From Distributed Systems to Ubiquitous Computing – The State of the Art, Trends, and Prospects of Future Networked Systems

Friedemann Mattern
Department of Computer Science
Federal Institute of Technology (ETH Zürich)
CH-8092 Zürich, Switzerland
mattern@inf.ethz.ch

Peter Sturm
Department of Computer Science
University of Trier
54286 Trier, Germany
sturm@uni-trier.de

Abstract: We summarize trends in communication paradigms for networked systems, mention well-established as well as innovative software infrastructures for distributed systems (such as COM+, CORBA, .NET and Jini), and give an overview of application domains such as grid computing, peer-to-peer computing, and mobile agents. We then discuss issues in spontaneous networking, and we explain the vision of Ubiquitous Computing and its intriguing prospects.

Keywords: RPC, RMI, software bus, publish/subscribe, COM+, .NET, CORBA, Jini, application server, grid computing, peer-to-peer computing, mobile code, mobile agents, pervasive computing, smart devices, spontaneous networking, smart labels, RFID tags, middleware, distributed systems, ubiquitous computing

1. Introduction

A distributed system consists of several computers that communicate over a network to coordinate the actions and processes of an application. Distributed systems techniques have attracted much interest in recent years due to the proliferation of the Web and other Internet-based systems and services.

Well-established techniques such as interprocess communication and remote invocation, naming services, cryptographic security, distributed file systems, data replication, and distributed transaction mechanisms provide the run-time infrastructure supporting today's networked applications [CDK 00]. The dominant model is still the traditional client-server architecture. However, application development for distributed systems now relies more and more on middleware support through the use of software frameworks (e.g. CORBA or Web services) that provide higher-level abstractions such as distributed shared objects, and on services including secure communication, authentication, yellow pages, and persistent storage.

In the future, distributed application frameworks may support mobile code, multimedia data streams, user and device mobility, and spontaneous networking [CDK 00]. Scalability, quality of service, and robustness with respect to partial component failures will become key issues.

Clearly, a shift towards large-scale systems has occurred in recent years: not only the pure Internet with its basic protocols, but also the higher-level World Wide Web is becoming a standard platform for distributed applications. Here, the Internet (or an all-encompassing intranet) and its resources are viewed as the global environment in which computations take place. Consequently, high-level protocols and standards, such as XML, enter the focus of distributed system research while low-level issues (such as operating system peculiarities) become less important. The increasing number of computers connected to the Internet has also laid the foundations for new application domains such as grid computing and peer-to-peer computing. Grid computing emphasizes the fact that the Internet can be viewed as a globally distributed computer with an enormous potential for computing power. In contrast to this, peer-to-peer computing underscores the needs of the people behind all these machines. Their desires for immediate and unrestricted information exchange, for anonymity, and for independence from restrictive rules imposed by providers or governments are about to shape the Internet again.

Rapidly evolving network and computer technology, coupled with the exponential growth of the services and information available on the Internet, will soon bring us to the point where hundreds of millions of people have fast, pervasive access to a phenomenal amount of information, through desktop machines at work, school and home, through mobile phones, personal digital assistants (PDA), and car dashboards, from anywhere and everywhere [KG 99]. The challenge of distributed system technology is to provide flexible and reliable infrastructures for such large-scale systems that meet the demands of developers, users, and service providers.

Looking further into the future, essential techniques of distributed systems will be incorporated into an emerging new area, called "Ubiquitous Computing" [Wei 91]. The vision of Ubiquitous Computing (or "pervasive computing", as it is sometimes called) is in some sense a projection of the Internet phenomenon and the mobile phone proliferation phenomenon we observe today into the future, envisioning billions of communicating smart devices forming a world-wide distributed system several orders of magnitude larger than today's Internet.

2. Communication Paradigms

There are many ways in which application software components residing on different machines can communicate with one another over a network. One low-level technique is to directly use the call interfaces of the transport layer, such as the socket mechanism, together with a custom communication protocol. However, programming at this level of abstraction is advisable only in special circumstances since it leaves complex problems such as the guarantee of security, the management of heterogeneity, and concurrency control completely to the application programmer. Instead, developers should choose from a variety of higher level communication protocols and frameworks the one that best suits their needs. Some of these protocols such as the well-known remote procedure call are self-contained and can be used in any application program with almost no additional overhead. Other protocols and frameworks are confined to specific programming languages or execution platforms.

2.1 Remote Procedure Call

One well-established classical communication scheme that fits well with the client-server model is the remote procedure call (RPC). In this model, a component acts as a client when it requests some service from another component; it acts as a server when it responds to a request from a client. RPC makes calling an external procedure that resides in a different network node almost as simple as calling a local procedure. Arguments and return values are automatically packaged in an architecture-neutral format sent between the local and remote procedures.

For each remote procedure, the underlying RPC framework needs a so-called stub procedure on the client side (which acts as a proxy for the remote procedure) and a similar object on the server side. The role of the stub is to take the parameters passed in a regular local procedure call and pass them to the RPC system (which must be resident on both the client and server nodes) [carl b]. Behind the scenes, the RPC system cooperates with the stubs on both sides to transfer the arguments and return values over the network [wash].

To facilitate the creation of stubs, RPC toolkits include special tools. The programmer provides details of an RPC call in the form of specifications encoded in an Interface Definition Language (IDL). An IDL compiler is used to generate the stubs automatically from the IDL specifications [carl]. The stubs can then be linked into clients and servers [carl b].

Several different RPC systems coexist today. On Unix/Linux platforms, an RPC system introduced by Sun Microsystems [ONC] is used for accessing many system services such as the network file system NFS and the network information system NIS. The DCE-RPC [DCE] is used as the basis for Microsoft's COM+ middleware. RPC frameworks have become an established technique, and are typically invisible to application programmers since they merely represent the basic transport mechanism used by more general middleware platforms as presented in section 3.

2.2 XML-based RPC

Although RPC systems explicitly address the issue of interoperability in open systems, client and server programs that make use of this principle are tied to a single RPC framework in order to cooperate successfully. The main reason for this is that every RPC system defines its own encoding for data structures. Despite this difference, the basic semantics of most RPC systems are quite similar since all of them are based on a synchronous procedure call in a C-like syntax format.

Viewing the messages sent between clients and servers in an RPC system as documents with a given syntax was a major breakthrough in RPC technology. Using XML to define the syntax of RPC requests and replies was a simple idea, but it paved the way for interoperability between different RPC systems [BSL 00, McL 01]. As a result, XML-based RPC systems are used today to integrate otherwise incompatible application programs, and they are thus an essential part of solutions in collaborative business which integrate the different legacy applications used by cooperating companies. For legacy software, additional wrapper programs must be implemented that translate a specific set of XML-based RPC requests into proprietary function calls. Some modern database-centric execution platforms are even capable of processing a given XML-based RPC dialect directly.

The essential idea in XML-based RPC frameworks is to use XML to define a type system that can be used to communicate data between clients and servers. These type systems specify primitive types such as integers, floating points, and text strings, and they provide mechanisms for aggregating instances of these primitive types into compound types in order to specify and represent new data types. One of the first XML-based RPC frameworks was the simple object access protocol SOAP [SOAP], defined by a consortium of companies including Microsoft, IBM, and SAP. SOAP is now an integral part of Windows operating systems (as part of the COM+ and .NET middleware). A key advantage of SOAP is its extensibility by use of XML schemas and the fact that the widespread HTTP protocol can be used as the transport mechanism between clients and servers, thus using the Web as a communication infrastructure and a tunnel between cooperating distributed applications. Servers may appear in this scenario as dedicated Web servers with the ability to trigger servlets, scripts, or any other means of program execution.

Other XML-based RPC dialects such as standard XML-RPC have also been specified [McL 01]. Most of them, however, do not incorporate XML schemas and are thus limited to a fixed set of primitive data types, as are traditional RPC systems. Fortunately, having to cope with different dialects of XML-based RPC systems is not as involved as dealing with incompatible classical RPC frameworks, because in many situations simple transformation rules can be defined to convert XML encoded messages between different RPC platforms.

2.3 Remote Method Invocation

Whilst RPC is reasonably well suited to the procedural programming paradigm, it is not directly applicable to the object-oriented programming style that has gained much popularity in recent years. Here, Remote Method Invocation (RMI) – a newer technique for Java-based systems – comes into its own. RMI is similar to RPC, but integrates the distributed object model into the Java language in a natural way [MIT]. With RMI, it is not necessary to describe the methods of remote objects in a separate type definition file. Instead, RMI works directly from existing objects, providing seamless integration [jsol]. Furthermore, remote objects can be passed as parameters in remote method calls, a feature that classical RPC systems usually do not possess.

In classical RPC systems, client-side stub code must be generated and linked into a client before a remote procedure call can be made. RMI is more dynamic in this respect: owing to the Java capability of transferring code, the stubs that are needed for an invocation can be downloaded at runtime (in architecture-neutral bytecode format) from a remote location, for example directly from the server just before the remote method is actually invoked. Internally, RMI makes use of object serialization to transmit arbitrary object types over the network, and because downloaded code can potentially be harmful to the system, it uses a security manager to check this.

RMI seems to fit well with the general trend of distributed systems becoming increasingly dynamic.

2.4 Asynchronous Protocols

RPC and RMI are basically synchronous calls: the client is blocked while the call is processed by the server. Asynchronous variants that require multithreading are difficult to handle and are error prone. However, the trend in distributed systems is towards asynchronous and reactive systems – systems that cannot wait indefinitely for a synchronous call to terminate. Typical examples are user interface systems or real-time systems.

Such reactive and asynchronous systems are better served by a more abstract communication paradigm based on events. Events are simple asynchronous messages, but they are attractive because they represent an intuitive (although somewhat restricted) way of modeling that something has happened that is potentially of interest to some other objects. Distributed infrastructures based on this paradigm are often called "publish and subscribe middleware" or simply "event channel" or "software bus".

The concept of such a software bus is quite simple: all clients (i.e. consumers of events) are connected to a shared medium called a "bus". They announce their interest in a certain topic (i.e. type of events) by subscribing to it. Objects or processes that want to send a message or an event to clients publish it under a certain topic to the bus. If the receiver's topic matches the sender's topic, the message is forwarded to the receiver.

From a software design point of view, the event paradigm offers the benefits of direct notification instead of busy-probing: an object tells the environment to inform it when something happens, and when something happens it reacts accordingly [cal]. It thus eliminates the need for consumers to periodically poll for new events. Furthermore, the concept easily allows the use of adapters that act as programmable middlemen in the event streams. They may for example multiplex, filter, or aggregate events.

However, although the publish/subscribe paradigm is attractive due to its conceptual simplicity and has the benefit of decoupling objects in space and time (while at the same time abstracting from the network topology and the heterogeneity of the components), it requires a powerful brokering mechanism that takes care of event delivery. Furthermore, its use requires the implicit or explicit semantics of the brokering mechanism to be carefully analyzed – a priori it is not clear how quickly events are to be delivered, whether events are to remain on the bus for a certain period of time (for example, so that subscribers who miss anything due to system failure can catch up on missed events) and whether there are any guarantees on the delivery order. It should be noted that the publish/subscribe paradigm is basically a multicast scheme, and it is well known from distributed systems theory that broadcast semantics, in particular for dynamic environments, is a non-trivial issue.

The overall trend in communication paradigms seems to be clear: it is progressing from the simple procedural paradigm requiring relatively little system support, via the object-oriented method invocation principle, towards Web-based infrastructures and more abstract schemes for loosely-coupled asynchronous systems that require complex run-time infrastructures. One can expect platforms supporting such abstract communication paradigms on the global Internet scale to become increasingly important in the future, probably being integrated with the routing and management schemes for general Internet-based messaging services. Scalability and efficient realization of such infrastructures is a real challenge, however.

3. Trends in Infrastructures and Middleware

Middleware and software infrastructures for distributed systems provide basic communication facilities to application components and handle issues such as platform heterogeneity that are due to differing hardware systems, operating systems, or programming languages. Furthermore, they provide a set of standard services that are typically needed by distributed applications such as directory services and cryptographic security.

One of the first commercially available middleware systems was DCE [DCE], which is still being used in many large legacy applications. Communication in DCE is based on RPC, and it essentially provides directory services, security services (based on the well-known Kerberos system), and a distributed file service. Many middleware concepts were initially implemented using DCE and evolved over time as part of more modern middleware systems. Some of today's most prominent candidates are presented in the following sections.

3.1 COM+

The component object model COM and its latest version COM+ have their roots in the basic clipboard mechanisms provided by early Microsoft operating systems to enable the exchange of data between different applications [EE 99]. This simple mechanism was iterated over the years and evolved into a pragmatic component platform. The COM+ platform is widely used in industry and a huge variety of COM+ components are available and can be purchased from specialized vendors.

Components in COM+ are defined through a set of interfaces. The interface definition language MIDL is an extension of the DCE IDL. This IDL allows the definition of common static and dynamic data types (essentially variable-length arrays). It is object-oriented in the sense that single inheritance is possible for interfaces. Unfortunately, there is no implementation inheritance, thus a component defining a derived interface must implement all functions of the base interfaces again, no matter whether the semantics of the method are changed or not. The root of all interface hierarchies is the IUnkown interface with its main method of downcasting to any derived interface if possible. Interfaces are identified by 128-bit random numbers called global unique identifiers (GUID). COM+ allows the representation of meta-information about interfaces by means of type libraries (generated automatically by the IDL compiler). This type information is used in a process called "automation", where the calling application can interrogate implemented interfaces, method signatures, and data types of a component at runtime. Based on this information, the caller can build and issue method calls to a component step by step by using the rather involved IDispatch interface.

Since COM+ is a binary component standard, a component itself ships as a dynamic link library (DLL) with tables holding function pointers to the methods of each interface. Each component is again identified by another 128-bit GUID. The most prominent usage of components is through so-called inproc servers by linking the requested DLL to the virtual address space of the client program and accessing methods using indirect function calls. The mapping between the component identifier (GUID) and the path of the dynamic library is stored in the registry of the Windows operating system. Additionally, components can also be executed within their own virtual address space, for example as dedicated servers or more commonly as a dynamic library linked into a specialized surrogate process. A third possibility is to use a component of a remote server. In this case, RPC requests and replies are generated transparently by the COM+ runtime platform. The required stub and proxy functions, generated automatically by the MIDL compiler, are implemented as additional dynamic libraries. They are again identified by the COM+ runtime system using the Windows registry and the 128 bit interface identifier.

Most of the functionality of Microsoft Windows operating systems has subsequently been converted to COM+ services. Some of these components are rather heavyweight such as Internet Explorer, the office program suite, or DirectX (for efficiently accessing multimedia devices), and most of them are used only locally. But an increasing number of COM+ components form an integral part of standard services in Microsoft Windows networks and may be used remotely, such as directory services (Active Directory), event propagation, file and printer sharing, and even remote desktops.

Despite its widespread use, COM+ middleware has a number of deficiencies and limitations. Most importantly, the implementation of components is quite complex and requires the implementation of an impressive number of additional methods. Integrated development environments such as Visual Studio try to tackle this problem by offering assistance (for example, using so-called wizards) to ease the burden on the component developer. Because of the flat namespace, the COM+ components available on a given system are also difficult to manage and organize. Furthermore, the packing of components into one or more dynamic libraries and the flat identifier namespace for interfaces as well as components within the Windows registry are susceptible to system instability in the event of program installation or updates.

3.2 The .NET Framework

Some of the deficiencies of COM+ middleware are addressed by Microsoft's more recent initiative, the .NET framework [TL 02, Ric 02]. Primarily introduced as a rival product to Sun's Java platform, .NET defines a virtual runtime system CLR (Common Language Runtime) for most components and applications. The CLR is based on a Common Type System (CTS), a single-inheritance object-oriented hierarchy of common built-in types such as integers, floating points, and text strings. Ideally, components and application programs implemented in a given programming language are translated into an intermediate language (IL), comparable to Java bytecode, using the CTS. This so-called "managed code" can be executed by the CLR. Currently, IL compilers are available for Visual Basic, JScript, C++ (with restrictions, of course), and C# as part of Visual Studio .NET. Any IL code is translated into native machine language on a per method base before it is executed within the CLR. In contrast to the Java virtual machine, this intermediate language is never interpreted at the instruction level. The CTS enables the immediate use of built-in types as well as user-defined types within different programming languages, e.g. a class in managed C++ code (C++ code within the restrictions defined by the CTS) can be derived from a Visual Basic super class (something all serious developers have been wanting to do for years).

More interesting with respect to distributed systems are advances in component packaging, so-called assemblies. Assemblies are used to keep track of all the resources required by a single component or application. Each assembly is accompanied by a manifest which stores the assembly name and version, security information, names and versions of other assemblies referenced by this assembly, and type information about the public classes defined in this assembly. Additionally, a code base URL can be defined where all the files of an assembly can be downloaded if necessary. Since all the meta-information making up an assembly is stored in a single manifest file, management of installed assemblies and their interdependencies is substantially improved compared to the classical registry approach. The technique used in .NET resembles the packaging of enterprise beans in Java (see also section 4.1) in a broader sense, although the manifest information itself is not described by a self-descriptive language such as XML. In contrast to enterprise beans which ship as a single jar file, assemblies may still consist of several files in order to allow for the partial download of complex components.

3.3 CORBA

One of the most widely-used infrastructures for distributed systems based on an object-oriented model is still CORBA, the Common Object Request Broker Architecture, which is supported by a large industry consortium. The first CORBA standard was introduced in 1991, and it has undergone continual and significant revisions ever since [wash]. Part of the specification describes an Interface Definition Language (IDL) that all CORBA implementations must support. The CORBA IDL is based on C++ and is used by applications to define the externally visible parts of object methods that can be invoked by other objects. It has a similar function to the IDL of COM+.

The central component of a CORBA system is the object request broker (ORB). The ORB provides a mechanism for transparently communicating client requests to target object implementations. It simplifies distributed programming by decoupling the client from the details of the method invocations: when a client invokes an operation, the ORB is responsible for finding the object implementation, transparently activating it if necessary, delivering the request to the object, and returning any response to the caller [Corba].

In addition to the ORB, CORBA defines what are known as object frameworks: object services (application-independent services at the system level such as naming, trading, or security services), common facilities (application-independent services at the application level such as printing services), and domain facilities (services for specific application fields).

CORBA has gained much momentum in industry and research. Implementations of the standard are available from a large number of vendors and even exist as freeware [PR 01]. CORBA supports all major programming languages, is suited for almost every combination of hardware and operating system, and is also being used to realize mission-critical applications in industries as diverse as health care, telecommunications, banking, and manufacturing.

While in the past almost all additions to the CORBA specification have been integrated by vendors into their products over time, this will probably become increasingly difficult in the future. The recent CORBA-3 specification is huge and includes the following major additions: Internet integration (CORBA through firewalls, URL addressing for CORBA objects), quality of service control (fault tolerance, real-time), and a component model similar to EJB (Enterprise Java Beans). It is questionable whether CORBA and in particular implementations of the standard can, in the long run, adopt all these and other foreseeable developments in the field of distributed systems.

Furthermore, CORBA was conceived for static distributed systems, requires considerable resources at run time, and uses the traditional client-server model as the basic metaphor. It is therefore not well suited for small devices, highly dynamic systems, and services that are spontaneously integrated into a federation. This, however, is a major trend in distributed systems, to which Jini and similar systems are better adapted.

3.4 Jini

Jini is an infrastructure that runs on top of Java and RMI to create a federation of devices and software components implementing services. Jini enables any device that can run a Java Virtual Machine to interoperate with others by offering and using services. A service is defined as an entity that can be used by a person, a program, or another service [darm]. Typical examples of services are printing a document or translating from one data format to another, but functional hardware devices are also considered to be services.

All services are granted as leases. Each service, when in use, is registered as being leased by another service. Leases are time-dependent and have to be renewed upon expiration. If the lease is not renewed, then Jini removes the service from the list of services offered.

A device or a service uses a standard mechanism to register with the federation. First, it polls the local network to locate a so-called lookup service. Then, it registers itself with the lookup service. The lookup service is similar to a bulletin board for all services in the federation. It can store not only pointers to the services, but also the code for these services or proxies representing the service, as well as defining service characteristics and attributes (e.g. a printer may specify whether it supports color printing) [jini].

When a client wants to use a service offered to the community, it can download the required proxy object from the lookup service (after having located and contacted the lookup), including any code such as device drivers or user interfaces. Dynamic code mobility enables clients to take advantage of services without pre-installing or loading drivers. The downloaded proxy object can then be used locally to interact directly with the selected device or service, with any device-specific details being hidden by the proxy.

Jini and similar infrastructures (e.g. Universal Plug and Play or UPnP, which is a service discovery framework at a somewhat lower level than Jini) are thus well-suited for highly dynamic distributed systems where components may be mobile and typically form spontaneous networks – an important trend in distributed systems [Ker 00].

3.5 Bridges Between Different Middleware Systems

Each middleware system presented in the previous sections is supported by strong commercial forces. Together with the inertia of existing legacy software – all closely tied to one of these infrastructures (especially in the case of Microsoft’s COM+ platform) – it is unlikely that any one of them will surpass its competitors. For this reason, bridges have been defined and standardized for the most widely used middleware platforms.

One such bridge mediates between the two major middleware parties CORBA and COM+. It is derived from the standard IOP protocol (Inter-ORB protocol) that allows basic interaction between CORBA objects served by ORBs from different vendors. Because CORBA-1 compliant ORBs were not able to talk to ORBs from other vendors, this Inter-ORB protocol was a major improvement in CORBA-2. It still allows proprietary protocols between two or more ORBs to exist, but requires a General Inter-ORB protocol (GIOP) to be understood by any CORBA-2 compliant ORB [GIOP]. This abstract protocol defines the syntax and semantics of messages to allow independently developed ORBs to communicate over any connection-oriented transport protocol. Two instances of this GIOP exist: the IIOP (Internet Inter-ORB protocol) implements GIOP over TCP streams, and the DCE-IOP, which forms the basis for the interaction between COM+ components and CORBA objects [DCE-IOP].

The Java platform tightly integrates CORBA through its org.omg.CORBA package, a library with more than 100 classes that can interact with CORBA ORBs and CORBA objects [j2ee]. These classes represent the mapping between Java and the interface of a CORBA object defined in the CORBA IDL. The package also comprises a Java implementation of an ORB.

Within the Microsoft middleware, bridges are also defined to provide backward compatibility. COM+ components written in languages such as Visual Basic or C# are now compiled to and thus accessible in the .NET environment. Moreover, any managed component (code that is based on the Common Type System) can directly be used as a COM+ component. For COM+ components implemented in unmanaged programming languages such as C++, appropriate .NET wrappers can be generated automatically if type libraries for these components exist. With the aid of these wrappers, unmanaged components are accessible within .NET-compliant applications and vice versa. In the open source community there is also a project called MONO [MONO] which aims at creating a .NET conforming platform for the Linux operating system.

Basic obstacles in bridging middleware systems are substantial performance penalties due to additional marshalling costs, friction losses because of incompatible models, and limitations in the expressiveness of the involved IDLs. The mutual sympathies of the participating companies and their licensing plans as well as their patent politics also have a strong influence on the proliferation of bridges between different middleware platforms. Currently, most of the bridges have been used in simple applications only as a proof of concept, and serious commercial usage seems to be rare.

4. Trends in Distributed Application Structures

Dramatic changes at the application level accompany the evolution of distributed middleware. In e-commerce and cooperative business, the Web and its underlying HTTP protocol are becoming the standard execution platform for distributed and component-based applications. The increasing number of computers and users on the Internet has led to new cooperation structures such as peer-to-peer computing, an interesting exercise in scalability with almost political implications for the networked society. It has also stimulated new developments in the area of computing clusters and so-called grids. Independent of the application domain, the integration of mobile clients into a distributed environment and the ad-hoc networking of dynamic components are becoming ever more important.

4.1 Application Servers

The client-server model is still the prevalent pattern of communication in distributed systems. But programmed client-server systems based on a traditional RPC scheme are losing ground in favor of Web-based communication between clients and servers. Enhanced HTTP servers receive client requests (typically a URL get request) and are capable of performing the necessary computation to satisfy the client request and replying with a dynamically generated HTML document. A variety of technologies are available for servers as well as for clients. Client-side technologies primarily deal with user interface issues.

Server-side technologies, on the other hand, define how application code gets executed upon the arrival of a client request. Heavyweight mechanisms such as CGI (Common Gateway Interface) and ASP (Active Server Pages, [ASP]) are widely used where scripts or whole executables are started on receipt of each incoming request. However, besides performance penalties, these mechanisms raise a number of problems regarding security, authentication, and state management.

With respect to application architectures, a shift from 2-tier solutions with front-end clients and back-end servers (legacy programs or database platforms) towards 3-tier structures can be observed. In a 3-tier client-server system, functional issues (what is known as business logic) are separated from the issues concerning data modeling and persistent storage. The latter are still realized by traditional database systems, but the business logic is typically executed on a so-called application server (middle-tier), a specialized platform for the execution and management of application components. Typically, this application server is tightly coupled with a Web server, and client requests are delivered directly to the responsible business object. Two opposing approaches on modeling, deploying, and managing business logic exist today – the Enterprise Java Beans (EJB, [EJB]) standard defined on top of the Java platform, and the COM+ component model extended with the Microsoft Transaction Server (MTS).

The EJB standard defines two types of beans, where a bean is a specialized Java class that conforms to certain rules in order to reflect the class structure. Entity beans represent data objects inside the application server. They are instantiated when they are first referenced by some other component. Two types of instantiation are possible: the data members of the object are retrieved automatically from databases using the Java database connectivity mechanism JDBC (container-managed persistency) or manually by any application-defined mechanism (bean-managed persistency). Container-managed persistency can be used as an automated migration of a database-centric system structure to a more modern object-oriented application architecture. The bean-managed approach may serve as an object-oriented wrapper for legacy applications. The flow of control is modeled in EJB by so-called session beans. Session beans come in a stateless and a stateful flavor. A stateful session bean stores conversational state encompassing successive client requests only inside the application server. It is never stored persistently. A similar architecture is used with COM+ and MTS, the Windows counterpart of the application server. In this environment, COM+ components implementing specific interfaces correspond to entity beans or session beans.

In component-based application architectures, a tendency to distinguish between functional and non-functional aspects can be observed. Entity components and session components are still being developed with traditional programming languages such as C, C++, Java, or even Visual Basic, for implementing the functional part. But non-functional issues such as transactional scope, role-based security mechanisms, connection and object pooling (reusing connection and objects for independent requests), as well as load balancing are not being implemented as part of a component anymore; instead, each component is being augmented by supplementary attributes. In the case of COM+, some of these attributes are defined as part of the source code or the interface definition using a special syntax. In the EJB context, all attributes are defined in a dedicated XML manifest file that is part of the bean's jar file. Attributes are recognized and implemented by the application server together with specific wrappers that surround each component (containers in the context of EJB or so-called interceptors in the Windows environment). By these means, complex applications can be managed on a more abstract and descriptive level without the need for specific programming skills. For this purpose, dedicated management consoles are used by application administrators that provide access to all non-functional aspects of an application.

4.2 Peer-to-Peer Computing

The Internet boom in the late 1990s led to an unexpected explosion in the number of private computers connected to the global network. According to [isc], a rough estimate for the number of hosts connected to the Internet in January 2002 was 147 million (hosts advertised in the DNS). The major technical challenge for the Internet and its protocol suite IPv4 was to keep up with this growing number of hosts and users. One specific risk was that of running out of IP addresses. In particular, a shortage of class B networks and the small size of class C networks have led to a number of competing enhancements to the existing addressing schemes of IPv4.

Besides classless Internet domain routing (CIDR, [CIDR]) and network address translation (NAT, [NAT]), a widely-used approach to remedy this situation is based on the assumption that most private computers are connected to the Internet only sporadically. Therefore, no static IP addresses are assigned to these hosts anymore. Instead, the Internet provider uses a fixed contingent of addresses and assigns a dynamic IP address temporarily whilst one of these private hosts is connected to the Internet. As a consequence, the majority of computers in the Internet cannot be identified by unambiguous IP addresses anymore, since these addresses change frequently over time.

The millions of new users mostly had rather different expectations about the network and its opportunities. This new community did not fit into the traditional client-server scheme where only a minority of servers stored most of the information accessed by the majority of clients. Many of the new users were primarily interested in exchanging information with each other, be it mp3 audio files, movies, or more questionable things such as license keys, operating systems, application programs, and cracked games.

This shift in the perception of the Internet and its functionality has given rise to the peer-to-peer (P2P) movement [P2P 01]. Today, P2P comprises a variety of different application domains, but the above-mentioned exchange of data amongst peers is the most prominent one. Technically, programs for this purpose are complicated due to scalability issues and because of the dynamic IP addresses of most of their clients. Basically, these systems store only meta-information about the documents available in the P2P network, the data itself being kept by the peers and downloaded directly by the client (disregarding the fact that the widespread use of asymmetric DSL modems – a remnant of the earlier client-server structure – leads to weak uplinks when peers act as servers). Due to legal and copyright issues, storing this meta-information in a single place, e.g. by dedicated servers, can be prohibitive. Napster [naps], one of the first P2P networks, implemented this approach and became much less attractive due to several lost lawsuits. Other P2P networks such as Gnutella [gnut] and its clients (e.g. BearShare [bear], Morpheus [morph], and LimeWire [lime]) bypass this single point of storage by circulating the meta-information within the overlay network itself. By these means, however, they introduce a huge additional network load.

The peer-to-peer movement also has a political dimension [P2P 01]. The classical client-server structured Internet can be viewed as a political system where a small number of residents define the rules for the masses. In this respect, peer-to-peer is sometimes interpreted as a process of democratization. This is also one of the reasons why the Open Source community has strong relations with the P2P movement. A number of projects emerged from this political motivation. One driving force behind these projects is the impetus to provide anonymity for clients accessing sensitive information, to protect against governmental supervision, to prohibit censorship, and to protect privacy.

4.3 Grid Computing

The latent computing power of the Internet is vast, and with the ever-increasing quality and bandwidth of network technology it is possible to tap some of this potential. Unlimited computing power is needed in many application domains, especially in research disciplines such as physics, chemistry, bio-chemistry, astronomy, and meteorology, where many simulations are computationally expensive. Many of these simulations have previously been performed on vector supercomputers, but the availability of high-performance networks has led to a shift towards arrays of inexpensive personal computers and workstations [FK 98, grid]. In the Linux environment, there is even a competition for the largest number of Linux boxes connected to form a single virtual supercomputer. Impressive systems are also commercially available – not for everyone's wallet – with more than 8000 processors [top].

Because of the quality of the network connection, computer clusters with an increasing diameter can be deployed successfully, although the vision of a global computer grid comprising most of the hosts available in the Internet is still several years away. The project coming closest to this vision is seti@home [seti]. Technically, seti@home is a client-server based loosely-coupled cluster system, where any client computer can download a set of radio data. Clients perform a Fourier analysis during idle times to look for signals of artificial origin. The results are then sent back to the server system. The total statistics of this project as of March 23, 2002 are quite impressive: more than 3.6 million clients have contributed to seti@home until now, 1.42E+21 floating point operations have been performed, and a total computation time of 921,190 CPU years has been spent so far.

A number of middleware projects such as JXTA [jxta], Globus [glob], and Legion [leg] aim at providing a virtual platform for the execution of distributed applications on such grid computer systems. Most of them are based on early middleware for cluster computers such as PVM [pvm] and MPI [mpi]. In the latest middleware systems, emphasis is put on the establishment and maintenance of a computer grid, the ease of communication between heterogeneous computer systems, and enhanced communication patterns essential to vector-based mathematics.

4.4 Mobility

People have an increasing desire for ubiquitous access to information, anywhere, anyplace, and anytime. For that, they need not only mobile and portable devices, but also adequate communication systems and software infrastructures.

Mobile devices in the form of portable telephones, PDAs, and notebook computers are now commonplace. Technologies such as WAP, imode, GSM, and in particular UMTS and similar so-called third generation cellular communication standards will soon give rise to new mobile devices providing fast and immediate (i.e. “always connected”) access to the Internet.

However, mobile devices are currently poorly integrated. One example is data synchronization: since from the mobile worker perspective it is crucial that data (such as phone numbers and calendar information) remains consistent across the various devices, automatic synchronization is a necessity. Current synchronization software consists of proprietary products that only allow synchronizing between specific devices and applications. The trend here is moving towards standards (such as SyncML, propagated by an industry consortium) and more general synchronization middleware [HMNS 00].

Another infrastructure problem is transparent roaming. Although protocols and systems such as mobile IP provide users the freedom to roam beyond their home subnet whilst consistently maintaining their home IP address, this is not as simple as roaming in cellular phone networks and has several drawbacks with respect to efficiency and scalability.

It is not only people and computing devices that can be mobile, but also program code. Mobile code is executable program code that moves from a source machine to a target machine where it is executed. Mobile code may help to support user mobility: personalized environments can follow a user between computers [cam]. Platform independence of mobile code is usually achieved by using scripting languages for which interpreters are available on most systems, or by compiling into some platform-independent representation such as Java bytecode.

Mobile code is an important programming paradigm and opens up new possibilities for structuring distributed software systems in an open and dynamically changing environment. It can improve speed, flexibility, structure, and the ability to handle disconnections, and it is particularly well-suited if adaptability and flexibility are among the main application requirements. It has applications in many areas, such as mobile computing, active networks, network management, resource discovery, software dissemination and configuration, electronic commerce, and information harvesting [KM 00].

Java applets are a prime example of mobile code components. Applets are best known as small Java programs, embedded in a Web page, that can be executed within the Web browser. However, applets together with the ubiquitous availability of the Java Virtual Machine, Java’s class loading mechanism, its code serialization feature, and RMI make Java a full mobile code system where arbitrary code can be downloaded over the network and executed locally. Of course, security is a major concern in this context.

A more elaborate form of mobile code, based on the “push principle” as opposed to the “pull principle” of mere code downloading, is that of mobile agents [KM 00]. They consist of self-contained software processes which can autonomously migrate from one host to another during their execution. In contrast to simple mobile code systems, mobile agents have navigational autonomy, they decide on their own (based on their programmed strategy and the current state of the context) whether and when they want to migrate. While roaming the Internet or a proprietary intranet and visiting other machines, they do some useful work on behalf of their owners or originators.

Compared to traditional distributed computing schemes, mobile agents promise, at least in some cases, to cope more efficiently and elegantly with a dynamic, heterogeneous, and open environment which is characteristic of today's Internet. Certainly, electronic commerce is one of the most attractive areas in this respect: a mobile agent may act (on behalf of a user or owner) as a seller, buyer, or trader of goods, services, and information. Accordingly, mobile agents may go on a shopping tour of the Internet – they may locate the best or cheapest offerings on Web servers, and when equipped with a negotiation strategy (i.e. if they are "intelligent agents") they may even carry out business transactions on behalf of their owners [FM 99].

Another general application domain is searching for information on the Internet or information retrieval in large remote databases when queries cannot be anticipated. Other uses of mobile agent technology include monitoring, remote diagnosis, groupware applications, and entertainment.

In general, mobile agents seem to be a promising technology for the emerging open Internet-based service market. They are well-suited for the personalization of services, and dynamic code installation by agents is an elegant means of extending the functionality of existing devices and systems. Agent technology therefore enables the rapid deployment of new and value-added services. Furthermore, mobile code and mobile agents are of interest for future Ubiquitous Computing applications, where small mobile devices may be "spontaneously" updated with new functionality or context-dependent program code.

Some important problems remain to be solved, however. The most important issues are probably security concerns: protecting hosts from malicious agents, but more crucially also protecting agents and agent-based applications from malicious hosts. The second issue is crucial for applications such as electronic commerce in an open world, but unfortunately it is difficult to tackle. The main point is that, as an agent traverses multiple hosts which are trusted to different degrees, its state can be changed by its temporary hosts in ways that adversely impact its functionality [nist]. Transactional semantics for migration (i.e. "exactly-once migration" in the event of communication failures), interoperability with other systems, coordination issues, and the management of large societies of mobile agents also still pose non-trivial challenges. Furthermore, a seamless integration of mobile agents into the Web environment is crucial for the success of mobile agent technology.

4.5 Spontaneous Networking

Device mobility and the emergence of information appliances are spurring on a new form of networking: unmanaged, dynamic networks of devices, especially mobile devices, which spontaneously and unpredictably join and leave the network. Underlying network technologies already exist – for example, Bluetooth. Consumers will expect these ad hoc, peer-to-peer networks to automatically form within the home, in networked vehicles, in office buildings, and in various arbitrary environments [IBM].

In such environments, the ability to dynamically discover devices and services ("service discovery") is a key component for making these networks useful [ibm]. Service discovery protocols provide a standard method for applications, services, and devices to describe and to advertise their capabilities to other applications, services, and devices, and to discover their capabilities. Such protocols also enable them to search other entities for a particular capability, and to request and establish interoperable sessions with these devices to utilize those capabilities [eet].

The most important technologies to date for service discovery are Jini (as described in section 3.4), Salutation, Universal Plug and Play (UPnP) from Microsoft, E-speak from Hewlett-Packard, and the Service Location Protocol (SLP), which has been jointly developed by researchers from both academia and industry as a widely accepted and usable Internet standard for service discovery.

However, a crucial point is service mediation: matching requests for services with service descriptions. This is not as easy as it seems (e.g. does or should a request for a 300 dpi printer match a 600 dpi printer?), and designing a framework for service types is an especially difficult problem for open, dynamic networks, in which the communicating parties may have never encountered each other before, and cannot assume shared code or architectures [ncsa]. Technologies such as XML may provide at least a syntactical basis for that problem. Another technology currently being developed by the World Wide Web Consortium (W3C) is the Resource Description Framework (RDF), which provides interoperability between applications that exchange machine-understandable information. However, none of these technologies addresses the underlying fundamental conceptual and semantic problem of service mediation, which remains an important open issue.

5. Towards Ubiquitous Computing

Given the continuing technical progress in computing and communication, it seems that we are heading towards an all-encompassing use of networks and computing power, a new era commonly termed "Ubiquitous Computing". Its vision is grounded in the firm belief amongst the scientific community that Moore's Law (i.e. the observation that the computer power available on a chip approximately doubles every eighteen months) will hold true for at least another 15 years. This means that in the next few years, microprocessors will become so small and inexpensive that they can be embedded in almost everything – not only electrical devices, cars, household appliances, toys, and tools, but also such mundane things as pencils (e.g. to digitize everything we draw) and clothes. All these devices will be interwoven and connected together by wireless networks. In fact, technology is expected to make further dramatic improvements, which means that eventually billions of tiny and mobile processors will occupy the environment and be incorporated into many objects of the physical world.

Portable and wireless Internet information appliances are already now a hot topic in the computer industry. Soon everything from laptops and palmtops to electronic books, from cars and telephones to pagers, will access Internet services to accomplish user tasks, even though their users may have no idea that such access is taking place [Kot]. It is clear that today's mobile phones and PDAs, connected to the Internet, are only the first precursors of completely new devices and services that will emerge. This will give rise to many interesting new applications and business opportunities.

5.1 Smart Devices

Progress in technologies for sensors (and thus the ability to sense the environment), together with the expected increase in processing power and memory, will render classical devices or everyday objects "smart" – they may adapt to the environment and provide useful services in addition to their original purpose. Most of these new emerging "smart devices" will be small and therefore highly mobile; some might even be wearable and be worn much as eyeglasses are worn today. They will be equipped with spontaneous network capabilities and thus have access to any information or provide access to any service "on the net". Connected together and exchanging appropriate information, they will form powerful systems.

Future smart devices will come in various shapes and sizes and will be designed for various task-specific purposes. They all have in common the fact that they are equipped with embedded microprocessors and are connected (usually by wireless means) to other smart devices or directly to the Internet. Some of these devices may also be equipped with appropriate task-specific sensors. Others, known as "information appliances", will allow users to gain direct and simple access to both relevant information (e.g. daily news and email) and services [Nor 98] [ibm]. Their user interface may be based on speech recognition, gesture recognition, or some other advanced natural input mode technology that is appropriate for their purpose, size, and shape. Multimodal human-computer interaction techniques will also help to identify people and thus protect access to the device. All these devices will be so highly optimized to particular tasks that they will blend into the world and require little technical knowledge on the part of their users – they will be as simple to use as calculators, telephones or toasters [ibm].

Extremely flat and cheap screens that can be fixed to walls, doors and desks are conceivable. The displays could be configured to present information such as weather, traffic, stock quotes or sports results extracted from the Web. Once configured, users could place these displays wherever they felt it was convenient. As humans, we are accustomed to looking in particular places for particular pieces of information [MIT b]. This way, dynamic information would become much easier to find and assimilate – a user might, for example, place today's weather forecast on the wardrobe door.

Smart toys are another appealing prospect. Compared to an ordinary toy, a networked toy would have access to a huge world of information and could be more responsive to its owner and environment. For example, a toy that teaches spelling could access a large dictionary. It could also invoke a speech recognition process running on a remote computer to convert a child's story into text. Or it might know the current weather and other global or local news and events. A toy (such as a smart teddy bear) might also act as a telecommunication device or even a telepresence device and serve as an avatar for the friends and family of the toy owner [MIT c].

Of course, many other types of smart devices are conceivable. Wearable computing devices will be used to keep people informed, connected, and entertained. Just like the carriage clock of 300 years ago that subsequently became a pocket watch and then a wristwatch, personal electronic devices will become items that can be worn as clothing, jewelry, and accessories [phil]. Wearable electronics may even become a new clothing concept: textiles that are electrically conductive but also soft and warm to touch exist already. As a result, it is relatively easy to move audio, data, and power around a garment. Conductive fibers can be integrated into woven materials, and conductive inks allow electrically active patterns to be printed directly onto fabrics.

One of the unique aspects of mobile devices is that they can have an awareness of the location where they are used. However, location-awareness is only one aspect of context-awareness as the encompassing concept, which describes the ability of a device or program to sense, react to, or adapt to the environment in which it is running. Context-awareness enables new applications based on the special nature of the context, for example interactive guide maps. However, it may also be exploited in determining the form of interaction supported and modifying interface behavior. For example in a car system, unimportant feedback may be limited during periods of rapid maneuvering [RCDD 98].

A dominant constraint for many information appliances will be their power consumption. If we assume a future where many task-specific devices exist instead of few general purpose machines, clearly users will not be interested in charging or replacing dozens of batteries. For mobile and wearable devices, however, only one of the devices (e.g. a mobile phone) that a user carries will need to communicate with wide-area networks. Other, more power-thrifty personal devices may communicate over a link that only covers a person's immediate space [wash b].

Prototypes of such personal area networks already exist. IBM has developed a technology that uses a tiny electrical current to transmit information through the body of a person [IBM b]. In this way, a user's identification and other information can be transmitted from one person to another, or even to a variety of everyday objects such as cars, public telephones and ATMs. The bandwidth is relatively small, but more than enough to carry identification, or medical information [IBM c].

Networked embedded processors, which form the heart of all smart devices, will become an important research and development field. New types of low-power processors will be developed specifically for networked embedded applications. Also of primary interest are advances in networking technology that could allow large numbers of embedded computers to be interconnected so they can share information and work together as part of a larger system. Reliability is crucial in embedded computing systems, since such systems will increasingly control critical functions where safety is a factor (e.g. braking, navigation, driving) and, in some applications, may be given authority to take some actions with little or no human intervention. Ensuring the reliability of networked embedded computing systems could be difficult since large, interconnected information systems are notorious for becoming unstable. Such problems could be magnified by the presence of millions of interconnected embedded systems [NRC].

Progress in material science, chemistry, and physics will eventually change the appearance of information appliances. For example, light emitting polymer (LEP) displays that are now available in first prototypes offer many processing advantages, including the possibility of making flexible large-area or curved displays capable of delivering high-resolution video-rate images at low power consumption, visible in daylight and with wide viewing angles [cdt].

Somewhat speculative are techniques known as “electronic ink” or “electronic paper”. Although initial prototypes exist, there is still a long way to go before paper can be replaced by electronic versions. However, the impact of this technology, once it is available, is significant: just imagine carrying your calendar and contact list on a foldable piece of electronic paper, or pulling out a large screen from a mobile phone or a tubular scroll containing the remaining electronics of a PC! Combined with small GPS receivers, maps that display their exact location (“you are here”) are a real possibility.

5.2 Remote Identification

One of the major problems in Ubiquitous Computing is the identification of objects [HMNS 00]. For retail-based applications, barcode labels are typically used, but these have a number of drawbacks (such as the visibility of the barcode and the fact that it is a read-only item). So-called smart labels or RFID tags represent a newer and more interesting concept for identifying objects [Fin 99].

A smart label is a small, low-power microchip combined with an antenna, implanted in a physical object. Each label has a unique serial number and can contain other programmable information. The information contained in a label can be transmitted to a nearby reader device by a radio frequency (RF) signal [aim]. Hence, smart labels are contactless, and require neither touch nor line of sight. In particular, they work through plastic, wood, and other materials [MIT d].

Such identification tags can be battery powered or unpowered, and can be manufactured in a variety of sizes and configurations. The simplest give out a single pre-programmed number when placed in the vicinity of a reader [MIT d]. It is also possible to store a limited amount of information on the tags. More elaborate forms of smart labels are contactless smart cards. In addition to storage memory, they contain a processor (with an operating system) and are able to process data (e.g. perform cryptographic functions and thus disclose their state only to authorized entities). Smart card technology is already well-established (smart cards have been used for many years as secure tokens for access control and authentication), but low-power requirements for contactless variants pose new challenges. Research is currently also underway to produce tags that report information about their physical environment, such as temperature or position [MIT d].

Battery-powered labels, known as active tags, can transmit their information over relatively large distances. Their disadvantages are that they have a limited operating life, and are more expensive than passive devices [cwt]. Passive tags, on the other hand, collect the energy they require to operate from the RF field emitted by a reader device, and therefore do not need a battery. They are less expensive (approx. 10 cents to $1), but can only be sensed at distances of up to a few meters. In some of these passive systems, the antenna is replaced by non-metallic, conductive ink. The substrate of the labels is usually paper or plastic, yielding a paper-thin and flexible label, which can be self-adhesive and be printed on. The labels are small enough to be laminated between layers of paper or plastic to produce low-cost, consumable items.

Smart labels are already used to identify packages for express parcel services, airline baggage, and valuable goods in retail. Industry estimates the smart label market will reach 1 billion items in 2003. Future smart labels equipped with sensors and more processing power will be capable of monitoring their environmental conditions, actively sending alerts or messages, and recording a history of status information, location, and measurement data.

Once most products carry an RFID tag (similar to today’s ubiquitous barcode labels), scenarios that go beyond the original purpose of those tags are conceivable. For example, an intelligent refrigerator may make use of the labels attached to bottles, which could be useful for minibars in hotel rooms. Even more intriguing are scenarios where prescriptions and drugs talk to a home medicine cabinet, allowing the cabinet to say which of those items should not be taken together, in order to avoid harmful interactions

[MIT e]. In a similar manner, packaged food could talk to the microwave, enabling the microwave to automatically follow the preparation instructions.

The underlying idea is that everyday objects can, in some sense, become smart by having RFID labels attached to them and by equipping the environment with sensors for those labels. The information on the label would then not be merely an identity, but a URL-like pointer to an infinite amount of information somewhere on the Internet. This means that each object could acquire an electronic identity in addition to its physical structure [MIT f]. These electronic identities might then interact within a virtual world, independently from the physical world. If more and more of the physical world were to become "smart", and both worlds were more closely linked, they would both be richer [hp].

5.3 The Prospects

The ultimate vision of Ubiquitous Computing, where inanimate everyday objects communicate and cooperate, seems to be realizable in the long term. This would then provide us with an unparalleled level of convenience and productivity [MIT e]. Much remains to be done, however, on the conceptual level and quite a considerable infrastructure would have to be implemented before this vision could become a reality. The long-term consequences of a world in which things "talk" to each other are not yet clear, but the prospects are fascinating.

However, there are many concerns and issues to consider, even on the political, legal, and social level. Privacy is of course a primary concern when devices or smart everyday objects can be localized and traced, and when various objects we use daily report their state and sensor information to other objects. Another issue is information overload. Internet users are already overwhelmed by the sheer volume of information available, and the problem will get worse as the Internet grows and we enter the era of Ubiquitous Computing. Search engines, portals, and email filtering are existing technologies that allow the user to reduce the torrent to a manageable stream, but these technologies are still quite limited [Kot]. New technologies and services are definitely required.

It is clear that we are moving only gradually towards the ultimate vision of Ubiquitous Computing. Much progress in computer science, communication engineering, and material science is necessary to render the vision economically feasible and to overcome current technological hurdles such as energy consumption. However, it is also clear that Ubiquitous Computing will, in the long run, have dramatic economic effects: many new services are possible that could transform the huge amount of information gathered by the smart devices into value for the human user, and an entire industry may be set up to establish and run the infrastructure for the new smart and networked information appliances.

Applications for Ubiquitous Computing will certainly be found in areas where the Internet already plays an important role, such as mobile commerce, telematics, and entertainment, but without doubt many other traditional areas (e.g. healthcare and education) and newly emerging areas will benefit from Ubiquitous Computing technologies.

6. Conclusion

Mark Weiser, the pioneer of Ubiquitous Computing, once said "*in the 21st century the technology revolution will move into the everyday, the small and the invisible*". And at the turn of the century Randy Katz, Professor at UC Berkeley, succinctly stated "*as we approach 2001, we are in the Information Age, not in the Space Age.*" Seen that way, the future of distributed systems and Ubiquitous Computing is indeed promising, fascinating, and challenging at the same time!

7. References

[BSL 00] D. Box, A. Skonnard, J. Lam: Essential XML – Beyond Markup, Addison-Wesley, 2000

[CDK 00] G. Coulouris, J. Dollimore, T. Kindberg: Distributed Systems – Concepts and Design, 3rd Edition, Addison-Wesley, ISBN 0-201-62433-8, 2000

[EE 99] G. Eddon, H. Eddon: Inside COM+ Base Services, Microsoft Press, 1999

[Fin 99] K. Finkenzeller: RFID-Handbook, Wiley, ISBN 0-471-98851-0, 1999

[FK 98] I. Foster, C. Kesselman (Editors): The Grid: Blueprint for a New Computing Infrastructure, Morgan Kaufmann, 1998

[FM 99] S. Fünfrocken, F. Mattern: Mobile Agents as an Architectural Concept for Internet-based Distributed Applications – the WASP Project Approach, in: Steinmetz (ed.) Proc. KiVS'99, Springer, pp. 32-43, 1999

[HMNS 00] U. Hansmann, L. Merk, M.S. Nicklous, T. Stober: Pervasive Computing Handbook, Springer, ISBN 3-540-67122-6, Oct. 2000

[Ker 00] R. Kehr: Spontane Vernetzung, Informatik-Spektrum, Volume 23, Issue 3, pp. 161-172, 2000

[KG 99] D. Kotz, R. Gray: Mobile Code – the Future of the Internet, Third International Conference on Autonomous Agents, Seattle, 1999

[KM 00] D. Kotz, F. Mattern (Eds.): Agent Systems, Mobile Agents, and Applications, Springer-Verlag, ISBN 3-540-41052-X, 2000

[McL 01] B. McLaughlin: Java & XML, O'Reilly, 2nd Edition, 2001

[Nor 98] D.A. Norman: The Invisible Computer, MIT Press, ISBN 0262140659, 1998

[NRC] National Research Council: Study Committee on Networked Embedded Computers

[P2P 01] A. Oram (Editor): Peer-to-Peer – Harnessing the Power of Disruptive Technologies, O'Reilly, 2001

[PR 01] A. Puder, K. Römer: MICO – MICO is CORBA, 2nd Edition, dpunkt Verlag, 2001

[RCDD 98] T. Roden, K. Chervest, N. Davies, A. Dix: Exploiting Context in HCI Design for Mobile Systems, First Workshop on HCI for Mobile Devices, 1998

[Ric 02] J. Richter: Applied Microsoft .NET Framework Programming, Microsoft Press, 2002

[TL 02] T. Thai, H.Q. Lam: .NET Framework Essentials, O'Reilly, 2nd Edition, 2002

[Wei 91] M. Weiser: The Computer for the 21st Century, Scientific American, September 1991, pp. 94-10

Quotations and other text fragments come from several Web resources. Although not all of them could be identified in retrospect, the most important are:

[aim] www.aimglobal.org/technologies/rfid/resources/papers/rfid_basics_primer.htm

[bear] www.bearshare.com/

[ASP] msdn.microsoft.com/library/default.asp?URL=/library/psdk/iisref/aspguide.htm

[cal] www.cs.caltech.edu/~adam/phd/why-events.html

[cam] www.cl.cam.ac.uk/users/dah28/thesis/thesis.html

[carl] www.sce.carleton.ca/courses/94587/Introduction-to-open-distributed-computing.html

[carl b] www.sce.carleton.ca/courses/94580/Introduction-to-open-distributed-computing.html

[cdt] www.cdtltd.co.uk/

[CIDR] ftp://ftp.isi.edu/in-notes/rfc1519.txt

[Corba] www.cs.vu.nl/~eliens/online/tutorials/corba/overview.html

[cwt] www.cwt.vt.edu/faq/rfid.htm

[darm] www.kom.e-technik.tu-darmstadt.de/acmmm99/ep/montvay/

[DCE] www.opengroup.org/dce/

[DCE-IOP] www.omg.org/docs/formal/01-03-08.pdf

[eet] www.eet.com/story/OEG20000110S0027

[EJB] java.sun.com/j2ee/
[GIOP] www.omg.org/news/whitepapers/iiop.htm
[glob] www.globus.org/
[gnut] www.gnutella.com/
[grid] www.gridcomputing.com/
[hp] www.cooltown.hp.com/papers/webpres/WebPresence.htm
[IBM] www-3.ibm.com/pvc/ in particular:
www-3.ibm.com/pvc/index_noflash.shtml
www-3.ibm.com/pvc/sitemap.shtml
www-3.ibm.com/pvc/pervasive.shtml
www-3.ibm.com/pvc/nethome/
[IBM b] www.research.ibm.com/journal/sj/384/zimmerman.html
[IBM c] www.research.ibm.com/resources/magazine/1997/issue_1/pan197.html
[isc] www.isc.org/ds/
[jini] www.sun.com/jini/factsheet/
[jsol] docs.rinet.ru/JSol/ch16.htm
[j2ee] java.sun.com/j2ee/corba/
[Kot] www.cs.dartmouth.edu/~dfk/papers/kotz:future/
[leg] www.cs.virginia.edu/~legion/
[lime] www.limewire.com/
[MIT] tns-www.lcs.mit.edu/manuals/java-rmi-alpha2/rmi-spec/rmi-intro.doc.html
[MIT b] www.media.mit.edu/pia/Research/AnchoredDisplays/
[MIT c] www.media.mit.edu/~r/academics/PhD/Generals/Hawley.html
[MIT d] www.media.mit.edu/ci/research/whitepaper/ci13.htm
[MIT e] auto-id.mit.edu/index2.html
[MIT f] www.media.mit.edu/~jofish/ieee.paper/ieee.cga.jofish.htm
[MONO] www.go-mono.com/
[morph] www.musiccity.com/
[mpi] www.mpi-forum.org/
[naps] www.napster.com/
[NAT] ftp://ftp.isi.edu/in-notes/rfc3022.txt
[ncsa] www.ncsa.uiuc.edu/People/mcgrath/Discovery/dp.html
[nist] csrc.nist.gov/nissc/1996/papers/NISSC96/paper033/SWARUP96.PDF
[ONC] www.ietf.org/rfc/rfc1831.txt
[phil] www.extra.research.philips.com/password/passw3_4.pdf
[pvm] www.csm.ornl.gov/pvm/pvm_home.html
[seti] setiathome.ssl.berkeley.edu/
[SOAP] www.w3.org/TR/SOAP/
[top] www.top500.org/
[wash] www.cs.washington.edu/homes/abj3938/cse588/MotivatingCORBA.htm
[wash b] portolano.cs.washington.edu/proposal/