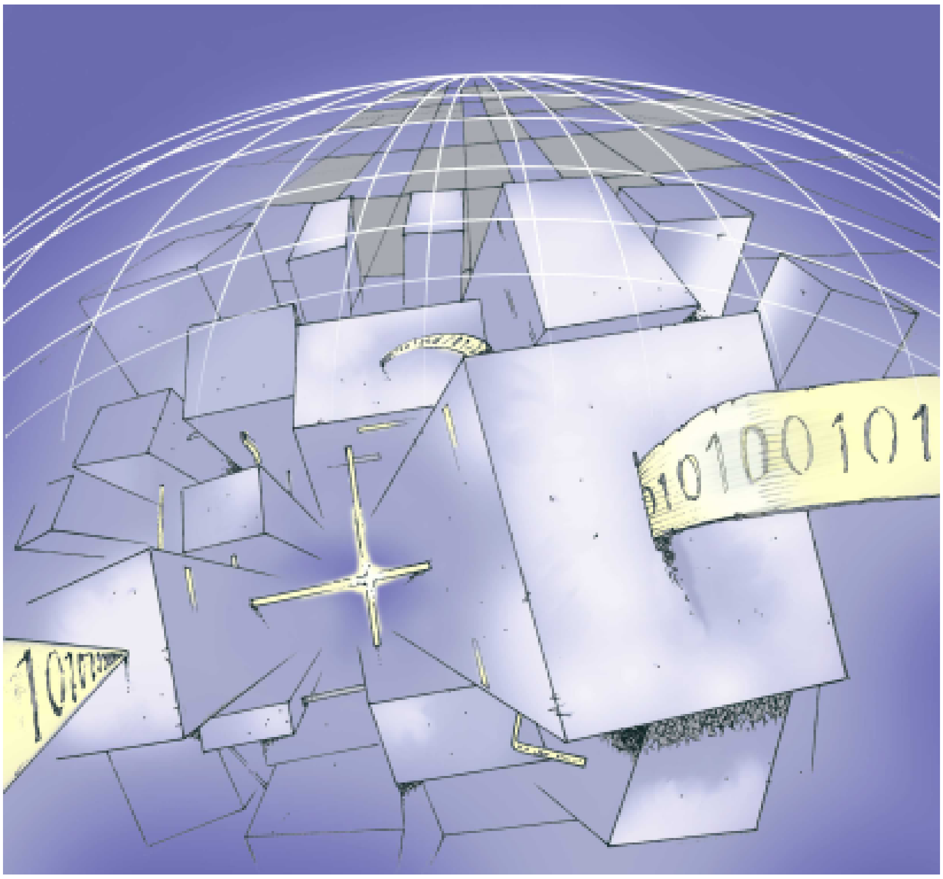
ESPOO 2003 VTT PUBLICATIONS 490



Teemu Vaskivuo

Software architecture for decentralised distribution services in spontaneous networks

VTT PUBLICATIONS 490

Software architecture for decentralised distribution services in spontaneous networks

Teemu Vaskivuo

VTT Elektroniikka

ISBN 951–38–6034–5 (soft back ed.)
ISSN 1235–0621 (soft back ed.)

ISBN 951–38–6035–3 (URL: http://www.inf.vtt.fi/pdf/)
ISSN 1455–0849 (URL: http://www.inf.vtt.fi/pdf/)

JULKAISIJA – UTGIVARE – PUBLISHER

VTT, Vuorimiehentie 5, PL 2000, 02044 VTT
puh. vaihde (09) 4561, faksi (09) 456 4374

VTT, Bergsmansvägen 5, PB 2000, 02044 VTT
tel. växel (09) 4561, fax (09) 456 4374

VTT Technical Research Centre of Finland, Vuorimiehentie 5, P.O.Box 2000, FIN–02044 VTT, Finland
phone internat. + 358 9 4561, fax + 358 9 456 4374

VTT Elektroniikka, Kaitoväylä 1, PL 1100, 90571 OULU
puh. vaihde (08) 551 2111, faksi (08) 551 2320

VTT Elektronik, Kaitoväylä 1, PB 1100, 90571 ULEÅBORG
tel. växel (08) 551 2111, fax (08) 551 2320

VTT Electronics, Kaitoväylä 1, P.O.Box 1100, FIN–90571 OULU, Finland
phone internat. + 358 8 551 2111, fax + 358 8 551 2320

Technical editing Marja Kettunen

Otamedia Oy, Espoo 2003

Vaskivuo, Teemu. Software architecture for decentralised distribution services in spontaneous networks [Ohjelmistoarkkitehtuuri keskittämättömien hajautuspalveluiden ohjelmakehykselle spontaanisti muodostuvissa verkoissa]. Espoo 2003. VTT Publications 490. 99 p.

Keywords distributed systems, ad-hoc networking, decentralised systems, framework, middleware

Abstract

Factors that drive the design of distributed systems are experiencing a phase of rapid changes. Mobility and the new methods of interconnectivity brought along with it have to be faced by the fundamentals of distributed systems. Simultaneously, hardware tasks are being adopted by software, making it possible to make those system elements configurable that have traditionally been considered static. Spontaneous changes in configurations, connections, and physical environment are common factors that are increasingly brought along with the distributed systems design. This thesis considers an architecture for a software framework that faces those challenges by providing interconnectivity for distributed pieces of software in a new way.

The original idea presented here is to create middleware services that arise in a distributed and spontaneous manner from the interconnections of the interconnected, distributed pieces of software themselves. The complete independence of any centralised middleware service producer is the key issue in the proposed solution. Other issues are the means of communication over different media and the ability to assure the robustness of the provided services despite changes in the configuration or the presence of different software elements. The solution has been presented in the form of the software architecture of a proposed design. A major part of the introduced solutions has been validated by distinct cases related to both industry and research.

Vaskivuo, Teemu. Software architecture for decentralised distribution services in spontaneous networks [Ohjelmistoarkkitehtuuri keskittämättömien hajautuspalveluiden ohjelmakehykselle spontaanisti muodostuvissa verkoissa]. Espoo 2003. VTT Publications 490. 99 s.

Avaisanat distributed systems, ad-hoc networking, decentralised systems, framework, middleware

Tiivistelmä

Hajautettujen järjestelmien suunnitteluun vaikuttavat tekijät kokevat tällä hetkellä nopeita muutoksia. Järjestelmien perustoimintojen täytyy ottaa huomioon laitteiden liikkuvuus sekä sen mukanaan tuomat uudenlaiset yhteysmuodot. Aiemmin laitteistolähtöisesti ratkaistuja tehtäviä toteutetaan yhä enenevissä määrin ohjelmistolla, minkä ansiosta useita kiinteiksi käsitettyjä tekijöitä voidaan nykyisin pitää muunneltavina. Hajautettujen järjestelmien suunnittelussa täytyy uudenlaisten vaatimusten mukaisesti ottaa yhä useammin huomioon spontaanit muutokset ohjelmiston kokoonpanossa, kytkennöissä, sekä ohjelmistoa suorittavan laitteen fyysisessä ympäristössä. Tässä diplomityössä käsitellään uutta hajautettujen järjestelmien ohjelmistokehyksen ohjelmistoarkkitehtuuria, joka kohtaa spontaanin ympäristön asettamia haasteita tarjoten uudenlaista ratkaisua hajautettujen ohjelmiston osien yhteistoiminnalle.

Tässä työssä esitettävä alkuperäinen ajatus on luoda yhteen liitettyjen, hajautettujen ohjelmiston osien pelkästä yhteen liittämisestä syntyvä välitason ohjelmistokerros. Avainasia ratkaisussa on sen tarjoama riippumattomuus yhdestäkään keskistetystä välitason palveluiden tuottajasta. Muita työhön liittyviä aiheita ovat ratkaisut ohjelmien osien väliselle yhteydenpidolle eri tiedonsiirron välittäjien kautta, sekä kyky taata tarjottujen palveluiden saatavuus hajautetun järjestelmän ohjelmistojen osien sekä asetuksiltaan että läsnäololtaan vaihtelevasta toiminnasta huolimatta. Ratkaisu on esitetty ehdotetun kaltaista toimintaa toteuttavan ohjelmistoarkkitehtuurin muodossa. Pääosa esitetyistä ratkaisuista on vahvistettu osin tutkimuksessa, osin teollisessa ympäristössä sovellettujen esimerkkitapauksen avulla.

Preface

The basis for the research presented in this thesis emerges from the work done at the VTT Technical Research Centre of Finland by the Software Architectures Group in the research field of Embedded Software and operating unit of VTT Electronics during the years 1999-2002. Contribution to the work has been provided by several research topics in several different projects. The most important of those have been VERSO in 1999, EMU in 2000-2001, ITEA/VHE in 2001-2002, CORAL in 2001-2002 and SAMI within the PLA programme in 2001-2002. Although being disperse in an organisational sense, the conceptual approach has only developed during the process; many valuable viewpoints from several cases of research have been gained along the way.

Several people have made it possible for me to get to this point, looking back to the work that is now done. Of them, I would especially like to mention Research Professor Eila Niemelä for providing me with support and valuable reviews during the process of writing. I would also like to thank Mr. Aki Tikkala, not only for work done together concerning the issues related to this thesis, but also for questioning my approaches in a positive sense, giving me a particular need to use my imagination. Mr. Juhani Latvakoski is acknowledged for reviewing my manuscripts and also for the constructive discussions we have had about many related topics. My supervisor, Professor Juha Röning, deserves my appreciation for his comments and guidance.

During my studies, the growth that has taken place in me has not only been professional or educational. For that, I would like to express my deepest gratitude to my family and to my closest friends. Similarly, I thank all the people involved in my life during these times. You have all been irreplaceable, thank you for making it possible for me to keep the balance between work and joy!

Last but not least, I thank you Saija, for providing me with a chance to think about something other than this thesis or work.

Oulu, November 18, 2002

Teemu Vaskivuo

Contents

Abbreviations

API	Application Programming Interface
ATM	Automatic Teller Machine
COM	Component Object Model
CORBA	Common Object Request Broker
DCOM	Distributed Component Object Model (Distributed COM)
DDP	Dynamic Distribution Platform
DHCP	Dynamic Host Configuration Protocol
DisMis	Distributed Middleware Services
DSL	Digital Subscriber Line
EJB	Enterprise Java Beans
GSM	Global System for Mobile Communications
HTTP	Hypertext Transfer Protocol
IDL	Interface Definition Language
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IP	Internet Protocol

IrDA	Infrared Data Association
ISO	International Organization for Standardization
IT	Information Technology
ITEA	International Technology Education Association
JAR	Java Archive
JDK	Java Development Kit
JPG/JPEG	Joint Photographic Experts Group
JVM	Java Virtual Machine
LAN	Local Area Network
MIDL	Microsoft Interface Definition Language
ODP	Open Distributed Processing
ODP-RM	Open Distributed Processing Reference Model
OMA	Object Management Architecture
OMG	Object Management Group
ORB	Object Request Broker
OSG	Open Services Gateway
OSGi	Open Services Gateway Initiative
PC	Personal Computer

QADA	Quality-driven Architecture Design and quality Analysis
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SOAP	Simple Object Access Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UPnP	Universal Plug and Play
URL	Uniform Resource Locator
VHE	Virtual Home Environment
WLAN	Wireless Local Area Network
XHTML	eXtensible HyperText Markup Language
XML	eXtensible Markup Language

1. Introduction

A spontaneous environment is reality for each computerised system. In such systems, every event that is not endogenous may be considered spontaneous in a sense. User interactions with computers, sudden incoming data from the network or connection to a new device, are occurrences that cannot be anticipated by the computer, or if they can, usually with poor predictability. Systems that operate within a spontaneous environment have to be built so that adaptation to spontaneous actions is possible. This thesis considers the spontaneous actions that take place amongst service elements of distributed peer-to-peer systems. Under consideration is the adaptability required for enabling the necessary operation of a system regardless of the spontaneous characteristics the operating environment might address.

Currently, there are no commonly known solutions for a computing infrastructure of distributed systems that operate in a completely spontaneous environment. In such an environment, there are no reliable sources of services. The main challenge of this thesis is to provide a view to a solution that provides an infrastructure for distributed systems that operate in completely spontaneous environments.

The purpose of this thesis is to present a view on an architectural framework developed for distributed components of distributed applications. Along with the required adaptability, an important issue is the spontaneous environment itself and the requirements and difficulties that come along with it. The practical aim in this thesis is to take a look at the developed DisMis software architecture proposed as a solution to overcome the challenges introduced by those issues.

At least as important as the proposed solution itself, is the survey in which the identified challenges are discussed together with the existing, related solutions and products that are already in common use. The most important features of those existing solutions will be compared to the features of the presented approach. Yet another topic that has been given attention in this thesis is the questions that may arise when representing or planning the architecture of a software system. Accordingly, the definition of software architecture has been given its respective attention.

1.1 Contribution of this thesis

Based on existing and studied architectures and styles of distribution technologies, a new way of building independent and adaptive software components for spontaneous environments is proposed in this thesis. The approach is concretised as the software architecture of a *Distributed Middleware Service Framework*, abbreviated as DisMis framework hereafter. The architectural structures of the DisMis framework are the main contribution of the work presented in this thesis. The proposed approach is discussed and validated by comparing it with a preceding, implemented *Dynamic Distribution Platform* and its software architecture.

The fundamental idea in the presented DisMis framework is to embed all the necessary middleware services and communication protocols in the distinct fragments or components of a distributed application itself. Therefore, the distributed applications or parts of those applications, built by utilising the DisMis framework, will interact and inter-operate by requiring only a known and supported interconnection method enabled between them.

The solution encourages development of peer-to-peer and service-oriented applications that operate by adapting to the concurrent resources and other attributes of the environment that they have available, instead of placing strict requirements for the existence of such attributes.

2. Distributed systems for spontaneous environments

This chapter provides the necessary information that plays a fundamental role in understanding the technical viewpoints and the terminology presented in the later chapters. The discussion has been divided into the conceptual areas of *terminology, heterogeneity, middleware, spontaneous issues, and different entity roles in spontaneous systems.*

2.1 Fundamental definitions

Most computerised, modern information processing systems exhibit various characteristics of a distributed system in a different way and quantity. The term “distributed system” may have a different meaning depending on the background of the person giving the definition. The definition has also varied during the evolution of computer technology. Previously, the definition of a distributed system has often been understood as a multiprocessor arrangement for parallel processing of large, information-intensive tasks. The software related to such systems has been more application specific, hardware related, and protocol oriented than what it is nowadays. At times when computing performance has been a rarer utility, distributed systems have also been considered to be networks that perform collective operations in order to share processing capabilities or peripherals amongst different users, usually by having a central computer that is connected with several client terminals. Nowadays, distributed systems are increasingly understood as peer-to-peer systems where more or less equivalent peer devices share different kinds of services with each other, simultaneously exhibiting less centralised control.

One definition that suits modern distributed systems is given in [1, p. 1]. According to that definition, a distributed system is a system “*in which components located at networked computers communicate and coordinate their actions only by passing messages*”. In this thesis, the definition given in [1, p. 1] is adequate up to some point. Three direct consequences of the presented definition that characterise distributed systems are:

Concurrency: the networked computers may operate concurrently but still access the same resources with certain limitations.

No global synchronisation: the messages are the only way to synchronise operations between different components – there is no global clock implicitly available.

Independent failures: the other components of a distributed system keep on running even if one of the components ceases to operate.

The given definition does not state whether the distributed components of a distributed system are related to hardware or software. Nowadays, it is very common that simple electronic devices also incorporate and execute software. Pure hardware solutions are usually more inflexible and more expensive than *embedded systems* that conservatively embed a dedicated piece of software to execute in a dedicated hardware environment. According to the current trends, increasingly efficient multipurpose hardware processing units offer better changes to move more and more functionality to software that has previously been taken care of by dedicated hardware. That evolution has also led to ambiguity in the definition of the concept of an embedded system.

Pure software-related execution components are possible in virtual execution environments or *virtual machines* like Java [2]. There, software is executed in a software-generated execution environment. Although being conceptually ideal in a sense, the software-generated execution environment is fully dependent on the hardware that it is being executed in. Therefore pure-software components exist only in an abstract sense – they do not have to be aware of the hardware environment underneath. Such independence over hardware platforms is a valuable property for software components in distributed systems. It increases consistency and enables patterns for functions and design that would be difficult to implement otherwise. Transportation of executable programs from one execution environment to another at runtime is one such pattern [3].

The following terms have been used in this thesis in the discussion related to distributed systems:

1) *Execution environments* are logical entities that execute programs. An execution environment may, for example, be a physical processor or a virtual machine.

2) *Devices* are physical constructions that carry the required instrumentation and embody a logical structure required to perform the tasks of at least one execution environment. Additionally, devices have the ability to communicate with other devices through different media – thus connecting different execution environments. A device may also control equipment that has been attached to it.

3) *Programs* are instruction sequences that express a certain task for processing information. Programs are executed in an execution environment.

4) *Components* are entities that are built of certain programming structures and programs. Components have public and private parts – the public parts are visible to other components through an interface. Components are usually for the composition of larger entities like subsystems or systems.

5) *Applications* (or application programs or just systems) are entities dedicated for producing one or several services for a certain user: a person device, program, component, or another application. Applications consist of both static and dynamic behaviour of one or more different components that have been bound together.

6) *Nodes* are conceptual units of deployment in a networked environment. The practical presence of a node may vary. It is usually a networking-capable device executing a certain application that enables the networking.

2.2 Heterogeneity

Distributed computing systems may be heterogeneous in several ways. In general, a heterogeneous system has been composed of separable entities or components that differ from each other in some of their functional properties. In a homogeneous environment, the whole system would therefore be composed of similar elements or components. Both homogeneous and heterogeneous systems exist and are possible in distributed systems. Due to the increasing heterogeneity of devices that have the capabilities to operate as a part of a distributed system, as well as the developing infrastructure of software support for distributed systems and communication means required by them, the evolution is moving increasingly towards heterogeneous distributed systems. It is a fact that cannot be omitted in the design – heterogeneous systems cause many constraints and propose many challenges for the design. Heterogeneity may appear in various levels of functionality, for example as differences in

- networks between execution environments,
- connections between software components,
- operating systems,
- hardware components, operation, and capabilities,
- programming languages,
- data representation, and
- communication protocols.

The differences have to be handled by some means. A common approach is to create abstract representations of the software or hardware mechanisms related to distribution and to make them available for application components. According to that approach, each component may have a uniform view on its environment, independent of the differences that originate from those hardware or software elements that are beyond the direct influence of the component in question. Such abstract representations are created with software. The approach that uses software in abstracting the elements that are related to distributing data and functionality yields the term *middleware*, a producer of such abstractions, also called *transparencies* in several contexts [4, 5, 1, p. 23].

2.3 Middleware

Middleware is a generic keyword when solving the challenges caused by heterogeneous systems. Heterogeneous systems are not, however, the only reason for the presence of middleware. Middleware can also be seen as machinery for performing the abstraction that is often helpful for the understanding of the complex issues related to distributed computing in general. A common conceptual definition for the word sounds somewhat like the following, presented in [6, p. 209]: "*it (middleware) is the glue that holds together disparate systems in a distributed computing environment.*"

Middleware means a software layer that hides the complexity of the distributed environment, thus making many features of the list presented under the headline of heterogeneity in this chapter invisible.

Therefore, middleware is software. In particular, it is software that resides between the operating system and the applications. In case there is no operating system in a system that utilises a middleware solution, middleware operates between the networking hardware and the application. Middleware may be controlled and utilised in many separate ways, depending on its particular implementation. Middleware may appear as a set of software interfaces, a programming model, a software framework or some other means of bringing the services of the middleware within the reach of the application. The way middleware is then utilised is also completely dependent on the particular implementation of the middleware and the application. In some cases, the level of abstraction may be low, while in others it may be high. Other implementations may offer or require a solution or a service that does not exist in others. Examples of such services could be co-ordinated transactions, directory services, discovery services and many others. The definition of the term middleware is not unambiguous. However, the concept is explanatory enough for being as widely used as it is. The term pops up quite often in recent publications that are related to distributed systems.

2.4 Spontaneous networking

Some distributed systems operate in static conditions. Static means here that there are no fundamental changes in the connections, requirements or deployment of the devices that the distributed system is built of. A network of ATMs (Automatic Teller Machine) may be considered an example of a mostly static environment. Adding a new ATM to the network is not a frequent occurrence and in many systems manual configuration is most presumably required.

However, in case of a failure in one of the ATMs in the network, the distributed system should be able to continue its operation. Most distributed systems exhibit at least some dynamic features. For example, in the presented case of a partial failure where only one ATM of the whole network is not functioning, the rest of the system should be able to keep on going. If a system is engineered to operate in increasingly changing conditions, it is considered to be more dynamic.

Some distributed systems are designed to operate under constantly dynamic conditions. In such dynamic distributed systems the configuration of the elements that form the system is not fixed. There are several dimensions in which the configuration may vary:

- interconnections between the execution environments may change,
- new resources may become available,
- existing resources may become unavailable,
- applications that exist in execution environments may change,
- models for information processing may change, and
- models and protocols of information transfer may change.

Small portable devices that have the capabilities to operate as peer members in distributed systems introduce new challenges for the ability to accept and also utilise the dynamics involved in the nature of the use of such devices. Bluetooth [7, 8], WLAN [9, 10] and IrDA [11] are current examples of enabling technologies for close-range spontaneous connections for those kinds of devices. The limited physical range of the mentioned connectivity methods may cause constant connections and disconnections to take place between the devices whenever they are in motion with respect to each other. It is evident that the infrastructure of the distributed information processing of such systems has to be based on completely different principles than that of the ATM example provided.

Connectivity between different applications and devices over more conservative wired networking methods may also require spontaneous operation. In this thesis, any environment with such an unpredictable nature is called a *spontaneous environment*. The following subchapters present concepts that are related to distributed systems that operate in spontaneous environments. Those systems are referred to here as *spontaneous systems*. The formalisation of logical and functional connections between software entities that takes place in spontaneous systems is referred to respectively as *spontaneous networking*.

Spontaneous networking takes place when two or more software entities those have not been previously introduced to each other start communicating. The communication may lead to a long session of interaction, or it may just be an act of introduction. Software entities in this case often reside at distinct devices, but spontaneous networking may also take place inside a device, between different execution environments or just between different software components. The fundamental concept is not and should not be limited to the device boundaries. However, when considering spontaneous networking in distributed systems as presented in this thesis, there are connections between the physical processing units often involved.

In spontaneous systems, an element of unpredictability is always present. Faced with this unpredictability, a system architect has to develop a systematic means of overcoming it when, for example, connecting devices that have no knowledge of each other. Connectivity between different software entities in a spontaneous environment is an important topic in this thesis. There are certain fundamental issues that are often present in systems that exhibit such spontaneous interconnectivity. This subchapter provides a view and analysis of some of those that are considered to be important in the light of this thesis.

2.4.1 Connectivity

A device that is about to participate in spontaneous interactions with other devices has to have the ability to transfer information over a medium that is also understood by other devices. A device is considered here as defined in 2.1. The basic ability that is required is an implemented protocol combined with the required hardware such as a radio transmitter or a connector plug with appropriate wiring. However, the ability to transfer information is not always enough to actually enable the operation of networked software in practice. As an example, a conservative connection to a fixed IP network requires knowledge about a local IP address, default gateway, and a subnet mask. Such configuration is currently very common to every advanced computer user (even though the use of DHCP [12] is nowadays beginning to ease the configuration burden).

When considering spontaneous networking, the connection to the medium should not be complicated. Moreover, there should not be manual intervention or configuration involved at the time the connection takes place. The interconnecting medium should, in practice, be such that the information transfer might take place without any configuration or external information required. This kind of connectivity is considered here as *implicit*. It means that a device or an entity that has the ability for such implicit connectivity, includes all the functionality and information required to transfer information over certain media to other devices or entities. Additionally, the information and functionality should not have to be supplemented by any means at the time the connectivity takes place. There are two ways to achieve such implicit connectivity.

1) The nature of the medium is such that the connectivity within the range is a direct consequence of it. This is true in many cases, such as close-range radio transmitters, broadcasts in IP networks as well as, for example, simple connections with serial or parallel cables.
2) The nature of the medium is such that the connectivity within the range requires additional information, related for example to the address, port, access code, modulation, or frequency, just to mention a few. A TCP connection works as an example. If the connectivity is implicit, the required additional information also has to be provided implicitly.

If comparing the presented concept of implicit connectivity with interactions that take place between people, the case in which a device has the ability to communicate but does not have the required configuration information, is comparable to having a telephone but no phonebook or phone numbers. On the other hand, the implicit connectivity could be compared to the ability to speak – no addressing or phone numbers are required to reach the people within the range. The case where the address or other required information is also provided implicitly is thus comparable to a situation in which the required phone numbers would be available. The phone numbers in that case could, of course, be available in a phone book, or by making a directory inquiry. In that case, the number to the directory inquiry also has to be implicitly available. Figure 1 visualises implicit connectivity with the given real-world examples. The example with the directory inquiry approaches the subject discussed under the next subchapters, namely the steps that may be required to take place in spontaneous networking in order to achieve a spontaneous connection between two separate software entities.

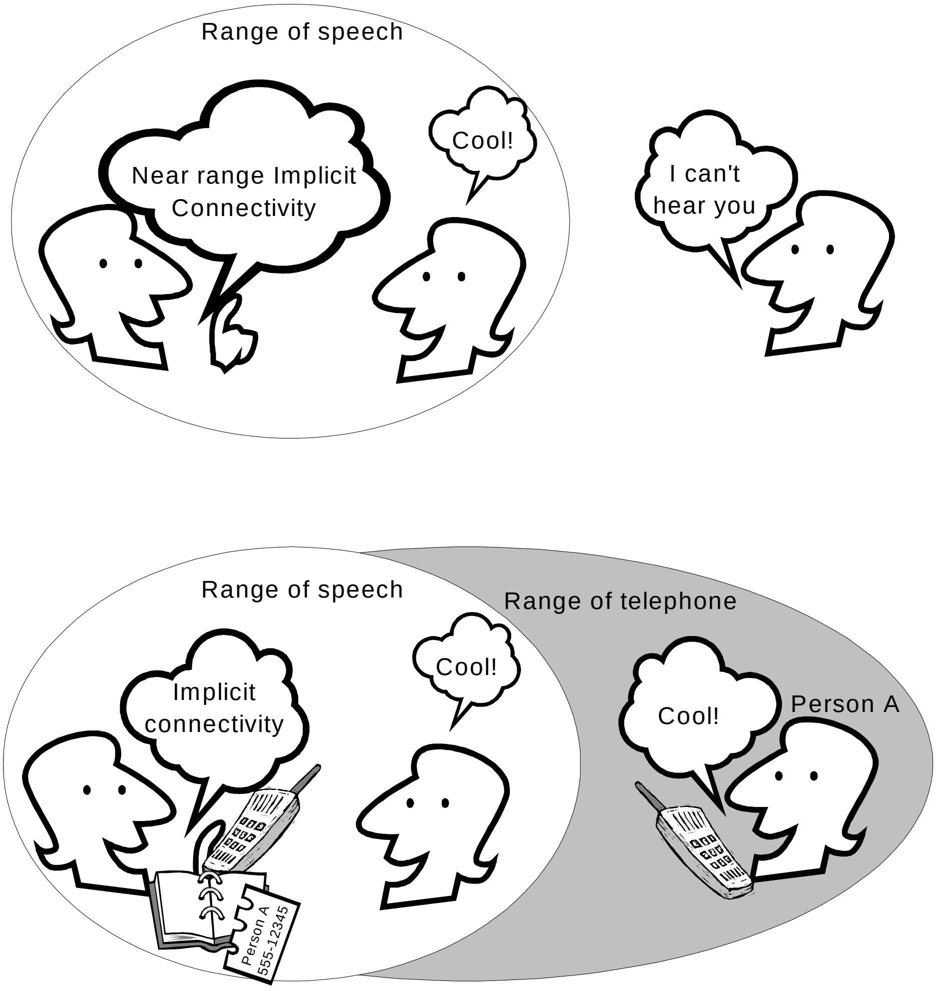


Figure 1. Examples of implicit connectivity.

2.4.2 Discovery

If previously non-introduced devices are about to start communicating, the destination for the first message of the communication cannot be known beforehand. Therefore it is presumed that the devices may send messages in a way that *all* potential participants can receive them. Here, the process that takes place when such previously non-introduced devices establish a connection and achieve a mutual understanding in one way or another is called *discovery*. The definition of the concept of discovery is consistent with the one presented in [13, p. 65] and with slight modifications it suits the one given in [1, p. 43], where an explicit discovery service is considered to comprise both the process of discovery and a directory service. In this thesis, directory services are considered in the next subchapter 2.4.3.

The primary purpose of the discovery process is to make the spontaneously networking devices aware of each other’s capabilities, resources and services. The discovery itself may take place in several ways, depending on the chosen protocol and needs. In practice, according to the technologies that have been considered in this thesis, two common ways of implementing the discovery would seem to be:

1) A beacon signal that is broadcast frequently to a suitable medium. The responsibility to send the beacon signal may vary. It may be random or reasoned by a certain algorithm. New devices wait for that beacon signal when entering a network. If a newcomer does not receive any signals over several periods of a constant frequency at which the message should be transmitted, the new device may start sending the beacon signals by itself.
2) A new device entering the network sends a query or an advertisement to the suitable medium and waits for responses from other devices. There are no frequent messages. In case of no responses, the device just waits for someone to enter the network by listening for the queries or advertisements.

Examples of different discovery procedures will be given in several points later in this thesis:

- Discovery in Jini technology (3.2.2),
- Discovery in UPnP technology (3.3.1),
- Discovery in the presented DDP (Dynamic Distribution Platform) architecture and implementation (4.6.1), and
- Discovery in the DisMis (Distributed Middleware Services) framework architecture (4.4.2).

2.4.3 Directory services

As a result of the discovery process, networked devices, or *network nodes*, and the software entities within them, may sense other nodes and entities in their vicinity (i.e. accessible through a network that provides the required means for information transfer). After a node has gained knowledge about the presence of other nodes, a logical step is the interest towards more specific information about the services and resources that the discovered nodes are able to provide. One solution for a node to gain that information is to transfer the knowledge about a service or a resource within the discovery messages. In large systems, however, that method is not appropriate. Service information may comprise large amounts of data and all the participants in the system should not be forced to store or process that data. In particular, some of the devices in a heterogeneous environment may have very restricted capabilities for doing so. Another approach is to query each particular service or resource only when they are required. That solution has the same fundamental problem. If each service usage in each of the service users would cause a broadcast query for the service, the amount of broadcast messages would explode in large systems, thus making it difficult for the nodes with restricted capabilities to keep up with the pace.

To reduce the global load in a distributed system with spontaneous nature, the information about the service capabilities of the nodes in a network may be gathered to a certain place that may distribute that information further when requested. Such an entity is called a *directory service*. Directory services in general are simply catalogues for more complex data structures than just strings or numbers. They may be used also for other purposes than spontaneous networking. Two generic responsibilities of a directory service may be stated as follows [1, p. 371]:

1) Store collections of bindings between names and attributes.
2) Look up entries that match attribute-based specifications.

In the case of spontaneous networking, directory services store the attributes and values that identify the services, capabilities and resources that exist in the nodes that are connected to a particular network of them. Users of the services and resources may then look up those attributes from a directory service in order to know whether the requested capabilities are available in any of the nodes currently connected to the network.

The following examples of directory services and entities with similar responsibilities are given at a later point in this thesis:

- Look up service in Jini technology (3.2.2),
- Service Registry in the OSGi (3.2.3),
- RMI registry in RMI (3.2.1),
- Dynamic directory service in the DDP platform presented in this thesis (4.6.1), and
- Dynamic directory service in the DisMis platform (4.4.2).

The role of a directory service in a distributed system is fundamental in several applications. There may be also other such important roles: managers for transactions, sources of data, etc. In general, the fundamental organisation of a distributed system can be seen to comprise entities with service provider roles and service user roles, just as for example the roles of a directory service provider and a directory service user. The following subchapter considers them in more detail.

2.4.4 Roles of entities in spontaneous systems

In the discussion of this thesis, spontaneous systems are considered to have two fundamental roles that the entities in the network may express. Those roles are:

- *the role of a* service *provider and*
- *the role of a service user.*

One entity in a network may perform both roles as well as just one of them. One node may also provide many services and accordingly have many distinct service roles, just as it may have many distinct service user roles.

There may be certain fundamental service provider roles in a spontaneous system that some or all of the entities have to fulfil. Consequently, there has to be certain service user roles that have to be fulfilled as well. Those fundamental roles may vary from system to system, but a common feature of those fundamental service user/provider roles is that they participate in the practical operation of the infrastructure of the spontaneous system.

A good example of a fundamental service role from the viewpoint of this thesis is the *directory service provider* role. If the operation of a spontaneous system relies on a directory service as explained, there has to be a directory service provider-role that is implemented by an entity that hosts the directory service. Additionally, there has to be a *directory service user* role that has to be implemented by all of the entities in the spontaneous system, thus making them able to use the services of the directory service. If the use of a directory service is the only means for different entities to come to know about each other's capabilities, it has to be:

- implemented by at least one entity,
- used by all the entities that have service – roles (e.g. service registration) and
- used by all the entities that have service user – roles (e.g. service lookup).

Considering the capabilities used or offered by different entities as roles usually helps in conceptualising the involved complexity. The roles are a tool to see the system as a set of services and their users. The roles may more easily be encapsulated within certain distinct components in the system, as will be shown in practical examples presented in this thesis.

3. Enabling technologies and building blocks

This chapter presents some of currently most popular methods and technologies that are related to building spontaneous distributed systems, even though many of them have been developed to suit more conservative methods of distributed computing or software development in general. The solutions presented in this chapter have been categorised to represent some of the following conceptual classes:

- broker pattern and distributed objects,
- virtual machines and mobile code,
- peer-to-peer connectivity, and
- software architecture.

Independent of the particular technologies, the above categories represent the most important state-of-the-art fields that are related to this thesis. One or several examples or explanations of each of the categorised groups has been provided in the following subchapters.

3.1 Broker-pattern and distributed objects

3.1.1 CORBA - Common Object Request Broker Architecture

Common Object Request Broker Architecture, CORBA, is a part of the OMA, the Object Management Architecture, which is a multi-vendor standard for object-oriented, distributed computing. OMA is consequently produced by OMG, the Object Management Group. Currently OMG is a consortium of 800 companies, making a large contribution to creating standardised technological solutions for the field of distributed software, middleware and component software. OMG also influences several other fields of information technology and standardising committees [14, 15, pp. 1–13].

CORBA is based on the idea of object interoperability. According to it, all the services an object provides are expressed in a specific contract. That contract is the interface between the object and the rest of the system. The contract may be seen to have two distinct purposes [15, p. 7]:

1) It tells the users or clients of a service how to construct a message to invoke the service
2) It tells the communication infrastructure the format of all the messages an object will send and receive

Additionally, according to CORBA, each object also needs a unique handle that is used by the communications infrastructure in routing the messages. That handle identifies the object and it is not changed even if the object changes place. Figure 2 visualises the object interaction. The components presented in it are the client, the object and the object request broker. According to the architecture, requests always proceed through an Object Request Broker (ORB). In addition, both interacting components are isolated from it by an Interface Definition Language (IDL) interface.

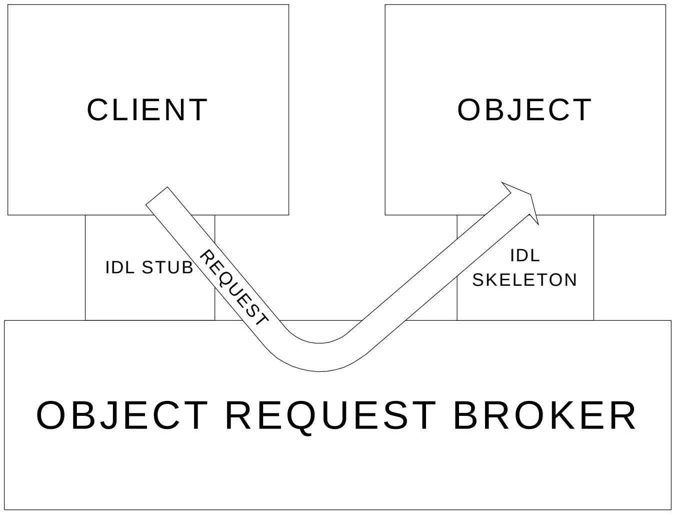


Figure 2. Request passing through IDL interfaces and ORB in CORBA.

CORBA requires that the interfaces have to be especially defined in OMG IDL. The use of IDL clarifies much of the actual operation of CORBA; details of each object’s implementation are hidden behind the interface definition. That assures the substitutability of the objects. If only the interfaces and overall functional patterns remain unchanged, the object implementations themselves may change. It is an essential feature for a distributed environment with a spontaneous – nature.

ORBs handle details of the distribution – therefore there has to be a functional ORB in a network of distributed CORBA objects. ORBs are usually manufactured by third parties, separate from the software developers that use them. Different ORBs operate in different environments. Due to that, different ORBs also have different characteristics. The ORB-skeleton-interface is proprietary and therefore ORBs and their IDL compilers appear in pairs; if the ORB is changed, the IDL compiler may have to be changed to mate the new ORB – thus being able to generate compatible stubs and skeletons.

CORBA is commonly used in a variety of systems worldwide and it is available for several platforms. Software that utilises CORBA may be built by using the most popular programming languages such as C++ and Java.

3.1.2 DCOM - Distributed Component Object Model

The Distributed Component Object Model, DCOM, is an extension for distributed components to the preceding Component Object Model, COM. Microsoft has created both COM and DCOM and they play a fundamental role in Microsoft’s Windows operating systems. To start with COM, which as the name quite explicitly implies, is a framework for component programming:

“The Component Object Model is an object-based programming model designed to promote software interoperability; that is, to allow two or more applications or ‘components’ to easily cooperate with one another, even if they were written by different vendors at different times, in different programming languages, or if they are running on different machines running different operating systems.” [16, p. 6]

COM specifies a certain way of building components that conform to the Component Object Model. COM is an object-based programming model. It defines a COM object as an instance of its representative COM class. According to object-oriented paradigms, a COM class defines the prototype of the object instances derived from it. A COM component is an entity that may comprise several objects and relations to other components. COM is based on definitions of COM interfaces, which are a way of accessing the operations of the components. One COM class may comprise several COM interfaces. A COM class instance may be queried for the interfaces it fulfils by calling a certain method in its standard interface. All the interfaces of the components that are in public use have to be standardised, and they may not be altered after their publication, versioning of the interfaces is strongly supported in COM.

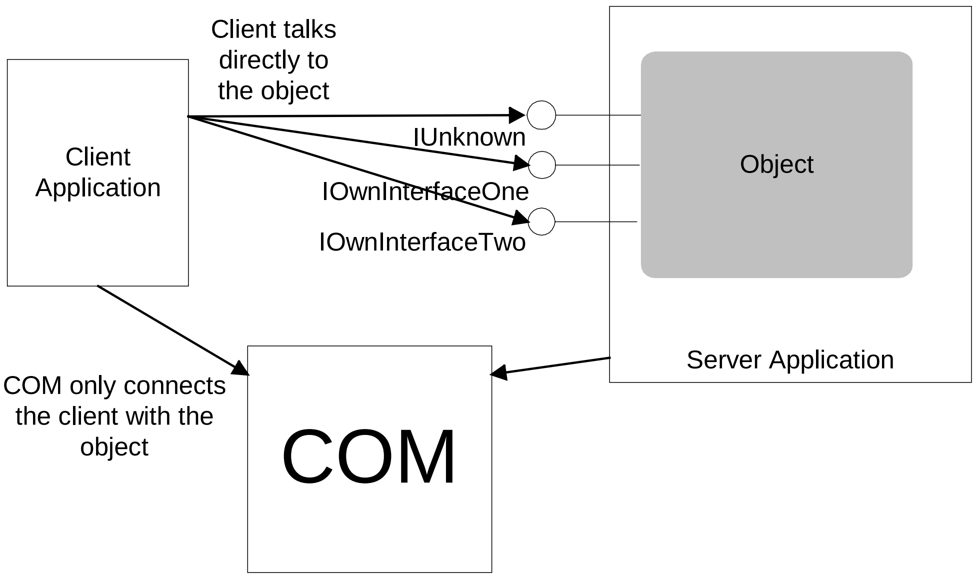


Figure 3. Component Object Model.

COM components may interact within the same computer. In addition to the specification part that defines the requirements for a COM component, COM also has an implementation part that enables the component interactions in practice. The COM functions as a practical runtime entity that provides the COM services, such as for creating and locating COM components. In Figure 3 the COM model has been visualised with an example in which a client application utilises a server COM component. The COM component has been exposed to the client by the COM runtime entity that is located at the bottom of the Figure 3. After the client gets to know the COM component's interface, it may call it directly.

Each COM object has to implement the IUnknown – interface. It comprises three simple methods:

- QueryInterface,
- AddRef, and
- Release.

IUnknown is the most fundamental interface in a COM component. The common availability of the IUnknown – interface also makes it possible to reach other interfaces of the component in question. The IUnknown interface enables access to the component's functions without knowing about them beforehand. Interfaces to COM components have to be especially defined by MIDL by Microsoft [17, pp. 19, 29].

COM supports different ways for the COM components to operate. Those ways are:

1) *In-Process*, in which the COM component is loaded into the client's address space and executed within the same process with the client.
2) *Local*, in which the COM component is executed as an independent executable and the calls are mediated between the client process and the server component by the COM.
3) *Remote*, in which the server component runs on a separate machine that is connected to the machine that hosts the client by a network connection. The remote type may be implemented by utilising the Distributed COM, DCOM.

Thus the distributed version of the COM is DCOM. The most essential point in the use of DCOM with COM components is that the client makes the call exactly the same way as it would if the server component were of the In-Process type, independent of the overhead and complexity involved in establishing communication through a network connection to another machine. In [18, p. 439], the essence of DCOM is clearly expressed in the following phrase: *“Whereas COM is a specification for building interoperable components, Distributed COM is simply a high-level network protocol designed to enable COM-based components to interoperate across a network”*

DCOM is considered as a high-level protocol as it may utilise many protocols that are available in the system when it is in operation. DCOM chooses the best underlying protocol by basing on the information about the available protocols on the client application that uses the component and the server that hosts the component. [18, p. 439].

3.1.3 Enterprise JavaBeans Component Model

Enterprise JavaBeans emerge from a preceding technology of JavaBeans. Here, JavaBeans are discussed first and then complemented with the features introduced by the enterprise edition.

In common, beans are Java classes or sets of Java classes that are designed to work as a reusable component. The definition of JavaBeans is given in [19] and [20, p. 11]:

“A JavaBean is a reusable software component that can be manipulated visually in a builder tool”

Multiple beans can be assembled together with minimal programming to create larger systems. There are many development tools for graphically manipulating and connecting beans. One of the main goals of the JavaBeans component architecture is to provide a platform-neutral component architecture [19, p. 7]. The component nature of JavaBeans may be considered to approach the component nature of integrated circuits [21, pp. 455–456]. It is a widely adopted paradigm in describing component software, introduced, but also criticised, for example in [22, pp. 3–13] by C. Szyperski. Just as integrated circuits, beans have a self-contained, well-defined behaviour and they adhere to some design framework. Families of components or beans share the same communication protocols, thus being able to communicate directly with each other.

Beans follow a slightly different composition than the typical object-oriented nature in Java would propose. There is no specific interface for beans, nor there is any extendable base class. Instead, beans are defined by a set of design patterns [23, pp. 60–65] that dictate how certain operability is to be developed and what kinds of design rules are to be followed in order to fit in to the design framework of beans [19, pp. 9–15, 21, pp. 456–510]. Documentation, localisation and packaging are also parts of the bean design and the required component way of thinking.

Enterprise JavaBeans is a technology that combines the multi-tier applications architecture with the component model introduced by the JavaBeans component architecture. EJB logically extends the JavaBeans component model to support server components. According to [20, pp. 1–2]:

"The Enterprise JavaBeans architecture is a component architecture for the development and deployment of object-oriented distributed enterprise-level applications. Applications written using the Enterprise JavaBeans architecture are scalable, transactional and multi-user secure. These applications may be written once, and then deployed on any server platform that supports the Enterprise JavaBeans specification".

Operation of Enterprise JavaBeans is concentrated on the EJB container, which is a manager of the enterprise beans within it. The container is responsible for performing multiple tasks for the enterprise bean objects. The most important of those tasks have been listed in the following:

- provide a remote interface for the objects,
- create and destroy object instances,
- check security for the objects,
- manage the active state for the objects,
- coordinate distributed transactions, and
- manage persistent data within the objects (optional).

Containers provide an application context for the components to run in. In practice it means providing an operating system process or a thread in which the component can be executed. EJB containers reside at the EJB server, which provides an environment that supports the execution of applications developed using Enterprise JavaBeans technology. The EJB server manages and coordinates the allocation of resources of the platform on which it is running. The EJB server must, for example, provide an access to a distributed transaction management service.

Portability of EJB components relies on the strength of Enterprise Java API, whose subset EJB technology is. Enterprise Java API provides access to multiple kinds of existing infrastructure services like, naming and directory services, messaging services, transaction services and database access, thus making it possible for the beans to select suitable services that exist in the platform in which they are being executed.

In a typical Client/Server model, enhanced by EJB, there are components on the server and on the client. Server components are typically non-visual and execute within a container that is provided by an application server. Client components, on the other hand, execute within a graphical container like a document or a window. Components willing to qualify in an EJB environment have to fulfil certain requirements described by design patterns of the EJB component framework.

3.2 Virtual machines and mobile code

3.2.1 Java RMI

Remote Method Invocation is a feature built in the Java Virtual Machine (JVM) since its version 1.1. The fundamental purpose of RMI is to make it possible to invoke a method in an object that resides in a remote location. Accordingly, RMI comprises the means to pass the required parameters to the remote object as well as to receive a response from it. This all takes place in a synchronised and elegant way. The difference between RMI and pure RPC is that RMI operates on Java objects rather than bare procedure or function calls. In comparison, RMI offers more sophisticated methods of building complex but controllable solutions. RMI develops with Java language as a product and therefore the latest versions are not necessarily fully compatible with the older ones. The features under consideration here are from the version shipped with the Java Development Kit (JDK) 1.2.2.

Use of RMI is simple. Interfaces of remotely available objects have to be defined and derived from the java.rmi.Remote interface, which is barely a marker for the system to know that the interface in question is a remote one. A ready-made implementation for performing the actual RMI function is the java.rmi.server.UnicastRemoteObject class. Extending the UnicastRemoteObject is a straightforward way to build remote objects and to distribute references to them. In the most common case, a UnicastRemoteObject - extending class is made to implement the remote interface class with all the required functionality. Then when instantiated, the UnicastRemoteObject class performs the functionality required in exporting the object when necessary. The reference of a remote object may be connected to a simple directory facility called the RMI registry. By binding the object reference with a certain identifier and registering them with the RMI registry, other systems may look up the identifier in order to obtain the reference and to establish contact with the remote object.

A powerful tool that has been integrated with Java RMI is the dynamic code downloading. In practice, dynamic code downloading means that the parameters of remote method invocations may be instances of classes that are not known at the destination or the return value may be an instance of a class that is not known by the caller. In such cases, the class file of the parameter class or the return value class is transferred to the JVM that requires it, at runtime. Mechanisms involved with the dynamic code downloading in Java have been discussed in further detail in the end of the next subchapter 3.2.2 that considers Jini.

3.2.2 Jini

Jini, introduced by Sun Microsystems in 1999, is an advanced technology that offers distributed and spontaneous solutions for many types of services. It is based on different concepts than most of the technologies seen today. First of all, all functions of Jini are completely based on Java, thus making it independent of any specific processing environment. Secondly, the operation of Jini is based on mobile code, which is still quite a unique feature.

Jini itself is a set of concepts and interfaces that define the requirements for the actual implementation. Currently the only existing implementation is the one by Sun Microsystems. In its fundamental idea Jini is not an implementation, but rather an idea defined by those concepts and interfaces.

Jini provides a networking infrastructure that allows services to find each other, and thus enable a flexible way of distributing services and parts of services across the network. Additionally, Jini provides functions that help distributed applications operate efficiently together in a consistent manner. Architecture and operation of Sun's implementation of Jini is described in the Jini Architecture Specification [24]. Five key concepts of Jini are defined in [13, pp. 64–102] as follows:

- The **Lookup** concept of Jini refers to finding, placing and removing objects within a lookup service, which is a versatile storage and name service. Join protocol is used when inserting objects into the lookup service. The objects are received through RMI remote invocations through a specific registrar object received from the lookup service [25, pp. 117–126, 26, pp. 113–126].
- **Discovery** is the process of finding the available lookup services from the network through protocols specified in the Jini Technology Core Specification [26, pp. 1–34].
- **Leasing:** A lease is a contract between two members of a Jini community. It represents a period of time granted for the use of a resource. If a lease expires and is not renewed, the resource allocated by the owner of the lease is released [26, pp. 35–53].
- **Remote Events** are supported by Jini. The concept of an event is the ability of one object to notify another object when something has happened [27, pp. 102-116]. Remote events extend the basic Java event paradigm first defined in the JavaBeans specification [19, pp. 24–39] to operate in a distributed environment.
- **Transaction** is a sequence of operations performed by transaction participants in order to reach a logically atomic operation. A system-level

support for transactions helps distributed systems to control the consistency of the shared data [26, pp. 83–112].

Jini defines a concept of a *community*, which is a set of services in a particular network. The services of a community make themselves available by registering their representative object(s) to a lookup service as shown in Figure 4 and Figure 5. In registration, a special kind of object (proxy object) is transferred to the lookup service (1 in Figure 4). The proxy object represents the interface to the service when transferred to and activated by a service user. The lookup service will receive the proxy object in order to host it and accordingly to grant the service a lease for using its resources. The registered service will have to renew the lease periodically in order to stay registered.

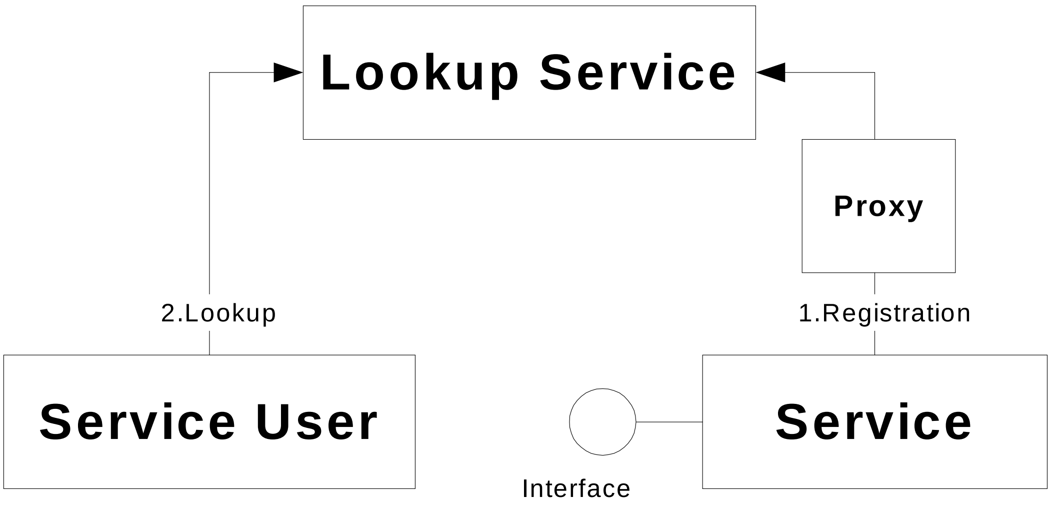


Figure 4. Service registration and lookup in Jini.

A service user may utilise a registered service by contacting a lookup service and by introducing to it a template of requirements that describes the service that would serve its purpose (2 in Figure 4). The lookup service will return a list of matching services and their proxy objects. Finally the proxy object is moved to the service user, (3 in Figure 5).

When using a service, the service user has access to the proxy object that provides an interface to the service. The proxy object performs the service function at the service user end. It will contact the service when necessary by using the service-specific method (4 in Figure 5) that is not by any means restricted by Jini. The services may of course freely utilise the fundamental services provided by Jini such as leases, transactions or remote events, in their operation.

The heart of the abstract structure of Jini is the concept of lookup service. The lookup services act as connecting points in the communities. The technology below the abstract level in Jini is based on Java RMI and especially on its dynamic code downloading paradigm [3, 28]. Dynamic code downloading is one of the most essential building blocks in Jini. The implementation of the proxy-pattern in Jini is dependent on mobile code, and the most essential in the service distribution as well as in the service distribution in Jini is based on the proxy-pattern. The combination however works well.

Indeed, according to [28], Java is the only known system incorporating such a powerful combination of features essential for mobile code mechanisms. In Java the appropriate way of implementing mobile code is dynamic class loading. Jini does not offer any magical tricks in order to perform the mechanisms of dynamic code downloading, all the required functionality is included in Java and RMI.

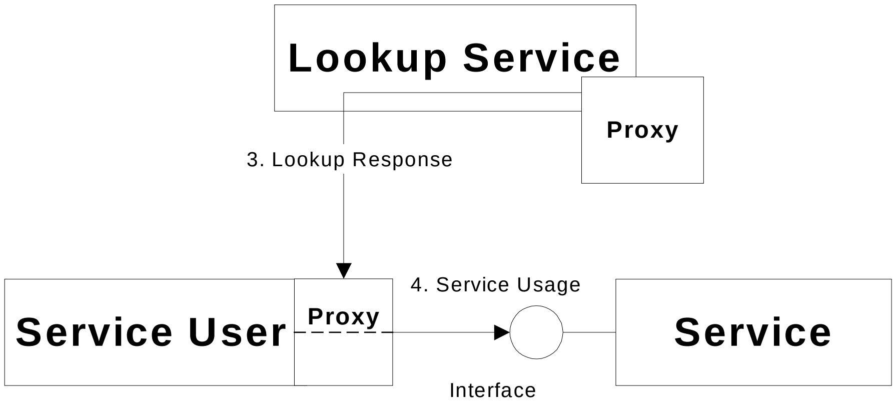


Figure 5. Lookup response and service usage.

In Java, serialised [29] objects may be moved around without the definition of the actual class that they represent, i.e. the data may move without the code. A serialised object contains only the data contents of an object without its functionality. When an object is then de-serialised for use, the class definition is again required. The use of marshalled objects solves some of the problems related to acquiring the class file. Marshalled objects are used for example by RMI. A marshalled object is able to store a serialised object with additional information about the location of the class file of the class that the object represents. If the class definition is not found locally, dynamic code downloading is performed in order to acquire it. A specific class loader takes care of transporting the class file from an HTTP server that is there to deliver the class files. The address of that HTTP server moves around with the additional information that is stored in marshalled objects.

3.2.3 OSGi - Open Services Gateway Initiative

The Open Services Gateway Initiative is an effort to create a specification for distribution of services in networks. The purposes of the OSGi consortium, formed by more than 75 member organisations, concentrate in the following phrase:

"The OSGi Framework and Specifications facilitate the installation and operation of multiple services on a single Open Services Gateway (set-top box, cable or DSL modem, PC, Web phone, automotive, multimedia gateway or dedicated residential gateway)."[30]

OSGi concentrates on building the required technology for an Open Services Gateway, which is an embedded server with a capability to enable, consolidate and to manage a wide range of connections. OSG acts as a gateway between the large public network and the smaller local networks. Another purpose of the OSG is to provide interconnecting infrastructure for different kinds of devices and networks.

OSG is also able to operate as an application service provider. Services may be executed directly in the gateway. A context for developers is provided along with the service framework. The context is intended for developing services that are to be run in the gateway. The OSGi specification includes APIs for utilising the gateway services. Services are uploaded to the gateway as JAR-bundles. The encapsulated nature of the bundles increases the overall simplicity and consistency.

The general architecture of OSGi is more implementation-oriented if compared to the other solutions presented in this thesis, as it proposes a straightforward solution to a certain application area of networking. OSGi has not been made too complex and it does not try to solve all problems at once, which is a strong benefit.

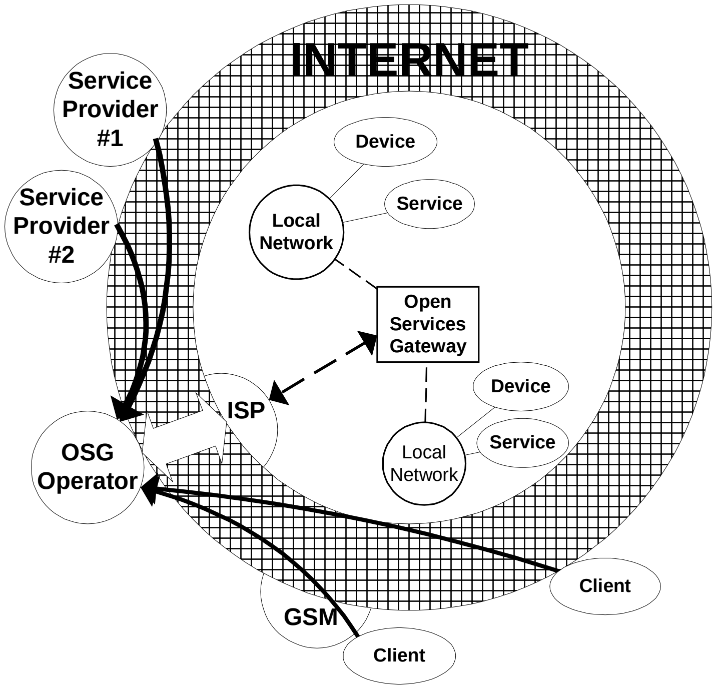


Figure 6. Overview of OSGi general architecture and deployment.

The architecture of OSGi as presented in [31], is visualised in Figure 6. The service gateway is in connection with:

- the Internet,
- public service providers,
- a OSG operator, and
- local networks.

An OSG is a secure, zero local administration device with connectivity to local devices and the Internet Service Provider. All the services in the gateway are managed and maintained by the gateway operator. Another management role in OSGi is as a Service Aggregator. A Service Aggregator offers a set of interoperable services and/or equipment for certain purposes, for controlling electricity, alarms, etc.

The first version of OSGi gateway has been implemented as a Java API. It comprises:

- Java Environment – required packages and classes,
- Service Framework – API for creating and running services,
- Device Access Manager – API for accessing devices,
- Log Service – required service for logging information, and
- HTTP Service (not a requirement for all OSGi compliant gateways).

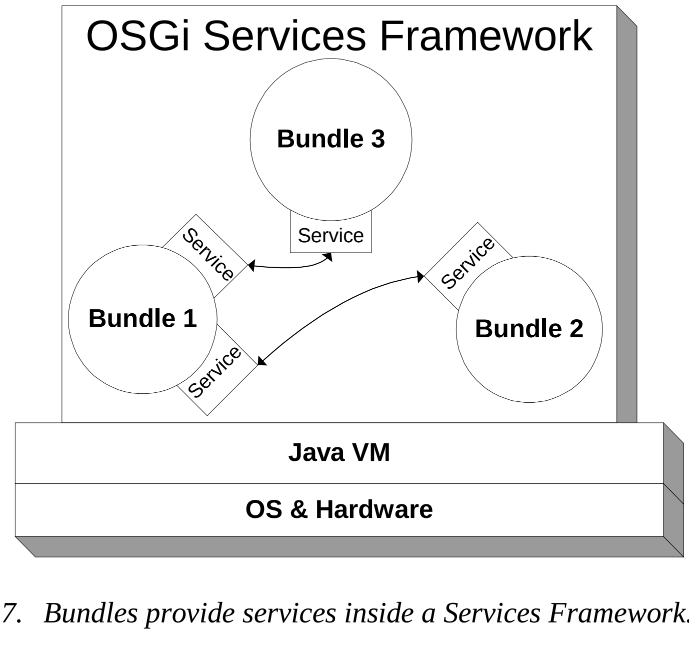


Figure 7. Bundles provide services inside a Services Framework.

OSGi offers a solution to the real-world problem of connecting devices, users, services, networks and service providers together. The main element of the

OSGi specification is the OSG framework. The framework accordingly relies on three key elements:

- Services – made to perform certain functionality,
- Bundles – in which the services are shipped, and
- Bundle Contexts – in which the bundles are activated.

The specification defines the programming guidelines for bundles. The bundle model of OSGi resembles JavaBeans, as the bundles are components packed in to JAR packages and executed in a certain predefined context with their definitions carried inside a manifest file. There are also differences. JavaBeans are primarily visual components, which is not the case with OSGi bundles. JavaBeans also rely more on pre-defined programming patterns, while OSGi provides a ready-made API. Figure 7 visualises the bundle-structure inside an OSG framework.

3.3 Peer-to-Peer connectivity

3.3.1 UPnP - Universal Plug and Play

Universal Plug and Play is an architecture for peer-to-peer network connectivity from Microsoft. The fundamentals for it have been created in 1999 by a group of companies and individuals. Currently, UPnP is represented by the Universal Plug and Play Forum [32]. In 2002, the number of members in it was already over 500 [32]. The purpose of UPnP is to establish seamless interconnectivity between PCs, intelligent appliances, and wireless devices in a form of peer-to-peer networking. For that purpose, UPnP describes the use of certain common protocols to enable connection from one device to another via any known media for information transfer. UPnP also utilises several commonly known protocols in its operation. The practical domain of UPnP is mainly an enclosed network with a reasonable amount of participants.

There are six major issues in UPnP that also give an overview on its operation. Those issues are *addressing, discovery, description, control, events, and presentation* [33, 34]. The issues are presented briefly in the following:

- *Addressing:* Each UPnP device has its own IP address. The address may be achieved either by DHCP[12] or Auto-IP[35].
- *Discovery:* Each UPnP device has to advertise itself. Advertising takes place by so-called control points. Control points are entities that advertise services and provide responses for search requests by issuing device searches in the network.
- *Description:* Each UPnP device may be described with a description that is separated into two logical parts, a *device description* and a *service description*. Information about the devices and the use of their services is carried within those descriptions.
- *Control:* A control point is able to use the functionality of a device or a service, after it has received enough descriptions about it. Controlling takes place according to a convention that utilises HTTP, XML and SOAP.
- *Events:* There is an event subscription/notification convention in the UPnP. Services may contain variables that publish events when stated changes in them occur. Control points may subscribe to those notifications.
- *Presentation:* A service may provide a URL that presents its device description and a control point may fetch the document from the specified location accordingly.

According to one of the principles in UPnP, no executable code is transferred in the service activities. Thus, only data is transmitted in a platform-independent manner. Often some generally accepted data formats such as JPG or XHTML, are used. Communication between non-IP networks takes place through specific UPnP protocol bridges. In general, the use of Internet protocols, such as IP, TCP, UDP, and HTTP, makes the use of UPnP suitable for the development of spontaneous services for the already existing environments and networks, which is a strong benefit. UPnP version 1.0 standards have quite recently (2002) started to appear as approved.

3.3.2 Gnutella

Gnutella is an Internet peer-to-peer file system protocol. Due to its nature in advancing the dissemination of illegal copies of digital media, such as music and films, there is no official product called “Gnutella”. The first program carrying that name popped up in March 2000, written by Justin Frankel and Tom Pepper of Nullsoft (owned by AOL/Time Warner). The parent corporation declared the accidentally released beta version of the software as an “unauthorised freelance project”. The original version 0.56 spread out, however. Currently there are several clones available for major operating systems such as Windows, Linux and Macintosh, which have been implemented through a reverse-engineering effort on the original software [36].

The primary purpose of Gnutella is to share files in the Internet between equal peers that may provide their own files for uploading, and simultaneously seek files they desire or download them. The most interesting feature in Gnutella is the decentralised structure of the peers involved in the file sharing process. It implements a simple protocol that enables a pattern in which each node of the network may operate as a server and as a client at the same time. These nodes are popularly called servents. Each servent in a Gnutella network implements the Gnutella protocol [36].

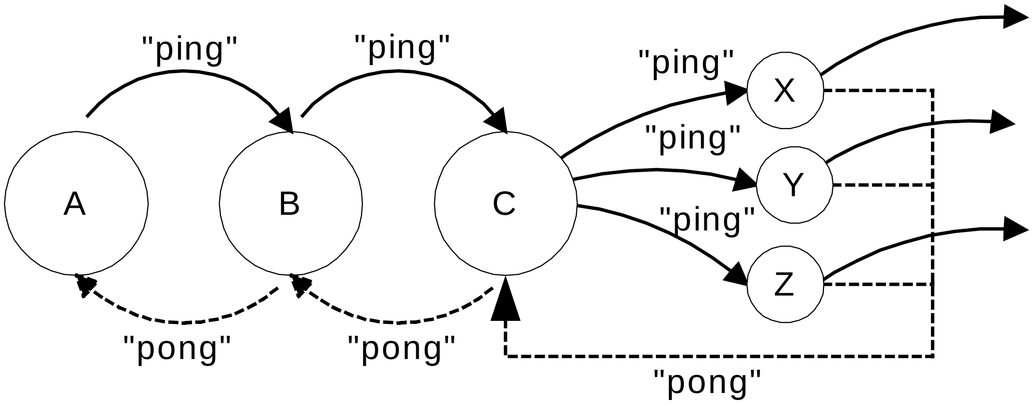


Figure 8. A simple example of message interchange in a Gnutella network.

In order to join a Gnutella network, a servent has to know the address and port of one existing Gnutella servent that is already connected to the network. Such starting points are being generally delivered through dedicated web sites. A simplified example of the Gnutella protocol has been explained as follows:

In an imaginary situation, three Gnutella servents A, B and C exist in the network. The example follows the process of servent A in both delivering files and searching for files from the Gnutella network. The example has been visualised in Figure 8. Initially, the servent A has to know the IP address and the port number of some other Gnutella servent, which is a prerequisite for joining the community. Assuming that servent A knows those parameters of servent B, it is able to connect with it by sending an initialisation “ping” packet. When B receives the “ping” packet, it will respond to it by sending a “pong” packet to A. The “pong” packet contains information about the number and the sizes of the files it has to share with the community. Simultaneously, B forwards the initial “ping” packet on to the other servents it knows. In the case explained, B forwards the “ping” packet to servent C. Servent C will then respond by a similar “pong” message containing the number and the sizes of the files it has to share. The “pong” message sent by C is sent to B, whom will forward it to A. Forwarding of the messages assures the anonymity of the servents. C does not know about the existence of the servent A. Thus the servents know only other servents that are one node away from them. Searches for files in the Gnutella network work similarly to the “ping” and “pong” messages. In transporting files, Gnutella peers communicate directly through a TCP connection.

3.4 Software architecture

Software architecture is a tool to reach design goals, the step between the intention and implementation. Although being a solution to a problem in a fundamental sense just as any other or the solutions presented previously in this chapter is, software architecture addresses issues that give it a special status amongst them. Most importantly, software architecture is not an implementation. Instead it is a way of thinking in terms of software, through different kinds of abstractions.

This subchapter provides basic information about the concept of software architecture. The overview given here is more intense if compared with the previous subchapters 3.1-3.3 that provided only a brief overview of the presented technologies and solutions. A software architecture of a distributed middleware service framework has been created in the work this thesis intends to introduce. Moreover, software architecture is an often misunderstood discipline and it has several definitions. In that light, it is essential to have a look on the concept of software architecture in order to clarify its use here.

3.4.1 What is software architecture?

Each software has an architecture. With refererence to [37, p. 23], the concept of the software architecture of a program or computing system is explained in the following way:

> *"The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them."*

When looking at the complete architecture of a system, the information about it has to be brought to some level of abstraction in order to make it understandable. If this were not the case, instead of the architecture, the whole system would be under consideration. This makes the architecture an abstract representation of the system. *Everything* about a system cannot be explained in any sense by any architectural description. Hence, it is not even the purpose.

Instead, the purpose of the architectural description of a system is to provide the necessary information about the system for analysis in several dimensions. Throughout the analysis, the architecture may be developed towards meeting the given requirements without building the entire system. According to the definition given in [38, pp. 3-7] the software architecture will provide a design plan and a way to manage the complexity of the system through the introduced abstraction.

There is a structure or there are structures in a software system. Structures describe the different relationships between the abstract parts of the system. The IEEE Recommended Practice for Architectural Description of Software Intensive Systems [39] defines software architecture with the following powerful phrase as:

> *"the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the priciples governing its design and evolution"[39, p. 3]*

The mentioned *fundamental organization* is the primary structure that comprises all the structures that can be drawn out of a system. In practice, that amount is nearly infinite. That is the reason why abstraction is so important when considering software architectures. Structures can be found from the temporal behaviour of a system, the connections between the parts of the system, the functional sequences in it and the relations of the synchronisation in it. Structures exist in models that relate to the design, planning, scheduling, implementation, and an excessive set of other similar kinds of elements and concepts. Many of those structures are dependent only on how or by whom the system is looked at in each particular case. Almost as important as the definition of the architecture as the fundamental organisation of the structures is to understand that *none of the single structures of the system is able to describe the architecture of it*. This, however, makes the structures no less important. It is essential to count in the structures when describing the system through architecture, because even without being *the* architecture, most of the structures and descriptions about them comprise important information about the architecture – without them the system would not be the same.

As an implication of the statement borrowed from [37, p. 23] at the beginning of this subchapter, architectures define components. Components are parts of systems that enclose certain functionality whose internal technique of operation is hidden from external observers. Components are abstraction tools that help us achieve those structures that our mind is able to produce. Human thinking and development process favours object-orientation and component-oriented thinking. In general, our brains naturally favour such models that imitate the real world. Just as real-world objects have surfaces used in interactions with other objects, components also have such surfaces, called interfaces. One of the most important characteristics of a component is the interface it has with the outside world. This is, of course, a simple implication of the presented fact that the internals of the components are hidden.

An important link between components and the architecture comes from the visibility of the component’s internals. The behaviour of the components in a system is said to be architecturally relevant only if it can somehow be observed by another component. Each component, of course, has an internal operation of some sort. The difference between architectural and non-architectural information in a component may be explained with the citation *“externally visible properties of those components…”* given at the beginning of this subchapter. Thus, if the implementation technique of a certain operation in a component may be changed without altering the way other components see the changed component through the interface it exposes to them, the information about the operation may be abstracted away at from the architectural descriptions of certain abstraction level.

Every system can be shown to be composed of components and relations among them [37, p. 24]. According to the definition used here, all systems also have an architecture. However, the fact that the architecture exists, does not straightforwardly imply that it would be visible in any sense. The architecture does exist in each implementation of any system, but in order for it also to be visible, it has to be specifically described in a suitable form.

There are many different ways for expressing architectural information. In practice many of the system architects have created their own methods of using the popular conventions for doing so. Fundamentally there is no difference whether it is a circle or a rectangle that represents a component, both of them will do just fine. Similarly, the understanding of any other structure or structures of a system may be expressed with varying conventions without affecting the effectiveness of the expression a great deal.

3.4.2 Quality attributes

Software is most often developed to accommodate needs. The needs of the domain as well as the needs of the stakeholders set the overall requirements for software.

Depending on the case, such requirements may vary on a wide scale and new ones may come out during the evaluation of the issues that are related to the architecture and its domain. Requirements are semantically connected to attributes called qualities. Requirements need certain qualities to be fulfilled. Some requirements appear in almost every piece of software developed for a certain domain, just as the qualities do. Some of them are rarer just as some are application specific. The more common the domain of the architecture under consideration happens to be, the more common is the induced set of requirements for the qualities. Sometimes a ready-made piece of software architecture or implemented software may be evaluated in order to reveal how well it fulfils certain qualities.

According to quality-driven architectural design, those and other such requirements are referred to as quality attributes [37, pp. 75–90] – they define attributes, whose quality should be evaluated. One could ask, “how well does the software complete reusability?” or more generally, “how reusable is the software?” The answer is of course not obvious since a quality attribute like reusability may be defined in very many dimensions. It is, however, a starting point. Different dimensions of software may be described by expressing abstracted structures of it. The overall reusability of the software may be expressed if required, without going into details, by presenting the relevant structures that comprise the details of those decisions and artefacts in the design that have the most impact on reusability. If the attribute cannot be expressed unambiguously with such presentation, there is also something wrong with the attribute’s implementation in the overall architecture of the software.

ISO/IEC has developed the standard 9126:1991 in order to maintain the persistence of the definitions of the quality-attributes related to software [40]. The explanations given there are not the only possible ones indeed. Another related standard is the IEEE 1061[41] standard for software quality metrics methodology. This thesis does not make an attempt to define quality attributes as such. Independent of the involved diversity, quality attributes are an excellent way to approach the fundamental issues that question the relevance of a certain software architecture in fulfilling the task it has been initially proposed for.

Without trying to make a defective list of all possible quality attributes, those that have importance in the work presented in this thesis have been introduced in the following Table 1.

Table 1. Quality attributes briefly explained.

Quality attribute	Brief explanation	Means of achievement
Adaptability	Attributes of software that bear on the opportunity for its adaptation to different specified environments without applying other actions or means than those provided for this purpose for the software considered [40].	Case-driven architectural solutions. No restrictions to granularity. Generality in abstraction, interfaces, and responsibilities. Asynchronous communication, architectural definitions of behavioural dependencies.
Integrability	The ability to make the separately developed components of the system work correctly together [37, p. 84].	Simplicity of the components, their interaction mechanisms and protocols. Clean partitioning of responsibilities. Clear and complete specifications of the components' interfaces. [37, p. 85].
Functionality	The ability of the system to do the work for which it was intended [37, p. 81].	Correct and careful design of the responsibilities. Orthogonal to structure, thus non-architectural in nature. Often bound by other quality attributes.[37, p. 81].
Availability	The proportion of time the system is up and running, measured by the length of time between failures. Ability of the system to resume operation in the event of failure [37, p. 80].	Redundant components, attention to error reporting and failure handling, specific monitor components. [37 p. 80].
Usability	Breaks down into learnability, efficiency, memorability, error avoidance, error handling and satisfaction, which are measured from the user's point of view [37, p. 81].	Matching the user interface to the product to respond to the use. Familiar metaphors, standards, interface conventions. [37, p. 81] Applies both to user interfaces and software interfaces.
Extensibility	The ability to make changes quickly and cost effectively, the ability to acquire new features. [37, p. 83]. Often considered to be a subcategory of maintainability, in a similar way as adaptability is [40].	In general, the same rules apply as to adaptability.

3.4.3 Frameworks and patterns

Here, the terms framework and pattern are briefly defined so that their use in the context of this thesis would be unambiguous. A framework or a pattern, are both generic expressions that carry much of their content already in their semantic meaning.

A framework may be thought of as a piece of unfinished software architecture that defines the skeleton of a system but requires complementation. Complementation to an architecture cannot be made without also altering the architecture, thus defining such architectures that require modification as unfinished is justified in that sense. An architectural construction that is a framework may be implemented by several different methods, for example, through the use of abstract classes, as suggested in [23, p. 54]. Another method to formalise an approach for creating architecture for a framework is to consider an integrated set of components that can be reused and customised according to the needs of the framework user. Such a viewpoint has been taken, for example, in the work explained in [42, p. 20] and it addresses greater flexibility than the use of abstract classes would. The approach in the framework for distribution services presented in this thesis has been influenced by both approaches. A generic view of the approach considering the use of frameworks in this thesis has been given as an example in [43].

Patterns are related to frameworks and to software architecture as well. A pattern is a common practice for *doing a thing*. "Doing a thing" is a very generic expression. Similarly, the concept of patterns is enormously flexible. Patterns may be found from software architecture design, software development and software implementation techniques. When different software architectures are repeatedly designed by using a certain, similar way of solving a certain, similar problem, the way of solving the problem by those architectural means may be considered a design pattern. The size of the problem is not restricted by any means thus, the size of the patterns is not restricted either. Large patterns that govern the construction of a complete software architecture may be called architectural patterns. In software implementation, certain commonly used conventions may be considered patterns, even though they would not encompass any architectural relevance. In some certain software architecture, a certain way of doing things may repeatedly emerge in large amounts of elements. That kinds of patterns are just as valid as the other kinds of patterns, even though they would not have been introduced anywhere else outside that particular architecture. The work described in this thesis comprises some such patterns.

3.4.4 Role of software architecture in this thesis

The DisMis framework (chapter 4), as well as also other work presented in this thesis (chapter 4.6), has been affected by the definition of software architecture as a fundamental organisation that is present in any software system, as explained in 3.4.1. The design work of the DisMis architecture has been supported by the fact that the author has had previous experience in creating architectures of similar kinds of solutions. In terms of software architecture design, previous design experience in certain domains is one of the most important factors that facilitate top-down system design on those domains.

Less effort has been given to any individual architectural method or style. Also, the research nature of the work has given freedom to leave some qualities left with less effort without causing any permanent damage to anything or anyone, thus making it possible to neglect certain issues in the architecture that would be impossible to be neglected in an industrial environment. On the other hand, it has provided the opportunity to keep the development on a conceptually high level, making it possible to keep the solutions generic. As the second phase that has not yet taken place, the applicability of the DisMis architecture should be proven in real implementations, fitted to operate on different application domains. Such a process would provide valuable information that is not possible to acquired otherwise.

In documenting the process of creation, software architecture has provided a fundamental tool that has been required in completing that process, at least as much as in recording the process. The origins of that process have leveraged the related technologies as well as the actual implementations of distinct cases that have been used, not only in validating the concepts, but also in giving inspiration, for the presented DisMis architecture. One of those cases is presented in this thesis as validation for the conceptual architecture given in chapter 4.

The model used in designing and presenting the architecture of the framework in this thesis has been adopted from the basic principles of the QADA method, developed at VTT [44]. According to the approach in QADA, the architectural design has been divided into a system requirement specification, conceptual architecture design and concrete architectural design, which all have a concurrent analysis task that processes the results and feeds them back to the architecture or requirement specification, depending on the phase. As the development of the work presented in this thesis has lasted over a period that extends to time preceding the development of the QADA method, all the principles could not have been followed as suggested in [44]. However, some of those principles adopted from the method have been listed in the following:

- System requirements engineering has been completed as a continuous task that has lasted for several architectures and implementations that are based on the same requirements.
- Requirement analysis has been experienced, although it has been done the "hard way" – by implementing the software architectures of actual systems.
- Conceptual architectural design has followed the QADA, at least in principle, by defining the functional responsibilities of the system (4.3). However, the methods used have been more consensual. One version of an earlier design of the same family of software architectures has been presented in [44, pp. 84–115].
- The conceptual system has been divided into subsystems, as suggested.
- Conceptual deployment has been considered (Chapter 4), deployment nodes have been identified, and the deployment units have been allocated to nodes, as suggested.
- A hierarchical structure was created when designing the concrete architecture. The structure has been presented in diagrams to a certain extent.

Additional information about the QADA method may be found from [44]. With a limited set of the given analysis methods concerning the architectural design of a software system, the following chapter 4 introduces the architecture of the DisMis framework.

4. Decentralised, distributed middleware

The main contribution of this thesis is a solution for decentralised, distributed middleware. The conceptual approach and the terminology used will be provided here first within the first two subchapters. Then, the details of the distributed middleware solution will be presented by considering the software architecture of a system that would be able to accomplish the required tasks. The explanation related to the architecture has been divided into several subchapters. In order to limit the length of the presentation in the context of this thesis, the level of details in the architectural descriptions has been limited to a certain abstraction. However, the most important purpose has been to provide a balanced look at each part of the mechanisms and structures that belong to the presented architecture. The selected technologies have been provided for comparison between characteristics that have been suggested in existing solutions (3.1–3.3) that belong to the domain. The comparison has been presented at the end of this chapter, in subchapter 4.5.

4.1 Overview of the concepts

The purpose of this subchapter is to introduce the ideology behind the DisMis architecture. All of the features in the different technologies, solutions, and theories presented in the previous parts of this thesis have played their respective role in the process of creating the idea of a platform for decentralised, distributed middleware services. In addition, understanding the requirements of applications that operate in spontaneous environments has been another strong motivator for the solutions and decisions. Much of the most important experience related to the practical operation of the technologies as well as to understanding the requirements has been gathered in practice through real implementations.

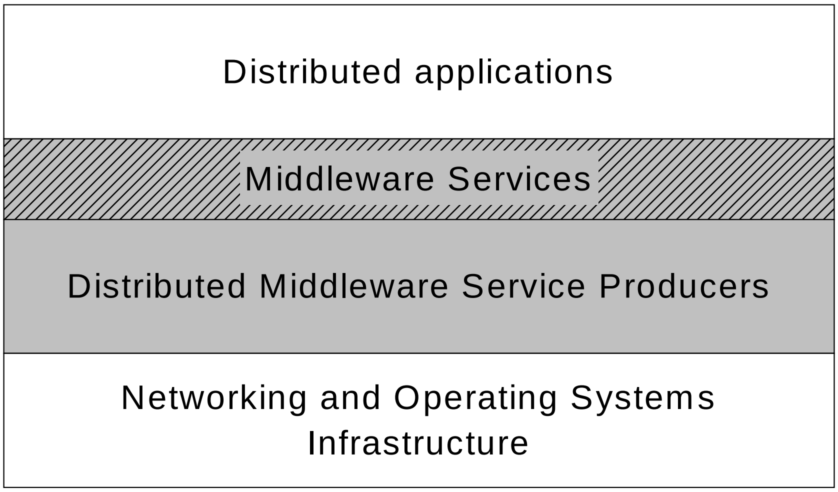

Distributed applications
Middleware Services
Distributed Middleware Service Producers
Networking and Operating Systems Infrastructure

Figure 9. Layered view of Distributed Middleware Services.

The presented solution is called here *Distributed Middleware Services*. To be specific, the solution involves *decentralised* middleware that is distributed. A layered model presented in Figure 9 clarifies the location of Distributed Middleware. As shown in Figure 9, middleware services used by distributed applications, are produced by Distributed Middleware Service Producers. The work presented in this thesis is a solution for Distributed Middleware Service Producers.

Distributed middleware services are middleware (2.3) services that have their internal functionality distributed somehow. A shorthand term "*DisMis*" has been used throughout this thesis when referring to "Distributed Middleware Services". The term decentralised has also been used, although it has been derived from the name DisMis for the sake of simplicity (considering the abbreviation). *Decentralised* refers to the nature of the solution as an infrastructure for spontaneous systems. It is based on a principle that provides each distributed middleware user a chance to participate in producing the middleware services that they utilise. Accordingly, there cannot be any specific middleware service producer, whose tasks would be irreplaceable. This generates the fundamental idea of a decentralised service platform – there is no irreplaceable centralised element that produces the services.

Table 2. *Use of terms related to the DisMis framework.*

Term	**Brief explanation**
Framework	A ***framework*** may be thought of as a piece of unfinished software architecture that defines the skeleton of a system but requires complementation. It may be represented through an implementation of a component or a set of components that encloses functionality but requires complementation with an external implementation in order to operate as purposed and designed.
DisMis	***Distributed middleware services*** in general.
DisMis Architecture	Software architectures related to a system that is able to create ***distributed middleware services***. Use of this term refers often to certain design decisions in the architecture of the ***DisMis framework***.
User Application	A ***distributed*** application that has to operate in a spontaneous distributed environment. In the case presented, ***user applications*** are the users of the ***distributed middleware services***. A ***user application*** may comprise several distributed components. Each of those components has been built atop the ***DisMis framework***.
DisMis Framework	The ***work*** done in this thesis. A complete implementation of the ***DisMis architecture*** that appears as a ***framework***. It is presented as a solution for establishing an infrastructure for ***distributed middleware services***. The ***framework*** consists of a component structure that has to be complemented with a ***user application***.
DisMis Framework instance	A run-time instance of an implemented ***DisMis framework***. A ***user application*** may comprise several ***DisMis framework instances***, depending on the deployment.
DisMis Platform	A ***common distributed middleware services platform*** that has been produced by the interaction of separate ***DisMis framework instances***.

The practical solution presented in this chapter is an architecture of a software framework for producing distributed middleware services. That architecture is referred to as *DisMis architecture*. An implementation of that framework architecture is called here accordingly as *DisMis framework*. The term *framework* refers to the nature of the solution as unfinished software architecture. Actual distributed applications that are to utilise the distributed middleware services, have to be built atop the architectural framework. The set of middleware services produced by the DisMis framework has been considered as *DisMis platform*. Table 2 summarises the use of the terms that are related to the solutions that are to be discussed in the following chapters.

4.2 Decentralised, distributed middleware services

Decentralised, distributed middleware services are middleware services that are not executed or generated by a certain centralised component or system, which is the case in many of the existing and legacy middleware solutions. On the contrary, the approach proposed in this thesis suggests a solution in which the middleware services required by the applications are produced by the applications themselves.

It would be impossible to demand that each distributed application developer would have to implement large software entities in order to make his/her application participate in producing the middleware services as proposed. Instead of such demands, this thesis proposes a DisMis framework that provides an infrastructure for the distributed entities of a distributed user application. Just as any other middleware solution, the purpose of the DisMis framework is to hide most of the complexity involved in building the middleware services. The strength of the DisMis framework is that its use should also be easy for the application developer in practice. The model comprises only one structural element, the DisMis framework. If considering its use in system composition, it should be possible to think of the DisMis framework as a single component of the system, thus making its use practical and visibility unambiguous. The DisMis framework may, under certain circumstances, have a varying configuration, but normally it is always the same. Therefore, knowledge over only one component and its interface is required. Additionally, the interface should comprise only a few operations. The only fundamental and structural requirement of the proposed DisMis concept is that each part of a user application that is about to operate as a separate entity within a distributed system has to be built by using the DisMis framework as its basis.

In practice, the DisMis framework may be, for example, an abstract class, of which all of the component’s main classes in a distributed system have to be derived. Such an approach has been taken in the concrete architecture created according to the conceptual structures presented in this thesis. The concrete architecture has, however, been excluded from the issues presented in this thesis.

Distinct DisMis framework instances that operate in a networked environment are able to discover each other as well as the services and capabilities offered by each other. All the operation may take place over several interconnecting media. DisMis framework is able to internally decide the responsibility for producing certain middleware services as well as the responsibility for securing those services. This takes place invisibly from the user applications that are built on the DisMis framework. The benefit for the applications is that even in an unstable and spontaneous environment, they may utilise more stable and reliable (guaranteed to a certain distinct) services of the DisMis platform. An explanatory view of the use of the DisMis framework in building a spontaneous distributed application has been given in Figure 10, Figure 11 and Figure 12.

Figure 10 represents a fictitious configuration of four devices that form a spontaneous network. As shown in the figure, the devices have been connected with each other via different communication media such as Bluetooth, Ethernet and IrDA.

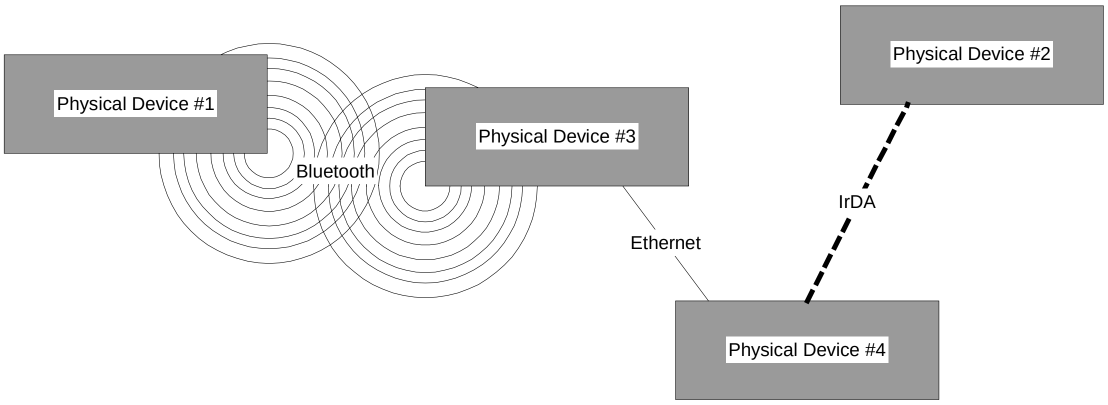


Figure 10. Interconnected devices in a fictitious configuration.

The imaginary software components in the devices of Figure 10 have been built on the DisMis framework. The interconnected software constructs a layered composition that has been visualised in Figure 11.

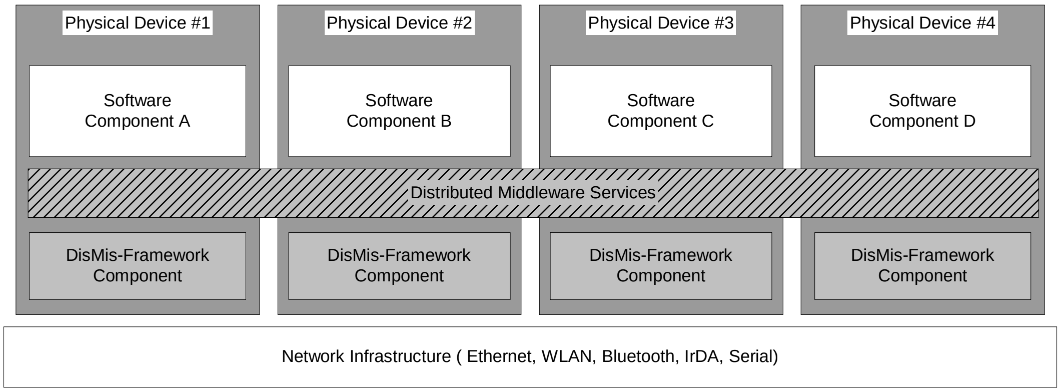

Physical Device #1
Physical Device #2
Physical Device #3
Physical Device #4
Software Component A
Software Component B
Software Component C
Software Component D
Distributed Middleware Services
DisMis-Framework Component
DisMis-Framework Component
DisMis-Framework Component
DisMis-Framework Component
Network Infrastructure (Ethernet, WLAN, Bluetooth, IrDA, Serial)

Figure 11. A layered composition of software in the interconnected devices.

As shown in Figure 11, the DisMis platform, produced by the DisMis framework instances, connects all of the software components within the different devices present in Figure 10. From the viewpoint of the software components themselves, the composition may be considered as simple as shown in Figure 11.

A spontaneous distributed application may comprise software components that reside in different devices and that are reachable over different media. This does not of course mean that any software component could be connected with any other software component in a spontaneous sense. On the contrary, the forms and patterns of interaction have to be planned carefully. One of the most useful approaches for such spontaneous interaction is that software components consider each other as services that facilitate them to perform certain tasks such as displaying image data, outputting sound, sending e-mail, etc. Applications that are based on spontaneous awareness of such facilities may be functional without any external services, but may be enhanced with additional features if they become available. This, of course, depends completely on the particular application. Some other application may merely wait until a necessary set of external services is available before being able to start its own operation.

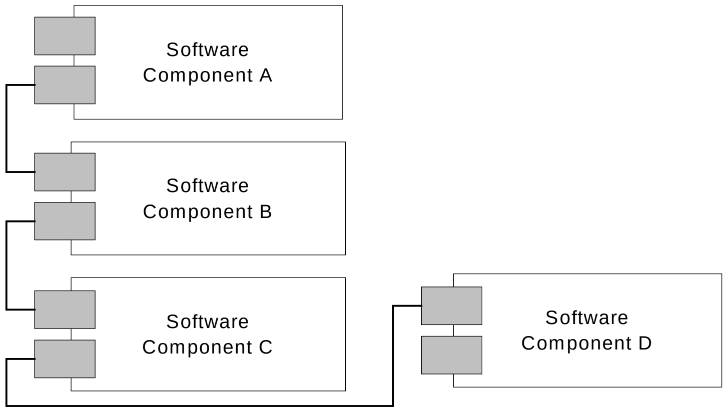


Figure 12. An abstracted view of a spontaneous distributed application.

The software components presented in Figure 11 have to take the influence of middleware into account in some of their operations. That would make the presentation given in Figure 12 slightly too abstract in that sense, as the middleware has been completely hidden in it. It is difficult, if not impossible, to create pervasive transparencies that suit every purpose and every environment. For example, the capabilities of different distribution media may sometimes set severe limitations on the operation of a distributed application. This should be taken into account in the development of possible interaction patterns of the components in a distributed spontaneous application. Therefore the connectivity between the user application components in Figure 12 should represent only one of the possible configurations of the components, which accordingly constructs only the concurrent system.

The flexibility in both the components and the connectors should be a built-in feature for pieces of applications that are about to operate in a completely spontaneous environment. That built-in flexibility is not visible in Figure 12. The purpose of this thesis is to concentrate on the infrastructure that enables the construction of such flexibility. The characteristics of the distributed and spontaneous applications are left outside the discussion, even though they would provide an interesting topic.

4.3 Key drivers for the architectural decisions

There were several factors that had to be considered in the architectural decisions related to the design of the DisMis framework. These key factors have been approached here through the most desired quality attributes as suggested in 3.4.2, and their impact on the design is presented.

Adaptability – attributes of software that bear on the opportunity for its adaptation to different specified environments without applying other actions or means than those provided for this purpose for the software considered [40]. Adaptability is the key motivator for the concept of distributed distribution services, in which the middleware services are produced by a certain pattern of interaction between all the interconnected application components that have been built by utilising the DisMis framework.

Integrability – the ability to make the separately developed components of the system to work correctly together [37, p. 84]. The reasons identified as integrability were the key factors when deciding that the system should act as a framework for all the components of distributed applications in a certain distributed system. This means that the distributed applications are built on the DisMis framework so that it becomes an integral and indistinguishable part of the design.

Functionality – the ability of the system to do the work for which it was intended [37, p. 81]. Functionality as a quality attribute is essential to the operation of the DisMis framework; other designs in the distributed system could rely on it during the complete time of their operation. Therefore, neglecting the importance of functionality here could lead to serious decrease of functionality in a system built on the DisMis framework. Functionality as a quality attribute is non-architectural in its nature (3.4.2) [37, p. 81], thus it is mainly restricted by the other quality attributes.

Availability – measures the proportion of time the system is up and running [37, p. 80]. This quality attribute is related to two slightly different things in the DisMis framework:

1) availability of the DisMis framework to the distributed application that is running on it – this availability is related to the reliability of the DisMis framework, and
2) availability of the DisMis platform, produced by the interacting DisMis framework instances – this availability is related to the functionality of a distributed system in which several user application components interact.

Usability – breaks down into learnability, efficiency, memorability, error avoidance, error handling and satisfaction [37, p. 81]. The usability in this case is not completely considered as a run-time attribute. There are two cases in which the usability may be considered here:

1) the usability of the architecture to a developer who creates the software that uses the DisMis framework, and
2) the usability of the platform services at run-time to the software that has been built on the DisMis framework.

Extensibility – the ability to make changes quickly and cost effectively, the ability to acquire new features [37, p. 83]. The requirement for extensibility was already identified at the beginning of the design – the DisMis framework was to act as a limited demonstrative system that could be extended at a future time to better suit the purposes of certain application area. In order to support extensibility, the system structure was planned to be composed of fairly independent components that are loosely coupled with each other. The system was planned to be composed of:

1) a communication service component that has protocol subcomponents in it,
2) system service components that utilise each other and the communication service component,
3) platform service components that utilise system services, and
4) application service components that utilise system service components and platform service components.

Additionally the communication between the distributed components was chosen to follow a simple XML – based schema so that extending it in the future would be straightforward.

4.4 Conceptual architecture of the design

According to the quality requirements presented in 4.3, the conceptual architecture of the DisMis framework was drawn together. The primary function of the DisMis framework is to produce middleware services for the use of distributed applications. The secondary but no less important function is to enable the distributed user applications to provide different services for each other so that a community of interacting distributed application components would be possible. In order to achieve those objectives, the responsibilities of the system were divided between the components presented in Figure 13.

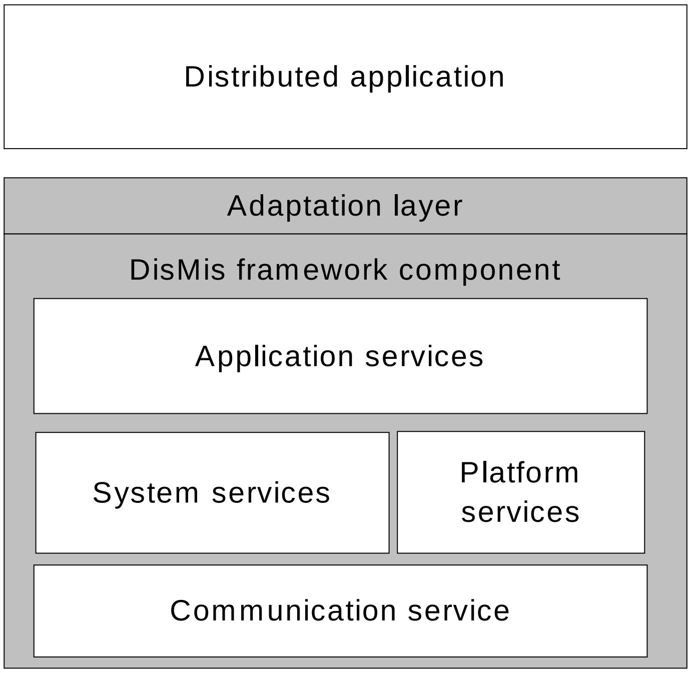


Figure 13. High-level conceptual view on the DisMis framework.

The general responsibilities of the main components of the system are briefly given in the following:

- Adaptation layer – an interface component for connecting the DisMis framework with the application. The application services component is visible to user applications through the adaptation layer.

- Communication service – asynchronous data between the separated distributed components flows in and out through the Communication service. Several services use the communication services.
- System services – services that are essential for the internal operation of the DisMis framework.
- Platform services – services that are offered also to other DisMis framework instances present in the network. A combination of these services forms the DisMis platform.
- Application services – a combination of services that is applicable by the user application components and accessible through an API. Application service components are partly abstractions of the underlying platform services and system services, but they may also comprise active operation.

Distributed applications are built on the DisMis framework. The applications utilise the services of the DisMis framework through certain relations that have been implemented to the Adaptation layer. The adaptation layer is visible in Figure 13. It is the layer just below the application. The internal operation of the adaptation layer has been more precisely described in the concrete architecture, excluded from this thesis.

The simple deployment in Figure 14 presents three components of a certain distributed system that has been labelled as System X. The components have been connected together with a distribution media. All components of the System X have been built by utilising the DisMis framework. In practical operation, the DisMis framework instances are constructed along with the components of System X. Due to the operation of the DisMis framework instances, all of the distributed components of the System X will be spontaneously aware of each other. Each of the DisMis framework instances will have a unique identifier that makes it possible to distinguish them from each other.

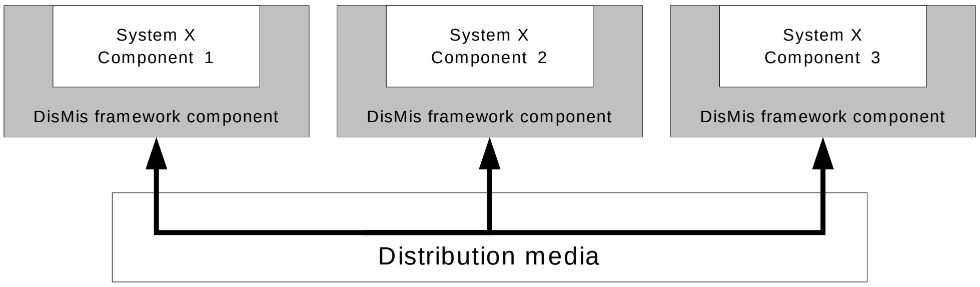

System X Component 1
DisMis framework component
System X Component 2
DisMis framework component
System X Component 3
DisMis framework component
Distribution media

Figure 14. A simple example of deployment.

The conceptual deployment of a distributed application or a system that uses the DisMis framework does not have to consider the physical deployment practically by any means. The only thing that has to be taken into account is that the devices running the software (i.e. the DisMis framework and the user application component built atop it) have to be able to communicate over a distribution media that is supported by the communication service of the DisMis framework. In accordance with that requirement, the restrictions of the distribution media have to be understood and conformed to by the user application.

4.4.1 Subsystem structure

The conceptual parts present in Figure 13 were further divided according to their responsibilities into the following conceptual subsystems that may be expressed as components.

Adaptation layer

- No subsystems

Application services

- Directory service user
- System services user

Platform services

- Directory service

System services

- Lease service
- Discovery service

Communication service

- XML – parser
- Protocol adapter
- Protocol components

Generic features that cross the subsystem borders have been discussed in this subchapter. Each of the subsystems will be discussed in the next subchapter that considers each conceptual component separately.

Fundamentals of the Platform Services

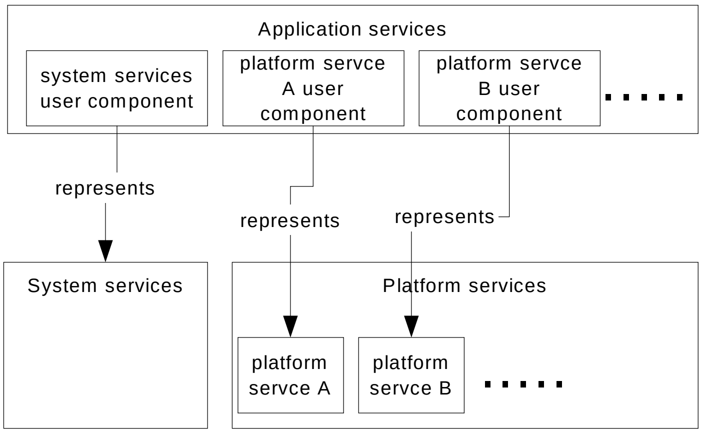


Figure 15. Application services comprise abstractions of underlying services.

One of the fundamental ideas in the distributed and decentralised middleware services is that the platform services do not necessarily have active operation in each of the DisMis framework instances that operate together within the connectivity of each other. It may be enough that only one DisMis framework instance in the network is providing a certain platform service, for example, the Directory service. To serve that purpose, each of the platform service components has a *user role* component that makes it possible to use the corresponding platform service, independent of its location. The operation of user roles is presented in Figure 16. There are also user roles for system services, which are always provided by the local DisMis framework instance.

Figure 15 presents the structure of the application services. The purpose of the application services is to provide a common point of attachment to the adaptation layer that has the task of finally presenting all the services of the DisMis framework to the user application in a suitable form. System services are present in the application services through a System services user component. The System services user component always uses the local system services. Platform services are present in the application services through a Platform service user role component. A Platform services user role component may refer to either a local or a remote service.

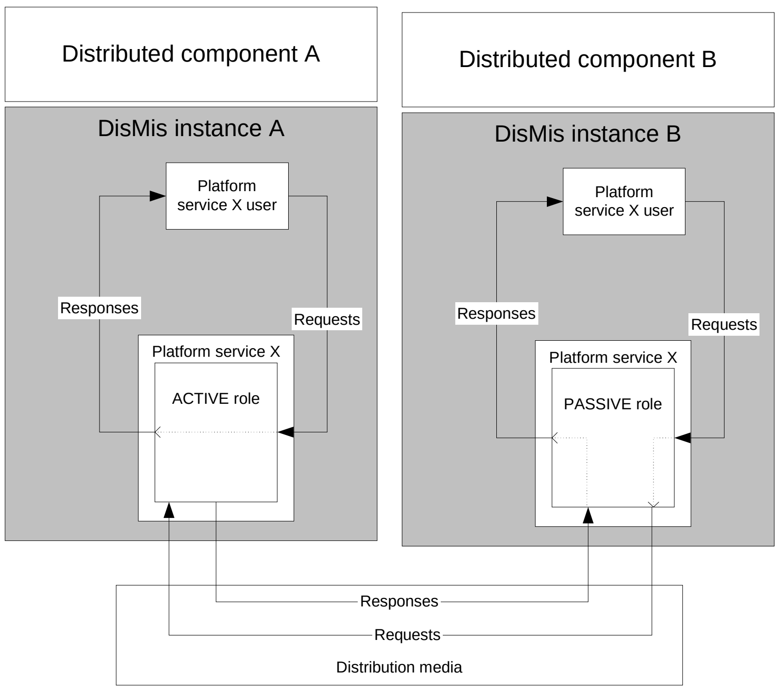


Figure 16. User roles hide the locations of the platform services.

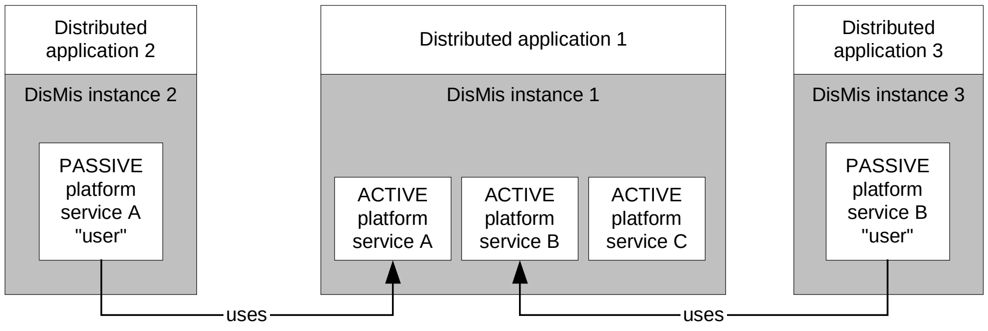

Distributed application 2
DisMis instance 2
PASSIVE platform service A "user"
Distributed application 1
DisMis instance 1
ACTIVE platform service A
ACTIVE platform service B
ACTIVE platform service C
Distributed application 3
DisMis instance 3
PASSIVE platform service B "user"
uses
uses

Figure 17. Passive and active service roles.

User role components of the platform services provide all the functionality that is required to use the platform service they represent. All the requests to use the service are first given to the local platform service component, which is then responsible for redirecting them if the corresponding platform service is not locally available. In that case, the task will be assigned to an active platform service component in another DisMis framework instance, as presented in Figure 16. Thus, the platform components introduce location transparency [4, 5]. The different forms of availability of the platform services can be represented by three possible cases:

1) The platform service may be locally present and active: thus locally available.
2) The platform service may be locally present but passive: not locally available.
3) The platform service may also be not locally present: not locally available.

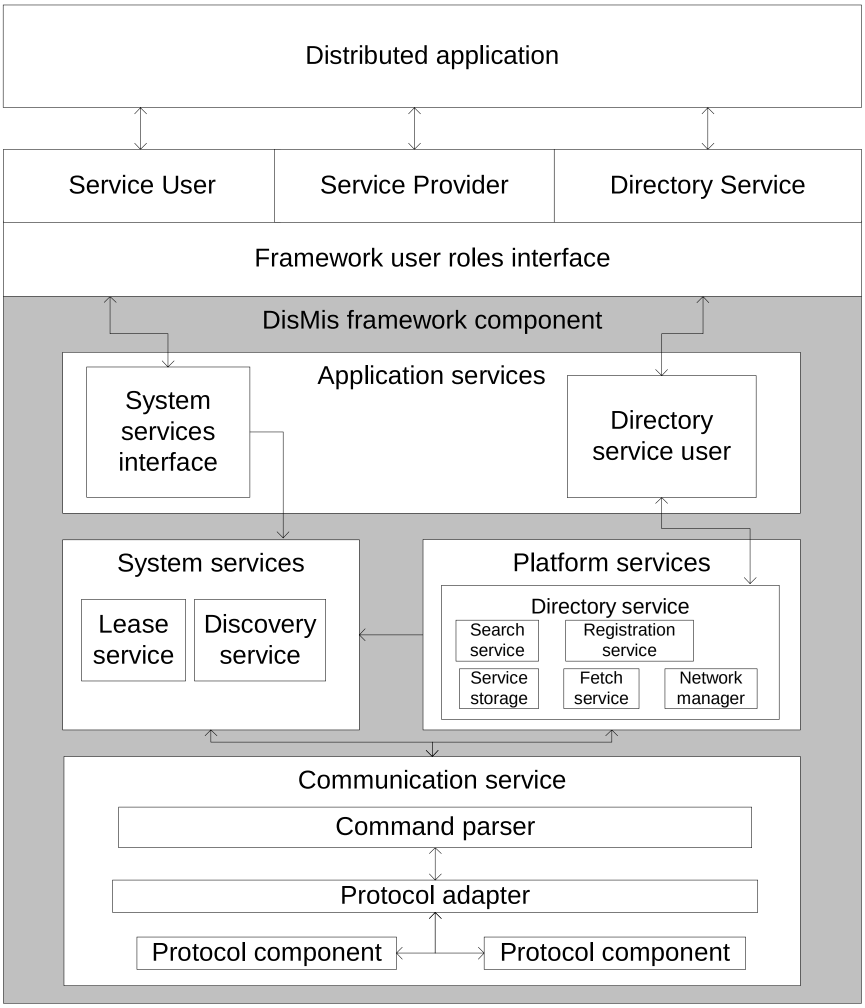


Figure 18. Relationships between the components in the system.

Active and passive roles of platform services make it possible to create DisMis-framework components that do not implement an active platform service, but only the passive role and the corresponding user component. Therefore, small devices that have more limited resources may execute applications that use lighter versions of the DisMis framework, but still have the same services available. This will, however, require a network of interconnected DisMis framework instances in which the required active platform services are always provided by some of those instances. Figure 17 illustrates a case in which the active platform services of a large unit are exploited by smaller devices that have more limited resources. Some platform services may also distribute the responsibility of producing the services to several DisMis framework instances at a time.

Figure 18 presents the relationships between all the components in the DisMis architecture. Arrows in Figure 18 represent the generic command relationships between the components. A component at the end of the arrow may be commanded, and accordingly utilised, by the component from which the arrow originates. Most of the relations are bi-directional.

The interface to the distributed user application itself takes place through an adaptation layer. It represents all that is necessary for the distributed application to know about the DisMis framework, in a suitable form. In the presented DisMis architecture, the adaptation layer is a class hierarchy. According to it, the distributed application is extended from one of the abstract base classes that implement the DisMis framework in practice. The adaptation layer is implemented as the *Framework user roles interface*, which is visible on top of the DisMis framework component in Figure 18.

In the DisMis architecture presented, the amount of application services was restricted only to the *directory service user* and to the *system services interface*. Many other possible and planned services, such as transaction service and synchronous messaging service were left for further development. An increase in the possible set of platform services would, in turn, increase the amount of choices in the adaptation layer. At present, three is adequate to represent the repertoire offered on a necessary scale. The consequent framework user roles have been labelled as follows:

> *Service user* is the most limited of the DisMis framework user roles. A *service user* does not provide any platform services, and the directory service user role is limited to search and fetch operations. Thus, a *service user* may only use the application services provided by other distributed applications, but it may not issue such services by itself. Additionally, a set of basic system services provided by the DisMis framework includes operations for sending and receiving messages between different DisMis framework instances in the network that is connected by the available,

suitable and supported communication media. Use of leased resources is also possible by utilising the services provided by the *service user* role.

Service provider has additional functionality in comparison with the *service user*. *A service provider* may use application services just as the *service user*, but it may also register its own services to the directory service and request leases to be applied by remote users utilising those services.

Directory service provider has additional functionality in comparison with the *service provider*. In addition to registering, looking for, and using application services, it may also host a directory service that provides services to other DisMis framework-based applications. Thus, in the presented DisMis architecture, *directory service provider* is the only role that may host an active platform service.

4.4.2 Component structure

After the overview given on the generic issues that cross the subsystem borders, this subchapter concentrates on the conceptual issues related to each of the subsystems in turn. Those conceptual issues comprise the responsibilities, the functional goal and the tasks that may be assigned to each of the subsystems present in the DisMis architecture. An overview of the conceptual architecture is also visible in Figure 18.

Application services subsystem

The application services subsystem consists of components with functionality that is visible to the distributed user application. The number of application service components may be increased at a later time. Currently there are two distinct components in it: the *System services interface* and the *Directory service user*.

The System services interface – component is responsible for converting the available system services so that they are applicable to the use of the distributed user application and possibly other components present in the application services subsystem. This is not merely a question of converting the required data structures or calling conventions, but the System services interface may also include active operation related to monitoring the use of the services combined with storing and caching of information when necessary.

The main functional goal of the System services interface component is to make the internal system services applicable to the distributed application in the most suitable way. Separate system services are applicable to each other most probably in some other way – in that sense, the question also concerns performing an interface conversion so that the system services do not have to expose those interfaces that have relevance only in the internal operation of the framework component. System services include the following operations:

- use of leases (see 3.2.2),
- provision of leases, and
- use of communication services to send and to receive information.

Directory service user – component comprises all the functionality that is required to use a Directory service (see 2.4.3), including:

- registering a service to the Directory service,
- unregistering a registered service from the Directory service,
- searching for a service from the Directory service with a certain search pattern, and
- fetching the service proxy data from the Directory service.

The Directory service user does not have to consider whether there is an active directory service instance in the local DisMis framework instance or not, that task is left to the Directory Service component itself. This has been presented in the fundamental idea of the platform services in 4.4.1. In that sense, the operation of the Directory service user component is very straightforward. Its main responsibility is to pass the requests and responses back and forth between the actual platform service component and the adaptation layer.

System services subsystem

System services are the fundamental services on which the core operation of the DisMis framework relies on. The most important difference between the system services and the application services is that the System services subsystem does not use any services from the Application services subsystem. Instead, system services only offer services to the application services. System services are also allowed to use the other system service components in their operation. In addition, the system services rely on the Communication service. There are currently two distinct components in the System services subsystem. Those components are the *Discovery service* and the *Lease service*.

The Discovery service – component (see 2.4.2) has the responsibility of discovering the nearby components (that are built by using the DisMis framework) when necessary, by utilising the multicast or broadcast capable media provided by the communication service. Additionally, the discovery component has the responsibility of announcing its presence to other components if required. This also takes place by utilising the broadcast or multicast capable media through the communication service. Discovery follows a certain functional pattern in exchanging information with another Discovery service components that reside in other DisMis framework instances. As a result of the operation of the Discovery service, it issues a list locally that comprises the applicable discovered components. The applicable components are, for example, active Directory service components, Lease service components, or other Discovery service components. In accordance with that, the Discovery service also advertises those local components that are present in the local DisMis framework instance to other, remote instances of the DisMis framework. Operation of the Discovery service is rather autonomous, and after its initialisation, other components may concentrate on utilising the list of available components, provided by the Discovery service.

The Lease service – component introduces the concept of leasing to the DisMis architecture. A lease is a contract between a resource and its user, as presented in 3.2.2. According to a lease, a certain resource is leased for use for a certain period of time. The Lease service has two primary purposes that introduce two distinct roles present in the concept of leasing:

1) Lease provider: The Lease service may be asked to provide a certain amount of leases to a certain resource. The leases may have various lengths.
2) Lease user: A lease user may ask for a lease from a lease provider (the Lease service). If the lease expires and it is not renewed, the lease user is no longer allowed to use the leased resource.

The Lease service has the responsibility of keeping count of leases that have been assigned to its responsibility. Typically the Lease service provides leases for the platform services that are available in the DisMis framework. If a lease to a service expires, the Lease service provides an announcement about it to the corresponding platform service, which may then discard the user of the service accordingly. Distributed applications built by using the DisMis framework may also provide or use their own leased resources.

Platform services subsystem

The platform services subsystem comprises services that may be utilised by the user application of the local DisMis framework instance, as well as by the user applications of remote DisMis framework instances. Platform services of multiple DisMis framework instances together form a single Distributed Middleware Service (DisMis) platform that is applicable for all the involved DisMis framework instances as well as for the user applications that are running on them. Platform services use the Communication service and the System services subsystems in order to fulfil their task.

Platform services have two modes of operation. Those modes are simply either *active* or *passive*. A platform service that has an active role in some of the DisMis framework instance may be considered an instance of that service, residing at the same location as the DisMis framework instance. The active platform service instance then serves other framework components that have the *user role* for that platform service.

In its passive mode of operation, the platform service has the task of locating an active platform service of similar kind from its neighbourhood. This takes place by utilising the Discovery service. When a suitable service has been located, the passive platform service will assign the tasks given to it to the active component that has been found. Currently, the only platform service available is the Directory service.

The Directory service – component is responsible for gathering information from the surrounding network of interconnected DisMis framework instances by utilising the Discovery service. According to that information, it will either enable or disable its active mode of operation. (See 2.4.3 for more information on directory services).

Active Directory service operation is started if there is not enough Directory services in the network discovered. On the other hand, if there are too many Directory services in the network, some of them will implicitly be shut down.

A Directory service has the following responsibilities during its active operation:

1) It has to accept registrations and store the accompanying proxy objects from the surrounding application components.
2) It has to keep count whether the registered application components are still available and remove registrations if necessary.
3) It has to react to search requests by browsing through the services registered within it that match requested attributes and then by responding with a list that comprises the matching services.
4) It has to respond to fetch requests by sending the requested proxy objects as a response.

In the passive mode of operation, the Directory service component is responsible for locating an active Directory service from the network of framework components and then for passing the commands to it, as described earlier in this chapter.

The Directory service component comprises the following subcomponents: *Network Monitor*, *Fetch Service*, *Search Service*, *Registration Service*, and *Service Storage*. The operation of the Directory service component is in practice provided by them. Tasks of those subcomponents have been presented in only in the concrete architecture, excluded from this thesis.

Communication service subsystem

The Communication service subsystem provides the DisMis framework with a uniform way of transporting data and command from one DisMis framework instance to another. Its fundamental purpose is to provide implicit connectivity, as described in 2.4.1, between different DisMis framework nodes and the software elements within them. The Communication service is used internally by the subcomponents of the DisMis framework as can be seen from Figure 18. User applications may also transfer information by using the Communication service through the system services abstraction in the Application services subsystem.

Before any data can be transported, it is has to be converted into a uniform type that may be produced, sent, received, and understood easily. Internal representation of a programming language, the use of serialised objects for example, is not usually such an easily understandable form of representation. A tagged XML-style data format, on the other hand, is more suitable for that purpose. The use of XML-style tagged language has been encouraged in order to make the transformed information more understandable, even though the practical choice of ending up with any other form of encoding the information would not have had any significant architectural relevance to the level of conceptualisation presented.

Thus, when data is received, it is first transported to the internal representation, described by the DisMis architecture, and then passed on to the corresponding receiver component, depending on the message content. Each component that belongs to system services or to platform services has the right to sign itself up as a message receiver in the Communication service. When a message to a registered message receiver arrives at the Communication service, it is then passed on to the corresponding receiver component.

The Communication service may implement several methods of transporting information through different media and with different protocols. This takes place through an abstraction provided by the *Protocol adapter* component. The Protocol adapter component provides two tasks:

1) It abstracts the sending of data through multiple media to a simple and conceptual level, in which only the relevant subjects have significance, namely the identifier of the destination and the message content.
2) It combines all the data received from the protocol components and passes them on to the XML parser to be transformed into the local representation.

Protocol components are connected to the Protocol adapter, all with a similar interface that provides the necessary functionality but hides the involved complexity.

4.5 Comparison of DisMis architecture with the technologies presented

As stated in chapter 3, there are similarities between the technologies presented (3.1-3.3) and the solutions provided by the DisMis architecture. In order to bring out the influence of the existing solutions, this subchapter lists the most important of the existing relations and links them with the features that have been chosen for the DisMis architecture. The most important dissimilarities have also been identified. The similarities and dissimilarities have been presented respectively in Table 3 and Table 4.

Table 3. Similar features of the DisMis architecture and the existing solutions.

Technology	Similarities
CORBA	1. Both solutions enable component interaction in a distributed environment. 2. Both solutions offer a certain core set of middleware services that act as fundamental building blocks when constructing distributed applications [15, pp. 151–242]. 3. A unique handle given for the elements of distribution is the only information required to reach that element in a distributed environment [15, pp. 15, 24–25]. 4. Both solutions introduce a single protocol for the information interchange between distribution nodes (ORBs in CORBA). In CORBA, that standard is the General Inter-Orb Protocol, GIOP [45, 15, pp. 102–104]. In DisMis architecture, the name of the protocol has not been specified.
COM/ DCOM	1. Both solutions enable component interaction in a distributed environment. 2. Both solutions have the ability to choose a suitable medium for communication between separate devices in a distributed environment [18, p. 439]. 3. Both solutions comprise a runtime entity that is responsible for locating the functional elements of distributed applications. That entity is called the Service Control Manager (SCM) in COM [18, pp. 62, 334]. The DisMis framework component performs the same action through the Directory Service.
EJB	1. Both solutions enable component interaction in a distributed environment. 2. Both solutions enable a plug-and play assembly of software components [20, p. 6]. 3. Both solutions have functionality for locating other functionality from the present service environment [20, pp. 36–39]. 4. Both solutions provide a framework for building software components that are used in constructing a certain system [20 pp. 21–31].
RMI	1. Both solutions enable component interaction in a distributed environment 2. Both solutions have a certain registry that stores bindings between implementations and a certain set of parameters that describes them. In RMI that registry is called the RMI registry [27, p. 42]. The DisMis framework uses the Directory Service for the same purpose. 3. Both solutions, when used with Java, encourage the use of mobile code in distributing functionality between elements of distributed applications.

Jini	1. Both solutions enable component interaction in a distributed environment. 2. Both solutions enable spontaneous networking. 3. Both solutions provide a versatile Directory Service that can store different kinds of information entities and look them up when given certain attributes. 4. Both solutions use leasing in keeping count of registrations and service usage. 5. Both solutions provide the means to perform a discovery operation in a broadcast-capable medium.
OSGi	1. Both solutions enable component interaction in a distributed environment. 2. Both solutions connect several means for exchanging information. 3. Both solutions are able to operate with devices that use point-to-point connectivity methods. 4. Both solutions comprise a service account that may be searched for a suitable service. In OSGi that service account is called the service registry [31, pp. 31–37].
UPnP	1. Both solutions enable software interaction in a distributed environment. 2. Both solutions provide the means for spontaneous, peer-to-peer connectivity of devices. 3. Both solutions provide a way of advertising services and resources in a spontaneous environment. 4. Both solutions provide a way of delegating the task of advertising services and resources to a certain entity that may be accessed by most of the network nodes. 5. Operation of UPnP is dependent on certain control points that have to reside in one of the devices in the network of systems that form a UPnP network. DisMis architecture is not dependent on any centralised service elements, but a directory service is a similar element to the UPnP control point.
Gnutella	1. Both solutions enable spontaneous peer-to-peer connectivity that is based on the assumption that the knowledge about the presence and location of the peers is received during the operation of the system. 2. Both solutions enable information transfer in peer-to-peer communities. 3. Both solutions distribute information about the information that one or several nodes in the network is delivering (DisMis directory service vs. Gnutella protocol). 4. Both solutions provide the means for searching for information from a distributed community.

Table 4. Dissimilar features between existing solutions and the DisMis architecture.

Technology	Similarities
CORBA	1. CORBA uses objects as elements of distributed composition of applications. The DisMis framework connects complete components. 2. CORBA provides interconnectivity for distributed objects by using RPC [15, pp. 14–16, 28–29]. The DisMis framework does not support the means for RPC or connectivity on object-level. 3. CORBA uses static elements called Object Request Brokers (ORBs) in connecting the elements of distributed applications. The DisMis framework component creates a dynamic service infrastructure that is completely included in the components that utilise the framework. 4. CORBA relies on TCP/IP in its operation [15, pp. 103–104]. DisMis architecture supports the use of various media.
COM/ DCOM	1. COM uses instances of COM classes, called COM objects, as elements of distributed composition of applications [18, pp. 39, 10–26]. DisMis architecture connects complete components. 2. COM programming model requires a certain set of base services from the platform to support its operation [18, pp. 26–27]. DisMis architecture requires only access to a distribution medium through protocol components. 3. DCOM provides interconnectivity for distributed objects by using RPC [18, pp. 439–478]. The DisMis framework does not support RPC or connectivity on the object-level. 4. DCOM utilises a static infrastructure that has to be supported by the infrastructure in which the COM applications are executed. DisMis framework component creates a dynamic infrastructure that is completely included in the components that utilise the framework. 5. Interfaces to distributed entities have to be defined by using the MIDL. DisMis architecture does not provide the means for interface definitions. A common understanding has to be reached on the application level by utilising the related programming model.
EJB	1. EJB relies on TCP/IP in its communication. DisMis architecture does not constrain the distribution medium [20, pp. 51–59]. 2. EJB provides a container for the operation of software encapsulated within beans. DisMis architecture does not provide a container for software to execute in. 3. EJB defines a multi-tiered client-server model that is based on either a Session Beans model or an Entity Beans model [20 pp.21-31]. The DisMis architecture does not provide a model for software component interaction. Instead it provides the means for peer-to-peer kinds of solutions to access each other's resources.
RMI	1. RMI provides the means for connectivity on the object-level. DisMis architecture supports connectivity only on the component level through message passing. 2. RMI relies on TCP/IP in its operation. DisMis architecture is not bound to any specific distribution medium.

Jini	1. Jini does not support a means for building a dynamic service infrastructure. Instead, static services are used. DisMis architecture always relies on a dynamic service infrastructure that may be established by any component in the network. It is seen as a more suitable solution for a truly spontaneous environment. 2. Jini relies on Java RMI in most of its internal operation [24, 26], thus binding the solution to TCP/IP. DisMis architecture encapsulates its commands into simple messages that are transferred by using the internal Communication Service, thus enabling the use of any medium in its internal operation.
OSGi	1. OSG is a static element in the network of distributed appliances [31, pp. 11–12] Thus, the infrastructure related to it cannot be established in a dynamic manner. 2. OSGi provides a context for the services to operate in a single entity called the Service Management Framework. DisMis architecture connects several equal distributed components that are connected by a suitable medium.
UPnP	1. UPnP is not a framework for applications, it is a protocol and a reference for applications that are willing to implement that model. 2. UPnP is based on common conventions about the used protocols between devices. DisMis framework tries to reach a level that is independent of the protocols. 3. UPnP control points do not have the capability of replacing each other's operation in case of a failure or shutdown. In the DisMis framework, robustness has been achieved through the replication of data and services.
Gnutella	1. Gnutella is not a framework for applications, it is a protocol. 2. Implementations of the Gnutella protocol do not provide any means for building applications or functionality that are based on them. 3. Gnutella does not offer any other modes of communication than searching and distributing the files that are stored within the community. 4. Gnutella does not provide any services that would be established as functional entities or instances. DisMis architecture creates an adaptive distribution service for at least one node of the network.

4.6 Validation of the architecture

No implementation of the DisMis architecture has been built as such. The architecture has been validated with separate implementations, each of which covers one or several parts or design decisions present in the architecture of the DisMis framework. There are also components whose operations have not been validated through implementations, but through analysis of design decisions and patterns present in existing commercial or freely available solutions. This subchapter presents an implementation of the Dynamic Distribution Platform (DDP), which is the most important single source of practically implemented validation for the DisMis framework.

4.6.1 Dynamic Distribution Platform

The Dynamic Distribution Platform (DDP) was an implementation of a framework with many similar features to the DisMis framework presented in this thesis. The author’s contribution to the ideas, design, and implementation of the DDP was 50% of the total work. Thus, the DDP may be considered as the first pilot implementation of the concept of distributed distribution services presented. That also makes it the most important source of validation and inspiration for the work done in this thesis. The DDP framework is presented as validation that considers all of the subsystem components (presented in 4.4.1 and 4.4.2) within the DisMis framework.

Description of the implementation

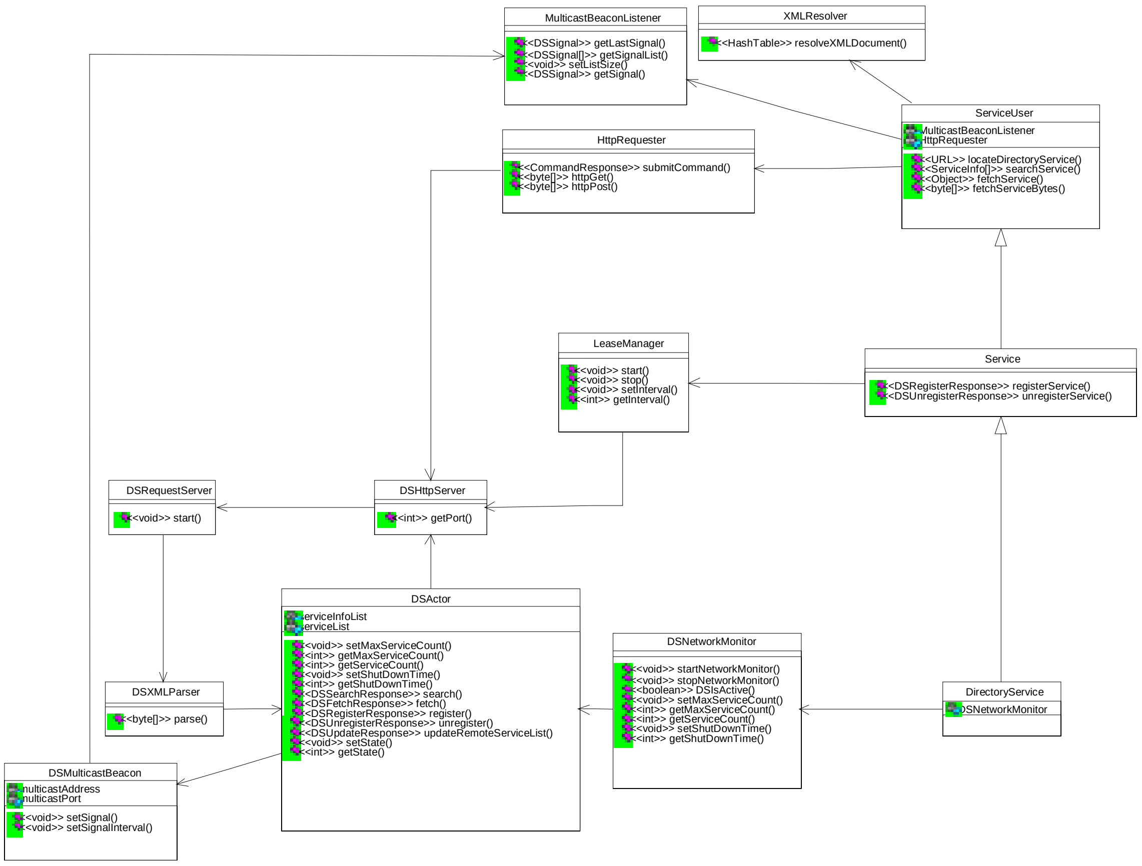
The original motivation for the DDP was an idea to build a framework for distributed applications which would comprise all the functionality that is required for the task of distribution. Therefore, an application that has been built by utilising the framework would not require any other external components in performing the act of distribution. The independence over any external service provider has been reached by distributing the services that are required for the distribution.

In the DDP, the actual distribution services are the discovery service and the directory service. Distribution of the distribution services has been implemented so that each of the participating nodes in a network of distributed applications should be able to host any of the distribution services. Thus, if a service is not available for a node that hosts an application or a piece of an application, the required service may always be created by the DDP framework instance on which the software in the node has been built upon. Activation of the services takes place automatically without any user intervention and it is therefore fully transparent to the application or its user. Transparent in this context means that the application or the user does not have to know whether the utilised distribution service, for example the directory service, is located in and hosted by the local node, or any other node, as long as there is a node that hosts the service.

Another important goal of the DDP was to demonstrate an approach that makes it possible to connect the pieces of distributed applications that may reach each other over various means of communication. That goal was not fully satisfied in the final design in which the nodes may reach each other only through IP-based network connections. In the DisMis architecture, however, the idea of the ability to communicate over different media has been preserved and concretised.

Several issues that were in the original plan for design and implementation of the DDP were left unanswered. Many of those questions have been answered in the DisMis architecture presented in this thesis. The most important of those issues were:

1) The implementation supported only IP-based means of communication.
2) There was no intelligent control or means of transferring the responsibility of a distribution service from a node to another.
3) There was no means of securing the information located by a node that hosts a service.
4) There was no means of securing a quick recovery of a distribution service in a situation of sudden failure.



MulticastBeaconListener
<<DSSignal>> getLastSignal()
<<DSSignal[]>> getSignalList()
<<void>> setListSize()
<<DSSignal>> getSignal()
XMLResolver
<<HashTable>> resolveXMLDocument()
ServiceUser
MulticastBeaconListener
HttpRequester
<<URL>> locateDirectoryService()
<<ServiceInfo[]>> searchService()
<<Object>> fetchService()
<<byte[]>> fetchServiceBytes()
HttpRequester
<<CommandResponse>> submitCommand()
<<byte[]>> httpGet()
<<byte[]>> httpPost()
LeaseManager
<<void>> start()
<<void>> stop()
<<void>> setInterval()
<<int>> getInterval()
Service
<<DSRegisterResponse>> registerService()
<<DSUnregisterResponse>> unregisterService()
DSRequestServer
<<void>> start()
DSHttpServer
<<int>> getPort()
DSActor
serviceInfoList
serviceList
<<void>> setMaxServiceCount()
<<int>> getMaxServiceCount()
<<int>> getServiceCount()
<<void>> setShutDownTime()
<<int>> getShutDownTime()
<<DSSearchResponse>> search()
<<DSFetchResponse>> fetch()
<<DSRegisterResponse>> register()
<<DSUnregisterResponse>> unregister()
<<DSUpdateResponse>> updateRemoteServiceList()
<<void>> setState()
<<int>> getState()
DSNetworkMonitor
<<void>> startNetworkMonitor()
<<void>> stopNetworkMonitor()
<<boolean>> DSIsActive()
<<void>> setMaxServiceCount()
<<int>> getMaxServiceCount()
<<int>> getServiceCount()
<<void>> setShutDownTime()
<<int>> getShutDownTime()
DSXMLParser
<<byte[]>> parse()
DirectoryService
DSNetworkMonitor
DSMulticastBeacon
multicastAddress
multicastPort
<<void>> setSignal()
<<void>> setSignalInterval()

Figure 19. A diagram representing the core classes and their relations in the DDP.

The class diagram of the core classes of the DDP is presented in Figure 19. In the class diagram of Figure 19, the structure that corresponds to the user roles-hierarchy also present in the DisMis architecture may be seen on the right hand side. The user-role classes e.g. ServiceUser, Service, and DirectoryService utilise some or all of the core classes e.g. XMLResolver, MulticastBeaconListener, HttpRequester, LeaseManager and DSNetworkMonitor, seen on the left side of the user role classes. Those core classes utilise further the DSHttpServer, DSActor, DSRequestServer, DSXMLParser and DSMulticastBeacon. As a result, the framework performs the actions of discovery, leasing, directory service, message parsing, and HTTP client and server operation.

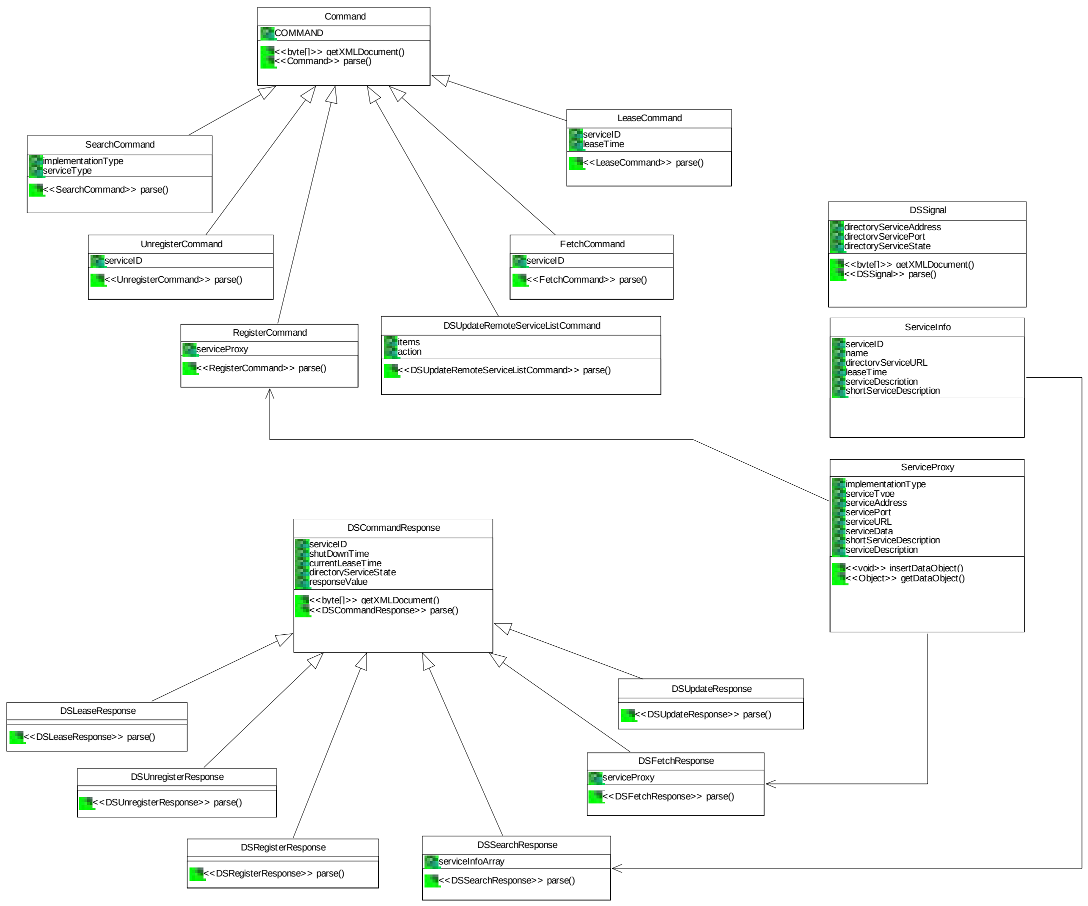


Figure 20. The message classes used in the DDP.

The message classes used in the parsing and processing of the requests as well as in responding to them, have been shown in Figure 20. The message classes have been divided into two taxonomic groups: *requests* and *command responses*. There is also a special message class, called ServiceProxy. An instance of a ServiceProxy may contain a serialised instance of a Java class whose class file has been located in another node within the range of TCP. A functional instance of the enclosed class may be extracted from the ServiceProxy instance. During the activation of any method in the imported proxy, JVM is guided to download the class file that contains the actual code from a remote location by using a separate class loader. The use of the ServiceProxy class is the main issue in transferring functionality in a distributed system when utilising the DDP. However, because the architecture and functionality of the DDP framework is not the main focus of this thesis, those topics beyond this brief introduction will be left without further consideration.

Validation by implementation

The conceptual architecture of the DDP has many similarities to the DisMis architecture presented in this thesis. The most important of them have been identified and listed in Table 5. Design of the DDP was, however, deficient in the sense that no high-level conceptual architecture was drawn. The design was implementation-centric and started from concrete architectural things like class diagrams and interface definitions. Accordingly, responsibilities between different concepts were left undefined right from the beginning. Difficulties in understanding all the design rationale in the later phases of the implementation made it impossible to introduce large modifications.

For the author the DDP was, however, essential practice towards understanding the meaning and importance of appropriate architectural design in software systems. The approach presented in the design concerning the DDP is a practical example of a bottom-up style of building software architectures and systems. When many issues that are related to the design are unknown at the time of the design, some of those decisions may have to be made without precise knowledge about their effect on the later phases of the design. In the case of DDP, the bottom-up style was the only appropriate method since no previous knowledge of building such systems was available at that time.

Despite the defects, there were also good practical design decisions in the DDP that have been imported from it to the design of the DisMis framework as such, or with slight modifications. Most of the similarities can be spotted from the following Table 5.

Table 5. Comparison of features between the DDP and the DisMis framework.

Service	**DDP**	**DisMis framework**
User roles interface	Three user roles implemented as a class hierarchy.	Three user roles implemented as a class hierarchy.
	No adaptation layer between the user roles and the framework implementation.	An adaptation layer between the user roles and the framework component makes it possible to create different kinds of interfaces to the framework.
Distributed Platform Services	Directory Service is the only platform service.	Directory service is the only platform service.
	No container component for the platform services. Each user role uses the components directly.	A container component for the platform services. Each user role gets to access the platform services through the Application Services component.
	Passive and active directory service roles, active directory service serves the passive ones.	Passive and Active platform service roles, active platform services serve the passive ones.
System Services	Lease Service and Discovery Service are the only System Services.	Lease Service and Discovery Service are the only System Services.
	System Services are not subcomponents of a System Services component.	System Services component has separate system services as subcomponents that act independently.
	System Services partly integrated to the framework main component.	System Services is a distinct component that is utilised by the platform services and the Application Services.
Lease Service	Lease Service is only used by the Directory Service in maintaining the leases of the owners of registered information.	Lease Service is used by the Directory Service and the Application Services component. Application Services component provides the Lease Service to the user applications.
	Lease Service is partly integrated with the Directory Service. A separate Lease Manager component takes care of the lease updates of a lease user.	Lease service is a subcomponent of System Services. It provides leasing services in a universal way so that any identifiable resource may be considered a leased resource.

Discovery Service	Discovery Service is a separate component that acts independently and provides discovery information for other components.	Discovery Service is a subcomponent of System Services. It provides discovery information for other components.
	Discovery Service is integrated with the UDP communication protocol and utilises Multicast in discovering framework instances.	Discovery Service is not bound to any specific communication protocol, instead, it utilises all the broadcast capable Protocol Components of the Communication Service.
Communication Service	Communication Service in practice bound to the HTTP and TCP protocols.	Communication Service may have several Protocol Components attached to it. New means of communication may be introduced by creating new Protocol Components.
	It is assumed that the communication between framework instances is synchronous.	It is assumed that the communication between framework instances is asynchronous.
	Some responsibilities of the Communication Service are also implemented by other components.	Communication Service is responsible for all communication.
	Communication Service is bound to the components that receive data, the message types are hardcoded to the software.	Components register within the Communication Service in order to receive messages. Within the registration, the components provide a way of performing the transformation between the internal Object representation and the XML form, used in transferring the messages.
	Messages are represented internally by objects that may be transformed to XML form and back at any time. Messages are transferred in XML form.	Messages are represented internally by objects that may be transformed to XML form and back at any time. Messages are transferred in XML form.

4.7 Summary of the validation

The example presented in this chapter was chosen especially according to its relevance in providing the initial motivation for the DisMis architecture. Although there were also other sources of inspiration and validation for the work presented in this thesis, the DDP may be considered as the most important example of them because it actually trials a concept that is very similar to the DisMis architecture. The most important contributions of the DDP for supporting the task of validation were:

- the overall concept of distributing middleware services,
- the operation of the discovery service,
- the operation of leasing and
- the operation of an XML-based message representation and serialisation.

There were also other related trial approaches and cases that have had an important task in validation. Those considered are such that have been implemented and primarily contributed by the author, but excluded from the discussion of this thesis. The main topics of those cases or trials have been listed below:

- A trial with a distribution framework for spontaneous networking with Jini.
- A trial with a distributed and spontaneous video camera application that utilises a distribution framework [46].
- An industrial case with a spontaneous UDP communication protocol and middleware.
- A trial with a Bluetooth connectivity component for Java.
- A trial with small devices with a mobile code interface [47].
- A trial approach with semantic middleware in spontaneous networking of software [48].

Validating an architectural concept through ready-made implementations that are more or less dissimilar with the one being validated, is not straightforward. Even though most of the implementations that were brought out have originally provided the necessary knowledge, innovations and inspiration for the DisMis architecture, a complete validation is not possible in the context of this thesis.

Additionally, the success of validation by such a method is not axiomatic. Especially if considering the act of validation as a step-by-step description of the system’s operation, proceeding mechanism by mechanism until all the technical details could be considered as trivially solved. Such an approach for validation is not suitable for the case with the DisMis architecture with the current sources of validation, most importantly because of the lack of a trial implementation or simulation of the DisMis platform itself.

Instead of such a complete description, the separate implementations used for validation have provided valuable information about the state of the research work brought into practice. Bringing them out here is an indication that many of the technical details considered in the DisMis architecture have been implemented in practice, one way or another, thus providing knowledge and necessary background to support the issues presented in this thesis.

There are issues in the DisMis architecture that have not been validated by any examples. Those topics can be divided into two categories. Either they are trivial for a person skilled in the art and therefore do not require further explanation, or they are such that the author has not implemented any experiments that could be related to them. The most important factors of those issues that lack experimental validation have been given in the following:

1) *Use of data or service replication in securing service availability*. The issue has been studied, for example, in [49, 50]. Further work would be required to suit the service and data replication scheme as presented in this thesis.
2) *Performing ad-hoc routing between nodes (DisMis framework instances) that have a distance of more than one hop, each of which occurs through a different medium*. The issue has been excessively studied, as illustrated by the examples and studies in [51, 52, pp. 53–295, 53, 54]. The presented DisMis architecture does not propose a solution for the question.
3) *Mechanisms for an error model*. The architecture developed so far gives less contribution to the development of an error model that would operate in a distributed and spontaneous environment. In practice, development of such a model, practices and related patterns would be crucial.
4) *The practical mechanism for decision making that considers the activation, shutdown and replication of platform services*. The required mechanisms are application specific and therefore no direct example may be drawn from the related research. However, appropriate methods might comprise, for example, the use of fuzzy logic [55], neural networks [56] or case-based reasoning [57]. Their applicability to the problems should be tested with simulations. Thus, further research is required.
5) *The best way of accepting the limitations caused by differences in the connection speeds of different media and protocols*. Here, the issue is strictly related to the particular operation of the DisMis framework. Therefore,

a straightforward and precise answer cannot be found from the related research. The problem is apparent, for example, when faced with a task of replicating a large directory service through a thin connection. In a case where the connection speed is not enough to keep up with the replication due to the amount of transactions that take place on the service, the service is in danger of being vulnerable to errors all the time. An excessive overload on a connection may also cause data transfer buffers to overflow, creating a requirement for a replication data filter of some sort.

6) *How to build software that utilises the DisMis framework in the most sophisticated way*. The topic has intentionally been left outside the discussion presented in this thesis, but it should be extensively studied in further research.

All the presented issues that lack experimental validation can generally be considered to be topics for further work. Further effort would also be required in order to actually implement and test a DisMis framework in practice. Some practical results, like processor load compared with the node amount in a network, or practical performance and robustness of a DisMis system, could not otherwise be achieved without a complete simulation. Accordingly, practical work and implementations are most evidently required.

The validation of the DisMis architecture is not completed within this chapter. Similarly, the architecture of the DisMis framework is not completed within this thesis. Further development of the concept that would also comprise implementations is an interesting and challenging task to accomplish. To answer the questions of a complete solution, a complete architecture, or a complete implementation can be considered the goal that the task of the related further research is set out for from this day forward.

5. Conclusion

An imaginable, perfect instance of distributed, decentralised middleware would always be available to its user despite the spontaneous nature of its environment and without any compromises whatsoever. When building the DisMis framework architecture, it was soon found out that it is quite impossible to produce a system that would create such a perfect instance of distributed middleware. Adaptability in a spontaneous environment is a trade-off with the simplicity of the system structure that produces it. Similarly, it is a trade-off between the ease of use of the middleware that maintains it. Extreme adaptability is likely to lead to the development of complicated mechanisms and administration. The trade-off shows explicitly in the level of abstraction that the DisMis framework offers to a software developer who is utilising the framework. As a consequence of the trade-off, the application developer has to face the spontaneous environment, thus it cannot be completely hidden or made transparent.

When the environment or domain of software development changes radically e.g. from static to spontaneous, the correct way to build software architecture for that environment is likely to change as well. All the features present in the model of building software for static environments cannot be used in their existing form when building software for spontaneous and dynamic environments. Similarly it is impossible to use the concept of transparency as an adapter that would convert between the requirements issued by either static or spontaneous domains. The task of the DisMis framework can be considered an attempt to provide the functionality of such an adapter. As an implication, it is impossible to completely solve the task. However, it would be an important benefit if a distributed middleware could provide the developer of spontaneous systems with even a bit more static, and therefore a more familiar environment to work in.

Current requirements demand that software has to be produced to operate in ever more demanding environments. The increasing complexity and dynamics of the operating environment of a software unit should not make its development *proportionally* more difficult. In that sense, a more static environment is precisely what the DisMis framework attempts to provide, static elements to an environment that might otherwise be completely dynamic.

One of the overall goals of the work presented here has been an attempt to provide simplicity for a software developer. If a solution fails to appear simple to the end user, its applicability will suffer severely despite its technical content. On the contrary, some solutions and tools have succeeded in reaching a symbolic value that may be understood just by giving an abstract explanation that comprises only a few phrases. Similarly, the DisMis framework should appear to the user as an abstract and simple tool. The goal of simplicity has mainly been provided within the DisMis architecture by restricting the amount of operations that a user might have to face when using the DisMis framework. Another point that considers simplicity is a logical point of attachment for the user applications. Those issues of simplicity have been considered within the complete architectural development of the DisMis framework.

Another goal of the work was that it should provide practically relevant solutions and content to the field of distributed computing. Currently (2002), there are no middleware solutions that would produce distributed middleware services in a completely decentralised manner. At least such solutions do not exist to the knowledge of this author. This thesis provides insight into the design challenges faced when producing such services. The solutions, models and ideas that have been provided here should also be applicable in the use of industry. The amount of distributed software in the use of industrial applications is vast and increasing. Inflexibility faced in rearranging and reconfiguring the software components or products may lead to financial losses. Approaches adopted from the DisMis architecture might provide new insight into the development of industrial products that carry a long legacy of doing things in the one and only "right" way. There may be also other "right" ways.

When evaluating the approach presented here, it should be considered that the particular purpose of the work has been to find new ways of doing things, and the applications have been mainly research related. Accordingly, some requirements such as real-time characteristics have not been considered. It is evident that some of those characteristics that have not been considered in the DisMis architecture might appear crucial in an industrial environment. However, it should not be seen as a restriction to the applicability of the ideas proposed here. A software architecture is a trade-off between the required quality attributes. In industrial use, the same original ideas might have led to a partly different architecture, fulfilling some requirements better and probably neglecting some of the existing ones.

The use of certain patterns, such as embedding middleware services within a framework that is a fundamental building block of all the distributed elements in a network of software, has been shown to be useful and possible in the work that has been presented in this thesis. The purpose of the complete process that has led to the architecture and evaluation presented, has been to further develop the concept of distributed middleware services. In that sense, the architecture proposed in the presented work has served as an excellent sandbox for developing ideas. Additionally, it has provided an excellent toolkit for new ideas that have proven useful in implementing software architectures for more practical purposes in cases related to both industry and research.

The pervasive and ubiquitous distributed systems of the future will be based on fundamental concepts that are so far mostly concealed. As an example, a way of seeing the composition of a future middleware solution has been provided in [48] in the form of a proposal for future research. Even though the future concepts and solutions would provide any sophisticated, conceptually high-level applications, the requirement for basic middleware services will still remain. Many of the present middleware solutions fail to address such dynamism that may be imagined to exist as a basic requirement of such future applications. As the basis for such systems, an approach adopted from the concept of distributed middleware such as presented in this thesis, might provide a valuable and flexible tool.

This thesis has presented a conceptual look on the results of work in which an architectural documentation for a distributed middleware service platform has been produced. The proposed model is one step towards the middleware of the future, characterised by the requirements set by ever-increasing dynamism. Related long-term work is likely continue in one form or another. More detailed views and aspects on presented kinds of distributed middleware architectures will be published accordingly in due course.

References

[1] Coulouris, G., Dollimore, J. & Kindberg, T. (2001). Distributed Systems Concepts and Design. 3rd ed. Addison-Wesley, Harlow, Essex. 772 p. ISBN 0-201-619-180

[2] Sun Microsystems, Inc. (17.11.2002). Java Homepage. URL: http://www.java.sun.com

[3] Fuggetta, A., Picco, G.P. & Vigna, G. (1998). Understanding Code Mobility. IEEE Transactions on Software Engineering 24(5), pp. 342–361. ISSN 0098-5589

[4] International Organization for Standardization, International Electrotechnical Commission (1998). Open Distributed Processing - Reference Model: Overview 1746-1:1998(E).

[5] Anthony, R.J. (2001). A Taxonomy of Transparency and a Dependency Graph. In: Proceedings on IASTED International Conference on Applied Informatics, Symposium 3. Software, February 19-22, Innsbruck, Austria, pp. 692-699. ISBN 0-88986-320-2

[6] Berson, A. (1996). Client/Server Architecture. 2nd ed. McGraw-Hill, New York, 569 p. ISBN 0-07-005664-1

[7] Bluetooth SIG, Inc. (17.11.2002). The Official Bluetooth Website. URL: http://www.bluetooth.com/

[8] Chatschik, B. (2001). An overview of the Bluetooth wireless technology. IEEE Communications Magazine 39(12), pp. 86-94. ISSN 0163-6804

[9] Institute of Electrical and Electronics Engineers, Inc. (17.11.2002). IEEE 802.11 Wireless Local Area Networks. URL: http://grouper.ieee.org/groups/802/11/

[10] Stallings, W. (2001). IEEE 802.11: Moving Closer to Practical Wireless LANs. IT Professional 3(3), pp. 17–23. ISSN 1520-9202

[11] Infrared Data Association (17.11.2002). Infrared Data Association Homepage / Standards. URL: http://www.irda.org/standards/specifications.asp

[12] Droms, R. University of Southern California, Information Sciences Institute (17.11.2002). Dynamic Host Configuration Protocol, RFC 2131, Draft Standard. URL: ftp://ftp.isi.edu/in-notes/rfc2131.txt

[13] Edwards, W.K. (1999). Core Jini. Prentice Hall, Inc., Upper Saddle River, New Jersey. 772 p. ISBN 0-13-0114469

[14] Object Management Group (OMG) (17.11.2002). Object Management Group Homepage. URL: http://www.omg.org

[15] Siegel, J. (1996). CORBA fundamentals and programming. John Wiley & Sons Inc., New York, 693 p. ISBN 0-471-12148-7

[16] Microsoft Corporation (17.11.2002). The COM Specification version 0.9. URL: http://www.microsoft.com/com/resources/comdocs.asp

[17] Pyhäluoto, T. (1997). Ohjelmistokomponenttien rajapintojen kuvaaminen. VTT Research Notes 1816. VTT Technical Research Centre of Finland, Espoo. 55 p. + app. 22 p. ISBN 951-38-5091-9

[18] Eddon, G. & Eddon, H. (1998). Inside Distributed COM. Microsoft Press, Redmond, Washington. 582 p. ISBN 1-57231-849-X

[19] Sun Microsystems, Inc. (17.11.2002). Java Beans API Specification version 1.01. 114 p. URL: http://java.sun.com/products/javabeans/docs/spec.html

[20] Jubin, H. & Friedrichs, J. (1999). Enterprise JavaBeans by example. International Business Machines Corporation (IBM), Upper Saddle River, New Jersey. 223 p. ISBN 0-13-022475-8

[21] Berg, C.J. (1998). Advanced Java Development for Enterprise Applications. Prentice-Hall Inc., Upper Saddle River, New Jersey. 584 p. ISBN 0-13-080461

[22] Szyperski, C. (1998). Component Software - Beyond Object-Oriented Programming. ACM Press, Addison-Wesley, New York. 411 p. ISBN 0-201-17888-5

[23] Pree, W. (1994). Design Patterns for Object-Oriented Software Development. ACM Press, Addison-Wesley, Reading, Massachusetts. 268 p. ISBN 0-201-42294-8

[24] Sun Microsystems, Inc (17.11.2002). Jini Architecture Specification Revision 1.2. 26 p. URL: http://wwws.sun.com/software/jini/specs/

[25] Sun Microsystems, Inc. (17.11.2002). Java Remote Method Invocation Specification Revision 1.8. 112 p. URL: ftp://ftp.java.sun.com/docs/j2se1.4/rmi-spec-1.4.pdf

[26] Sun Microsystems, Inc. (17.11.2002). Jini Technology Core Platform Specification Revision 1.2. 126 p. URL: http://wwws.sun.com/software/jini/specs/

[27] Oaks, S. & Wong, H. (2000). Jini in a Nutshell. O'Reilly & Associates, Inc., Sebastopol, California, 400 p. ISBN 1-56592-759-1

[28] Liang, S. & Bracha, G. (1998). Dynamic Class Loading in the Java Virtual Machine. In: Proceedings on Object-Oriented Programming Systems Languages and Applications (OOPSLA'98), 18-22 October, Vancouver, Canada. Addison-Wesley. Pp. 36–44. ISBN 0-201-30989-0

[29] Sun Microsystems, Inc. (17.11.2002). Java Object Serialization Specification for Java2 SDK Standard Edition v1.4. Beta 2. 74 p. URL: ftp://ftp.java.sun.com/docs/j2se1.4/serial-spec.pdf

[30] Open Service Gateway Initiative (OSGi) (17.11.2002). OSGi homepage/Overview. URL: http://www.osgi.org/resources/spec_overview.asp

[31] Open Services Gateway Initiative (17.11.2002). OSGi Service Platform Release 2 Specification. URL: http://www.osgi.org/resources/spec_download2.asp

[32] Universal Plug and Play Forum (17.11.2002). Universal Plug and Play (UPnP) Homepage. URL: http://www.upnp.org

[33] Kastner, W. & Leupold, M. (2001). How Dynamic Networks Work: A Short Tutorial on Spontaneous Networks. In: Proceedings on 8th IEEE International Conference on Emerging Technologies and Factory

Automation, 15-18 October, Antibes-Juan les Pins, France, Vol. 1, pp. 295–303. ISBN 0-7803-7241-7

[34] Miller, B.A., Nixon, T., Tai, C. & Wood, M.D. (2001). Home Networking with Universal Plug and Play. IEEE Communications Magazine 39(12), pp. 104–109. ISSN 0163-6804

[35] Cheshire, S. Universal Plug and Play Forum (17.11.2002). Dynamic Configuration of IPv4 link-local addresses, IETF Internet Draft (Expired March 2000). URL: http://www.upnp.org/download/draft-ietf-zeroconf-ipv4-linklocal-01-Apr.txt

[36] Tilley, S. & DeSouza, M. (2001). Spreading knowledge about Gnutella: a case study in understanding net-centric applications. In: Proceedings on IWPC 2001, 9th International Workshop on Program Comprehension, 12-13 May, Toronto, Canada, pp. 189-198. ISBN 0-7695-1131-7

[37] Bass, L., Clements, P. & Kazman, R. (1998). Software Architecture in Practice. Addison-Wesley Longman Inc., Reading, Massachusetts, 452 p. ISBN 0-201-19930-0

[38] Hofmeister, C., Nord, R. & Soni, D. (2000). Applied Software Architecture. Addison-Wesley Longman Inc., Reading, Massachusetts, 397 p. ISBN 0-201-32571-3

[39] Software Engineering Standards Committee of the IEEE Computer Society, Institute of Electrical and Electronics Engineers (IEEE), Inc. (21.9.2000). IEEE Std 1471-2000: IEEE Recommended Practice for Architectural Description of Software Intensive Systems. 23 p.

[40] International Organization for Standardization, International Electrotechnical Commission (1991). Information technology - Software Product Evaluation - Quality characteristics and guidelines for their use. ISO/IEC 9126:1991 A 2.6.1.

[41] Software Engineering Standards Committee of the IEEE Computer Society, USA (31.12.1998). IEEE Std 1061-1998: IEEE standard for a software quality metrics methodology.

[42] Niemelä, E. (1999). A Component Framework of a Distributed Control Systems Family. VTT Publications 402. VTT Technical Research Centre of Finland, Espoo, 188 p. + app. 68 p. ISBN 951-38-5549-X

[43] Vaskivuo, T. (2001). The Infrastructure of Interactive Devices in a Future Home. In: Proceedings on Dreaming for Future, Future Home Conference, 17–19 May, Helsinki, Finland. Yliopistopaino, Helsinki. 10 p.

[44] Matinlassi, M., Niemelä, E. & Dobrica, L. (2002). Quality-driven architecture design and quality analysis method. A revolutionary initiation approach to a product line architecture. VTT Publications 456. VTT Technical Research Centre of Finland, Espoo, 128 p. + app. 10 p. ISBN 951-38-5967-3

[45] Chung, E., Huang, Y., Yajnik, S., Liang, D., Shih, C., Wang, C.-Y. & Wang, Y.-M. (1998). DCOM and CORBA Side by Side, Step by Step, and Layer by Layer. C++ Report 10(1), pp. 18–30. ISSN 1040-6042

[46] Vaskivuo, T. (2001). A Framework for an Easy Jini Extension Demonstrated with a Video Camera Example. In: Proceedings on Java/Jini Technologies, 19-24 August, Denver, Colorado, Vol. 4521. SPIE, Washington. Pp. 134-145. ISBN 0-8194-4245-3

[47] Vaskivuo, T., Tikkala, A. & Latvakoski, J. (1999). Ohjain ja sen ohjausmenetelmä. Finnish Patent No. 109951. 31.10.2002, PRH, Helsinki.

[48] Vaskivuo, T. & Latvakoski, J. (2002). Semantic Middleware for Spontaneous, Context-Aware and Adaptable Systems. In: Proceedings on WWRF#6 meeting, 25–26 June, London, 8 p.

[49] Hongisto, M. (2002). Mobile data sharing and high availability. Diploma Thesis. Department of Electrical Engineering, University of Oulu, Oulu, 84 p.

[50] Sun, G. & Mori, K. (1999). Flexible and autonomous service replication technique. In: Proceedings on IEEE International Conference on Systems, Man, and Cybernetics, 1999. IEEE SMC '99, 12–15 October, Tokyo, Japan, Vol. 3, pp. 113-118. ISBN 0-7803-5731-0

[51] Park, V. & Corson, M. (1997). A Highly Adaptive Distributed Routing Algorithm for Mobile Wireless Networks. In: Proceedings on INFOCOM '97. Sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Driving the Information Revolution., 7–11 April, Kobe, Japan. Vol. 3, pp. 1405–1413. ISBN 0-8186-7780-5

[52] Perkins, C.E., ed. (2001). Ad Hoc Networking. Addison-Wesley, Upper Saddle River, New Jersey. 370 p. ISBN 0-201-30976-9

[53] Perkins, C.E. & Royer, E.M. (1999). Ad-hoc On-Demand Distance Vector Routing. In: Proceedings on 2nd IEEE Workshop on Mobile Computing Systems and Applications, 25–26 February, New Orleans, Louisiana. Pp. 90–100. ISBN 0-7695-0025-0

[54] Perkins, C.E. & Royer, E.M. (1999). Multicast Operation of the Ad-Hoc On-Demand Distance Vector Routing Protocol. In: Proceedings on 5th annual ACM/IEEE International Conference on Mobile Computing and Networking. Pp. 207–218. ISBN 1-581-13142-9

[55] Driankov, D., Hellendoorn, H. & Reinfrank, M. (1993). An Introduction to Fuzzy Control. Springer-Verlag, Berlin. 316 p. ISBN 3-540-56362-8

[56] Haykin, S. (1994). Neural Networks: a Comprehensive Foundation. Macmillan, New York. 696 p. ISBN 0-02-352761-7

[57] Aha, D.W. (1998). The omnipresence of case-based reasoning in science and application. Knowledge-Based Systems 11(5–6), pp. 261–273. ISSN 0950-7051

Published by

Series title, number and report code of publication

VTT Publications 490
VTT–PUBS–490

Author(s)
Vaskivuo, Teemu

Title

Software architecture for decentralised distribution services in spontaneous networks

Abstract

Factors that drive the design of distributed systems are experiencing a phase of rapid changes. Mobility and the new methods of interconnectivity brought along with it have to be faced by the fundamentals of distributed systems. Simultaneously, hardware tasks are being adopted by software, making it possible to make those system elements configurable that have traditionally been considered static. Spontaneous changes in configurations, connections, and physical environment are common factors that are increasingly brought along with the distributed systems design. This thesis considers an architecture for a software framework that faces those challenges by providing interconnectivity for distributed pieces of software in a new way.

The original idea presented here is to create middleware services that arise in a distributed and spontaneous manner from the interconnections of the interconnected, distributed pieces of software themselves. The complete independence of any centralised middleware service producer is the key issue in the proposed solution. Other issues are the means of communication over different media and the ability to assure the robustness of the provided services despite changes in the configuration or the presence of different software elements. The solution has been presented in the form of the software architecture of a proposed design. A major part of the introduced solutions has been validated by distinct cases related to both industry and research.

Keywords
distributed systems, ad-hoc networking, decentralised systems, framework, middleware

Activity unit
VTT Technical Research Centre of Finland, Vuorimiehentie 5, P.O.Box 2000, FIN–02044 VTT, Finland

ISBN 951–38–6034–5 (soft back ed.) 951–38–6035–3 (URL:http://www.inf.vtt.fi/pdf/)	Project number E2SU00082

Date	Language	Pages	Price
February 2003	English, Finnish abstr.	99 p.	B

Name of project	Commissioned by
Series title and ISSN VTT Publications 1235–0621 (soft back ed.) 1455–0849 (URL: http://www.inf.vtt.fi/pdf/)	Sold by VTT Information Service P.O.Box 2000, FIN–02044 VTT, Finland Phone internat. +358 9 456 4404,fax +358 9 456 4374

Julkaisija	Julkaisun sarja, numero ja raporttikoodi VTT Publications 490 VTT–PUB–490

Tekijä(t) Vaskivuo, Teemu			
Nimeke **Ohjelmistoarkkitehtuuri keskittämättömien hajautuspalveluiden ohjelmakehykselle spontaanisti muodostuvissa verkoissa**			
Tiivistelmä Hajautettujen järjestelmien suunnittelun vaikuttavat tekijät kokevat tällä hetkellä nopeita muutoksia. Järjestelmien perustoimintojen täytyy ottaa huomioon laitteiden liikkuvuus sekä sen mukanaan tuomat uudenlaiset yhteysmuodot. Aiemmin laitteistolähtöisesti ratkaistuja tehtäviä toteutetaan yhä enenevissä määrin ohjelmistolla, minkä ansiosta useita kiinteiksi käsitettyjä tekijöitä voidaan nykyisin pitää muunneltavina. Hajautettujen järjestelmien suunnittelussa täytyy uudenlaisten vaatimusten mukaisesti ottaa yhä useammin huomioon spontaanit muutokset ohjelmiston kokoonpanossa, kytkennöissä, sekä ohjelmis-toa suorittavan laitteen fyysisessä ympäristössä. Tässä diplomityössä käsitellään uutta hajautettujen järjestelmien ohjelmistokehyksen ohjelmistoarkkitehtuuria, joka kohtaa spontaanin ympäristön asettamia haasteita tarjoten uudenlaista ratkaisua hajautettujen ohjelmiston osien yhteistoiminnalle. Tässä työssä esitettävä alkuperäinen ajatus on luoda yhteen liitettyjen, hajautettujen ohjelmiston osien pelkästä yhteen liittämisestä syntyvä välitason ohjelmistokerros. Avainasia ratkaisussa on sen tarjoama riippumattomuus yhdestäkään keskistetystä välitason palveluiden tuottajasta. Muita työhön liittyviä aiheita ovat ratkaisut ohjelmien osien väliselle yhteydenpidolle eri tiedonsiirron välittäjien kautta, sekä kyky taata tarjottujen palveluiden saatavuus hajautetun järjestelmän ohjelmistojen osien sekä asetuksiltaan että läsnäololtaan vaihtelevasta toiminnasta huolimatta. Ratkaisu on esitetty ehdotetun kaltaista toimintaa toteuttavan ohjelmistoarkkitehtuurin muodossa. Pääosa esitetyistä ratkaisuista on vahvistettu osin tutkimuksessa, osin teollisessa ympäristössä sovellettujen esimerkkitapauksen avulla.			
Avainsanat distributed systems, ad-hoc networking, decentralised systems, framework, middleware			
Toimintayksikkö VTT Elektroniikka, Kaitoväylä 1, PL 1100, 90571 OULU			
ISBN 951–38–6034–5(nid.) 951–38–6035–3(URL: http://www.inf.vtt.fi/pdf/)			Projektinumero E2SU00082
Julkaisuaika Helmikuu 2003	Kieli Englanti, suom. tiiv.	Sivuja 99 s.	Hinta B
Projektin nimi		Toimeksiantaja(t)	
Avainnimeke ja ISSN VTT Publications 1235–0621 (nid.) 1455–0849 (URL: http://www.inf.vtt.fi/pdf/)		Myynti: VTT Tietopalvelu, PL 2000, 02044 VTT, puh. (09) 456 4404, faksi (09) 456 4374	

Tätä julkaisua myy	Denna publikation säljs av	This publication is available from
VTT TIETOPALVELU	VTT INFORMATIONSTJÄNST	VTT INFORMATION SERVICE
PL 2000	PB 2000	P.O.Box 2000
02044 VTT	02044 VTT	FIN–02044 VTT, Finland
Puh. (09) 456 4404	Tel. (09) 456 4404	Phone internat. +358 9 456 4404
Faksi (09) 456 4374	Fax (09) 456 4374	Fax +358 9 456 4374

ISBN 951–38–6034–5 (nid.)
ISSN 1235–0621 (nid.)

ISBN 951–38–6035–3 (URL: http://www.inf.vtt.fi/pdf/)
ISSN 1455–0849 (URL: http://www.inf.vtt.fi/pdf/)

VTT PUBLICATIONS 490 Software architecture for decentralised distribution services in spontaneous networks Teemu Vaskivuo