistler and Satayanarayanan [KisS92] that extended Andrew File System to address the problem of providing support for disconnected operations caused by network failures and partitions or due to the unreachability of mobile devices.

In this paragraph we analyse some examples of middleware for data sharing; XMIDDLE belongs to this class of systems. We will describe it accurately in the next chapters. In Chapter 5 we will consider the problem of inconsistency management of replicated data and we will describe our solution with a comparison with existing systems. Furthermore, it is important to underline that tuple space based systems that we have examined in the previous paragraph are also used to provide data sharing in distributed environment.

2.4.6.2 Coda

Coda [Sat96] has been developed in a research project undertaken by Satyanarayanan and his group at Carnegie Mellon University in the 1990s and is now integrated with a number of popular UNIX-based operating systems such as Linux; it is a descendant of version 2 of Andrew File System (AFS) and it inherits from this the general architecture. In contrast to AFS where read-write volumes are stored on just one server, the design of Coda relies on the replication of file volumes to achieve a higher throughput of file access operations and a greater degree of fault tolerance; another difference is the presence of an extension of the mechanism used in AFS for caching copies of files at client computers to enable those computers to operate when they are not connected to the network.

In fact, the problem that Coda wants to address is that in a large distributed system it may easily happen that some or all of the file servers are temporarily unavailable for a network or a server failure or for a disconnection of the mobile client.

Coda nodes are partitioned into two groups: one consists of a relatively small number of centrally administered file servers (called Vice), the other consists of a larger group of workstations that give users and processes access to the file system (called Virtue). Every Virtue workstation hosts a user-level process called Venus, whose role is similar to that of a NFS client. A Venus process is responsible for providing access to the files that are maintained by the Vice file servers and for allowing the client to continue operation even if access to file is temporarily impossible.

Coda appears to users as a traditional UNIX-based file system; it supports most of the VFS operations; unlike NFS, it provides a globally shared name space that is maintained by the Vice servers; clients have access to this name space by means of a special subdirectory in their local name space (for example, */afs*). Whenever a client looks up a name in this subdirectory, Venus ensures that the appropriate part of the shared file system is mounted locally.

We now analyse two key aspects of Coda, the replication strategy and the operations during a disconnection. Coda allows file servers to be replicated; the unit of replication is the volume. The collection of servers that maintains a replica of a volume is known as Volume Storage Group (VSG). Because of possible failures or disconnections, a client may not have access to all servers that are included in a VSG. In other words, at any instant, a client that wishes to open a file in a certain volume can access a subset of these servers, known as the Available Volume Storage Group (AVSG). Normally, file access is similar to AFS: cached copies of files are supplied to the client by one of the servers in the current AVSG. Clients are notified of changes by a *callback promise* that is a mechanism for the distribution of updates to each replica. When a file is closed, copies of modified files are broadcast in parallel to all of the servers in the AVSG. If the AVSG is empty, we have the case of disconnected operation.

Coda uses an optimistic strategy for file replication; in fact it allows modifications of files to proceed when the network is partitioned or during disconnected operations.

The solution that is adopted for detecting and resolving inconsistency is based on a versioning scheme that is similar to that used in Locus [PopW85]: a server S_i in a VSG maintains a Coda Version Vector $CVV_i(f)$ for each file f. Each element of

CVV is an estimate of the modifications performed on the file that is stored in that server. If $CVV_i(f)[i]=k$ then server S_i knows that server Sj has seen at least version k of file f; $CVV_i(f)[i]$ is the number of the current version of f stored at server Si.
Now we analyse the process of reintegration after for example a partition recovery: if the CVV at one of the sites is greater than or equal to all corresponding CVVs at other sites then there is no conflict (automatic updating of the older replicas); instead, if the numbers of versions are different, Coda finds a conflict that needs to be repaired: in many cases, conflict resolution can be automated in an application-dependent way [KumS95]. The essence of the approach of the authors of Coda is to provide a framework for installing and invoking customized pieces of code called Application Specific Resolvers (ASRs); each ASR encapsulates knowledge related to its corresponding application. If the resolution succeeds, the system does not need the intervention of the user; if it fails, the user has to solve the conflict manually. Updates of files are performed by an ASR atomically. Users are able to select which ASR gets invoked for a particular application by the specification of policies.
It is worth noticing that we can find a similar approach in Ficus File System [HeiPGP92] [ReiKRSP94] that uses optimistic replication and provides facilities for automating the resolution of file conflicts. Like Coda, for example, it utilizes a rule-based approach for selecting a resolver. There are also important differences in the execution model of the resolution; for instance, since Ficus uses a peer-to-peer rather than a client-server model, this process can take place at any site.
Another interesting aspect of Coda is how a client can continue to operate during disconnections. Normally a client (Venus) is in the HOARDING state: it can interact with the server and fill its cache with data; a user can explicitly specify which files are essential for his work in a *hoard database*. If a disconnection takes place, the client enters the EMULATION state in which it uses the cached files. Since cache misses cannot be masked, they appear as failures to application programs.
The persistence of changes made during disconnection is achieved by an operation log, called the *change modify log*. Venus implements some optimisations to reduce the size of this log file: before a record is appended to it, the client checks if it cancels or overrides the effects of earlier records. For

example we can consider the creation of a file, followed by a store and an unlink operations; in this case all three records (and data associated to them) can be deleted.
When a reconnection occurs, the client enters the REINTEGRATION state in which it transfers the modifications to the server; if during reintegration some conflicts are detected, the system tries to resolve them, where possible, in automatic way, as we have described before. If the reintegration is successful, the new state is HOARDING, else the client is brought back into the EMULATION state. The next figure illustrates the state-transition diagram that describes Coda client behaviour.

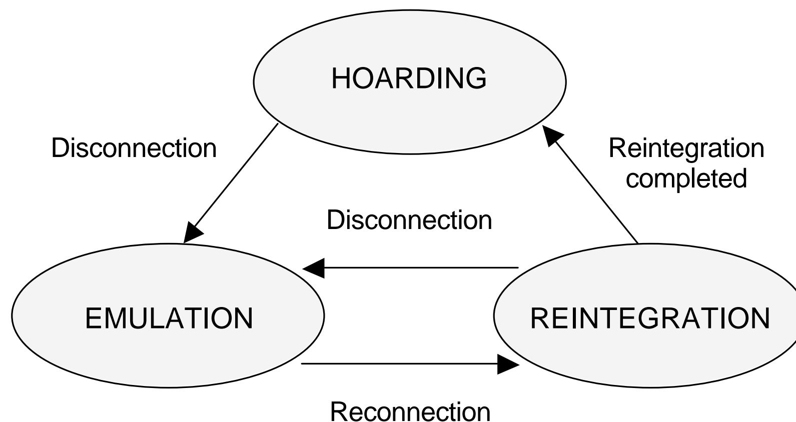

Figure 2.9 The state-transition diagram of a Coda client

2.4.6.3 Bayou

The Bayou project [TerTPDS95] developed at Xerox Palo Alto Research Center, provides a support for data-sharing among mobile users. The architecture is composed of *servers*, which store data, and *clients*, which read and write data managed by the first ones. In general, a server is a host that stores a complete copy of one or more collections of data items (a generic, non strictly relational, database) and it may also reside on a mobile device. We also underline that clients and servers may be co-resident on a host. For this reason, we can point out that the Bayou architecture differs from systems (such as Coda) that maintain a strong distinction between servers, which hold databases or file systems and are

usually more capable machines, and clients, which use only personal caches and resides on devices with scarce resources. This approach is similar to that is present in Lotus Notes [KalBHOG88] or in Ficus [HeiPGP92]. A possible configuration of a system based on Bayou is showed in Figure 2.10.

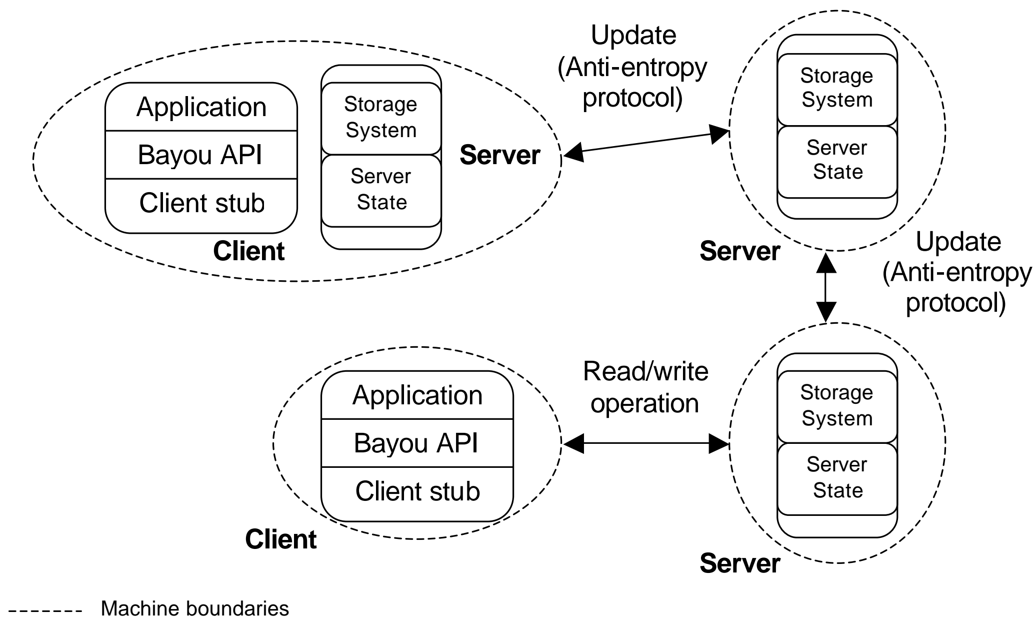

Figure 2.10 Bayou System Model

In order to achieve high availability, each data collection is wholly replicated at a number of servers; applications runs as clients and interacts with servers through the Bayou Application Programming Interface, which supports essentially two basic operations, read and write. A read-any/write-any approach is applied; in other words, users are able to read from and write to any copy of the database. Therefore, in this system we find a so-called *eventual consistency,* according the definition of the authors: Bayou system only guarantees that all servers that receive the same set of updates (writes operation), by means of a so-called anti-entropy protocol, have identical data contents. Furthermore, updates are initially marked as tentative; the system arranges that tentative updates are eventually placed in a canonical order and marked as committed. While updates are in the tentative state, the system may undo and reapply them; once committed, they remain applied in a determined order. This can be achieved by designating a *primary* replica manager, which might be, for example, a fast machine that is

generally available. Another possibility is to assign priority to each host. As usual, primary replica manager decides the committed order as that in which it receives the tentative updates. We can also point out that servers maintain logical clocks [Lam78] to timestamp new updates and to order them.

Bayou detects update conflicts in an application-specific manner; every update contains a dependency check and a merge procedure in addition to the specification of the operation. A replica manager calls the dependency check procedure before applying the operation; if a conflict is detected, Bayou invokes the merge procedure, called *mergeproc*, which resolves the inconsistency by producing an alternate set of updates that are appropriate in that particular case. Merge procedures are similar to mobile agents, because they are originated at clients and are passed to the servers, where they are executed. Another aspect to point out is the deterministic behaviour of mergeprocs, since Bayou replica managers are state machines; they also may fail to find a suitable alteration of the operation and in this case the system indicates an error [TerPST98].

Moreover, it is worth noting that while Coda locks file volumes when conflicts have been detected but not yet resolved, Bayou allows replicas to remain always accessible: this is a cause of possible inconsistencies and it represents a drawback of this system.

The Bayou model is very expressive, even if there is a greater complexity for application programmers, because they must supply dependency checks and merge procedures. In general, it is very complex to define them, given the large number of conflicts that need to be detected and resolved. Another disadvantage of this system is its client-server architecture; although in principle the coexistence of client and server on the same host is possible, in practice data are stored only in a certain number of more powerful devices; for this reason Bayou is most suited for nomadic rather than ad-hoc applications.

2.4.7 Service discovery in mobile computing middleware

2.4.7.1 Overview

Service discovery [CouDK01] is one of the most challenging issues in a mobile environment, because of the extreme dynamism of this context. In a classic client-server model, clients usually know the name of the server that can provide a certain type of service *a priori*; another widespread approach is to utilize a centralized name server (i.e., DNS). A third possibility, that is the most suitable for a mobile setting, is to use a service discovery infrastructure.

In a nomadic environment the problem can be solved in an easy way, because it is possible to concentrate all information about services in the infrastructure, where mobile hosts register and ask for them. In an ad-hoc network setting, discovery cannot be centralised in a single point and thus this process become very complex and expensive from a computational point of view.

Most of the ad-hoc systems that we have described in this chapter have their own discovery service. In Lime [HanR02], hosts continuously monitor their environment to check the presence of other entities and what services these are able to offer. The disadvantage of this approach is related to the characteristics of mobile devices: a periodical broadcast (or multicast) produces relevant power consumption; this is a typical example of the trade-offs that need to be evaluated in mobile computing systems design. Furthermore, broadcast may cause a waste of the scarce bandwidth resources that are available in wireless communication.

A remarkable number of standard service discovery frameworks have been developed in recent years: we describe the most significant features of two technologies, Jini and Salutation, in order to understand and compare possible solutions of this problem.

2.4.7.2 Jini

Jini [Wal99] [Sun02d] has been designed to be used in a spontaneous networking setting, for example composed of mobile devices or appliances. It provides facilities for service discovery, transactions, shared data spaces (JavaSpaces, that we have discussed before, are essentially a type of Jini service) and event notification. We discuss only the first of these, because it is the most relevant for our purposes.

The central concept in Jini architecture is the *service*, which is defined as an entity that can be used by a person, a program or another service. With respect to

discovery service the entities that are involved belong essentially to three categories: lookup services, Jini clients and servers. A service (for example, printing) registers itself to a lookup service by providing a set of *attribute-value* pairs that describe what it can offer and where it can be contacted. Afterwards, it is associated to a globally unique 128-bit service identifier generated by the lookup service. Every service is described by a so-called *service item* composed of three field: *ServiceID*, *Service*, *AttributeSets*. The first is the unique identifier, the second contains a reference to an object (usually in a remote location) that allows clients to invoke it using Java RMI, the third is a set of tuples similar to those used in a JavaSpace, thus a client can look for a service providing a template.

We underline another interesting feature of Jini related to reference management, the use of *leases*. Objects that offer services are at risk of having to maintain the resources when the users are no longer interested (or, in a mobile setting, they have moved to another place); for this reason, in order to avoid complicated discovery protocols, resources are offered for a granted limited period of time expressed by a so-called *lease*. Objects have to maintain these until the corresponding leases expire; on the other hand users are responsible for requesting their renewals, if they need them after the expiration.

In Jini, clients and services announce their presence using multicasting; for example, a client that needs a service but does not know its location sends a message using a well-known multicast IP address and lookup services that listen on the sockets bound to this, receive the request and reply to it. The response contains the unicast address, where the service can be found.

The large footprint of Jini (3 Mbytes), due mainly to the use of Java RMI, makes it not suitable for smaller devices such as PDAs. A possible alternative is Jmatos [Psi02] [SmiRK02], producted by Psinaptic that is compliant with Jini specifications and is characterised by a very small footprint (just 100 Kbytes).

2.4.7.3 Salutation

Salutation [Sal02] is a service discovery developed by leading information technology industries such as IBM, Canon, Xerox and Matsushita; it is an open

standard independent of operating systems, communication protocols and hardware platforms.

The central elements of this system are Salutation Managers that function as service brokers and interact with others through RPC, exchanging service registration information. Different features of a service are represented by functional units (e.g. fax, print, scan) characterised by sets of attributes. Salutation Managers can be discovered by entities in different ways: the most common are the use of static tables that stores their transport addresses or broadcast requests exploiting the protocol defined in the standard specification.

2.5 Future research directions

In the next years, mobile networks will be more and more heterogeneous in the sense that many different devices will be available on the market, with various operating systems, user interfaces, connectivity and communication capabilities. For these reasons, mobile computing middleware systems will have to adapt according the quality of available resources.

As we have discussed before, the most promising solution is to exploit application information in order to adapt the behaviour of the middleware. This must interact with the underlying network operating system and keep updated information about the execution context in its internal data structures, which can be available to applications; thus, they can listen to the changes in context (*inspection*) and influence the behaviour accordingly (*adaptation*). Therefore, a key issue in current mobile computing is how to enable this bi-directional flow of information between middleware and applications; an interesting proposal is to use reflection and metadata in order to build systems that provide context-awareness.

Another direction of research concerns security. Portable devices are particularly exposed to security attacks because it is easy to access to the wireless media. The aim of recent academic works and industrial products is to integrate existing technologies (such as advanced cryptographic techniques, PKI or VPN) in systems designed for mobile computing.

Chapter 3
XMIDDLE, a data sharing middleware for mobile computing

In this chapter we present XMIDDLE, a data sharing middleware for an ad-hoc network environment, describing its most interesting and innovative features. The essential motivation of this project is the need of sharing structured data among mobile hosts in this very dynamic context, where disconnections are the norm rather than the exception. XMIDDLE addresses this issue providing efficient mechanisms for automatic data replication and synchronization and hiding the complexity due to intermittent communication. Firstly, we analyse its architecture with a systematic approach; afterwards, we discuss some general issues related to networking and data replication.

3.1 Overview

XMIDDLE [MasCZE02] is a middleware for mobile computing that allows mobile hosts (for example PDAs, mobile phones, laptop computers and other wireless devices) to share information with each other, without assuming the existence of any fixed network infrastructure. Furthermore, it specifically addresses ad-hoc networking scenario that we have presented in the previous chapters. In XMIDDLE, connection is symmetric but not transitive; for example, we can consider the following situation: a host H_i is connected to host H_j and this to host H_k; in this case, H_i and H_k may be not connected to each other. In other words, we do not consider a multi-hop scenario, where routing through mobile nodes is allowed.

Firstly, we assume that mobile hosts store their data in tree structures, which allow sophisticated manipulations due to the different node levels, hierarchy and relationships between elements that can be expressed. XMIDDLE uses XML technologies (see Appendix A) in order to store, access and modify the data; we will analyse this aspect later, when we describe the implementation of our system.

Data sharing is possible only after a connection between hosts. Host H_A is connected with host H_B when it is *in reach* of this. Since we do not consider a particular network infrastructure, the definition of *in reach* varies depending on different protocols and transmission devices that are used; if we consider a wireless network, this expression means *in radio range*.

As we have discussed before, in a mobile scenario we have to address the problem of frequent disconnections; in other words, we must provide mechanisms in order to access and modify data in different working conditions, also in the case of disconnections; for this reason, XMIDDLE also allows off-line data manipulation, providing synchronization functionalities through a complex and efficient application dependent reconciliation process.

On each host, a set of possible access points for the owned data tree are defined in order to allow other nodes to link them. This operation that we call *link* is very similar to the remote directory mounting, a typical of distributed file systems. The access points in a host are grouped in an `ExportLink` set. For example, we can consider the following scenario: host H_i exports the branch A and hosts H_j and H_k link to this; thus, the owner of the branch A is still host H_i, but data now are shared by the three hosts, in other words these can access and modify them. When a host performs the *link* operation, a download of the linked branch is started.

After a branch modification, hosts have to notify the others that share it about all the changes introduced (immediately, if these are connected, otherwise, after a reconnection), in order to guarantee data consistency *reconciling* the copies stored in the different hosts running XMIDDLE applications.

A host is also allowed to make off-line changes to its replica of shared data; when it gets in reach again with other nodes, *a reconciliation process* of inconsistent replicas starts; we will describe it in detail in the next chapter.

Each host uses two sets in order to store information about the nodes that share their data. These sets are called `LinkedBy` and `LinkedFrom` and contain respectively the hosts that are linking to one of its branches and from whom a part of its tree is linked; they are structured as lists of tuples *(host, branch)*. Clearly, the `LinkedFrom` set does not mirror the current connection configuration; in other words, a host can be in the set of another one even if the two are disconnected; only specific primitives for linking and unlinking modify this set.

The `LinkedBy` set is updated by connection and disconnection operations that can be explicit or implicit (if two hosts get out of reach); it is used to know to whom to notify the eventual changes of parts of the tree. Applications can also explicitly disconnect from the network; this feature allows a host to save battery power or to perform changes without receiving updates from other nodes.

3.2 XMIDDLE architecture

The architecture of XMIDDLE is that typical of a middleware system; we present it according the ISO/OSI reference model. As shown in Figure 4.1, XMIDDLE implements the session and presentation layers on a standard transport protocol, such as UDP, that is provided in mobile networks on top of, for instance, a Bluetooth or a 802.11 data-link (i.e., Logical Link Control and Adaptation Protocol), MAC and physical layer.

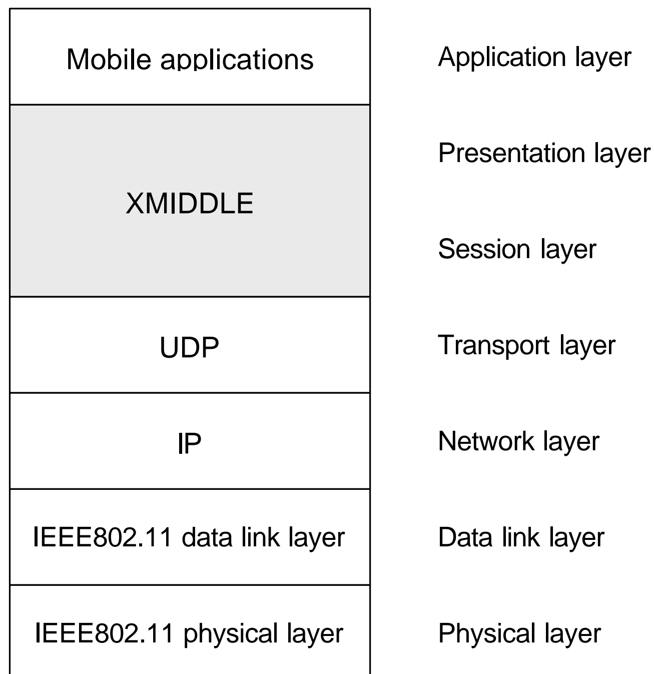

Figure 3.1 The ISO-OSI protocol stack for mobile environments using XMIDDLE

With respect to the presentation layer, XMIDDLE maps XML documents to DOM trees and provides the mobile applications with the primitives to manipulate

and to share data; the session layer implementation manages connections and disconnections.
The essential component of the architecture of our system is the XMIDDLE Controller, a concurrent thread that is able to communicate with the underlying transport protocol and to handle new connections and reconnections. Moreover, it can also trigger the reconciliation process of replicas according user-specific policies. Another fundamental module is the XMIDDLE Manager that provides the API in order to develop applications using this middleware; we underline that programs only have access to this component, which abstracts the rest of the middleware, providing a unified and high-level view of the system.
XMIDDLE is entirely implemented in Java and so it relies on a Java Virtual Machine; it is also worth noting that Java provides classes that allow to implement easily applications using TCP/UDP transport protocols.

3.3 Networking issues

The current implementation of the prototype of our system relies on UDP over IP. UDP was chosen over TCP, since the second is connection-oriented and so it introduces additional overhead. Moreover, TCP is inappropriate for situations where bandwidth is limited and where the network does not have a fixed structure, as in a mobile ad-hoc network setting.
On the other hand, UDP does not provide guaranteed packet delivery as TCP does; it is connectionless and, for this reason, it is more suitable for a mobile environment. XMIDDLE also exploits IP multicast in order to detect the presence of hosts that are in reach (i.e., in the same radio coverage) and to negotiate how to establish private communications between the nodes. We will discuss these issues in details in section 5.7.
Therefore, XMIDDLE can run on any hardware on which UDP over IP can be implemented, such as Bluetooth, IrDA, 802.11, Ethernet-based LAN. Consequently, XMIDDLE can also be used in a fixed environment, even if it addresses particularly mobile setting issues, like frequent disconnections. However, it might be exploited in some situations such as the case of a user without a permanent

network access and who uses a dial-up connection: in fact, in this case, the user might exploit XMIDDLE in order to work off-line.
We have been testing our system using HP iPAQ PDAs provided with 802.11 wireless LAN support and PCs connected by an Ethernet-based LAN.

3.4 Data manipulation and XML technologies in XMIDDLE

As we have discussed before, XMIDDLE stores data in XML documents that can be semantically associated to trees. Applications are enabled to manipulate them through the DOM API [W3C02b], which we present in Appendix A.
Other XML technologies are exploited in the design of this platform; in particular `LinkedBy`, `LinkedFrom`, `ExportLink` sets are formatted using XPath syntax [ClaD99]. Furthermore, the reconciliation of data trees is based on this class of technologies; we will discuss this issue in next chapter in details.
In order to understand how XMIDDLE represents and manipulates the shared data, we present an example from a possible collaborative e-commerce application.
For example, we consider a family, composed of three members, that usually does its weekly shopping electronically; it has a personal computer, where a replica of shop catalogue is maintained and each member owns a PDA, where a subset of products is replicated according his particular interest; every device has an embedded technology to establish an ad-hoc network, such as 802.11 connection. Furthermore, each family member has a replica of the shared shopping basket; they can buy products that are present in the catalogue, which is daily updated (for example by downloading it from an Internet site). Reconciliation of product catalogues and shopping baskets happens whenever the PDAs establish connection to each other or to the PC. This is a possible simplified representation of the catalogue:

```
<?xml version="1.0"?>
<shop>
  <category name="Hardware">
    <item>
      <name>Drill</name>
      <price>50</price>
```

```
    </item>
    <item>
      <name>Hammer</name>
      <price>5.50</price>
    </item>
    <item>
      <name>Screwdriver</name>
      <price>2.50</price>
    </item>
  </category>
  <category name="Dairy">
    <item>
      <name>Milk</name>
      <price>1.30</price>
    </item>
    <item>
      <name>Cheddar cheese</name>
      <price>3</price>
    </item>
    <item>
      <name>Parmesan</name>
      <price>8</price>
    </item>
  </category>
</shop>
```

In this example we have omitted some attributes that XMIDDLE uses to deal with data conflicts, since now we analyse only some general aspects of our system; we will introduce and discuss the details related to the reconciliation process in the next chapter.

The family members can link the catalogue and the basket that are present in the personal computer. At the beginning, obviously, they share an empty basket. A member may decide to link only a branch of the tree representing the catalogue, for example because he is interested only on hardware products. To link only to the hardware category, he has to specify the corresponding path of the DOM tree. For example, $Member_A$ may link to the hardware category, $Member_B$ to the dairy products and $Member_C$ to the whole catalogue. They also link to the basket. In order to do this, they have to use the following instructions:

```
//MemberA's link requests
link("mypc.home.net", "/shop/category[@name="Hardware"]",/);
link("mypc.home.net", "/basket",/);


//MemberB's link requests
link("mypc.home.net", "/shop/category[@name="Dairy"]",/);
link("mypc.home.net", "/basket",/);
```

```
//MemberC's link requests
link("mypc.home.net", "/shop",/);
link("mypc.home.net", "/basket",/);
```

The first parameter of `link()` operation is the server host name, in this case the personal computer (we discuss the problem of naming later); the second is the XPath [ClaD99] expression representing the root of the branch to be linked; the third is the "mounting point" on the local host.
The resulting XML tree that is maintained by $Member_A$ is showed in Figure 3.2.

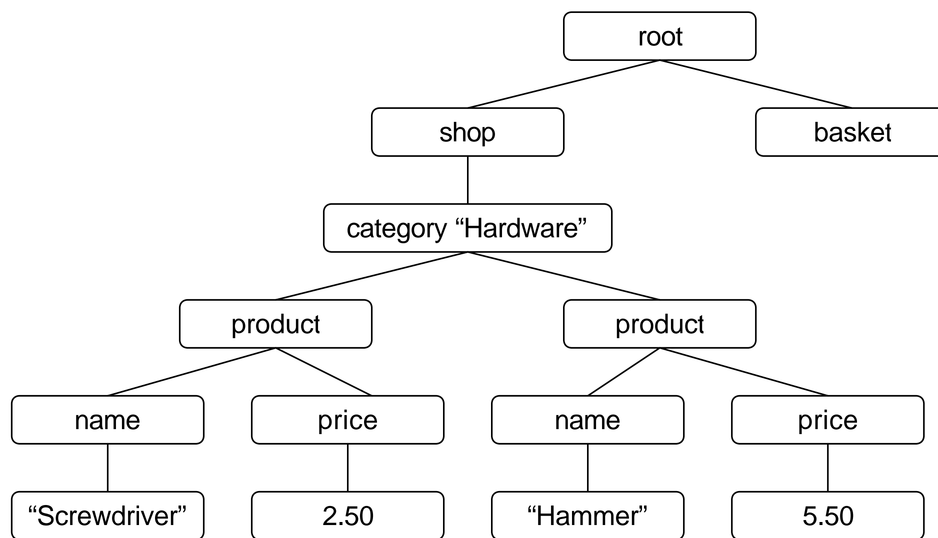

Figure 3.2 The tree representation of the data structure stored on Member A's PDA

After these linking operations, the application on each PDA can traverse and manipulate this data tree structure using DOM primitives, for example, in order to display the available products in the catalogue.
We may consider the common case of the purchase of a product; let us suppose that $Member_A$ buys a product (for example one hammer). The modifications on $Member_A$'s tree are showed in Figure 3.3.

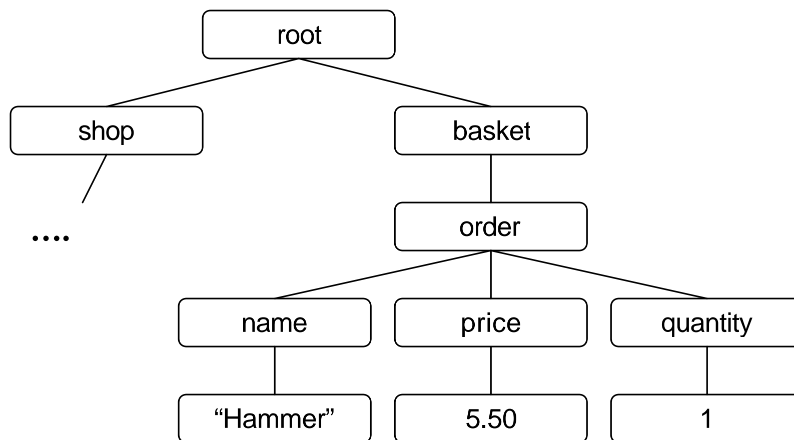

Figure 3.3 The tree representation of the data structure stored on $Member_A$'s PDA after an order has been entered.

Moreover, let us suppose that $Member_A$'s PDA is either out of reach or disconnected, while this update happens. When the device establishes a connection with the personal computer, the reconciliation protocol will reconcile any conflicts due to modifications of the catalogue (for instance, new products or discounted prices); likewise, any update that members have introduced to their shopping basket will be incorporated in the data structure on the PC. In our case, a new sub-branch describing the product that $Member_A$ has bought will be added in the branch with the `shop` element as root. Thus, the shopping basket on the PC is gradually filled through synchronization with the PDAs owned by the different members of the family.

Every user can update his product basket during a disconnection allowing for possible inconsistent replicas of the data tree. For example, in the application that we are considering two members of the family might buy the same product in different quantities, when they are disconnected. The reconciliation algorithm, which we will describe in Chapter 4, identifies that a conflict occurred, as the quantities are different; in other words, we have two different values in the same position. It is worth noting that we cannot solve this conflict without using application-specific knowledge (for example, in this case, the highest value has to be chosen) and without considering the "history" of the previous operations performed by users on the data tree. For example, another common case may be the addition of a new product by a member during a disconnection; after a re-

connection, the replicas stored in the other PDAs have to be re-synchronized accordingly.

Let us consider a simplified situation in order to better understand this issue: two members of the family buy a product when they are disconnected (or not in reach), having their baskets initially empty. For example $Member_B$ buys a bottle of milk and $Member_C$ buys a drill. The replicas of the documents are clearly inconsistent. For this reason, when they get in reach, a reconciliation process between these different copies has to be performed. The "reconciled" data structure must contain the two different products, as you can see in Figure 3.4 (in the illustration each sub-branches is represented without specifying its structure).

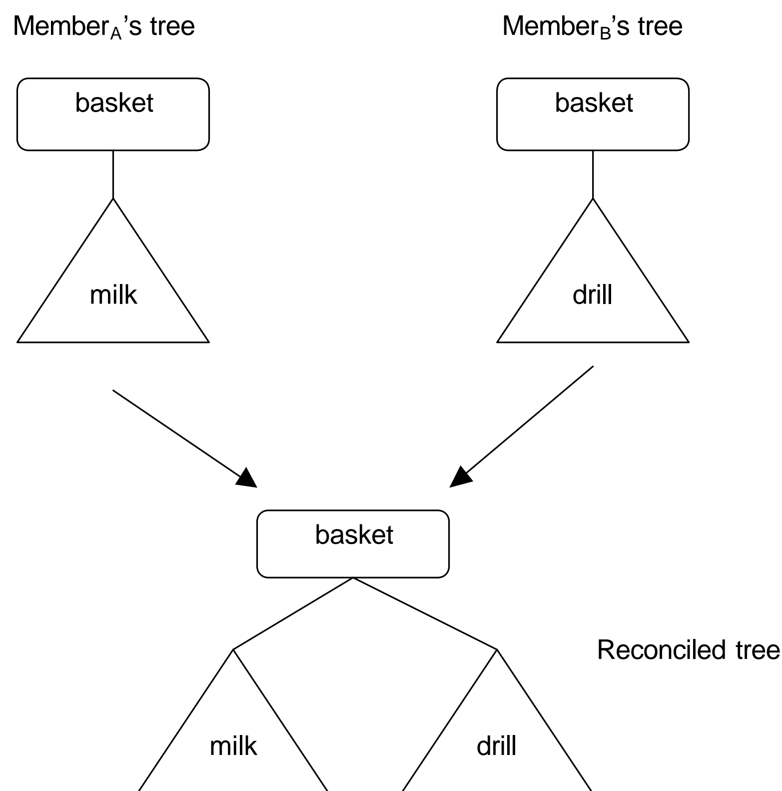

Figure 3.4 Example of reconciliation of inconsistent data replicas

We will discuss all possible cases in the next chapter extensively, analysing the possible conflicts between XML documents and how XMIDDLE solves them, focusing on the reconciliation algorithm and protocol.

3.5 XMIDDLE primitives

XMIDDLE provides a simple API that essentially allows applications to connect to and disconnect from the network and manage the shared XML data structures. Now we describe the most important XMIDDLE primitives from an application programmer perspective; in Chapter 5 we will analyse extensively the implementation of the XMIDDLE API.

3.5.1 The `connect()` and `disconnect()` primitives

As already discussed, each entry in the `LinkedFrom` and `LinkedBy` tables identifies a remote host and one of its branches and the `ExportLink` set maintains a list of the branches of the local tree that can be linked from remote hosts. Another fundamental table is `InReach` that stores the list of the hosts that are in reach.

The `connect()` primitive allows applications to notify their (re)connection to these nodes; upon this operation, a reconciliation process starts with all the hosts that are linking (or are linked) to a branch of the re-connected node and, at the same time, are in reach. In fact, the `LinkedBy` set is used to notify the changes to its tree to the other hosts that are included in this list. This is performed after a reconnection or during a connection (we refer to this case as *on-line reconciliation*). It is worth noticing that a host may be in the *connected* state but not in reach. In this case the reconciliation process is performed only when it gets in reach again. We can also reformulate in a more precise way the definition of *in reach* given before: two hosts are connected if they are in the same radio coverage *and* they have performed the `connect()` operation.

The `disconnect()` primitive allows a host to perform an explicit disconnection; this can also be implicit, when the host moves in an area where it is *out of reach*. The disconnection process modifies the content of the `InReach` table; we developed a specific protocol to handle this possible and frequent situation, which we will describe later.

3.5.2 The `link()` and `unlink()` primitives

The XMIDDLE `link()` primitive allows an application to link an exported tree (or a sub-tree) from a remote host; this is very similar to a mounting operation; in

fact, the parameters are the server host, the path in the remote XML document and the position in its data tree where the branch has to be appended. XMIDDLE records the details of this process in the `LinkedFrom` table.

The corresponding `unlink()` primitive modifies the local `LinkedFrom` table "unmounting" a particular branch of the shared tree.

3.5.3 The DOM primitives

XMIDDLE allows application programmers to access XML documents according DOM Level 1 Recommendation (see Appendix A), using a standard W3C implementation. It is also possible to retrieve particular sub-branches of the tree, for example the ones that are exported by the host. Essentially, the middleware provides two fundamental primitives to access the data: the `open()` primitive returns the sub-tree specified by using an XPath expression, whereas the `close()` primitive "commits" the changes. After this operation the local copy is modified and a synchronization process with the remote replicas stored in the other hosts is performed, in order to guarantee data consistency.

It is worth underlining that all these concurrent operations on the data structure stored in each host are synchronized; monitor and resource sharing concepts and techniques are applied in order to avoid accesses to inconsistent information.

3.6 XMIDDLE protocols

Another interesting feature of XMIDDLE is its sophisticated protocol execution mechanism. Protocols can be plugged into the middleware; the system allows their dynamic negotiation between hosts in order to perform the requested operation in an efficient way. Moreover, if a host does not have a specific protocol needed for a particular service, it transfers it from another node transparently.

Furthermore, an application can add a new protocol, which can be accessed and used by the middleware. Each protocol must inherit from a specific Java class (`edu.UCL.xmiddle.framework.lib.protocols.Protocol`) and must follow determinate specifications (see section 5.6.1.4). For example, our implementation based on UDP includes the reconciliation and linking protocol (see the chapter describing the implementation of XMIDDLE).

Another important aspect to point out is the concept of XMIDDLE protocol session; we define it as a structured networked communication between two or more XMIDDLE hosts; to start it, all the hosts that are involved needs to receive a request for the session. This can be automatically sent by the middleware in the case of an automatic reconciliation process or applications can explicitly send it, as in the case of linking protocol. We will describe these mechanisms in details in Chapter 5, analysing their implementation.

3.7 Host identification

A fundamental problem in distributed systems design is naming; in our case we have to distinguish different hosts in the same radio coverage, which form an ad-hoc network. We cannot use IP addresses, because they might change in different environments; for instance, this is the case of dynamic IP addresses.

Our solution is to use two types of identification credentials, the *Primary ID* and the *Secondary ID.* The first is assumed to be unique, in other words the existence of two different hosts with the same Primary ID is not allowed; it can be an Ethernet address, a Bluetooth's BD_ADDR, a number, etc. In the current implementation of XMIDDLE we simply use a large random integer, generated by `java.util.random` class.

The Secondary ID is used to identify hosts in a user-friendly manner; we do not assume that it has to be unique. Possible options are hostnames, usernames, etc.

Furthermore, it is worth noticing that the Secondary ID is not used in order to identify hosts during communication protocols; its purpose is only to allow applications to present to end users an understandable representation of the hosts that are currently in reach.The current implementation of XMIDDLE uses hostnames as Secondary IDs.

3.8 The target application domains

XMIDDLE addresses the needs of all developers that have to design applications based on data sharing in a mobile environment.

The range of possible applications is large, covering various fields from collaborative work to health-care service, from m-commerce to disaster management.
We underline that XMIDDLE is suitable both for an ad-hoc network setting, where hosts are peers and usually characterised by scarcity of resources, and for a nomadic setting where mobile devices interacts with more powerful machines, such as database services. However, XMIDDLE targets specifically the needs of developers that have to design ad-hoc networking applications; since the constraints imposed by this scenario are very restrictive, our system can also be used in situations where resource-rich devices are available, such as in a nomadic setting.
It is worth noticing that data structures must not be complex, due to memory and computational capability limitations. For example, users may link only to a portion of data in XML exported by a fixed database service host, in order to be able to store and manipulate them efficiently.
We have developed some applications based on the XMIDDLE platform in order to test and evaluate the performance of our middleware, a collaborative e-commerce application and a meeting organizer; we will describe and analyse them in Chapter 6.
Moreover, XMIDDLE is currently used in a project developed jointly by University College of London and Elan Technologies in the field of health care data management.

Chapter 4
Replication and inconsistency management in XMIDDLE

Since the service that our middleware provides is data sharing, the most important aspects of XMIDDLE are data replication and synchronization. Inconsistency management of distributed replicated data is an open and complex research area; also recently a great number of systems and innovative approaches have been presented. XMIDDLE manages trees that are classic but, at the same time, very expressive data structures, described by means of XML documents. In this chapter we will analyse in details these issues, discussing existing related works, showing our original solutions and pointing out the advantages of our approach.

4.1 Replication and data consistency

4.1.1 Concepts and models

A fundamental issue in distributed systems is data replication [CouDK01], since it is a widely used technique to enhance service provision in several application scenarios. For example, data may be replicated in order to share workload or to increase availability, especially in a mobile context, where disconnections are probable and, in some cases, frequent.

In fact, if we do not consider delays due to pessimistic concurrency control conflicts (i.e., data locking), the factors that are relevant to high data availability are server failures, network partitions and unexpected disconnections. On the other hand, distributed data are not necessarily consistent, because, for instance, they may be out of date: for this reason, disconnected operations are only feasible if users (or applications) can cope with stale data and can resolve any conflicts that arise.

A common requirement is *replication transparency*; in other words, clients should not have to be aware that multiple copies of data exist. This is not only the possible solution; in fact we can also find examples of systems in which the user can decide the replication strategy and the reconciliation policy.

Considering for example an object-oriented context, we can define two possible models: the first possibility is that objects are replication-aware; the second approach is based on using middleware for replica management (this is the case of XMIDDLE); the Figure 4.1 summarises these general patterns.

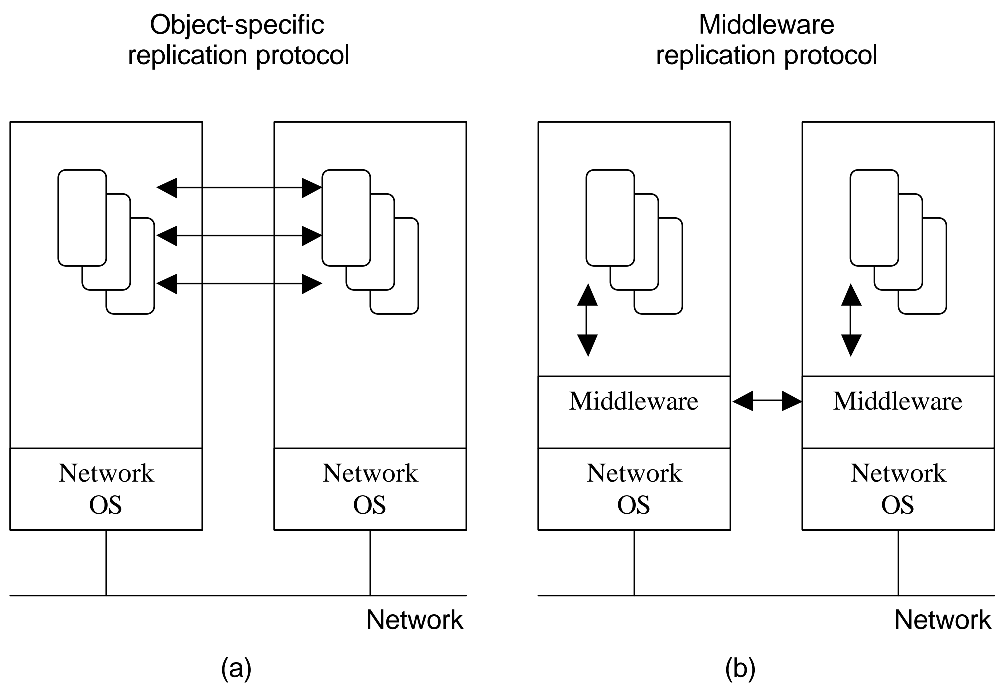

Figure 4.1 (a) A distributed system for replication-aware distributed objects. (b) Middleware system responsible for replica management

Middleware systems for data sharing hide the complexity related to inconsistency management allowing developers to build applications without having to deal with this challenging issue.

4.1.2 Possible approaches to the problem of inconsistency management of distributed data replicas

In Chapter 2 we have analysed examples of data sharing in some existing middleware platforms and we have pointed out how the problem of inconsistency of replicated data is tackled. Now we discuss some systems that do not belong to this class, examining other interesting approaches to this problem; then, we will present the solution that we have adopted in XMIDDLE.

4.1.2.1 Lotus Notes

Lotus Notes [Lot00] is a database-oriented system firstly developed by Lotus Development Corporation and now handled by IBM, which bought Lotus at the end of the 1990s. The model of this system is client-server; it is composed of three types of components: clients, servers, databases (that are associated locally both to a client and to a server); the key elements are *notes*, which form the local databases.

Lotus Notes, like Bayou, uses pair-wise reconciliation between replicas. It detects conflicting updates to a document using timestamps but it does not provide a support for application-specific conflict resolution, because the synchronization policies are hard-wired: for instance, an update/delete conflict always gives priority to the delete operation. In some cases, after various attempts, Lotus cannot solve the conflict and manual intervention is needed: however, the system chooses a "winning document", demoting the other documents as "losers" but maintains them so the user can decide to change the default resolution.

4.1.2.2 CVS

CVS [Ced02] is a source code versioning tool largely used in the software engineering community; developers manage their own replicas and a repository holds the master copy of each file. Programmers manage the coherence of the different versions; when they want to synchronise with the repository, they retrieve the master copy and try to integrate their modifications. The CVS conflict resolution is similar to the "Copy-Modify-Merge" technique included in the Sun Network Software Environment (NSE) [Cou89]. This paradigm enables concurrent collaborative work; after copying data (usually source code), users can

work disconnected and modify local replicas. During this stage, divergences between copies can occur; thus, in order to reach a consistent state again, developers have to re-synchronize with the master copy, without overwriting concurrent changes.
CVS detects conflicts, but it is the user that resolves them. We also point out that this detection process is purely based on textual comparison and always needs the intervention of the user; in fact, it simply prepares a list of conflicts with the indication of their position in the file.

4.1.2.3 IceCube

IceCube [KerRSD01] [ShaRK00] is a project developed at Cambridge Microsoft Research Centre concerning the problem of the reconciliation of divergent data replicas. It is a log-based system: the input to the reconciler component is a common initial state and the logs of actions that were performed on each replica; in other words, these represent the history of user actions. IceCube provides programmers with a parameterisable framework for reconciliation; developers can influence this process, either by providing local semantic information (in the form of pre-conditions and post-conditions) or using a global policy.
According to the IceCube model, there are two possible states for an application: *isolated execution* and *reconciliation phase*. In the first one, it operates on a local replica of shared objects, which are called the *object universe*; the operations are recorded in a local log, as ordered set of actions. During the second phase, the logs of two or more replicas are merged to bring the replicas to a consistent state.
The reconciliation phase is composed of three stages:

- *scheduling stage*, during which the logs are merged and a new log is produced; from this, a certain number of possible schedules are deduced;
- *simulation stage*, during which the schedules are tested to verify the constraints; a schedule that does not satisfy a constraint is aborted;
- *selection stage*, during which the application decides among the possible schedules after a ranking operation.

To sum up briefly, IceCube provides applications with powerful and efficient mechanisms, addressing the problem of the inconsistency management.

4.1.2.4 SynchML

SynchML [Syn02] is an industry initiative (supported, among others, by Ericsson, IBM, Motorola, Nokia, Palm and Psion) to develop and promote a single, common data synchronization protocol. It targets all networks, both wireless and wired, and supports multiple transport protocols, including HTTP, WSP (Wireless Session Protocol), OBEX (Bluetooth, IrDA), SMTP, TCP/IP and some common personal data formats such as vCard or vCalendar.

The programming framework is based on two client-server protocols, SyncML Representation protocol and SyncML Synchronization protocol; the first defines the representation format of SyncML messages (in XML) and details the inner workings of the framework, while the second describes the actions between a SyncML client and a SyncML server. The specification of this standard requires that both client and server maintain information about changes or modification to the data (for example, replacement, addition or deletion) in their databases by a mechanism called *change log*. In SynchML a version conflict happens when a single data item is modified on both the client and the server databases.

To operate in environments with limited bandwidth, SyncML utilizes WAP Binary XML (WBXML) [OMA02] to keep the data packets as small as possible. It also tries to reduce the number of request-response interactions between devices in order to minimize transmission costs.

For the reconciliation of divergent copies, servers (but also clients can have a restricted number of functionalities of this type) use a so-called synch engine, which adopts some pre-defined status codes for the notification of the chosen conflict resolution policy to the other hosts. Here are some examples of these:

- 207: Conflict resolved with data merge
- 208: Conflict resolved with client "win"
- 209: Conflict resolved with data duplication

To support context-awareness, SyncML also enables clients and servers to exchange information about their respective capabilities during an initialisation phase; either device can request this before a synchronization session. Since SynchML provides a synchronization model based on a client-server protocol, it is not suitable for an ad-hoc environment.

We can also point out that Microsoft does not participate in this initiative; since it holds a significant market-share with its proprietary operating system (Pocket PC) and solutions for back-end databases (SQL Server), it instead developed the ActiveSync software [Mic02], a synchronization tool for PocketPC, which is optimized for retrieving data from an SQL Server database.

4.2 Versioning system

Data replication and synchronization are two fundamental aspects of XMIDDLE; we now describe how our system manages different versions of XML documents using DOM and, then, we detail the reconciliation process.

In order to facilitate the automatic detection of conflicts and to support data reconciliation, XMIDDLE uses a *versioning system*.

The principal structure maintained on every host is a tree that contains a directed version graph of the data elements generated by the hosts that forms the ad-hoc network. There are two types of elements: *versions* and *editions*. Versions contain changes that have been performed locally (without communicating them to the other hosts). Editions are, in a sense, stable versions; they contain changes that have been agreed with another host, after a reconciliation process.

We refer to the process of establishing a new edition as *releasing* a version. Moreover, an edition can be saved to a persistent storage medium, such as Flash RAM, whereas a version is still subject to changes and it is only kept in main memory. Therefore, an edition can have both versions and editions as directed descendents in the version graph, whereas a version does not have descendants, because first it has to be turned into an edition. Consequently, versions always contain the most recent information. Figure 4.2 shows these concepts in the case of three different hosts. In this illustration, $Host_A$ and $Host_C$ have released two editions of the tree and, currently, $Host_A$ works on a modified copy (a version) of the latest common edition (marked with number 2). $Host_B$ has only a version (it has not reconciled it with other hosts).

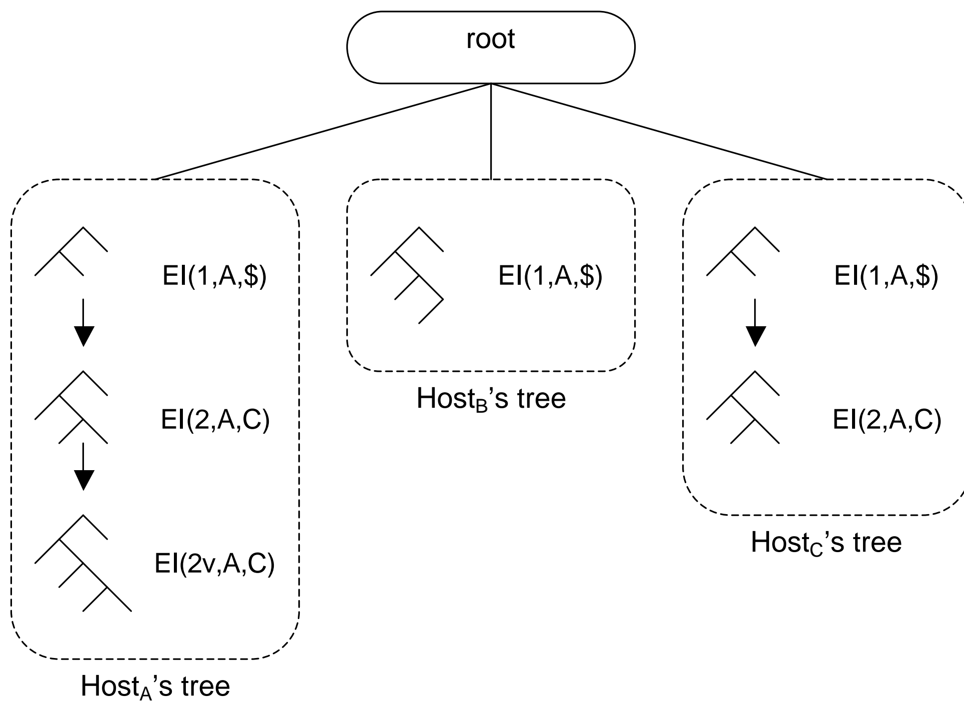

Figure 4.2 Version history graph of DOM trees

Shared elements are distinguished by an edition identifier, which uniquely identifies an edition. The identifier is a tuple, like

`EdID (editionNumber,` H_i`,` H_j`)`

In this expression, H_i and H_j identify the two hosts that agreed in releasing this edition and `editionNumber` is the maximum of the two previous edition numbers incremented by one. This is used to distinguish between subsequent editions agreed by the same couple of hosts. We also assume that the sequence of edition numbers always starts from number one, when the element is first exported by the owning host. When a version is derived from an edition, a letter "v" is appended to the edition number. Moreover, we use the symbol $ when the creation of the edition does not involve a second host (for example, this is the case of a host that generates the first edition after the export operation performed by an application).

This versioning scheme solves the problem of identifying different editions of shared information in a distributed system that lacks a central authority to issue edition numbers; in fact, it is possible for two hosts to reconcile a tree they copied from another host, without interacting with the owner (in our case, a possible central authority). This is a coordination pattern that is suitable for a mobile setting; in other words, we apply a typical peer-to-peer solution, without the presence of a host that is able to provide particular service (in our case, number edition issuing).

Now we may consider an example, which is illustrated in Figure 4.3, with four hosts H_A, H_B, H_C and H_D that store edition 1 of a tree `T` linked from host H_X.

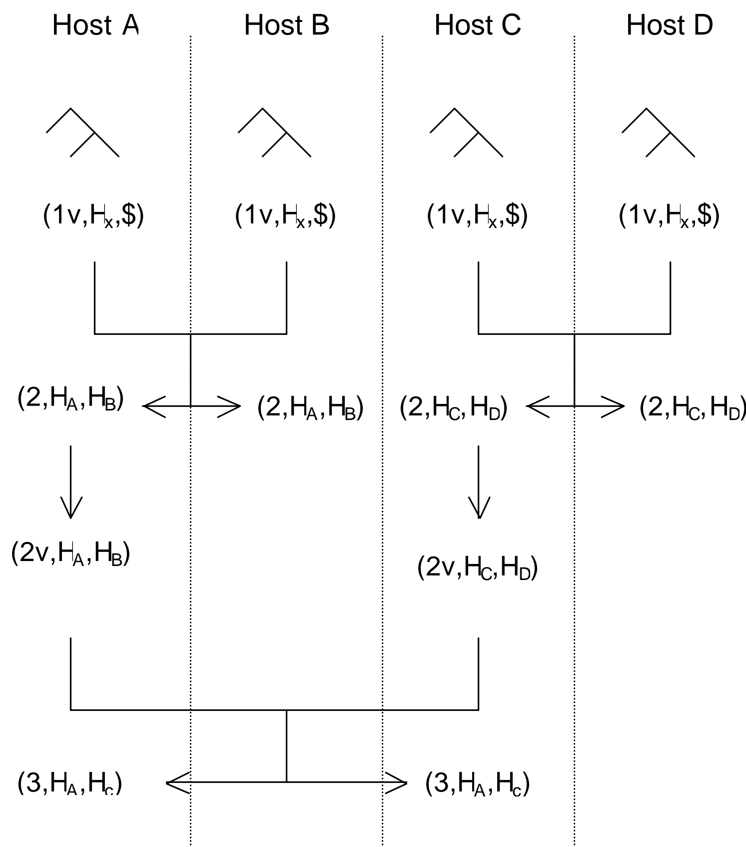

Figure 4.3 An example of edition identifiers in XMIDDLE

During a disconnection, they can modify their local version independently from each other; when two hosts get in touch again, for example H_A and H_B, as showed in the picture, they reconcile their copies, creating a new edition with

`editionNumber` equal to `2`. The same may happen to H_C and H_D with another reconciliation process leading to a different edition `2`. In this situation, if H_A and H_C connect and look at the edition number only, they may erroneously assume their latest common version of `T` is number `2`; the technique adopted in XMIDDLE eliminates the problem. In fact, H_A and H_C are able to recognize that they have two different versions of the tree, with different labels, respectively `EdId(2,`H_A`,`H_B`)` and `EdId(2,`H_C`,`H_D`)`. Therefore, they can reconcile their different editions and generate a new one with `editionNumber` equal to `2`, labelled with `EdId (3,`H_A`,`H_C`)`. We can observe that in the figure two versions are also represented; they can be distinguished by the letter `v` attached to their edition numbers.

4.3 The XML TreeDiff algorithm

In order to detect changes in its XML tree structure, XMIDDLE uses an algorithm similar to IBM XML TreeDiff [IBM98] [PooCE99].

The computational process of differentiating two labelled tree structures is expensive; in fact, for example, its cost for ordered trees is at least quadratic in the number of nodes. A traditional approach to this problem is to use a tree-tree comparison technique, which was first presented by Tai [Tai79].

IBM XMLTreeDiff is based on a technique proposed by Zhang [ZhaS89] to find differences between two generic trees. It uses a two-step process: firstly it reduces the size of both trees by fast sub-tree matching using the DOMHash technology [MarTU00] developed at the IBM Tokyo Research Laboratory. Subsequently, Zhang's optimal editing distance algorithm is applied to the sub-trees. IBM XML TreeDiff is provided as a set of Java Beans, which directly manipulate the DOM representation of XML documents. The output is an XML document as well, where the differences between data tree structures are showed according to a pure object-oriented style, in the form of edit operations (i.e., change a label on a node, insert and delete nodes, and prune and graft sub-trees). It is worth nothing that IBM XML TreeDiff does not use the traditional approach that consists in the indication of line mismatches.

In XMIDDLE we use a simplified version of this algorithm, which is also suitable for mobile applications, since it is not too heavy as regards memory and computational requirements. In order to understand how it works, we show an example using the same simplified XML documents that we have presented in Chapter 3 to illustrate the data representation in our system. We will then introduce the complete structure of the documents managed in XMIDDLE. We may consider the following two XML documents (Doc_A and Doc_B) representing the shopping baskets of two users that we call respectively $Member_A$ and $Member_B$.
$Member_A$ stores the following data structure:

```
<?xml version="1.0"?>
<basket>
      <item>
            <name>pen</name>
            <quantity>1</quantity>
            <price>1.15</price>
      </item>
      <item>
            <name>lightbulb</name>
            <quantity>1</quantity>
            <price>0.9</price>
      </item>
</basket>
```

$Member_B$'s document is the following:

```
<?xml version="1.0"?>
<basket>
      <item>
            <name>pen</name>
            <quantity>2</quantity>
            <price>1.15</price>
      </item>
</basket>
```

In order to find the differences between Doc_A and Doc_B, we use our XML TreeDiff algorithm; before analysing the output returned by this, we notice that, there are two divergences: firstly, the root of Doc_B has only one child, while the root of Doc_A has two children; secondly, the values of the text nodes that are children of the quantity node are different.

The output of the XML TreeDiff algorithm, with Doc_A as first parameter and Doc_B as second, is the following:

```
<?xml version="1.0"?>
<treediff>
   <changepcdata newvalue="2" oldvalue="1"
xpath="/basket/item[1]/quantity[1]">
   </changepcdata>
   <delsub-tree xpath="/basket/item[2]"></delsubtree>
</treediff>
```

It is worth noting that the operations that are reported in this XML document have to be applied to document Doc_A in order to obtain Doc_B. In other words, if we change the order of parameters, the result is different, since the output of this algorithm is not a list of differences, but the operations that must be performed on the first document in order to produce the second. In fact, the result of this algorithm, with Doc_B as first parameter and Doc_A as second, is the following:

```
<?xml version="1.0"?>
<treediff>
   <changepcdata newvalue="1" oldvalue="2"
xpath="/basket/item[1]/quantity[1]">
   </changepcdata>
   <addsubtree insertOrder="1" xpathleftsibling="/basket/item[1]"
xpathparent="/basket">
            <item>
              <name>lightbulb</name>
              <quantity>1</quantity>
              <price>0.18</price>
            </item>
   </addsubtree>
</treediff>
```

This asymmetry has to be considered in the design of the synchronization process; therefore, we underline that, in XMIDDLE, this algorithm is prevalently used to reconstruct the current different trees from a common edition in a host, before starting the reconciliation between them.

4.4 Reconciliation protocol

The reconciliation protocol is one of the most important design aspects of XMIDDLE and, for this reason, we will analyse it in details. Essentially, its aim is

to obtain a consistent version of the same tree once two hosts become connected. It is based on DOM tree differentiating and merging techniques (mainly on XML TreeDiff). In fact, the design goals of this protocol are to minimise data transfers, only transmitting the differences between data structures and, at the same time, to be able to reconstruct diverging replicas from a common previous edition on the same host, in order to reconcile them locally. Then, the result of the reconciliation is propagated to the other host, communicating only the changes performed on the common latest edition.

In order to describe the protocol, we use the following notation:

$X \rightarrow_{msg} Y$

The meaning of this expression is: the message `msg` has been sent from `X` to `Y`.

1) $H_B \rightarrow_{LinkedFromB,\ ExportLinkB} H_A$

When two hosts get in reach, in order to see whether they share some information they exchange their `LinkedFrom` and `ExportLink` sets. Firstly, H_B sends to H_A its `LinkedFrom` and `ExportLink` sets.

2) $H_A \rightarrow_{LinkedFromA,\ ExportLinkA} H_B$

Similarly, H_B sends to H_A its `LinkedFrom` and `ExportLink`. If they realize that they are sharing a branch `T`, they first lock it and then they start the reconciliation process.

3) $H_B \rightarrow_{T,\ ListOfEI} H_A$

H_B sends to H_A the list of all edition identifiers previously released for `T`.

4) $H_A \rightarrow_{LastSharedEI,\ ListOfChanges} H_B$

H_A parses the list of edition identifiers sent to it and finds the latest shared edition identifier by searching its version graph. Then H_A computes the changes performed and sends this list of differences together with `lastSharedEI` to H_B.

5) $H_B \rightarrow_{NewChanges,NewEI} H_A$

In order to reconstruct an up-to-date copy of H_A's tree `T'`, H_B applies the differences that it has received. Then it computes the differences between its own latest version and `T'` and *reconciles* these XML tree structures, using the techniques that we will describe later in this chapter to resolve the possible conflicts. The next step is to communicate these changes to the other host; thus, H_B computes the differences between the reconciled tree, called `T''`, and `T'` and sends the list of changes to H_A (in the expression we have defined it `newChanges`), along with the new edition identifier. This has the form `(maxversion+1,` H_A`,` H_B`)`, where `maxversion` is the maximum of the two previous edition numbers.

6) $H_A \rightarrow_{ACK} H_B$

H_A computes the up-to-date edition and stores it. It also acknowledges the transfer to H_B.

7) $H_A \rightarrow_{ACK} H_B$

These last two steps are needed to acknowledge the two hosts that the protocol has been successfully completed; in other words, when H_B receives the `ACK` message from H_B, it realize that H_A has the new edition of `T`. If any acknowledgement is not received, the hosts automatically rollback to the state the tree was in before the reconciliation process. The initial lock on the trees is also removed.

We now analyse the possible failures of this algorithm and the relative consequences. In case of a failure before step 5, the reconciliation process simply aborts. Otherwise, if the protocol is stopped between step 5 and 6, a rollback procedure is started by both hosts. If it fails between steps 6 and 7, H_A rolls back while H_B completes successfully the reconciliation process. This does not cause problems, since they will get in reach again, they will reconcile using the `lastSharedEI`, because H_A does not possess `newEI`.

We have described the protocol that is executed by two hosts, but it is worth noting that the reconciliation process involves all the hosts that share the same sub-tree. In other words, we may consider the situation of three hosts H_A, H_B, H_C that share the same branch `T`. If H_A, for example, modifies `T`, it has to *reconcile* its replica with H_B and H_C. A related possible future research issue is the design of a

group protocol that involves three or more hosts contemporaneously based on the same versioning system (for example, considering the latest common edition shared by all participating hosts).

4.5 Linking protocol

Now we analyse briefly the linking protocol, since it is strictly related to that which we have just described. In fact, the link operation may be performed as reconciliation between a tree `T` and an empty one, considered as edition 0 of T. The following are the essential steps of this protocol.

1) $H_B \rightarrow_{\texttt{LinkedFromB, ExportLinkB}} H_A$

The first step is exactly the same as in the reconciliation protocol: H_B sends to H_A its `LinkedFrom` and `ExportLink` sets.

2) $H_A \rightarrow_{\texttt{LinkedFromA, ExportLinkA}} H_B$

The second step is the same as well: H_B sends to H_A its `LinkedFrom` and `ExportLink` sets.

3) $H_B \rightarrow_{\texttt{T, (0,\$,\$)}} H_A$

H_B sends to H_A the identifier of the empty edition; we use the notation `(0,$,$)` to indicate this. It can be considered as a particular case of the third step of the reconciliation protocol; in fact, in this case the list of edition identifiers contains only one entry.

4) $H_A \rightarrow_{\texttt{(1,HA,\$),LastEdition,ActiveChanges}} H_B$

This is the fundamental step of the linking protocol; when H_A receives the tuple `(0,$,$)`, it realizes that H_B wishes to link to `T` and replies with the latest common edition, together with a list of changes previously performed on the local initial empty tree. H_B can now store this data structure and apply the changes in order to perform the re-synchronization with H_A. We also underline that H_A sends the first edition of `T` to H_B which is now able to reconcile with any other hosts

linking to the same tree. In fact, all the hosts that are linking to the same branch have at least the first edition in common.
The other phases of the protocol are similar to those that we have presented in the previous paragraph.

4.6 Connections and disconnections

Another interesting issue to discuss is how connections and disconnections are managed in XMIDDLE, considering the issues related to data consistency.
The connection procedure is always performed by an application, by using the `connect()` method; it is worth noting that a host may be connected but at the same time not in reach, as we have discussed in the previous chapter. The reconciliation process is performed automatically in these cases:

- considering a host that is initially not connected, after a re-connection, if hosts that share the same sub-tree are in reach;
- considering a host that is initially connected but not in reach with other hosts which shares the same sub-tree, when it gets in reach again with those;
- considering a group of hosts that share the same sub-tree and are connected and in reach, when one of these performs a modification on its replica (*on-line reconciliation*).

The disconnection procedure is started by the application in case of an explicit disconnection, whereas it is performed by the middleware in case of an implicit one. It is very simple: for each version not yet released, the host releases it; afterwards, the versioning process is started and the resulting tree is stored. The edition identifiers that are issued during this procedure have the form `(editionNumber, `H_B`, $)`. In case of explicit disconnections, H_B does not have to notify the other nodes about its decision, because the middleware takes care of this aspect, providing the list of the hosts that are currently in reach (we will explain the implementation of this mechanism in the next chapter). In other words, if a host gets disconnected (as a consequence of an explicit operation or if the device is not reachable), the other nodes are automatically aware of this fact and start the same disconnection process. We underline an important aspect of this

procedure: if it is started by an application, its result is the complete disconnection of the host from the network; if it is initiated by the middleware, it allows the disconnection from a subset of the hosts forming the ad-hoc network. For example, we may consider the case of three hosts initially in reach; host H_A may implicitly disconnect from H_B and at the same time it may remain connected with H_C. This situation is illustrated in Figure 4.4.

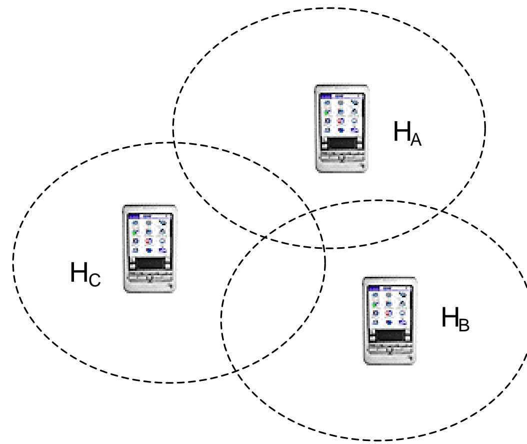

Figure 4.4 a Hosts H_A, H_B, H_C are in reach

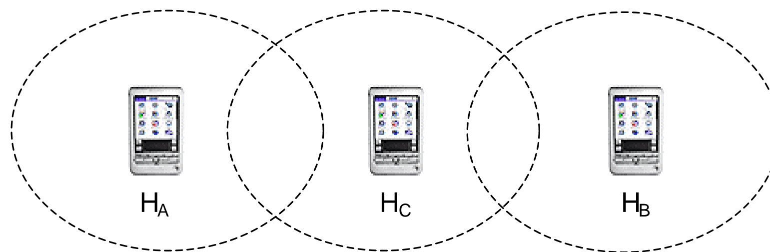

Figure 4.4 b Host H_A is now implicitly disconnected from host H_B

4.7 Conflict resolution

4.7.1 Overview

XMIDDLE provides developers with primitives and mechanisms to control the reconciliation process of the possible conflicts between different replicas. In other

words, it can be performed in an application-specific way. This feature of our middleware is fundamental for a large class of applications.
It is worth noting that XMIDDLE is a middleware and for this reason it provides mechanisms, but it does not implement any particular policy. From this point of view, it is extremely flexible: in fact, programmers can easily develop complex mobile applications that need data sharing, without considering the problems related to disconnections and possible data inconsistencies, but, at the same time, our middleware is application-aware, that is, its behaviour can be modified by developers according to their requirements.
We also underline that the reconciliation process is transparent to the users: they are not aware of the automatic detection and resolution of the conflicts between the different copies and the system does not need their intervention.
Furthermore, XMIDDLE provides mechanisms to change reconciliation policies dynamically; therefore applications are able to modify them if necessary.

4.7.2 The semantic problems in conflict resolution

First of all, it is worth noting that if conflict detection is a *syntactic* problem, conflict resolution is a typical *semantic* issue; we can compare our data structures, but we cannot solve conflicts without knowing what they represent.
The design of XMIDDLE must be absolutely independent from any particular application requirement and, at the same time, it has to be able to deal with application dependent conflict resolution. In other words, our middleware must be *application-independent*, but at the same time *application-aware*. This is a typical issue in mobile systems design in general, for example in order to address the problems related to the changes of context (bandwidth, position, etc.) or the scarcity of resources. In fact, as we have discussed in the previous chapters, a key aspect in mobile system research [CapEM01] is how to enable this bi-directional flow of information between middleware and applications.
XMIDDLE uses reflection and metadata in order to enable the middleware to decide transparently how to behave in different situations. Through metadata, we can distinguish two aspects of middleware, the mechanisms that it implements and the policies according to which it works. We exploit this approach using XML technologies; application developers can modify the behaviour of the middleware

through a small number of documents that are used to describe data and configure the system and the interaction between applications that are located on different hosts.

In XMIDDLE the central aspect is how to manage the synchronization between replicas; now we detail our solution, starting from the classification of the possible conflicts in tree data structures. In fact, we underline again that it is possible to describe XML documents as tree structures and vice versa; we will follow this approach in our explanation.

4.7.3 Possible conflicts

4.7.3.1 Terminology and general considerations

Trees [Sta99] are one of the most commonly used data structures; a familiar example is Unix file system structure. Now we introduce the terminology widely adopted to describe trees. Firstly, we give some preliminary definitions: a *node* is a single object within a tree; an *edge* is a connection between two nodes; a *path* is a list of nodes connected in order by edges. Therefore, we define a *tree* as a collection of nodes connected by edges so that there is one and only one path between two nodes. If there is more than one path, the collection of nodes is usually called a *graph*; if there is not a path between some nodes, the collection is named a *forest* or a *disjoint set of trees*.

XML documents can be described as *rooted trees*; in other words they have a node designated as the top of the tree, called root. It is possible to identify other relationships considering the tree as a genealogical structure; given a node, we call *parent* the node directly above and *child* the node that is directly below. A node has one parent and none or more children. We name *siblings* the children of the same parent and *leaf* a node with no children.

In the XML documents, for example, text nodes are leaves since they cannot have children; we discuss the essential aspects of the XML language (and its possible tree-like DOM representation [W3C02b]) in Appendix A. We define *descendants* all the nodes that can be reached by going downward along any path and *ancestors* all the nodes that can be reached by going upwards along the path to the root.

We frequently number the levels in a rooted tree, starting with 0 for the root node level and increasing downwards. We call the number of the highest level the *height* or *depth* of the tree. In a rooted tree, each node is the root of a *sub-tree* (or a *branch*) that consists of it and all its descendants. In computer science, many different varieties of trees are used and studied, but we do not present them, because it is beyond the scope of this thesis.

Now we examine a classification that is of primary importance in our system, the discrimination between order and unordered trees. In an *ordered tree*, the children of each node have a certain order (usually thought of as going from left to right). In an *unordered tree*, the children have no order; we can try to visualize this as a 3-D tree where the children float around under the parent. An unordered tree can be represented with many different ordered trees (Figure 4.5). For this reason, finding out whether two ordered trees represent the same unordered tree is difficult (this issue is known as the *tree-isomorphism problem*).

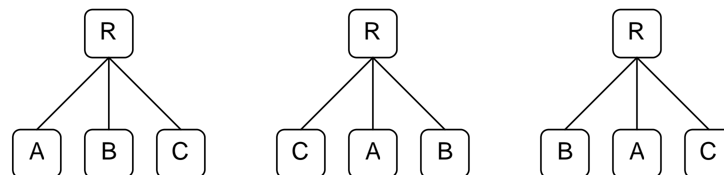

Figure 4.5 Different ordered tree representing the same unordered tree

In XMIDDLE we apply this definition in a particular way, considering macro-elements with a predefined ordered structure that we call branches; in other words, in an *XMIDDLE unordered tree*, branches (considering monolithically not only the direct children of the root, but all the levels below) has no order. In other words, the term unordered is referred to the sub-trees rather than single nodes. This is very useful in a great number of applications, where the "logical units" are entire sub-trees and not only the children of a node. For this reason we need a mechanism to describe branches; we will describe this interesting aspect later.

We can also motivate our design choices, using XML documents; we may consider the example of the shopping basket, where the "logical units" are the items; the position in the structure (the "`item` sub-tree") that is used to describe

the products in the tree it is not important. Let us consider, for example the following two documents:

```
<?xml version="1.0"?>
<basket>
      <item>
          <name>pen</name>
          <quantity>1</quantity>
          <price>1.15</price>
      </item>
      <item>
          <name>rubber</name>
          <quantity>1</quantity>
          <price>0.5</price>
      </item>
</basket>
```

DocA

```
<?xml version="1.0"?>
<basket>
      <item>
          <name>rubber</name>
          <quantity>1</quantity>
          <price>0.5</price>
      </item>
      <item>
          <name>pen</name>
          <quantity>1</quantity>
          <price>1.15</price>
      </item>
</basket>
```

DocB

We point out that the structure of these documents represents the same contents of the shopping basket, even if the position of the item sub-trees is different.

At the same time, if we want to compare two items, we have to consider the corresponding branches, which are ordered sub-trees. In other words, in order to detect a conflict in leaves we have to compare the values that are in the same position considering the structure of that particular branch (i.e., in an ordered `item` sub-tree, the children of a `quantity` node). Let us consider two XML documents, which we denominate respectively Doc_A and Doc_B. The first is the following:

```
<?xml version="1.0"?>
<basket>
      <item>
            <name>pen</name>
            <quantity>1</quantity>
            <price>1.15</price>
      </item>
</basket>
```

Doc_B is the following:

```
<?xml version="1.0"?>
<basket>
      <item>
```

```
            <name>rubber</name>
            <quantity>1</quantity>
            <price>0.5</price>
      </item>
</basket>
```

We notice that during the reconciliation of these documents, we find two conflicts: the value of text node that is child of the `name` node and the values of `text` node that is child of the price node. In an application like that we have presented, a logical resolution of the conflict is the duplication of the `item` sub-tree; so we have to deal with another problem, because we must describe the sub-tree that has to be replicated. In fact, conflict detection mechanism provides only the position of the conflicts (in our case we have two `changePCData` commands with new and old values) and it does not specify how to solve it. Moreover, we cannot simply duplicate the parent node with two different text nodes as children; the next example shows the wrong resolution of the conflict in the node that is child of `name` node.

```
<?xml version="1.0"?>
<basket>
      <item>
            <name>rubber</name>
            <name>pen</name>
            <quantity>1</quantity>
            <price>0.5</price>
      </item>
</basket>
```

The logical (and correct) output of the reconciliation process in this case is a basket with two `item` branches, one from Doc_A and one from Doc_B:

```
<?xml version="1.0"?>
<basket>
      <item>
            <name>pen</name>
            <quantity>1</quantity>
            <price>1.15</price>
      </item>
      <item>
            <name>rubber</name>
            <quantity>1</quantity>
            <price>0.5</price>
      </item>
</basket>
```

Later we will describe our technique that we use to address this issue. It is worth noting that duplication is not the solution to all these types of conflicts. We may consider, for example, two customers that buy the same product in different quantities. The following XML documents describe their shopping basket:

```
<?xml version="1.0"?>
<basket>
      <item>
            <name>pen</name>
            <quantity>1</quantity>
            <price>1.15</price>
      </item>
</basket>
```

```
<?xml version="1.0"?>
<basket>
      <item>
            <name>pen</name>
            <quantity>2</quantity>
            <price>1.15</price>
      </item>
</basket>
```

In this case we have a conflict in the text node that is the child of the `quantity` node; now we should not use the duplication of the branches to solve it, but only choose a new value. Moreover, we need to express a particular policy in order to reconcile it (for example “the greater value must be choosen”, or more complex policy considering also the latest common edition).

Furthermore in other cases the topology of the data structure is fixed and we cannot modify it. Let us consider a tree representing the medallists of an Olympic competition.

```
<?xml version="1.0"?>
<medallists competition="100m">
      <goldMedallist>
            <name>George Johnson</name>
            <nationality>USA</nationality>
            <result>9.94</result>
      </goldMedallist>
      <silverMedallist>
            <name>Mark Lewis</name>
            <nationality>USA</nationality>
            <result>9.98</result>
      </silverMedallist>
```

```
	<bronzeMedallist>
		<name>Paul Smith</name>
		<nationality>UK</nationality>
		<result>10.01</result>
	</bronzeMedallist>
<medallists>
```

In this case, the topology of the tree is fixed and, for this reason, the detection and the resolution of the conflicts is easier, because we only have to compare the values in the same positions (for example, two timing stations running XMIDDLE may sense two different results for the second athlete and they need to reconcile this conflict); thus, this is a typical example of an ordered tree.

4.7.3.2 Value and structure conflicts

In the previous example we have analysed *value conflicts*, which are related to the values of text nodes. The other possible case of this kind of conflicts is related to attribute nodes that contain different strings. For example, we may use an attribute to indicate the availability of a product in a XML document representing a shop catalogue and different replicas may present stale values that need to be reconciled.

During the reconciliation process we can also detect *structure conflicts*, related to the topology of the tree. A possible example of structure conflict is the following:

```
<?xml version="1.0"?>
<basket>
	<item>
		<name>pen</name>
		<quantity>1</quantity>
		<price>1.15</price>
	</item>
	<item>
		<name>rubber</name>
		<quantity>1</quantity>
		<price>0.5</price>
	</item>
</basket>

<?xml version="1.0"?>
<basket>
	<item>
		<name>pen</name>
		<quantity>1</quantity>
		<price>1.15</price>
	</item>
</basket>
```

We can notice that the difference between these two documents is the number of the children of the root element; in general, a structure conflict arises when we have different relationships among the nodes of the tree. In this case a possible resolution of the conflict is the addition of the sub-tree describing the item *rubber*. We may also have a combination of structure and value conflicts, in ordered and unordered trees; we may analyse these documents:

```
<?xml version="1.0"?>
<basket>
	<item>
		<name>rubber</name>
		<quantity>1</quantity>
		<price>0.5</price>
	</item>
	<item>
		<name>pen</name>
		<quantity>1</quantity>
		<price>1.15</price>
	</item>
</basket>
```

```
<?xml version="1.0"?>
<basket>
	<item>
		<name>pen</name>
		<quantity>1</quantity>
		<price>1.15</price>
	</item>
</basket>
```

In this case, the conflict detection mechanism (by using XML TreeDiff algorithm) finds both structure and value conflicts; this is a typical case of an unordered tree; in order to reconcile these documents, we cannot simply compare the values in the same position, but we have to consider the branches and confront them. This is quite difficult, because, for example, two sub-trees describing the same product may have different values in the `quantity` text node. In this case we have to compare the "pen" sub-tree with the others that are present in the first document, without considering the quantities.

Furthermore, in order to solve a conflict like the one we have just described, it is not sufficient to consider the two documents that have to be reconciled, but we also need to analyse the latest common edition. For example, the addition of the

item `rubber` may lead to a wrong reconciliation; let us suppose that, after a synchronization, the two customers had both products (rubber and pen) in their basket and then the second customer cancelled the purchase of the rubber obtaining these documents. In this case, probably, the better resolution of the conflict is not the *addition* of the rubber sub-tree in the second document, but its *deletion* from the first. As you can see from this example, it may be fundamental to consider the common latest edition in the reconciliation process.

Furthermore, it is worth noting that in our system conflicts related to the names of elements (in other words, the *name* property of a node object) are not possible, because the structure of the documents must be predefined; in other words developers must describe them by using XML Schema and they might also validate them. A determinate element can be found only in specific levels of the tree; moreover, we underline that the approach of the XML TreeDiff algorithm is level-based and it is not possible to detect conflicts related to positions in different levels of the tree. In fact, the comparison between two trees starts from the root element, considering the differences at each level and, on the other hand, the structure of sub-trees is reconstructed by using the XML Schema description of the document that is hierarchical.

In the previous examples we have presented possible ways to solve conflicts that we have adopted as default policies of XMIDDLE. Moreover, our middleware also offers a non-standard resolution mechanism; this allows developers to specify particular policies in order to solve value conflicts. They can do it by following a very simple procedure by using the attribute `resolutor` in the element that is the parent of the text node that needs to be managed in a non-standard manner, as you can see in this example from the collaborative shopping case study:

```
<?xml version="1.0"?>
<basket>
      <item>
            <name>pen</name>
            <quantity resolutor="greater">1</quantity>
            <price>1.15</price>
      </item>
</basket>
```

When programmers specify the value of this attribute, the middleware solves the conflict related to this element using the suggested policy. For instance we may consider this other document.

```
<?xml version="1.0"?>
<basket>
      <item>
            <name>pen</name>
            <quantity resolutor="greater">5</quantity>
            <price>1.15</price>
      </item>
</basket>
```

The reconciled document in this case will be the following:

```
<?xml version="1.0"?>
<basket>
      <item>
            <name>pen</name>
            <quantity resolutor="greater">5</quantity>
            <price>1.15</price>
      </item>
</basket>
```

We observe that in this simple case the middleware chooses the greater value according the specified policy.

XMIDDLE provides efficient solutions to address all these issues implementing a standard conflict resolution process that can be easily modified by using application-dependent policies. In the next paragraphs we detail these mechanisms provided by our middleware.

4.7.4 Standard conflict resolution

In the previous chapter we have described some common cases, considering data structures from possible applications based on XMIDDLE. Now we introduce and analyse the mechanisms and the solutions that allow our middleware to address the problem of the transparent reconciliation of different replicas with great flexibility. The reconciliation process in XMIDDLE involves three data structures: the two documents that have to be reconciled and their latest common edition; in fact, as we have discussed before, all the hosts that are linking to the same branch have at least the first edition in common.

As we have discussed before, it is necessary to distinguish, according to our definitions, between ordered and unordered data structures. In order to identify the type of tree, developers must use an attribute in the root element called `order`. This attribute can assume two values, `yes` or `no`; if it is not present the middleware assumes that the order is important. A possible example is the following:

```
<?xml version=”1.0”?>
<basket order=”no”>
	<item>
		<name>pen</name>
		<quantity>1</quantity>
		<price>1.15</price>
	</item>
	<item>
		<name>rubber</name>
		<quantity>1</quantity>
		<price>0.5</price>
	</item>
</basket>
```

Evidently, the reconciliation processes for an ordered and unordered trees are quite different; now we start analysing the case of an ordered tree.

4.7.4.1 Conflict resolution in an ordered tree

We will discuss the reconciliation process for ordered trees analysing the cases of the conflict resolution of value and structure conflicts separately.

In order to analyse the resolution of *value conflicts*, we consider the following two documents describing for example a colour image, which can be represented with an array of pixels using the RGB format. In this case, the order (from left to right) in the tree is important, since it represents the position in the array of pixels. We present a simplified document with only four pixels. For example, we may consider the following two documents:

```
<?xml version=”1.0”?>
<image order=”yes”>
	<pixel>
		<R>0</R>
		<G>0</G>
		<B>0</B>
	</pixel>
	<pixel>
		<R>0</R>
```

```
<?xml version=”1.0”?>
<image order=”yes”>
	<pixel>
		<R>0</R>
		<G>0</G>
		<B>0</B>
	</pixel>
	<pixel>
		<R>0</R>
```

```
            <G>0</G>
            <B>255</B>
      </pixel>
      <pixel>
            <R>255</R>
            <G>255</G>
            <B>255</B>
      </pixel>
      <pixel>
            <R>0</R>
            <G>0</G>
            <B>0</B>
      </pixel>
</image>
```

Doc_A

```
            <G>0</G>
            <B>255</B>
      </pixel>
      <pixel>
            <R>255</R>
            <G>255</G>
            <B>255</B>
      </pixel>
      <pixel>
            <R>255</R>
            <G>255</G>
            <B>255</B>
      </pixel>
</image>
```

Doc_B

We notice that the differences in these documents are related to the fourth pixel; in order to reconcile them we have to compare them with the latest common edition. Otherwise, the choice of the "winning document" (using a Lotus Notes terminology) would be arbitrary. By this approach we utilise the *common history* of the documents in order to decide how to solve the conflict.
For example let us assume that the latest common edition is the following:

```
<?xml version="1.0"?>
<image order="yes">
      <pixel>
            <R>0</R>
            <G>0</G>
            <B>0</B>
      </pixel>
      <pixel>
            <R>0</R>
            <G>0</G>
            <B>255</B>
      </pixel>
      <pixel>
            <R>255</R>
            <G>255</G>
            <B>255</B>
      </pixel>
      <pixel>
            <R>0</R>
            <G>0</G>
            <B>0</B>
      </pixel>
</image>
```

Doc_{old}

In this case XMIDDLE chooses the values that are present in Doc_B, because these are different from those that are contained (in the same positions) in Doc_{Old} and, at the same time, the values of Doc_A are equal to those in Doc_{Old}. In other words, after the last reconciliation process, $Host_B$ has modified his replica and so we can consider the copy stored in $Host_A$ as *stale*. In the symmetric case (Doc_B is a stale copy) the middleware chooses the values that are present in Doc_A.

The case of different values (in the same positions) in all the three documents is the most problematic, because we are not able to make a decision; we implement different behaviours in the prototype of our system: it is possible to opt for a random choice between the two values or to introduce a priority, expressed by a number in the root of the document (if the numbers are equal, we decide randomly between the values of the two copies).

We summarize these concepts in the following table using three different letters (a, b, c) to indicate different values in the same position.

Value in Doc_A	**Value in Doc_B**	**Value in Common Latest Edition**	**Choosen value**
a	b	a	b
a	b	b	a
a	b	c	a or b (priority-based choice)

Table 4.1 Resolution of value conflicts in an ordered tree

Now we analyse the case of the *structure conflicts*; as discussed before the "logical unit" in XMIDDLE is a branch; for this reason, we do not use a particular XML structure to describe how the reconciliation works, but generic tree diagrams. First of all, it is worth noting that we adopt the same approach used for value conflicts, exploiting the opportunity of comparing the different copies with the latest common edition.

We will study a simple case with a tree with sub-trees that are directly children of the root node; more complex structures are managed in the same way. Let us consider the following two diagrams representing Doc_A (replica in $Host_A$) and Doc_B (replica in $Host_B$) that are showed in Figure 4.6.

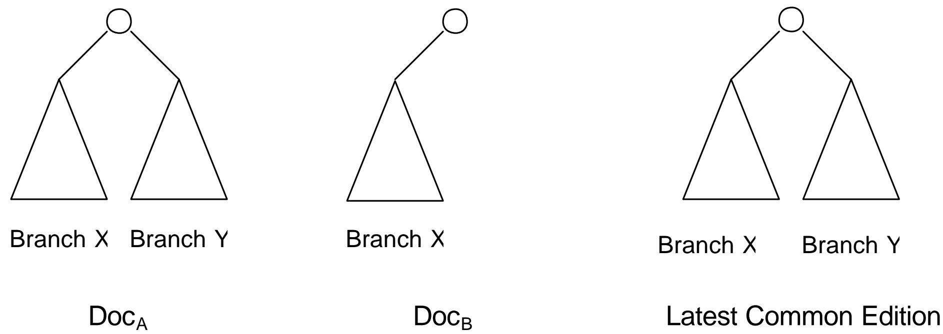

Figure 4.6 Structure conflict reconciliation in an ordered tree (example 1)

In order to reconcile these two data structures, we consider the latest common edition. We notice that in Doc_B there is not a sub-tree that is present in both Doc_A and in the common reconciled document: probably this is the result of the deletion of this branch performed by the application running on $Host_B$, after the last reconciliation process. Therefore, the middleware chooses the structure of Doc_B, deleting branch Y from Doc_A. Let us consider another situation, represented in Figure 4.7.

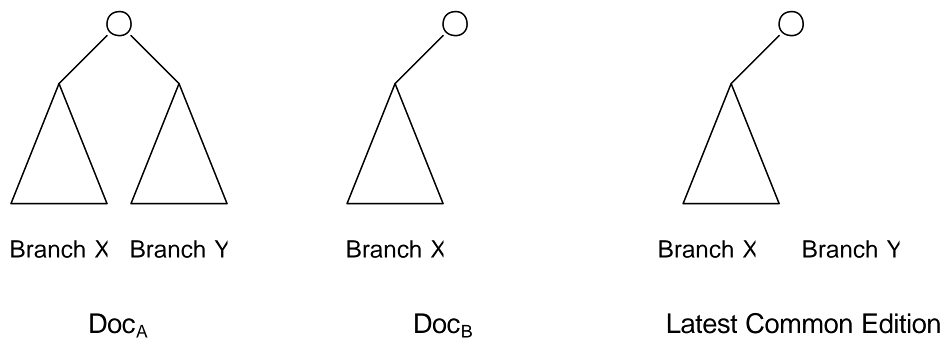

Figure 4.7 Structure conflict reconciliation in an ordered tree (example 2)

In this case branch Y is present in Doc_A, but not in Doc_B and in the latest common edition; probably this means that branch Y has been added after the last reconciliation process. Therefore, in order to solve this conflict, we have to add branch Y in Doc_B. The symmetric cases are dealt with in similar ways. We can sum up these concepts in this table:

Doc_A	Doc_B	Latest Common Edition	Reconciled document
Sub-tree T present	Sub-tree T not present	Sub-tree T present	Sub-tree T not present.
Sub-tree T not present	Sub-tree T present	Sub-tree T present	Sub-tree T not present
Sub-tree T present	Sub-tree T not present	Sub-tree T not present	Sub-tree T present
Sub-tree T Not present	Sub-tree T present	Sub-tree T not present	Sub-tree T present

Table 4.2 Resolution of structure conflicts in an ordered tree

4.7.4.2 Conflict resolution in an unordered tree

The conflict resolution in an unordered tree is more complex, since we need to compare elements (and sub-trees) that may be in different positions in the tree structure.

First of all, we study the resolution of value conflicts in these data structures. It is worth noticing that, in this case, the process chooses a document as base and performs the change on this in order to obtain the reconciled document. For example, we can consider these two documents (for the moment, we assume that the two hosts has only an empty edition in common) with Doc_A as “base document”:

```
<?xml version="1.0"?>
<basket order="no">
  <item>
    <name>pen</name>
    <quantity>1</quantity>
    <price>1.15</price>
  </item>
</basket>
```

Doc_A

```
<?xml version="1.0"?>
<basket order="no">
  <item>
    <name>pencil</name>
    <quantity>4</quantity>
    <price>0.75</price>
  </item>
</basket>
```

Doc_B

The middleware detects three value conflicts in this document (the name of the product, its quantity and its price); the order of sub-tree in this case is not important, since from an application point of view it is not relevant (the items may be purchased in a different sequence). For this reason, the correct way to perform

the reconciliation is to generate a document with two items, pen and pencil; in other words we have to *duplicate* the `item` structure.

XMIDDLE exploits the expressiveness of XML Schema in order to identify the structure of sub-trees correctly. This process is composed of the following steps: the middleware detects the value conflict (for example pen and pencil in the same positions) and reconstruct the branch structure in Doc_B (in this case the `item` sub-trees) in which the inconsistency occurs. Afterwards, it checks whether the branch that has been identified is present in Doc_A in *any* position. If it is not present the middleware adds it to the base document, otherwise Doc_A is not modified. It is worth noticing that this process is not performed in the case of application-specific resolution (in other words, when a non standard resolutor is specified); we will discuss this aspect later in this chapter.

We underline that XMIDDLE also checks whether it is present in the latest common edition using the same rules that we have examined before, when we have discussed the case of ordered trees. In other words, only if the sub-tree is not present in the base document and in the latest common version, the middleware performs the branch addition. In order to understand this aspect, we will examine an example in detail, analysing each step of the reconciliation process. Let us consider the tree structure in Figure 4.8, indicating with different letters sub-trees with inconsistent structures or values.

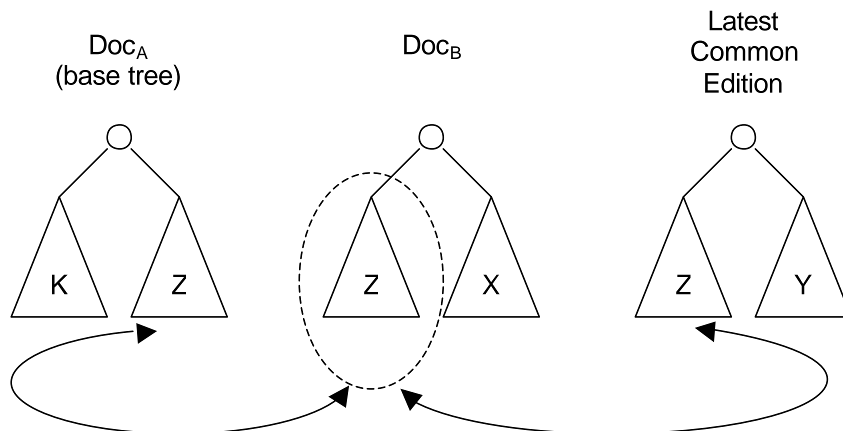

Figure 4.8 Example of the reconciliation process for unordered trees (comparison between sub-trees).

Doc_A is the base document; in other words, we compare each branch that is present in Doc_B with those that are present in Doc_A and in the common latest edition.
Firstly, we consider branch Z, as you can see in the figure. This sub-tree is present in Doc_A and in the Latest Common Edition and for this reason it is not added to the base tree.
Afterwards, we consider the branch marked with the letter X; since it is not present in the other two documents, it is added to the base tree. In fact, this is the case of an addition of a sub-tree after the last synchronization.
The reconciliation process is not completed after these two steps; in fact we have to perform the same comparisons with the other tree structure considering Doc_A: the branch marked with the letter K is not present in the other documents and it is maintained in the base tree; the other sub-tree is not deleted as well, since it is present in Doc_B and in the last common edition. Furthermore, we underline that a branch has to be deleted from the base tree only if it is present in the last common edition and not in the other document.
We can sum up these concepts using the following table (T is a sub-tree of Doc_B):

Sub-tree T present in base document	Sub-tree present in latest common edition	Operation performed on base document
Yes	No	Delete sub-tree T from base document
Yes	Yes	None
No	No	Add sub-tree T to base document
No	Yes	None

Table 4.3 Resolution of structure conflicts in unordered trees.

It is worth noticing that this process is extremely optimised, using several techniques, for example to avoid unnecessary comparisons; we will describe its implementation in details in the next chapter. However, the synchronization is quite complex from a computational point of view and introduces a remarkable overhead, especially for large data structures. On the other hand, the reconciliation process takes place only after a modification of the data structure (if the hosts are

connected) or a reconnection between two hosts. The frequency of this event is generally tolerable and, anyway, the advantages that derive from this mechanism are noteworthy. Furthermore, we underline that the communication between the hosts is very limited, since the reconciliation process is not based on a complex negotiation phase.

4.7.4.3 The branch structure problem

In the previous paragraph we have presented the XMIDDLE synchronization process for an unordered tree structure that is essentially based on the comparisons between sub-trees; now we discuss the techniques that are used in our middleware to reconstruct the topology of the branches that must be compared, starting from their leaves (in an XML documents, the text nodes).

Our solution is to exploit XML Schema, since it represents the most expressive standard language for the description of XML documents and at the same time it is extremely simple to use. We use it essentially in order to define the maximum number of possible occurrences of a node in a certain level of the data structure that we are considering. In fact, in other words, our goal is to find the root of the sub-tree; this is usually characterised by not having a predefined maximum number of occurrences.

We start considering the leaves of the tree where the value conflict has been detected (more precisely the node that is the parent of this text node). Analysing the corresponding XML Schema document, if the maximum number of occurrences of this element is unbounded, this is the root of the sub-tree. Otherwise, we consider its parent node and we repeat the same test, recursively. Obviously, this is not a general and standard definition of a branch; however, our approach is suitable for a very large class of applications and developers can modify this standard resolution process by using application dependent policies.

Moreover, it is worth noticing that it is necessary to perform this procedure only once, since it is sufficient to know the root of only one sub-tree to determine the parent node of all branches.

Now we analyse an example of XML Schema, using the collaborative shopping case study; the XML Schema document is the following:

```
<xsd:schema xmlns:xsd="http://www.w3.org/1999/XMLSchema">
```

```
 <xsd:element name="basket">
  <xsd:complexType>
   <xsd:element name="item" type="xsd:string" minOccurs="0"
maxOccurs="unbounded">
    <xsd:complexType>
     <xsd:element name="name" type="xsd:string"/>
     <xsd:element name="price" type="xsd:decimal"/>
     <xsd:element name="quantity">
      <xsd:complexType base="decimal">
       <xsd:attribute name="resolutor" use="default" value"add"/>
      </xsd:complexType>
     </xsd:element>
    <xsd:complexType>
   </xsd:element>
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

In XML Schema you can specify the minimum number of times that an element appears with the `minOccurs` attribute and the maximum number with the `maxOccurs` attribute. The default value for `minOccurs` is 1; if you do not specify a value for `maxOccurs`, its default value is the value of `minOccurs`. To indicate that there is no upper bound to the `maxOccurs` attribute, it must set to the value `unbounded`.

Developers have to provide only this simple document in order to enable the XMIDDLE sophisticated automatic reconciliation process. This is a process that we call *XML Schema document registration*. In the next chapters, we will illustrate the guidelines for designing applications based on our middleware and we will also detail this aspect.

4.7.5 Application-specific conflict resolution policies

In the previous paragraphs we have analysed the standard reconciliation process that is automatically performed by the middleware and provide the conflict resolution system with the built-in rules that we have discussed before.

XMIDDLE also supports the definition of application-specific policies for the management of data conflicts. This is achieved through the definition of the `Resolutor` attribute in the elements that have to be treated using a specific resolution strategy. When this attribute is present, the middleware does not apply the standard method to manage the possible inconsistency in this element (more

precisely, using a DOM representation of the tree, in the text node that is child of this element), but uses specific procedures. Some examples of available application specific policies for the resolution of conflicts of numerical values are `greater`, `lesser`, `add`, `random`, and `average`. In order to understand how this mechanism works, let us consider these following documents:

```
<?xml version="1.0"?>
<basket order="no">
      <item>
            <name>pen</name>
            <quantity resolutor="greater">1</quantity>
            <price>1.15</price>
      </item>
      <item>
            <name>rubber</name>
            <quantity resolutor="greater">1</quantity>
            <price>0.5</price>
      </item>
</basket>
```

$\texttt{Doc}_B$

```
<?xml version="1.0"?>
<basket order="no">
      <item>
            <name>pen</name>
            <quantity resolutor="greater">2</quantity>
            <price>1.15</price>
      </item>
</basket>
```

$\texttt{Doc}_B$

In this example, we use an application specific policy to solve the possible conflicts in the `quantity` element. The policy chosen by the developer in this case is `greater`. In this case, with respect to the structure conflicts and the other value conflicts, the resolution process is performed in the usual manner, but we have to underline that, when the item describing the `pen` product in the second document is compared to those which are present in the first document (base document), the middleware does not check the values of the text nodes that are children of the quantity node, because of the presence of the `resolutor` attribute. In other words the middleware considers the two sub-trees in the first

and the second document equal and for this reason it does not add another branch to the base document.

However, after the detection of the presence of an “equal” sub-tree in the base document, XMIDDLE analyses the structure of the document and, for each element with a `resolutor` attribute, it performs the corresponding specific conflict resolution process. In our case, the middleware uses the `greater` resolutor; according to this, as the name suggests, the greater value is chosen (in our example two). Therefore, the reconciled document is the following:

```
<?xml version="1.0"?>
<basket order="no">
	<item>
		<name>pen</name>
		<quantity resolutor="greater">2</quantity>
		<price>1.15</price>
	</item>
	<item>
		<name>rubber</name>
		<quantity resolutor="greater">1</quantity>
		<price>0.5</price>
	</item>
</basket>
```

It is worth noticing that these resolutors also consider the latest common edition to manage the resolution of conflicts; for example, let us examine the following documents:

```
<?xml version="1.0"?>
<values>
  <value resolutor="add">
  2
  </value>
<values>
```

Doc_A

```
<?xml version="1.0"?>
<values>
  <value resolutor="add">
  1
  </value>
<values>
```

Doc_B

We assume that these documents have generated the following common latest edition that is used during the reconciliation process.

```
<?xml version="1.0"?>
<values>
  <value resolutor="add">1</value>
<values>
```

Doc_{old}

In this case, if we do not consider the common latest edition, the reconciled value for the value element is 3; but it is not correct, because only the value in Doc_A is different from the latest common edition (it is incremented by 1); for this reason, we use the following simple formula:

$$ReconciledVal=(val_A-val_{LatestCommonEdition})+(val_B-val_{LatestCommonEdition})$$

If the result is negative we set the value to 0; for example, this is the case with these values: $val_A=0$, $val_B=0$ and $val_{LatestCommonEdition}=1$.
Therefore, in our example the reconciled document will be the following:

```
<?xml version="1.0"?>
<values>
  <value resolutor="add">2</value>
<values>
```

Let us consider another example; we analyse the following situation, where apparently there is no conflicts:

```
<?xml version="1.0"?>
<values>
  <value
resolutor="add">1</value>
<values>
```

Doc_A

```
<?xml version="1.0"?>
<values>
  <value
resolutor="add">1</value>
<values>
```

Doc_B

Let us assume that the latest common edition is the following document:

```
<?xml version="1.0"?>
<values>
  <value resolutor="add">0</value>
<values>
```

It is worth noticing that the application specific resolution is performed even if there are no inconsistencies between the two documents, since it is triggered only by the presence of `resolutor` attributes; this is fundamental in order to obtain a correct synchronization process of the replicas in each host.

In fact, some resolutors, such as `add`, analysing also the latest common edition can update the values considering the "history" of the document in a correct manner. In this case, for example, $Host_A$ and $Host_B$ have changed the value from 0 to 1. Only by the observation of these documents we cannot detect any conflict, but the middleware, checking the latest common edition, finds the changes and choices the correct up-to-date value; therefore, the reconciled document will be the following:

```
<?xml version="1.0"?>
<values>
  <value resolutor="add">2</value>
<values>
```

We also underline that it is possible to use these application-specific resolution policies in both ordered and unordered tree structures. It is worth noting that developers can use various types of resolutors in the same document.

Chapter 5
Implementation of XMIDDLE

In this chapter we present the implementation of the prototype of our middleware, underlining some remarkable aspects and identifying possible general solutions to common design issues.

5.1 Preliminary considerations about the implementation of XMIDDLE

Before describing the architecture of our system, we point out some general design choices that we made during the development of XMIDDLE.

5.1.1 XMIDDLE framework and implementation

The key words of the implementation of XMIDDLE are modularity and extensibility; this approach is a natural consequence of the adoption of object-oriented techniques in the design of our system [Jac94]. We use the Java language that allows developers to define complex object-oriented architecture providing a complete set of programming structures such as classes, abstract classes, interfaces and packages and inheritance mechanisms.

Essentially, the XMIDDLE prototype consists of two parts, an abstract *framework* and its *implementation*. The de-coupling between framework and implementation adds flexibility and introduces modularity in the project, according to the principle of reusability; at the same time, the definition of a general framework specifies the structure for the future implementations of the middleware. The classes of the XMIDDLE framework are packaged under `edu.UCL.xmiddle.framework`, the classes of the implementation under `edu.UCL.xmiddle`.

5.1.2 Networking issues

As already discussed, the current implementation of XMIDDLE is based on UDP over IP. UDP has been chosen over TCP, as it is more suitable than TCP for a mobile ad-hoc setting. In fact, even if TCP provides a more reliable service, it needs to establish a connection before the communication and introduces remarkable packet overhead not appropriate in relation to the average wireless bandwidth usually available. On the other hand, UDP does not guarantee packet delivery and does not prevent duplicate data messages. For this reason XMIDDLE needs to handle data packet duplication and loss. However, during our testing process, we used the 802.11 wireless technology, observing an appreciable reliability as concerns packet loss.

5.1.3 XML technologies

XMIDDLE uses DOM technology [W3C02b] to access XML documents; this was an important preliminary decision in the design of our middleware. In fact another alternative solution to parse XML documents using the SAX standard interface [SAX02] (that stands for Simple API for XML); this provides an event-based framework for accessing XML documents, that are parsed sequentially: when the SAX parser encounters an element or a processing instruction, it treats them as events and calls the corresponding registered listeners. The most remarkable advantage of this technology is that it does not require a large amount of memory in order to perform the parsing process.

DOM is an object-oriented technology for fast random access and manipulation of XML documents (we describe it in Appendix A). We chose DOM technology on the basis of the following considerations. Firstly, random access to data is essential for a large number of applications and this is not provided by SAX parsers. Secondly, DOM represents data using trees that are classic and powerful structures, providing also a complete API in order to traverse and manipulate them; this is a fundamental feature, since we need to perform complex operations especially during the reconciliation process. Finally, in an object-oriented context, DOM is preferable, since it maps XML documents to object structures in a very easy and elegant way.

5.1.4 XMIDDLE and the Java platform

We implemented XMIDDLE using Java [Sun02e], the widely used platform independent language from Sun Microsystems. We chose it because of its outstanding characteristics: firstly, it provides a platform independent networking and multi-threading support; secondly, a very large number of reliable and optimised XML technologies are available for this platform; finally, in recent years wireless-oriented versions of this language have been developed. The current implementation of XMIDDLE is compliant with the PersonalJava specification [Mah02]. We used Xerces Java Parser [Apa02a] as XML and DOM parser and Xalan-Java [Apa02b] as XPath processor [ClaD99]; they were chosen for their wide distribution and appreciation in industry and in the research community. More in details, Xerces Java Parser supports the XML 1.0 recommendation and also provides a support for the W3C's XML Schema Recommendation version, DOM Level 1 and DOM Level 2. Xalan-Java is essentially an XSLT processor for transforming documents into other formats (such as HTML or WML); it implements the W3C Recommendations for XSL Transformations (XSLT) [Cla99] and XPath standard language. These software components are very easy to use, since they provide a simple API, supported by a detailed documentation. It is possible to find a presentation of the essential aspects of these technologies in Appendix A.

5.2 System architecture

Now we analyse the XMIDDLE architecture considering its fundamental components and fundamental entities; the platform is essentially composed of the following modules:

- Manager, which provides the access point to the middleware services for applications, supplying a simple API for these to access data, request linking, view hosts that are in reach, connect and disconnect, and to perform other secondary operations; moreover, it is worth noticing that XMIDDLE applications have only access to this component, which, in a sense, abstracts away from the rest of the middleware;

- Network Controller, which handles all network connections and manages all communications among hosts;
- Locator, which identifies the hosts that are in reach and is responsible for keeping the list of available hosts up-to-date;
- Tree, which stores all application information;
- LocalHost, which is responsible for accessing and modifying the local data tree structure. It also manages protocols requests (for example linking operation and reconciliation process);
- Protocol, which represents the protocols that XMIDDLE can handle;
- ProtocolChooser, which essentially controls the execution of protocols; it is worth underlining that our middleware also supports a mechanism that allows the negotiation of protocols and their dynamic load.

5.3 The Manager module

The Manager module provides a "high-level view" of the XMIDDLE platform; it represents the only access point that applications have to the middleware, providing a unified and simple API to developers. It is implemented by `edu.UCL.xmiddle.SimpleManager`, which inherits from the interface `edu.UCL.xmiddle.framework.Manager`. Manager includes methods to connect and disconnect from the network, initiate protocols, send data to other hosts, access data from the tree, load and manipulate XML documents.

The most important ones are the following:

- `void init()` starts the XMIDDLE platform, initialising each module of the system.
- `boolean connect()` performs the connection to the network. It returns the boolean value `true` if this operation has been successful, `false` otherwise. This provides a mechanism to request an explicit connection to the network. After a connection, the platform searches for the hosts that are currently in reach and eventually reconciles the local data with other nodes.
- `boolean disconnect()` allows applications to disconnect explicitly from the network.

- `void link (Host host, Integer appID, String remoteElement, Integer localAppID, Element localRoot)` allows applications to link a remote element stored on another host. `host` is the host on which the exported element resides; `appID` is the identifier of the remote application; `remoteElement` is an XPath expression that specifies the position of the element to be linked; `localAppID` is the local application identifier; `localRoot` is the local root element under which the linked element will be placed.
- `void unlink (Element element, Integer appID)` stops linking to the specified `element`. `appID` is the identifier of the application running on the host on which the element is stored.
- `void export(String path, Integer appID)` allows applications to export a sub-tree of the local tree specified by the `path` string, which is an XPath expression that must identify a unique element in the document. The second argument is the application ID of the application that exports this branch of the local tree. This method also creates the version 0 related to this exported element.
- `Element open (String path, Integer appID)` returns the latest version or edition of the sub-tree specified by the `path` string (an XPath expression). This must identify a unique element of the tree. `appID` is the identifier of the application that creates the tree structure.
- `void close(Element element, Integer appID)` "commits" the performed changes. In fact, an application after accessing `element`, must close it, allowing the middleware to update the local tree and start the reconciliation process.
- `void send Data(Data data)` allows applications to send data (using the XMIDDLE object `Data`, which contains the receiver address and the information to be sent).

We will introduce other methods of the XMIDDLE application interface later in this chapter, when we will discuss in details some communication issues.

5.4 The network controller

The network layer of XMIDDLE is implemented by the `edu.UCL.xmiddle.framework.controller` package. The corresponding framework package is `edu.UCL.xmiddle.controller`.

The most important classes of this package are `Data`, `UDPNetwork` and `UDPLocator`.

The `Data` class provides a representation of the data packets that are exchanged by hosts; it includes methods to set (and get) the data contained in the packet and the host to which the data are addressed (or from which the data were received).

The `UDPNetwork` class (which inherits from the framework class `Network`) implements the thread that manages all network connections; it is responsible for sending and receiving `Data` packets, including protocol requests, which are queued to the `LocalHost` module. It also provides other functionalities related to the execution of protocols that we will describe later.

Finally, the `UDPLocator` class (which inherits from the framework class `Locator`) is responsible for detecting the hosts that are currently in reach. Clearly, it includes methods to get an up-to-date list of the host in reach.

5.5 Host and tree structure representation

The elaboration of the messages received from other hosts and the coordination of the operations performed on the local tree are provided by the `LocalHost` class (which implements the `Runnable` interface).

As the name suggests, this models the local host (as concerns elaboration and coordination), providing methods for queuing protocol requests, handling the local tree and exporting elements. For example, it provides methods to load an XML document into the local data structure and to manage the local tables describing the relationships with other hosts (i.e., the *exportLink*, *linkedBy* and *linkedFrom* tables).

The class that is used to represent the tree structure stored by the middleware is `LatestTree`, which inherits from the abstract class of the framework `Tree`. It includes methods to access or modify specific elements or collections of elements using XPath expressions. It encapsulates the latest version of the tree and the

previous editions released by the host that stores the data structure. In the previous chapter we have presented the versioning system of XMIDDLE; it is worth noting that we implement it in an optimised way to avoid an excessive amount of different versions stored in the memory of each host, keeping only the versions that are really necessary to perform the reconciliation process.
Another interesting aspect is the modelling of the tables containing the linking relations between hosts. This information is represented by means of the `LinkTable` class; this class provides methods in order to create, access and modify the *exportLink*, *linkedBy* and *linkedFrom* tables. It is possible to retrieve the instance of the `LocalHost` class by using the `getLinks()` method of the `LocalHost` class.

5.6 Protocols

In this section we present some issues related to the implementation of protocols in XMIDDLE, focusing, in particular, on the reconciliation process.

5.6.1 Protocol execution

5.6.1.1 Registration of new protocols

The XMIDDLE framework provides a mechanism to dynamically load new protocols; each class that implements a protocol must inherit from `edu.UCL.xmiddle.framework.lib.protocols.Protocol` and must follow some specific guidelines that we will describe in the following sections.
The registration procedure of protocols is handled by the Manager (using the `Manager.registerProtocol()` method); new protocols are registered in an instance of the `ProtocolRegistry` class; this can return any registered protocol by its name.
The implementation of the XMIDDLE prototype includes the linking and the reconciliation protocols. We have described them in Chapter 4, underlining that we can consider the first as a particular case of the second one; for this reason we designed only one class to implement them; its name is

edu.UCL.xmiddle.lib.protocols.Reconciliation and the protocol is called "SYNC".

5.6.1.2 Protocol sessions

A protocol session is defined as a structured network communication between two or more XMIDDLE hosts. The module that manages the negotiation in order to start a protocol session is LocalHost; the first step is the generation of a request of a new session to the involved hosts (these requests are handled by the corresponding LocalHost instances). Requests can be sent automatically by the middleware (for example, this is the case of the automatic reconciliation process) or applications can explicitly send requests by calling the send() method provided by the Manager class. With respect to the first case, we may consider, for example, the reconciliation process, that is triggered automatically by the middleware.

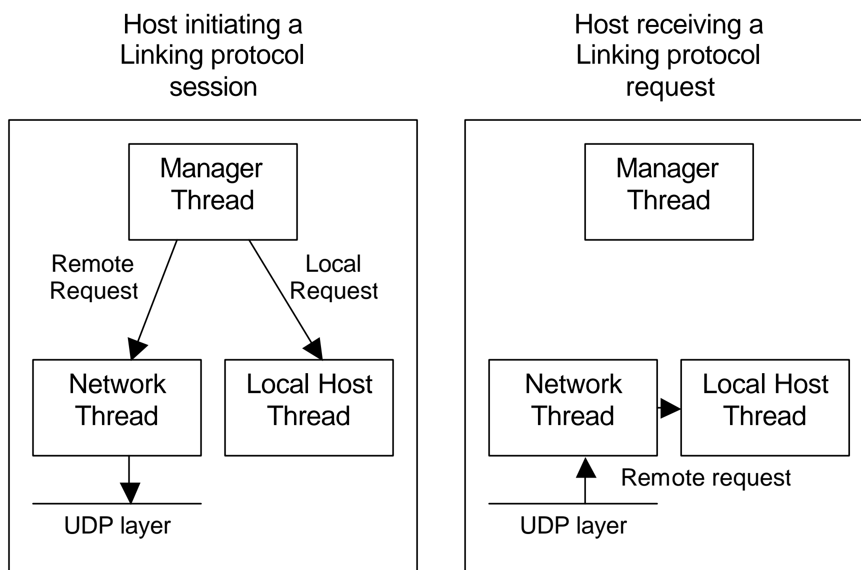

Figure 5.1 Modules involved in the negotiation before a linking protocol session

As it is possible to observe in Figure 5.1, remote protocol requests received by the Network thread are automatically passed to the LocalHost thread.

When a LocalHost module receives a protocol request, it starts a ProtocolChooser thread in order to perform the service.

5.6.1.3 The ProtocolChooser thread

The `ProtocolChooser` thread is responsible for the following tasks during the execution of a protocol:

- negotiating the appropriate protocol to be executed;
- supervising communications between hosts;
- finding the negotiated protocol from the `ProtocolRegistry` instance or downloading it from one of other hosts;
- starting and synchronizing the protocol during its execution.

When it starts, the `ProtocolChooser` thread firstly generates a unique session identifier corresponding to that particular protocol session. This is generated independently on each host and, at the same time, it has to be univocal. XMIDDLE uses the primary identifiers of all participants concatenated in descending order and parts of the protocol request to create the session identifier. For example, we may consider two hosts H_A and H_B that are performing a linking protocol (see Figure 5.2); let us assume that H_A is identified by the primary identifier 4444, whereas H_B by 5555 and the request is "LINK to /root/one". In this case, the session identifier will be automatically set to `LINK/root/one5555-4444`.

After the generation of the session ID, `ProtocolChooser` registers the session with the `Network` instance. This uses this information to create the `UDPListener` (which inherits from the framework class `Listener`) and `UDPSender` (which inherits from the framework class `Sender`) objects and returns the corresponding object references to the `ProtocolChooser`. `UDPListener` and `UDPSender` are responsible respectively of receiving from and sending data to a specific host (which is specified upon the creation of the instances).

Furthermore, it is worth noting that the `ProtocolChooser` threads can also negotiate directly the communication protocol, bypassing the networking modules. Protocols are stored in a protocol registry (modelled by the `ProtocolRegistry` class); if the agreed code for the protocol is not available, it can be transferred from another host. Afterwards, instances of the `Protocol` objects are created and their references are passed to the `UDPListener` and `UDPSender` objects.

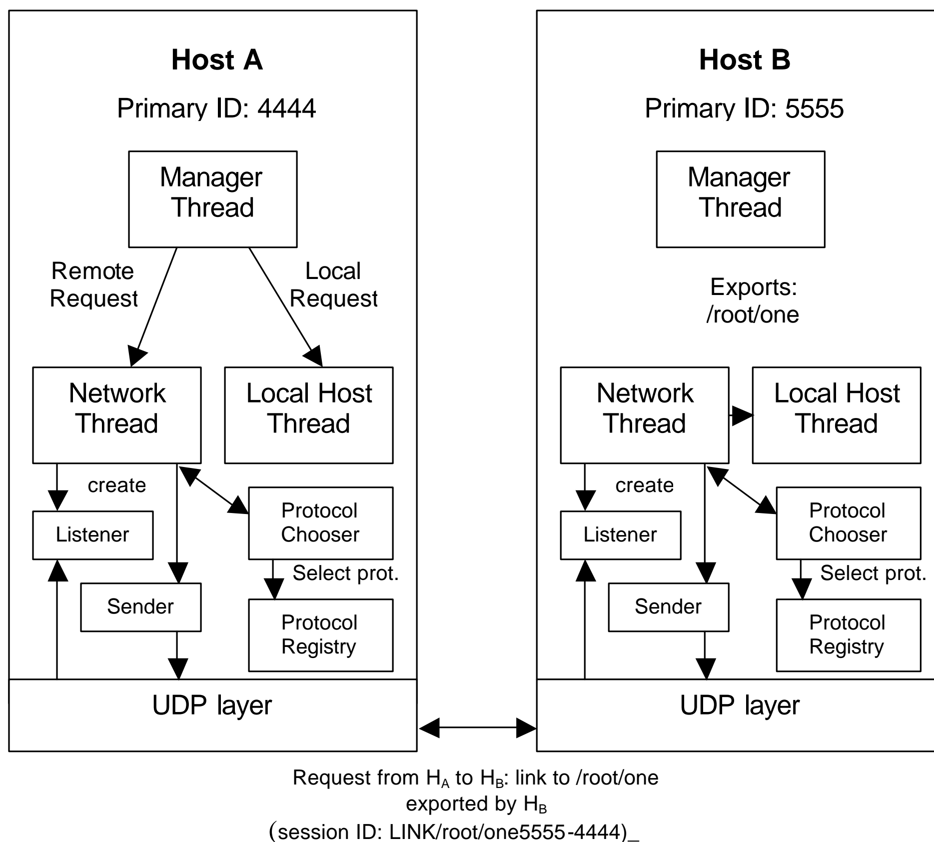

Figure 5.2 Example of a linking protocol session

Another interesting issue is the order of transmission (in other words, the choice of the host that has to send the first message). XMIDDLE deals with this problem defining two types of sessions, *active* and *passive*. Each host involved in the execution of a protocol performs a type of session: we label as *active* the host that sends information first and with the term *passive* to the host that waits a message from another one before starting interaction. In order to decide on the type of the session, XMIDDLE exploits the numerical order of the primary identifiers of the involved hosts.

The `ProtocolChooser` thread is also responsible for the operations that are performed at the termination (that may be successful or not) of the execution of the protocol: for example, they have to free memory and network resources and de-register themselves from the networking modules. We also point out that it is possible to interrupt the execution of a protocol using the `ProtocolChooser.abort()` method.

5.6.1.4 Development of a protocol for XMIDDLE

In this section we will describe the process of the development of a protocol for XMIDDLE in details. First of all, XMIDDLE protocols have to inherit from `edu.UCL.xmiddle.framework.lib.protocols.Protocol`. Furthermore, essentially, they must implement two methods, `execute()` and `abort()`. The first one starts the execution of the protocol and it is called by the `ProtocolChooser` class; the second one aborts the execution. It is also expected that every protocol is able to provide rollback functionalities in case of the failure of transaction-oriented operations.

Moreover, as we have discussed before, protocols must have a unique name, under which they will be registered with the `ProtocolRegistry` module; they are also expected to provide the implementation for the two types of XMIDDLE protocol sessions (active and passive).

The constructors of protocols have to be designed to accept the following values (in this order): a `Listener` object (from which the protocol can receive data sent by the other hosts involved in the protocol session), a `Sender` object (using which, the protocol can send data to the other hosts that are involved in the session), the session type (which can assume two values `Protocol.ACTIVE` and `Protocol.PASSIVE`), a `LocalHost` object (using which, the protocol is able to access the data structures that are stored in the host), the session identifier, the identifier of the remote host involved in the session and an array of Java objects that contains the parameters that are necessary to perform the protocol (for example, the reconciliation protocol uses an additional parameter to specify the sub-tree that has to be reconciled).

One of the future improvements of XMIDDLE will be a mechanism that allows applications to specify and use new protocols defined by developers.

5.6.2 Reconciliation protocol

5.6.2.1 Implementation of the reconciliation algorithm

The reconciliation algorithm is implemented through a set of classes that are responsible of handling the protocol and representing the structures that are

involved during its execution (i.e., editions). The class that implements the reconciliation protocol is `Reconciliation`. This has been developed according the model that we have described in the previous section of this chapter. The execution of the protocol is based on the functionalities provided by a certain number of classes that are responsible for implementing the algorithms that enable the reconciliation of distributed and inconsistent replicas of XML documents. These classes are essentially the following:

- `LevelTreeDiff` class, which implements the LevelTreeDiff algorithm;
- `LevelTreeMerge`, which implements a "merge" algorithm using a base XML document and a "diff" XML document (it can be considered as the "inverse operation" of the LevelTreeDiff algorithm);
- `LevelTreeReconcile`, which implements the reconciliation algorithm.

We have presented and discussed in details these algorithms in Chapter 5; in next sections we will analyse briefly some details of their implementations.

5.6.2.2 The `LevelTreeDiff` class

The `LevelTreeDiff` class allows to compare two XML documents, returning a "diff file", which is an XML documents that contains the differences between them according to an "operational" notation. Given two documents Doc_A and Doc_B, this file specifies the operations that have to be performed on Doc_A to obtain Doc_B.

The main method of this class is `compute()` that, given a base document and a "changed" DOM document as arguments, produces a "diff" document with the format defined by the following DTD specification:

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT treediff
(addsubtree,delsubtree,changeattr,changepcdata)*>
<!ELEMENT addsubtree ANY>
<!ATTLIST addsubtree
      xpathparent CDATA #REQUIRED
      xpathleftsibling CDATA #REQUIRED
      insertOrder CDATA #REQUIRED>
<!ELEMENT delsubtree EMPTY>
<!ATTLIST delsubtree
      xpath CDATA #REQUIRED>
<!ELEMENT changeattr (attribute)>
<!ATTLIST changeattr
      xpath CDATA #REQUIRED>
<!ELEMENT attribute EMPTY>
```

```
<!ATTLIST attribute
      name CDATA #REQUIRED
      oldvalue CDATA #REQUIRED
      newvalue CDATA #REQUIRED>
<!ELEMENT changepcdata EMPTY>
<!ATTLIST changepcdata
      xpath CDATA #REQUIRED
      oldvalue CDATA #REQUIRED
      newvalue CDATA #REQUIRED>
```

5.6.2.3 The `LevelTreeMerge` class

The “diff “ document that is produced by the `compute()` method of the `LevelTreeDiff` class (with two documents in input as arguments, a base document and a modified one) is used by `LevelTreeMerge`.

This class, applying the modifications described by means of the “diff” document to the base document, is able to generate the modified document that has been used as an argument of the LevelTreeDiff algorithm. In other words, this class is used to reconstruct an XML document from a base document, knowing the changes that have been made on it. We have described this *merge* operation as the inverse of the operation performed by the LevelTreeDiff algorithm; we can motivate this assertion using the example showed in Figure 5.3.

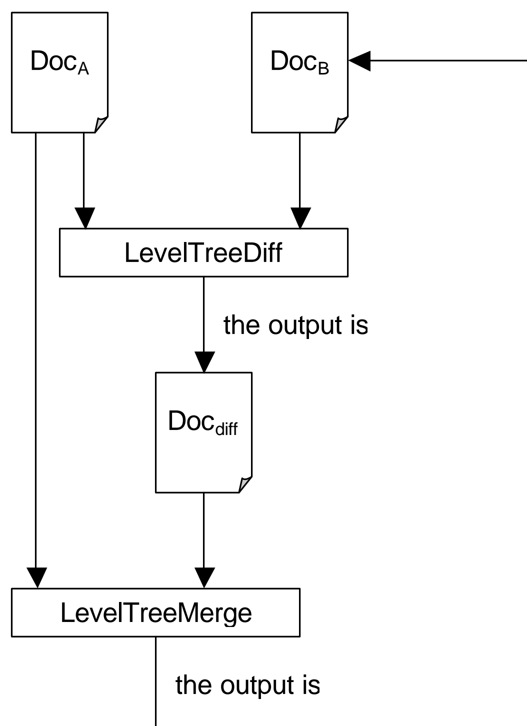

Figure 5.3 Use of the `LevelTreeDiff` and `LevelTreeMerge` classes

As you can see from this example, the `LevelTreeDiff` class (by means of the `compute()` method) generates the "diff" document Doc_{diff} that contains the operations that must be performed on Doc_A in order to obtain Doc_B; moreover, using the `LevelTreeMerge` class (by means of the `merge()` method) it is possible to reconstruct Doc_B from Doc_A using Doc_{diff}.

5.6.2.4 The `LevelTreeReconcile` class

The `LevelTreeReconcile` class is responsible of implementing the reconciliation algorithm. Its most important method is `reconcile()`; the arguments of this method are the two XML documents that have to be reconciled; the output of this class is the reconciled document. The order of the arguments is not important in this case. `LevelTreeReconcile` exploits the LevelTreeDiff algorithm in order to find the differences between the two XML documents. The reconciliation of these is based on the techniques described in Chapter 4.
The exploration of the DOM tree structures is performed using recursion in an optimised way. `LevelTreeReconcile` is able to deal with the reconciliation of ordered and unordered trees (using the terminology used in the previous chapter), analysing the `order` attribute of the root node of the two XML documents to be reconciled. If the order attribute is not expressed, this class treats these documents as ordered trees.

5.6.2.5 Implementation of the reconciliation process

To better understand the reconciliation algorithm, we analyse an example from an implementation point of view, without considering all the details (i.e., generation of the edition numbers, etc.) that we have discussed in the previous chapter.
$Host_B$ starts the reconciliation protocol; we define the operations performed by the application that is running in this host as *local*; we identify the other host involved in this process as $Host_A$; it receives the request to perform the protocol from $Host_B$ and therefore we name the operations performed in this host as *remote*. It is worth noting that we also used this terminology in the development of the `Reconciliation` class: the two main methods corresponding to the active and the passive sessions are respectively called `ExecuteLocal()` and

`ExecuteRemote()`. In other words, in order to perform the reconciliation, after the identification of the type of sessions (active or passive), the middleware on $Host_A$ uses the `ExecuteRemote()` method and on $Host_B$ the `ExecuteLocal()` method.

Moreover, we refer to the copy of the document stored on $Host_A$ as Doc_A and that maintained on $Host_B$ as Doc_B. Let us also suppose that, after the execution of the first part of the protocol, the document Doc_{com} has been choosen as Latest Common Edition, in other words as base document of the LevelTreeDiff algorithm. The documents that are involved in the reconciliation process are represented by DOM `Document` objects; we use the Xerces and Xalan packages that we present in Appendix A.

`ExecuteRemote()` on $Host_A$ calls the `compute()` method of the `LevelTreeDiff` class with Doc_{com} and Doc_A as arguments (Doc_{com} is the base document, whereas Doc_A is the modified document). The output of the execution of this method will be the "diff" document Doc_{diff}, which will be sent to host H_B as you can see in Figure 5.4. The exchange of the DOM `Document` objects is based on object serialization provided by the Java language.

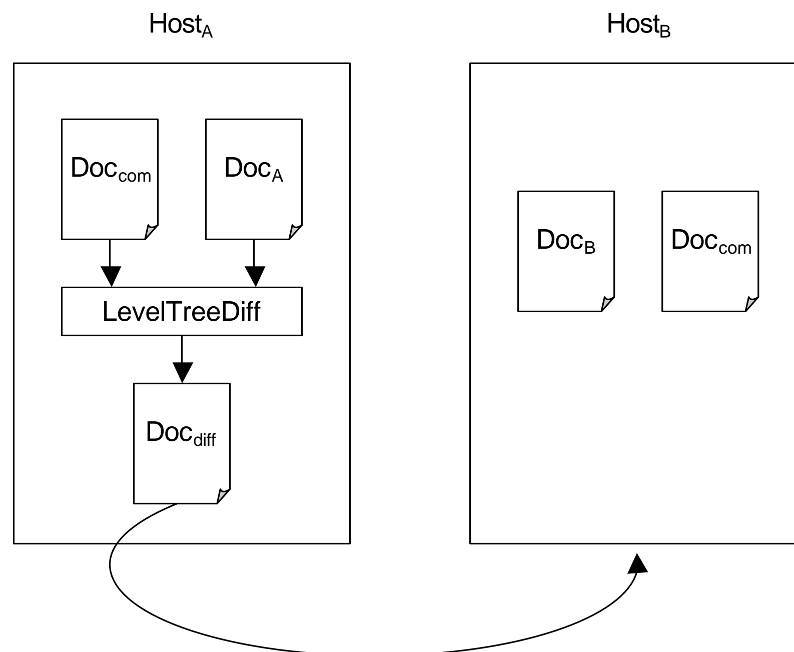

Figure 5.4 Generation of the "diff" document on host H_A from the latest common edition and the local replica of the data document using the `LevelTreeDiff` class.

After receiving Doc_{diff}, the middleware running on host H_A calls the `merge()` method of the `LevelTreeMerge` class with Doc_{com} and Doc_{diff} as arguments in order to reconstruct Doc_A locally.

Therefore, on host H_B now there are a local copy of Doc_A and, naturally, Doc_B. Thus, the reconciliation between these two documents is performed on host H_B without exchanging information with host H_A during the execution of the algorithm. This process is illustrated in Figure 5.5.

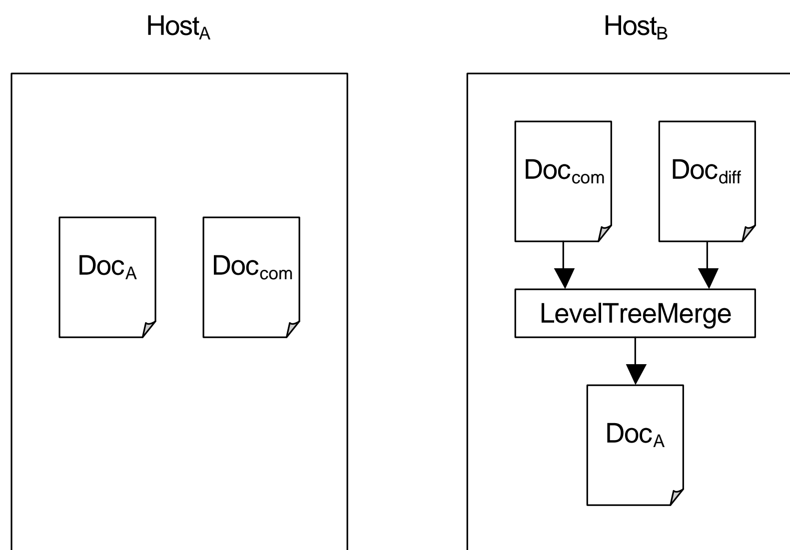

Figure 5.5 Using the common latest edition a local copy of the replica stored in the other host is generated.

It is possible to reconcile Doc_A and Doc_B using the method `Reconcile()` of the `LevelTreeReconcile()` class. The arguments of this method are the local copy of the document (Doc_B), the remote copy (Doc_A) and the latest common edition (Doc_{com}). The output will be a "reconciled document" called Doc_{rec} (as you can see in Figure 5.6).

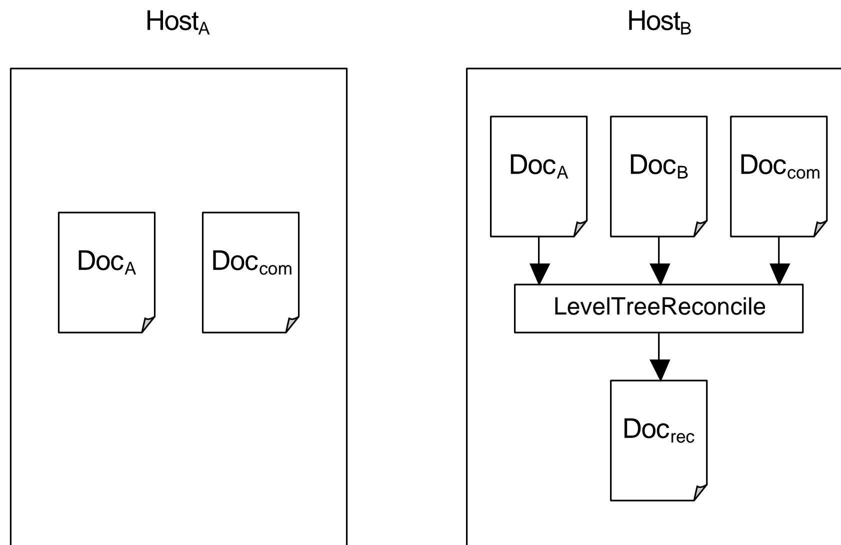

Figure 5.6 In host H_B by means of the `LevelTreeReconcile` class the reconcilied document of the two inconsinstent replicas is generated.

The final step is the generation of the reconciled document on host H_A; the middleware uses the `compare()` method of the `LevelTreeDiff` class again, with Doc_{com} and Doc_{rec} as arguments, in order to compute a new "diff" document. Afterwards, this is sent to H_A (in Figure 5.7 we call it Doc_{diffn}).

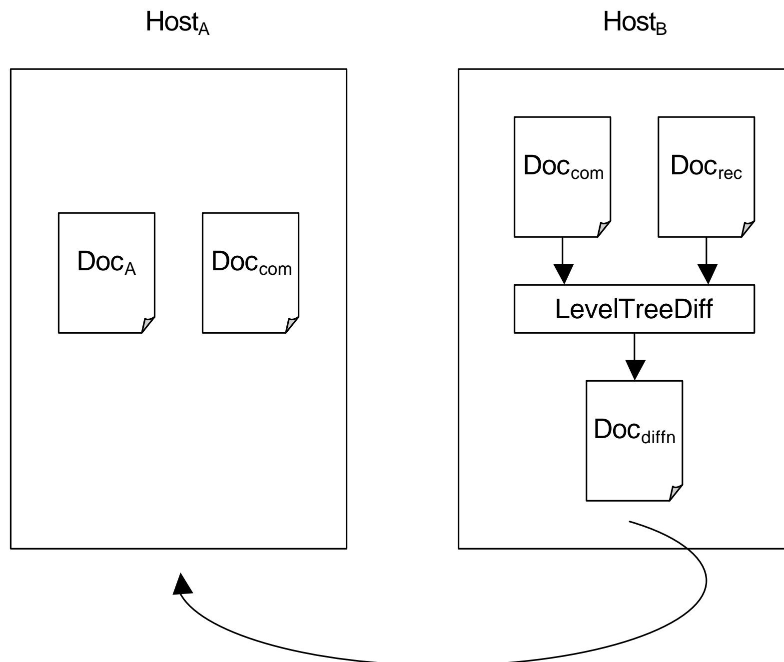

Figure 5.7 Using the LevelTreeDiff algorithm, the middleware generates the "diff" document between Doc_{com} and Doc_{rec}.

The middleware then reconstructs the reconciled copy in the usual manner, exploiting the `merge()` method of the `LevelTreeMerge` class with Doc_{com} and Doc_{diffn} as arguments (see Figure 5.8). Now, $Host_A$ and $Host_B$ store the reconciled copy, which will be the new latest common edition.

The versioning system is implemented by a certain number of classes, such as `LatestTree` (which inherits from `Tree`), according to the specification described in the previous chapters. `LatestTree` provides methods to navigate and modify the versioning tree. We also point out that the new edition number is released on $Host_B$ after the generation of the reconciled document.

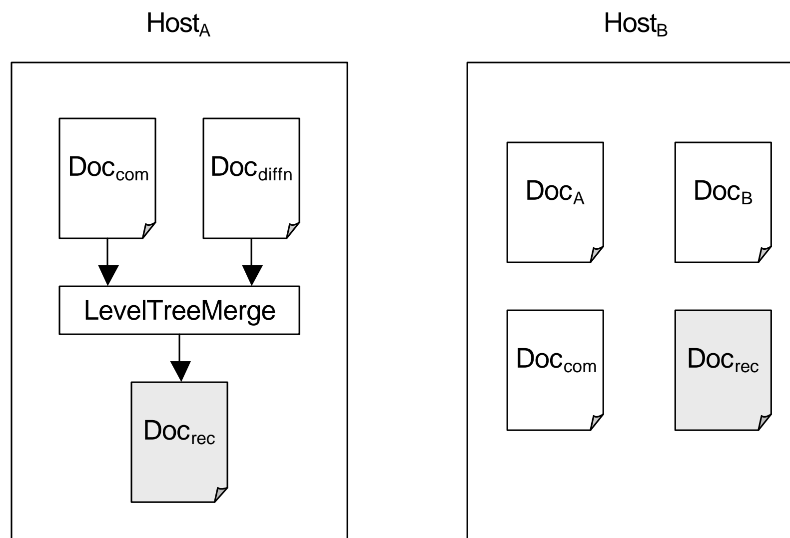

Figure 5.8 The reconciled document is generated on $Host_A$ using the `LevelTreeMerge` again. Now Doc_{rec} is present in both hosts.

It is worth noting that all these computations rely on Xerces and Xalan. This limits the performance of the system, since these packages require a large amount of memory and, for this reason, they are quite unsuitable for resource-poor devices. One of the future improvements will be the replacement of these components with more lightweight ones such as NanoXML [Des02] or kXML [GHKP02].

5.7 Networking aspects

5.7.1 Overview

As we have discussed before, the current implementation of XMIDDLE relies on UDP over TCP. In this section we discuss some solutions that has been adopted in order to implement the modules that are responsible for network communications.

XMIDDLE uses the `java.net.MulticastSocket` class to verify the presence of other hosts *in reach* (independently from the wireless communication technology used by mobile devices). This class allows programmers to send and receive IP multicast packets. A Java multicast socket is a datagram socket (in fact `java.net.MulticastSocket` inherits from `java.net.DatagramSocket`), with additional capabilities for joining IP-multicast groups. A multicast group is specified by a class D IP address and by a standard UDP port number. Class D IP addresses are in the range `224.0.0.0` to `239.255.255.255` (inclusive); moreover, the address `224.0.0.0` is reserved and should not be used. We chose the class D IP address `234.5.6.7` and the port number `6789`.

The class `edu.UCL.xmiddle.controller.UDPNetwork` (which extends `edu.UCL.xmiddle.framework.controller.Network`), is responsible for creating the multicast socket and joining the group, when the host is connected to the network, and leaving the group and closing the socket, when the host disconnects from it. It is worth noting that two hosts may be in the same radio coverage (if they rely, for example, on 802.11 technology) and may be disconnected, since connection and disconnection are not automatic operations, but are decided by applications (in fact, the middleware provides primitives in order to perform connections and disconnections).

In addition, each XMIDDLE host has a *private socket* for the requests that are sent only to it; the port number is not fixed and is specified by the application. The `UDPNetwork` thread opens the socket (using `java.net.DatagramSocket`) and listens for requests sent to the host, when this is connected to the network; the `UDPNetwork` thread is also responsible for parsing the packets received and queueing the corresponding requests to the `LocalHost` module (in the case of protocol request). After a disconnection, `UDPNetwork` closes the socket and stop the thread that is responsible for receiving message from other hosts.

Now we can analyse in detail how XMIDDLE is able to provide host-awareness, which is an essential feature for a middleware system that has been designed to support the development of applications for an ad-hoc network context.

5.7.2 Host awareness

As we have discussed in the previous section, when an XMIDDLE host is connected, it maintains two sockets to interact with other hosts, a multicast socket and a private socket. Using the first one, it periodically multicasts to any other hosts a message containing information that are necessary to enable communication between two hosts that get in reach for the first time. This message includes the primary and secondary IDs of the transmitting host, its IP address number, the number of the port bound to the private socket, the `ExportLink` and the `LinkedFrom` tables.

The `edu.UCL.xmiddle.controller.UDPNetwork.OnlineMessage` class is responsible for multicasting this information at predefined intervals, while the `edu.UCL.xmiddle.controller.UDPLocator` class is responsible for receiving information by the other hosts that are in reach, parsing it and updating the `InReach` table. `UDPLocator` also checks the presence of a new host in reach in the `exportLink` and `linkedFrom` tables. If the two hosts are sharing information, `UDPLocator` starts the automatic reconciliation process, queuing a request to `LocalHost`. If the new host in reach is present in the `inReach` table, `UDPLocator` updates its `LinkTable`. To detect possible disconnections `UDPLocator` checks the time that has occurred since a host has last multicast its message. If this value is greater than a pre-defined threshold, the host is deemed to be out-of reach and consequently the `inReach` table is updated.

In order to provide an example of this mechanism, let us consider a host with primary ID `555` and secondary ID `pda32`, with `192.168.0.4` as assigned IP address, which has a private socket bound to port `2000`. We also assume that this host exports the `/shared_data/data1` branch. In this case the message that is multicast to other hosts will be:

```
555,pda32,192.168.0.4,/shared_data/data1
```

`UDPLocator` includes methods to retrieve the hosts that are currently in reach by their IP addresses and primary IDs; exploiting this feature, it is possible to identify which host has sent a specific message. However, we underline that the implementation is not based on IP addresses, since they are not assumed as fixed. The only piece of information that is expected to be constant and unique (in order to identify univocally each host) is the primary ID. Moreover, when a host becomes in reach, `UDPLocator` queues a request to the `LocalHost` in order to start the reconciliation process.

Chapter 6

XMIDDLE applications

In this chapter we describe two applications that we have developed using the XMIDDLE platform. They belong to two different application domains: the first one, which we called *Shop*, allows mobile electronic commerce in a nomadic and ad-hoc network context; the second one (*Meeting*) provides a support for collaborative work as concerns the organization of meetings. We chose these scenarios, since they are realistic examples of possible uses of XMIDDLE.

6.1 Development of an XMIDDLE application

The development methodology of an application based on XMIDDLE is very simple; in this section we present the most interesting related issues. First of all, developers have to consider the general requirements of wireless applications design. Essentially, they have carefully to evaluate the limitations related to the scarcity of resources of portable devices in terms of memory, computational and input-output capabilities (for example, PDAs are usually not equipped with a keyboard).

However, using XMIDDLE, programmers do not have to deal with networking issues, since communication between hosts is provided by the middleware. Furthermore, clearly, the aspects related to data replication and synchronization are hidden from developers, who can modify the default system behaviour, using the techniques that we have presented in the previous chapter.

Applications relying on the XMIDDLE platform have clearly to be written in Java and they have to follow some specific and simple guidelines. First of all, they have to implement the `edu.UCL.xmiddle.framework.lib.XApp` interface (see the corresponding UML class diagrams in Figure 6.1), which allows the middleware to interact with applications. These have to register with the

middleware providing a reference, which allows the invocation of their methods by the platform for the notification of particular events, such as, for example, the termination of the execution of a protocol (i.e., reconciliation process).

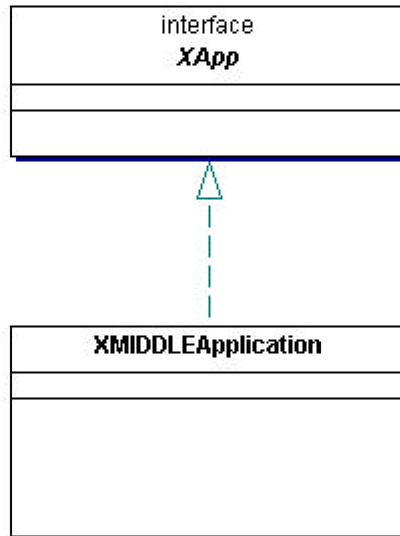

Figure 6.1 UML class diagram showing the relationship between the `XApp` interface and a class that implements an XMIDDLE applications.

It is possible to briefly outline the sequence of design steps that developers have to follow in order to build an application based on XMIDDLE. The class that provides a high-view representation of the platform and that represents the only "access point" to the middleware is `SimpleManager` (which inherits from the framework class `Manager`). This class has to be instanced in the constructor method of the application invoking the method `getInstance()`.

```
SimpleManager manager = SimpleManager.getInstance();
```

The next step is the registration of the application with the middleware, calling the `registerApplication()` method in the constructor method of the application like this:

```
Integer this.appID = manager.registerApplication(this);
```

By invoking this method, an application obtains a reference to the middleware (specifically to the `SimpleManager` class) and, at the same time, the XMIDDLE platform registers the application, generating also its application identifier and its *profile*. An application can retrieve its profile using the `getProfile()` method.

```
ApplicationProfile profile=manager.getProfile(appID);
```

Then, an application has to load the data into the middleware, reading them, for example, from a disk or a flash memory. The tree data structure must be represented using W3C DOM specifications, exploiting the packages Xerces [Apa02a] and Xalan [Apa02b], provided by the Apache Software Foundation. After creating the complete structure of the tree (including the empty sub-trees that are only successively used and modified) developers have to load it using the method `setData()` of the `applicationProfile` class. The following code exemplifies this, using `dataDocument`, a DOM `Document` object as the argument of the `setData()` method.

```
Tree tree=profile.setData(dataDocument);
```

This method returns a `Tree` object that represents the data structure stored in the XMIDDLE host. Now an application can access the tree using the methods that we have described in the previous chapter. Essentially it can access the tree with an `open()` operation, eventually modify it using the DOM API and finally commit the changes using the `close()` method.
An application has also to specify which part of the tree is exported, by using the `export()` method provided by the `Manager` class.

```
manager.export(exportedElement,appID);
```

`exportedElement` is the DOM `Element` object representing the root of the exported sub-tree; `appID` is the application identifier obtained before.
The next step is the registration of the XML Schema document used to perform the reconciliation process. Again, it may be read from disk or flash memory. We

register this document with the application profile using the `setDataSchema()` method as follows:

```
profile.setDataSchema(dataSchema);
```

`dataSchema` is a well-formed DOM `Document` object created by parsing the XML Schema document that specifies the structure of the exported sub-tree.

It is worth noting that it may be possible that an application does not export any branch, linking only data stored in other hosts (using the `link()` method of the `Manager` class). Vice versa, an application may only export a branch of its tree, without linking to any host.

Applications also have to provide functionalities in order to allow users to connect or disconnect from the network (using the `connect()` and `disconnect()` methods).

6.2 The *Shop* application

6.2.1 Application scenario

One of the most interesting application fields of wireless technologies is mobile electronic commerce (*m-commerce*). Mobile devices that are suitable for this kind of applications are essentially PDAs (with networking capabilities) and cellular phones. The term m-commerce usually identifies the possibility of shopping on a website using mobile devices, such as WAP-enabled mobile phones.

The target of our application is similar, even if we focus only on the aspects of support to collaborative shopping in ad-hoc network context. The aim of our application is to allow a group of people (i.e., a family) to do collaborative shopping. Each member can browse the catalogue of the available products and can buy them by using its mobile device. The shopping basket is shared among the members of the same group. The portable devices form an ad-hoc network, since it is expected that any type of network infrastructure is available.

XMIDDLE enables and simplifies the development of this kind of applications, since it supports the data sharing in such context. Developers have only to

concentrate on the aspects related to the applicative problem without considering networking and data consistency issues.

6.2.2 Description

As we have discussed in the previous section, *Shop* allows a group of people to do collaborative shopping. Each member of the group needs two types of information, the product catalogue and an updated list of the contents of the shopping basket that is, in this case, the shared data structure. Therefore, each member of the group has to link to the branch representing the list of purchased products stored in one of the hosts forming the ad-hoc networks. It is worth noting that the whole data tree is not exported, since each member has a copy of the catalogue, which cannot be modified by other users. It is possible to extend the current implementation, providing synchronization functionalities in order to share a catalogue stored for example in a PC connected to the Internet, which maintains an update version of the available products.

We now discuss some interesting aspects of the design of *Shop*, pointing out some general guidelines for the development of applications based on the XMIDDLE platform. We start considering the issues related to the interaction with the user and then we will analyse the design of the shared data structures.

The design of the application user interface for mobile applications is a challenging issue, since developers have to evaluate the limited interactions that are possible with this kind of devices. In fact the size constraints on a portable computer require a small user interface.

We develop this application designing a graphical user interface that can be used by means of pointing devices, such as a pen (with a touch display) or a mouse. Essentially, the target device of the *Shop* application is a PDA (we tested it using Compaq iPAQ 3660 PDAs), which is usually provided with a small screen and is not equipped with a keyboard.

Figure 6.2 The start-up screen of the *Shop* application

We use the Java AWT API in order to design the graphical user interface and the event handling system. We point out that the user interface is essentially composed of buttons (using the `java.awt.button` class), since we have to provide interaction with users without a keyboard.

For this reason, the selection of a single product requires some steps (selection of department, category, product, quantity, etc.) using a sequence of panels. In Figure 6.3 the panel related to the choice of the product category is showed.

Figure 6.3 Selection of a product category in *Shop*

Each user can display the current contents of his basket. It is worth noting that after a reconciliation process, the application automatically refreshes the visualization of the current contents, by means of the `notifyProtocolEnd()` method called by the middleware.

6.2.3 Application data structures

The data structures that we used in *Shop* allow us to point out some common aspects of the use of XML documents in the development of an XMIDDLE application.

Essentially we need to store two types of information: the catalogue of the available products and the current basket contents. We have to design the tree structure considering this requirement. We use a tree with two branches, one representing the catalogue and one the shared basket. The structure of the XML document is the following:

```
<shop>
  <catalogue>
  …
  </catalogue>
  <basket>
  …
  </basket>
</shop>
```

Since an XML document can be mapped into a tree structure, we can represent this document as follows:

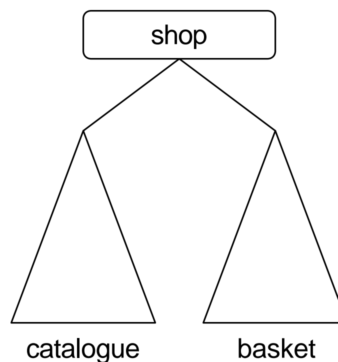

Figure 6.4 Structure of the tree used by the *Shop* application

During the initialization, the application firstly appends the catalogue sub-tree to the document root loading it from a file stored in the local disk. The catalogue XML document has the following structure:

```
<?xml version="1.0"?>
<catalog>
  <department>
```

```
    <name>Bakery and Cakes</name>
    <aisle>
      <name>Bread</name>
      <shelf>
        <name>Sliced Loaves</name>
        <item>
           <name>White</name>
           <price>0.60</price>
        </item>
        <item>
          <name>Wholemeal</name>
          <price>0.80</price>
        </item>
      </shelf>
      <shelf>
        <name>Italian</name>
        <item>
          <name>Ciabatta</name>
          <price>0.95</price>
        </item>
        <item>
          <name>Focaccia</name>
          <price>1.10</price>
        </item>
      </shelf>
    </aisle>
    <aisle>
      …
    </aisle>
  </department>

  <department>
   …
  </department>
 </catalog>
```

Afterwards, an empty basket sub-tree is created; this is the branch that is exported by each host. In other words, this is the part of the tree that is possible to link from another host. It can be seen as an autonomous document with the basket node as root. It is also the document that will be reconciled during the synchronization process. Its structure is the following:

```
<basket order="no">
 <item>
  <name>Granny Smith apple</name>
  <price>0.20</price>
  <quantity resolutor="greater">3</quantity>
 </item>
 <item>
  <name>Milk 1L</name>
  <price>0.40</price>
  <quantity resolutor="greater">1</quantity>
 </item>
</basket>
```

By analysing this document it is possible to make some interesting observations. We note that the root element has the attribute `order` set to `"no"`; this specifies that the middleware has to consider this document as an unordered tree and to apply consequently the proper reconciliation process. Since this is the exported tree, it is also necessary to provide the corresponding XML Schema document, in order to enable the synchronization process that we have described in Chapter 4.

```
<xsd:schema xmlns:xsd="http://www.w3.org/1999/XMLSchema">
 <xsd:element name="basket">
  <xsd:complexType>
   <xsd:element name="item" type="xsd:string" minOccurs="0"
maxOccurs="unbounded">
    <xsd:complexType>
     <xsd:element name="name" type="xsd:string"/>
     <xsd:element name="price" type="xsd:decimal"/>
     <xsd:element name="quantity">
      <xsd:complexType base="decimal">
       <xsd:attribute name="resolutor" use="default"
value="greater"/>
      </xsd:complexType>
     </xsd:element>
    </xsd:complexType>
   </xsd:element>
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

This document is loaded during the configuration process of the application from the disk. It is worth noting that we use `greater` as default resolutor for this application.

Moreover, we underline that this application exploits the possibility to define application-specific reconciliation policies related to possible value conflicts. Let us consider this fragment of the `basket` sub-tree:

```
…
  <item>
   <name>Milk 1L</name>
   <price>0.40</price>
   <quantity resolutor="greater">2</quantity>
 </item>
…
```

We note that the `greater` resolutor is indicated to solve this conflict; in other words, the highest value is chosen between the two different values in case of conflicts. This is a possible choice, but it is not the only one; in fact an alternative

is to use the `add` resolutor, which also considers the latest common version in order to reconcile the conflicts, as we have discussed before.

6.2.4 Application evaluation

We designed this application essentially to test our middleware; for this reason, we used data structures that allow to check all the mechanisms of the reconciliation process in details.

However, this is an example that shows how XMIDDLE can be exploited in order to build applications that need the sharing of complex data-structure, as in the case of mobile commerce.

By analysing the code of this application, it is possible to observe that developers have to concentrate only on the aspects related to the graphical user interface and the management of possible events, without considering the networking and data synchronization. The use of XMIDDLE hides all the underlying complex mechanisms such as the detection of the hosts that are currently in reach or the problems related to the locking of data structures to avoid generating additional inconsistencies.

6.3 The *Meeting* application

6.3.1 Application scenario

Another promising application field for wireless software applications is the support for collaborative work; for this reason, we developed *Meeting*, an application based on XMIDDLE that allows the members of working groups in an organization to schedule their meetings using devices that communicate by means of an ad-hoc network.

In recent years, information systems research has examined a large range of issues dealing with the provision of synchronous support for electronic meetings, negotiations and so on. This research area is known as computer supported cooperative work (CSCW) [DavSK01].

One of the classic problems is the synchronization of different users that work alone or without interacting with each other continuously, but, at the same time, they have to cooperate, since they have to operate, for example, on the same data.

6.3.2 Description

Meeting addresses the problem of the sharing of a meeting schedule between members of the same working group. Each member of a group can call a meeting and can view the list of the planned ones. XMIDDLE provides a support that enables users to schedule a meeting, even if they are disconnected and to re-synchronize with other users after a reconnection.

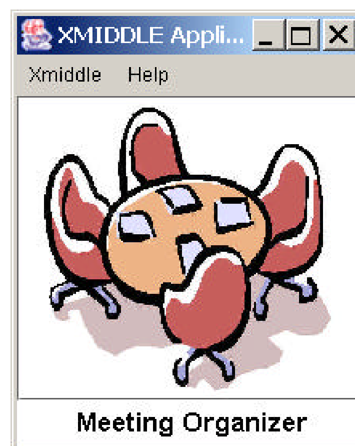

Figure 6.5 The start-up screen of the *Meeting* application

Meeting is specifically designed for running on PDAs or other similar mobile devices. Users interact with the application by means of a touch screen or a mouse; in other words, also in this case, the presence of a keyboard is not required.

The start-up screen provides a login procedure. Each user has to log in to specify his identity. If he adds a new meeting this information will be used in order to identify the organizer. Users can view and modify only the list of current scheduled meeting related to the projects in which they are involved. For example let us consider three members of an organization. $Member_A$ works on a project called X, $Member_B$ on a project called Y, whereas $Member_C$ is involved in both projects X and Y. $Member_A$ is able to visualize only the meetings scheduled by

the hosts involved in his project (in this case Member_C). On the other hand, for the same reason, Member_C can visualize the meetings called by both Member_A and Member_B. After the login procedure, a user can choose one of the three options that are available in the main menu (see Figure 6.6).

Figure 6.6 The main menu of the Meeting Application

Each user can schedule a new meeting or view the list of the current meetings related to the projects in which he is involved (Figure 6.7).

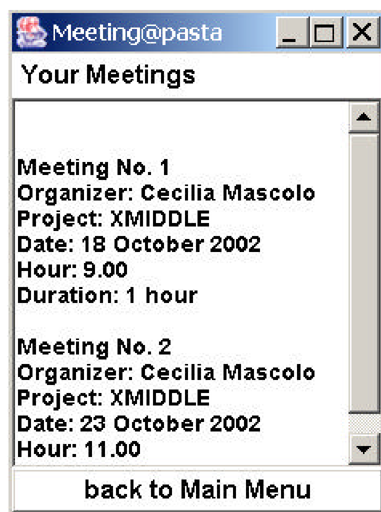

Figure 6.7 The panel displaying the current scheduled meetings

Naturally, he can also link to another host in order to share its list of meetings. It is worth noting that it is possible to extend this application in order to share not only simple data structures, such as a list of meetings, but also entire documents, which can be stored using XML and visualize by means of the XSLT technology [Cla99].

6.3.3 Application data structures

We designed a more complex data structure tree in this case. Our aim was to give the possibility to change the composition of each group in an easy way, simply modifying an XML document.

We applied a technique derived from database design, considering two XML documents, which can be considered as two tables (according to an information system terminology) that are linked by means of a key (in our case an univocal alphanumerical string, created using the first three letters of the name and the first three letters of the surname of the group members), one representing the data related to each person and one describing the composition of each working group.

The XML document describing the data of each person has the following structure:

```
<?xml version="1.0"?>
 <people>
  <person>
   <firstname>Wolfgang</firstname>
   <surname>Emmerich</surname>
   <role>Senior Lecturer</role>
   <pcode>wol_emm</pcode>
  </person>
   <person>
   <firstname>Nima</firstname>
   <surname>Kaveh</surname>
   <role>PhD Student</role>
   <pcode>nim_kav</pcode>
  </person>
  <person>
   <firstname>Cecilia</firstname>
   <surname>Mascolo</surname>
   <role>Lecturer</role>
   <pcode>cec_mas</pcode>
  </person>
  <person>
   <firstname>Mirco</firstname>
   <surname>Musolesi</surname>
   <role>Research Assistant</role>
```

```
    <pcode>mir_mus</pcode>
   </person>
   <person>
    <firstname>James</firstname>
    <surname>Skene</surname>
    <role>PhD student</role>
    <pcode>jam_ske</pcode>
   </person>
   <person>
    <firstname>Stefanos</firstname>
    <surname>Zachariadis</surname>
    <role>PhD student</role>
    <pcode>ste_zac</pcode>
   </person>
  </people>
```

It is worth noting that we use a simplified set of information describing each member of the organization, which is easy to extend. The XML document that represents the composition of the groups of the organization has the following structure:

```
<?xml version="1.0"?>
 <groups>
   <group project="XMIDDLE">
    <pcode>wol_emm</pcode>
    <pcode>cec_mas</pcode>
    <pcode>mir_mus</pcode>
    <pcode>ste_zac</pcode>
   </group>
   <group project="Tapas">
    <pcode>wol_emm</pcode>
    <pcode>cec_mas</pcode>
    <pcode>jam_ske</pcode>
   </group>
</groups>
```

It is worth nothing that a person can be involved in more than one project; we also note that it is extremely easy to change the composition of working groups, since it is only necessary to modify this XML document

The other type of information that we have to store in our tree is the meeting schedule. This is represented by means of another XML document, which presents the following structure:

```
<meetings order="no">
 <meeting>
  <organizer>Cecilia Mascolo</organizer>
  <project>XMIDDLE</project>
  <year>2002</year>
  <month>October</month>
```

```
  <day>20</day>
  <hour>10.00</hour>
  <duration>2</duration>
 </meeting>
 <meeting>
  <organizer>Wolfgang Emmerich</organizer>
  <project>TAPAS</project>
  <year>2002</year>
  <month>December</month>
  <day>5</day>
  <hour>9.00</hour>
  <duration>3</duration>
 </meeting>
</meetings>
```

We observe that this document has to be processed as an *unordered tree* and for this reason the corresponding `order` attribute is set to `"no"`. Let us consider for example the case of two users that add two different meetings when they are disconnected (let us also assume that there are no meetings scheduled previously). Each user generates a document that contains only one branch describing the characteristics of the meeting.

In this case, in order to reconcile these documents, we have to treat them as unordered trees; the "branch" sub-unit in this case is the structure defined by the `meeting` tag. The middleware detects the value conflicts and resolve them, using the techniques described in Chapter 4. The reconciled document will be composed of two sub-trees describing these two meetings.

In order to manage these data using XMIDDLE, we use a tree structure, which is composed of the three sub-trees corresponding to the XML documents that we have presented before.

```
<meetingAppData>
  <meetings>
   …
  </meetings>
  <people>
   …
  </people>
  <groups>
   …
  </groups>
</meetingAppData>
```

The *people* and the *groups* branches are created by reading the corresponding XML documents from a persistent memory. The meetings sub-tree is originally created as an empty document. In other words, the first common edition of the

exported document has one only entity, the root element. In Figure 6.8 we show the tree that can be semantically associated to this XML document.

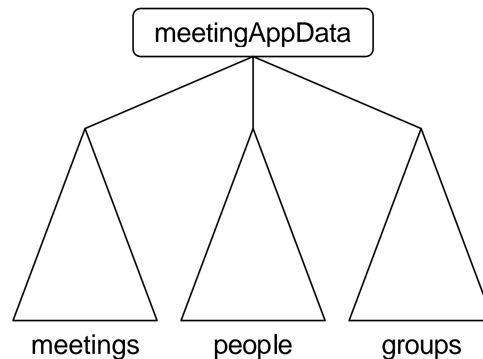

Figure 6.8 Structure of the tree used by the *Meeting* application

The exported sub-tree is that which describes the scheduled meetings. In other words the `people` and the `meeting` sub-trees can be seen as private branches of each host. At the same time, on the other hand, the `meetings` sub-tree is managed by the middleware as a "stand-alone document", with the `meetings` tag as root element.

In order to enable the reconciliation process, we have to provide the related XML Schema document that we present below:

```
<xsd:schema xmlns:xsd="http://www.w3.org/1999/XMLSchema">
 <xsd:element name="meetings">
  <xsd:complexType>
   <xsd:element name="meeting" type="xsd:string" minOccurs="0"
maxOccurs="unbounded">
    <xsd:complexType>
     <xsd:element>
      <xsd:element name="organizer" type="xsd:string"/>
      <xsd:element name="project" type="xsd:string"/>
      <xsd:element name="year" type="xsd:decimal"/>
      <xsd:element name="month" type="xsd:decimal"/>
      <xsd:element name="day" type="xsd:decimal"/>
      <xsd:element name=”hour” type=”xsd:decimal”/>
     </xsd:element>
    </xsd:complexType>
   </xsd:element>
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

It is worth noting that since the branch that can be duplicated is the sub-structure defined by the `meeting` tag, the value of its attribute `maxOccurs` is set to `unbounded`.

6.3.4 Evaluation

Meeting represents another interesting example of a possible application based on the XMIDDLE platform. We underline that we designed only a prototype with a basic set of functionalities, focusing only on the aspects related to the distributed reconciliation process. However, this represents the key point of the development of every application based on our middleware. In other words, we implement the core of the application, which may be used as a basis for future improvements and tests.

It is possible to extend it, providing, for example, functionalities in order to check the current schedule of the meetings, by exploiting, for example, XML Schema expressiveness. As we have discussed before, it is also possible to add a functionality that allows users to access and load data from a remote database server, which is able to export them in an XML standard format.

Chapter 7

Testing and evaluation

In this chapter we describe the testing process of XMIDDLE and we present an evaluation of our work, identifying some possible enhancements of the current implementation of the system and outlining the future research directions.

7.1 The testing process

We tested XMIDDLE using a wired and a wireless LAN, in order to verify the performance of the platform. Our aim was to test some essential characteristics of the system, focusing on its correctness, reliability and scalability.

7.1.1 Test devices

In order to test our platform in an ad-hoc network setting, we use a certain number of Compaq iPAQ 3660 [HP02] based on the StrongARM SA-110 [Int02] processor running at 206 Mhz, which is optimised for meeting portable and embedded application requirements. It contains two flash memories and two SDRAM chips onboard (32-bit wide data bus). In particular, we use devices with 64 MB of RAM and 16 MB of ROM. It also provides a high quality colour screen (with a resolution of 320x240 pixels) and an expandability support that we exploit to add an 802.11 wireless card in order to use it to form an ad-hoc network. More specifically, we use an Avaya wireless PC card [Awa02] that provides data rates of up to 11 Mb/s; its radio range covers considerable distances depending on the number and type of obstacles and the data rate. For example, it is possible to have a transmission range of up to 550 m through an open office at 1 Mb/s data rate.

With respect to the operating system running on the iPAQ PDAs, we use Linux: in particular, we installed a distribution of Linux developed by the Handhelds

Group, sponsored by Hewlett-Packard, called Familiar Project Linux [Han02] (version 0.5.3).

We also tested our platform using computers connected by the Fast-Ethernet LAN of the Department of Computer Science of University College London. The principal motivation of this test is to verify the scalability of the system considering a large number of machines.

7.1.2 Test results and evaluation

Since XMIDDLE is a very complex software system, we perform testing at different levels, evaluating each unit and, afterwards, the entire middleware using the applications that we have developed.

Firstly we analysed the aspects related to the execution of threads: they are initialised correctly and terminated gracefully. With respect to networking issues, we tested the platform using several hosts focusing on the correct behaviour of host-awareness mechanisms. In order to design and test the reconciliation algorithm, we developed some testing tools that allow to execute locally, without transferring data from other hosts. Afterwards, we tested the correctness of the linking and reconciliation protocols execution, also considering the versioning system. The final result of the testing phase on the system correctness was positive; the platform meets the requirements that we have presented in the previous chapters.

We also tested the performance of the systems, focusing on the execution of the reconciliation protocol. With respect to scalability, we performed the reconciliation process with the same data structure with an increasing number of hosts. The test results are shown in Figure 7.1 In the diagram we indicate the average values that we obtained from a test using the *Shop* application with a tree structure with the same size in each case. More specifically, the reconciliation algorithm was performed with two different documents with ten branches each. The execution time of the protocol is calculated from the time of sending request to start the reconciliation process to the other hosts to the time of receiving the acknowledgement message.

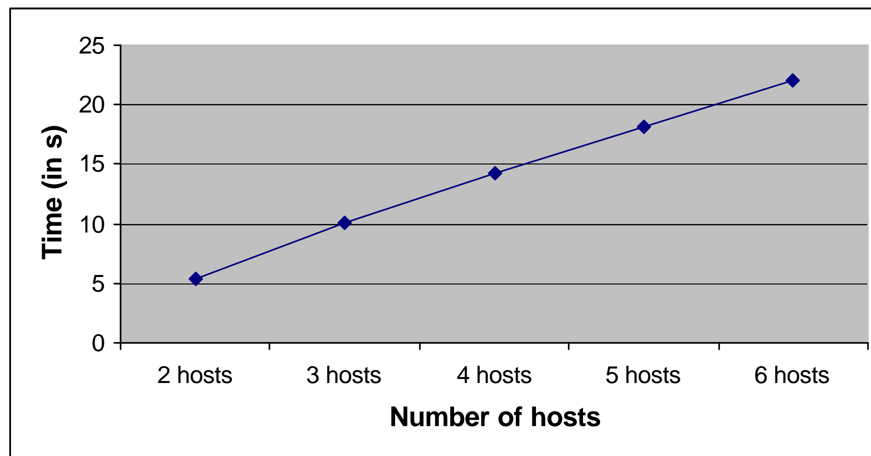

Figure 7.1 Execution time of the reconciliation protocol with a different number of hosts (in the diagram average values are indicated).

It is possible to note that the performance of the system decrease as the number of involved hosts increase. It is worth observing that, even if the reconciliation protocol is performed concurrently with any host that is in reach and connected, the access to the tree structure of the host that has made changes and therefore has started the reconciliation process is mutually exclusive. In other words, the elaboration on this tree is sequential. However, it is important to underline that the number of hosts that are sharing the same branch in the same radio coverage (and consequently that can be involved in possible reconciliation processes) is usually very limited.

We also studied the performance of the system with data structures of different size; it is interesting to note that the execution time of the reconciliation algorithm is strictly correlated to the dimension of the three XML documents that are used to perform the synchronization process.

We studied the execution time of the reconciliation algorithm (that is a significant phase of the reconciliation process) with different data structures in order to formalise and verify a simple model of the complexity of the problem that can be described using the number of branches in each tree. In particular, we used the *Shop* application: as we have discussed in Chapter 6, the shared document represents the shopping basket and it is composed of a certain number of branches corresponding to its current contents.

We analysed two particular cases that are the extremities of the entire range of possible sizes of the documents involved in the reconciliation process (without considering the trivial case of two empty documents).

The simplest case is that of the reconciliation of an empty document and a document with *n* branches. In the diagram in Figure 7.2 you can see the results obtained with a different number *n* of branches. The relation is linear; this experimental results derives from the complexity of the algorithm that is *O(n)*. This is a possible and common real case in applications; for example this type of reconciliation process takes place when a new host joins a group of hosts, which forms an ad-hoc network for the first time, and it has to synchronize its shared data structure (that may be empty, for instance, if the user has just started his application).

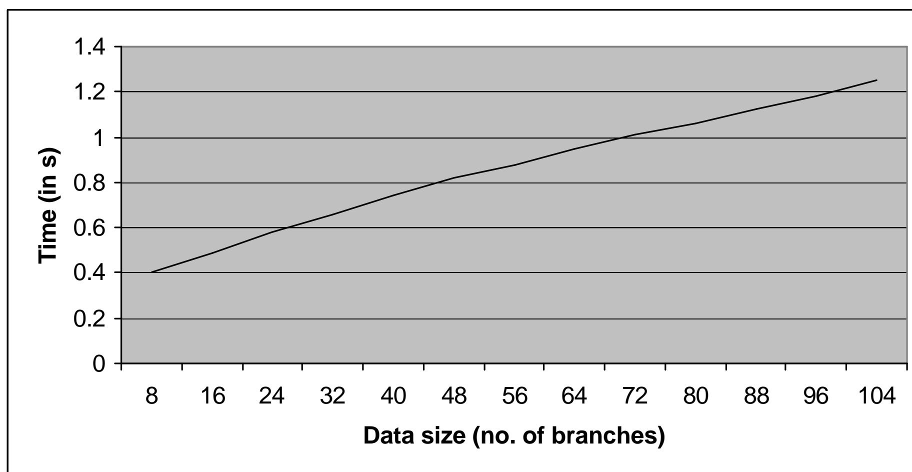

Figure 7.2 Execution time of the reconciliation algorithm in relation to data size (case 1)

We also examined the case of the reconciliation of two documents with *n* branches each (the two documents are also completely different); this can be considered the worst case of the execution of this algorithm, since its complexity is $O(n^2)$ (see Chapter 4). The results that we obtained are shown in Figure 7.3. However, it is worth noting that this situation is not common in applications; moreover, we point out that the reconciliation process usually involves documents with a small number of branches. In fact, XMIDDLE is designed to share small amounts of data; in other words, the shared XML documents have to be composed

of a limited number of branches. In fact, also for this reason, the platform provides the possibility of linking only parts of the entire tree. Therefore, developers has to design carefully the data structures used and shared by applications in order to reduce their size and so to limit the time of the execution of the reconciliation protocol.

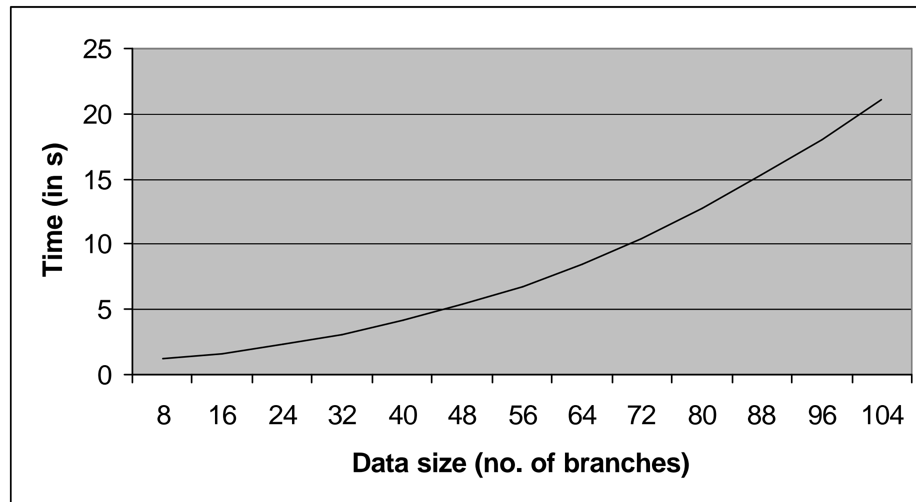

Figure 7.3 Execution time of the reconciliation algorithm in relation to data size (case 2).

It is also worth noting that the average execution time of the reconciliation algorithm in the case of `n` branches with `n<50` is less than 1 second; for this reason it does not influence remarkably the total execution time of the synchronization protocol.

From the analysis of the test results, we also discovered that one of the main bottlenecks of the system is the elaboration of XML documents based on Xerces and Xalan, which slow down the execution of the reconciliation algorithm considerably.

We also point out that the performances of XMIDDLE using PDAs equipped with 802.11 transmission technology are similar to that obtained testing the system with a Fast-Ethernet network, since the amount of data exchanged by the hosts is very limited.

7.2 A comparison between XMIDDLE and other existing mobile middleware systems

In Chapter 2 we have presented the state-of-art of mobile middleware systems focusing on the characteristics related to the design of XMIDDLE. Now we will evaluate our platform comparing it between with the other similar systems.

First of all, we consider the tuple-based middleware systems, such as Lime [PicMR99], JavaSpaces [Sun02c] or TSpaces [WicMSF98]. As we have discussed before, these provide a form of communication decoupled in time, since senders and receivers do not need to be present at the same time in order to exchange information, and in space, since a globally shared space is provided to all processes regardless of machine boundaries. Exploiting these features it is possible to address the typical issues related to the design of a system targeting a mobile context, where disconnections are very frequent. However, this model is lacking in expressiveness, since it is difficult to build complex data structures by using tuples.

The use of XML allows programmers to design more elaborated data structures, since this language introduces the possibility of defining hierarchical relationships or, in other words, to exploit a tree-like organization of information. In other systems XML has been exploited to satisfy more complex requirements with respect to coordination and data representation such as in XMARS [CabLZ00], which is a mobile agents system that uses XML to model information that can be accessed by means of classic tuple space operations.

However, some existing tuple based systems have been extended in order to manage more complex data structures; for example, recently, new functionalities have been added to IBM TSpaces in order to provide support for storage and indexing of XML documents.

From a conceptual point of view, we can identify another drawback of tuple space based systems related to the reconciliation process. By definition, tuple spaces are multi-sets, in other words tuples can be duplicated in the global space. Thus, in a mobile system, it is difficult and, however, it is unnatural to apply the concept of synchronization of modified replicas after a re-connection, since there is not the concept of univocal copy of a piece of information in the shared data space.

It is also interesting to compare XMIDDLE with other middleware systems specifically designed for data sharing. The first works in this area were related to distributed file systems: the most remarkable systems to be considered are essentially Coda [KisS92], Odissey [Sat96a] (its successor) and Bayou [TerTPDS95].

Coda addresses the problem of data replication (and synchronization) that is necessary in a large distributed file system, since access failures may happen, due to, for example, temporary server unavailability or to a disconnection of a mobile client. In Coda there is a sharp distinction between clients and servers, which are generally more powerful machines; for this reason it is not suitable for pure ad-hoc settings. Furthermore, it is worth noting that Coda provides conflict detection and reconciliation functionalities. It offers a framework for installing and invoking customized pieces of code called Application Specific Resolvers (ASRs), enabling an application dependent reconciliation process; each ASR encapsulates knowledge related to its corresponding application. Users are able to select which ASR gets invoked for a particular application by the specification of policies. If the resolution succeeds, the system does not need the intervention of the user; if it fails, the user has to solve the conflict manually. However, Coda only entire files as uninterpreted stream of bytes. Therefore it is not possible to reconcile parts of files according to a specific user policy.

XMIDDLE is not suitable for managing complex file system, but it can provide very efficient functionalities in order to deal with different type of conflicts in an application dependent way. XMIDDLE reconciliation process does not need the intervention of the user in any case, since the system can deterministically solve any type of conflicts in XML documents either in a default manner or according to an application policy, which it is easy to specify by means of two simple XML documents (the XML document that represents the data and the XML Schema document that describes them).

The Bayou architecture differs from systems (such as Coda), that maintain a strong distinction between servers, which hold databases or file systems and are usually more capable machines, and clients, which use only personal caches and resides on devices with scarce resources. For this reason, it is better suited for a mobile environment. The Bayou data synchronization model is very expressive,

there is a greater complexity for application programmers, because they must supply dependency checks and merge procedures. In general, it is very complex to define them, given the large number of conflicts that need to be detected and resolved. XMIDDLE does not provide dependency checks functionalities but, at the same time, it is possible to add easily at application level this kind of features, for example, exploiting XML technologies, such as XML Schema.
Moreover, Bayou is able to support database services (for example transaction operations, queries, etc.); XMIDDLE does not offer the possibility of accessing information from a database server, but it is possible to develop applications that are able to import data remotely stored, for example, in relational database and exported using XML and to load them for sharing. In Bayou we can also find the concept of *eventually* consistent copies; in XMIDDLE we do not have this intermediate state, which might generate additional problems: in fact, in our system, two copies can be inconsistent (different version from the same edition) or consistent (after the termination of a reconciliation process). Therefore, in Bayou, applications have to deal with data that may be still tentative and with the fact that an operation performed by a user may turn out to be altered. For example, using a tentative update not yet committed, a user may schedule a meeting in an available slot in a timetable and find later that his booking has been moved.
Odissey has been designed to operate in a mobile environment, but it presents some remarkable limitations. Firstly, the size of data that are transferred across mobile hosts may be too large for a mobile environment (for example, MPEG files), in presence of devices with scarce resources and expensive (and generally low-quality) connections. Secondly, the architecture is composed of thin clients and servers with different capabilities, so it is not suitable for an ad-hoc scenario. XMIDDLE is not designed to provide a file system services and it is not able to manage large files, such as multimedia documents; however, it is worth noting that in a mobile scenario the transfer of large amounts of data should be avoided. Our platform tries to minimise data transfer using the techniques described in Chapter 4: hosts do not send the entire documents, but only the modifications performed on their local copies of the exported sub-tree. Moreover, the current implementation of XMIDDLE does not provide adaptation; this is one of the

possible future improvements of our platform. For example, it is possible to design a mechanism that enables the execution of the reconciliation protocol depending on the context (for instance, only in the case of sufficient bandwidth availability, etc.).

With respect to synchronization issues, XMIDDLE is able to solve conflicts without the intervention of users, which instead is needed in other systems, such as CVS [Ced02]. Our platform does not use the standard SynchML protocol [Syn02] in order to perform the reconciliation of divergent replicas, since it is not suitable for an ad-hoc environment for its characteristics. First of all, in SynchML there is a strong distinction between clients and servers. In brief, this protocol is based on *change logs* that are databases in which client and server maintain information about changes or modification to the data. A version conflict happens when a single data item is modified on both the client and the server database. In XMIDDLE we do not use a log to register the operations performed on data. In order to reconcile divergent replicas, we consider their latest common edition, which, in a sense, represents the previous history of the document. As we have discussed in Chapter 4, in order to synchronize the inconsistent copies, two types of information are necessary, an initial agreed version of the document (which might be an empty document) and the current modified documents. In fact, it is possible to observe that this technique also optimises the reconciliation process, since, for example, some sequence of operations does not modify the structure of the document (i.e., an addition and a deletion of the same element) and for this reason it is useless to record them.

The IceCube [KerRSD01] project adopts a technique similar to ours, providing programmers with a parametrisable framework for reconciliation; it addresses a more general application domain (modifications on distributed objects) and it is based on logs which contain a history of performed operations. IceCube provides a very expressive language to deal with inconsistencies, but, at the same, this is not easy to use, since programmers have to identify all possible types of conflicts and describe them by means of pre-conditions and post-conditions.

Lotus Notes [Lot00] offers a large numbers of functionalities for data sharing, but at the same time it does not provide an efficient system in order to deal with possible inconsistencies. It detects inconsistencies between documents using

timestamps and it has not support for application-specific conflict resolution as XMIDDLE.

Our platform does not support location-awareness, since it is not important for a middleware designed to provide data sharing; however, it is possible to add a mechanism that enables the execution of the synchronization only in particular places, for example, for security reasons or according to particular application requirements.

Conclusions and future research issues

We realised a working prototype of XMIDDLE, which includes the features that I described in this thesis. In particular I designed and implemented the distributed reconciliation algorithm and I optimised some related aspects, such as the versioning system and the data locking mechanisms. I also contributed to develop the applications that we used to test and evaluate our platform.

XMIDDLE represents an interesting example of middleware specifically designed for an ad-hoc network environment. It provides an efficient support to deal with frequent disconnections and to work off-line. Data synchronization is performed in a transparent way for users and, at the same time, developers can build easily complex applications that need the sharing of structured data according their specific requirements. In fact XMIDDLE also represents an example of *application-aware* middleware, since it allows developers to build applications that can modify the default behaviour of the underlying platform, specifying policies, in this case by means of simple XML documents.

We can also identify some possible improvements that we now describe briefly, underlining some promising research directions.

With respect to implementation issues, the major bottleneck of the current implementation of XMIDDLE is the use of the Xerces parser [Apa02a] that is not suitable for mobile devices because it is too heavy from a computational point of view. Xerces 2 will be used in a future release of the platform, since it presents some remarkable performance improvements in comparison to the first version of this technology. Moreover, Xerces 2 also provides an XML Schema processor fully conformed to the W3C specifications. The alternative is to exploit XML parsers specifically designed for mobile and embedded applications, such as NanoXML [Des02] or kXML [GerHKP02] (see Appendix A).

Another possible improvements concern security. XMIDDLE currently does not provide secure communications between the hosts involved in the execution of a protocol. However, it is worth noting that a basic encryption communication

system in an-hoc environment is often provided by the underlying wireless transmission technologies [Int01]. Thanks to the modular architecture of XMIDDLE it is possible to introduce encrypted communication simply modifying the `Listener` and `Sender` classes, by using, for example, the Java Cryptography Extension [Sun02f] that has been integrated in the version 1.4 of the Standard Edition of the Java environment.

Another related possible improvement is the definition of *permissions* as in Unix in order to restrict the sharing of a particular branch. In fact, the current implementation of XMIDDLE allows only read-write linking; in other words, any host that links to an exported sub-tree can modify it. A possible modification is to differentiate the data sharing service, introducing a read-only access to exported data; in other words, a host could reject a request of data synchronization from a host that has not the right permissions. It is also possible to introduce the concept of group of users; in other words, according to this model, a host can only link to (and share) the data that are exported from a host that belongs to its same group. Also in this case it may be sufficient to use an attribute in the root element of the tree to specify the group name.

XMIDDLE currently does not provide a support for adaptation; we have designed but not yet implemented a mechanism based on *host profiles*; by using these, the middleware can adapt its behaviour according to the capabilities of the mobile device on which the platform is running. For example, in case of devices with limited battery resources, middleware can limit the frequency of execution of the reconciliation protocol. Host profiles can also be used to define input/output capabilities (i.e., screen size, presence of a keyboard, etc.) and to define a hierarchy among hosts, using priority numbers, which can be exploited during the reconciliation process. It is possible to integrate the mechanism of host profiles with the use of the Mobile Information Device Profile (MIDP) of the Java 2 Platform Micro Edition [Mah02] (see Appendix A).

When we have described the reconciliation process, we have pointed out that currently the reconciliation process is point-to-point; in other words we do not implement any particular group semantics. One of the possible solutions is to introduce protocols with multiple hosts involved at the same time. This implies a complete re-definition of the reconciliation protocol. This is a natural extension of

the model that we have adopted, but, at the same time, we have to consider carefully the particular characteristics of the execution context of XMIDDLE. Since the composition of groups in an ad-hoc environment is highly dynamic, a possible drawback could be the more probable failure of the execution of the reconciliation protocol due to a disconnection of a host and consequently an excessive number of rollback procedures might be performed. From an implementation point of view, it is quite easy to extend the current prototype of XMIDDLE in order to allow the execution of this “group reconciliation” protocol. In fact it is sufficient to use multicast sockets (using the `java.net.Multicast` class) in the implementation of the `Sender` and `Receiver`. It is worth noting that it is also possible by this extension to use different multicast groups in order to identify corresponding different groups of users.

Another possible improvement is the introduction of support for exchanging data according to the SynchML protocol [Syn02]. This could allow applications to retrieve data from any SynchML-compliant device and share them among a group of XMIDDLE hosts.

Appendix A
Java and XML technologies for mobility

In this appendix we briefly describe some aspects of Java and XML technologies in relation to the wireless scenario, focusing on the features that are at the basis of the implementation of XMIDDLE.

A.1 Java technologies

A.1.1 Overview

In recent years, we have witnessed the widespread diffusion of the Java programming environment [Sun02e], both in research and industrial software development communities. The Java language is object-oriented, platform independent, multithreaded and also provides a large number of additional functionalities, such as a complete support for networking. The description of the features of this language is beyond the scope of this thesis; you can find a comprehensive introduction in [HorC01]. The Java programming environment is available in three different editions, a standard one with the basic functionalities (Standard Edition), one designed to support the development of multi-tier enterprise applications (Enterprise Edition) and one, extremely optimised, addressing specifically the systems characterised by a scarcity of resources, such as mobile devices and appliances (Micro Edition).

We now present an essential description of the features of the editions of this programming environment that we used to implement XMIDDLE.

A.1.2 Java 2 Standard Edition and Micro Edition

As we have discussed before, the core of the Java environment is provided by the Standard Edition, that is available for a certain number of platforms, such as for Windows, Linux and Solaris. It is suitable for a large class of systems with

different computational capabilities; however, it cannot be used with devices with scarce resources, such as with the so-called first generation of PDAs, for their limitations in memory and in processor performances.

Java 2 Micro Edition (J2ME) is aimed at the mobile and embedded devices market, which includes, for example, cellular phones, PDAs, set-top boxes and electrical appliances. It provides a complete set of functionalities that enables developers to create applications for small devices.

The J2ME platform defines the following components:

- a series of Java Virtual Machines, each for use on various types of devices with different requirements;
- a set of libraries and APIs that can be run under each of the virtual machines (these are known as configurations and profiles);
- a certain number of programming tools (in particular in order to deal with configuration issues).

Mobile devices are characterised by different forms and functionalities, but, at the same time, they can be grouped according to their memory and computational capabilities; therefore, it is possible to identify configurations targeting specific classes of devices. A typical configuration is comprised of a Java Virtual Machine, core libraries, classes and APIs. Currently, there are two standard configurations provided by J2ME, the Connected Limited Device Configuration (CDLC) and the Connected Device Configuration (CDC). The first one is designed for devices with constrained computational and memory resources; more specifically, the memory requirement is 160 to 512 kilobytes of total memory available for the Java platform (including RAM, flash memory or ROM). The second one addresses more powerful devices that are intermittently connected to a network and that are characterised by having 2 Mbytes or more memory available for the Java platform and the applications; moreover, they also must have a 32-bit processor.

CDC and CDLC are based on specific Java Virtual Machines; the virtual machine for the CDLC is called Kilo Virtual Machine (KVM) and the one for the CDC is called Compact Virtual Machine (CVM). The KVM is a complete Java runtime environment specifically designed for portable devices; the CVM is designed for less resource-constrained devices and, for this reason, it is able to support all Java

2 Version 1.3 features and to provide libraries for advanced features such as security, JNI and RMI.
In order to develop more complex applications, it is possible to extend the J2ME configurations using the so-called *profiles*. These are sets of APIs based on particular configurations; for example, the Mobile Information Device Profile (MIDP) is designed to be used with the CDLC, providing advanced functionalities for the development of mobile applications (for instance, including classes for user interface design and persistence data storage). Another example is the Foundation profile that extends the APIs provided by the CDC, but does not provide any classes for the user interface design; as its name implies, this profile is meant to serve as a foundation for the others, such as the Personal Profile. A detailed description of this platform can be found in [Mah02].
Recently, the PersonalJava edition [Sun01], a previous specification of the Java environment targeting the development of applications designed for devices with limited resources, has been redefined as the J2ME Personal Profile. This profile provides compatibility with applications developed for Personal Java 1.2 and 1.3.
The Figure A.1 sums up these concepts, showing the architecture of the Java 2 Micro Edition Environment.

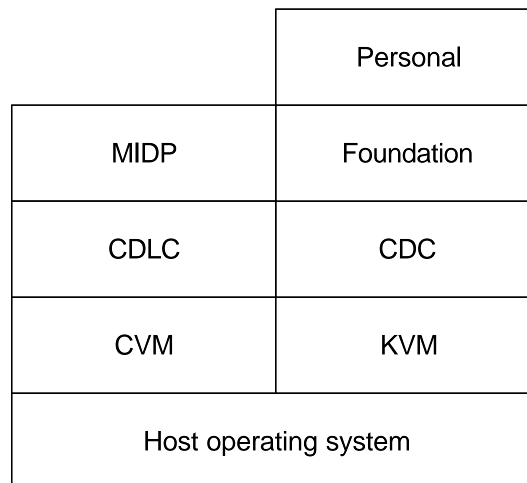

Figure A.1 J2ME environment

A.2 XML overview

In this section we briefly introduce XML [BraPS00], the language that is used in XMIDDLE to model the data structures shared between the hosts.

Java makes applications portable among different platforms; in a similar manner, XML makes data portable among different applications and context. XML is a mark-up language defined by the World Wide Web Consortium; its aim is to define a platform independent and standardized representation of structured information. XML is based on the Standard Generalized Markup Language (SGML) [ISO86], which is very powerful and expressive, but at the same time, complex and difficult to learn. Moreover, this language contains a number of features that make SGML processing software hard to implement.

XML may be thought also as an evolution of the HyperText Markup Language (HTML) [W3C02a], but its purpose is completely different. In fact, although HTML has evolved into a very rich representation language, it still defines only one particular document type. Furthermore, it is not extensible, since it is composed of a finite set of functionalities (i.e., types of tags). A large number of today's applications need a more flexible way of structuring documents. Furthermore, for example, by HTML it is not possible to define application-specific data representation and description.

XML was introduced in order to overcome these problems; it can be seen as a simplified subset of SGML, since it is easy to use and, at the same time, very powerful and flexible. Moreover, XML does not define the way in which documents are shown, but it focuses on their structure and contents.

We will not describe the characteristics of this language in details, since it is beyond the scope of this thesis. However, we present and briefly discuss an example of XML document in order to underline some fundamental features and to introduce an essential terminology. Let us consider the following document:

```
<?xml version="1.0"?>
<basket>
	<item category="stationery">
		<name>pen</name>
		<quantity>1</quantity>
		<price>1.15</price>
	</item>
	<item category="hardware">
		<name>lightbulb</name>
```

```
            <quantity>1</quantity>
            <price>0.9</price>
      </item>
</basket>
```

As you can see from this example, an XML document has the classic mark-up structure with *elements* defined by the opening tag (such as `<name>`), the element content (text or other elements, if any; in the case of `name`, the text string `pen`) and the ending tag matching with the opening one (`</name>`). The first line is the *prologue*; the tags in this section start with `<?` and terminate with `?>`. The prologue does not contain any information about the content of the document, but it provides information about it (*metadata*). In this example, the first line specifies that this document uses the version 1.0 of the XML language. Furthermore, the *root element* of the document is identified by the `basket` tag, which contains all other tags, such as `item` or `quantity` and text strings (*text element*), such as `pen` or `1`, according to a structure that can be semantically associated with a tree structure. Every document has always one only root element. Another interesting feature is the possibility of using *attributes* (in this example `category`).

A.3 XML-related technologies

In this section we will describe some XML-related technologies focusing only on the features that have been exploited during the implementation of XMIDDLE. A comprehensive discussion of these technologies is beyond the scope of this thesis and can be found in [Hol01]; we focus only on the aspects related to the design of our middleware, showing the features that we have exploited in the implementation of XMIDDLE.

A.3.1 DOM

DOM (Document Object Model) [W3C02b] is a standard interface for accessing XML documents, which are represented as trees of object nodes. Treating a document as a tree of nodes is a natural way of handling XML documents, since it reflects easily the *container/contained* relationship between XML elements.

Referring to the example of the XML document in paragraph A.2, according to a DOM terminology, the `<BASKET>` node is the *root* of the document and the

<ITEM> sub-nodes are its *child nodes*; furthermore, the two <ITEM> nodes are *sibling nodes* of each other. Therefore, it is worth noticing that, for example, the <QUANTITY> node has also a sub-node, a so-called *text node*. It is possible to give a tree representation of this document, as you can see in Figure A.2.

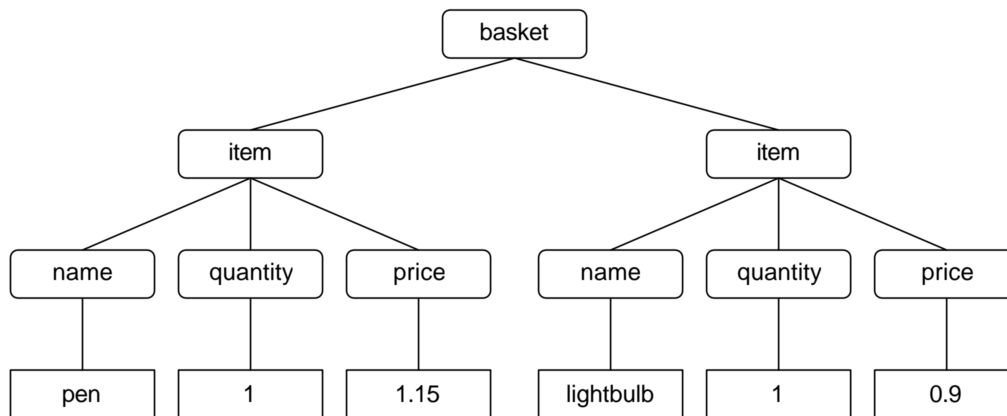

Figure A.2 Example of the DOM representation of an XML document

Using the methods defined in the W3C DOM, it is possible to navigate along the various branches of the tree representation of an XML document or to extract nodes with a particular characteristic (for example the nodes with a given name). The W3C DOM specification is composed of four levels. Level 0 is the way W3C refers to the DOM that is implemented in the early versions of the web browsers, such as Netscape Navigator 3.0 and Microsoft Explorer 3.0. Level 1 focuses on HTML and XML document models. It contains functionalities for document navigation and manipulation. Level 2 adds a style sheet object model to Level 1, and defines functionalities for manipulating the style information attached to a document; it also provides an event model and a support for XML namespaces. Level 3 specifies content models (DTD and Schemas) and document validation. It also specifies document loading and saving, views and formatting. The alternative to DOM is the SAX standard [SAX02], which allows accessing XML documents sequentially, providing the possibility of associating a specific event to each tag. In other words, it is an event-based approach to the parsing of XML documents; this means that when a parser encounters an element, it treats this as an event and calls the corresponding code in order to deal with it.

A.3.2 XML Schema

XML Schema [Fal01] has been introduced in order to define the structure of an XML document. In fact, one of the most interesting advantages of using XML documents is the possibility of *validating* them. XML documents whose syntax has been checked successfully are called valid documents; in particular, an XML document is considered valid if there is an XML Schema document (or a DTD document [BraPS00]) associated with it and if it is compliant to this definition of its structure.

For example, the XML Schema Document related to the XML file that we have presented before is the following:

```
<xsd:schema xmlns:xsd="http://www.w3.org/1999/XMLSchema">
 <xsd:element name="basket">
  <xsd:complexType>
   <xsd:element name="item" type="xsd:string" minOccurs="0"
maxOccurs="500">
     <xsd:complexType>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="price" type="xsd:decimal"/>
      <xsd:element name="quantity" type="xsd:decimal"/>
     <xsd:attribute name="category"/>
    </xsd:complexType>
   </xsd:element>
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

As you can see from this example, XML Schema describes the structure that an XML instance must have, through the definition of its elements and the relations between them. Each element may have attributes and may contain other elements, called children elements. In this case, the element `item` is characterised by having the `category` attribute and by containing the `name`, `price` and `quantity` elements. Moreover, it is worth noticing that the `category` element is defined as a complex type, since it contains other elements. Another fundamental characteristic is the possibility to specify the number of occurrences of a particular element, respectively by means of the `minoccurs` (minimum number) and `maxoccurs` (maximum number) attributes (we exploit this feature in order to specify a valid topology of the XML documents managed by XMIDDLE and to

perform the reconciliation process, as we have discussed in Chapter 4). If these attributes are not present, the element must appear only once by default. If the value of `maxoccurs` is not specified, its default value is equal to the value of `minoccurs`. Furthermore, in order to indicate that there is no upper bound, it is possible to set it to the value `unbounded`.

A.3.3 XPath

The standard way to locate information within an XML document is through a language known as the XML Path Language or XPath [ClaD99]. It can be used to refer to textual data, elements, attributes and other information in an XML document.

As we have discussed before, XML documents can be represented as tree structures. In order to address a specific element of the tree, it is possible to use the XPath expressions, which are compact strings with non-XML style syntax, which describe how to navigate from a starting location (the so-called *context node*) to the requested element. XPath models an XML document as a tree of nodes. There are different types of nodes, including element nodes, attribute nodes and text nodes. XPath also defines a way to compute a string value for each type of node. For some types of node, the string value is a property of the node; for other ones, the string value is computed from the string value of descendant nodes (for example in the case of a parent node of a text node). An exhaustive explanation of the definition of the XPath expressions is beyond the scope of this appendix; a detailed presentation can be found in [Hol01].

A.3.4 XML parsers

The XML parsers provide the essential functionalities for accessing XML documents, according the standards that we have presented before, such as SAX and DOM. There are essentially two basic types of XML parsers: validating and non-validating. The first ones check an XML document against a DTD or a XML Schema to ensure that its structure is correct (according the specified syntax). This requires an additional computational work and slows down the document processing.

A non-validating parser skips this step and just ensures that an XML document is well-formed, in other words, that it conforms to the W3C standard.

A.3.4.1 Xerces and Xalan

The Xerces project [Apa02a] (a part of the open source XML Apache project) provides XML parsers for a variety of languages, including Java, C++ and Perl. It implements the W3C XML and DOM (Level 1 and 2) standards as well the SAX standard. The most interesting features of this parser are its modularity and high reconfigurability. It is worth noting that it also supports the W3C XML Schema standard.

Xalan [Apa02b] is an XSLT processor implementing the W3C XSLT [ClaD99] (we have not presented this technology in detail, since it was not used in the development of XMIDDLE) and XPath specifications. The XPath processor can be used as a stand-alone unit. Xalan is currently available for Java and C++.

A.3.4.2 NanoXML and kXML

Currently the main alternatives to Xerces and Xalan are NanoXML [Des02] and kXML [GHKP02]. The first one is a small (the core requires only 6 KB) and fast parser that also supports the SAX standard. Moreover, it is possible to extend the core, using different modules; in particular, one of these is specifically designed for the J2ME Connected Limited Device Configuration.

The second one is a more powerful parser for the Java environment (including Java 2 Micro Edition). Because of its small footprint size, it is especially suited for Java applications running on mobile devices like small PDAs or cellular phones. The key features of kXML are a complete support for XML parsing (including, for example, namespaces) and the possibility to deal with other SGML formats (i.e. HTML). It also provides optional modules such as kDOM and a WAP support [OMA02].

It is worth noting that both kXML and NanoXML are non-validating parsers. Moreover, these XML parsers can also be classified by how they process and represent an XML document [Gig01]. NanoXML is a single-step parser. Given a document, NanoXML parses it in a single operation and returns the document as a tree of objects. Instead kXML is an incremental parser: it parses documents one

piece at a time. There are advantages and disadvantages to either approach: in case of large documents, the single-step approach uses much more memory because the entire document is held in memory, but, at the same time, it is the best choice in the case of applications that need to traverse the document multiple times; in fact an incremental processing sensibly slows down the computation. The XML Apache project parsers follow the single step approach.

A.4 XML and mobile middleware systems

As we have discussed in the previous chapters, XML can also be exploited to represent meta-information in order to enable the development of a new generation of mobile middleware that have been defined *application-aware*. In fact, by using XML, it is possible to specify the rules or, simply, to describe the execution context or the data managed by applications (as in XMIDDLE). For example, it is possible to exploit XML and related technologies to define the profile of a device (for instance its resources or memory capabilities) and on the other hand, by using this language, the middleware can provide applications with information about the context (for instance the level of battery charge or the quality of the available bandwidth).

Therefore, one of the most interesting XML applications is the *Wireless Markup Language* (WML) [OMA02]. This and its associated protocol, the *Wireless Application Protocol* (WMP), target hand-held devices such as cellular phones and PDAs. WML is characterized by a limited syntax that is relatively easy to implement for devices with restricted hardware capabilities. Moreover, it is worth noticing that it is possible to transform an XML document into a WML document (or other different formats) in a very easy way, using the Extensible Stylesheet Language (XSL) [ClaD99].

Appendix B

XMIDDLE API

In this appendix we present the complete API that is provided to programmers in order to build applications based on the XMIDDLE platform. This documentation was obtained by using the JavaDoc tool [Sun02g] included in the Standard Edition of the Java Software Development Kit. The API is composed of the public methods of the `SimpleManager` class, which implements the `Manager` interface: as we have discussed previously, this class provides a high-level abstraction of our middleware system.

Method Summary	
void	**addHostListener**(Host host, HostListener application) Adds a `HostListener` object for the given `Host` object.
void	**addHostListener**(HostListener application) Adds a `HostListener` object.
void	**addHostListenerPrimID**(java.lang.Object primID, HostListener application) Adds a `HostListener` object for the `Host` object identified by the given primary ID.
void	**addHostListenerSecID**(java.lang.Object secID, HostListener application) Adds a `HostListener` object for the `Host` object identified by the given primary ID.
void	**close**(org.w3c.dom.Element element, java.lang.Integer appID) Commits the changes performed on the given `Element`.
boolean	**connect**() Connects to the network.
void	**disconnect**() Disconnects from the network.

void	**exit**() Terminates the middleware services.
Boolean	**export**(org.w3c.dom.Element element, java.lang.Integer appID) Starts exporting the given element.
Boolean	**export**(java.lang.String path, java.lang.Integer appID) Starts exporting the given element.
java.util.Enumeration	**getExports**() Returns the ExportList.
java.util.Enumeration	**getHosts**() Returns the list of the hosts that are currently in reach.
static SimpleManager	**getInstance**() Returns an instance of a SimpleManager object.
java.util.Enumeration	**getLinkedBy**() Returns the LinkedBy table.
java.util.Enumeration	**getLinkedFrom**() Returns the LinkedFrom table.
java.lang.Object	**getPrimaryID**() Returns the primary ID of the local host.
ApplicationProfile	**getProfile**(java.lang.Integer appID) Returns the profile of the application with the given ID.
java.lang.Object	**getSecondaryID**() Returns the secondary ID of the local host.
Boolean	**getStatus**() Returns the connection status (CONNECTED or DISCONNECTED).
Void	**init**() Initialises a new XMIDDLE session.
Void	**link**(Host host, java.lang.Integer appID, java.lang.String remoteExport, java.lang.Integer localAppID, org.w3c.dom.Element localRoot) Links to a remote element (mounting of a sub-tree exported by another host).
org.w3c.dom.Element	**open**(java.lang.String path, java.lang.Integer appID) Retrieves a sub-tree identified by the given XPath expression.
void	**queue**(java.lang.Object[] param1) Queues a protocol request to the Localhost thread.
java.lang.Integer	**registerApplication**(XApp app)

	Registers a new application to the middleware.
void	**registerProtocol**(java.lang.String className, java.lang.String protocolName) Registers a new protocol as available to the middleware.
void	**removeHostListener**(Host host, HostListener application) Removes the given HostListener object, referring to the given host.
void	**removeHostListener**(HostListener application) Removes the given HostListener object.
void	**removeHostListenerPrimID**(java.lang.Object primID, HostListener application) Removes the given HostListener object, referring to the given host identified by the given primary ID.
void	**removeHostListenerSecID**(java.lang.Object secID, HostListener application) Removes the given HostListener object, referring to the given host identified by the given secondary ID.
void	**send**(Data param1) Sends a data packet to another host.
void	**sync**(Host host, java.lang.Integer origAppID, java.lang.String origPath, java.lang.Integer localAppID, java.lang.String localPath) Synchronizes (reconciles) our data with the given Host.
void	**unexport**(org.w3c.dom.Element element, java.lang.Integer appID) Stops exporting the given element.
void	**unexport**(java.lang.String path, java.lang.Integer appID) Stops exporting the given element.
void	**unlink**(org.w3c.dom.Element element, java.lang.Integer appID) Stops linking to the specified element of the specified ID.

Method Detail

addHostListener

```
public void addHostListener(HostListener application)
```

Adds a `HostListener` object.

Specified by:

`addHostListener` in interface `Manager`

Parameters:

`application` - the object to be notified.

---

addHostListener

```
public void addHostListener(Host host,
                            HostListener application)
```

Adds a `HostListener` object for the given `Host` object.

Specified by:

`addHostListener` in interface `Manager`

Parameters:

`host` - the host that needs a `HostListener`.

`application` - the host listener

---

addHostListenerPrimID

```
public void addHostListenerPrimID(java.lang.Object primID,
                                  HostListener application)
```

Adds a `HostListener` object for the `Host` object identified by the given primary ID.

Specified by:

`addHostListenerPrimID` in interface `Manager`

Parameters:

`primID` - the primary ID of the host

`application` - the host listener

---

addHostListenerSecID

```
public void addHostListenerSecID(java.lang.Object secID,
                                 HostListener application)
```

Adds a `HostListener` object for the `Host` object identified by the given primary ID.

Specified by:

`addHostListenerSecID` in interface `Manager`

Parameters:

`application` - the host listener
`secID` - The host, identified (NOT uniquely) by its secondary ID.

---

close

```
public void close(org.w3c.dom.Element element,
                  java.lang.Integer appID)
```

Commits the changes performed on the given `Element`.

Specified by:

`close` in interface `Manager`

Parameters:

`element` - The element to "close"
`appID` - The ID of the application under the tree of which the given element resides.

---

connect

```
public boolean connect()
```

Connects to the network.

Specified by:

`connect` in interface `Manager`

Returns:

true if the operation was successful or false otherwise

---

disconnect

```
public void disconnect()
```

Disconnects from the network.

Specified by:

`disconnect` in interface `Manager`

---

exit

```
public void exit()
```

Terminates the middleware services.

Specified by:

`exit` in interface `Manager`

---

export

```
public boolean export(org.w3c.dom.Element element,
                      java.lang.Integer appID)
```

Starts exporting the given element. This will also create version 0 of the element.

Specified by:

`export` in interface `Manager`

Parameters:

`element` - the element to export

`appID` - the ID of the application under the tree of which the given element resides

Returns:

true if the operation was successful or false otherwise.

---

export

```
public boolean export(java.lang.String path,
                      java.lang.Integer appID)
```

Starts exporting the given element. This will also create version 0 of the element.

Specified by:

`export` in interface `Manager`

Parameters:

`path` - the element to be exported (XPath)

`appID` - the ID of the application under the tree of which the given element resides

Returns:

true if the operation succeded or false otherwise.

---

getExports

```
public java.util.Enumeration getExports()
```

Returns the ExportList.

Specified by:

`getExports` in interface `Manager`

Returns:

the ExportList

---

getHosts

```
public java.util.Enumeration getHosts()
```

Returns the list of the hosts that are currently in reach.

Specified by:

`getHosts` in interface `Manager`

Returns:

an enumeration of `UDPHosts` currently in reach

---

getInstance

```
public static SimpleManager getInstance()
```

Returns an instance of a `SimpleManager` object.

Returns:

an instance of a `SimpleManager` object

---

getLinkedBy

```
public java.util.Enumeration getLinkedBy()
```

Returns the LinkedBy table.

Specified by:

`getLinkedBy` in interface `Manager`

Returns:

the LinkedBy table

---

getLinkedFrom

```
public java.util.Enumeration getLinkedFrom()
```

Returns the LinkedFrom table.

Specified by:

`getLinkedFrom` in interface `Manager`

Returns:

the LinkedFrom table

---

getPrimaryID

```
public java.lang.Object getPrimaryID()
```

Returns the primary ID of the local host.

Specified by:
`getPrimaryID` in interface `Manager`

---

getProfile

```
public ApplicationProfile getProfile(java.lang.Integer appID)
```

Returns the profile of the application with the given ID.
Specified by:
`getProfile` in interface `Manager`
Parameters:
`appID` - the ID of the application the profile of which is requested
Returns:
the profile of the application with the given ID

---

getSecondaryID

```
public java.lang.Object getSecondaryID()
```

Returns the secondary ID of the local host.
Specified by:
`getSecondaryID` in interface `Manager`

---

getStatus

```
public boolean getStatus()
```

Returns the connection status (CONNECTED or DISCONNECTED).
Specified by:
`getStatus` in interface `Manager`
Returns:
CONNECTED or DISCONNECTED

---

init

```
public void init()
```

Initialises a new XMIDDLE session.
Specified by:
`init` in interface **`Manager`**

---

link

```
public void link(Host host,
```

```
java.lang.Integer appID,
java.lang.String remoteExport,
java.lang.Integer localAppID,
org.w3c.dom.Element localRoot)
```

Links to a remote element (mounting of a sub-tree exported by another host).

Specified by:

`link` in interface `Manager`

Parameters:

`host` - the host on which the (exported) element resides

`appID` - the application ID of the remote application exporting the element

`remoteExport` - the exported element itself as advertised by the application

`localAppID` - the local application ID on the tree of which we want the linked element to reside

`localRoot` - the local root element under which the linked element will be placed

---

open

```
public org.w3c.dom.Element open(java.lang.String path,
                                java.lang.Integer appID)
```

Retrieves a sub-tree identified by the given XPath expression.

Specified by:

`open` in interface `Manager`

Parameters:

`appID` - the id of the application under the tree of which the given element resides

`path` - an xpath expression describing the element to open.

Returns:

the element identified by the given XPath expression or NULL if the operation failed (invalid XPath, etc)

---

queue

```
public void queue(java.lang.Object[] param1)
```

Queues a protocol request to the `Localhost` thread.

Specified by:

`queue` in interface `Manager`

Parameters:

`param1` - the request, as an `Object[]` value

---

registerApplication

```
public java.lang.Integer registerApplication(XApp app)
```

Registers a new application to the middleware.

Specified by:

`registerApplication` in interface `Manager`

Parameters:

`app` - the application to be registered

Returns:

the applicationID assigned to the application just registered

---

registerProtocol

```
public void registerProtocol(java.lang.String className,
                             java.lang.String protocolName)
```

Registers a new protocol as available to the middleware.

Specified by:

`registerProtocol` in interface `Manager`

Parameters:

`className` - the absolute class name of the protocol (which must extend `edu.UCL.xmiddle.framework.lib.protocols.Protocol`

`protocolName` - the unique name by which this protocol is identified

---

removeHostListener

```
public void removeHostListener(HostListener application)
```

Removes the given `HostListener` object.

Specified by:

`removeHostListener` in interface `Manager`

Parameters:

`application` - the host listener

---

removeHostListener

```
public void removeHostListener(Host host,
                               HostListener application)
```

Removes the given `HostListener` object, referring to the given host.

Specified by:

`removeHostListener` in interface `Manager`

Parameters:

`host` - the host

`application` - the host listener

---

removeHostListenerPrimID

```
public void removeHostListenerPrimID(java.lang.Object primID,
                                     HostListener application)
```

Removes the given `HostListener` object, referring to the given host identified by the given primary ID.

Specified by:

`removeHostListenerPrimID` in interface `Manager`

Parameters:

`primID` - the primaryID of the host

`application` - the host listener

---

removeHostListenerSecID

```
public void removeHostListenerSecID(java.lang.Object secID,
                                    HostListener application)
```

Removes the given `HostListener` object, referring to the given host identified by the given secondary ID.

Specified by:

`removeHostListenerSecID` in interface `Manager`

Parameters:

`secID` - the secondary ID of the host

`application` - the host listener

---

send

```
public void send(Data param1)
```

Sends a data packet to another host.

Specified by:

`send` in interface `Manager`

Parameters:

`param1` - the `Data` packet to send

---

sync

```
public void sync(Host host,
                 java.lang.Integer origAppID,
                 java.lang.String origPath,
                 java.lang.Integer localAppID,
                 java.lang.String localPath)
```

Synchronizes (reconciles) our data with the given `Host`.

Specified by:

sync in interface Manager

Parameters:

host - the host on which the element resides.

localAppID - the application ID on the localhost.

localPath - the path name on the localhost.

---

unexport

```
public void unexport(org.w3c.dom.Element element,
                     java.lang.Integer appID)
```

Stops exporting the given element.

Specified by:

unexport in interface Manager

Parameters:

element - the element to stop exporting

appID - the application ID to which this element belongs

---

unexport

```
public void unexport(java.lang.String path,
                     java.lang.Integer appID)
```

Stops exporting the given element.

Specified by:

unexport in interface Manager

Parameters:

path - the element to stop exporting (as XPath)

appID - the application ID to which this element belongs

---

unlink

```
public void unlink(org.w3c.dom.Element element,
                   java.lang.Integer appID)
```

Stops linking to the specified element of the specified ID. Note that this does not delete the element.

Specified by:

unlink in interface Manager

Parameters:

appID - the application ID on the tree of which this element resides

---

References

[AngCKL98] Angin, O., Campbell, A. T., Kouvanis, M. E., Liao, R. R. F. The Mobiware Toolkit: Programmable Support for Adaptive Mobile Networking. *IEEE Personal Communications Magazine, Special Issue on Adaptive Mobile Systems*, August 1998.

[AhuCG86] Ahuja, S., Carriero N., Gelernter D. Linda and Friends. *IEEE Computer*, Vol. 19, No. 8, pp. 26-34, August 1986.

[Apa02a] The Apache Software Foundation. *Xerces 2 Java Parser 2.2.0 Release*, 2002.
http://xml.apache.org/xerces2-j/

[Apa02b] The Apache Software Foundation. *Xalan-Java Version 2.4.0*, 2002.
http://xml.apache.org/xalan-j/

[Ava02] Avaya Inc. *Avaya Wireless Network Solutions*, 2002.
http://www.avaya.com

[Bak02] Bakker, D., E. Middleware. In *Encyclopedia of Distributed Computing*, Kluwer Academic Press, 2002.

[BelCS99] Bellavista, P., Corradi, A., Stefanelli, C. A Secure and Open Mobile Agent Programming Environment. *Proceedings of the Fourth International Symposium on Autonomous Decentralized Systems (ISADS '99)*, Tokio, March 1999.

[BelCS01] Bellavista, P., Corradi, A., Stefanelli, C. Mobile Agent Middleware to Support Mobile Computing. *IEEE Computer*, Vol. 34, No. 3, pp.73-81, March 2001.

[Ber96] Bernstein, A. Middleware, A Model for Distributed System Services, *Communications of the ACM*, Vol. 39 No. 2 February 1996

[BirN84] Birrell, A., D., Nelson, B., J. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, Vol. 2, No. 1, pp 33-59, February 1984.

[Blu02] The Bluetooth Special Interest Group. *Bluetooth V1.1 Public Specifications*, 2002.
http://www.bluetooth.org

[BraPS00] Bray, T., Paoli, J., Sperberg-McQueen, C., M. *Extensible Markup Language (XML) W3C Recommendation*, World Wide Web Consortium, October 2000.
http://www.w3.org/TR/REC-xml

[CabLZ00] Cabri, G., Leonardi, L., Zambonelli, F. XML Dataspaces for Mobile Agent Coordination. *Proceedings of the 2000 ACM Symposium on Applied Computing (SAC 2000).* Como, Italy, 2000.

[CamCK99] Campbell, A., Coulson, G., Kounavis, M. Managing Complexity: Middleware Explained. *IT Professional*, IEEE Computer Society, Vol. 1, No. 5, pp. 22-28, September/October 1999.

[CapBMEG02] Capra, L., Blair, G. S., Mascolo, C., Emmerich, W., Grace, P. Exploiting Reflection in Mobile Computing Middleware. *ACM SIGMOBILE Mobile Computing and Communications Review*, To appear.

[CapEM01] Capra, L., Emmerich, W., Mascolo, C. Exploiting Reflection and Metadata to Build Mobile Computing Middleware. *Proceedings of the Workshop on Mobile Computing Middleware*, Heidelberg, November 2001.

[CarG89] Carriero, N., Gelernter D. Linda in Context. *Communications of the ACM*, Vol. 32, No. 4, pp. 444-458, April 1989.

[Ced02] P. Cederquist et alii. *Version Management with CVS*, 2002.
http://www.cvshome.org

[Cla99] Clark, J. *XSL Transformations (XSLT) Version 1.0 W3C Recommendation*, World Wide Web Consortium, November 1999.
http://www.w3.org/TR/xslt

[ClaD99] Clark, J., De Rose, S. *XML Path Language (XPath) Version 1.0 W3C Recommendation,* World Wide Web Consortium, November 1999.
http://www.w3.org/TR/xpath

[Cou89] Courington, W. The Network Software Environment, *Technical Report FE197-0*, Sun Microsystem Inc., February 1989.

[CouDK01] Couloris, G., Dollimore, J., Kindberg, T. *Distributed Systems Concepts and Design*. Third Edition. Addison-Wesley, 2001.

[DavFWB98] Davies, N., Friday, A., Wade, S., Blair, G. L^{2}imbo A distributed systems platform for mobile computing. *ACM Mobile Networks and Applications (MONET) Special Issue on Protocols and Software Paradigms of Mobile Network*, Vol. 3 No. 2, 1998.

[DavSK01] Davis, G., D., Subrajmanian, E., Konda, S., Granger, H., Collins, M., Westerberg, A., W. Creating Shared Information Spaces to Support Collaborative Design Work. *Information Systems Frontiers Special Issue on Workflow management systems*, July 2001.

[Des02] De Scheemaecker, M. *NanoXML*, SourceFourge Project, 2002. http://nanoxml.sourceforge.net/

[Emm00] Emmerich, W. *Engineering Distributed Objects*, Wiley, 2000.

[Exo02] Exolab Group. *OpenORB*, 2002. http://openorb.exolab.org/

[Fal01] Fallside, D. C. *XML Schema Part 0: Primer W3C Recommendation*, World Wide Web Consortium, May 2001. http://www.w3.org/TR/xmlschema-0/

[For94] Forman, G. H., Zahorjan, J., The Challenges of Mobile Computing. *IEEE Computer*, Vol. 27, No. 4, pp. 38-47, April 1994.

[FriKV00] Fritsch, D., Klinec, D., Volz, D. Nexus Positioning and Data Management for Location Aware Applications. *Proceedings of the 2nd Symposium on Telegeoprocessing*, pp. 171-184, Nice, France, 2000.

[FugPV98] Fuggetta, A., Picco, G. P., Vigna, G. Understanding Code Mobility. *IEEE Transactions on Software Engineering*. Vol. 34, No. 5, 1998.

[Gel85] Gelertner, D. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems.* Vol. 7, No. 1, pp. 80-112, January 1985.

[GerHKP02] Gerard, A., Haustein, S., Kroll, M., Pleumann, J. *kXML Project Documentation*, 2002. http://www.kxml.org/

[Gig01] Giguere, G. Parsing XML in CLDC-Based Profiles. *Java Wireless Developers*, July 2001. http://wireless.java.sun.com/configurations/ttips/xmlparse/

[Gra78] Gray, J. Notes on Database Operating Systems. In Bayer, R., Graham, R., Seegmuller, G. (eds.), *Operating Systems: An Advanced Course*, Lectures Notes in Computer Science, Vol. 60, pp. 393-481, Springer-Verlag, 1978.

[HaaCC99] Haahr, M., Cunningham, R, Cahill, V. Supporting CORBA Applications in a Mobile Environment. *MobiCom '99: 5th International Conference on Mobile Computing and Networking*. Seattle, August 1999.

[HaaCC00] Haahr, M., Cunningham, R., Cahill, V. Towards a Generic Architecture for Mobile Object-Oriented Applications. *SerP 2000 Workshop on Service Portability*, San Francisco, December 2000.

[Hal96] Hall, C., L. *Building Client/Server Applications using TUXEDO,* John Wiley and Sons, 1996.

[Han02] The Handhelds Group. *The Familiar Project Linux Distribution*, 2002.
http://www.handhelds.org

[HanR02] Handorean, R., and Roman, G., C., Service Provision in Ad-Hoc Networks. In *Proceedings of the 5th International Conference COORDINATION 2002.* Lecture Notes in Computer Science 2315, Springer-Verlag, pp 207 – 219, April 2002.

[HeiPGP92] Heidemann, J. S., Page, T. W., Guy, R. G., Popek, G. J. Primarily disconnected operation: Experiences with Ficus. *Proceedings of the Second Workshop on the Management of Replicated Data*, Monterey, California, pp. 56-60, November 1992.

[Hol01] Holzer, S. *Inside XML*. New Riders Publishing, Indianapolis, 2001.

[HorC01] Horstmann, C., S., Cornell, G. *Core Java 2*. Prentice Hall Professional Technical Reference, 2001.

[HP02] Hewlett Packard Company. *iPaq 3660 Technical Documentation*, 2002.
http://www.hp.com

[IBM98] IBM Corp. XML TreeDiff, November 1998.
http://www.alphaworks.ibm.com/tech/xmltreediff

[IBM02a] IBM Corp. *CICS*, 2002.
http://www.software.ibm.com/ts/cics/

[IBM02b] IBM Corp. *TSpaces, Intelligent Connectionware*, 2002. http://www.almaden.ibm.com/cs/TSpaces/

[IEEE02] Institute of Electrical and Electronics Engineers. *IEEE Wireless Standards Overview*, 2002. http://standards.ieee.org/wireless/overview.html

[IETF01] The Internet Engineering Task Force. *ONC Remote Procedure Call*, 2001. http://www.ietf.org/html.charters/oncrpc-charter.html

[IETF02] The Internet Engineering Task Force. *Mobile Ad-Hoc Networks (MANET),* 2002. http://www.ietf.org/html.charters/manet-charter.html

[IrDA02] The Infrared Data Association. *IrDA Serial Infrared Data Link Standard Specifications*, 2002. http://www.irda.org

[Int01] Intel Corporation. *Mobile Systems and Security Technologies for Safe, Anywhere/Anytime Computing, 2001.* http://www.intel.com/ebusiness/products/related_mobile/

[Int02] Intel Corporation. *Intel StrongARM SA-110 processor*, 2002. http://www.intel.com/design/strong/sa110.htm

[IONA02] Iona Technologies. *Orbix E2A Application Server Platform*, 2002. http://www.iona.com/products/appserv.htm

[ISO86] International Organization for Standardization (ISO). *ISO 8879:1986(E). Information Processing – Text and Office Systems – Standard Generalized Markup Language (SGML).* October 1986.

[Jac94] Jacobson, I., *Object-Oriented Software Engineering A Use Case Driven Approach.* Addison-Wesley, 1994.

[JosTK97] Joseph, A. D., Tauber, J., A., Kaashoek, M., F. Mobile Computing with the Rover Toolkit. *IEEE Transactions on Computers, Special Issue on Mobile Computing*, Vol. 46, No.3, March 1997.

[KalBHOG88] Kalwell Jr., L., Beckhardt, S., Halvorsen, T., Ozzie, R., Grief, I. Replicated document management in a group communication system. *Proceedings Conference on Computer-Supported Cooperative Work*, Portland, Oregon, September 1988.

[Kaz88] Kazar, M., L. Synchronization and caching issues in the Andrew File System. *Proceedings of the Winter 1988 Usenix Conference*. Usenix Association, January 1988.

[KerRSD01] Kermarrec, A, Rowstron, A., Shapiro, M., Drueschel, P. The IceCube approach to the reconciliation of divergent replicas. *Proceedings of the Twentieth ACM Symposium on Principles of Distributed Computing (PODC 2001)*, Newport, Rhode Island, August 2001.

[KisS92] Kistler, J., J. Satyanarayanan, M. Disconnected Operations in Coda File System. *ACM Transactions on Computer Systems*, Vol. 10, No. 2, pages 3-25, February 1992.

[Kle95] Kleinrock, L. Nomadic Computing An Opportunity. *ACM SIGCOMM Computer Communication Review*, Vol. 25, No. 1, pp 36-40, January 1995.

[Kle96a] Kleinrock, L. Nomadicity: Anytime, Anywhere in A Disconnected World. Mobile Networks and Applications, Vol.1, No. 4, pp. 351-357, January 1996.

[Kle96b] Kleinrock, L. Nomadic Computing. *Proceedings of the IFIP/ICCC International Conference on Information Network and Data Communication*, Trondheim, Norway, June 1996.

[Kon00] Kon, F., Román, M., Liu, P., Mao, J., Yamane, T., Magalhães, L., C., Campbell, R., H. Monitoring, Security, and Dynamic Configuration with the DynamicTAO Reflective ORB. *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*. New York, 2000.

[KumS95] Kumar, P., Satyanarayanan, M. Flexible and Safe Resolution of File Conflicts. *Proceedings of Winter Technical Conference, USENIX 1995*, pp 95-106, 1995.

[Lam78] Lamport, L. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, Vol. 21, No. 7, pp. 77-101, 1978.

[LanO98] Lange, D., Oshima, M. *Programming and Developing Java Mobile Agents with Aglets*. Addison-Wesley, 1998.

[Lot00] Lotus Development Corporation. *Inside Notes: The Architecture of Notes and the Domino Server*. Lotus Development Corporation, Cambridge, MA, October 2000.

[MacC98] M
acker, J, Corson, M. S., Mobile Ad-hoc Networking and the IETF. *ACM Mobile Computing and Communication Review*. Vol. 2, No. 1, January 1998.

[Mah02] Mahmoud, Q., H. *Learning Wireless Java*, O'Reilly, 2002.

[MacT82] MacGregor, W., I., Tappau, D., C. *The Cronus Virtual Local Network*, RFC 824, August 1982.

[MasCE02] Mascolo, C., Capra, L., Emmerich, W. Middleware for Mobile Computing (A Survey) in *Networking 2002 Tutorial Papers*. Gregori, E., Anastasi, G., Basagni, S. (eds.), Lecture Notes on Computer Science, No. 2497. To appear.

[MasCZE02] Mascolo, C., Capra, L., Zachariadis, S., Emmerich, W., XMIDDLE: A Data-Sharing Middleware for Mobile Computing, *Personal and Wireless Communications Journal*, Vol. 2, No. 1, Kluwer, April 2002.

[MarTU00] Maruyama, H., Tamura, K., Uramoto, N. Digest Values for DOM*, RFC 2802*, IETF, April 2000.
http://www.ietf.org/rfc/rfc2803.txt

[Mic02a] Microsoft Corp. *Pocket PC Operating System*, 2002.
http://www.microsoft.com/mobile/pocketpc/software/default.asp

[Mic02b] Microsoft Corp. *Mobile Information Server*, 2002.
http://www.microsoft.com/miserve

[MilDW99] Milojicic, D., Douglis, F., Wheeler, R. (eds.). *Mobility Processes, Computers, and Agents*. Addison-Wesley, 1999.

[Nob00] Noble, B. Support for Mobile, Adaptive Applications. *IEEE Personal Communications.* Vol. 7, No.1, February 2000.

[OMA02] Open Mobile Alliance. *The WAP Forum*. 2002.
http://www.wapforum.org

[OMG02] Object Management Group. *CORBA Component Model*, 2002.
http://www.corba.org

[Open02] The Open Group. *Distributed Transaction Processing*, 2002.
http://www.opengroup.org

[Palm02] PalmSource Inc., *Palm OS: The Foundation of the Future of Mobile Computing*, 2002.

http://www.palmos.com

[PeiS97] Peine, H., Stolpmann T. The Architecture of the Ara Platform for Mobile Agents. In Rothermel, K., Popescu-Zeletin, R. (eds.) *Proceedings of the First International Workshop on Mobile Agents.* Lecture Notes in Computer Science. No.1219, pp. 50-61 Springer-Verlag April 1997.

[Per01] Perkins, C., E. (ed.) *Ad-hoc Networking*. Addison-Wesley, 2001.

[PicMR99] Picco, G., P., Murphy, A., M., Roman, G., C. Lime: Linda Meets Mobility. *Proceedings of the 21st International Conference on Software Engineering (ICSE'99),* Los Angeles (USA), Garlan, D., Kramer, J. (eds.), ACM Press, pp 368-377, May 1999.

[Pic01] Picco, G., P. Mobile Agents: An Introduction. *Journal of Microprocessors and Microsystems, Special Issue on Mobile Agents*, A. Corradi (ed.), Vol. 25, No. 2, pp. 65-74, April 2001.

[PitM01] Pitt, E., McNiff, K. *Java.rmi: The Remote Method Invocation Guide*. Addison Wesley, June 2001.

[PooCE99] Poon, T., Curbera, F., Epstein, D. A. *Efficient Enconding of XML Updates*. Slides of the talk at T.J. Watson Research Center, August 1999.

[Pop98] Pope, A. *The Corba Reference Guide: Understanding the Common Object Request Broker Architecture*. Addison-Wesley, January 1998.

[PopW85] Popek, G., J, Walker, B., J. (eds.), *The LOCUS Distributed System Architecture*, Computer Systems Series, Cambridge, MA, MIT Press, 1985.

[Psi02] PsiNaptic Inc. *JMatos Technology*, 2002. http://www.psinaptic.com

[Raa97] Raatikainen, K. Bridging and Wireless Access for Terminal Mobility in CORBA. *OMG Technical Meeting*, Helsinki, 1997.

[Rec02] Recursion Software Inc. *Voyager*, 2002. http://www.recursionsw.com/products/voyager/voyager.asp

[ReiKRSP94] Reiner, P., Keidermann, J., Ratner, D., Skinner, G., Popek, G. Resolving File Conflicts in the Ficus File System. *USENIX 1994 Summer Conference Proceedings*, Boston, MA, 1994.

[Rog97] Rogerson, D. *Inside COM*, Microsoft Press, 1997.

[Sal02] The Salutation Consortium. *Find-and-Bind for Pervasive Computing*, 2002.
http://www.salutation.org/

[SatNKP94] Satyanarayanan, M., Noble, B., Kumar, P., Price, M. Application-Aware Adaptation for Mobile Computing. *Proceedings of the 6th ACM SIGOPS European Workshop*, Dagstuhl, Germany, September 1994.

[Sat96a] Satyanarayanan, M. Fundamental Challenges in Mobile Computing. *Fifteenth ACM Symposium on Principles of Distributed Computing,* Philadelphia, PA, May 1996.

[Sat96b] Satyanarayanan, M. Mobile Access Information. *IEEE Personal Communications*, Vol. 3, No. 1, February 1996.

[SAX02] SAX Open Development Group. *SAX, A Simple API for XML*, 2002.
http://www.saxproject.org/

[Sch00] Schiller, J. *Mobile Communications*. Addison-Wesley, 2000.

[SchBBK95] Schill, A., Bellmann, W., Bohrnak, W., Kummel, S. System Support for mobile distributed applications. *Proceedings of 2nd International Workshop on Services in Distributed and Networked Environments*, Whistler, British Columbia, 1995.

[ShaRK00] Shapiro, M., Rowstrom, A., Kermarrec, A. Application independent reconciliation for nomadic applications. *Proceedings of the SIGOPS European Workshop "Beyond the PC: New Challenges for Operating Systems",* Kolding (Denmark), 2000.

[SmiRK02] Smith, L., Roe, C., Knudsen, K., S. A Jini Lookup Service for Resource-constrained Devices. *4th IEEE International Workshop on Networked Appliances*, Gaithersburg, USA, 2002.

[Sta99] Standish, T., A. *Data structures techniques*. Addison-Wesley, April 1999.

[Sun01] Sun Microsystem Inc. PersonalJava Specification, 2001
http://java.sun.com/products/personaljava/

[Sun02a] Sun Microsystem Inc. *Enterprise JavaBeans Technology*, 2002.
http://java.sun.com/products/ejb/

[Sun02b] Sun Microsystems Inc. *Java Message Service,* 2002. http://java.sun.com/products/jms/

[Sun02c] Sun Microsystem Inc. *JavaSpaces Technology*, 2002. http://java.sun.com/products/javaspaces/

[Sun02d] Sun Microsystem Inc. *Jini Technology*, 2002. http://java.sun.com/products/jini/

[Sun02e] Sun Microsystem Inc. *The Java Language: an overview*, 2002. http://java.sun.com/docs/overviews/java/java-overview-1.html

[Sun02f] Sun Microsystem Inc., *Java Cryptographic Extension (JCE),* 2002. http://java.sun.com/products/jce/

[Sun02g] Sun Microsystem Inc. *Java Doc Tool*, 2002. http://java.sun.com/j2se/javadoc/

[Sym02] Symbian Ltd. *Symbian OS Technology*, 2002. http://www.symbian.com/technology/technology.html

[Syn02] SynchML Initiative Ltd. *SynchML Technology*, 2002. http://www.synchml.org

[Tai79] Tai, K, C. The Tree-To-Tree Correction Problem. *Journal of the ACM*, Vol. 29, No. 3, pp. 422-433, 1979.

[TerTPDS95] Terry, D., B., Theimer, M., M., Petersen, K., Demers, A. J., Spreitzer, M., J. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. *SIGOPS '95*, Copper Mountain, CO, U.S.A., 1995.

[TerPST98] Terry, D., B., Petersen, K., Spreitzer, M., J, Theimer, M., M. The Case for Non-trasparent Replication: Examples from Bayou. *IEEE Data Engineering Bulletin.* Vol. 21, No. 4, December 1998.

[WacAS99] Wackerow, D., Armitage, D., Skinner, T. *MQSeries Version 5.1 Administration and Programming Examples*. IBM Redbooks, 1999. http://www.redbooks.ibm.com

[Wal99] Waldo, J. The Jini Architecture for Network-centric Computing. *Communications of ACM.* Vol. 42, No. 7, pp 76-82, 1999.

[W3C02a] World Wide Web Consortium, *HyperText Markup Language*

(HTML), 2002.
http://www.w3.org/MarkUp/

[W3C02b] World Wide Web Consortium, *Document Object Model (DOM)*, 2002.
http://www.w3.org/DOM/

[WicMSF98] Wickoff, P., McLaughry, S., Lehman, T., Ford, D. TSpaces. *IBM Systems Journal*, Vol. 37, No. 3, August 1998.

[ZhaS89] Zhang, K., Sasha, D., Simple Fast Algorithm for the Editing Distance between Trees and Related Problems. *SIAM Journal of Computing*. Vol. 18, No. 6. pp. 1245-1262, 1989.