Mobile Computing Middleware

Cecilia Mascolo, Licia Capra and Wolfgang Emmerich

Dept. of Computer Science
University College London
Gower Street, London, WC1E 6BT, UK
{C.Mascolo|L.Capra|W.Emmerich}@cs.ucl.ac.uk

Abstract. Recent advances in wireless networking technologies and the growing success of mobile computing devices, such as laptop computers, third generation mobile phones, personal digital assistants, watches and the like, are enabling new classes of applications that present challenging problems to designers. Mobile devices face temporary loss of network connectivity when they move; they are likely to have scarce resources, such as low battery power, slow CPU speed and little memory; they are required to react to frequent and unannounced changes in the environment, such as high variability of network bandwidth, and in the resources availability. To support designers building mobile applications, research in the field of middleware systems has proliferated. Middleware aims at facilitating communication and coordination of distributed components, concealing complexity raised by mobility from application engineers as much as possible. In this survey, we examine characteristics of mobile distributed systems and distinguish them from their fixed counterpart. We introduce a framework and a categorisation of the various middleware systems designed to support mobility, and we present a detailed and comparative review of the major results reached in this field. An analysis of current trends inside the mobile middleware community and a discussion of further directions of research conclude the survey.

1 Introduction

Wireless devices, such as laptop computers, mobile phones, personal digital assistants, smartcards, watches and the like, are gaining wide popularity. Their computing capabilities are growing quickly, while they are becoming smaller and smaller, and more and more part of every day life. These devices can be connected to wireless networks of increasing bandwidth, and software development kits are available that can be used by third parties to develop applications [70]. The combined use of these technologies on personal devices enables people to access their personal information as well as public resources *anytime* and *anywhere*.

Applications on these types of devices, however, present challenging problems to designers. Devices face temporary and unannounced loss of network connectivity when they move, connection sessions are usually short, they need to discover other hosts in an ad-hoc manner; they are likely to have scarce resources, such as

low battery power, slow CPUs and little memory; they are required to react to frequent changes in the environment, such as change of location or context conditions, variability of network bandwidth, that will remain by orders of magnitude lower than in fixed networks.

When developing distributed applications, designers do not have to deal explicitly with problems related to distribution, such as heterogeneity, scalability, resource sharing, and the like. *Middleware* developed upon network operating systems provides application designers with a higher level of abstraction, hiding the complexity introduced by distribution. Existing middleware technologies, such as transaction-oriented, message-oriented or object-oriented middleware [21] have been built trying to hide distribution as much as possible, so that the system appears as a single integrated computing facility. In other words, distribution becomes *transparent* [3].

These technologies have been designed and are successfully used for stationary distributed systems built with fixed networks. In the following we analyse the aspect that might not suit mobile settings. Firstly, the interaction primitives, such as distributed transactions, object requests or remote procedure calls, assume a stable and constant connection between components. In mobile systems, in contrast, unreachability is the norm rather than an exception. Secondly, synchronous point-to-point communication supported by object-oriented middleware systems, such as CORBA [53], requires the client asking for a service, and the server delivering that service, to be up and running simultaneously. In a mobile environment, it is often the case that client and server hosts are not connected at the same time, because of voluntary disconnections (e.g., to save battery power) or forced disconnection (e.g., no network coverage). Finally, traditional distributed systems assume a stationary execution environment, characterised by stable and high bandwidth, fixed location for every hosts. Recent developments in object oriented middleware have introduced asynchronous primitives in order to allow a more flexible use. As we will see asynchronous primitives could be a better choice in mobile scenarios.

In mobile systems look-up service components are used to hide service location in order to allow reconfiguration with minimal disruption. In mobile systems, where the location of a device changes continuously, and connectivity fluctuates, service and host discovery becomes even more essential, and information on where the services are might have to reach the application layer. While in stationary systems it is reasonable to completely hide context information (e.g., location) and implementation details from the application, in mobile settings it becomes both more difficult and makes little sense. By providing transparency, the middleware must take decisions on behalf of the application. It might, however, be that in constrained and dynamic settings, such as mobile ones, applications can make more efficient and better quality decisions based on application-specific information.

In order to cope with these limitations, many research efforts have focused on designing new middleware systems capable of supporting the requirements imposed by mobility. As a result of these efforts, a pool of mobile middleware

systems has been produced. In this survey, we provide a framework and a classification of the most relevant literature in this area, highlighting goals that have been attained and goals that still need to be pursued. Our aim is to help middleware practitioners and researchers to categorise, compare and evaluate the relative strengths and limitations of approaches that have been, or might be, applied to this problem. Because exhaustive coverage of all existing and potential approaches is impossible, we attempt to identify key characteristics of existing approaches that cluster them into more or less natural categories. This allows classes of middleware systems, not just instances, to be compared.

Section 2 describes the main characteristics of mobile systems and highlights the many extents to which they differ from fixed distributed systems. Section 3 presents a reference model; Section 4 describes the main characteristics of middleware for distributed systems and their limitations in terms of mobility requirements. Section 5 contains a detailed and comparative review of the major results reached to date, based on the framework presented before. For every class, we give a brief description of the main characteristics, illustrate some examples, and highlight strengths and limitations. Section 6 points out future directions of research in the area of middleware for mobile computing and Section 7 concludes the paper.

2 What is a Mobile Distributed System?

In this section, we introduce a framework that we will use to highlight the similarities but more importantly the differences between fixed distributed systems and mobile systems. This preliminary discussion is necessary to understand the different requirements that middleware for fixed distributed systems and middleware for mobile systems should satisfy.

2.1 Characterisation of Distributed Systems and Middleware

A distributed system consists of a collection of components, distributed over various computers (also called hosts) connected via a computer network. These components need to interact with each other, in order, for example, to exchange data or to access each other's services. Although this interaction may be built directly on top of network operating system primitives, this would be too complex for many application developers. Instead, a middleware is layered between distributed system components and network operating system components; its task is to facilitate component interactions. Figure 1 illustrates an example of a distributed system.

This definition of distributed system applies to both fixed and mobile systems. To understand the many differences existing between the two, we extrapolate three concepts hidden in the previous definition that greatly influence the type of middleware system adopted: the concept of *device*, of *network connection* and of *execution context*. These concepts are depicted in Figure 2.

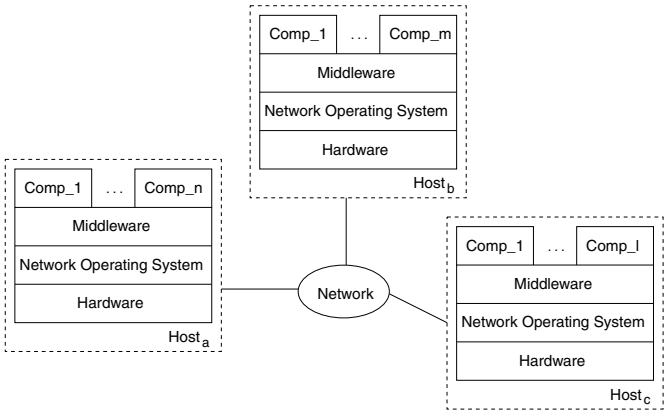

Fig. 1. Example of a distributed system [20].

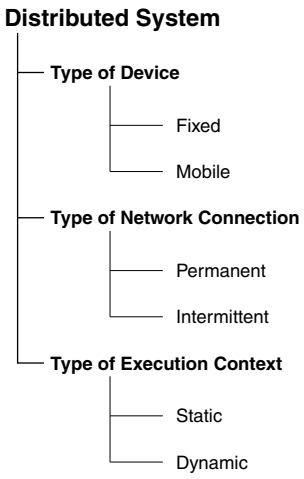

Fig. 2. Characterisation of mobile distributed systems.

Type of Device: as a first basic distinction, devices in a fixed distributed system are *fixed*, while they are *mobile* in a mobile distributed one. This is a key point: fixed devices vary from home PCs, to Unix workstations, to IBM mainframes; mobile devices vary from personal digital assistants, to mobile phones, cameras and smartcards. While the former are generally powerful machines, with large amounts of memory and very fast processors, the latter have limited capabilities, like slow CPU speed, little memory, low battery power and small screen

size.

Type of Network Connection: fixed hosts are usually *permanently* connected to the network through continuous high-bandwidth links. Disconnections are either explicitly performed for administrative reasons or are caused by unpredictable failures. These failures are treated as exceptions to the normal behaviour of the system. Such assumptions do not hold for mobile devices that connect to the Internet via wireless links. The performance of wireless networks (i.e., GSM networks and satellite, WaveLAN [30], HiperLAN [47], Bluetooth [10]) may vary depending on the protocols and technologies being used; reasonable bandwidth may be achieved, for instance, if the hosts are within reach of a few (hundreds) meters from their base station, and if they are few in number in the same area, as, for some of the technologies, all different hosts in a cell share the bandwidth, and if they grow, the bandwidth rapidly drops. Moreover, if a device moves to an area with no coverage or with high interference, bandwidth may suddenly drop to zero and the connection may be lost. Unpredictable disconnections cannot be considered as an exception any longer, but they rather become part of normal wireless communication. Some network protocols, such as GSM, have a broader coverage in some areas but provide bandwidth that is smaller by orders of magnitude than the one provided by fixed network protocols (e.g., 9.6 Kbps against 10Gbs). Also, GSM charges the users for the period of time they are connected; this pushes users to patterns of short time connections. Either because of failures or because of explicit disconnections, the network connection of mobile distributed systems is typically *intermittent*.

Type of Execution Context. With context, we mean everything that can influence the behaviour of an application; this includes resources internal to the device, like amount of memory or screen size, and external resources, like bandwidth, quality of the network connection, location or hosts (or services) in the proximity. In a fixed distributed environment, context is more or less *static*: bandwidth is high and continuous, location almost never changes, hosts can be added, deleted or moved, but the frequency at which this happens is by orders of magnitude lower than in mobile settings. Services may change as well, but the discovery of available services is easily performed by forcing service providers to register with a well-known location service. Context is extremely *dynamic* in mobile systems. Hosts may come and leave generally much more rapidly. Service lookup is more complex in the mobile scenario, especially in case the fixed infrastructure is completely missing. Broadcasting is the usual way of implementing service advertisement, however this has to be carefully engineered in order to save the limited resources (e.g., sending and receiving is power consuming), and to avoid flooding the network with messages. Location is no longer fixed: the size of wireless devices has shrunk so much that most of them can be carried in a pocket and moved around easily. Depending on location and mobility, bandwidth and quality of the network connection may vary greatly. For example, if a PDA is equipped with both a WaveLan network card and a GSM module,

connection may drop from 10Mbs bandwidth, when close to a base station (e.g., in a conference room) to less than 9.6 Kpbs when we are outdoor in a GSM cell (e.g., in a car on our way home).

According to the type of device, of network connection and of context, typical of distributed systems, we can distinguish: *traditional* distributed systems, *nomadic* distributed systems, and *ad-hoc* mobile distributed systems.

2.2 Traditional Distributed Systems

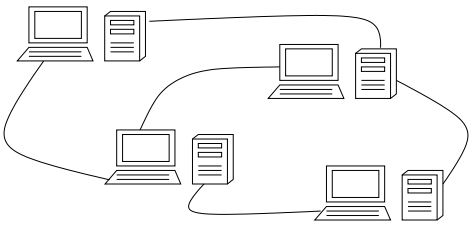

Fig. 3. Structure of a traditional distributed system.

According to the framework previously described, traditional distributed systems are a collection of fixed hosts, permanently connected to the network via high-bandwidth and stable links, executing in a static environment. Application designers building distributed applications on top of this physical infrastructure (Figure 3) often have to guarantee the following non-functional requirements:

- scalability, that is, the ability to accommodate a higher load at some time in the future. The load can be measured using many different parameters, such as, for instance, the maximum number of concurrent users, the number of transactions executed in a time unit, the data volume that has to be handled;
- openness, that is, the possibility to extend and modify the system easily, as a consequence of changed functional requirements. Any real distributed system will evolve during its lifetime. The system needs to have a stable architecture so that new components can be easily integrated while preserving previous investments;
- heterogeneity, that calls for integration of components written using different programming languages, running on different operating systems, executing on different hardware platforms. In a distributed system, heterogeneity is almost unavoidable, as different components may require different implementation technologies. The ability to establish communication between them is essential;

- fault-tolerance, that is, the ability to recover from faults without halting the whole system. Faults happen because of hardware or software failures (e.g., software errors, ageing hardware, etc), and distributed components must continue to operate even if other components they rely on have failed;
- resource sharing. In a distributed system, hardware and software resources (e.g., a printer, a database, etc.), are shared among the different users of the system; some form of access control of the shared resources is necessary in order to grant access to authorised users of the system only.

Traditional distributed systems were the first form of (fixed) distributed system. Since they started to be investigated and employed about 20 years ago, much research effort has been directed to the solutions of the above mentioned problems. Successful middleware technologies have been designed and implemented. We will briefly review some of them in Section 4.

2.3 Nomadic Distributed Systems

Nomadic systems are, in a sense, a compromise between totally fixed and totally mobile systems. Nomadic systems are usually composed of a set of mobile devices and a core infrastructure with fixed and wired nodes.

In nomadic systems mobile devices move from location to location, while maintaining a connection to the fixed network. Usually, the wireless network connects the edges of a fixed infrastructure to the mobile devices. The load of computation and connectivity procedures are mainly carried out on the backbone network. Services are mainly provided by the core network to the mobile clients. In some of these, network disconnection is also allowed and services for transparent reconnection and re-synchronisation are provided. The non-functional requirements listed in the section above still hold as the core of these systems is still a fixed network. The scalability of the system is related to the ability of serving larger numbers of mobile devices in an efficient way. Openness has to deal with the extensibility of the functionality provided by the core network. Heterogeneity is complicated by the fact that different links are present (wireless/fixed), and that many different wireless technologies may coexist in the same network. Fault tolerance: depending on the type of application, disconnection may not be a fault or exception but a functionality. Upon reconnection sharing or services should be allowed. Resource sharing, as in fixed networks: most of the resources are on the core network. However, the picture gets more complex if we allow services to be provided by the mobile devices; in this case discovery, quality, and provision need to be thought differently.

2.4 Ad-Hoc Mobile Distributed Systems

Ad-hoc mobile (or simply ad-hoc) distributed systems consist of a set of mobile hosts, connected to the network through high-variable quality links, and executing in an extremely dynamic environment. They differ from traditional and nomadic distributed systems in that they have no fixed infrastructure: mobile

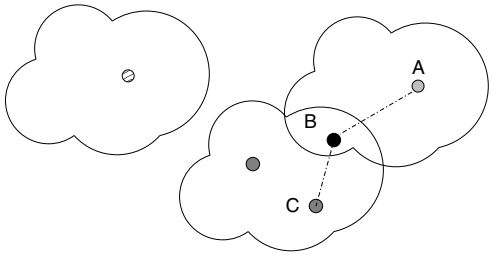

Fig. 4. Structure of an ad-hoc distributed system.

hosts can isolate themselves completely and groups may evolve independently, opportunistically forming clusters depicted as clouds in Figure 4 that might eventually rejoin some time in the future. Connectivity may be asymmetric or symmetric depending, for instance, on the radio frequency of the transmission used by the hosts. Radio connectivity defines the clouds depicted in Figure 4 implying that connection is, by default, not transitive. However ad-hoc routing protocols have been defined [51] in order to overcome this limitation and allow routing of packets through mobile hosts.

Pure ad-hoc networks have limited applications that range from small ad hoc groups to share information in meetings for a short time to military application on battle-fields and discovery or emergency networks in disastered areas.

The non-functional requirements discussed for nomadic systems still hold in this kind of networks. Scalability becomes an issue when big networks need to be coordinated. In contrast with nomadic systems here every service is provided by a mobile host. In big ad-hoc networks with ad-hoc routing enabled, for instance, routing tables and messages might become big if the network is large. Bandwidth and connectivity may vary depending on concentration and interference among the devices. As for heterogeneity, the network might provide different connectivity strategies and technologies (e.g., Bluetooth areas connected to WaveLan areas) that need coordination. In terms of fault tolerance, given the highly dynamic structure of the network, disconnection has to be consider the norm rather than an exception. Security is even more difficult to obtain than in fixed networks: some message encryption techniques can be used in order to avoid message spoofing.

The physical structure of a mobile ad-hoc network is completely different from the one of traditional fixed networks. In between the two types of network there is a large range of other network solutions which adopt aspects of both. We believe these heterogeneous networks, where fixed components interact with ad-hoc areas, and where different connectivity technologies are used, are going to be the networks of the future and that middleware should provide support for them. Much effort has recently been devolved towards middleware for mobile

networks, even if the term mobile network has often been used meaning different things. We will now introduce some of the most significant middleware, together with a categorisation of their peculiarities.

3 Middleware Systems: a Reference Model

Building distributed applications, either mobile or stationary, on top of the network layer is extremely tedious and error-prone. Application developers would have to deal explicitly with all the non-functional requirements listed in the previous section, such as heterogeneity and fault-tolerance, and this complicates considerably the development and maintenance of an application. However, given the novelties of mobile systems this approach has been adopted by many system developers: we will give more details about this in Section 5.

To support designers building distributed applications, middleware system layered between the network operating system and the distributed application is put into place. *Middleware* implements the Session and Presentation Layer of the ISO/OSI Reference Model (see Figure 5) [39]. Its main goal is to enable communication between distributed components. To do so, it provides application developers with a higher level of abstraction built using the primitives of the network operating system. Middleware also offers solutions to resource sharing and fault tolerance requirements.

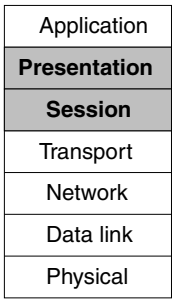

Fig. 5. The ISO/OSI Reference Model.

During the past years, middleware technologies for distributed systems have been built and successfully used in industry. For example, object-oriented technologies like OMG CORBA [53], Microsoft COM [58] and Sun Java/RMI [52], or message-oriented technologies like IBM MQSeries [55]. Although very successful in fixed environments, these systems might not to be suitable in a mobile setting, given the different requirements that they entail. We will discuss this issue in more details in Section 4, and show how traditional middleware has been

adapted for use in mobile setting in Section 5. Researchers have been and are actively working to design middleware targeted to the mobile setting, and many different solutions have been investigated recently. In order to classify, discuss and compare middleware developed to date, and understand their suitability in a mobile setting, we introduce the reference model depicted in Figure 6.

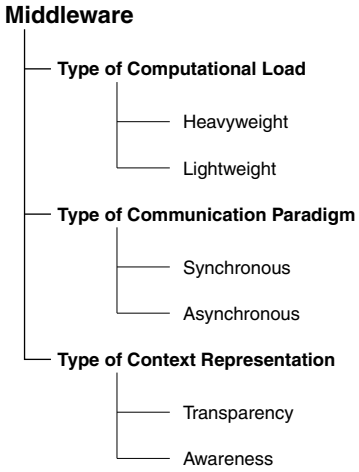

Fig. 6. Characterisation of middleware systems.

As shown in the picture, we distinguish middleware systems based on the *computational load* they require to execute, on the *communication paradigms* they support, and on the kind of *context representation* they provide to the applications.

Type of Computational Load. The computational load depends on the set of non-functional requirements met by the system. For instance, the main purpose of any middleware is to enable communication, e.g., allowing a user to request a remote service; what distinguishes different middleware systems, is the reliability with which these requests are handled. It is in fact much more expensive to guarantee that a request will be executed *exactly once*, instead of providing only *best-effort* reliability, that is, that the request may or may not be executed. As another example, we may consider replication. Replication is widely used by middleware systems in order to achieve fault tolerance and to improve scalability. Keeping the replicas synchronised with the master copy, however, requires a lot of effort and resources (e.g., network communication). Depending on how consistent the replicas are, the computational load of middleware varies accordingly. We use the term *heavy-weight* to denote a system that requires a large

amount of resources to deliver services to the above applications, as opposed to a *light-weight* one, which runs using a minimum set of resources (e.g., CPU, main memory, lines of code, etc.). Different computational loads imply different qualities of service and, depending on the characteristics of the distributed system we are addressing (e.g., amount of resources available, non-functional requirements, etc.), different middleware may suit better than others, as we will discuss in Section 3.1 and 3.2.

Type of Communication Paradigm. There are basically two types of communication paradigms a middleware system can support: *synchronous* or *asynchronous*. The former requires that both the client asking for a service, and the server exporting that particular service, are connected and executing at the same time, in order for the request to be successfully processed. The latter, instead, does not require the sender and the receiver of a request to be connected simultaneously. Other forms of communications exist that sit in between these two: *one-way* and *deferred synchronous*. One-way requests return control to the client, without awaiting for the completion of the operation requested from the server. This implies that the semantics of the client does not depend on the result of the requested operation. Deferred synchronous requests return control to the client immediately; unlike one-way requests, however, the client is in charge of re-synchronising with the server to collect the result, and this may cause the client to block if the answer is not yet ready.

Type of Context Representation. The last parameter that we identified to distinguish between different classes of middleware refers to the fact that either information about the execution context is fed to the above applications (i.e., *awareness*) or it is kept hidden inside the middleware itself (i.e., *transparency*). The middleware interacts with the underlying network operating system and collects information about the actual location of a device, value of network bandwidth, latency, remote services available, etc. By transparency, we mean that this context information is used privately by the middleware and not shown to the above applications; for example, the middleware may discover a congestion in a portion of the distributed system and therefore redirect requests to access data to a replica residing on another part of the distributed system, without informing the application about this decision. By awareness, instead, we mean that information about the execution context (or parts of it) is passed up to the running applications, that are now in charge of taking strategic decisions. It would, for example, be the application layer to choose which replica to contact in the non-congestioned portion of the network. Due to the complexity introduced by the awareness approach, middleware for distributed systems usually chose transparent strategies. This is justifiable as the lack in behavioural optimisation with application awareness is compensated by the abundance of resources.

Now that we have discussed a model to characterise distribute systems (Section 2) and a model to characterise middleware for distributed systems (Section 3), we need to understand the relationships between the two models and

determine the characteristics that middleware for mobile distributed systems should satisfy and how they differ from the ones required in fixed distributed systems.

3.1 Middleware for Fixed Distributed Systems

With respect to the conceptual model presented above, middleware for fixed distributed systems can be mainly described as resource-consuming systems that hide most of the details of distribution from application designers. With the exception of message-oriented middleware (Section 4.2), they mainly support synchronous communication between components as the basic interaction paradigm. We now analyse in more details the relationship between the physical structure of fixed distributed systems and the characteristics of associated middleware.

Fixed Devices → Heavy Computational Load. As discussed in Section 2, wired distributed systems consist of resource-rich fixed devices. When building distributed applications on top of this infrastructure, it is worthwhile exploiting all the resources available (e.g., fast processors, large amounts of memory, etc.) in order to deliver the best quality of service to the application. The higher the quality of service, the heavier the middleware running underneath the application. This is due to the set of non-functional requirements that the middleware achieves, like fault tolerance, security or resource sharing.

Permanent Connection → Synchronous Communication. Fixed distributed systems are often permanently connected to the network through high-bandwidth and stable links. This means that the sender of a request and its receiver (i.e., the component asking for a service and the component delivering that service) are usually connected at the same time. Permanent connection allows therefore a synchronous form of communication, as the situations when client and server are not connected at the same time are considered only exceptions due to failures of the system (e.g., disconnection due to network overload).

Asynchronous communication mechanisms are however also provided by message oriented middleware (MOM) and recently also by the CORBA specification. Asynchronous communication is used also in fixed networks, however the bulk of middleware applications have been developed using synchronous communication.

Static Context → Transparency. The execution context of a fixed distributed system is generally static: the location of a device seldom changes, the topology of the system is preserved over time, bandwidth remains stable, etc. The abundance of resources allows the disregard of application specific behaviours in favour of a transparent and still efficient approach. For example, to achieve fault tolerance, the middleware can transparently decide on which hosts to create replicas of data and where to redirect requests to access that data in case a network failure inhibits direct access to the master copy, in a

completely transparent manner. Hiding context information inside the middleware eases the burden of application programmers that do not have to deal with the achievement of non-functional requirements (e.g., fault tolerance) explicitly, concentrating, instead, on the real problems of the application they are building.

3.2 Middleware for Ad-hoc and Nomadic Distributed Systems

Nomadic and ad-hoc mobile systems differ in some aspects, however they present a set of similar characteristics that influence the way middleware should behave. We now justify a set of choices which are generally made by the middleware that we will describe.

Mobile Devices → Light Computational Load. Mobile applications run on resource-scarce devices, with little memory, slow CPU, and generally limited battery power. Due to these resource limitations, heavy-weight middleware systems optimised for powerful machines do not seem to suit mobile scenarios. Therefore, the right trade-off between computational load and non-functional requirements achieved by the middleware needs to be established. An example of this might be to relax the assumption of keeping replicas always synchronised, and allow the existence of diverging replicas that will eventually reconcile, in favour of a lighter-weight middleware. We will see examples of this in some of the middleware described in Section 5.

Intermittent Connection → Asynchronous Communication. Mobile devices connect to the network opportunistically for short periods of time, mainly to access some data or to request a service. Even during these periods, the available bandwidth is, by orders of magnitude, lower than in fixed distributed systems, and it may also suddenly drop to zero in areas with no network coverage. It is often the case that the client asking for a service, and the server delivering that service, are not connected at the same time. In order to allow interaction between components that are not executing along the same time line, an asynchronous form of communication is necessary. For example, it might be possible for a client to ask for a service, disconnect from the network, and collect the result of the request at some point later when able to reconnect.

Dynamic Context → Awareness. Unlike fixed distributed systems, mobile systems execute in an extremely dynamic context. Bandwidth may not be stable, services that are available now may not be there a second later, because, for example, while moving the hand-held device loses connection with the service provider. The high variability (together with the constrained resources) influences the way middleware decides and chooses. The optimisation of the application and middleware behaviour using application and context aware techniques becomes then more important, also given the limited resources.

4 Middleware for Fixed Distributed Systems

Middleware technologies for fixed distributed systems can be classified into three main categories[1]: object-oriented middleware, message-oriented middleware and transaction-oriented middleware. We briefly review the main characteristics of these technologies and assess the feasibility of their application in mobile settings.

4.1 Object-Oriented and Component Middleware

Object-oriented middleware supports communication between distributed objects, that is, a client object requests the execution of an operation from a server object that may reside on another host. This class of middleware systems evolved from Remote Procedure Calls [69] (RPCs): the basic form of interaction is still synchronous, that means the client object issuing a request is blocked until the server object has returned the response. Products in this category include implementations of OMG CORBA [53], like IONA'S Orbix [7] and Borland's VisiBroker [46], the CORBA Component Model (CCM) [48], Microsoft COM [58], Java/RMI [52] and Enterprise JavaBeans [43]. Despite the great success of these technologies in building fixed distributed systems, their applicability to a mobile setting is rather restricted because of the heavy computational load required to run these systems, the mainly synchronous form of object requests supported, and the principle of transparency that has driven their design and that prevents any forms of awareness. The most recent CORBA specification allows for asynchronous communication, however at the time of writing no implementation of it exists. The use of these systems in mobile context has been investigated and is reported in Section 5.

4.2 Message-Oriented Middleware

Message-oriented middleware supports the communication between distributed components via message-passing: client components send a message containing the request for a service execution and its parameters to a server component across the network and the server may respond with a reply message containing the result of the service execution. Message-oriented middleware supports asynchronous communication in a very natural way, achieving de-coupling of client and server, as requested by mobile systems: the client is able to continue processing as soon as the middleware has accepted the message; eventually the server will send a reply message and the client will be able to collect it at a convenient time. However, 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 processed. Sun's Java Message Queue [44] and IBM's MQSeries [55] are examples of very successful message-oriented middleware for traditional distributed systems. We believe there is scope for use of

[1] Other categories can be identified, like middleware for scientific computing, but we omit their discussion here because they are only loosely related to the mobile setting.

these middleware in mobile settings, and we will discuss how some adaptation of JMS has recently been ported to mobile.

4.3 Transaction-Oriented Middleware

Transaction-oriented middleware systems are mainly used in architectures where components are database applications. They support transactions involving components that run on distributed hosts: a client component clusters several operations within a transaction that the middleware then transports via the network to the server components in a manner that is transparent to both clients and servers. These middleware support both synchronous and asynchronous communication across heterogeneous hosts and achieve high reliability: as long as the participating servers implement the two-phase-commit protocol, the atomicity property of transactions is guaranteed. However, this causes an undue overhead if there is no need to use transactions. Despite their success in fixed systems, the computational load and the transparency that are typical of this kind of middleware (such as IBM CICS [31] and BEA's Tuxedo [28]), make them look not very suitable for mobile settings.

The classification of middleware for fixed distributed systems in the three categories (i.e., object-oriented, message-oriented and transaction-oriented) discussed above is actually not rigid. There is in fact a trend of merging these categories together, as shown by the CORBA Object Transaction Service, a convergence of object-oriented and transaction-oriented middleware, or by the CORBA Event Service and the publish/subscribe communication à la CCM, a union of object-oriented and message-oriented middleware. We will now assess the impact of traditional middleware in mobile settings and also describe a set of middleware developed specifically for mobile.

5 Middleware for Mobile Distributed Systems

There are different examples of use of traditional middleware systems in the context of mobile computing. We will show some examples of adaptation of object-oriented middleware and message oriented middleware to small and mobile devices. The main problem with the object-oriented approach is that it relies on synchronous communication primitives that do not necessarily suit all the possible mobile system architectures. The computational load of these systems is quite high and the principle of transparency they adhere to does not always fit mobile applications.

As we have seen, the requirements for mobile applications are considerably different from the requirements imposed by fixed distributed applications. Some of the developed systems for mobile environments adopted the radical approach of not having a middleware but rather rely on the application to handle all the services and deal with the non-functional requirements, often using a context-aware approach that allows adaptation to changing context [15]. Sun provides J2ME (Java Micro Edition) [70] which is a basic JVM and development package

targeting mobile devices. Microsoft recently matched this with .Net Compact Framework [42], which also has support for XML data and web services connectivity.

However this approach is a non-solution, as it completely relies on application designers for the solution of most of the non-functional requirements middleware should provide, starting from scalability.

On the other hand, recently, some middleware specifically targeting the needs of mobile computing have been devised [59]; assumptions such as scarce resources, and fluctuating connectivity have been made in order to reach light-weight solutions. Some of the approaches however target only one of the mobility aspects: for instance, many location-aware systems have been implemented to allow application to use location information to provide services.

We now describe some of the developed solutions to date, starting from the examples of adaptation of object-oriented middleware such as CORBA or Java Messaging Server to mobile platforms, to solutions targeting completely ad-hoc scenarios.

5.1 Traditional Middleware applied in Mobile Computing

Object-oriented middleware has been adapted to mobile settings, mainly to make mobile devices inter-operated with existing fixed networks (i.e., nomadic setting). The main challenge in this direction is in terms of software size and protocol suitability, as already mentioned. IIOP (i.e., the Internet Inter-ORB Protocol) is the essential part of CORBA that is needed to allow communication among devices. IIOP has been successfully ported to mobile setting and used as a minimal ORB for mobile devices [27]. IIOP defines the minimum protocol necessary to transfer invocations between ORBs. In ALICE [27] hand-helds with Windows CE and GSM adaptors have been used to provide support for client-server architectures in nomadic environments. An adaptation of IIOP specifically for mobile (i.e., LW-IOP, Light-weight Inter-Orb Protocol) has been devised in the DOLMEN project [57]: caching of unsent data combined with an acknowledgement scheme to face wireless medium unreliability. Also actual names of machines are translated dynamically through a name server, which maintains up-to-date information of the hosts location.

In [56] CORBA and IIOP are used together with the WAP (Wireless Access Protocol) stack [24] in order to allow the use of CORBA services on a fixed network through mobile devices connected through WAP and a gateway. IIOP is used to achieve message exchange.

In general, the synchronous connectivity paradigm introduced by traditional middleware assumes a permanent connectivity that cannot be given as granted in most of the mobile computing scenarios. The above mentioned systems are usually targeted to nomadic settings where hand-offs allow mobile devices to roam while being connected. Some minimal support for disconnection is introduced.

There have been more serious attempts in the direction of using traditional middleware using a sort of semi-asynchronous paradigm. Some involved RPC based middleware enhanced with queueing delaying or buffering capabilities in

order to cope with intermittent disconnections. Example of these behaviours are Rover [32] or Mobile DCE [65]. As we write, an implementation of the message oriented middleware JMS (Java Messaging Server) has been released [68]. It supports point to point and publish/subscribe models, that is a device can either communicate with a single otehr (through its queue), or register on a topic an be notified of all the messages sent to that topic. We believe this is a good answer to the need for adaptation of traditional middleware to mobile and that the use of publish/subscribe and message oriented systems will be taken further as they offer an asynchronous communication mechanism that allows for disconnected operations. However, communication is not the only aspect that mobile computing middleware should tackle: other important aspects such as context awareness and data sharing need to be addressed.

In the existing examples of use of traditional middleware on mobile, the focus is on provision of services from a back-bone network to a set of mobile devices: the main concerns in this scenarios are connectivity and message exchange. In case of a less structured network or in case services must be provided by mobile devices, traditional middleware paradigms seems to be less suitable and a new set of strategies needs to be used.

The importance of monitoring the condition of the environment, and adaptation to application needs, maybe through communication of context information to the upper layers, becomes vital to achieve reasonable quality of service.

Given the highly dynamic environment and the scarce resources, quality of service provision presents higher challenges in mobile computing. Nevertheless, researchers have devised a number of interesting approaches to quality of service provision to mobile devices [15]. Most of the time the devices are considered terminal nodes and the clients of the service provision, and the network connectivity is assumed fluctuating but almost continuous (like in GSM settings). The probably most significant example of quality of service oriented middleware is Mobiware [2], which uses CORBA, IIOP and Java to allow service quality adaptation in mobile setting. As shown in Figure 7, in Mobiware mobile devices are seen as terminal nodes of the network and the main operations and services are developed on a core programmable network of routers and switches. Mobile devices are connected to access points and can roam from an access point to another.

The main idea in Mobiware is that mobile devices will have to probe and adapt to the constantly changing resources over the wireless link. The experimental network used by Mobiware is composed of ATM switches, wireless access points, and broadband cellular or ad-hoc connected mobile devices. The toolkit focuses on the delivery of multimedia application to devices with adaptation to the different quality of service and seamless mobility.

Mobiware mostly assumes a service provision scenario where mobile devices are roaming but permanently connected, with fluctuating bandwidth. Even in the case of the ad-hoc broadband link, the device is supposed to receive the service provision from the core network through, first the cellular links and then some ad-hoc hops.

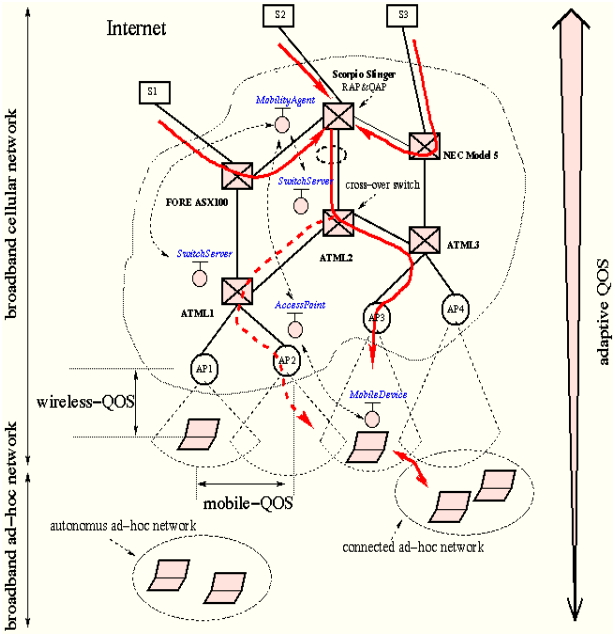

Fig. 7. Mobiware Architecture (from [2]).

In more extreme scenarios, where links are all ad-hoc, these assumptions cannot be made and different middleware technologies need to be applied. One of the strength of Mobiware is the adaptation component to customise quality of service results. It is more and more clear that middleware for mobile devices should not ignore context and that adaptation is a key point, given the limited resources and changing conditions.

Another interesting example of quality of service oriented middleware is L2imbo [17], a tuple space based quality of service aware system. For a description of L2imbo, we refer the reader to Section 5.4, where we describe the advantages of tuple space based models.

5.2 Context-Awareness based Middleware

To enable applications to adapt to heterogeneity of hosts and networks as well as variations in the user's environment, systems must provide for mobile applications to be aware of the context in which they are being used. Furthermore, context information can be used to optimise application behaviour counter balancing the scarce resource availability.

User's context includes, but is not limited to:

- location, with varying accuracy depending on the positioning system used;
- relative location, such as proximity to printers and databases;

- device characteristics, such as processing power and input devices;
- physical environment, such as noise level and bandwidth;
- user's activity, such as driving a car or sitting in a lecture theatre.

Context-aware computing is not a new computing paradigm; since it was first proposed a decade ago [64], many researchers have studied and developed systems that collect context information, and adapt to changes.

The principle of *Reflection* has often been used to allow dynamic reconfiguration of middleware and has proven useful to offer context-awareness. The concept of reflection was first introduced by Smith in 1982 [67] as a principle that allows a program to access, reason about and alter its own interpretation. Initially, the use of reflection was restricted to the field of programming language design [33]; some years later, reflection has been applied to the field of operating systems [78] and, more recently, distributed systems [41].

Examples of traditional middleware built around the principle of reflection include, but are not limited to, OpenORB [22], OpenCorba [36], dynamicTAO [34], Blair et al. work [9]. The role of reflection in distributed systems has to do with the introduction of more openness and flexibility into middleware platforms. In standard middleware, the complexity introduced through distribution is handled by means of abstraction. Implementations details are hidden from both users and application designers and encapsulated inside the middleware itself. Although having proved to be successful in building traditional distributed systems, this approach suffers from severe limitations when applied to the mobile setting. Hiding implementation details means that all the complexity is managed internally by the middleware layer; middleware is in charge of taking decisions on behalf of the application, without letting the application influence this choice. This may lead to computationally heavy middleware systems, characterised by large amounts of code and data they use in order to transparently deal with any kind of problems and find the solution that guarantees the best quality of service to the application. Heavyweight systems cannot however run efficiently on a mobile device as it cannot afford such a computational load. Moreover, in a mobile setting it is neither always possible, nor desirable, to hide all the implementation details from the user. The fundamental problem is that by hiding implementation details the middleware has to take decisions on behalf of the application; the application may, however, have vital information that could lead to more efficient or suitable decisions. Both these limitations can be overcome by reflection. A reflective system may bring modifications to itself by means of inspection and/or adaptation. Through inspection, the internal behaviour of a system is exposed, so that it becomes straightforward to insert additional behaviour to monitor the middleware implementation. Through adaptation, the internal behaviour of a system can be dynamically changed, by modification of existing features or by adding new ones. This means that a middleware core with only a minimal set of functionalities can be installed on a mobile device, and then it is the application which is in charge of monitoring and adapting the behaviour of the middleware according to its own needs.

The possibilities opened by this approach are remarkable: light-weight middleware can be built that support context awareness. Context information can be kept by middleware in its internal data structures and, through reflective mechanisms, applications can acquire information about their execution context and tune the middleware behaviour accordingly. No specific communication paradigm is related to the principle of reflection, so this issue is left unspecified and depends on the specific middleware system built.

Some recent approaches have investigated the use of reflection in the context of mobile systems, and used it to offer dynamic context-awareness and adaptation mechanisms [60]. UIC (Universally Interoperable Core) [74] is a minimal reflective middleware that targets mobile devices. UIC is composed of a pluggable set of components that allow developers to specialise the middleware targeting different devices and environments, thus solving heterogeneity issues. The configuration can also be automatically updated both at compile and run time. Personalities can be defined to have a client-side, server-side or both behaviours. Personalities can also define with which server type to interact (i.e., Corba or Java RMI) as depicted in Figure 8 : single personalities allow the interaction with only one type while multi personalities allow interaction with more than one type. In the case of multi personalities the middleware dynamically chooses the right interaction paradigm. The size of the core goes, for instance, from 16KB for a client-side CORBA personality running on a Palm OS device to 37KB for a client/server CORBA personality running on a Windows CE device.

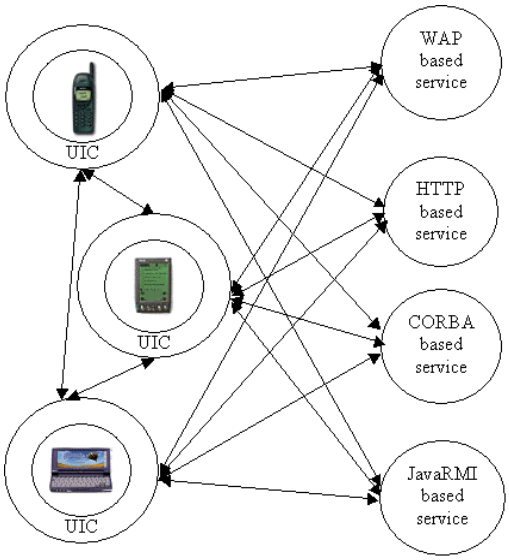

Fig. 8. The UIC Interaction Paradigm (from [60]).

On top of a framework very similar to UIC, Gaia [14] has been developed adding support for dynamic adaptation to context conditions. Gaia defines active spaces where services, users, data and locations are represented and manipulated dynamically and in coordination. Gaia defines a set of further services such as event distribution, discovery, security and data storage. Some other approaches in this direction have been developed focusing on meta-data representation for services and application depended policies. Mechanisms for dynamic adaptation and conflict resolutions have also been put in place [35].

In [76] another approach to context-awareness in mobile computing is presented. The paper presents a middleware for event notification to mobile computing applications. Different event channels allow differentiation of notification of different context and environmental variables. Figure 9 shows the architecture of the system. The asynchronous event channel suits the mobile computing setting allowing for disconnection; however the model the authors assume is based on permanent but fluctuating connectivity. Applications can register for notification of specific events depending on the task they have to perform. Applications relying on this middleware are constructed decoupling the application functionalities from the application context-awareness, where policies are defined to adapt to context.

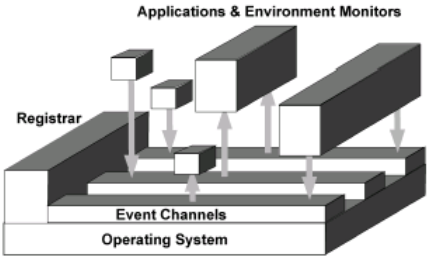

Fig. 9. The event notification architecture (from [76]).

The main idea of these systems is to offer the ability to change the behaviour of the middleware and application based on the knowledge on the changing context. This seems to be a valid idea given the scarce resources and the dynamicity of the mobile environment.

Much research has recently been investigating context-awareness issues. Another example of context-aware systems is Odyssey [62], a data-sharing middleware with synchronisation policies depending on applications. We will describe this middleware together with other data-sharing oriented middleware in Section 5.3.

Location-aware Middleware. Location is one of the most studied aspects of context awareness. Location awareness has attracted a great deal of attention and many examples exist of applications that exploit location information to:

offer travellers directional guidance, such as the Shopping Assistant [6] and CyberGuide [37]; to find out neighbouring devices and the services they provide, such as Teleporting [8]; to send advertisements depending on user's location, such as People and Object Pager [12]; to send messages to anyone in a specific area, such as Conference Assistant [19]; and so on. Most of these systems interact directly with the underlying network OS to extract location information, process it, and present it in a convenient format to the user. One of their major limitations concerns the fact that they do not cope with heterogeneity of coordinate information, and therefore different versions have to be released that are able to interact with specific sensor technologies, such as the Global Positioning System (GPS) outdoors, and infrared and radio frequency indoors.

To enhance the development of location-based services and applications, and reduce their development cycle, middleware systems have been built that integrate different positioning technologies by providing a common interface to the different positioning systems. Examples include Oracle iASWE [49], Nexus [25], Alternis [1], SignalSoft [66], CellPoint [13], and many others are being released.

We describe briefly *Nexus* [25], an example of middleware that supports location-aware applications with mobile users. The idea that has motivated the development of this system is that no migration to an homogeneous communication environment is possible, and therefore an infrastructure that supports a heterogeneous communication environment is necessary. Nexus aims to provide this infrastructure.

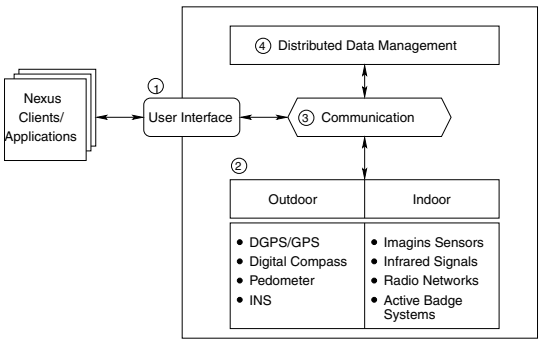

Fig. 10. Nexus Architecture.

The architecture of the Nexus infrastructure is depicted in Figure 10. As the picture shows, there are four different components working together.

The User Interface component is running on the mobile device carried by the user and contains basic functionality, which is required by Nexus Clients to interact with the Nexus platform, and to display and navigate through the

model. It also provides support for adapting to devices with different levels of computing power, different amounts of memory, different levels of network connection or different displays.

The interior of a Nexus platform is then split into three main elements: communication, distributed data management and sensors.

Sensors: Nexus applications run both in outdoor and indoor areas. It would be difficult to use only one sensor for positioning in both environments (e.g., GPS can be used outdoor but not indoor, as its satellite signals are blocked by buildings). Therefore, a multi-sensor tool is needed, based on different positioning systems.

Communication: To access information, mobile devices need to be able to connect to the information source, e.g. the Internet, using wireless communication. For a wide area network, data services of mobile telephone systems, such as GSM or UMTS, can be used. Inside a building, wireless LAN, such as Bluetooth, can be used instead. The Nexus communication layer acts as a broker to bridge the differences between existing heterogeneous networks.

Distributed data management: according to the demands of different location aware applications, spatial data have to be offered in multiple representations. Hence, appropriate algorithms to deduce all the necessary levels of detail have to be implemented into the platform. In order to guarantee the interoperability, relationships between different models must be defined and data formats must be exchangeable. All these different aspects concerning the management of data within Nexus are managed by the Distributed data Management component.

5.3 Data sharing-oriented Middleware

One of the major issues targeted is the *support for disconnected operations* and data-sharing. Systems like Coda [63], its successor Odyssey [62], Bayou [18, 73] and Xmiddle [40] try to maximise availability of data, giving users access to replicas; they differ in the way they ensure that replicas move towards eventual consistency, that is, in the mechanisms they provide to detect and resolve conflicts that naturally arise in mobile systems. Despite a proliferation of different, proprietary data synchronisation protocols for mobile devices, we still lack a single synchronisation standard, as most of these protocols are implemented only on a subset of devices and are able to access a small set of networked data. This represents a limitation for both end users, application developers, service providers and device manufacturers.

Coda

Coda [63] is a file system for large-scale distributed computing environments. It provides resilience to server and network failures through two distinct but complementary mechanisms: server replication and disconnected operation. The first mechanism involves storing copies of a file at multiple servers; the second one is a mode of execution in which a caching site temporarily assumes the role of a replication site; this becomes particularly useful for supporting portable computers.

Coda makes a distinction between relatively few servers, which are physically secure, run trusted software and are monitored by operational staff; and clients, which are far more numerous, may be modified in arbitrary ways by users, are physically dispersed, and may be turned off for long periods of time. The Coda middleware targets therefore fixed or, at most, nomadic systems, as it relies on a fixed core infrastructure.

The unit of replication in Coda is a volume, that is, a collection of files that are stored on one server and form a partial subtree of the shared file system. The set of servers that contain replicas of a volume is its volume storage group (VSG). Disconnections are treated as rare events: when disconnected, the client can access only the data that was previously cached at the client site; upon reconnection modified files and directories from disconnected volumes are propagated to the VSG. Coda clients view disconnected operations as a temporary state and revert to normal operation at the earliest opportunity; these transitions are normally transparent to users. Disconnected operation can also be entered voluntarily, when a client deliberately disconnects from the network; however, clients have no way to influence the portion of the file system that will be replicated locally. Moreover, it is the system that bears the responsibilities of propagating modifications and detecting update conflicts when connectivity is restored. Venus (the cache manager) has control over replication of volumes in use. Venus can be in three states: *Hoarding*, when the client is connected to the network, in this case Venus takes care of caching the information; *Emulating* when disconnected, using the cache and throwing exceptions when the needed data is not in the cache; *Integrating*, upon reconnection, when the data modified need to be reconciled with the server copy. Venus attempts to solve conflicts during reconciliation in an application transparent way; however application specific resolvers (ASR) may be specified.

Odyssey

The mostly application transparent approach adopted by Coda has been improved introducing context-awareness and application-dependent behaviours in Odyssey [62], and allowing the use of these approaches in mobile computing settings. Odyssey, again, assumes that applications reside on mobile clients but access or update data stored on remote, more capable and trustworthy servers; once again the nomadic scenario is targeted.

Odyssey proposes a collaborative model of adaptation. The operating system, as the arbiter of shared resources, is in the best position to determine resource availability; however, the application is the only entity that can properly adapt to given context conditions, and must be allowed to specify adaptation policies. This collaborative model is called application-aware adaptation and it is provided using the architecture depicted in Figure 11.

To allow reaction to changes, applications first need to register an interest in particular resources. For every resource, they define the acceptable upper and lower bounds on the availability of that resource and they register an up-call procedure that must be invoked whenever the availability of the resource falls

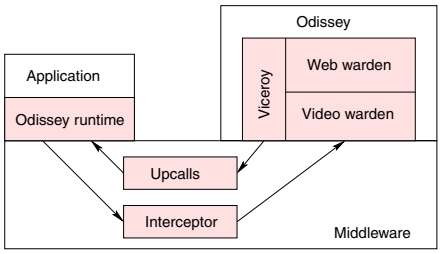

Fig. 11. Odyssey Client Architecture.

outside the window of acceptance. The Viceroy component is then responsible for monitoring resource usage on the client side and notifying applications of significant changes, using the registered up-calls. When an application is notified of a change in resource availability, it must adapt its access. Wardens are the components responsible for implementing the access methods on objects of their type: the interceptor module redirects file systems operations to corresponding wardens, which provide customised behaviour (e.g., different replication policies) according to type-specific knowledge.

Although better suited to the mobile environment than its predecessor Coda, Odyssey suffers from some limitations: the data that can be moved across mobile devices (i.e., a collection of files) may be too coarse-grained in a mobile setting, where devices have limited amount of memory and connection is often expensive and of low quality. Moreover, the existence of a core of more capable and trustworthy servers does not fit the ad-hoc scenario. Finally, files are uninterpreted byte streams; this lack of semantics complicates the development of conflict detection and reconciliation policies from an application point of view.

Bayou

The Bayou storage system [18, 73] provides an infrastructure for collaborative applications. Bayou manages conflicts introduced by concurrent activity while relying only on the fluctuating connectivity available in mobile computing. Replication is seen as a requirement in the mobile scenario as a single storage site may not be reachable by some mobile clients or within disconnected work-groups. Bayou allows arbitrary read and write operations to any replica without explicit coordination with the other replicas: every computer eventually receives updates from every other, either directly or indirectly, through a chain of peer interactions. The weak consistency of the replicated data is not transparent to applications; instead, they are aware they may be using weakly consistent data and that their write operations may conflict with those of other users and applications. Moreover, applications are involved in the detection and resolution of conflicts since these naturally depend on the semantics of the application. In particular, the system provides the application with ways of specifying its own

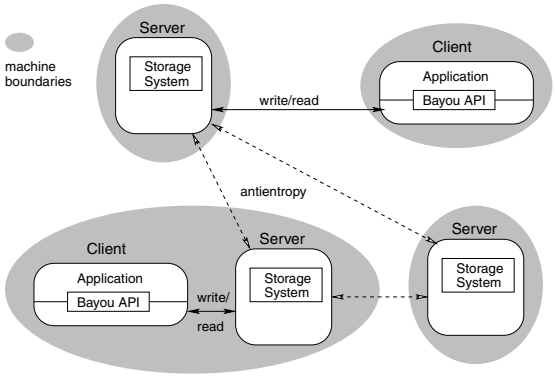

Fig. 12. Bayou System Model.

notion of a conflict, along with its policy for resolving it. In return, the system implements the mechanisms for reliable detection of conflicts, as specified by the application, and for automatic resolution when possible.

Automatic detection is achieved through a mechanism called *dependency check*: every write operation is accompanied by a dependency set that consists of a query and its expected result. A conflict is detected if the query, when run at a server against its current copy of the data, does not return the expected result. This dependency check is therefore a pre-condition for performing the update. As an example consider someone trying to add an appointment in an agenda, without knowing the content of the agenda. A dependency check could make sure that the time for the appointment is free before adding it. If the check fails, the requested update is not performed and the server invokes a procedure to resolve the detected conflict. Once a conflict has been detected, a *merge procedure* is run by the Bayou server in an attempt to resolve it. Merge procedures are provided by application programmers in the form of templates that are then filled in with the details of each write and accompany each write operation. Users do not have to know about them, except when automatic conflict resolution cannot be done and manual resolution is needed. In this, Bayou is more flexible than Odyssey as customisation can be performed on each write operation, and not by type.

Bayou guarantees that all the servers will move towards eventual consistency. This means that all the servers will eventually receive all the write operations (through a process called anti-entropy), although the system cannot enforce bounds on write propagation delays since these depend on network connectivity factors that are outside Bayou's control. Eventual consistency is guaranteed by two factors: writes are performed in the same well-defined order on all the servers, and detection and resolution procedures are deterministic so that servers resolve the same conflict in the same way.

Unlike previous systems like Coda, that promote transparency of conflict detection and resolution, Bayou exploits application knowledge for dependency checks and merge procedures. Moreover, while Coda locks complete file volumes

when conflicts have been detected but not yet resolved, Bayou allows replicas to always remain accessible. This permits clients to continue to read previously written data and to issue new writes, but it may lead to cascading conflict resolution if the newly issued operations depend on data that are in conflict.

One of the major drawbacks of Bayou is its client-sever architecture. Although in principle client and server may co-exist on a host (see Figure 12), in practise the system requires that each data collection is replicated in full on a number of servers. This is, of course, unaffordable for hand-held devices that can therefore only play the role of clients in this architecture. Bayou is therefore most suited for nomadic rather than ad-hoc applications.

Xmiddle

Xmiddle allows mobile hosts to share data when they are connected, or replicate the data and perform operations on them off-line when they are disconnected; reconciliation of data takes place once the hosts reconnect.

Unlike tuple-space based systems (as we will see in Section 5.4), which store data in flat unstructured structures, Xmiddle allows each device to store its data in a tree structure. Trees allow sophisticated manipulations due to the different node levels, hierarchy among the nodes, and the relationships among the different elements which could be defined.

When hosts get in touch with each other, they need to be able to interact. Xmiddle allows communication through sharing of trees. On each device, a set of possible access points for the private tree is defined; they essentially address branches of the tree that can be modified and read by peers. The size of these branches can vary from a single node to a complete tree; unlike systems such as Coda and Odyssey (Section 5.3), where entire collections of files have to be replicated, the unit of replication can be easily tuned to accommodate different needs. For example, replication of a full tree can be performed on a laptop, but only of a small branch on a PDA, as the memory capabilities of these devices differ greatly.

In order to share data, a host needs to explicitly *link* to another host's tree. The concept of linking to a tree is similar to the mounting of network file systems in distributed operating systems to access and update information on a remote disk.

Figure 13 shows the general structure of Xmiddle and the way hosts get in touch and interact. As long as two hosts are connected, they can share and modify the information on each other's linked data trees. When disconnections occurs, both explicit (e.g., to save battery power or to perform changes in isolation from other hosts) and implicit (e.g., due to movement of a host into an out of reach area), the disconnected hosts retain replicas of the trees they were sharing while connected, and continue to be able to access and modify the data.

When the two hosts reconnect, the two different, possibly conflicting, replicas need to be reconciled. Xmiddle exploits the tree differencing techniques developed in [72] to detect differences between the replicas which hosts use to concurrently and off-line modify the shared data. However, it may happen that the

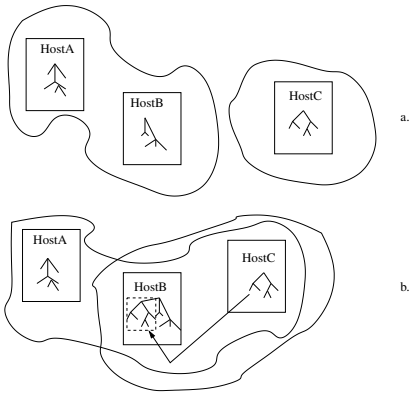

Fig. 13. a. Host $\mathtt{H}_B$ and Host $\mathtt{H}_C$ are not connected. b. Host $\mathtt{H}_B$ and Host $\mathtt{H}_C$ connect and Host $\mathtt{H}_B$ receives a copy of the tree that it has linked from Host $\mathtt{H}_C$.

reconciliation task cannot be completed by the Xmiddle layer alone, because, for example, different updates have been performed on the same node of the tree. In order to solve these conflicts, Xmiddle enables the mobile application engineer to associate application-specific conflict resolution policies to each node of the tree. Whenever a conflict is detected, the reconciliation process finds out which policy the application wants the middleware to apply, in order to successfully complete the merging procedure.

Xmiddle implements the tree data structure using the eXtended Markup Language (XML) [11] and related technologies. In particular, application data are stored in XML documents, which can be semantically associated to trees. Related technologies, such as the Document Object Model (DOM) [4], XPath [16] and XLink [38], are then exploited to manipulate nodes, address branches, and manage references between different parts of an XML document. Reconciliation policies are specified as part of the XML Schema [23] definition of the data structures that are handled by Xmiddle itself.

Xmiddle moves a step forward other middleware systems which focus on the problem of disconnected operations. In particular, unlike Coda and Odyssey, the unit of replication can be easily tuned, in order to accommodate different application and device needs. This issue may play a key role in mobile scenarios, where devices have limited amount of memory and the quality of the network connection is often poor and/or expensive. Moreover, Xmiddle addresses pure ad-hoc networks and not only nomadic ones, as no assumption is made about the existence of more powerful and trusted hosts which should play the role of servers and on which a collection of data should be replicated in full.

Although representing a good starting point for developing middleware for mobile computing, at present Xmiddle suffers from some limitations that require further investigation: the communication paradigm (i.e., sharing of trees) pro-

vided is too poor and needs to be improved in order to model more complex interactions that can occur in mobile settings.

5.4 Tuple Space-based Middleware

The characteristics of wireless communication media (e.g., low and variable bandwidth, frequent disconnections, etc.) favour a decoupled and opportunistic style of communication: decoupled in the sense that computation proceeds even in presence of disconnections, and opportunistic as it exploits connectivity whenever it becomes available. The synchronous communication paradigm supported by many traditional distributed systems has to be replaced by a new asynchronous communication style.

As we have seen, some attempts based on events [76], or queues (Rover [32] or Mobile JMS [68]) have been devised. However, a completely asynchronous and decoupled paradigm (tuple space based) have also been isolated as effective in mobile settings. Although not initially designed for this purpose (their origins go back to Linda [26], a coordination language for concurrent programming), tuple space systems have been shown to provide many useful facilities for communication in wireless settings. In Linda, a tuple space is a globally shared, associatively addressed memory space used by processes to communicate. It acts as a repository (in particular a multi-set) of data structures called tuples that can be seen as vector of typed values. Tuples constitute the basic elements of a tuple space systems; they are created by a process and placed in the tuple space using a `write` primitive, and they can be accessed concurrently by several processes using `read` and `take` primitives, both of which are blocking (even if non-blocking versions can be provided). Tuples are anonymous, thus their selection takes place through pattern matching on the tuple contents. Communications is de-coupled in both time and space: senders and receivers do not need to be available at the same time, because tuples have their own life span, independent of the process that generated them, and mutual knowledge of their location is not necessary for data exchange, as the tuple space looks like a globally shared data space, regardless of machine or platform boundaries.

These forms of decoupling assume enormous importance in a mobile setting, where the parties involved in communication change dynamically due to their migration or connectivity patterns. However, a traditional tuple space implementation is not enough. There are basic questions that need to be answered: how is the globally shared data space presented to mobile hosts? How is it made persistent? The solutions developed to date basically differ depending on the answers they give to the above questions.

We now review three tuple-space middleware that have been devised for mobile computing applications: Lime [45], TSpaces [77] and L2imbo [17].

Lime

In Lime [45], the shift from a fixed context to a dynamically changing one is accomplished by breaking up the Linda tuple space into many tuple spaces, each

permanently associated to a mobile unit, and by introducing rules for transient sharing of the individual tuple spaces based on connectivity.

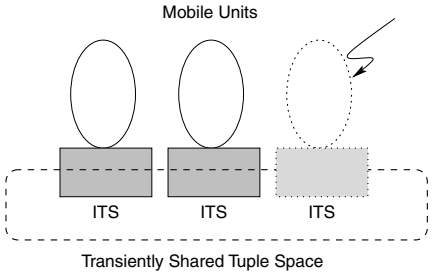

Fig. 14. Transiently shared tuple spaces in Lime.

As shown in Figure 14, each mobile unit has access to an *interface tuple space* (ITS) that is permanently and exclusively attached to that unit and transferred along with it when movement occurs (like in the data tree of Xmiddle). Each ITS contains tuples that the unit wishes to share with others and it represents the only context accessible to the unit when it is alone. Access to the ITS takes place using conventional Linda primitives, whose semantics is basically unaffected. However, the content of the ITS (i.e., the set of tuples that can be accessed through the ITS) is dynamically recomputed in such a way that it looks like the result of the merging of the ITSs of other mobile units currently connected. Upon arrival of a new mobile unit, the content perceived by each mobile unit through its ITS is recomputed taking the content of the new mobile unit into account. This operation is called engagement of tuple spaces; the opposite operation, performed on departure of a mobile unit, is called disengagement. The tuple space that can be accessed through the ITS of a mobile unit is therefore shared by construction and transient because its content changes according to the movement of mobile units.

The term mobile unit can be understood either as mobile agent or as mobile host. In the first case, the context is logical mobility, in the second one, physical mobility. The Lime notion of transiently shared tuple space is applicable to a generic mobile unit, regardless of its nature, as long as a notion of connectivity ruling engagement and disengagement is properly defined.

Lime fosters a style of coordination that reduces the details of mobility and distribution to changes to what is perceived as the local tuple space. This powerful view simplifies application design in many scenarios, relieving the designer from explicitly maintaining a view of the context consistent with changes in the configuration of the system. However, this may be too restrictive in domains

where higher degrees of context awareness are needed, for example to control the portion of context that has to be accessed.

Lime tries to cope with this problem, first by extending Linda operations with tuple location parameters that allow to operate on different projections of the transiently shared tuple space. Secondly, information about the system configuration is made available through a read-only transiently shared tuple space called LimeSystem, containing details about the mobile components present in the community and their relationship; finally, reactions can be set on the tuple space, to enable actions to be taken in response to a change in the configuration of the system.

An important aspect of Lime is tuple access and movement; events are used to notify users when a new tuple is available.

TSpaces

TSpaces [77] is an IBM middleware system. The goal of TSpaces is to support communication, computation and data management on hand-held devices. TSpaces is the marriage of tuplespace and database technologies, implemented in Java. The tuplespace component provides a flexible communication model; the database component adds stability, durability, advanced query capabilities and extensive data storage capacity; Java adds instant portability.

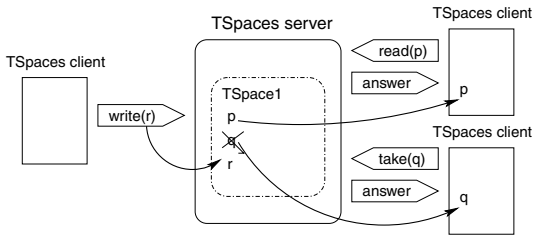

Fig. 15. Examples of interactions in TSpaces.

The TSpaces design distinguishes between clients and servers. A tuple space exists only on a TSpaces server, while a server may host several spaces. Once a tuple space has been created, a TSpaces client is allowed to perform operations, like read and write, on it. A TSpaces server is a centralised server that listens to client requests: each time a client issues an operation, information is sent to the server that, using a lookup operation, finds out the tuplespace on which the operation needs to be performed and passes the operation and tuple operand to it to process (see Figure 15).

There are two specific server systems tuple spaces: Galaxy, that contains tuples describing each tuple space that exists on a server; and Admin, that

contains access control permissions for each tuplespace and whose goal is to check whether the issuer of each operation has the proper access control privileges.

TSpaces is different from other tuple space based systems for the following reasons: first, the behaviour of the middleware is dynamically modifiable. New operators can be defined, and new datatypes and operators can be introduced. Second, TSpaces employs a real data management layer, with functionalities similar to relational database systems. Operations are performed in a transactional context that ensures the integrity of data. Support for indexing and query capability is provided. Data (i.e., tuples stored in the database) and operations that act on data are kept separated, so that operations can be added or changed without affecting the database.

Unlike Lime, TSpaces mainly targets nomadic environments where servers containing tuple data bases are stored on fixed and powerful machines, reachable by mobile devices roaming around. The transactional approach to tuple read/write is also a limitation in terms of mobility as the paradigms might be too heavy if the connection is fluctuating. Furthermore disconnection is seen as a fault and, when disconnected, clients do not have access to the tuple spaces.

L2imbo. L2imbo [17] is a tuple space based middleware with emphasis on quality of service. Some of the features of L2imbo are: multiple tuple spaces, tuple type hierarchy, quality of service attributes, monitoring and adaptation agents.

Like TSpaces and Lime, L2imbo provides the ability to create multiple tuple spaces. Tuple spaces can be created when needed, provided that creation and termination go through the universal tuple space, which is L2imbo main tuple space. The tuple spaces are implemented in a distributed system fashion, where each host holds a replica of the tuple space, so to allow for disconnected operations. L2imbo also provides quality of service features; quality of service fields can be associated with the tuples, such as deadline for a tuple, which indicates how long a tuple should be available in the space. Associated to these quality of service fields is the QoS monitoring agent, which monitors the conditions of the network, costs of connection and power consumption. Information on quality of service can then be placed into tuples and made available to other agents or hosts.

Given the support for disconnected operations and the use of an asynchronous communication paradigm, L2imbo seems to be well suited for highly mobile environments.

5.5 Service Discovery in Mobile Computing Middleware

In traditional middleware systems, service discovery is provided using fixed name services, which every host knows the existence of. The more the network becomes dynamic, the more difficult service and host discovery becomes. Already in distributed peer-to-peer network [50] service discovery is more complex as hosts join and leave the overlay network very frequently. In mobile systems service

discovery can be quite simple, if we speak about nomadic systems where a fixed infrastructure containing all the information and the services is present. However in terms of more ad-hoc or mixed systems where services can be run on roaming hosts, discovery may become very complex and/or expensive.

Most of the ad-hoc systems encountered till now have their own discovery service. Lime and Xmiddle use a completely ad-hoc strategy where hosts continuously monitor their environment to check who is available and what they are offering. A trade-off between power consumption (i.e. broadcast) and discovery needs to be evaluated. Recently, some work on Lime for service advertisement and discovery has been devised [29]. Standard service discovery frameworks have appeared in the recent years: UPnP [75], Jini [5], and Salutation [61]. UPnP stands for Universal Plug and Play and it is an open standard for transparently connecting appliances and services, which is adopted by the Microsoft operating systems. UPnP can work with different protocols such as TCP, SOAP, HTTP. Salutation is a general framework for service discovery, which is platform and OS independent. Jini is instead Java based and dependent on the Java Virtual Machine. The purpose of these frameworks is to allow groups of devices and software components to federate into a single, dynamic distributed system, enabling dynamic discovery of services inside the network federation. We now describe Jini and Salutation.

Jini and JMatos

Jini [5] is a distributed system middleware based on the idea of federating groups of users and resources required by those users. Its main goal is to turn the network into a flexible, easily administered framework on which resources (both hardware devices and software programs) and services can be found, added and deleted by humans and computational clients.

The most important concept within the Jini architecture is the service. A service is an entity that can be used by a person, a program or another service. Members of a Jini system federate in order to share access to services. Services can be found and resolved using a lookup service that maps interfaces indicating the functionality provided by a service to sets of objects that implement that service. The lookup service acts as the central marketplace for offering and finding services by members of the federation. A service is added to a lookup service by a pair of protocols called *discovery* and *join*: the new service provider locates an appropriate lookup service by using the first protocol, and then it joins it, using the second one (see Figure 16). A distributed security model is put in place in order to give access to resources only to authorised users.

Jini assumes the existence of a fixed infrastructure which provides mechanisms for devices, services and users to join and detach from a network in an easy, natural, often automatic, manner. It relies on the existence of a network of reasonable speed connecting Jini technology-enabled devices.

However the large footprint of Jini (3 Mbytes), due, mainly, to the use of Java RMI, prevents the use of Jini on smaller devices such as iPAQs or PDAs. In this direction Psinaptic JMatos [54] has been developed, complying with the

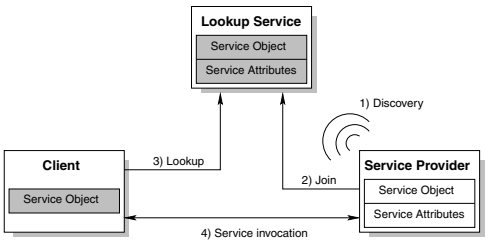

Fig. 16. Discovery, join and lookup mechanism in Jini.

Jini Specification. JMatos does not rely on Java RMI for messaging and has a footprint of just 100 Kbytes.

Salutation
Salutation aims at platform and operating system independence. It focuses on interoperability of different services through a set of common standards for the specification of the service functionalities. The discovery of services is managed by Salutation Managers that interact with each others through RPC, exchanging service registration information. Salutation managers also direct the communication among different clients, notifying them when new data is arriving, for example. The concept of service is decomposed in sets of functional units, representing specific features of the services. Salutation managers also specify the format of the transmitted data implementing transport protocol independence.

In general, despite being the most complete systems in terms of discovery, these systems hardly provide any support of the other important non-functional requirements that would grant the word "middleware" to them. We believe a number of functionalities of the systems we have seen could be combined in an efficient middleware for mobile computing.

6 Future Research Directions

According to our model, a middleware for mobile computing should be lightweight as it must run on hand-held, resource-scarce devices; it should support an asynchronous form of communication, as mobile devices connect to the network opportunistically and for short periods of time; and it should be built with the principle of awareness in mind, to allow its applications to adapt its own and the middleware behaviour to changes in the context of execution, so as to achieve the best quality of service and optimal use of resources.

Furthermore, the variability between different mobile systems, ranging from pure ad-hoc to nomadic with only few roaming hosts, and the different network connectivity paradigms, add further parameters which make it very difficult to isolate a general purpose middleware solution.

We believe research in terms of mobile computing middleware still faces many challenges that might not be solvable adapting traditional middleware techniques.

We believe the future mobile networks will be heterogeneous in the sense that many different devices will be available on the market, with possibly different operating systems and user interfaces. The network connectivity will also be heterogeneous even if an effort towards complete coverage through different connection technologies will be made. For these reasons mobile computing middleware will have to adapt and be customisable in these different dimensions, both at start-up time (i.e., in case of adaptation to different operating systems) and at run-time (i.e., in case of adaptation to different connection technologies).

We also believe application dependent information could play an important role in the adaptation of the behaviour of the middleware and in the trade-off between scarce resource availability and efficient service provision. In this direction, the effort of presentation of the information to the application, and the gathering of application dependent policies is an important presentation layer issue that should be integrated in any mobile computing middleware.

Discovery of existing services is a key point in mobile systems, where the dynamicity of the system is, by orders of magnitude, higher than traditional distributed systems. Recently, interesting research advances in peer to peer systems [71, 50] have focused on discovery issues, that might be applicable, at least partially, to mobile settings. However, considerations on the variability of the connection, of the load and of the resources might be different for mobile scenarios. Furthermore, the integration of quality of service consideration into the service advertisement and discovery might enable some optimisation in the service provision.

Another direction of research concerns security. Portable devices are particularly exposed to security attacks as it is so easy to connect to a wireless link. Dynamic customisation techniques seems somehow to worsen the situation. Reflection is a technique for accessing protected internal data structures and it could cause security problems if malicious programs break the protection mechanism and use the reflective capability to disclose, modify or delete data. Security is a major issue for any mobile computing application and therefore proper measures need to be included in the design of any mobile middleware system.

7 Summary

The growing success of mobile computing devices and networking technologies, such as WaveLan [30] and Bluetooth [10], have called for the investigation of new middleware that deal with mobile computing challenges, such as sudden disconnections, scarce resource availability, fast changing environment, etc. During the last years, research has been active in the field of middleware, and a considerable number of new systems has been designed to support this new computational paradigm.

In this survey we have presented a reference model to classify distributed systems into traditional, nomadic and ad-hoc ones. We have highlighted their similarities and mainly their differences. We then listed a set of characteristics that seem to be effective in mobile computing middleware. This list has been helpful to describe some of the unsuitable aspect of traditional middleware systems in a mobile setting, and the principles that have driven towards a new set of middleware for mobile computing

Several classes of middleware for mobile computing (i.e., reflective middleware, tuplespace-based middleware, context-aware middleware, data-oriented) have been identified, illustrated and comparatively discussed. Although addressing some of the issues related to mobility, none of these systems succeed in providing support for all the requirements highlighted by our framework.

Acknowledgements The authors would like to thank Zuhlke Engineering (UK) Ltd. for supporting Licia Capra; Stefanos Zachariadis and Gian Pietro Picco for comments on an earlier draft of this paper.

References

1. Alternis S.A. Solutions for Location Data Mediation. http://www.alternis.fr/.
2. O. Angin, A. Campbell, M. Kounavis, and R. Liao. The Mobiware Toolkit: Programmable Support for Adaptive Mobile Netwoking. In *Personal Communications Magazine, Special Issue on Adapting to Network and Client Variability*. IEEE Computer Society Press, August 1998.
3. ANSA. The Advanced Network Systems Architecture (ANSA). Reference manual, Architecture Project Management, Castle Hill, Cambridge, UK, 1989.
4. V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs, A. L. Hors, G. Nicol, J. Robie, R. Sutor, C. Wilson, and L. Wood. Document Object Model (DOM) Level 1 Specification. W3C Recommendation http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001, World Wide Web Consortium, Oct. 1998.
5. K. Arnold, B. O'Sullivan, R. W. Scheifler, J. Waldo, and A. Wollrath. *The Jini[tm] Specification*. Addison-Wesley, 1999.
6. A. Asthana and M. C. P. Krzyzanowski. An indoor wireless system for personalized shopping assistence. In *Proceedings of IEEE Workshop on Mobile Computing Systems and Applications*, pages 69–74, Santa Cruz, California, Dec. 1994. IEEE Computer Society Press.
7. S. Baker. *Corba Distributed Objects : Using Orbix*. Addison-Wesley, Nov. 1997.
8. F. Bennett, T. Richardson, and A. Harter. Teleporting - making applications mobile. In *Proc. of the IEEE Workshop on Mobile Computing Systems and Applications*, pages 82–84, Santa Cruz, California, Dec. 1994. IEEE Computer Society Press.
9. G. Blair, G. Coulson, P. Robin, and M. Papathomas. An Architecture for Next Generation Middleware. In *Proceedings of Middleware '98*, pages 191–206. Springer Verlag, Sept. 1998.
10. Bluetooth.com. Bluetooth. http://www.bluetooth.com.
11. T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language. Recommendation http://www.w3.org/TR/1998/REC-xml-19980210, World Wide Web Consortium, Mar. 1998.

12. P. Brown. Triggering information by context. *Personal Technologies*, 2(1):1–9, Mar. 1998.
13. CellPoint, Inc. The CellPoint System. http://www.cellpt.com/thetechnology2.htm, 2000.
14. R. Cerqueira, C. K. Hess, M. Romn, and R. H. Campbell. Gaia: A Development Infrastructure for Active Spaces. In *Workshop on Application Models and Programming Tools for Ubiquitous Computing (held in conjunction with the UBICOMP 2001)*, Sept. 2001.
15. D. Chalmers and M. Sloman. A Survey of Quality of Service in Mobile Computing Environments. *IEEE Communications Surveys*, Second Quarter:2–10, 1999.
16. J. Clark and S. DeRose. XML Path Language (XPath). Technical Report http://www.w3.org/TR/xpath, World Wide Web Consortium, Nov. 1999.
17. N. Davies, A. Friday, S. Wade, and G. Blair. L2imbo: A Distributed Systems Platform for Mobile Computing . *ACM Mobile Networks and Applications (MONET), Special Issue on Protocols and Software Paradigms of Mobile Networks*, 3(2), 1998.
18. A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and B. welch. The Bayou Architecture: Support for Data Sharing among Mobile Users. In *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications*, pages 2–7, Santa Cruz, California, Dec. 1994.
19. A. Dey, M. Futakawa, D. Salber, and G. Abowd. The Conference Assistant: Combining Context-Awareness with Wearable Computing. In *Proc. of the 3rd International Symposium on Wearable Computers (ISWC '99)*, pages 21–28, San Franfisco, California, Oct. 1999. IEEE Computer Society Press.
20. W. Emmerich. *Engineering Distributed Objects*. John Wiley & Sons, Apr. 2000.
21. W. Emmerich. Software Engineering and Middleware: A Roadmap. In *The Future of Software Engineering - 22nd Int. Conf. on Software Engineering (ICSE2000)*, pages 117–129. ACM Press, May 2000.
22. ExoLab. OpenORB. http://openorb.exolab.org/openorb.html, 2001.
23. D. C. Fallside. XML Schema. Technical Report http://www.w3.org/TR/xmlschema-0/, World Wide Web Consortium, Apr. 2000.
24. W. Forum. Wireless Application Protocol. http://www.fub.it/dolmen/, 2000.
25. D. Fritsch, D. Klinec, and S. Volz. NEXUS Positioning and Data Management Concepts for Location Aware Applications. In *Proceedings of the 2nd International Symposium on Telegeoprocessing*, pages 171–184, Nice-Sophia-Antipolis, France, 2000.
26. D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
27. M. Haahr, R. Cunningham, and V. Cahill. Supporting CORBA Applications in a Mobile Environment (ALICE). In *5th Int. Conf. on Mobile Computing and Networking (MobiCom)*. ACM Press, August 1999.
28. C. Hall. *Building Client/Server Applications Using TUXEDO*. Wiley, 1996.
29. R. Handorean and G.-C. Roman. Service Provision in Ad Hoc Networks. In *Coordination 2002*. Springer, 2002.
30. G. Held. *Data Over Wireless Networks: Bluetooth, WAP, and Wireless Lans*. McGraw-Hill, Nov. 2000.
31. E. Hudders. *CICS: A Guide to Internal Structure*. Wiley, 1994.
32. A. D. Joseph, J. A. Tauber, and M. F. Kaashoek. Mobile Computing with the Rover Toolkit. *IEEE Transactions on Computers*, 46(3), 1997.
33. G. Kiczales, J. des Rivieres, and D. Borrow. *The Art of the Metaobject Protocol*. The MIT Press, 1991.

34. F. Kon, M. Román, P. Liu, J. Mao, T. Yamane, L. M. aes, and R. Cambpell. Monitoring, Security, and Dynamic Configuration with the *dynamicTAO* Reflective ORB. In *International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, pages 121–143, New York, Apr. 2000. ACM/IFIP.
35. L. Capra and W. Emmerich and C. Mascolo. A Micro-Economic Approach to Conflict Resolution in Mobile Computing. March 2002. Submitted for Publication.
36. T. Ledoux. OpenCorba: a Reflective Open Broker. In *Reflection'99*, volume 1616 of *LNCS*, pages 197–214, Saint-Malo, France, 1999. Springer.
37. S. Long, R. Kooper, G. Abowd, and C. Atkenson. Rapid prototyping of mobile context-aware applications: the Cyberguide case study. In *Proceedings of the Second Annual International Conference on Mobile Computing and Networking*, pages 97–107, White Plains, NY, Nov. 1996. ACM Press.
38. E. Maler and S. DeRose. XML Linking Language (XLink). Technical Report http://www.w3.org/TR/1998/WD-xlink-19980303, World Wide Web Consortium, Mar. 1998.
39. B. W. Marsden. *Communication Network Protocols: OSI Explained*. Chartwell-Bratt, 1991.
40. C. Mascolo, L. Capra, S. Zachariadis, and W. Emmerich. XMIDDLE: A Data-Sharing Middleware for Mobile Computing. *Int. Journal on Personal and Wireless Communications*, April 2002.
41. J. McAffer. Meta-level architecture support for distributed objects. In *Proceedings of Reflection'96*, pages 39–62, San Francisco, 1996.
42. Microsoft. .NET Compact Framework. http://msdn.microsoft.com/vstudio/device/compactfx.asp, 2002.
43. R. Monson-Haefel. *Enterprise Javabeans*. O'Reilly & Associates, Mar. 2000.
44. R. Monson-Haefel, D. A. Chappell, and M. Loukides. *Java Message Service*. O'Reilly & Associates, Dec. 2000.
45. A. L. Murphy, G. P. Picco, and G.-C. Roman. LIME: A Middleware for Physical and Logical Mobility. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-21)*, May 2001.
46. V. Natarajan, S. Reich, and B. Vasudevan. *Programming With Visibroker : A Developer's Guide to Visibroker for Java*. John Wiley & Sons, Oct. 2000.
47. E. B. R. A. Networks. ETSI HIPERLAN/2 Standard. http://portal.etsi.org/bran/kta/Hiperlan/hiperlan2.asp.
48. OMG. CORBA Component Model. http://www.omg.org/cgi-bin/doc?orbos/97-06-12, 1997.
49. Oracle Technology Network. Oracle9i Application Server Wireless. http://technet.oracle.com/products/iaswe/content.html, 2000.
50. A. Oram. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. O'Reilly, 2001.
51. C. Perkins. *Ad-hoc Networking*. Addison-Wesley, Jan. 2001.
52. E. Pitt and K. McNiff. *Java.rmi : The Remote Method Invocation Guide*. Addison Wesley, June 2001.
53. A. Pope. *The Corba Reference Guide : Understanding the Common Object Request Broker Architecture*. Addison-Wesley, Jan. 1998.
54. Psinaptic. JMatos. http://www.psinaptic.com/, 2001.
55. I. Redbooks. *MQSeries Version 5.1 Administration and Programming Examples*. IBM Corporation, 1999.

56. T. Reinstorf, R. Ruggaber, J. Seitz, and M. Zitterbart. A WAP-based Session Layer Supporting Distributed Applications in Nomadic Environments. In *Int. Conf on Middleware*, pages 56–76. Springer, Nov. 2001.
57. P. Reynolds and R. Brangeon. Service Machine Development for an Open Long-term Mobile and Fixed Network Environment. http://www.fub.it/dolmen/, 1996.
58. D. Rogerson. *Inside COM*. Microsoft Press, 1997.
59. G.-C. Roman, A. L. Murphy, and G. P. Picco. Software Engineering for Mobility: A Roadmap. In *The Future of Software Engineering - 22^{nd} Int. Conf. on Software Engineering (ICSE2000)*, pages 243–258. ACM Press, May 2000.
60. M. Roman, F. Kon, and R. Campbell. Reflective Middleware: From your Desk to your Hand. *IEEE Communications Surveys*, 2(5), 2001.
61. Salutation Consortium. Salutation. http://www.salutation.org/, 1999.
62. M. Satyanarayanan. Mobile Information Access. *IEEE Personal Communications*, 3(1):26–33, Feb. 1996.
63. M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers*, 39(4):447–459, Apr. 1990.
64. B. Schilit, N. Adams, and R. Want. Context-Aware Computing Applications. In *Proc. of the Workshop on Mobile Computing Systems and Applications*, pages 85–90, Santa Cruz, CA, Dec. 1994.
65. A. Schill, W. Bellmann, and S. Kummel. System support for mobile distributed applications, 1995.
66. SignalSoft. Wireless Location services. http://www.signalsoftcorp.com/, 2000.
67. B. Smith. Reflection and Semantics in a Procedural Programming Langage. Phd thesis, MIT, Jan. 1982.
68. Softwired. iBus Mobile. http://www.softwired-inc.com/products/mobile/mobile.html, Apr. 2002.
69. W. R. Stevens. *UNIX Network Programming*. Prentice Hall, 1997.
70. Sun Microsystem, Inc. Java Micro Edition. http://java.sun.com/products/j2me/, 2001.
71. Sun Microsystems, Inc. Jxta Initiative. http://www.jxta.org/, 2001.
72. K. Tai. The Tree-to-Tree Correction Problem. *Journal of the ACM*, 29(3):422–433, 1979.
73. D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP-15)*, pages 172–183, Cooper Mountain, Colorado, Aug. 1995.
74. Ubi-core. Universally Interoperable Core. http://www.ubi-core.com, 2001.
75. UPnP Forum. Universal Plug and Play. http://www.upnp.org/, 1998.
76. G. Welling and B. Badrinath. An Architecture for Exporting Environment Awareness to Mobile Computing. *IEEE Transactions on Software Engineering*, 24(5):391–400, 1998.
77. P. Wyckoff, S. W. McLaughry, T. J. Lehman, and D. A. Ford. T Spaces. *IBM Systems Journal*, 37(3):454–474, 1998.
78. Y. Yokote. The Apertos reflective operating system: The concept and its implementation. In *Proceedings of OOPSLA'92*, pages 414–434. ACM Press, 1992.