UNIKASSEL VERSITÄT

ComTec

Fachgebiet für Kommunikationstechnik

Prof. Dr.-Ing. Klaus David

Framework for middleware executed on mobile devices

vom

Fachgebiet Kommunikationstechnik
im Fachbereich Elektrotechnik / Informatik
der Universität Kassel

genehmigte

Dissertation

zur Erlangung des akademischen Grades eines
Doktor-Ingenieur (Dr.-Ing.)

von

Bjoern Wuest

geboren am 21. Mai 1976 in Delmenhorst, Deutschland

Kassel 2005

Erstreferent: Prof. Dr.-Ing. Klaus David
Koreferent: Prof. Jari Porras

Tag der Einreichung: 01. Juli 2005
Tag der Disputation: 22. September 2006

Zusammenfassung

Die ubiquitäre Datenverarbeitung ist ein attraktives Forschungsgebiet des vergangenen und aktuellen Jahrzehnts. Es handelt von unaufdringlicher Unterstützung von Menschen in ihren alltäglichen Aufgaben durch Rechner. Diese Unterstützung wird durch die Allgegenwärtigkeit von Rechnern ermöglicht die sich spontan zu verteilten Kommunikationsnetzwerken zusammen finden, um Informationen auszutauschen und zu verarbeiten. Umgebende Intelligenz ist eine Anwendung der ubiquitären Datenverarbeitung und eine strategische Forschungsrichtung der *Information Society Technology* der Europäischen Union. Das Ziel der umbebenden Intelligenz ist komfortableres und sichereres Leben.

Verteilte Kommunikationsnetzwerke für die ubiquitäre Datenverarbeitung charakterisieren sich durch Heterogenität der verwendeten Rechner. Diese reichen von Kleinstrechnern, eingebettet in Gegenstände des täglichen Gebrauchs, bis hin zu leistungsfähigen Großrechnern. Die Rechner verbinden sich spontan über kabellose Netzwerktechnologien wie w*ireless local area networks* (WLAN), *Bluetooth*, oder *UMTS*.

Die Heterogenität verkompliziert die Entwicklung und den Aufbau von verteilten Kommunikationsnetzwerken. *Middleware* ist eine Software Technologie um Komplexität durch Abstraktion zu einer homogenen Schicht zu reduzieren. Middleware bietet eine einheitliche Sicht auf die durch sie abstrahierten Ressourcen, Funktionalitäten, und Rechner.

Verteilte Kommunikationsnetzwerke für die ubiquitäre Datenverarbeitung sind durch die spontane Verbindung von Rechnern gekennzeichnet. Klassische Middleware geht davon aus, dass Rechner dauerhaft miteinander in Kommunikationsbeziehungen stehen. Das Konzept der dienstorienterten Architektur ermöglicht die Entwicklung von Middleware die auch spontane Verbindungen zwischen Rechnern erlaubt. Die Funktionalität von Middleware ist dabei durch Dienste realisiert, die unabhängige Software-Einheiten darstellen.

Das *Wireless World Research Forum* beschreibt Dienste die zukünftige Middleware beinhalten sollte. Diese Dienste werden von einer Ausführungsumgebung beherbergt. Jedoch gibt es noch keine Definitionen wie sich eine solche Ausführungsumgebung ausprägen und welchen Funktionsumfang sie haben muss.

Diese Arbeit trägt zu Aspekten der Middleware-Entwicklung für verteilte Kommunikationsnetzwerke in der ubiquitären Datenverarbeitung bei. Der Schwerpunkt liegt auf Middleware und Grundlagentechnologien. Die Beiträge liegen als Konzepte und Ideen für die Entwicklung von Middleware vor. Sie decken die Bereiche Dienstfindung, Dienstaktualisierung, sowie Verträge zwischen Diensten ab. Sie sind in einem Rahmenwerk bereit gestellt, welches auf die Entwicklung von Middleware optimiert ist. Dieses Rahmenwerk, *Framework for Applications in Mobile Environments* (FAME²) genannt, beinhaltet Richtlinien, eine Definition einer Ausführungsumgebung, sowie Unterstützung für verschiedene Zugriffskontrollmechanismen um Middleware vor unerlaubter Benutzung zu schützen.

Das Leistungsspektrum der Ausführungsumgebung von FAME² umfasst:

- minimale Ressourcenbenutzung, um auch auf Rechnern mit wenigen Ressourcen, wie z.B. Mobiltelefone und Kleinstrechnern, nutzbar zu sein
- Unterstützung für die Anpassung von Middleware durch Änderung der enthaltenen Dienste während die Middleware ausgeführt wird
- eine offene Schnittstelle um praktisch jede existierende Lösung für das Finden von Diensten zu verwenden
- und eine Möglichkeit der Aktualisierung von Diensten zu deren Laufzeit um damit Fehlerbereinigende, optimierende, und anpassende Wartungsarbeiten an Diensten durchführen zu können

Eine begleitende Arbeit ist das *Extensible Constraint Framework* (ECF), welches *Design by Contract* (DbC) im Rahmen von FAME² nutzbar macht. DbC ist eine Technologie um Verträge zwischen Diensten zu formulieren und damit die Qualität von Software zu erhöhen. ECF erlaubt das aushandeln sowie die Optimierung von solchen Verträgen.

Abstract

Ubiquitous computing is an appealing research area today and most likely the future. It is about unobtrusive computer support of users in their everyday activities. This support is achieved by an omnipresence of computers and their ability to spontaneously form distributed computing systems, and to exchange and process information. Ambient Intelligence, which is an application of ubiquitous computing, became a strategic research direction of the Information Society Technology programme of the European Union to form a *knowledge society*. The objective of Ambient Intelligence is to make life more comfortable and safer.

Distributed computing systems for ubiquitous computing are characterised by heterogeneity of the computers used. They range from small sized computers embedded into everyday items like cars, heaters, toasters and coffee cups, to high performance computers like servers and mainframes. They all connect spontaneously via wireless network technology, e.g. wireless local area networks (WLAN), Bluetooth, Universal Mobile Telecommunications System (UMTS), etc.

This heterogeneity increases the complexity of distributed computing systems, and as such their programming. Middleware is a software technology concept and software, to reduce the complexity by abstracting from heterogeneity and providing a homogeneous layer on top of the different, heterogeneous computers used in distributed computing systems. Middleware provides unified access to the heterogeneous resources, functionalities, and computers it abstracts from.

Distributed computing systems for ubiquitous computing are characterised by the spontaneous connection of computers. Traditional middleware is designed for distributed computing systems without spontaneous connections. Recently developed middleware follows the *service oriented architecture* concept that takes into consideration spontaneous connections. Middleware functionality is realised as services, which are independent software elements.

The Wireless World Research Forum (WWRF) has described services that future middleware may provide. These services are hosted by a service execution environment. Yet, there is a lack of definition of such service execution environment.

This dissertation contributes to middleware development for distributed computing systems in ubiquitous computing. The focus is on middleware and enabling technologies to implement middleware. Concepts for service discovery, service update, and contracts between services are presented. The concepts are provided in a framework that is designed specifically for realising middleware for distributed computing systems in ubiquitous computing. This framework, called *Framework for Applications in Mobile Environments* (FAME²), includes guidelines for service development, a proposal of a service execution environment, and support for different levels of access control to protect middleware from malicious use.

The service execution environment of FAME² features:

- low resource use on resource limited mobile devices like cellular phones and embedded computers
- reconfiguration of middleware by changing the set of services at the middleware's runtime
- an open interface to utilise virtually any service discovery solution to locate services in a distributed computing system
- online-update functionality for corrective, perfective, and adaptive maintenance of middleware and middleware services

An accompanying work is the *Extensible Constraint Framework* (ECF) that makes *Design by Contract* available in the context of FAME². Design by Contract is a technology to increase the quality of software systems created from software elements by formulating contracts between them. ECF enables negotiation and refinement of these contracts.

Acknowledgements

I would like to thank my supervisor, Professor Klaus David, for helpful and the many fruitful discussions. His probing questions continually challenged me.

I would like to thank Professor Jari Porras for being my second supervisor. Furthermore I would like to thank Professor Gerd Stumme and Professor Dirk Dahlhaus as members of the doctoral committee. I gratefully acknowledge their comments, discussions and being available for me. I also thank Professor Kimmo Raatikainen for reviewing my work.

Thanks as well to all my colleagues at the chair of communication technology. They are a continual source of pleasure and inspiration to me.

I would like to express my appreciation to the students I supervised during my graduation for all the work they did, for all their input, for their literature reviews, and their unwillingness to give up even at times when it seemed to be impossible to get everything done (on time).

I am very grateful to my family who encouraged me to press on with my thesis. I will never forget my grandfather who regarded me as a “doctor” a long time before I had even finished my degree.

I would like to thank my friends for cheering me on and the good times we had together.

Special thanks go to my parents for raising me, giving me an education and their support.

My greatest thanks to Li Wei, I will always love you. She gave me the energy and the inspiration needed to defeat the beast of dissertation.

Preface

A major part of the work presented in this dissertation has been performed in the context of industry public-private partnership, national research projects of “Wireless Internet (WI) Zellular” and “Wireless Internet mik21”, and European research project of the IST FP6 “mCDN”. The research projects “WI Zellular” and “mik21” are funded by the German ministry for research and education (Bundesministerium für Bildung und Forschung, bmb+f).
Several results constituted in this dissertation have been published at national and international conferences.

In chronological order, the publications are:

- B. Wuest, K. David, Wireless Internet Applications, VDE Mobilfunktagung, Osnabrück, 2001
- O. Drögehorn, B. Wuest, C. Deist, T. Hohmann, H. Lauer, K. David, Mobile Middleware for Heterogeneous Environments, Middleware 2003, 16-20 June 2003, Rio, Brazil
- B. Wuest, O. Drögehorn, K. David, Service Platform Model for Heterogeneous Mobile Access Networks, In Proceedings of the International Conference of Internet Computing, Las Vegas, USA, June 2003
- B. Wuest, O. Drögehorn, K. David, FAME2: Software Architecture for B3G and 4G networks, In Proceedings of the IST Mobile & Wireless Summit 2003, Aveiro, Portugal, June 2003, pp. 837-841
- B. Wuest, O. Drögehorn, K. David, The FAME² Platform Concept – Moving Platforms to the Mobile, In Proceedings of the International Conference of Internet Computing, Las Vegas, USA, June 2004
- B. Wuest, O. Drögehorn, K. David, A Management Layer for Mobile Middleware, In Proceedings of the GI-Jahrestagung 2004, Workshop on Mobile Computing und Medienkommunikation im Internet, Ulm, Germany, September 2004
- B. Wuest, O. Drögehorn, K. David, Service Discovery API for Pervasive Computing, In Proceedings of the International Conference of Internet Computing, Las Vegas, USA, June 2005
- B. Wuest, O. Drögehorn, K. David, Framework for platforms in ubiquitous computing systems, In Proceedings of 16th Annual IEEE International Symposium on Personal Indoor and Mobile Radio Communications (PIMRC), Berlin, Germany, September 2005

Table of contents

List of figures

List of tables

List of programs

1 Introduction

It can be observed that advances in computer hardware and mobile communications have led to numerous new uses for computers in distributed computing systems. Today, computers are embedded in everyday items, e.g. cars, heaters, toasters, coffee machines, cameras, microphones, etc. and are capable of connecting spontaneously via wireless network technology, e.g. wireless local area networks (WLAN), Bluetooth, General Packet Radio System (GPRS), and Universal Mobile Telecommunications System (UMTS). They support users in typical situations of their life, e.g. navigation, time management, information retrieval etc. Distributed computing systems have evolved from *simple* structures with a few types of computers, like Uniplex Information and Computing System (UNICS or UNIX) servers and terminals, to *complex* structures with many different types of computers and their platforms. The computing platform includes the microprocessor architecture, the bus system used for communication between the different hardware components, peripherals connected to the microprocessor, and the operating system managing the resources of the hardware and controlling the execution of software [1].

Complex distributed computing systems are used in various situations. Among them is ubiquitous computing. In the vision of ubiquitous computing, first introduced by Mark Weiser in [2], distributed computing systems are used to support users unobtrusively in their everyday activities.

> *Sal awakens; she smells coffee. A few minutes ago her alarm clock, alerted by her restless rolling before waking, had quietly asked, "Coffee?" and she had mumbled, "Yes." "Yes" and "no" are the only words it knows.*
> *Sal looks out her windows at her neighborhood. Sunlight and a fence are visible through one, and through others she sees electronic trails that have been kept for her of neighbors coming and going during the early morning. Privacy conventions and practical data rates prevent displaying video footage, but time markers and electronic tracks on the neighborhood map let Sal feel cozy in her street.*
> *Glancing at the windows to her kids' rooms, she can see that they got up 15 and 20 minutes ago and are already in the kitchen. Noticing that she is up, they start making more noise.*
> *At breakfast Sal reads the news. She still prefers the paper form, as do most people. She spots an interesting quote from a columnist in the business section. She wipes her pen over the newspaper's name, date, section and page number and then circles the quote. The pen sends a message to the paper, which transmits the quote to her office.*
> *Electronic mail arrives from the company that made her garage door opener. She had lost the instruction manual and asked them for help. They have sent her a new manual and also something unexpected—a way to find the old one. According to the note, she can press a code into the opener and the missing manual will find itself. In the garage, she tracks a beeping noise to where the oilstained manual had fallen behind some boxes. Sure enough, there is the tiny tab the manufacturer had affixed in the cover to try to avoid Email requests like her own.*
> *On the way to work Sal glances in the foreview mirror to check the traffic. She spots a slowdown ahead and also notices on a side street the telltale green in the foreview of a food shop, and a new one at that. She decides to take the next exit and get a cup of coffee while avoiding the jam.*
> *Once Sal arrives at work, the foreview helps her find a parking spot quickly. As she walks into the building, the machines in her office prepare to log her in but do not complete the sequence until she actually enters her office. On her way, she stops by the offices of four or five colleagues to exchange greetings and news.*
> *Sal glances out her windows: a gray day in Silicon Valley, 75 percent humidity and 40 percent chance of afternoon showers; meanwhile it has been a quiet morning at the East Coast office. Usually the activity indicator shows at least one spontaneous, urgent*

meeting by now. She chooses not to shift the window on the home office back three hours—too much chance of being caught by surprise. But she knows others who do, usually people who never get a call from the East but just want to feel involved. The telltale by the door that Sal programmed her first day on the job is blinking: fresh coffee. She heads for the coffee machine.

Coming back to her office, Sal picks up a tab and "waves" it to her friend Joe in the design group, with whom she has a joint assignment. They are sharing a virtual office for a few weeks. The sharing can take many forms—in this case, the two have given each other access to their location detectors and to each other's screen contents and location. Sal chooses to keep miniature versions of all Joe's tabs and pads in view and three-dimensionally correct in a little suite of tabs in the back corner of her desk. She can't see what anything says, but she feels more in touch with his work when noticing the displays change out of the corner of her eye, and she can easily enlarge anything if necessary.

A blank tab on Sal's desk beeps and displays the word "Joe" on it. She picks it up and gestures with it toward her live board. Joe wants to discuss a document with her, and now it shows up on the wall as she hears Joe's voice:

"I've been wrestling with this third paragraph all morning, and it still has the wrong tone. Would you mind reading it?" Sitting back and reading the paragraph, Sal wants to point to a word. She gestures again with the "Joe" tab onto a nearby pad and then uses the stylus to circle the word she wants:

"I think it's this term 'ubiquitous.' It's just not in common enough use and makes the whole passage sound a little formal. Can we rephrase the sentence to get rid of it?"

"I'll try that. Say, by the way, Sal, did you ever hear from Mary Hausdorf?"

"No. Who's that?"

"You remember. She was at the meeting last week. She told me she was going to get in touch with you."

Sal doesn't remember Mary, but she does vaguely remember the meeting. She quickly starts a search for meetings held during the past two weeks with more than six people not previously in meetings with her and finds the one. The attendees' names pop up, and she sees Mary.

As is common in meetings, Mary made some biographical information about herself available to the other attendees, and Sal sees some common background. She'll just send Mary a note and see what's up. Sal is glad Mary did not make the biography available only during the time of the meeting, as many people do.... – cited from [2]

An application for ubiquitous computing is *ambient intelligence*, described in [3]. The objective of ambient intelligence is to make life more comfortable and safer. In ambient intelligence (AmI), information is provided where needed, and the computing devices are integrated into the environment to support the users in an unobtrusive, seamless, and invisible manner. To illustrate AmI, four scenarios describe typical activities that a human does throughout the day. The scenarios cover typical situations of everyday life.

In the scenario *Maria – Road Warrior*, a business woman uses numerous locally provided services, e.g. car navigation, electronic passport control, cultural advisors, environmental controls etc. via her mobile device, her P-Comm. Maria's device, the P-Comm, knows about her preferences, has the capability to detect provided services, and configure them accordingly to Maria's preferences. In the scenario, excessive use of discovery and configuration of services, including automatic updates of them, is made.

In the *Dimitrios* scenario a mobile device, the D-Me, is closely attached to, or implanted into, the user. The D-Me interacts with other D-Mes autonomously to exchange information and carry out decisions on behalf of its user. An example of information exchange is that the D-Me automatically advises local navigation to some place with the help of D-Mes of other users without requesting their help. An example of autonomous decision is to block incoming telephone calls automatically

because the user had blocked this call several times in the past. Decisions are based on the past behaviour that the D-Me learned from his user. The major focus is on applying constraints on interaction of services, and negotiation based on profiled information.
The focus of the *Carmen* scenario is on logistics and transportation. The ambient intelligence manages travelling, monitors and controls traffic, adapts traffic regulations to current situations, warns of accidents and deducts fees from road users. Furthermore, AmI supports the users in logistics, i.e. shopping and supply with basic needs like food. In the scenario, excessive use of personal area networks with direct communication between users is done.
In the scenario *Annette and Solomon* users collaboratively interact with each other to exchange knowledge and experiences. The AmI supports users to find other users with similar interests and schedules meetings for the two of them. Besides collaborative activities, the major focus is on negotiation based on profiled information and on direct communication between users.

In the scenarios for ambient intelligence it can be seen that the mobile devices of the users are actively involved in the distributed computing system, i.e. run software. This results in increased complexity of modern distributed computing systems, caused by the use of numerous, technically incompatible computing platforms and wireless network technologies in one system. This heterogeneity results in a situation where software developed for such complex distributed computing systems needs to consider the characteristics of every computing platform and network technology [4]. The consequence is increased effort for developing software.
Following Bernstein, Middleware is a software technology concept, whose implementations reduce the effort required for programming software for distributed computing systems by providing a homogeneous layer on top of the heterogeneous computing platforms [1]. Middleware provides unified access to the resources provided by the computers, i.e. microprocessor, peripherals and operating system functionality. Additionally, middleware provides communication protocols for communication between the different computers in a distributed system, and unified access to these protocols. Middleware is placed between the computing platforms and application software. The reduced effort required for programming application software for distributed computing systems results from the fact that middleware is developed only once for a distributed computing system. Middleware then enables the support of other software developed on top of it.
Newly emerging distributed computing systems are characterised by many different types of computers and computing platforms, and their spontaneous connection using wireless technology. Middleware, as proposed by Bernstein, is designed for distributed computing systems without spontaneous connections through. Recently developed middleware in the area of ubiquitous computing follows a concept named service oriented architecture (SOA), taking into consideration spontaneous connections (and disconnections, of course). In SOA, the functionality of middleware is realised as services. Services are independent software elements that have no fixed dependency on specific other software elements. Application software and services are clients of services. Additionally, the SOA concept includes the possibility to discover services on demand of the client. This discovery feature is mandatory in distributed computing systems with spontaneous connections to enable clients to detect newly available services provided by emerging computers, and to find replacements of services provided by computers disconnected from the distributed computing system.
A particular challenge is the development of the middleware and its various services for ubiquitous computing. While SOA is a concept on how to organise middleware, it lacks clear guidelines for drafting its architecture, design, and implementation. This has resulted in a situation where numerous middleware has been developed, designed for specific purposes, but most likely incompatible with each other. The authors in [5] describe an approach for developing reusable middleware and its services. In this approach, called design principles, the middleware services are managed by a service execution environment (SEE). The approach defines clear guidelines for drafting the architecture and design of the SEE and the middleware services, and integrating them to

a middleware. However, this approach does not define the required functionality of the SEE or any services.
An opportunity are standardisation activities. The Wireless World Research Forum (WWRF) is one such standardisation body. WWRF has proposed services that future middleware for I-centric communication may provide [6-8]. I-centric communication is a research area related to ubiquitous computing. The WWRF is a pre-standardisation body that defines a strategic vision of future research directions and is a platform for global research and development collaboration in mobile and wireless systems [9]. Part of the definition of the WWRF is a component model based on the design principles presented in [5]. So called Generic Service Elements (GSEs) implement middleware services. GSEs are hosted by an SEE that implements management for GSEs. The objectives of this component model are:

- **Composing** GSEs to enable the creation of complex services based on previously created ones
- **Loose coupling** between GSEs to cooperate dynamically with each other across heterogeneous middleware technologies and different administrative domains
- **Distribution** of services that does not constrain on their discovery and use
- **Easy deployment** to incorporate a GSE with a minimum of (if not zero) additional cost in term of announcement, configuration, compiling, etc
- **Discovery** of the GSE and its purpose

A challenge is the heterogeneity of available resources of the different computers that are used in distributed computing systems of ubiquitous computing. The used computers may range from embedded devices with limited resources like wearable and small sensors, to general purpose servers and mainframes with virtually unlimited resources. To realise middleware that can be used on any of these devices, the SEE must take this heterogeneity of available resources into consideration. In [10], one approach proposed to achieve this is by limiting the SEE to mandatory features, and enabling extension of the SEE by services to enhance middleware to provide more sophisticated features.
What remains open are concepts for harmonisation and integration of existing technologies. As an example, there are numerous solutions existing for discovering services, nevertheless, they are not designed for integration or cooperation but instead aim to replace each other. Ubiquitous computing, on the other hand, aims towards integration and cooperation.

1.1 Contributions

This work contributes to middleware development for distributed computing systems in ubiquitous computing. The contributions cover areas of service discovery, service update, and contracts between services. They are provided in a framework, which is designed specifically for realising middleware that is executed on mobile devices in distributed computing systems in ubiquitous computing. This framework is called *Framework for Applications in Mobile Environments 2* (FAME²). The framework includes guidelines for service development, defines a service execution environment, and supports operation-level access control.
Service discovery deals with the locating and exploitation of services available in a distributed computing system. There are numerous existing solutions for service discovery. They are designed for use in specific distributed computing systems, with specific assumptions of their developers in mind. This variety of existing solutions is the reason why they fail to be useful in multiple distributed computing systems designed for ubiquitous computing [11]. At the same time, there might be no universal solution. Interoperability of different service discovery solutions enables clients to use the optimal service discovery solution in different distributed computing systems. Existing solutions for integrating service discovery are the *Support for Service Discovery and Interaction* [12, 13], the *Open Service Discovery Architecture* [14], and the *Service Discovery Framework* of the *Reflective Middleware for Mobile Computing* [15]. Their drawbacks are that they do not support the dynamic integration of new service discovery solutions, i.e. at runtime of middleware, or expect that service discovery solutions use a universal yet proprietary format for queries and responses. The *Open Service Discovery Interface* (OSDI), being a part of FAME², is a

solution for dynamic integration of service discovery solutions. New service discovery solutions can be integrated during runtime of middleware, there is no need to use a universal yet proprietary format for queries and responses, and OSDI is self-similar which means that service discovery is discoverable as well.
Most non-trivial software contains more or less severe errors and is subject to modification to increase performance, optimise resource use, integrate new functionality, and to support new hardware. Updating software is done to fix errors, to perfect software, and to adapt software. Usually, updating software requires stopping, updating, and then restarting it. Software for ubiquitous computing is usually highly distributed, and thus it is not possible to stop and restart it for update purposes. Online-update performs software updates without the need to stop and restart software. Existing solutions for online-update are replication, memory manipulation, platform manipulation, interaction modelling, dynamic interfaces, and the proxy structural design pattern. Replication is considered as unsuitable for ubiquitous computing because it usually doubles the resources required by software. Manipulation of memory and platform requires low-level access and highly specialised and proprietary solutions, which seems impractical to be realised for the large number of different platforms being used in distributed computing systems in ubiquitous computing. Interaction modelling and dynamic interfaces usually require human intervention, opposing with the objective of unobtrusive user support by ubiquitous computing. The proxy pattern is a solution introducing minimal overhead, and can be applied to existing platforms without modification, and it enables online-update without user interaction [16-19]. However, the proxy pattern interferes with the publish/subscribe communication mechanism. Publish/subscribe is used for communication between loosely coupled services that supports flexible communication links [20, 21]. Subscribers subscribe at publishers to receive messages sent by the publisher. The concept of mutable reference endpoint is proposed to harmonise the proxy pattern with publish/subscribe. It enables updating subscription information at publishers without involvement of publishers and subscribers.
One of the primary objectives of creating software from services is reusability. Reuse is limited by the uncertainty of suitability [22-24]. Formulating contracts between services and its clients reduce this uncertainty [25, 26], and is called *Design by Contract* (DbC). The authors of [22] classify contracts into four types: basic, behavioural, synchronisation, and quality-of-service contracts. There are different concepts for implementing DbC: pre-compilers, special compilers, runtime instrumentation, documentation, wizards, and aspect oriented programming. The shortcomings of existing solutions are the incompleteness of supported contracts, e.g. not supporting formulation of quality-of-service contracts, the lack of negotiation and refinement of contracts at the runtime of services, and the intrusive nature that requires a full specification of all possible contracts before a service is deployed. The *Extensible Constraint Framework* provides a solution for formulating all four types of contracts, is unobtrusive, and facilitates negotiating and refining contracts at runtime of services.

1.2 Scope of the doctoral thesis

"as is often the case with trends in the IT industry, the term component has too many meanings" – cited from [27]

This section defines the scope of the thesis by defining the more frequently used terms. These terms then form the natural boundary of this doctoral thesis.

Definition 1: Component
In [28-30], a component is a software object, meant to interact with other components, encapsulating certain functionality or a set of functionalities. A component has a clearly defined interface and conforms to a prescribed behaviour common to all components within an architecture.

Developers assemble components to applications, and the ensemble of components does not change after the development of the application. As a consequence, components do not need to be discovered. When the components of an application are distributed throughout a network, the components need to be located, i.e. the addresses of the components need to be resolved, but a discovery in the sense that there might be different components providing the same functionality is not foreseen.
The concept *component oriented architecture* means that software is developed with components.

Definition 2: Component container
Components are executed within a *component container*.

> *"Containers are the interface between a component and the low-level platform-specific functionality that supports the component." – cited from [31]*

Depending on the approach, a component container may execute one or many components. The component container provides common functionality that is usually required by all components. Examples of such common functionality are persistency, communication protocols, configuration, monitoring, etc.

Definition 3: System

> *"Systems are groups of interrelated components designed to collectively achieve a desired goal or goals." – cited from [32].*

Thus, the system is the entity that encompasses all relevant elements. Such elements could be users, components, their containers, etc.

Definition 4: Platform / computing platform

> *"By platform, we mean a set of low level services and processing elements defined by a processor architecture and an OS's API, such as Intel x86 and Win-32" – cited from [1]*

Following this definition, a platform is the combination of operating system and hardware.

Definition 5: Service

> *"A service is a discoverable set of components accessible via one interface." – cited from [33]*

Services are providers of functionality. The difference between a component and a service is that services are discoverable, i.e. the ensemble of interacting services is defined at runtime and not at development time as it is the case with components [34, 35].

Definition 6: Service execution environment
In [5] it is proposed that services be executed within a service execution environment (SEE). An SEE has a similar task as a component container: hosting services and providing common and often needed functionality. In addition to component containers, the functionality of SEE includes discovery of other services and making the hosted services discoverable.

Definition 7: Service oriented architecture
Developing software using services is called *service oriented architecture* (SOA). In SOA, functionality is provided by services. Clients can use these services to implement their functionality. Services are published and clients can discover them [36].
Middleware developed using the concept of SOA is called *service oriented middleware* (SOM).

Definition 8: Middleware
Middleware is everything between applications and platforms.

> *"These services are called ‚middleware services', because they sit ‚in the middle,' in a layer above the OS and networking software and below industry-specific applications." – cited from [1]*

> *"Middleware is the slash (/) between client and server. It is the glue that lets a client obtain a service from a server." – cited from [37]*

Definition 9: Framework
> *"a component framework is a dedicated and focused architecture, usually around a few key mechanisms, and a fixed set of policies for mechanisms at the component level." – cited from [38]*

Frameworks help in creating systems by their guidelines, policies and mechanisms they define.

Definition 10: Mobile computing
> *"Mobile Computing is using a computer (of one kind or another) while on the move." – cited from [39]*

> *"Leichte, tragbare aber dennoch leistungsfähige Rechner werden drahtlos vernetzt und transparent in Kommunikations- und Informationsinfrastrukturen eingebunden." – cited from [40]*

As a result, mobility is an important aspect of mobile computing. Resources are immobile in mobile computing. These resources are consumed by mobile devices, which act as clients only.

Definition 11: Ubiquitous computing
> *"Ubiquitous computing names the third wave in computing, just now beginning. First were mainframes, each shared by lots of people. Now we are in the personal computing era, person and machine staring uneasily at each other across the desktop. Next comes ubiquitous computing, or the age of calm technology, when technology recedes into the background of our lives. Alan Kay of Apple calls this "Third Paradigm" computing." – cited from [41]*

Ubiquitous computing adds the aspect of mobility to mobile devices, which served as consumers of resources in mobile computing. Thus, mobile devices do not only act as clients and consume resources, but instead will provide their resources to others as well. With calm technology, the technology fades into the background to become "invisible".

The key idea behind ubiquitous computing is:

> *The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it. – cited from [3]*

Definition 12: Ambient Intelligence
> *"The concept of Ambient Intelligence (AmI) provides a vision of the Information Society where the emphasis is on greater user-friendliness, more efficient services support, user-empowerment, and support for human interactions. People are surrounded by intelligent intuitive interfaces that are embedded in all kinds of objects and an*

> *environment that is capable of recognising and responding to the presence of different individuals in a seamless, unobtrusive and often invisible way." – cited from [4]*

The objective of AmI is to make life more comfortable and safer. For AmI, systems need to be sensitive, responsive, interconnected, contextualised, transparent, and intelligent.

The scope of this doctoral thesis will be ubiquitous computing, and in particular the aspect of mobile resources. The different research topics of service discovery, online-update, and extensible constraint framework contribute to the support of mobility in ubiquitous computing by:

1. find available resources,
2. support adaptation and maintenance of software, and
3. improve the quality of interaction of software by using dynamic contracting mechanisms.

1.3 Structure of this doctoral thesis

The thesis is composed of seven chapters. This chapter introduces the vision of ubiquitous computing given by Mark Weiser. Since its introduction, ubiquitous computing has been an appealing area of research. In the introduction, a motivation for new approaches for middleware development is given. Among these approaches are new concepts for services and service execution environments, integrating service discovery, online-update functionality, and contracts for service interaction. The introductory chapter also gives a brief overview of the contributions of this dissertation, and includes a collection of definitions that are used throughout the thesis.

Chapter 2 focuses on frameworks for middleware development. Section 2.1 provides an overview of existing concepts and state of the art. In Section 2.2 the *Framework for Applications in Mobile Environments 2* (FAME²) is described. An evaluation and comparison of FAME² with other existing frameworks is given in Section 2.3.

Chapter 3 focuses on service discovery in general, and integration and interoperation of different service discovery solutions in particular. Existing service discovery solutions and approaches for their integration and interoperation are presented in Section 3.1. The *Open Service Discovery Interface* is an approach for seamless and flexible integration and interoperation, and is described in Section 3.1.10. In Section 3.3 a comparison and evaluation is given.

A solution for online-update is described in Chapter 4. This solution, called mutable reference endpoints, harmonises the proxy structural design pattern and the publish/subscribe communication mechanism. Usually, this pattern and communication mechanism are incompatible. The concept of mutable reference endpoint enables unobtrusive online-update of services while the services communicate using the publish/subscribe mechanism, thus harmonising these both, pattern and communication mechanism.

Chapter 5 outlines the possibilities for using explicit contracts for interaction of services. A review of state of the art in Section 5.1 results in the requirement for a different approach for defining and enforcing contracts in loosely coupled software based on the concept of service oriented architecture. This different approach, the *Extensible Constraint Framework*, is described in Section 5.2. In Section 5.3 the approach is discussed and evaluated.

Possibilities for future work are outlined in Chapter 6. The purpose of this chapter is to indicate future research directions, and to list still existing limitations for middleware development for distributed computing systems in ubiquitous computing.

Chapter 7 presents the conclusions and final remarks.

2 Framework for building middleware for ubiquitous computing

In the vision of ubiquitous computing, among other devices, mobile devices, like smart phones, personal digital assistants, etc., actively participate in distributed computing systems. Active participation means that mobile devices offer their resources to other devices, e.g. by executing software on their behalf. Middleware reduces the effort required for programming software for distributed computing systems by providing a homogeneous layer on top of heterogeneous computing platforms [1, 42-43]. Middleware has the following functions [44]:

- *Hiding distribution, i.e. the fact that an application is usually made up of many interconnected parts running in distributed locations;*
- *Hiding the heterogeneity of the various hardware components, operating systems and communication protocols;*
- *Providing uniform, standard, high-level interfaces to the application developers and integrators, so that applications can be easily composed, reused, ported, and made to interoperate;*
- *Supplying a set of common services to perform various general purpose functions, in order to avoid duplicating efforts and to facilitate collaboration between applications.*

To let software, which is running on mobile devices, benefit from middleware, that middleware needs to be executed on mobile devices as well. Consequently, middleware needs to consider the characteristics of mobile devices and distributed computing systems in ubiquitous computing. These characteristics are:

- **Heterogeneity:** The high number of different mobile device types and their capabilities, e.g. smart phones and personal digital assistants, with or without camera, global positioning system devices, etc., requires that middleware is reconfigurable to make advantage of the capabilities of the different devices [45-47]. *Reconfigurable middleware* adapts to the capabilities of the devices it is executed on, and to the requirements of applications. Other terms for reconfigurable middleware are *adaptive middleware* and *reflective middleware*. Optimally, middleware is reconfigurable without its restart and enables reconfiguration while it is in use.
- **Limited resources:** Mobile devices have, in comparison to stationary devices like general purpose servers, limited memory, central processing unit (CPU) power, battery energy, and storage capacity. This requires that middleware does not make excessive use of resources to leave as many resources as possible for applications [48-50].
- **Wireless connectivity:** Mobile devices are usually connected to a distributed computing system using a wireless network connection, e.g. WLAN (like IEEE 802.11b), Bluetooth, GPRS, or UMTS. On the one hand this wireless connectivity facilitates mobility of the mobile devices, i.e. they may change their physical location. On the other hand wireless connectivity is susceptible to disconnections from the distributed computing system. Other reasons for disconnections can be forced disconnections, for example by the user to save battery energy. This requires that middleware is aware that disconnections can happen, and includes strategies to counter them [51, 52, and 354].

As a result, middleware should be:

- **reconfigurable** to adapt to different platforms and situations, including disconnections
- **resource-optimising** to make best use of limited resources
- **executeable on mobile devices**, taking into account limited resources

Frameworks aid to implement such middleware. Frameworks provide guidelines (dedicated and focused architecture), templates and patterns (key mechanisms), and standard solutions for typical functions of middleware, e.g. reconfigurability, resource preserving strategies, disconnection management, etc. (fixed set of policies and mechanisms). A definition of the term framework is given in [38] as following:

> *"a component framework is a dedicated and focused architecture, usually around a few key mechanisms, and a fixed set of policies for mechanisms at the component level." – cited from Szyperski [38]*

The development of middleware that is running on mobile devices is accelerated and simplified by frameworks, because common and repeating tasks need to be done only once for the framework, and are then used in different middleware implementations. This chapter presents the *Framework for Applications in Mobile Environments 2* (FAME²), which is designed for middleware that runs on mobile, resource limited devices.
The next section describes basic concepts used for middleware, like programming and coordination models. To provide examples of realisations for different concepts, existing solutions are presented. Section 2.2 presents FAME², and describes several characteristics in detail. Some characteristics like service discovery (refer to Chapter 3) and service update (refer to Chapter 4) are described in detail in their own chapters. In the last section of this chapter an evaluation and comparison of FAME² with other existing solutions is given.

2.1 Survey of concepts and models for middleware, and existing solutions

This section gives an overview of existing programming models, architectures, frameworks, and middleware that are relevant in the area of ubiquitous computing. Some of the following concepts are discussed for mobile computing in [53]. Their recommendations are partially useful for ubiquitous computing, too.

2.1.1 Programming models

A programming model is an abstract, conceptual view of the structure and operation of a computer system. Programming models are high-level concepts that describe operation of the computer system in a general manner [52]. This subsection describes programming models particularly developed for distributed computing systems, and is inspired by the overview presented in [52].

2.1.1.1 Client-server

The *client-server* model is an abstraction for communication between computers. Software providing services are the servers, and software consuming these services are the clients. The software does not change its role, i.e. a server remains to be a server and a client remains to be a client. The client-server model is a well-studied programming model for distributed computing and has its advantage in the clear definition of roles and the predictability of roles over time [52, 54-56]. At any time it is clear who is a server and who is a client. The assumption behind the client-server model is that computers running software in the server role have more resources, e.g. processing power, memory, storage, network bandwidth, etc., than computers running client software. Thus clients are consumers of the services offered by the servers [54-55, 57]. Traditionally, clients were responsible for receiving input from users, pre-processing the input, e.g. perform validity checks, submitting the input to the server, and then displaying their responses to users [57].
Communication is always initiated by the client [52]. This is a disadvantage of the client-server model for use in middleware for distributed computing systems in ubiquitous computing where services, provided by servers, may have an interest to notify clients, which means that servers have to initiate communication with the client. Reasons for such notifications could be the announcement of an expected disconnection of the server from the distributed computing system. Nevertheless, several products whose implementation is based on the client-server model do permit such notification, e.g. Microsoft's *Object Linking and Embedding* (OLE) or the *X Window System*[1] that is often used as a graphical user interface for UNIX operating systems [57].

[1] The X Window System is based on the client-server model with an unusual property. The server is always at the *user's side*, while the client is remote. Thus, the server starts interaction with the client when passing input from the user. This definition in the X Window System originates from the idea that the computer at the *user's side* provides the service of input and output to the client.

In the vision of ubiquitous computing, every device can be active, i.e. initiate communication, the client-server model and the solutions based on this model are only partially suitable, because the roles of client and server are static and cannot change.

2.1.1.2 Peer-to-Peer

The *peer-to-peer* model (P2P) combines the roles of client and server from the above model, so software can be both client and server [58]. Software having both roles is also called *servent*, a portmanteau of the terms *client* and *server*, and introduced by the Gnutella project, the first truly distributed widespread P2P network [59, 60].
In the P2P model, the software on peers provides and consumes services. The P2P model is attractive for ubiquitous computing because software running on any peer can initiate an interaction with software executed on other computers [61, 62].
There are several variations of P2P. At one extreme, every peer must be powerful enough, i.e. having enough processing power, memory, etc., to execute software that provides services. This can be compared to a distributed computing system where all computers are servers. At another extreme, P2P can be something very close to the client-server model where some peers execute the software that provide services, while other peers execute software that consume these services [58]. Furthermore, the P2P model enables different organisational forms, i.e. centralised or distributed. In contrast to a decentralised organisation, central organisation enables full control over a P2P network with the disadvantage of central point of failure [58, 60].
The imprecise definition of the P2P model let it seem to be the perfect candidate for building middleware for distributing computing systems in ubiquitous computing. This is because the P2P model enables software to be servents as well as clients or servers. Furthermore, software can be organised centralised or distributed. P2P model let software change its roles between client and server and servent. Furthermore, the interaction of software is organised in a centralised or distributed manner.

2.1.1.3 Synchronous interaction

While the client-server and the P2P model define the roles of software in a distributed computing system, the s*ynchronous interaction* model defines a communication style between software elements.
In the synchronous interaction model, software sends a request to another software and waits for the response being sent back, blocking the server until a request is received, and blocking the client until a response has returned [52, 57, 63, and 64]. Typical implementations of the synchronous interaction model are the *remote procedure call* (RPC) [65] and the *remote object invocation* (ROI) or *remote method invocation* (RMI) [52, 57, and 66].
The synchronous interaction model requires that client and server have a permanent connection during the interaction, i.e. the request and response are sent via the same connection. Distributed computing systems in ubiquitous computing are characterised by using wireless connectivity, e.g. WLAN or Bluetooth. Wireless connectivity is suspicious to disconnections. As a consequence, using synchronous interaction requires effort to guarantee that a connection between client and server is not lost because of disconnection. This makes the synchronous interaction model less attractive for use in middleware for distributed computing systems in ubiquitous computing.

2.1.1.4 Asynchronous interaction

The *asynchronous interaction* model is the counterpart to the synchronous interaction model. In the asynchronous interaction model, the request from a client and the response of a service are decoupled. The request and the response do not need to use the same connection, and the requests and responses do not block clients and servers respectively [52, 57, 64, and 67]. As a consequence, asynchronous interaction is, to some extent, immune to disconnections [64, 67]. Typical uses of asynchronous interaction are in event and message oriented middleware [57].

The decoupling of request and response in the asynchronous interaction model makes it suitable for being used in middleware for distributed computing systems in ubiquitous computing. However, middleware might decide to facilitate other interaction models because sometimes it is desirable that clients or servers are blocked until a response or request arrives.

2.1.1.5 Shared memory

The *shared memory* model realises interaction between software by sharing their data in one virtual space. This virtual space provides the same information to all software, enabling the software to be executed on different computers to exchange information and to share data processing [68-70].
A prominent realisation of shared memory is the tuple space [71, 72]. Implementations of tuple spaces, e.g. Linda [73], *Linda in Mobile Environments* (LIME) [74], JavaSpaces [75, 76] and PerDiS [77] are insensitive to disconnections, allow for reconciliation, optimistic locking and concurrent data access, etc., but usually lack a notification mechanism that allows the asynchronous notification of software. Existing implementations support that software reads data from the shared memory that is not present, resulting in blocking the reading software until another software writes that data. But shared memory implementations do not support that a software writes data and instructs other software to read a specific data.
The shared memory model is an attractive concept to share data among software in distributed computing systems in ubiquitous computing because it uncouples the interacting software, allowing for reconfiguration and disconnections. However, because of the lack of notification of software it is not sufficient to be used as the only programming model for interaction.

2.1.1.6 Logical Mobility

Wireless connectivity and mobile devices have introduced the potential of *physical mobility*. Software running on different computers may appear and disappear, as the result of the movement of computers, leading to new use cases and new possibilities for interaction with and of software.
Mobile code highlights the capabilities of *logical mobility*. The idea of mobile code has been introduced by John von Neumann in his seminal work on automata and the universal constructor [78] and is a feature of the Java programming language [79]. Logical mobility allows software to move from computer to computer in distributed computing systems. One application is to reduce the effects caused by limited resources by moving software to more powerful computers when they become available. Another application is to reduce the effects caused by disconnection by moving software from remote computers to local ones before a disconnection happens.
Mobility is about movement of mobile entities (logical: software; physical: devices, i.e. computers) between locations [80-83]. In logical mobility the location is usually the computer running the software. The location in physical mobility is usually a point in a physical space (two or three-dimensional coordinates, room in a building, etc.).
While the advantage of mobility is the flexibility to adapt a distributed computing system and to optimally use resources, i.e. the resources of computers in a distributed computing system, the downside of mobility of software is the impact on security when allowing malicious software to *enter* (means: to move into) a distributed computing system that is designed to contain only trusted software.

2.1.1.7 Recommendation

As a result of the overview of programming models above, the following models should, or should not, be used for middleware for distributed computing systems in ubiquitous computing:

- Do not use the client-server model because it requires that software has fixed roles and that servers do not initiate communication with clients. This excludes client software in a distributed computing system in ubiquitous computing to provide services, and servers cannot initiate communication to notify clients about events.

- Use the peer-to-peer model because it allows software to define their role and, furthermore, allows software to change their roles. Additionally, in the P2P model, any software can initiate communication.
- Use the synchronous interaction model only when it can be guaranteed that the connection between two communicating software remains until both, request and response, are exchanged between the software.
- Use the asynchronous interaction model where it cannot be guaranteed that the connection between two software remains until their communication is finished.
- Use the shared memory model to exchange data without needing notifications that new data is available for processing.
- Support for mobility, logical and physical, for optimal resource use in a distributed computing system in ubiquitous computing, but take precautions for security threats, like introducing malicious code into a usually trustworthy distributed computing system.

2.1.2 Architectures

A definition of software architecture and its purpose is given by Kruchten in [84]:

> *Software architecture deals with the design and implementation of the high-level structure of the software. It is the result of assembling a certain number of architectural elements in some well-chosen forms to satisfy the major functionality and performance requirements of the system, as well as some other, non-functional requirements such as reliability, scalability, portability, and availability. Perry and Wolfe put it very nicely in this formula2, modified by Boehm:*
> *Software architecture = {Elements, Forms, Rationale/Constraints}*
> *Software architecture deals with abstraction, with decomposition and composition, with style and esthetics. – cited from [84]*

In this subsection, three basic concepts for software architecture are described: monolithic, layering, and service oriented architecture. A summary for the suitability of monolithic and layering architectures in ubiquitous computing can be found in [52]. Service oriented architecture is discussed in more detail in [85].

2.1.2.1 Monolithic

Middleware based on the concept of *monolithic architecture* is designed and implemented as *one block*. That is, this middleware cannot be divided into smaller parts and is not adaptable. The functionality of the middleware is fixed. The software implementation of the middleware is not replaceable in parts, while there are exceptions, e.g. the ISIS architecture described in [86, 87], or the Phoenix architecture described in [88].
Monolithic architecture allows for high optimisation because all its elements, forms, rationales and concepts are known in advance. Communication, i.e. data exchange and control flows, between the different parts of the middleware can be optimised to achieve the best possible result in terms of runtime, response time, memory consumption, etc. [52].
Ubiquitous computing is characterised by heterogeneity of platforms and limited resources of some platforms. As a result, monolithic middleware has to carry out a balancing act between supporting platform features and being as small as possible to fit onto resource constrained platforms, like mobile phones.
As the convenience of middleware increases with the number of features it provides, the concept of monolithic architecture is considered to be unsuitable for ubiquitous computing [52].

2.1.2.2 Layering

Layering structures the elements of the architecture into layers with well-defined functionality [66]. Layering supports separation of concern where each layer has to concern only about the functionality it is defined for. Modularity is supported by dividing the overall functionality of a middleware into smaller parts. Allowing adding and removing layers makes the resulting

middleware reconfigurable, which is a required feature to tackle the heterogeneity of platforms used in distributed computing systems in ubiquitous computing [52, 89].
In the concept of a layered architecture every layer defines an interface to the layer above, and an interface to the layer below. The interface to the layer above is used to provide the functionality to the higher layer. The interface to the layer below is used to request responses, e.g. acknowledgements, from the lower layer, for example for reconfiguration and optimisation of the control flow within the layer. Interaction between the layers always happens between two adjacent layers. The concept of layered architecture does not allow that a layer above another layer interacts with the layer below that other layer directly [89, 90]. Layers may be empty, i.e. do not implement any functionality. Yet, conceptually those empty layers do still exist and may not be pruned.
The Open Systems Interconnection ISO 7498 standard, also known as the OSI-model, is a well known layered architecture to describe communication systems [90]. Existing middleware for ubiquitous computing based on layered architecture are e.g. *The Location Stack* presented in [91] and the *Aura project* described in [92, 93].
As the concept of layered architecture allows for reconfigurability, and is already used for middleware for ubiquitous computing, this architecture concept is suitable for ubiquitous computing.

2.1.2.3 Service oriented architecture

In the concept of *service oriented architecture* (SOA), middleware functionality is realised as loosely coupled services. Services are independent, self-sufficient software elements, which reside somewhere in a distributed computing system, and that are discoverable [94-96]. Middleware based on the concept of SOA realise their functionality by a collection of loosely coupled services. Loosely coupling means that the collection of services that forms a middleware is changeable, i.e. the middleware can be reconfigured. SOA can be understood as an evolution of layered architecture where a service represents a layer, and where any service is able to interact with any other service directly.
Figure 2-1 illustrates SOA concept. Services, providing functionality, register at a broker. The broker makes services discoverable. Clients, consuming functionality, query the broker for

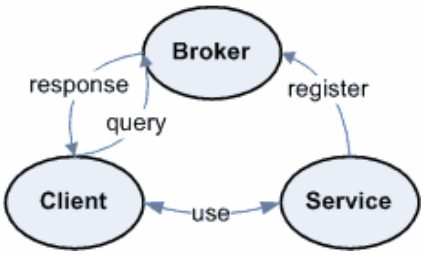

Figure 2-1: The service oriented architecture [36]

corresponding services. Clients interact with services directly, without involving the broker. The European project SeCSE created a conceptual view on service oriented architecture that illustrates the different stakeholders, elements and their relations. Those stakeholders are e.g. service consumers, service providers, service developers, etc. The different elements that may be part of software based on the concept of SOA are services and brokers [97]. This conceptual view is also used by the European Commission as a framework to classify European projects that use the concept of SOA [98]. The conceptual view structures the elements of systems based on the concept of service oriented architecture. This provides a better understanding of the characteristics, responsibilities, and relations of the elements.
Papazoglou extended the concept of service oriented architecture to integrate aspects of service composition, coordination between services, and service management [85]. This *extended service oriented architecture* (ESOA) distinguishes between *basic services*, which are identical to services in SOA, *composite services*, which are compositions of basic services, and *managed services*, which are composite services managed by a service operator / provider. Figure 2-2 illustrates the

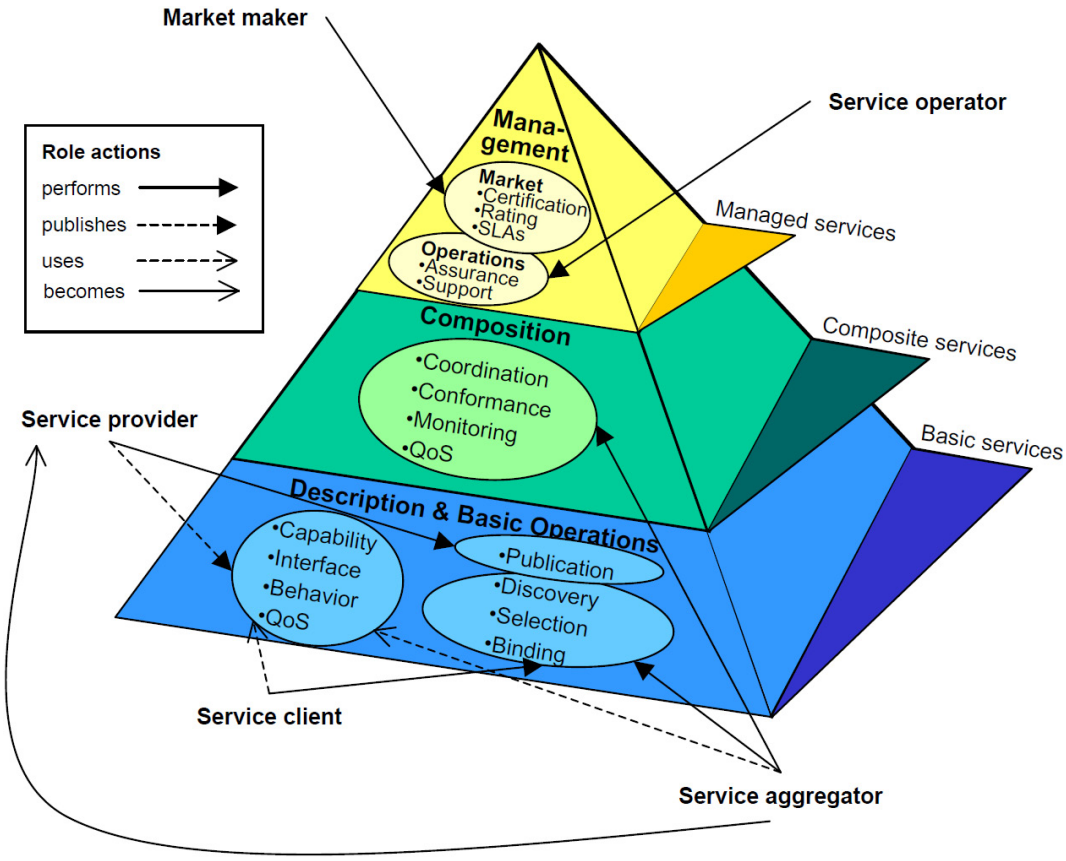

Figure 2-2: The extended service oriented architecture [85]

ESOA, its different stakeholders and the operations of the different service types, i.e. basic, composite and managed services.

SOA requires late binding [99]. Late binding supports to change the control flow of software at runtime [100], i.e. binding services to their clients at runtime. Examples for realisation of late binding are function pointers in the C and C++ programming language [101], dynamic class loading in Java [102, 103] and far jumps in assembler. The alternative to late binding is early binding. In early binding the compiler and linker binds together all resources to one piece of software before the software is executed. With early binding the control flow of software is fixed before its execution.

SOA is an attractive concept for designing middleware for distributed computing systems in ubiquitous computing, because the services are loosely coupled, allowing for reconfiguration of middleware, and the direct interaction between services, which promises a better use of the limited resources of the devices used in distributed computing systems in ubiquitous computing. Loosely coupled services are helpful for disconnected operation, because clients are able to rebind from disappeared services to other, available and compatible services.

2.1.2.4 Recommendation

The concept of monolithic architecture is not suitable for ubiquitous computing, because there is no support for reconfiguration. The concept of layered architecture facilitates separation of concern, reconfiguration, and modularity, which makes this concept suitable for ubiquitous computing. Service oriented architecture evolves the concept of layered architecture to allow direct interaction between any services.

As a recommendation, new middleware for distributed computing systems in ubiquitous computing should choose the concept of service oriented architecture, because this concept provides flexibility, reconfiguration, mobility, and disconnected operation. ESOA adds management and composition of services to SOA.

2.1.3 Coordination models

> *A coordination model is the glue that binds separate activities into an ensemble – cited from [104]*

In [105], the authors describe the basic ideas about coordination models, which is to model interaction between software elements. An evaluation of coordination models for mobile computing is given in [106].
This subsection evaluates existing coordination models for interaction of services in middleware for distributed computing systems in ubiquitous computing.

2.1.3.1 Direct communication model

In the *direct communication model*, services start a communication by explicitly contacting partners, like other services. This requires that the involved services agree on a communication protocol [70]. Direct coordination usually implies synchronisation of the communication, i.e. the use of the synchronised interaction model. The asynchronous interaction model can be simulated by decoupling request and response into two independent processes, for example by using the peer-to-peer concept [66, 107].
One disadvantage of the direct communication model is that services can hardly sustain the openness and dynamics of reconfiguration, i.e. appearing and disappearing communication partners [108]. There is no decoupling of communication partners. If a communication endpoint (i.e. a communication partner) disappears, the other endpoints (i.e. other partners) have to re-establish the communication by discovering a new endpoint and agreeing on a communication protocol.
Typical implementations of the direct communication model are RPC [109] and RMI [57, 110].

2.1.3.2 Shared data-space model

The *shared data-space model* is a concept of assisted coordination [111]. It uses centralised or distributed virtual spaces to share their data. A virtual space provides the same information to all services [108]. The shared data-space model decouples communication partners, allowing reconfiguration and disconnection of services [66].
Examples for shared data-space model, for instance tuple spaces, are described in the Paragraph 2.1.1.5.

2.1.3.3 Message oriented model

The *message oriented model* is a concept of assisted coordination [111]. Messages are sent from clients to recipients to request their services. Emmerich [43] writes that "Message-oriented middleware (MOM) supports the communication between distributed system components by facilitating message exchange."

> *MOM middleware is an event-driven, asynchronous, nonblocking and message-based communication method that guarantees message delivery. In its essence, a MOM allows separate, uncoupled applications to reliably communicate asynchronously. The MOM architecture generally replaces the client/server model with a peer-to-peer relationship between individual components, where each peer can send and receive messages to and from other peers. MOM systems provide a message queue between interoperating programs, so if the destination process is busy, the message is held in a temporary storage location until it can be processed. – cited from [112]*

As it is described in [112, 113], the message oriented model deals with disconnection and mobility. Disconnection and mobility are supported by the decoupling of the communication between the communication partners.
One advantage of the message oriented model is the asynchronous message delivery, allowing the sender to continue its work as soon as the message is sent, regardless whether the message is received and processed by the communication partner. Another advantage is the support for group communication, enabling to send a message to multiple recipients. A disadvantage of the message

oriented model is that synchronous behaviour, where clients wait for the service being delivered, requires additional effort.
Jung et al provide a comparison of the message oriented middleware, which is based on the message oriented model, with RPC (see Table 2-1).

Table 2-1: Comparison of message oriented middleware and RPC [14]

Feature	MOM: messaging and queueing	Remote Procedure Call (RPC)
Metaphor	Post office-like	Telephone-like
Client/Server time relationship	Asynchronous	Synchronous
Style	Queued	Call-return
Load-balancing	Single queue can be used to implement FIFO or priority-based policy	Requires a separate TP monitor
Transaction support	Yes	No
Asynchronous processing	Yes. Queues and triggers are required	Limited. Requires threads and tricky code for managing threads

Evaluations of middleware based on the message oriented model are in [115-117]. In [115, 116], a general overview of the functionality of existing middleware is given. The suitability of existing middleware and the message oriented model for Internet applications is studied in [117].
In [115], a *composite message oriented middleware* where the relation between messages can be expressed is introduced. The relations are expressed in messages that contain information about the related message (for example a unique message identifier).

2.1.3.4 Event oriented model

The *event oriented model* is very similar to the message oriented model. It is an assisted coordination model [111], and provides space, time and synchronisation decoupling of the communication partners [20]. In contrast to the message oriented model where messages are sent to a recipient, events are published and received by subscribed communication partners [20, 110]. That is, the sender of the event does not know who receives the event. In the message oriented model, the sender of the message knows who receives the message. The decoupling in the event oriented model is often achieved by an event broker that receives events from publishers, and forwards them to subscribers.
Eugster et al [20] give a detailed discussion on the event oriented model, there called publish/subscribe, and compare it with the direct communication model, the shared-data-space model and the message oriented model (see Table 2-2). They write that the event oriented model provides space, time and synchronisation decoupling, and group the event oriented model into three different categories: *type*, *topic* and *content*. Space decoupling means that interacting software does not need to be located in the same physical space, e.g. executed on one computer. Time decoupling means that interacting software does not need to be active at the same time, but instead the interactions can be buffered somewhere. Synchronisation decoupling means that software may interact independently from their state, e.g. without requiring to receive a response or even acknowledgement for a previous interaction.

Table 2-2: Decoupling abilities of different interaction models and implementations [20]

Abstraction	Space decoupling	Time decoupling	Synchronisation decoupling
Message passing	No	No	Producer-side
RPC/RMI	No	No	Producer-side
Asynchronous RPC/RMI	No	No	Yes
Future RPC/RMI	No	No	Yes
Notifications (observer pattern)	No	No	Yes
Tuple spaces	Yes	Yes	Producer-side
Message queuing (Pull)	Yes	Yes	Producer-side
Publish/Subscribe	Yes	Yes	Yes

Examples where the event oriented model between software is used are Infobus [118], DREAM [119], DERMI [120], JEDI [121], Hermes [122, 123] and STREAM [124]. They use different organisation forms for managing, storing and delivering events. Some of them, e.g. DERMI and JEDI are optimised for use in mobile environments by focusing on time and space decoupling. Others, e.g. Hermes and STREAM focus more on subscription aspects, i.e. the addressing and delivery of events based on e.g. context information like temperature etc. A demonstration of how the event oriented model can be used for reconfiguration in middleware for distributed computing systems by changing subscriptions and redirect events is given in [125].

2.1.3.5 Recommendation

The direct interaction model requires that communication partners can establish a direct connection and, in the case of synchronous interaction, can maintain that connection. This makes it less suitable for distributed computing systems in ubiquitous computing with their characteristic of disconnections. The shared-data-space model is a suitable interaction model when no notification of communication partners is required. The message oriented model allows senders of information, i.e. messages, to notify the recipients of the message. If notification is required but the recipient is not known by the sender then the event oriented model is used. All three models, shared-data-space, message oriented, and event oriented, are robust against disconnections and allow for reconfiguration and mobility of the communication partners. As a result, middleware shall be able to use all three interaction models.

2.1.4 Different forms of transparency

The following text is cited from Raymond, who gives an introduction into the *reference model of open distributed processing* (RM-ODP) and summarises the different forms of *standard* transparency:

> *Computational specifications are intended to be distribution-transparent, i.e., written without regard to the very real difficulties of implementation within a physically distributed, heterogeneous, multi-organisational environment. The aim of transparencies is to shift the complexities of distributed systems from the applications developers to the supporting infrastructure.*
> *RM-ODP defines a number of commonly required distribution transparencies and describes the computational refinements and use of engineering functions needed to provide these transparencies. The distribution transparencies defined in RM-ODP are:*
> *access transparency — hides the differences in data representation and procedure calling mechanism to enable interworking between heterogeneous computer systems*
> *location transparency — masks the use of physical addresses, including the distinction between local versus remote*
> *relocation transparency — hides the relocation of an object and its interfaces from other objects and interfaces bound to it*
> *migration transparency — masks the relocation of an object from that object and the objects with which it interacts*
> *persistence transparency — masks the deactivation and reactivation of an object*
> *failure transparency — masks the failure and possible recovery of objects, to enhance fault tolerance*
> *replication transparency — maintains consistency of a group of replica objects with a common interface*
> *transaction transparency — hides the coordination required to satisfy the transactional properties of operations*
> *– cited from [126]*

Transparency simplifies the development of software for distributed computing systems. Developers are released from tasks to deal with the different problems, e.g. appearing and disappearing services, logical mobility, etc. and instead use the functionality implementing the

different forms of transparency of the underlying platform, or middleware. As a consequence, it is desirable that the underlying platform and middleware realise as many forms of transparency as possible.

2.1.5 Existing frameworks

Frameworks for building middleware provide specific aids to support their implementation, and to facilitate the integration, deployment and execution of their services. Such frameworks are often based on the notion of a *component model*. In general, component models include the following features:

- a set of component types: they identify specific roles of middleware services that are considered important and of common usage in middleware
- the specification of a component container: it defines the environment in which a component lives. It acts as an intermediary as far as all communications between the component and the external environment, i.e. the platform, are concerned. Also, it can manage components life cycle, persistency, transactions, multithreading, load balancing, etc.
- the specification of a component descriptor: it describes the characteristics of components

Frameworks simplify and fasten the development of middleware. This subsection reviews existing frameworks for their suitability for being used to develop middleware that is then executed on mobile devices. Beside frameworks designed for ubiquitous computing, like PCOM from the University of Stuttgart [127] and the *Reconfigurable Ubiquitous Networked Embedded Systems* (RUNES) project [128, 129], existing frameworks with a related focus or with a widespread use are reviewed, e.g. the *Open Services Gateway initiative* (OSGi) [130, 131], the *Reflective Middleware for Mobile Computing* (ReMMoC) [15], the *(Enterprise) JavaBeans* [132, 133] concept and the *Common Object Request Broker Architecture Component Model* (CCM) [55]. Additionally, due to the relevance of the current situation, Web Services technology is included in this review.

2.1.5.1 Common Object Request Broker Architecture Component Model

The *Common Object Request Broker Architecture* (CORBA) *Component Model* (CCM) is a general purpose component model for distributed computing systems, based on the CORBA middleware

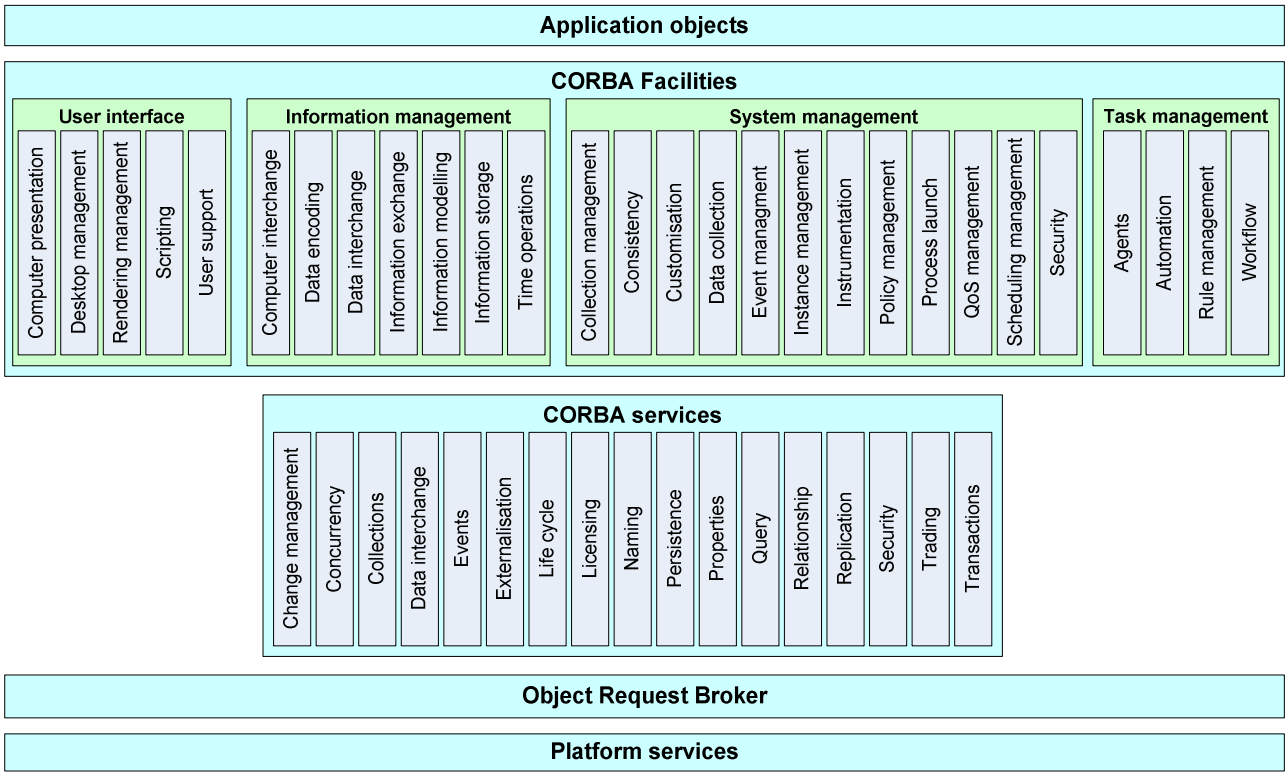

Figure 2-3: CORBA architecture reference model

[55, 134]. CORBA provides platform independent communication between software in distributed computing systems [134]. This is achieved by a standardised description of the application programming interfaces (API) of the software using the *Interface Description Language* (IDL) and the *General Inter-Operation Protocol* (GIOP) transport protocol. In CORBA, software registers at the *Object Request Broker* (ORB), so it can be found. Object Services provide basic support functions for the implementation of software, like life cycle. Support features that are not considered to be basic, like email server facilities, are Common Facilities. Application Objects use CORBA. Figure 2-3 depicts the CORBA architecture reference model with the features provided by the different elements. The number of features is continuously growing as new technologies are realised or transferred to CORBA, e.g. support for wireless communication protocols, disconnected operation, and context awareness [135, 136].
The CORBA Component Model advances the concept of software in CORBA to the concept of components [55]. For this purpose, CCM introduces the terms Basic Component and Extended

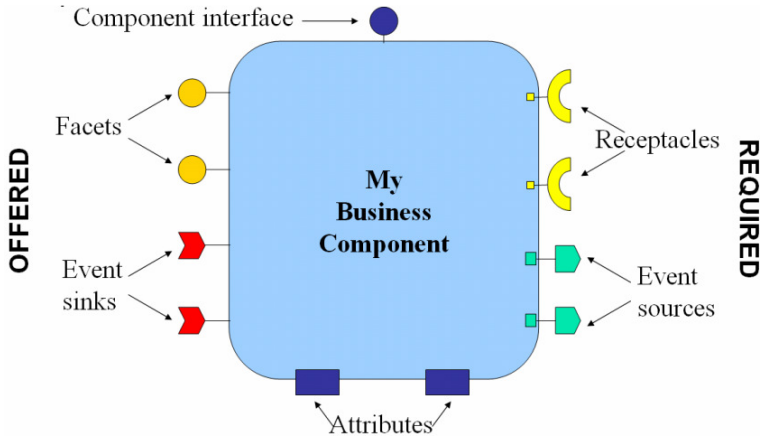

Figure 2-4: CCM architecture [137]

Component. Basic components are made of attributes and the equivalent interface. Attributes are used to write and read the component configuration. The equivalent interface represents the functionality of the component, i.e. its API. In addition to basic components, extended components are made of facets, receptacles and event source and sink. A facet is a single aspect of the component's API, i.e. a particular functionality implemented in the component. The sum of all facets results in the equivalent interface. Receptacles allow other components to "plug into" the component to be notified on events. The event source publishes defined events, whereas the event sink consumes them. The CCM architecture is depicted in Figure 2-4.
Middleware created from services implemented using the CCM can be based on the peer-to-peer model, may use the synchronous (facets and receptacles) and the asynchronous (event sinks and source) interaction model, and allow for logical mobility when the service implementation is written in the programming language of the target platform, i.e. executable by the platform the service moves to. CCM allows for building middleware based on service oriented architecture. With different extensions and additions of the CORBA middleware, the direct interaction model, the message oriented model, and the event oriented model are useable for interaction. Additionally, implementations of the shared-data-space model are possible. CORBA offers access and location transparency. Depending on the integrated CORBA services and CORBA facilities, relocation, migration, persistence, failure, replication, and transaction transparency is possible. CCM inherits these characteristics and features from CORBA. The disadvantage of CCM is its enormous number of features that are tightly integrated into the CORBA middleware and the CCM itself, resulting in typical implementations of several hundred megabytes in size (like StarCCM [138]), and enormous hardware requirements (CPU power, memory, etc.), which makes CCM unsuitable for deployment on resource limited devices used in ubiquitous computing, like smart phones.

2.1.5.2 (Enterprise) JavaBeans

JavaBeans is a component model designed for the Java programming language. The Java programming language features a virtual machine where software written in Java is executed in. This virtual machine abstracts from platform specifics, allowing running Java software on any platform where a compatible virtual machine is available. The result is that JavaBeans are portable, i.e. support logical mobility, relocation and migration transparency [132]. JavaBeans are executed in a container that is written in the Java programming language and that realises persistency, the event oriented model, a connection to data base systems and to the CORBA middleware. Properties are used for customisation of JavaBeans for reconfiguration. Introspection, realised by reflection technology [139], enables tools to investigate the behaviour of JavaBeans for plug-and-play development [140].

Reflection is the process by which software can observe and modify itself. A formal definition of reflection is given in [46, 141]:

> *In as much as a computational process can be constructed to reason about an external world in virtue of comprising an ingredient process (interpreter) formally manipulating representations of that world, so too a computational process could be made to reason about itself in virtue of comprising an ingredient process (interpreter) formally manipulating representations of its own operations and structures. – cited from [46]*

Enterprise JavaBeans are a more feature-rich variant of JavaBeans, designed for the use in enterprise computing. The architecture of *Enterprise JavaBeans* (EJBs) (or Enterprise Beans) is

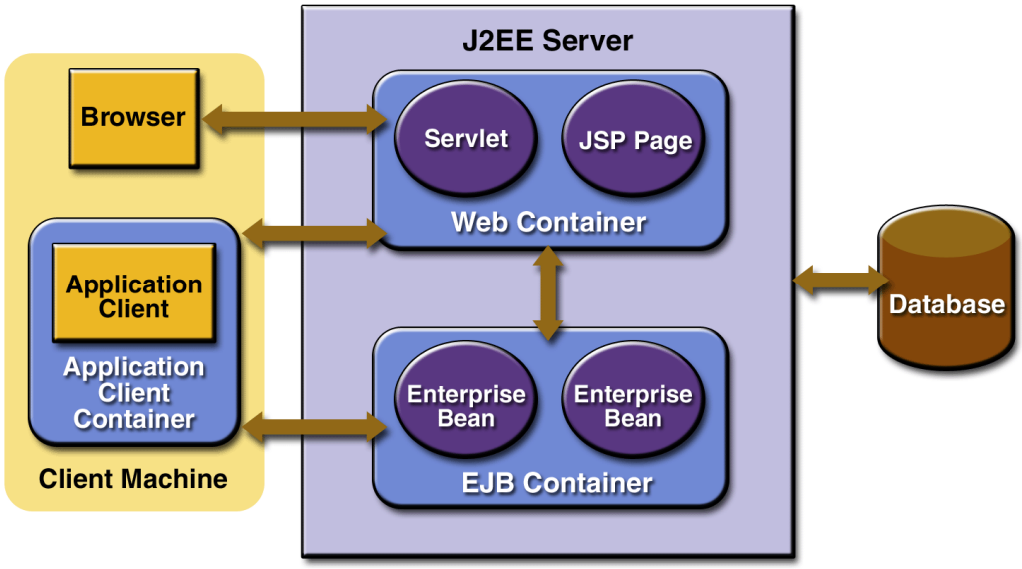

Figure 2-5: Architecture of Enterprise JavaBeans [133]

similar to CCM basic components [133]. EJBs are embedded into a container that executes the EJB and provides functionality that is of general interest in enterprise software, e.g. transaction management, remote communication protocols, security, and logging. Figure 2-5 depicts the architecture of EJBs [133].

There are three different types of EJB: session bean, entity bean, and message-driven bean [133]. Session beans are used when a state is required, i.e. the operation sets the value of variables that are used in a second operation. Usually, such beans are used by exactly one client to ensure that the client can predict the state of the bean. By definition, session beans are not persistent, i.e. their state is not saved when it is stopped. Entity beans are persistent. They are defined to be one row in a relational data base table. Entity beans are designed to interact with multiple clients concurrently, and save their state in the data base they are connected to. The third type, message-driven beans, is a stateless bean that processes events from various, unspecified sources. Message-driven beans are not meant to interact with clients directly, but with events sent by them instead. The communication

between message-driven beans and clients is asynchronous, whereas the communication with the other two bean types is synchronous.
Similar to CCM, EJB is a general purpose component model, while more elaborate for the use in enterprise distributed systems. EJB systems are not meant to change frequently. There is no entity comparable to CORBA object request broker to locate EJBs. This means that EJBs do not strictly adhere to service oriented architecture. EJB support all coordination models described in Subsection 2.1.3. Furthermore, access, location, persistence, and transaction transparency are supported. Extensions do exist for replication and failure transparency. But similar to CCM, implementations of EJB, for example the Sun Application Server, require a lot of resources (CPU, memory, disk storage) not available on mobile devices like smart phones.

2.1.5.3 Web Services

Web Services are an emerging technology. There are several solutions developed for Web Services in numerous areas. Yet, those manifold developments led to a fuzzy understanding about Web Services, as identified in [142]:

> *The hype surrounding Web Services has generated many common misconceptions about the fundamentals of this emerging technology – cited from [142]*

Simply, web services are software that is remotely accessible via standard web protocols. Web services may:

- be described [143, 145-149]
- interact via well defined interfaces [143, 146-147]
- fulfill a specific task [144, 147, 149]
- maintain a workflow or business transaction [144-145, 147]
- are self-contained [145]
- are published and discoverable [145-146, 148]
- use XML for their description and communication [146-151, 153]
- be independent of programming language and platform [147]
- combine software running on different middleware [152]

Web Services are developed in an active community that continuously creates new technologies for Web Services, e.g. transaction support, events and notification, group communication, quality of service, service level agreements, reconfiguration, disconnected operation, security and privacy, discovery (for examples see Subsection 3.1.6), and many more. They support the event oriented, message oriented and shared-data space coordination model, are based on the concept of service oriented architecture, and support several forms of transparency [154]. As a result, Web Services seem to be able to tackle virtually any characteristic and requirement of ubiquitous computing.
Web Services are designed for being executed in the Internet, as described by Vaughan-Nichols,

> *The services themselves would run on Web-based servers, not PCs, thereby moving functions from the desktop to the Internet. – cited from [155]*

Nevertheless, there are successful attempts to execute Web Services on mobile devices, for example the lightweight SOAP server described in [156].
The reason why Web Services are not the optimal solution for ubiquitous computing is the lack of logical mobility. While physical mobility, that is the mobility of devices, is supported by Web Services, there is no support that Web Services move from one device to another.

2.1.5.4 Open Services Gateway initiative

The *Open Services Gateway initiative* (OSGi) is a standardisation body driven by the industry to create a basis for service deployment in distributed computing networks [130, 157].

> *Its mission is to create open specifications for the network delivery of managed services to local networks and devices. The primary targets for the OSGi specifications are set top boxes, service gateways, cable modems, consumer electronics, PCs, industrial computers, cars and more. – cited from [158], similar statements are in [131]*

The following text is an introduction to OSGi cited from [131]:

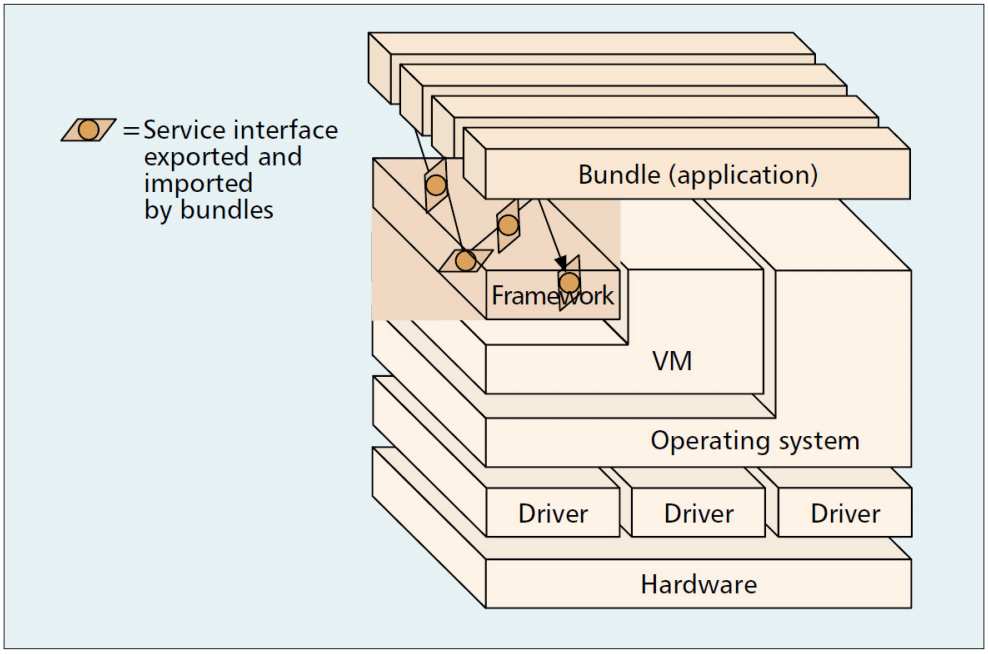

Figure 2-6: Relationship of components in OSGi [131]

The OSGi specifies the layer upon which services execute. The specifications deliver an open, common architecture that allows the deployment and management of services in a coordinated fashion. This common architecture can then be mapped onto the physical and logical components that go to make up the operational environment.
A set of principles have been established to guide the development of these specifications, and these principles go a long way toward explaining the characteristics of the standard:

- *Platform Independence: The OSGi software environment can be implemented on many platforms, with widely varying capabilities. Application Independence: OSGi provides a "horizontal" platform that is applicable in any computing environment where the capabilities of the software environment are useful.*
- *Multiple Service Support: OSGi environments are capable of hosting multiple applications from different service providers on a single service platform.*
- *Service Collaboration Support: The OSGi environment allows services to be deployed that provide functionality to other services. Applications can dynamically discover these services and adapt their behavior to the configuration of the environment and the other services that are present.*
- *Security: An OSGi environment can concurrently support many services from different service providers. Security between these services is of paramount importance.*
- *Multiple Network Technology Support: OSGi cannot mandate particular choices of network and it is network agnostic, as far as is reasonably practical.*
- *Simplicity: The OSGi environment offers a service environment where the complexity of managing the service environment can be placed into the hands of professionals in the form of the gateway operator. This does not, however, preclude individuals from configuring their own gateway as appropriate.*

– cited from [131]

The OSGi framework is based on Java, inheriting the features and capabilities of the Java programming language like mobile code and security [19]. The OSGi framework provides a general-purpose, secure, managed Java framework that supports the deployment of extensible and downloadable service applications known as bundles [131, 159]. Applications are encapsulated in bundles. A bundle is comprised of Java classes and resources of the application, like images and documents. Bundles are also services to other bundles. The OSGi service platform defines services

for logging, file serving over http, device access abstractions (i.e. low-level access to platform features), configuration and user management, protocol handling, etc. [131, 159, 160]. The relationship of components in OSGi is depicted in Figure 2-6.
The OSGi framework is executed on dedicated gateway computers that connect a wide-area network with a local-area network and provide their services (in the bundles) to devices in the local network [161, 162].
Figure 2-7 shows an example of OSGi infrastructure. It illustrates a distributed computing system to control a home environment, e.g. television set, lights, heater, etc., with the help of the OSGi framework. The home can be controlled from the outside via a mobile phone, for example via a

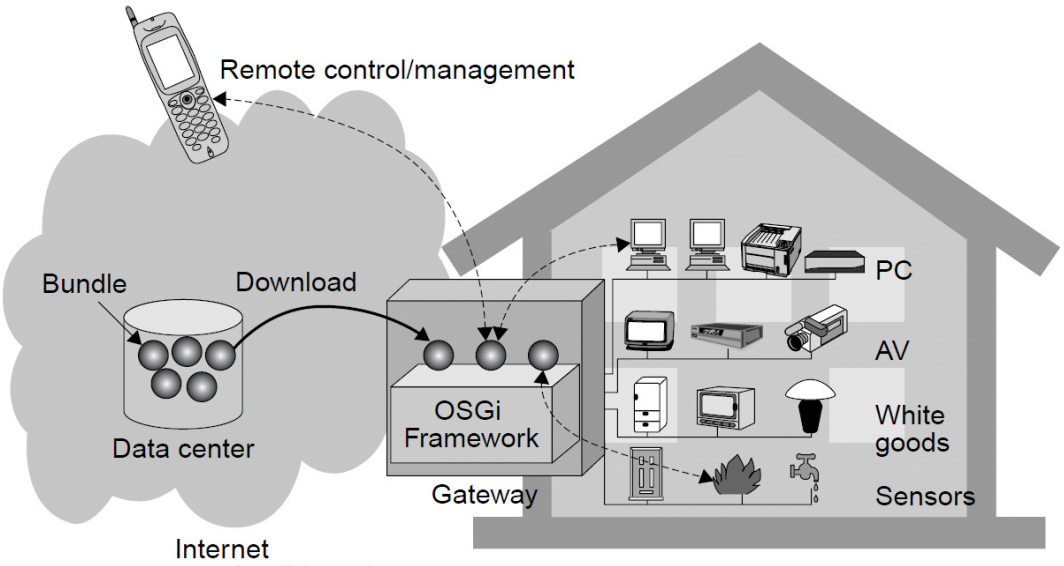

Figure 2-7: Example of OSGi infrastructure [161]

Web page interface. Implementations of such infrastructure are described in [163, 164]. The authors of [165-167] describe how software based on OSGi supports elder people in a smart house called the Gator House. Additional uses of OSGi are listed in [159].
While the OSGi provides its own implementation of the OSGi framework, there are alternative implementations with additional features. The OSCAR OSGi framework implementation, described in [168, 169], adheres to the peer-to-peer model, allowing OSGi frameworks to discover each other. The SOCAM framework [170] and the SENCHA middleware [171], extend the OSGi framework with context-awareness. Jadabs extends the OSGi framework with aspects, which allow for extension of service features without modifying the service implementation [172]. Beanome, presented in [173], extends the description of bundles to utilise advanced discovery protocols, for example JINI networking technology.
The open questions of the OSGi framework are the lack of update of bundles and the design of the framework to be run on gateways. The OSGi framework allows updating a bundle only while it is not used. A bundle that is in use cannot be updated. This is considered as essential drawback of the OSGi framework when used for ubiquitous computing. The OSGi framework is designed to be run on gateway computers that connect a wide area network, for instance the Internet, with a local area network. As a result, the capabilities and resources of some devices in a distributed computing system are exploited, while other resources are dormant. Additional questions of OSGi are discussed in [169, 171], of which some are tackled with the version 4.0 of the OSGi framework, published in August 2005.

2.1.5.5 Reflective Middleware for Mobile Computing

The motivation for the *Reflective Middleware for Mobile Computing* (ReMMoC) is the use of different middleware in distributed computing systems in mobile and ubiquitous computing. ReMMoC is a middleware framework that dynamically adapts to underlying, present middleware. ReMMoC masks the availability of middleware to applications, so applications do not need to care

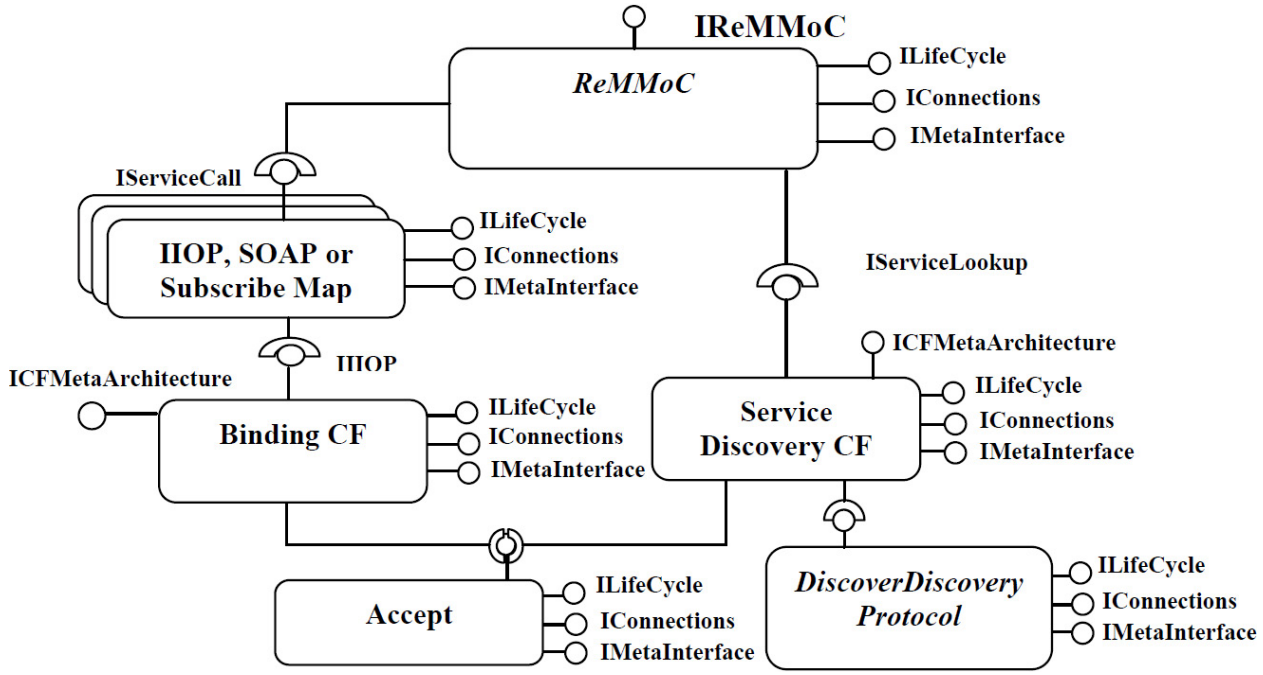

Figure 2-8: Architecture of ReMMoC [175, 176]

which middleware they use for e.g. communication. With ReMMoC, applications may communicate using CORBA in one location, and in another location they use SOAP [15, 174]. ReMMoC uses reflection to adapt to the underlying middleware. Reflection is described in [46, 141]. Applications access ReMMoC using Web Services technology, i.e. SOAP (an overview of Web Services is in Paragraph 2.1.5.3) [15, 175, 176].

Figure 2-8 illustrates the architecture of ReMMoC. The two key building blocks of ReMMoC are the *Service Discovery CF* (component framework) and the *Binding CF*. The *Service Discovery CF* integrates different service discovery solutions and will be described in Subsection 3.1.9. The *Binding CF* abstracts from different middleware, e.g. CORBA or SOAP, and binds applications to available middleware.

Applications, that are clients of ReMMoC, invoke the middleware functionality forwarded by ReMMoC using SOAP messages. ReMMoC clients explore the forwarded functionality from WSDL documents that ReMMoC creates at runtime, matching the current configuration [15]. The *Binding CF* forwards communication functionality, e.g. CORBA, SOAP, tuple-spaces, etc. (refer to Figure 2-9). Applications then use the abstraction from communication protocols provided by the *Binding CF* to access available ones [45].

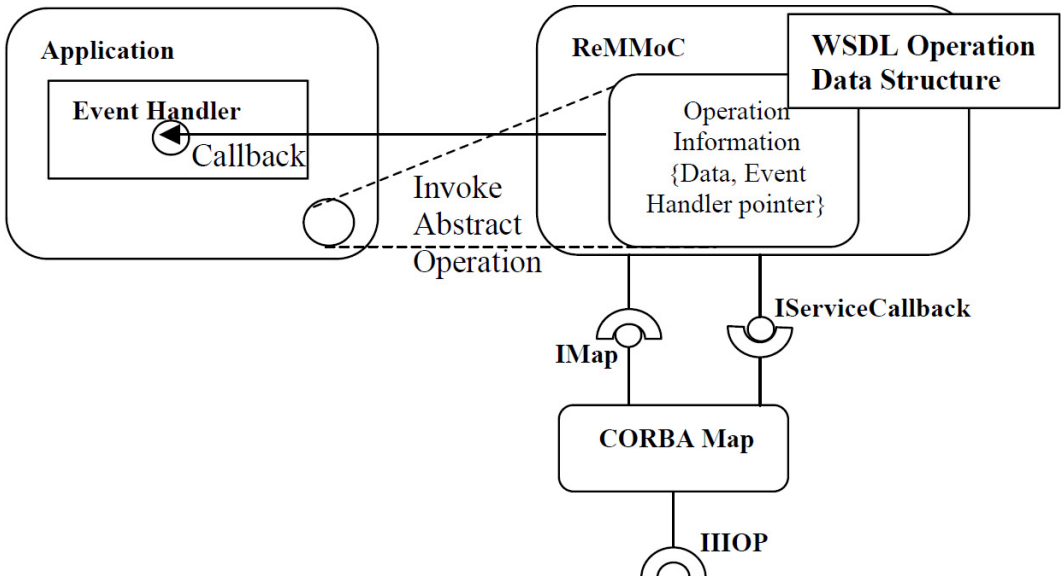

Figure 2-9: Communication in ReMMoC between client and service [15]

ReMMoC abstracts from the different middleware present in different locations, but considers middleware for communication protocols and service discovery only. ReMMoC does not provide any solution for integrating services used by applications, e.g. discovery, logging, security, and so on. Additionally, ReMMoC does not support mobility and has no support to integrate new middleware at runtime. When a new middleware is encountered that ReMMoC has no support for, the implementation and configuration has to be updated, and the whole ReMMoC system must be redeployed. This makes ReMMoC practically unusable for distributed computing systems in ubiquitous computing.

2.1.5.6 RUNES

The objective of the *Reconfigurable, Ubiquitous, Networked Embedded Systems* (RUNES) project is to create a component model for software in ubiquitous computing. In RUNES project, architecture for networked embedded systems is developed [128].

> *the RUNES middleware reaches down into layers that typically belong to the network and the operating system, therefore providing a unified approach to configuration, deployment and reconfiguration at multiple levels of abstraction. – cited from [128]*

Figure 2-10 depicts the elements of the RUNES component model. Components provide the functionality to applications. The components in RUNES interact with the runtime, applications, and other components through interfaces and receptacles exclusively. Interfaces make the services, implemented as components, accessible, and receptacles are required services of the component. The capsule is the runtime environment that hosts components. The capsule and the components are implemented for each platform, e.g. PDA running Linux, PC running Windows, and mobile phone running Symbian. The runtime API is identical across all platforms, so components have the same runtime API available. The component interface and the runtime API are described in CORBA's *Interface Description Language* (IDL), which is an established standard for technology independent interface specification [128, 177].

RUNES support adding and removing components compatible with the platform at runtime. Communication, discovery, etc. are realised as components. The middleware of RUNES is based on the peer-to-peer programming model and the concepts of service oriented architecture. It supports basically any interaction and coordination model because they are implemented in components. A limitation of RUNES is the missing support for update of components while they are in use. While the middleware of RUNES features logical mobility, a requirement is that the target computer where a component moves to has the same platform, i.e. operating system, as the source computer where the component moves from.

2.1.5.7 PCOM

Pervasive Components (PCOM) is a component model for ubiquitous (=pervasive) computing. It provides a high-level programming abstraction and captures dependencies between components. PCOM resolves dependencies between components, and detects available components in a distributed computing system [127, 178]. Figure 2-11 shows the architecture of PCOM.

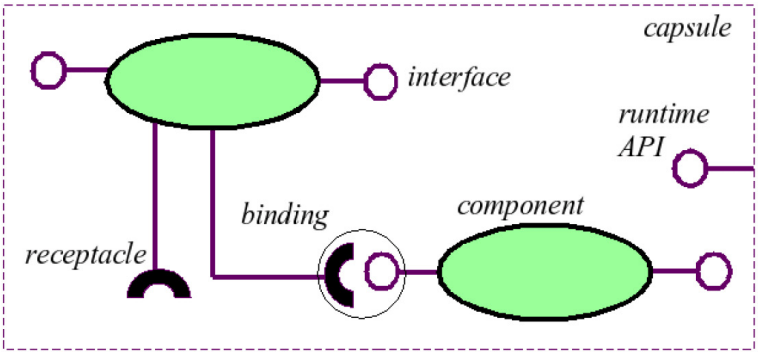

Figure 2-10: Elements of the RUNES component model [128]

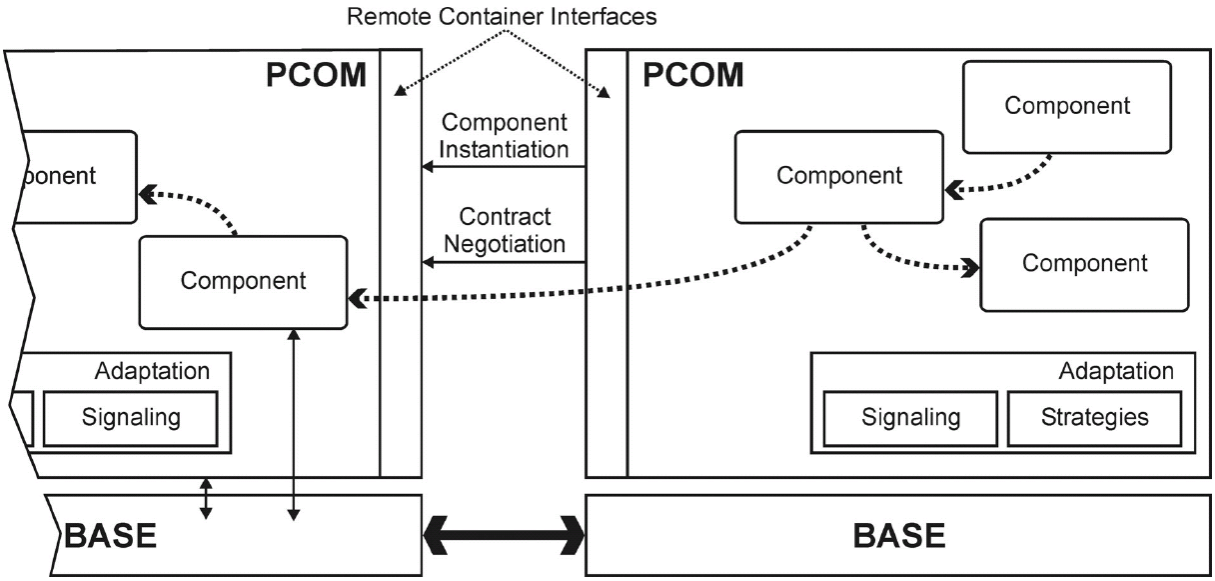

Figure 2-11: Architecture of PCOM [127]

PCOM uses the BASE middleware for detecting and interacting with components in a distributed computing system. BASE is a micro-broker based middleware for ubiquitous computing [179]. That means that BASE provides only rudimentary functionality that is required to integrate additional functionality on demand. BASE provides low-level adaptation support for interaction between components, i.e. abstraction from communication protocols. BASE enables selection of communication protocols upon availability, including the exchange of communication protocols on ongoing communication between components.
Figure 2-12 illustrates a situation where such replacement of the communication protocol is depicted. The design and implementation size of a few kilobytes allow for using BASE on resource limited devices like mobile phones as well as on resource rich devices like mainframe computers.

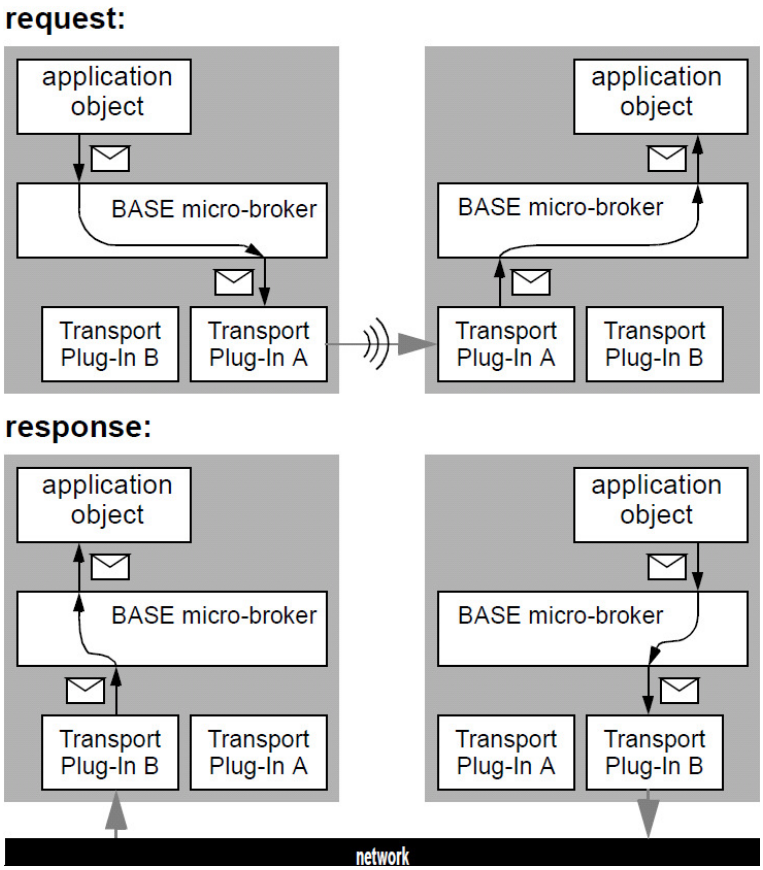

Figure 2-12: Request and response over different communication protocols in BASE [10]

The design of BASE, based on the Connected Limited Device Configuration (CLDC) Java profile for resource limited devices, does not allow middleware based on it to modify the set of components after the middleware is deployed [10].
PCOM resolves dependant components by evaluating component descriptions. Components describe the service (interfaces etc.) they offer, and formulate the interfaces etc. they depend on. Besides this, information about platform requirements (e.g. CPU type, operating system, memory consumption etc.), and implementation information used by PCOM to manage and control components, is contained in the component descriptions. Figure 2-13 shows an example of two components for an instant messenger application. In the example a *Keyboard Component* depends on a *Keyboard Component*. Implementation details used by PCOM for component life cycle and management are description in sections (d) and (g) in the component's descriptions. The platform requirements of the components are described in sections (c) and (f). Sections (a) and (e) describe the offerings of the components, i.e. the interfaces, events, etc., they provide. The dependency on other components is described in section (b). PCOM resolves the dependencies of components by comparing the requirements (section (b) in the component's description) with the offerings of other components (section (e) in the component's description).
A similar approach of automatic component dependency resolution for the Open Services Gateway initiative is presented in [180].
The limitations of PCOM result from the limitations of BASE that does not allow logical mobility and reconfiguration of components in the middleware once it is deployed. To change the set of

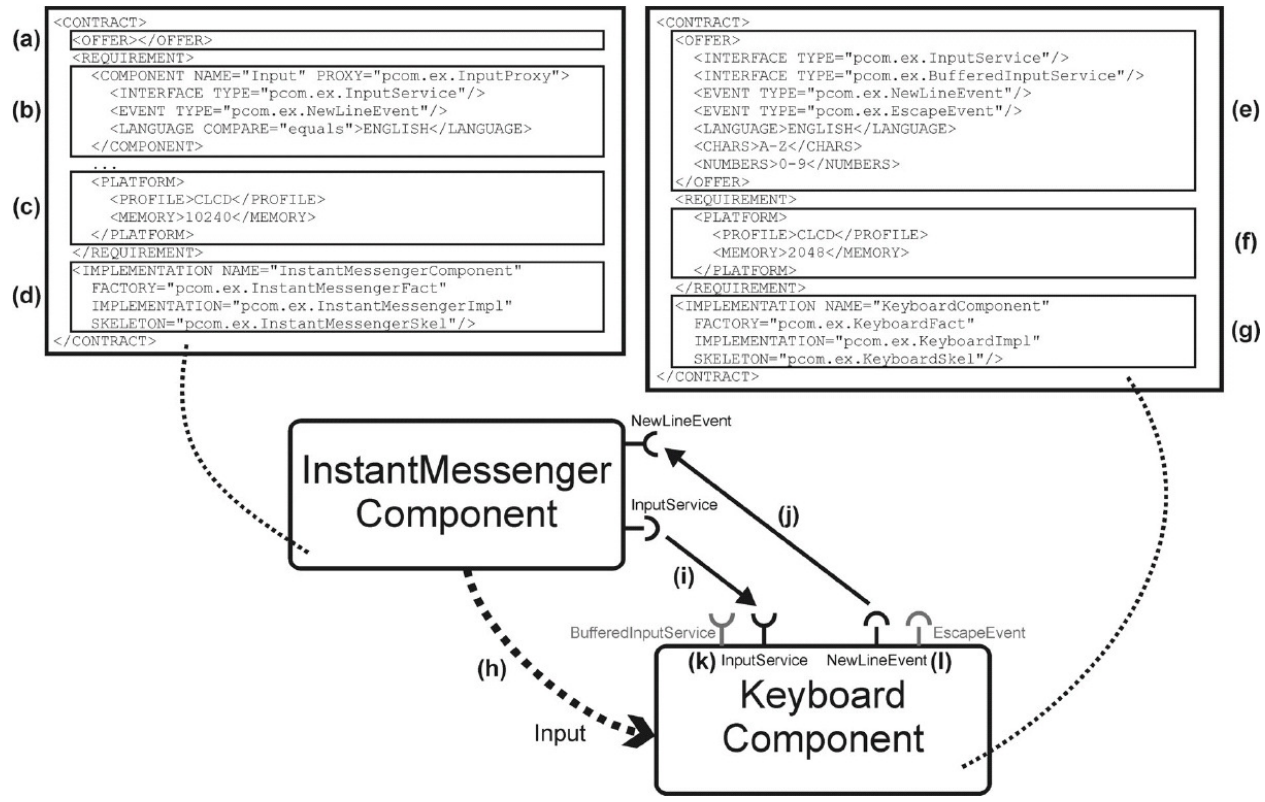

Figure 2-13: Example of two components for an instant messenger application [127]

middleware services, the middleware has to be redeployed first, which is unsuitable for ubiquitous computing.

2.1.5.8 Summary on reviewed frameworks

Table 2-3 on the next page summarises the characteristics of the reviewed frameworks.
All reviewed frameworks support reconfiguration. Web Services, ReMMoC, RUNES and PCOM are able to be executed on mobile devices. Web Services, ReMMoC and PCOM do not support logical mobility, resulting in ineffective use of resources in distributed computing systems in

ubiquitous computing. The RUNES project is still in its specification phase and does not support the update of components that are in use. Update is an important feature for middleware in non-stop operation, and in situations where it is not practicable to shutdown the distributed computing system to perform maintenance, e.g. bug-fixing components.
According to the comparison of existing frameworks and the recommendations given above, Web Services and RUNES are the most promising candidate to implement middleware for distributed computing systems in ubiquitous computing. However, several standards of Web Services are still pre-mature and it is unclear how they will be realised and how they can be used. Furthermore, Web Services lack support for logical mobility. RUNES has partial support for logical mobility, but does not support online-update. Consequently, there is the need for a new framework approach which is presented in the next sub-section.

Table 2-3: Comparison of characteristics of reviewed frameworks

	CCM	(E.) JavaBeans	Web Services	OSGi	ReMMoC	RUNES	PCOM
Reconfiguration	Yes	Yes	Yes	Yes	Yes	Yes	Yes [4]
Limited resources	No	No	Yes	Yes	Yes	Yes	Yes
Disconnected operation	Yes	Yes	Yes	No	Yes	Yes	Yes
Programming model							
Client-Server	Yes	Yes	Yes	Yes [1]	Yes	Yes	Yes
Peer-to-peer	Yes	No	Yes	Yes [1]	Yes	Yes	Yes
Synchronous interaction	Yes	Yes	Yes [1]	No	Yes	Yes	Yes
Asynchronous interaction	Yes	Yes	Yes	No	Yes	Yes	Yes
Shared memory	Yes [1]	Yes [1]	Yes [1]	No	Yes	Yes	Yes
(logical) Mobility	Partially [2]	No	No	Yes	No	Partially [2]	No
Architecture							
Monolithic	No	No	No	No	No	No	No
Layered	No	No	No	No	No	No	No
SOA	Yes	Partially [3]	Yes	Partially [3]	Yes	Yes	Yes
Coordination model							
Direct communication	Yes	Yes	Yes [1]	No	Yes	Yes	Yes
Shared data-space	Yes [1]	Yes [1]	Yes [1]	No	Yes	Yes	Yes
Message oriented	Yes	Yes	Yes	No	Yes	Yes	Yes
Event oriented	Yes	Yes [1]	Yes [1]	No	Yes	Yes	Yes
Supported transparencies	All [1]	Access, location, persistence, failure [1], replication [1], transaction	Access, location, persistence [1], replication [1], transaction [1]	Relocation, migration	Access, location, relocation, replication, transaction	All [1]	Access, location, relocation, transaction
Others	High number of tightly integrated features, with the result of enormous platform requirements (CPU power, memory, hard disk, etc.)	High number of tightly integrated features, with the result of enormous platform requirements (CPU power, memory, hard disk, etc.)	Very active research and standardisation community with the result of numerous, partially competitive proposals	Execution of software on dedicated gateway computers; no communication support (provided by other middleware, e.g. Java Remote Method Invocation)	Extending ReMMoC with new features requires redeployment	Does not support update of components that are in use.	No support for integrating new services into the middleware.

[1] : some features are supported by extensions to the standard or implemented as additional services.
[2] : If the component is written in a programming language that is supported by the computer that the component should move to, then logical mobility is supported.
[3] : Does not feature service discovery. However, extensions like the Java Remote Method Invocation Registry or the JINI network technology that provide service discovery can be used for discovery.
[4] : Reconfiguration between devices only. The middleware on a device cannot be reconfigured.

2.2 Framework for Applications in Mobile Environments 2

The purpose of the *Framework for Applications in Mobile Environments 2* (FAME²) is to support the development of middleware that is able to run on mobile devices, personal computers, and general purpose servers. The design goals of FAME² are:

- flexibility of resulting middleware, i.e. adding, removing and updating middleware services
- reusability of services so that they can be used in different middleware
- openness to allow for different programming models, coordination models, and transparencies
- minimal requirements to the platform, i.e. CPU power, memory, etc.

FAME² consists of guidelines, proposals for interfaces and functionality, and prototypes.

2.2.1 Development process

FAME² is based on the concept of service oriented architecture. Middleware, developed with FAME², is an ensemble of services. These services are hosted in a *Service Execution Environment*

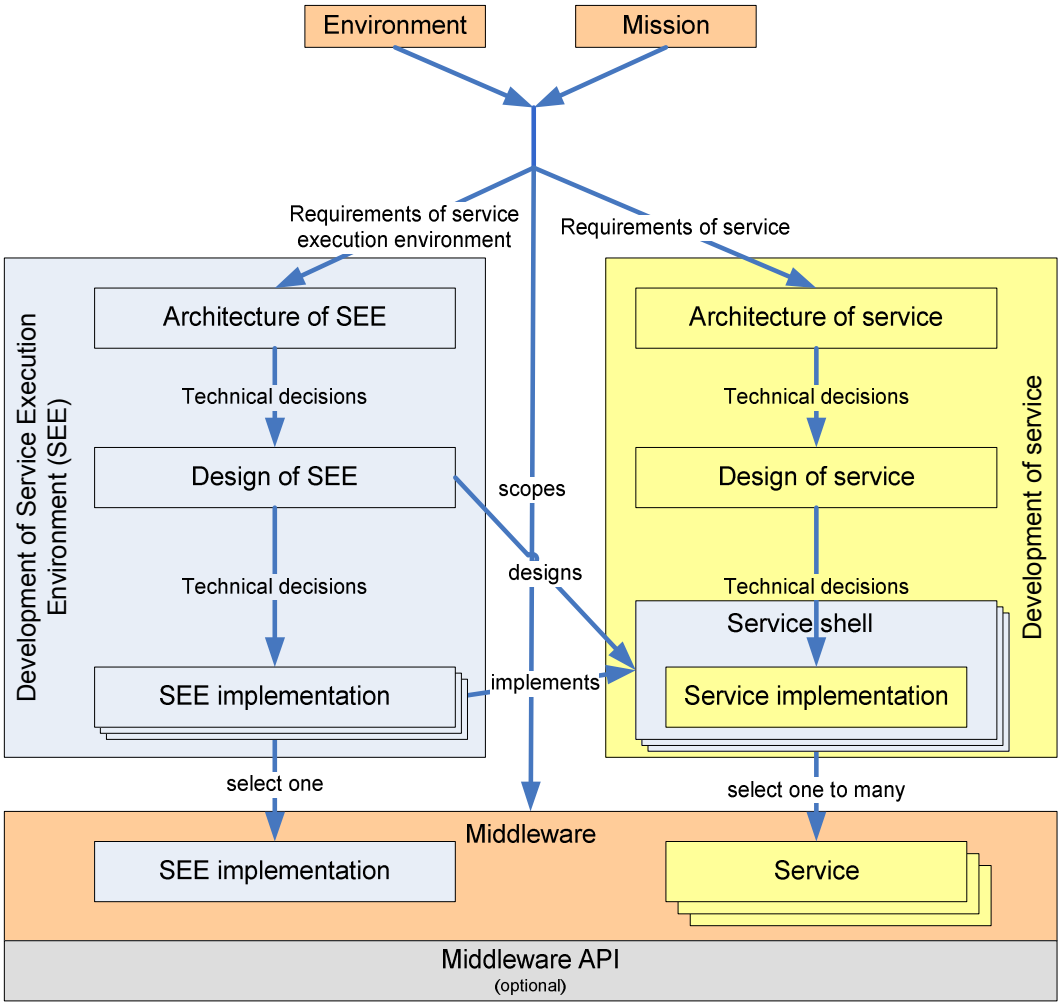

Figure 2-14: The development process used for FAME² (based on [5, 181])

(SEE). The SEE manages hosted services and the middleware that is composed of these services. Usually, the architecture of services and SEE are created together. As a result, services and SEE mutually influence each other, reducing the reusability of the architecture of services with other SEEs. FAME² follows a development process that strictly separates the development of the service execution environment and services. This development process is described in [5, 181], and depicted in Figure 2-14. Starting from an environment and a mission defining the requirements the middleware needs to consider, the requirements are separated into requirements for the SEE and for services. The environment for FAME² is ubiquitous computing. The mission of a middleware could

be "a personalisation middleware for multimedia home appliances in ambient intelligence", as described in [182]. The architecture and design of SEE and services are created independently from each other. The development process of the SEE yields a design and implementation of a service shell, where the implementation of services is embedded into. This service shell is responsible to make the service implementation compatible with the SEE, and reunifies the separate development of SEE and services. Middleware is created by selecting a SEE and a collection of services, where the services are implemented in the service shell defined by the SEE. In an optional step, a unified middleware application programming interface (API) may be provided to unify the use of the middleware.
The advantage of this development process is that the architecture and design for the SEE do not influence the architecture and design of services, vice versa. This separation promises better reuse of the architecture and the design of SEE and services [183]. Additionally, the separation allows developers to concentrate on their task, and abstract from related tasks [184].

2.2.2 Architecture

The architecture of FAME² comprises a service execution environment and a service shell. Figure 2-15 illustrates the architecture of FAME². The SEE consists of the following elements:

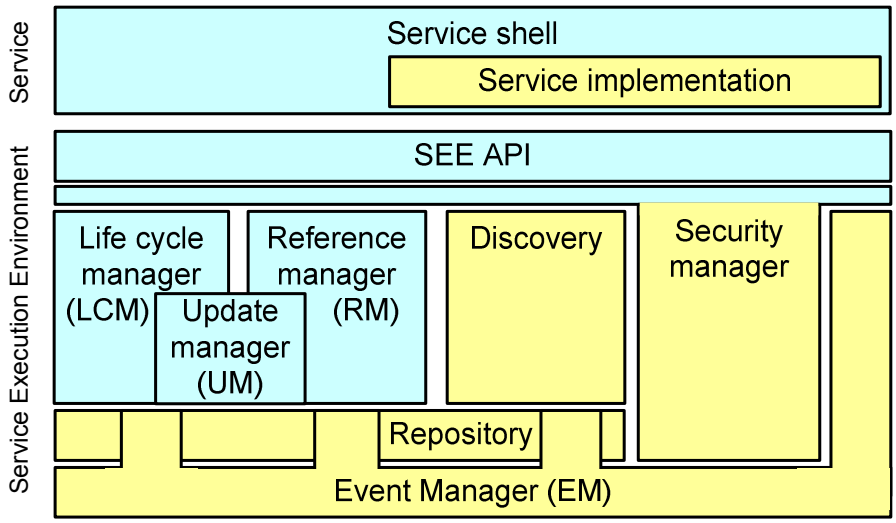

Figure 2-15: Architecture of FAME²

- The *life cycle manager* (LCM) element implements the life cycle of services, i.e. their instantiation, configuration, start and stop, and disposal. The LCM is responsible to integrate (add) and remove services from middleware. The LCM checks consistency and conformity of services before they are integrated into a middleware to ensure the middleware's integrity. The LCM can remove services from a running middleware without requiring a shutdown of the middleware.
- The *reference manager* (RM) element monitors and tracks service use. The RM is able to update the service reference[2] while a service is moved, for instance to another computer. Alternatively, the RM can redirect clients of a removed service to another service with similar functionality.
- The *update manager* (UM) element replaces services with new versions of the same service invisibly for the service clients. Optimally, clients do not detect the update of a service, but may experience a variation in delay between their request and the response from the service only. Chapter 4 describes the details how this update is realised within FAME².

[2] A service reference points to, or is a link to a service. Clients use the service reference to contact the service.

- The *repository* element manages the service logic, resources and descriptions of services installed at the device that executes the middleware, and thus the repository. The service logic implements the behaviour of the service, i.e. the algorithms of the operations of the service. The resources are data required, created, used, and managed by the service, e.g. documents, images, sounds, and so on. The repository implements operations for querying, installing, replacing, and uninstalling services.
- The *discovery* element locates services in the local middleware, as well as in remote middleware. Remote middleware are the middleware executed on computers other than the local one. Chapter 3 describes the discovery element in detail, and proposes a novel approach to integrate different existing and future service discovery solutions.
- The service execution environment uses the event oriented coordination model to notify subscribers about events, for example the presence of a new service or the request of discovering a service. The *event manager* (EM) element realises the event oriented coordination model and provides asynchronous and synchronous point-to-multipoint communication. Point-to-multipoint means that the publisher of an event may distribute it to multiple subscribers, and receives their responses. Services may use the EM for their own communication.
- The *security manager* (SM) element protects the middleware from malicious users and services, and controls the access to the SEE. The security manager is used by the repository to verify a digital signature of services that should be installed or replaced. Furthermore, the security manager controls access to all operations of the SEE, e.g. discovery, events, etc., by evaluating access control lists. An access control list is a concept to enforce privilege separation, i.e. to determine the appropriate access rights to a given resource depending on the identification of the requestor [185].
- The *service execution environment application programming interface* (SEE API) provides access to the elements described above. It is a common access point that delegates invocations to the responsible element, for example a discovery request is forwarded to the discovery element.
- The *service shell* is a "convenience building block" for service developers in which they can implement their services. The service shell ensures compatibility of the service with the SEE. The shell provides default implementations for the functionality common to all services. In FAME² this common functionality includes a service monitor and operations for a customised service life cycle. The service monitor provides information about the service vitality and state. Such information is useful to observe the operational state of the middleware, as well as to make decisions for optimisations. Possible optimisations are deactivating a never-used service, or "offloading" a heavily used service to a more powerful computer. The operations for the customised service life cycle enables configuration of the service. These operations are invoked by the LCM.

The SEE API, LCM, UM, RM and parts of the SM are part of the SEE implementation. Discovery, repository, EM and parts of the SM are implemented as services, and the SEE uses them as helper and provides access points to these services. This supports using different discovery, repository, EM and SM implementations for different middleware. Optimally, services are not aware of the current EM, SM, repository and discovery implementation of the middleware, but just use their functionality through the SEE API. How this abstraction works is demonstrated in Chapter 3 for the discovery element.

2.2.3 Design

Figure 2-16 shows the Unified Modeling Language (UML) class diagram of the service execution environment and service shell of FAME². The SEE API is the core of FAME², thus called *ICore*. It provides operations for service life cycle, discovery and global middleware configuration. The service life cycle operation *load(in Principal, in/out Service, in Parameters)* loads and configures a service and prepares it for execution, i.e. the service is integrated into the middleware. Configuring a service involves operations such as reserving resources, creating data structures like caches, etc.

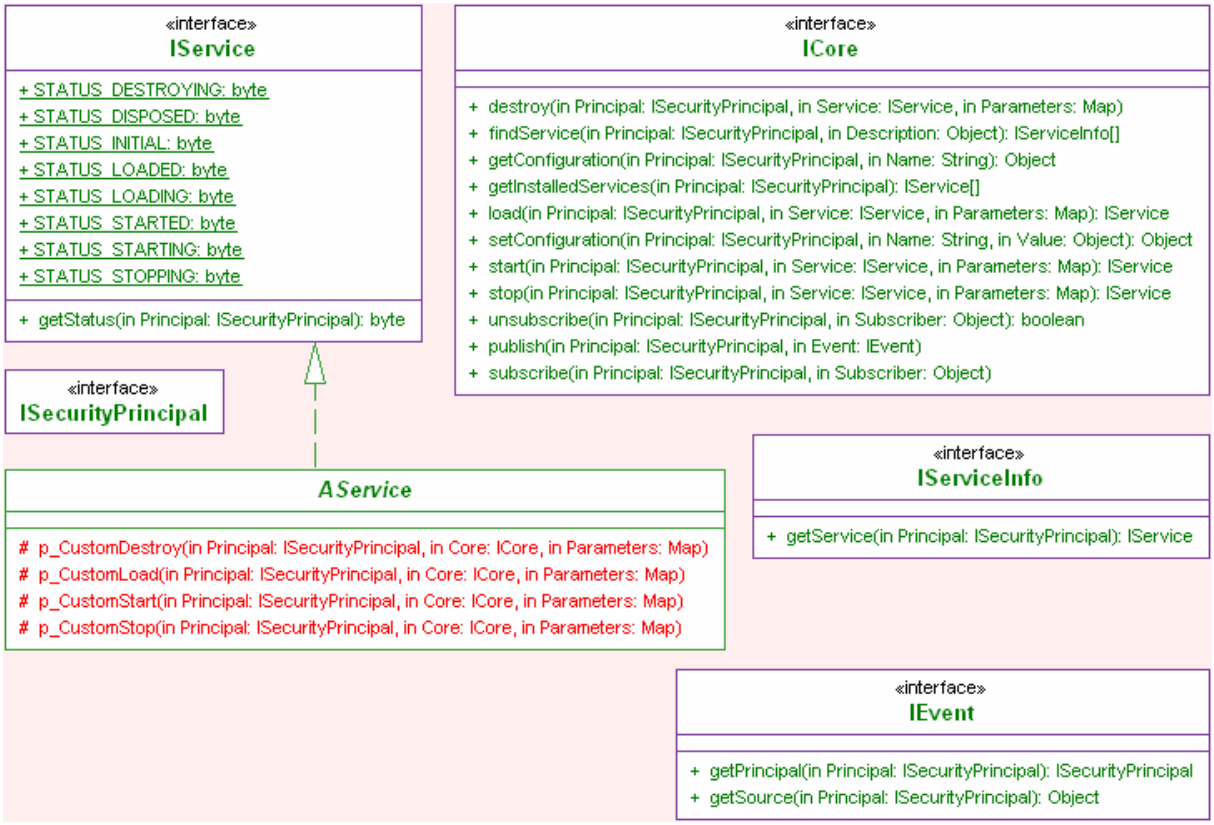

Figure 2-16: UML class diagram of the service execution environment and service shell

After successful execution of the *start(in Principal, in/out Service, in Parameters)* operation the service runs and provides its functionality to clients, e.g. applications or other services. The counterpart of this operation is *stop(in Principal, in/out Service, in Parameters)* that stops the service, i.e. its clients may no longer use its functionality. Finally, *destroy(in Principal, in Service, in Parameters)* removes the service from the middleware and free occupied resources. The service might still be installed on the local device that runs the middleware, but the service by itself is no longer part of the middleware.

The LCM ensures that the service life cycle is executed in the proper sequence, i.e. a service is first loaded, then started, then stopped, and then destroyed. After a service is stopped, it might be started again. The status of a service, e.g. whether the service is started, loaded, disposed, etc., is available from the service by itself via the *getStatus(in Principal, out Status)* operation in the *IService* interface. This interface defines status constants to determine the status of a service.

The access to discovery and the repository is realised by the operations *findService(in Principal, in Description, out ServiceInfos)* and *getInstalledServices(in Principal, out Services)* in the SEE API. The *findService(…)* operation is the service access point for the discovery element in the FAME² architecture, and called the *Open Service Discovery Interface* (OSDI). The OSDI abstracts from service discovery and allows integration of different service discovery solutions. OSDI is described in detail in Section 3.1.10. The *IServiceInfo* interface is part of the OSDI and instances of this interface provide information about found services. This interface is customised by the different service discovery solutions that OSDI abstracts from. The operation *getService(in Principal, out Service)* returns the reference to the service the instance describes. The *getInstalledServices(…)* operation in the *ICore* interface retrieves all services currently installed in the repository. Additional operations of the repository are available when discovering the local repository with the help of discovery.

Global middleware configuration is done via the *setConfiguration(in Principal, in Name, in Value)* and *getConfiguration(in Principal, in Name, out Value)*. Services may use these operations to share properties with other services in the middleware, for instance configuration parameters. The global middleware configuration is based on the shared data-space model.

The event manager element uses event channels for publishing events. An event channel is a topic based realisation of the event oriented coordination model. A topic represents a category of interest, e.g. service life cycle, security violations, and so on. The chosen design promises scalability when publishing many events to many subscribers, because the different topics are separated from each other. Subscribers subscribe to an event channel with the *subscribe(in Principal, in Subscriber)* and unsubscribe with the *unsubscribe(in Principal, in Subscriber)* operation. Events, being instances of the *IEvent* interface, are published in a channel via the *publish(in Principal, in Event)* operation. Events provide information about the source of the event (*getSource(in Principal, out Source)*) and the principal who created the event (*getPrincipal(in Principal, out Principal)*).
The *IService* interface and *AService* class constitute the service shell, a "convenience building block" for implementing services. The *IService* interface defines operations and constants to query the status of a service, e.g. whether the service is started, loaded, or disposed. The *AService* class declares operations that service developers can adapt to customise the service life cycle. Additionally, the *AService* class implements the update manager and reference manager. A detailed description is given in Chapter 4.
The principal (*ISecurityPrincipal* interface) is the first parameter of every operation of FAME². The principal identifies the invoker of an operation. This identification is necessary for features like:

- **Access control and encryption:** the principal identifies the invoker of an operation and thus permits to perform access control to check whether the invoker is granted to execute this operation, or not. Furthermore, the information may be used to decrypt parameters for operations that are passed encrypted, and to encrypt their result. This increases security when operations are invoked remotely over an unsecured network, for example WLAN.
- **Billing:** if the use of a service should be charged, then the user of the service, or the client that uses the service on behalf of a user, needs to be identified so he can be charged.
- **Auditing:** to trace the operation of middleware, and to analyse the behaviour of users, it is necessary to identify them.

2.2.3.1 Access control

Users may tamper software. In SOA, this tampered software may be services that then tamper middleware [186]. To limit negative effects imposed by malicious services and users, access control limits access to middleware resources, i.e. services [185]. Access control permits or denies subjects to access services. Subjects are clients of services. The decision for permit or deny is done by evaluating security policies. Access control models define the structure of security policies.
In the *discretionary access control* (DAC) model, first described by Lampson [187], the owner of resources defines who will have what kind of access. DAC is used in the first file systems for the UNIX operating system where the owner of a file defines who can access it. The disadvantage of DAC is that everyone who has access to a resource can overtake the ownership, allowing for theft of ownership. Furthermore, DAC does not support groups, resulting in a scalability problem for management of security policies [188].
The *mandatory access control* (MAC) model uses security policies defined by an administrator [189-191]. With MAC there is no theft of ownership possible. However, MAC intensifies the problem of scalability because security policies are defined by few system administrators, instead of a large group of owners as in DAC.
The *role based access control* (RBAC) model, described in [192-194], introduces groups of subjects, and privileges to grant privileges. Introducing groups increases scalability of defining security policies. Introducing privileges to grant privileges separates the concept of administration of security policies and owners of resources to protect.
The *context aware access control* (CBAC) model, developed by Moschgath [185], extends RBAC by introducing a new parameter, context, into security policies. This parameter influences the evaluation of security policies depending on the context of the resource to protect and the subject that accesses the resource. Moschgath demonstrated in her thesis that such parameter is especially useful in ubiquitous computing [185].

Another approach is the *trust access control* (TrustAC) model, described in [195, 196], where privileges are granted to trust levels instead of roles or subjects. TrustAC has advantages in ubiquitous computing where clients of services may be previously unknown and thus not have access permissions assigned. The more often a client behaves well, the higher its trust rating will be and the more privileges will be granted. Nevertheless, the client has to identify itself to be assigned with a trust level.

Beside the access control model, it is necessary to decide on what level access control is performed. For middleware based on the concept of service oriented architecture, three different levels for access control are possible:

- **Middleware level:** Access control on the middleware level grants or denies access to the middleware as a whole. The advantage is that the security policies need to be evaluated only once, when clients access the middleware for the first time. All subsequent uses of the middleware do not need to be checked against the security policies. The disadvantage is that access is granted on a very coarse level, granting or denying access to the whole middleware.
- **Service level:** Access control on the service level grants or denies access to single services of middleware. The advantage, in comparison to access control on the middleware level, is the finer granularity of access control, granting or denying access to services instead of the whole middleware. However, the access control on service level requires that security policies are evaluated every time a client accesses a service it did not access before.
- **Operation level:** Access control on the level of operations grants or denies access to single operations of services of middleware. This is a very fine granularity of access control, allowing clients to access parts of services, while denying access to other parts. The disadvantage of this fine granularity of access control is the increased number of required evaluations of the security policies, compared to access control on the middleware or service level.

Nevertheless, all three levels of access control have in common that clients need to identify themselves. FAME² supports all levels of access control, and requests identification information from clients with the invocation of any operation in the form of a principal that clients have to provide as parameter. FAME² supports any access control model where the identification of the client, that is the subject, is sufficient. As a consequence, all of the above described access control models, DAC, MAC, RBAC, CBAC and TrustAC, are useable with FAME².

As the evaluation of security policies consumes resources, and because there might be situations where access control is not required, for example in home networks with a fully trusted environment, FAME² designs access control as a service that can be integrated into, replaced, or removed from middleware. When leaving a fully trusted environment, a service implementing access control can be integrated into the middleware, without requiring a restart of the middleware and its services. When entering an environment where a different level and model of access control is required, the service implementing access control can be replaced, again without restarting the middleware. When entering a fully trusted environment, the service implementing access control can be removed from the middleware. Additionally, FAME² supports that multiple services implementing access control are used in the same middleware in parallel. However, this requires a proper configuration of security policies to avoid mutual exclusions with the result that all users are denied all permissions.

2.2.3.2 Tagging interfaces

Interfaces are contracts that define the interactions between entities, like objects in object oriented programming. Interfaces in programming languages are introduced in [197, 198] to specify software modules and their relations to each other. A review on the use of interfaces in the Java programming language can be found in [199, 200].

In addition to define shared boundaries and interactions between services, FAME² uses interfaces to tag them for restrictions and feature requests. Tags are descriptors, attached to objects, to identify and classify the tagged object.

FAME² defines three tags that are used by the service shell to control the behaviour of services. The *ILoadedInterface* tag is attached to services that need to be loaded before their operations are used. The *IStartedInterface* tag is attached to services that need to be started before their operations are used. Without these tags, operations of services can be used even before a service is loaded, i.e. configured. The third tag defined by FAME² is the *ISignalInvocation* tag that advices the service shell to signal the invocation of every operation of the tagged service in the event manager.
Services may introduce additional tags, e.g. to tag services to add functionality like remote communication, billing, load balancing, etc. In [201, 202] examples are given how tags are used to add remote communication protocols to services. In [203] it is demonstrated how to deal with loss of connectivity in ubiquitous computing by using tags.

2.2.4 Implementing services

FAME² is implemented in the Java programming language. Developing a FAME² service is very similar to a normal Java class, and requires only marginal extra work. The advantage of FAME² is that it allows integration of services realised for other frameworks, for example Enterprise JavaBeans, and on the other hand supports the integration of FAME² services into other frameworks.
Every FAME² service needs to declare its operations in an interface. Interfaces introduce a level of abstraction between the interface of a service and its implementation. This abstraction is of advantage when updating a service implementation, because clients of the service are bound to the service interface, and not its implementation. Clients explore and access the operations of a service by the service interface. Every operation that is not declared in the service interface is not available for clients. In FAME² operations have to follow the following three definitions:

- The first parameter of every operation must be the principal that identifies the invoker of the operation. This principal must be of the type *ISecurityPrincipal*. The principal is required for features like access control, billing, and auditing.
- The Java programming language, and its virtual machine that executes Java software, requires that operations declare the exceptions they throw. Exceptions indicate exceptional conditions, like division by zero, communication failures, or missing resources to perform an operation. Java does not permit operations to throw any other exception than it has declared. As FAME² is able to extend the functionality of services at runtime, i.e. without modifying the service by itself, it can be necessary to throw exceptions that might not be declared. To avoid this situation, where an extension cannot be assigned to a service because the extension throws an exception that was not declared, every operation shall throw a so called *untyped* exception. To declare an untyped exception, the operation has to declare the exception *java.lang.Throwable*. Other programming languages, for example C++, do not require declaring exceptions.
- The Java virtual machine (JVM), which executes software written in the Java programming language, features garbage collection that frees memory occupied by objects that are no more used. Objects are instances of classes. For performance reasons, the JVM caches the classes of objects. A class is only garbage collected, i.e. removed for the cache, when the JVM does not manage any objects of the class. This allows the JVM to detect similarities between objects, i.e. that they are instances of the same class. Only when the class is removed from the memory by a garbage collector, a new version of the class can be loaded by the JVM to reflect an update of e.g. a service (which is implemented as a collection of classes). It is assumed that the majority of updates affect the implementation of a class, i.e. the algorithms and control flow, and not the interface. As a result, the JVM may not share the class of objects, but instead should share the interfaces that classes implement. This allows that the JVM garbage collector removes the definition of the class of an object from the memory as soon as the object is removed from the memory, while the interface is cached until every object that is an instance of a class implementing the interface is garbage collected. To support the removal of classes from the memory as soon as their instance is removed from the memory, the signatures of operations shall contain parameters and return values of the following types only:

- Interfaces (e.g. ISecurityPrincipal or IServiceInfo)
- Native types of the Java programming language (i.e. void, boolean, char, byte, short, int, long, float, and double)
- Classes part of the Java standard class library (e.g. java.lang.String, java.util.Map)
- Classes which are part of a third party library that are endorsed into the Java virtual machine. Endorsing a library into the JVM requires access to the computer running the JVM, and can be done only before the JVM is executed
- However, instead of endorsing third party libraries, it is proposed to encapsulate the library as a FAME² service, allowing the library to benefit from all the features of FAME², e.g. the possibility of access control, or update. The bridge pattern is a structural design pattern to implement such encapsulation [16, 204].

The service implements the interfaces where its operations are declared, and the *IService* interface defined by FAME². Figure 2-17 shows the UML class diagram of an example service providing

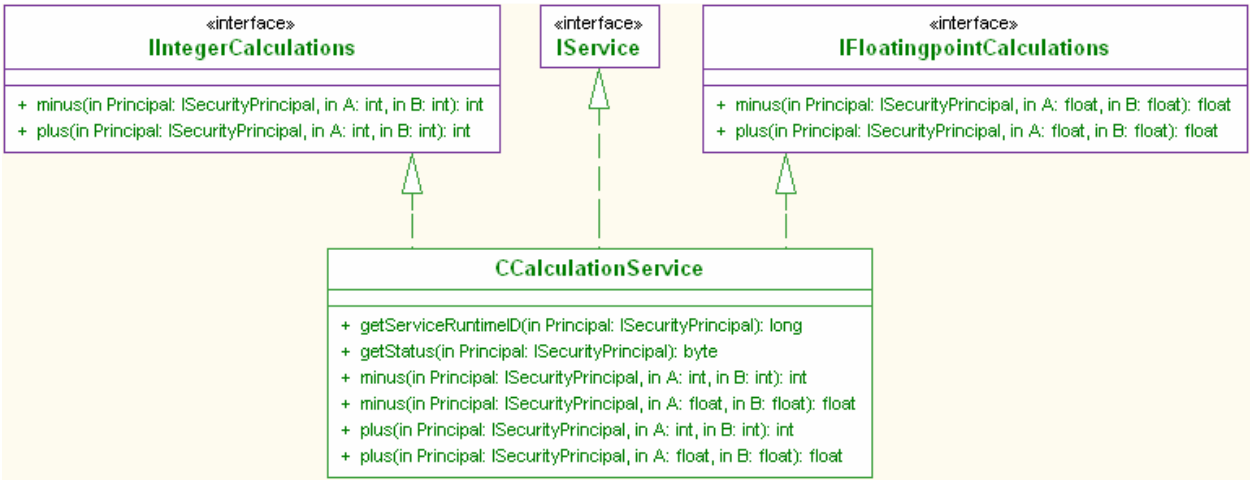

Figure 2-17: UML class diagram of a calculator service

mathematical operations. The operations of the service are declared in two interfaces, one for integer (*IIntegerCalculations* interface) and one for floating point calculations (*IFloatingpointCalculations* interface). The service (*CCalculationService*) implements the interfaces of the two interfaces and the operations declared in the *IService* interface. The source code of the calculation service is given below:

Program 2-1: Source code of the calculation service example

```
1   public class CCalculationService implements IFloatingpointCalculations,
      IIntegerCalculations, IService {
2     public float minus(ISecurityPrincipal P, float A, float B) throws Throwable {return A-
      B;}
3     public float plus(ISecurityPrincipal P, float A, float B) throws Throwable {return A+B;}
4     public int minus(ISecurityPrincipal P, int A, int B) throws Throwable {return A-B;}
5     public int plus(ISecurityPrincipal P, int A, int B) throws Throwable {return A+B;}
6
7     public long getServiceRuntimeID(ISecurityPrincipal Principal) throws Throwable { … }
8     public byte getStatus(ISecurityPrincipal Principal) throws Throwable { … }
9   }
```

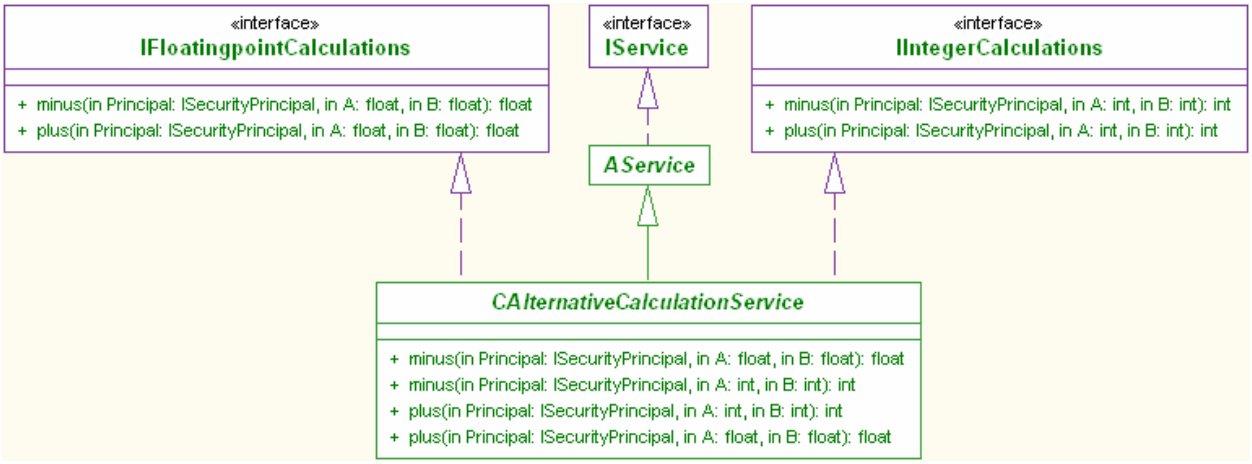

Figure 2-18: UML class diagram of an alternative calculator service

Alternatively, the service may be derived from the *AService* class that is a convenience building block and provides the implementation of the operations defined in the *IService* interface and the service life cycle operations required by the LCM. Figure 2-18 shows the UML class diagram of this alternative way of service creation. The *CAlternativeCalculationService* does not need to implement the operations declared in the *IService* interface because the *AService* class provides a standard implementation. The source code of the alternative calculation service is shown below:

Program 2-2: Source code of the alternative calculation service example

```
1    public class CCalculationService extends AService implements IFloatingpointCalculations,
       IIntegerCalculations {
2      public float minus(ISecurityPrincipal P, float A, float B) throws Throwable {return A-
       B;}
3      public float plus(ISecurityPrincipal P, float A, float B) throws Throwable {return A+B;}
4      public int minus(ISecurityPrincipal P, int A, int B) throws Throwable {return A-B;}
5      public int plus(ISecurityPrincipal P, int A, int B) throws Throwable {return A+B;}
6    }
```

In the UML diagram depicted in Figure 2-19 the calculation service is extended to require to be loaded before any integer calculations can be done, and to be started before any floating point

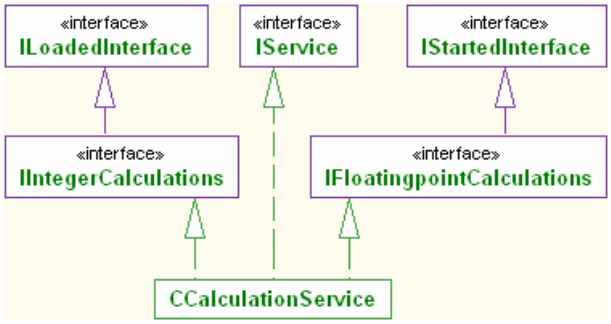

Figure 2-19: UML class diagram of the calculator service with tagged interfaces

calculations are possible. This behaviour is achieved by tagging the *IFloatingPointCalculations* interface with the *IStartedInterface* tag and the *IIntegerCalculations* interface with the *ILoadedInterface* tag. The implementation of the *CCalculationService* class remains unchanged. Similarly, the service can be tagged to signal the invocation of its operations.

2.3 Evaluation and comparison

The *Framework for Applications in Mobile Environments 2* is designed to support the development of middleware for distributed computing systems in ubiquitous computing. FAME² follows a minimal approach, where the mandatory functionality of a service execution environment that hosts

services is limited to an absolute minimum. Other functionality that is not implemented in the SEE is realised by services that the SEE integrates into the middleware at its runtime. A development process that emphasises separation of concern allows designing services without dependencies on the SEE.
Middleware developed with FAME² is reconfigurable because services can be added to and removed from the middleware without performing a restart. Services are updated by the update manager without requiring its clients to be aware of this update.
The minimal approach of FAME² makes it the perfect candidate for developing middleware executed on resource constrained devices. The uncompressed size of the prototype implementation of the FAME² SEE is only 33408 bytes. Compared with the several megabytes of typical CCM and EJB implementations, and the 90 kilobytes (packed) of the very minimal SOAP server described in [156], FAME² is the smallest implementation of a framework for reconfigurable middleware. The FAME² service execution environment requires 4208 bytes of memory on top of a JVM with the *CAlternativeCalculationService* service [3], whereas PCOM requires approx. 190 kilobytes, the minimal SOAP server requires 125 kilobytes, and the Sun Application Server, a container for EJB, requires more than 20 megabytes. This very low physical size and memory overhead makes FAME² suitable for embedded devices like sensors. Furthermore, the FAME² SEE does not consume any CPU power when no discovery, service life cycle or update is done, i.e. the middleware does not reconfigure.
Loss of connection to remote services is handled by the reference manager in the FAME² SEE. Upon detection of a loss of connection, the RM tries to discover another similar service that can be used instead. If there is no replacement, then the situation is signalled to interested applications. Middleware developers may decide to integrate services into their middleware that provide special handing of disconnected operation, e.g. caching of requests, and introduce new tags used to label services to improve their behaviour in the situation of disconnected operation.
The programming model of middleware based on FAME² is defined by its services. FAME² does not restrict the programming model. Logical mobility is supported by the reconfiguration features of FAME², and by providing a homogeneous platform for services, i.e. using a service shell that is designed for the SEE and where the service implementation is embedded into.
FAME² is based on the concept of service oriented architecture and allows creating middleware based on this concept.
The coordination model that a middleware may select depends on the selection of services that implement them. If there is a service implementing the event oriented coordination model, then FAME² may use it to notify subscribers about events in the SEE or service shell. Other coordination models, and their implementations (i.e. protocols), can be realised as services and then easily integrated into FAME².
The Java programming language provides access transparency for the FAME² prototype. Location transparency is realised by the discovery element, which does not distinguish between the discovery of local and remote services, and that returns at minimum a reference of the discovered services that clients may use immediately without doing any further operations. Relocation and some aspects of failure transparency are provided by the reference manager and update manager in the FAME² SEE. Services implement the migration, persistence, replication and transaction transparencies.
In addition to CCM, FAME² supports logical mobility and development of middleware that is executed on resource limited devices. Compared with EJB, FAME² supports the peer-to-peer programming model, logical mobility, and is based on the concept of service oriented architecture. Web Services provide all features that FAME² does except logical mobility. OSGi is a framework for deployment and does not provide support for any coordination model. Additionally, OSGi does not expect that a loss of connection to the devices providing services can happen. While the features of ReMMoC are close to be identical to FAME², ReMMoC lacks of reconfiguration of middleware

[3] All measurements for FAME² were done with the *Test & Performance Tools Platforms* that is part of the Eclipse project and initially developed by Fraunhofer FOKUS.

at its runtime. ReMMoC requires a restart when a middleware needs to integrate new services. The same restriction applies for PCOM. Furthermore, ReMMoC and PCOM do not support logical mobility. The features of RUNES are similar to FAME². RUNES is a recent development and in its early specification phase. Yet, the current design of RUNES does not support update of components while they are in use. Logical mobility is limited, because the platforms, i.e. operating system, a component moves between must be identical.
A limitation of all reviewed frameworks, which support logical mobility, is the lack of support for security. These frameworks delegate security concerns to the platform, i.e. operating system and a Java virtual machine where used. However, the security mechanisms of platforms are not sufficient for logical mobility. There is no support to check the integrity of services, and it is not possible to restrict partial access to services. FAME² supports security by access control on the following three levels: middleware, service, and operation. By realising access control as a service, the implementation of access control can be changed on demand and the environment, or deactivated at all to preserve resources.

3 Service discovery

A vital feature of ubiquitous computing is the ability to find available services and resources on demand. Also, service discovery is an important feature of service oriented architecture (SOA) [205]. Service discovery is used to discover and exploit new services in distributed computing systems. Numerous middleware, e.g. Java Remote Method Invocation (RMI), Jini Network Technology (JINI), Common Object Request Broker Architecture (CORBA), BASE, etc. feature service discovery. Service discovery consists of the following two aspects: finding services [12] and exploiting services [206].

Finding services is the process to discover the services itself, i.e. their location. Exploitation of services is the process to find out the capabilities of discovered services, i.e. how their interfaces are designed, what operations they provide, and how to use the interfaces to invoke required operations. In some service discovery solutions the exploitation is implicit because clients request for a service with e.g. a particular name and expect the discovered service to have a specific interface.

There are three types of service discovery [207]:

- **Naming:** Naming is the binding of a service name to the address of the described service. Services register at a lookup service, which is the broker in SOA. Each service registers with a more or less descriptive, dedicated name and location information, e.g. IP address, TCP port number and unique distributed object identifier. Clients query the broker to lookup the location of a service that has registered with a specific name. A service is found when the name used by the service for registration matches the name in the client's query.
- **Trading:** Trading binds services to service descriptions. In contrast to naming, trading allows a more detailed description of the service and its capabilities to be registered at the broker. Clients query services with a specific aspect in the service description. This allows clients to lookup services with a vague query, unlike naming, which requires a specific name. Furthermore, clients can request the descriptions of looked up services to explore their capabilities. The downside of trading is a more complex realisation of brokers, queries and descriptions in comparison to naming.
- **Flexible discovery:** Naming and trading are two types of service discovery that bind services to certain descriptions. But they do not tackle the question of how to discover a broker or lookup service. Flexible discovery is about discovering brokers that can be queried for services. There are two possibilities how this is solved in existing middleware. First, computers in distributed computing systems are probed with a request sent to all computers. Those computers that answer to this request provide broker functionality. Second, computers running a broker frequently announce themselves in the distributed computing system, e.g. via beaconing. Other computers listen to the announcements and record sources of these announcements, because the sources refer to computers running a broker.

Service discovery consists of descriptions of software elements that can be discovered, queries to lookup software elements, responses that link to the software elements, and a protocol that matches queries with descriptions and creates the responses.

There are numerous existing solutions for service discovery. They are designed for use in specific distributed computing systems, with specific assumptions of their developers in mind. This variety of existing solutions is the reason why they are not suitable for use in multiple distributed computing systems designed for ubiquitous computing [11]. At the same time, there might be no universal solution that fits well into all possible distributed computing systems designed for ubiquitous computing. As a result, clients that want to use service discovery in different distributed computing systems have to choose which service discovery solutions they support. This fact complicates the development of clients [12, 14]. Simplifying the development of clients is an objective of middleware. Interoperability of different service discovery solutions is a simplification for clients [12, 14, 208, and 209]. Optimally, interoperability and integration of service discovery take place without requiring modification of clients and services. Furthermore, a solution of

interoperability shall be able to integrate new service discovery solutions, without requiring modifications or restarting of middleware and clients.
The next section gives an overview of existing service discovery solutions and attempts that tackle interoperability questions between them. They all conclude that these existing attempts are not sufficient to comply with the requirement of interoperability. Based on the experiences of existing attempts, a new approach for interoperability of service discovery is designed and presented in Section 3.1.10. This approach allows interoperability and integration of existing and future service discovery solutions, as long as they adhere to the naming or the trading service discovery types. In Section 3.3 an evaluation of the approach is given and it is compared with existing attempts that tackle interoperability between different service discovery solutions.

3.1 Survey on existing service discovery solutions and approaches

In this section a collection of existing service discovery solutions and approaches is presented. The collection of works deals with questions of interoperability among different service discovery solutions and approaches. Detailed comparisons on different aspects, like security, communication in wireless networks, etc., can be found in [210-212].

3.1.1 Internet Domain Name Service

The Internet Domain Name Service (DNS) is a service discovery solution used to find Internet services. It is based on naming [213-215]. The address of Internet services is a series of numbers, known as the IP address. Additionally, some Internet services may have a TCP or User Datagram Protocol (UDP) port number as part of their address. E.g. the address of the Internet service *World Wide Web* (WWW) includes a TCP port number that is used by its transport protocol, the *hypertext transport protocol* (HTTP). As IP addresses are considered to be relatively difficult to memorise, whereas names in textual form are not, the DNS allows Internet services to map their IP addresses to names in textual form. The textual form is called the domain name. Clients use the DNS to lookup the IP address of a domain name.
The name, which the client looks up, needs to match exactly the domain name of the Internet service used when it registers its name at the DNS. Otherwise the IP address of the Internet service cannot be resolved and the client will not be able to locate that Internet service.
The DNS has no support for clients and services to find computers running DNS. The location of DNS consists of the IP address of the device running it, and a UDP port number, which is 53 by default.

3.1.2 Java Remote Method Invocation registry

The Java Remote Method Invocation (RMI) is a middleware that includes a service discovery solution based on naming, the RMI registry [216]. The service discovery of RMI is similar to DNS. Services register at the RMI registry with a free selectable name and the location information that consists of the IP address, the TCP port number and a unique distributed object identifier. A Client looks up a service by querying the RMI registry with a name, which the service had registered at the registry. If the name matches the registered name exactly, then the IP address, port number and unique distributed object identifier are resolved. The client uses the retrieved information to setup a communication channel with the service.
RMI has no support for clients and services to locate the RMI registry. The location of the RMI registry consists of the IP address of the computer running it, a TCP port number, which is 1099 by default, and a standardised unique distributed object identifier, which is 0 by default. Thus, the RMI registry must be known a priori to both clients and services. A partial workaround for this shortcoming is to execute an echo service that simply replies to broadcasts or multicasts. With such echo service computers running clients and services are able to locate the IP addresses of computers running the RMI registry. However, this does not provide information about the TCP port number. By default, TCP port number 1099 is used by default for the RMI registry, but this TCP port number can be changed to any other port number if desired. Another partial workaround is to use

DNS to resolve the IP address of computers running a RMI registry. This would require that computers running a RMI registry register in the DNS with a domain name. Still, clients then have to know this domain name and the port number the RMI registry listens at.

3.1.3 Jini network technology

The Jini network technology (JINI) middleware, which is based on RMI, includes the JINI lookup service which is a service discovery solution based on trading [36]. Services register at JINI lookup services, the brokers, with a description of their own. The description is mapped to the address of the service, which is identical to the address in Java RMI and consists of the IP address of the computer running the service, a TCP port number, and a unique distributed object identifier. The process of registering a service with its description at a JINI lookup service is called *join*. The client looks up services by querying the lookup services with service descriptions. The process of lookup is called *discover*.

The service description is realised as a collection of *entries*. An entry is a part of the service description. JINI allows definition of new entry types to describe any aspect of a service. For convenience, there are a number of predefined entries in JINI (cited from [217, 218]):

- *Address - the address of the physical component of a service.*
- *Comment - a free-form comment about a service.*
- *Location - the location of the physical component of a service. This is distinct from the Address class in that it can be used alone in a small, local organization.*
- *Name - the name of a service as used by users. A service may have multiple names.*
- *ServiceInfo - generic information about a service. This includes the name of the manufacturer, the product, and the vendor.*
- *ServiceType - human-oriented information about the "type" of a service. This is not related to its data or class types, and is more oriented towards allowing someone to determine what a service (for example, a printer) does and that it is similar to another, without needing to know anything about data or class types for the Java platform.*
- *Status - the base class from which other status-related entry classes may be derived.*

JINI supports features like events and leases for service registration. Leases are used to keep track of service registration. Services whose leases are expired are removed from the lookup services. Causes for expired leases could be that services forgot to deregister, or are unable to deregister once their device is about to leave the reachability of the JINI network, e.g. switching off the computer running the service, or loosing network connectivity. Events enable notification of clients upon new service registrations and unregistrations.

JINI supports federations of lookup services. Lookup services replicate themselves within a federation [217, 218] to improve scalability and reliability. The architecture of JINI is depicted in

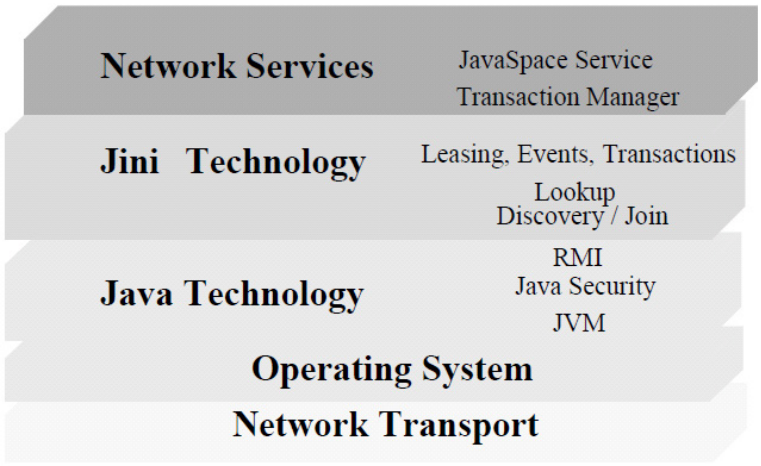

Figure 3-1: Architecture of Jini network technology [210]

Figure 3-1. The network services layer provide high level services, e.g. JavaSpace for tuple spaces. Own services might be implemented using those network services or being build directly on top of the JINI technology layer. This JINI technology layer provides the basic service discovery functionality, i.e. lookup to discover services, the registration mechanism for services, and the lease concept. JINI technology bases on Java technology, which provides the features like RMI that JINI is build upon. The operating system executes the Java environment, including JINI services, and also realizes access to the network transport used to communicate with other devices.
Flexible discovery is supported by JINI via network multicasts. Clients send a multicast datagram to detect JINI lookup services. Lookup service responds with the IP address and the TCP port number of the computer they are executed on. Furthermore, the unique distributed object identifier of the lookup service is returned.

3.1.4 Universal Plug and Play

> *Universal Plug and Play (UPnP) is an architecture for pervasive peer-to-peer network connectivity of intelligent appliances, wireless devices, and PCs of all form factors. Although it's introduced as an extension to the plug and play peripheral model, UPnP is more than a simple extension to it. In UPnP, a device can dynamically join a network, obtain an IP address, convey its capabilities upon request, and learn about the presence and capabilities of other devices. Finally, a device can leave a network smoothly and automatically without leaving any unwanted state behind. Universal Plug and Play leverages TCP/IP and the Web technologies, including IP, TCP, UDP, HTTP and XML, to enable seamless proximity networking in addition to control and data transfer among networked devices in the home and office. – cited from [210]*

The service discovery of UPnP is based on the concept of trading and supports flexible discovery. The objectives of UPnP are zero-configuration, location transparency and automatic discovery of services [219]. Zero-configuration means that there shall be no setup and no configuration needed in order to integrate new services into an UPnP enabled network. This is achieved by defining the IP address and the UDP port number that every computer must use to be integrated into a UPnP enabled network. Any violation to this definition results in an exclusion of that particular non-compliant computer from the UPnP enabled network. Invisible networking refers to the fact that services shall not be aware of that services may be running on another computer than on the client. This is achieved by using network communication protocols for all kinds of communication, even for communication within one computer where no network is involved. Automatic discovery of services is realised by periodic announcements of services and by multicasts that probe for new devices periodically.
Figure 3-2 shows the protocol stack of UPnP. The service discovery protocol of UPnP is *Simple Service Discovery Protocol* (SSDP) [220], which is a simplified version of the Service Location Protocol (SLP) [221-223]. SSDP uses *HTTP over multicast* (HTTPMU) and *HTTP over UDP* (HTTPU). It is used for the announcement of new services and locating existing ones. Services are described in eXtensible Mark-up Language (XML) documents.
A disadvantage of UPnP is the absence of dedicated brokers that would compute the comparison of a client's query and the service descriptions of available services. When a client looks for a particular service, the client has to download the XML documents with the service descriptions of available services, and to evaluate them locally. This may result in increased network load when many clients search for services often without caching the downloaded XML documents. This results in the XML documents to be downloaded by many clients many times.

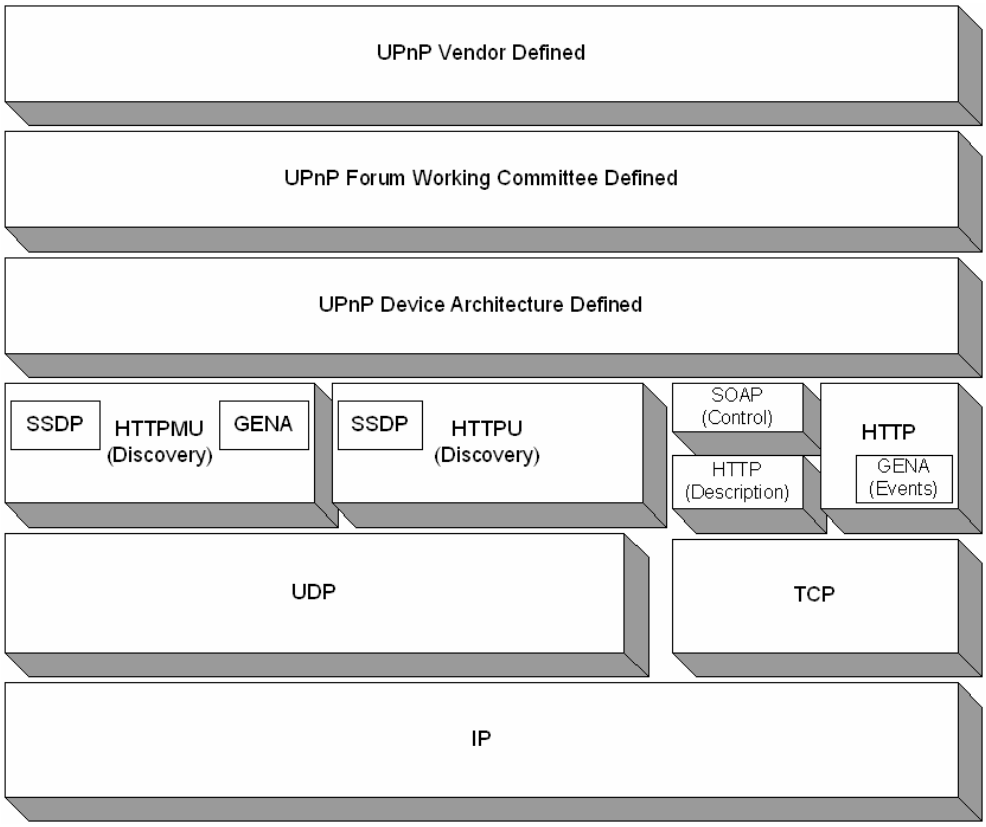

Figure 3-2: Protocol stack of Universal Plug and Play [219]

Furthermore, UPnP is developed for networks with a limited expanse, e.g. maximum a few dozens of computers. In contrast, other service discovery solutions, e.g. DNS, are designed for networks with millions of computers.

3.1.5 Bluetooth service discovery protocol

In [224], the Bluetooth service discovery protocol is described as the following:

> *Bluetooth SDP is an example of already existing SD working in an ad-hoc environment. SD in Bluetooth works by a request/response model where requests and responses use the Bluetooth specific Protocol Data Unit (PDU) units, which are sent over L2CAP channels.*
>
> *Service Discovery Protocol (SDP) in Bluetooth involves an SDP server and SDP clients. The server maintains service records and handles service requests from the clients.*
>
> *A service is described by a service record, that contains the attributes of the service, that consists of an attribute ID and a value. Two of the attributes describe service availability and estimated service lifetime. Services, capabilities, and devices are identified by so-called Universally Unique Identifiers (UUIDs). A service search is based upon a match in a list of requests UUIDs and any device UUID.*
>
> *The Bluetooth specification allows to create own service records, as long as the important attributes are represented as UUIDs, since those are needed for the service discovery protocol. SDP has an extension, Extended Service Discovery Protocol (EDSP), that enables SDP to connect and perform service discovery on UPnP networks. Solutions also exist to support UPnP and Salutation. – cited from [224]*

Summarised, the Bluetooth service discovery protocol requires that a service is mapped to a Universally Unique Identifier (UUID) and the clients have to know the UUIDs of the services they are looking for, similar to the service discovery of Java RMI. The Bluetooth service discovery protocol supports flexible discovery, i.e. no configuration is required to detect computers running services and to make them detectable. The Bluetooth service discovery protocol is limited to the

spatial extension of the Bluetooth radio, which is usually between 10 and 100 meters. Locating services that are running on computers more distant is not supported by Bluetooth.

3.1.6 Service discovery for Web Services

Web Services is a technology that gain attention within last years as part of the hype around Service Oriented Architecture. Part of Web Service technology is service discovery functionality, which by itself is a Web Service [225]. In Web Service technology, two basic concepts for service discovery are used: registration and lookup, and announcement. In *registration and lookup*, a Web Service registers itself, or is registered by other party, at a repository. A client queries in the repository by issuing a query that the repository compares with its registered service descriptions. The repository may be centralised (i.e. there is only one repository) or distributed (i.e. many repositories with possible replication of their information). In an *announcement*, Web Services announce and offer (or tout) themselves to clients. In the concept of announcement, repositories are considered as clients, allowing for integration of both concepts.
Due to its attention that Web Service technology got, numerous approaches for service discovery exist for this technology. This sub-section briefly explains some of them. Yet, as will be seen during the review of those service discovery approaches, while various solutions compete with each other, there is no solution that tries to integrate the existing solutions to form a of best-of-breed solution.

A very trivial solution of making Web Services discoverable is to describe them on a Web page (i.e. *Hypertext Mark-up Language* (HTML) page). The advantage of this solution is simplicity. Depending on human interaction to find the Web page describing a particular Web Services is the disadvantage of this solution. The solution requires a two-step process, where humans have to first discover the Web page, and only then the Web Service can be discovered.

The *Universal Description, Discovery and Integration* (UDDI) is a discovery solution for Web Services that is similar to the *Domain Name System* (DNS) [226, 227].

> *The UDDI specifications include a) SOAP APIs that allow querying and publishing of information, b) XML representation for the registry data model and the SOAP message formats, c) WSDL interface definitions of the SOAP and d) APIs Definitions of various technical models that facilitate category systems for identification and categorization of UDDI registrations – cited from [226]*

Part of UDDI is a repository, which itself is a Web Service. The repository stores the descriptions of registered Web Services and processes discovery requests. UDDI supports different access policies to restrict access to the repository. Access can be either public, allowing everyone to query the UDDI repository, protected, allowing access to a restricted group only, or private, allowing access to a restricted group that must be part of one administrative domain, e.g. one company. Shortcomings of UDDI are:

- UDDI does not notify clients about the changes of a Web Service registration. If a Web Service changes its registration, clients may end up using outdated, possibly invalid, information while accessing the previously retrieved Web Service. A workaround is to use *Web Services Notification* [228] that provides events (notification) for Web Services and their clients.
- UDDI does not provide any mechanism to find a UDDI repository. Clients need to know the location of a UDDI repository in order to be able to locate available Web Services.

In [229] an extension for UDDI to support *Quality of Service* (QoS) information with Web Services registrations is proposed. The extension adds *qualityInformation* to the Web Service registration information in UDDI. When querying UDDI for a Web Service, this *qualityInformation* can be taken into consideration for selecting a Web Service. The *qualityInformation* is linked to the **binding** and it enables a very fine granularity of QoS specifications for Web Services. The QoS information supported are *runtime related QoS*, *transaction support related QoS*, *configuration management and cost related QoS*, and *security related QoS*.

The advantage of the extension is that it does not interfere with traditional UDDI. On the other hand, because it is just an extension, the disadvantages of UDDI, like how to locate a UDDI repository, remain.

A functionality that is similar to UDDI is provided by *electronic business XML* (ebXML) [230]. ebXML describes business processes and makes them available via an XML description. In ebXML the business processes are registered at the repository. The business processes are composed of different collaborating Web Services. This is in contrast to UDDI where the Web Services are registered. The problem with ebXML is that the registration information does not provide any information revealing which business process is a Web Service, and which business process is not. Furthermore, ebXML expects that humans query the registry of ebXML, and evaluate its response.

The *Web Service Inspection Language* (WSIL) allows for discovery of Web Services that are not published in a well-known repository, e.g. UDDI, but are registered at the Web Server that provides the Web Service [231, 232]. A WSIL document contains a description of the Web Service, possibly many links to additional descriptions of the Web Service (i.e. *Web Service Description Language* (WSDL) documents), several links to other publications of the Web Services (e.g. in UDDI), and one link to the described Web Service. By definition, the WSIL document that describes a Web Service must be stored in the root path of the Web Server that hosts the described Web Service. The advantage of WSIL, in comparison to a simple HTML page, is that the WSIL document is formatted in XML and its processing can be done by computers. The disadvantage of WSIL is that developers of clients need to know that there is a WSIL document describing a Web Service and they need to know the location of that document, i.e. the location of the Web Server hosting the document.

Web Services Dynamic Discovery (WS-Discovery) is a true dynamic discovery of Web Services in networks. The design of WS-Discovery assumes a limited spatial extension, e.g. Bluetooth ad-hoc networks [233, 234]. WS-Discovery defines protocols for announcement and discovery of Web Services. For this purpose, Internet Protocol (IP) multicast messages are sent on the IP multicast address 239.255.255.250 (IPv4) and FF02::C (IPv6) on port 3702. Announcements are sent from a Web Service to this address where clients receive them. Clients send requests for a Web Service to this address where Web Services respond to the query. WS-discovery is not designed for an Internet scale discovery, but rather to support discovery of Web Services in ad-hoc networks with a minimum amount of devices. Furthermore, WS-discovery does not define queries, description documents, registration processes, etc. but leaves this to other discovery solutions, e.g. UDDI and WSIL.

The *Advertisement and Discovery of Services* (ADS) provides announcement of Web Services for easier discovery [227, 235]. The objective of ADS is to automatically register Web Services at UDDI and selects the concept of a “crawler”. This crawler will scan through the Internet and register every Web Service it may find in a UDDI repository. A Web Service is detected by storing a well defined Web Service description at a well defined location on the server hosting the Web Service. This file is called *svcsadvt.xml* and has to be stored in the root of a web server. Similar to the HTML page and WSIL, the ADS requires that the Web Servers providing ADS information are known to clients. Furhtermore, clients of Web Services have to know the UDDI repositories where ADS registers found Web Services at.

A solution for discovering Web Services based on ontology matching is the METEOR-S project [236]. The Web Service description is enriched with semantic information, either in the form of a new document, or as an extension to an existing document, like a WSDL document. This semantic description is published in the *METEOR-S Web Services Discovery Infrastructure* (MWSDI) registry where an ontology matching is performed between the descriptions of registered Web

Services and the semantic queries of clients. The advantage, in comparison to the previously presented discovery solutions for Web Services, is that METEOR-S realises discovery of Web Services by comparing semantic information with a relative amount of freedom, and not by fixed descriptions with fixed rules.

The *semantic discovery service* (SDS), proposed in [238], aims to provide seamless integration and interoperation of Web Services based on semantic descriptions and business process modelling. The idea behind SDS is that clients describe their interactions with Web Services in the *Business Process Execution Language for Web Services* (BPEL4WS) and forward this interaction description to the SDS. The SDS then discovers, selects and binds the most suitable Web Services. The description of the interaction is augmented with semantic information expressed in *DARPA Agent Markup Language Semantics* (DAML-S), *DAML Query Language* (DAML-QL), and the *Java Theorem Provider* (JTP). As a result, SDS can perform semantic translations to integrate Web Services that are usually incompatible with each other, i.e. integrating Web Services that require inputs and produce outputs which are of incompatible type but have a similar semantic meaning.

Another approach for semantic based Web Service discovery and integration is *Web Services Modelling Execution Environment* (WSMX) [239-241]. The difference to SDS is that Web Services are executed in a special runtime environment, the WSMX, and that interactions between Web Services are expressed in the *Web Services Modeling Ontology* (WSMO). An implementation of WSMO is done within the Glue project [242].

The Athena project aims at developing a search engine, similar to Google, but for Web Services. Key objective is to create a service discovery that can detect Web Services that are similar to each other, and thus being able to present clients with alternatives for a Web Service the client is interested in [243]. For this purpose, expressive queries, formulated by users, are used to discover the Web Services of choice [244]. In comparison to UDDI, queries are formulated relative to other Web Services. UDDI uses absolute queries, e.g. “give me that Web Service from company XY”, whereas Athena allows for relative queries, e.g. “give me *a* Web Service *similar to the ones* from company XY”.

There exist numerous more service discovery solutions for Web Services, some of them which are described in [225]. Yet, the various existing solutions focus on particular application domains or specific objectives. What is missing in the presented solutions is to form a best-of-breed solution that adapts to different situations and demands. What is needed is a solution that is able to integrate different service discovery solutions and make them accessible for clients via a unified interface. Such approaches are presented in the next sub-sections.

3.1.7 Support for Service Discovery and Interaction

The *Support for Service Discovery and Interaction* (SSDI) project is the first work on integrating different service discovery solutions like UPnP and SLP [12, 13]. In contrast to the above reviewed service discovery solutions, the middleware developed in SSDI does not aim to replace existing service discovery solutions, but to integrate them and combine all their benefits. The selected approach uses a modified form of the structured query language (SQL) to formulate queries for service discovery. The architecture of the middleware from the SSDI project is shown in Figure 3-3. The *client library* is the smallest part of the middleware that clients need to integrate. The client library is used to create queries in the custom SQL and to forward it to the middleware. The *state repository* keeps information about discovered services, the state of services (e.g. running, failed, temporarily unavailable, etc.), and the metadata augmenting the services (like service descriptions). The state repository also unifies the location information of services. This is necessary because different service discovery solutions may return different location information, e.g. JINI returns an IP address, a TCP port number and a unique distributed object identifier, whereas UPnP returns an

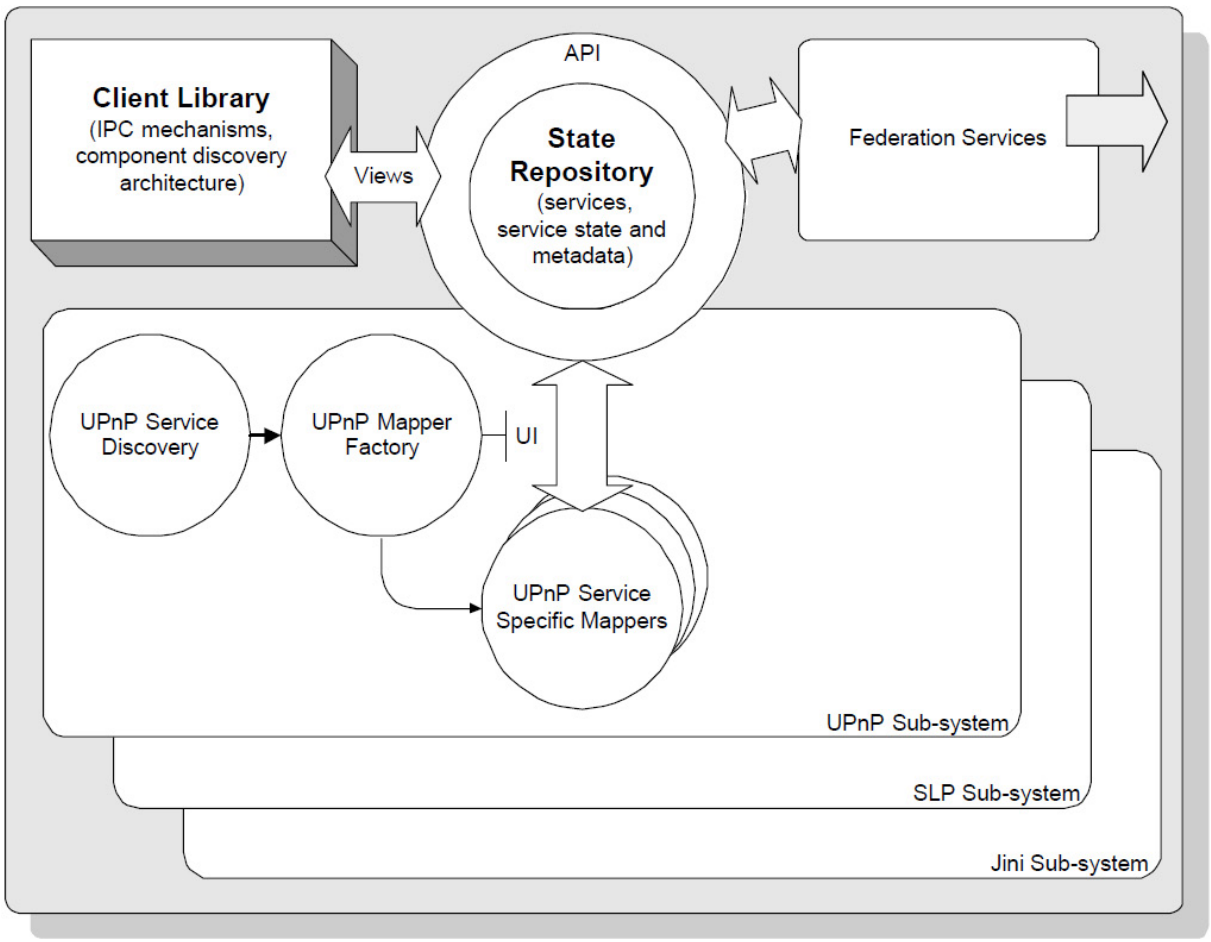

Figure 3-3: Architecture of the middleware of the SSDI project [12]

IP address, a TCP port number and a uniform resource identifier (URI) for the XML document describing the service. The different *sub-system*s are implementations of the different service discovery solutions that are integrated into the middleware. For combining different instances of the middleware running on different computers the *federation services* are used. The federation services facilitate exchange of data of the state repository.
The middleware enables integration of different service discovery solutions to exploit their services via a uniform interface, but has the severe disadvantage that caused existing clients not being able to use this middleware. Existing clients, that use a particular service discovery solution, already have their queries formulated in the required query language, e.g. a set of entries in JINI or an SSDP URI. Using the middleware of the SSDI project with existing clients requires them to rewrite their queries in the modified SQL that is used within the middleware of the SSDI project. Another disadvantage is the limited capability of integrating future service discovery solutions. Integrating service discovery solutions into the middleware requires that their query language to be compatible with the SQL used. Furthermore, the modified SQL may not be able to exploit all features of service discovery solutions, because the modified SQL does not allow addition of new language expressions.

3.1.8 Open Service Discovery Architecture

A recent work in the area of service discovery interoperability and integration is the *Open Service Discovery Architecture* (OSDA) [14]. OSDA provides uniform access to existing and likely future service discovery solutions. This is achieved by the introduction of a middleware that has standard interfaces to query service discovery and receive responses in well-defined formats. The middleware mediates between supported service discovery solutions by translating the client's query, formulated in the query language of OSDA, into the query languages of available service discovery solutions, e.g. JINI or UPnP. Responses are translated from the proprietary format of the

different service discovery solutions to the language format of OSDA by the middleware. This middleware of OSDA can integrate new service discovery solutions dynamically, i.e. a shutdown and restart of the middleware is not required.
Figure 3-4 illustrates the architecture of OSDA. A *domain* in OSDA refers to an administrative domain of one service discovery solution. The middleware of OSDA is organised in the *P2P Cross-*

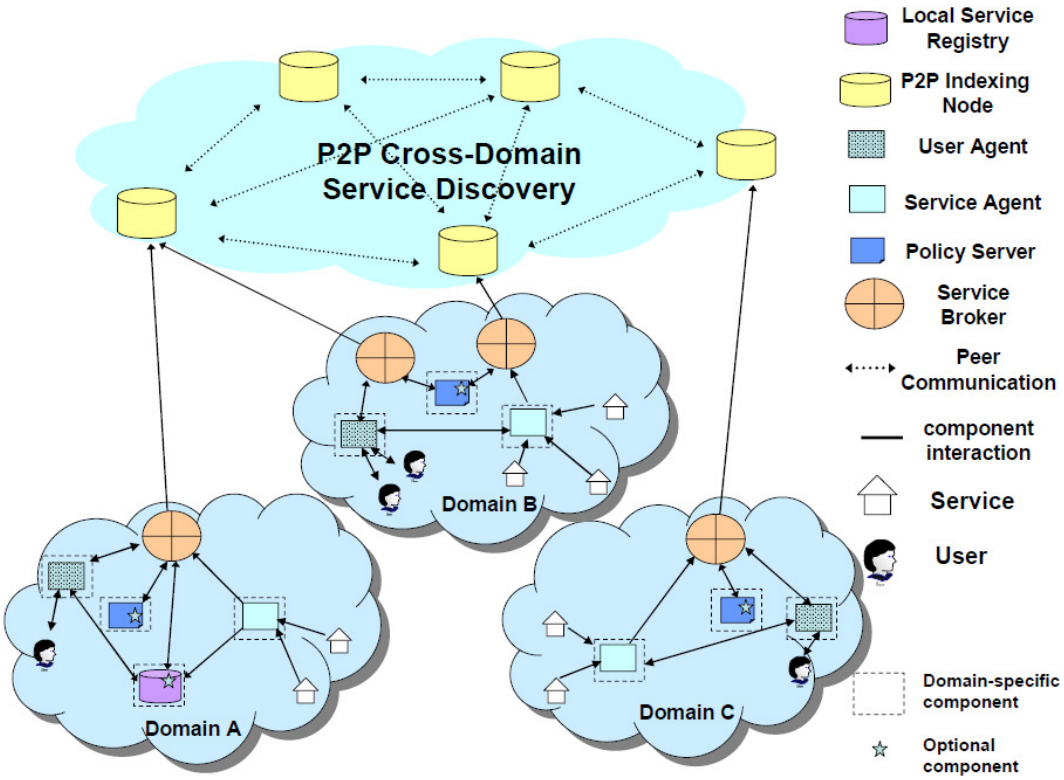

Figure 3-4: The architecture of the Open Service Discovery Architecture [14]

Domain Service Discovery. Clients interact with this *P2P Cross-Domain Service Discovery*. The other architectural elements, e.g. *Policy Server*, *Service Agent*, etc. are additional elements not directly related to OSDA but are introduced for additional control for service discovery.
OSDA integrates existing and future service discovery solutions, but requires clients and service discovery solutions to use its proprietary query language and response format.

3.1.9 Reflective Middleware for Mobile Computing

The *Reflective Middleware for Mobile Computing* (ReMMoC) is a middleware that provides integration of service discovery solutions. Additionally, ReMMoC provides the integration of communication protocols used for service interaction, e.g. TCP, HTTP, and RMI etc. [15, 45]. ReMMoC is composed of two frameworks, the *binding framework* that provides the integration of communication protocols, and the *service discovery framework* that provides the integration of service discovery solutions.

The service discovery framework of ReMMoC is depicted in Figure 3-5. Clients access the service discovery framework via the *IServiceLookup* interface. This interface expects a collection of

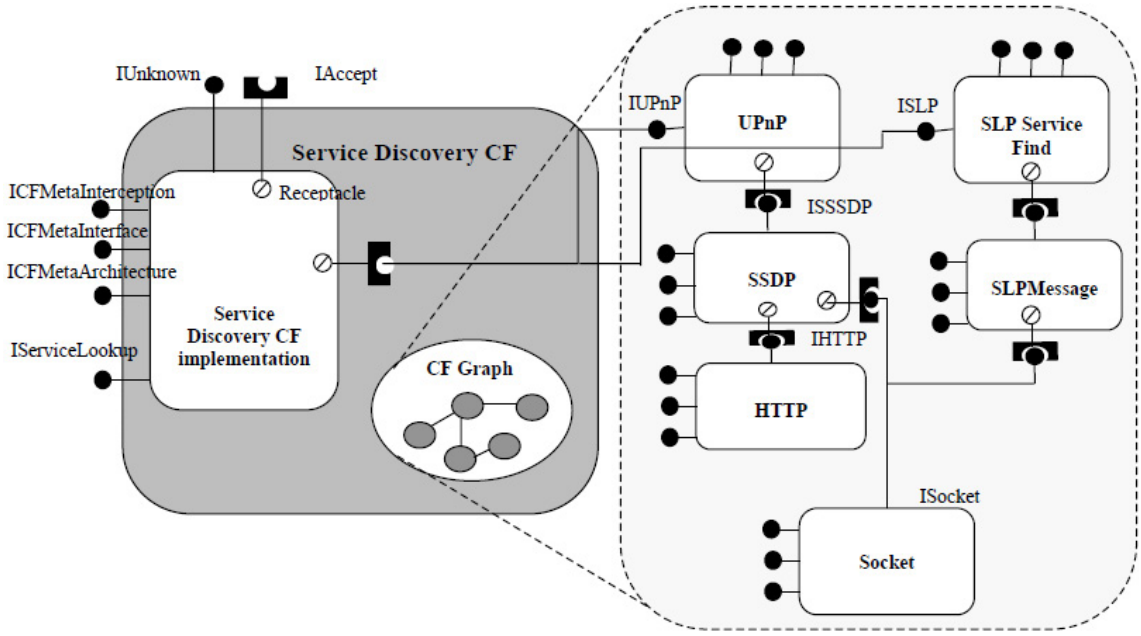

Figure 3-5: The service discovery framework of ReMMoC [245]

attributes that describes the requested service [245]. The interface is implemented in the *DiscoverDiscovery* component [45]. This component forwards queries from the clients to the *Service Lookup Personalities*, which are the implementations of service discovery solutions, e.g. SLP and UPnP. Additionally, ReMMoC exports a dedicated interface per Service Lookup Personality, e.g. *IUPnP*, to enable clients to have a direct access to a particular service discovery solution.

The *DiscoverDiscovery* component uses a *"Cycle and See" philosophy* to query all available *Service Lookup Personalities*.

> *ReMMoC uses a "Cycle and See" philosophy. This entails that the framework execute the discovery of discovery protocols by cycling through a set of tests for each individual protocol it is aware of. The probability of services being found increases as the number of tests to cycle through increases. "Cycle and See" does not rely on agreement between participating elements, and is evolvable to include future discovery mechanisms. – cited from [45]*

Yet, the "cycle and see" philosophy in ReMMoC has two shortcomings.

> *However, the "Cycle and See" approach is limited in two respects: 1) cycling through discovery protocol tests is both time and resource consuming, and 2) as the number of tests increase the performance of the platform degrades. – cited from [15]*

Another disadvantage of ReMMoC is the need to update the middleware whenever a new service discovery solution is introduced.

> *We have implemented the service discovery framework with two service lookup protocol implementations: SLP and UPnP, allowing us to demonstrate how to overcome the problems of the availability of multiple service discovery protocols. However, as with the binding framework, it is feasible for new discovery protocols to be dynamically integrated into the framework at a later date. This requires a new version of the DiscoverDiscoveryProtocol component, which can detect the new protocol, to be plugged into the framework. – cited from [246]*

3.1.10 Summary

As could be seen from the review of existing service discovery solutions, there are numerous stand-alone solutions. These solutions may coexist but do not integrate. Existing solutions that integrate

service discovery solutions are either not able to do so at their runtime, or they require the mapping of query languages and response formats to a proprietary format.
To provide maximum benefit to clients of service discovery, an integrating solution should be based on the following requirements:

- Stable: Clients shall be able to keep their queries they have already formulated. There should be no need to define a new query language, or extensions to query languages. This makes the solution compatible for legacy clients, i.e. already existing clients.
- Purposeful: Clients querying about a service expect a reference[4] for immediate use. Clients do not want to receive abstract information that they need to first resolve in order to get the reference to a service. In a comparative example, when someone asks "I need a screwdriver", it is unlikely that he would be happy with the answer "I have one in my toolbox", but instead he expects the screwdriver to be handed out. On the other hand, clients have the interest to receive additional information about the service. This enables them to have a final decision whether the services returned by the service discovery match their expectations.
- Self-similar: Service discovery solutions shall be implemented as services. This allows discovery of a service discovery service via service discovery.
- Flexible: While an integrating approach should enable clients to interact with service discovery in a uniform way, it should also enable clients to interact with service discovery services directly, using proprietary interfaces.
- Open: An integrating approach has to support future service discovery solutions. The integration of future service discovery solutions must not bring a negative impact on existing services, service discoveries and their clients.
- Continuous operation: The integration of new service discovery solutions must not require a restart of the middleware or clients. Their integration and removal should take place seamlessly.

The following Table 3-1 compares the different existing approaches for integration of service discovery with the above requirements. From the table it can be clearly seen that none of the existing approaches to integrate the different service discovery solutions complies with all of the requirements. Therefore, a new approach for integrating service discovery solutions is presented in the next section.

Table 3-1: Comparison of approaches integrating service discovery solutions

	Stable	**Purposeful**	**Self-similar**	**Flexible**	**Open**	**Continuous operation**
WSDA	Yes	No	No	Yes	Yes	Yes
SSDI	Yes	No	No	No	Yes	No
OSDA	No	No	No	No	No	Unknown
ReMMoC	Yes	Yes	Yes	Yes	Yes	No

3.2 FAME²: The open service discovery interface

As seen in Table 3-1, there is a need for an approach of integrating service discovery solutions that complies to the different requirements listed above. The open service discovery interface (OSDI) provides interoperation and integration of service discovery solutions. The idea of OSDI is comparable to meta-search engines for the World Wide Web, e.g. MetaGer, MetaCrawler, or Excite.

[4] A reference points to, or is a link to a software element, e.g. a service.

Figure 3-6 illustrates the idea of the open service discovery interface. In a distributed computing environment different services are provided, e.g. a projector service for displaying presentations to a

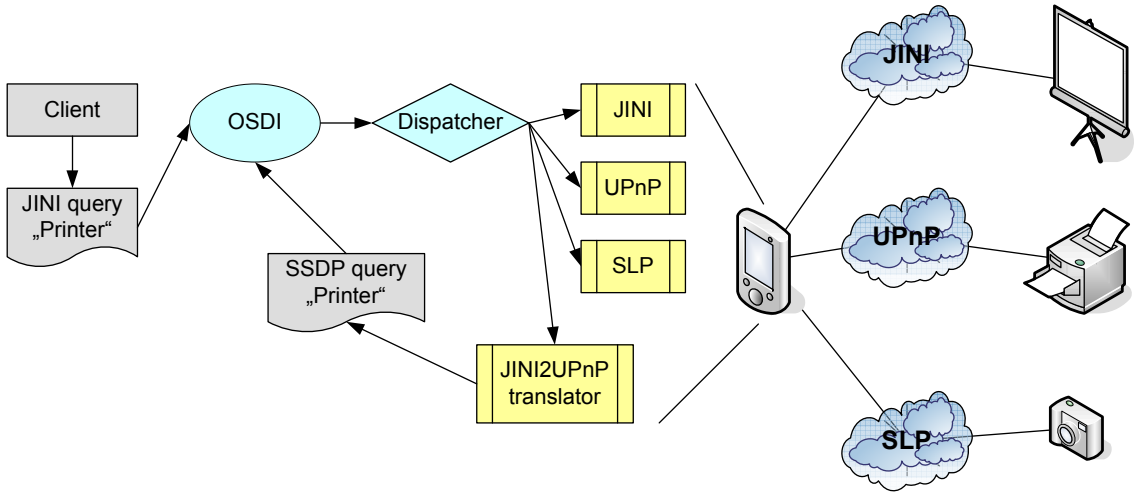

Figure 3-6: The idea behind the open service discovery interface

larger audience, a printer service to print handouts, and a camera service to record parts of the presentation. The different services are discoverable via different service discovery services, e.g. JINI for discovery of the projector, UPnP for discovery of the printer, and SLP for discovery of the camera. Additionally, there are services that behave like a service discovery service, but they are actually capable to translate a query from one query language into another one, e.g. from JINI to UPnP. Assuming that there is a presenter who uses a handheld computer, e.g. a personal digital assistant (PDA) or a smart phone, to do his presentation and to control the services he needs, i.e. the projector service and the printer service. His presentation software needs to discover the required services. The presentation software on the handheld computer is programmed to use JINI as the service discovery solution. The task of the OSDI is to abstract the different service discovery solutions and to provide the presentation software a uniform access to them. Integrating translators that translate queries from one query language into another language enables clients to exploit different service discovery solutions with a single query. In Figure 3-6 the client queries for a printer using a query formulated in the query language of JINI. However, the printer is discoverable via UPnP but not via JINI. But as the *JINI2UPnP translator* service translates the query into a UPnP compatible query, the client can then discover the printer even though with a JINI query. The client would not be aware of the translation of the query.

3.2.1 Architecture

The Open Service Discovery Interface uses an event dispatching system. The architecture of the OSDI is illustrated in Figure 3-7 on the next page. A client formulates a query in any query language, e.g. JINI, SLP, or SSDI's SQL dialect. Being able to formulate the query in any query language complies with the requirement of **stability**. The query is passed by the client to the open service discovery interface. The implementation of the open service discovery interface creates a *service discovery* event that carries the query of the client and is ready to store the references of services found by service discovery. This event is passed to an event dispatcher, which places the event on an event channel. Service discovery services, query translator services, and response filter services, listen to this service discovery event. Service discovery services check if the query stored in the event is compatible with their query language, and start to lookup services matching the query if the query is compatible. Query translator services translate the query stored in the event into another query language. The translation overwrites the already stored query. Based on policies, the response filter services analyse the references of services stored in the event and can decide to remove or manipulate references.

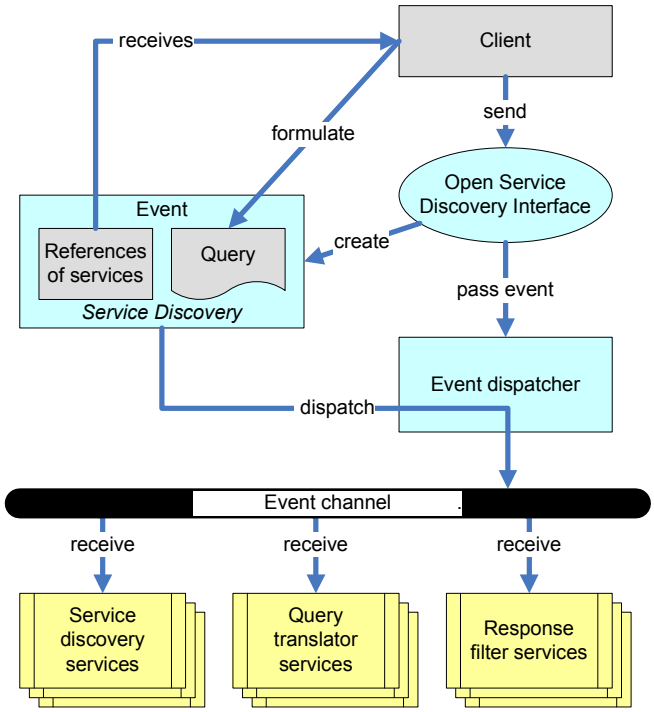

Figure 3-7: Architecture of the open service discovery interface

3.2.2 Design

Figure 3-8 shows the Unified Modeling Language (UML) class diagram of the OSDI. For better overview and easier understanding the diagram and the following explanations and examples omit the security principal. Clients formulate their queries and pass them to the *lookup(Query)* operation defined in the *IServiceDiscovery* interface. As a result, each receives a collection of *IServiceInfo* instances, one per found service. The *IServiceInfo* interface has an operation *getService()* that returns the reference of the found service. Service discovery services may enrich the *IServiceInfo* interface with additional information, e.g. the URL of an XML document that describes a service published in a UPnP network. This flexibility introduced with *IServiceInfo* meets the requirement of being **purposeful**. The *CDiscoveryEvent* is the event created by the OSDI implementation and then placed on the event channel by the implementation of *IEventChannel*. The different service discovery services, e.g. *ILocalDiscovery*, *IJINI*, *IRMI*, *IUPnP* and *ISLP*, and the query translator services, e.g. *IJINI2UPnP* and *ILocal2RMI*, are derived from the *IService* interface. As a result, service discovery services are discoverable via service discovery, complying with the requirement of **self-similarity**. Additionally, it meets the expected **flexibility** because clients are able to retrieve a particular service discovery service and then interact with it directly via proprietary application programming interfaces (APIs).

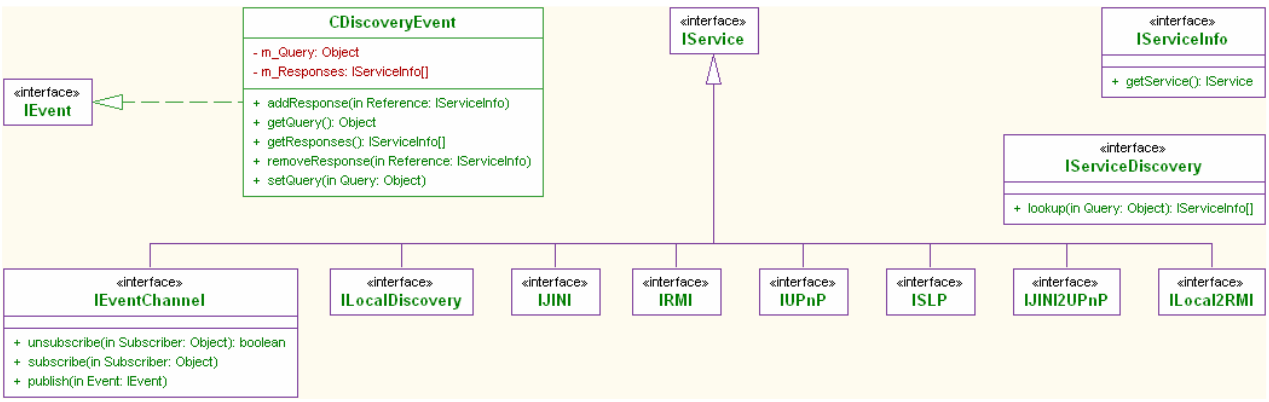

Figure 3-8: UML class diagram of the open service discovery interface

3.2.3 Implementation of service discovery services

The key aim of the OSDI is the easy and simple integration of new service discovery services without having a negative impact on existing services and clients. This section describes the actions required to develop and integrate a new service discovery service. The steps described in this section also apply to the integration of query translation services and response filter services.
Implementing a service discovery service is similar to the implementation of a FAME² service. First, the API of the service discovery service is defined. Second, this definition is derived from *IService*. An example with a service discovery service for the RMI service discovery is given below:

Program 3-1: Application programming interface definition of the RMI service discovery service

```
1   public interface IRMI extends IService {
2     public void bind(String Name, java.rmi.Remote Service) throws Throwable;
3     public String[] list() throws Throwable;
4     public java.rmi.Remote lookup(String Query) throws Throwable;
5     public void rebind(String Name, java.rmi.Remote Service) throws Throwable;
6     public void unbind(String Name) throws Throwable;
7   }
```

The interface is identical to the API of the Java RMI registry. Therefore, client developers do not need to learn a new API.
The next step is the implementation of the RMI service discovery service. The implementation of the service is shown below:

Program 3-2: Implementation of the RMI service discovery service

```
1   public class CRMI extends AService implements IRMI {
2     private java.rmi.registry.Registry m_Registry;
3     public void start(ICore Core, java.util.Map Parameters) throws Throwable {
4      m_Registry = java.rmi.registry.LocateRegistry.getRegistry();
5      Core.getEventChannel().register(p_This, CDiscoveryEvent.class);
6     }
7     public void stop(ICore Core, java.util.Map Parameters) throws Throwable {
8      throw new UnsupportedOperationException("Java RMI does not allow to be stopped.");
9     }
10    public void bind(String Name, java.rmi.Remote Service) throws Throwable {
11     m_Registry.bind(Name, Service);
12    }
13    public String[] list() throws Throwable { return m_Registry.list(); }
14    public java.rmi.Remote lookup(String Query) throws Throwable {
15     return m_Registry.lookup(Query);
16    }
17    public void rebind(String Name, java.rmi.Remote Service) throws Throwable {
18     m_Registry.rebind(Name, Service);
19    }
20    public void unbind(String Name) throws Throwable { m_Registry.unbind(Name); }
21    public void processEvent(IEvent Event) {
22     if (Event instanceof CDiscoveryEvent) {
23      CDiscoveryEvent ev = (CDiscoveryEvent)Event;
24      ev.addResponse(new CServiceInfo(m_Registry.lookup(ev.getQuery())));
25     }
26    }
27  }
```

In the lines 3 to 9 the service life cycle operations are implemented. As the Java RMI registry cannot be stopped, the RMI service discovery service cannot be stopped. Lines 10 to 20 are the implementation of the proprietary API of the RMI service discovery service that clients may use when requiring access to the full features of the RMI service discovery solution. The discovery for the OSDI is implemented in the lines 21 to 26.
There are no additional steps required for an implementation of a service discovery service. The example above demonstrates the **openness** of the OSDI, i.e. the ability to integrate new service discovery solutions without impacting existing service discovery services or clients. **Continuous operation** is already realised by FAME².

The **simple** use of the OSDI is achieved by the simple API, the *lookup(Query)* operation, based on the request/response paradigm. Clients formulate their query, pass it to the OSDI (i.e. request), and await the responses. **Simplicity** for service developers is achieved by the close similarity between the development of a *usual* FAME² service, and a service discovery service, query translator service or response filter service.

3.3 Evaluation

In comparison to the middleware of the *Support for Service Discovery and Interaction* project described in [12, 13], there is no need to define a common query language in the OSDI. The Open Service Discovery Interface enables clients to use different service discovery solutions in a uniform manner. The clients formulate their query about a service in any query language they like. As a result, the OSDI supports exploiting the full features of available query languages of service discovery solutions, and clients are not restricted to a proprietary query language that may provide some features only. Furthermore, the only modification that existing clients need to do is to redirect their query from the service discovery solution they currently use to the OSDI. The OSDI probes the available service discovery services, is able to translate queries into other query languages, and can filter the responses from service discovery to ease service selection for a client in the case where there are multiple services that match the query. In contrast to SSDI middleware where federation of service discovery requires an extra *federation service*, OSDI achieves federation of service discovery by implementing service discovery solutions as services by themselves. This allows the discovery of service discovery services via the same mechanisms as when looking for other services.

The Open Service Discovery Architecture is based on the ideas of SSDI middleware. In OSDA, a new query language and a new response format are defined. As a consequence, clients have to reformulate their queries and need to process the OSDA response format. Integration of new service discovery solutions requires that they are compatible with OSDA query language and response format. In comparison, OSDI does not require the query language of a service discovery solution to be compatible to any other query language. The only requirement is that the service discovery solutions need to be able to provide a reference to the discovered services, or at least enough information to create such reference.

Similar restrictions, which apply to SSDI middleware and the OSDA, also apply to the Reflective Middleware for Mobile Computing. Furthermore, while the OSDI enables seamless integration of new service discovery solutions, i.e. without updating a middleware implementing OSDI, ReMMoC requires such an update in order to be able to discover new service discovery solutions.

Concluding, the OSDI is a flexible solution for service discovery interoperability and integration that does not restrict clients to use a *standard* query language. It is able to directly return service references to clients, and that enables dynamic integration of new service discovery solutions as they become available, without any update to clients, other service discovery services, and middleware implementing OSDI. The OSDI complies with all the different requirements an approach of integrating service discovery solutions should comply with.

4 Online-update of services

The work in [247] mentions three reasons for update of software. The first reason is to correct specification, design, coding, and documentation faults. This reason is called *corrective maintenance*, and is more well-known in common speech under the terms *bug-fixing* or *patching*. The second reason is to improve performance and effectiveness of software, i.e. optimisations. Additionally, enhanced features for the software may be integrated. This second reason is called *perfective maintenance*. The third reason is to react to changes of the platforms where the software is executed on. As new hardware and software becomes available, there are times where the already deployed software needs to be adjusted to support them. This third reason is called *adaptive maintenance*.

Usually, software update requires stopping the software, updating it, and then restarting it. Software with high dependability, e.g. business transaction, telephone switching, emergency response, and highly distributed systems, are not supposed to be stopped and to be restarted for update purposes. Instead, such software is expected to continue operation. In order to still be able to update such software, online-update functionality is required. Online-update performs software maintenance without the need to stop and to restart the software. Instead, parts of the software are exchangeable while they might be in use.

The concept of service oriented architecture (SOA) aids in realising software that is online-updateable. Loosely coupled services enable mechanisms of redirection to updated services. However, this does not cover already existing interactions between services and their clients, but only affects new interactions. The major reason is that the concept of SOA requires late binding. Late binding supports the change of control flow of software at runtime [100], i.e. binding services to their clients at runtime. Examples for realisation of late binding are function pointers in the C and C++ programming language [101], dynamic class loading in Java [102, 103] and far jumps in assembler. The alternative to late binding is early binding. In early binding the compiler and linker binds together all resources to one piece of software before the software is executed. With early binding the control flow of software is fixed before its execution.

Optimally, online-updates do not interfere with ongoing interactions, require no additional communication between clients, services, and the platforms running them, and do not require special attention of humans. While there are numerous proposals for online-update, each of them imposes problems that limit their usability.

The authors of [248] have given a comparison of online-update solutions, including memory manipulation proposed in [249, 250], the proxy structural design pattern proposed in [17-19], and platform manipulation as proposed by the authors of [251, 252]. The conclusion of the authors of [248] is that none of the existing update solutions provide a convenient solution. Some of them are suspicious to the exposure of the self-reference (i.e. proxy pattern). Others require adaptation of existing software. Yet, others require direct access to memory and low level functionality of computers (i.e. memory manipulation) or create a proprietary and highly specialised run time environment (i.e. platform manipulation).

For online-update, there are six different strategies.

- **Replication:** The authors of [253] identified online-update capability as a mandatory requirement in Internet applications. They solved the problem by doubling the execution block: one that is actually processing and one that is used for standby. The software can switch between the execution blocks. This allows update of the software by updating the unused execution block first, then switching over and finally updating the other execution block. The limitation of their approach is that the code size is doubled on disc and in memory, and application developers need to implement the logic to switch between the execution blocks. Yet, meanwhile standard software patterns and middleware existing to realize this switching between execution blocks. The advantage of their solution is that it imposes minimal overhead, and optimally there is no overhead.

- **Memory manipulation:** In [250] low level memory manipulations inside of the operating system kernel are used for online-updates. Such approach is highly efficient, and allows for maximum flexibility while providing transparency of update to the software running on the operating system. However, this approach is highly proprietary and requires the redesign of operating systems. In [249] an update mechanism based on direct memory manipulation is presented. An update package contains the updated version of the software for the purpose of update, as well as a software that takes care of any contingencies that might happen, e.g. that the software to update is currently used. The approach described in [249] enables changing types and signature of software, i.e. the interface of services by using explicit casts and overloaded versions of the same operation in the software patch package. The problem of the solution presented in [249] is that it requires direct memory manipulation, which will partially bypass protection mechanisms of modern operating systems (e.g. Linux, Windows 2k or XP) and hardware (e.g. NX-protection of AMD and Intel CPUs), and security platforms like Paladium (for Trusted Computing).
- **Platform manipulation:** A Java Virtual Machine (JVM), modified to support replacement of parts of software (i.e. objects) while it is running is described in [251]. The authors argued that the behaviour of an object is bound to its class (a blueprint for an object), and the replacement of classes affects the creation of new objects only. However, they admitted that the coexistence of objects of the same type but of different versions (i.e. created from different versions of the same class) leads to confusion and may cause abnormal software behaviour, like inconsistencies and crashes. Nevertheless, the major drawback of their solution is the requirement of a modified JVM, which would require all computers to be first updated in order to support online-update. The authors of [252] enable online-update by using a dynamic class loader for the JVM. The class loader requires special interfaces to the JVM that were implemented in the Sun JVM version 1.2 for the Solaris operating system. The dynamic class loader supports reloading and replacing of classes, and is able to modify the behaviour of existing object instances by replacing their classes. The class contains the implementation of algorithms that are executed when a method in the object is invoked, so the replacement of a class updates all its objects. The modifications add a static overhead of 2.2% to the execution time of Java software.
- **Proxy pattern:** In [17], it is shown how to apply the proxy structural design pattern to allow online-update of classes in C++. The authors argued that the proxy pattern is the most effective solution because it does not require any change of already existing software. The proxy pattern wraps code around a software element. This code deals with update without affecting the proxied software element or its clients. The idea of adding new code to running software dates back to the earliest electronic computers [254]. The proxy pattern enables update without sacrificing type safety and performance. In [18] it is motivated that an online-update is a requirement for distributed applications because their shutdown might not be an option. The presented solution, based on the Java programming language, uses the proxy structural design pattern to wrap any object. Upon update, the proxy remains while the wrapped object is updated. Objects have references to the proxy only. The advantage of this solution is that there is no need to modify existing code and the JVM. The authors pointed a problem when using the self-reference operator as this bypasses the proxy. The only possibility to ensure that the proxy pattern is not bypassed is to modify the JVM. The measurements of their solution yielded that wrapping an object with a proxy adds a maximum of 3,36%, a minimum of 0,08% and an average of 1,13% to the execution time, and a maximum of 18,92%, a minimum of 3,98%, and an average of 6,36% to the memory consumption. Using the bridge pattern, a variant of the proxy pattern, that enables online-update of OSGi services, is described [19]. However, this work does not discuss the questions of the proxy pattern when the self-reference operator is used.
- **Interaction modelling:** The author of [255] described an online-update mechanism for the Java programming language, which is based on three ideas: indirection by using interfaces instead of classes; late binding; and factories that create objects. The concepts and language elements in Java do not allow for online-update.

> *A Java programmer can gain more power at run-time by using their own classloader, although they cannot force a class to be unloaded. However, using a classloader can lead to other problems as Java defines runtime type equivalence in terms of the fully-qualified name of the class plus the identity of the classloader that loaded the class. – cited from [255]*

His solution for online-update, a system called *Grumps*, is a component model where the single software element, i.e. an object of a class, is realised in a container. The interaction between the containers is based on the event oriented coordination model. If an object needs to be updated, the events are simply rerouted to the new version of that object, i.e. to its container. The Grumps solution requires that coordination between objects is based on events. This may be, however, not useful, and is not compatible with already existing code that uses method calls. Method calls are implementations of the direct coordination model, and of the intrinsic communication form in object oriented programming languages like Java.

A similar approach called weaves is proposed in [256, 257]. A weave is a set of components and connections. The weaves are visually modelled, i.e. the components are shown and developers draw connections between them. When updating a component, the developer erases the connections to this component, and draw new connections to the updated component.

The work presented in [258] uses interaction modelling and defines operations to ensure that no communication between components is ongoing. This is achieved by interception and redirection.

The authors of [259] described an online-update solution based on the runtime-change of architectures. Connectors combine components. The former realises any kind of communication, whereas the latter realises the computation. A connector connects two or more components. Their idea is to re-establish connectors between the components and to isolate old versions of components. Once a component has no more connectors, it can be removed. Instead of isolating communication to components on the implementation level (as it is done in [255]), they did it on the architecture level.

- **Dynamic interfaces:** The technology of dynamic interfaces aims to make the requirement for updates obsolete. Instead of programming software that invokes operations, software just passes arguments and let the platform decide which operation is best suitable for the given arguments. The Ninf is a global world-wide computing infrastructure that uses dynamic interfaces to exploit available computational power in a grid computer system [260]. A more general solution to dynamic interfaces for software based on the concept of service oriented architecture is presented in [261]. Part of this solution is an editor, where users and administrators can map the invocation of an operation programmed into software to the operations available in other software. Administrators define parameter mappings, i.e. assign default values for parameters that the invoking software does not provide, select to ignore parameters that the invoked software does not expect, or change the types of parameters, e.g. from integer number to floating point numbers. The *SOFtware Appliances/Dynamic Component UPdatating* (SOFA/DCUP) architecture combines the concepts of dynamic interface and proxy pattern [262]. SOFA/DCUP is based on a static part around a service with a static interface. Software is programmed to use this static interface. The static part contains a dynamic part of the service, where the interface of the dynamic part may change from a version to another version. The static part contains logic to combine the static interface with the dynamic interface of the dynamic part of the service. The author of [263] pointed out that frameworks would greatly benefit from dynamic interfaces but identified the ambiguity as the major problem for automatic mapping of dynamic interfaces.

The replication strategy doubles the required resources, physical disc storage capacity and memory consumption, so it is considered as unsuitable for ubiquitous computing. Memory manipulation requires low-level access, possibly bypassing the operating system. As new versions of operating systems will have strict rules on who is allowed to have low-level access to the memory, this solution is no longer applicable to modern computing systems. As a consequence, memory manipulation is not a future-oriented strategy. The platform manipulation strategy demands a

redeployment of an altered platform, e.g. Java Virtual Machine, onto all computers in a distributed computing systems used for ubiquitous computing. Since such system may consist of thousands of computers this strategy is expected to be impractical. The interaction modelling is a user-centred strategy that requires the users to perform rerouting (rewiring, reweaving). Yet, in the vision of ubiquitous computing, users are supported in an unobtrusive manner, resulting in a conflict with the interaction modelling strategy. Dynamic interfaces have demonstrated that they can make update mechanisms obsolete, but require human interaction to resolve ambiguities. What remains is the proxy pattern, which does not require low-level access to system resources, e.g. memory, does not consume double resources, does not need a redeployment of platforms, and is implemented automatically without requiring user intervention. However, the proxy pattern is vulnerable when the self-reference operator is used, i.e. a service exposes a reference to itself. Then the proxy around a service is bypassed, resulting in a failure of future update attempts [18, 264].
The proxy pattern can be applied to deployed services [265, 266]. The decoration of a service with a proxy does not modify the existing service but merely wraps it around. Ideally, the decorated service and the clients of the service are not aware of the existence of the proxy. For this purpose, the proxy is an authorised representative of the wrapped service, responsible for intercepting all communication to that service.
The communication mechanism used for communication between loosely coupled services, e.g. FAME² services, needs to support flexible communication links. The publish/subscribe coordination model is such a communication mechanism [20, 21]. Software based on publish/subscribe reacts to the occurrence of events. An event is defined to be an instantaneous, atomic (happens completely or not at all) occurrence of interest at a point in time [267]. In a publish/subscribe architecture, the subscriber, which may be a service, registers for a certain event of interest and will be notified when that event of interest occurs [20]. Registration is done by passing the reference of the subscriber to the publisher. The reference is a unique, immutable and direct link to the subscriber. The publisher, which may be another service, is responsible for notifying all subscribers by sending the event to all end-points of their references.
The combination of the two abovementioned concepts results in an incompatibility problem. In a publish/subscribe architecture the subscriber exposes its reference to the publisher during registration. If the subscriber is wrapped by a proxy that should intercept all communications, including receiving events, then the subscriber should not pass its own reference but the reference of the proxy instead. Yet, as the wrapped service should not be aware of the proxy, the proxy is bypassed by the publish/subscribe communication mechanism. The reasons are that:

- the subscriber does not know about its extension by the proxy and thus does not update its registration information (the reference) at the publisher
- the publisher does not know about the transparent extension of the subscriber by a proxy and thus has no reason to request a new reference of the subscriber
- the proxy does not know the publishers that the extended subscriber has subscribed to. As a consequence, the proxy cannot intercept direct communication between the subscriber and the publishers

This situation, where the publisher sends its events to the subscriber and hence bypassing the proxy, violates the purpose of transparently wrapping the subscriber and intercepting all communication to the subscriber. The proxy does know about the reference of the subscriber, but because references are immutable and the proxy does not know where the subscriber did subscribe, the proxy cannot update the reference kept by the publishers of events to refresh the link to the proxy instead of linking to the subscriber. Solutions to this problem range from avoid using the reference of objects (Liebermann [265] and Hölzle [21]) to manually engineered static proxies (Welch and Stroud [266] and Renaud and Evans [264]). Static proxies are proxies designed by the human for a particular service and they are aware of the use of the self-reference operator of their proxied object to intercept the use. But static proxies require human interventions, which is opposing the idea of ubiquitous computing.

The remainder of this chapter is structured as following: the next two sections describe details of the proxy pattern and publish/subscribe. Section 4.3 introduces the concept of mutable reference endpoint to harmonise the proxy pattern with publish/subscribe. Furthermore, this concept allows for wrapping the service with as many proxies as required, including the wrapping with proxies while it is in use and communicates with events. The implementation of the mutable reference endpoint, which is part of the FAME² update manager, is described in Section 4.4. The chapter closes with a summary.

4.1 The proxy structural design pattern

The proxy structural design pattern [16] is a structural design pattern to transparently wrap around software, e.g. a service. The process of creating a proxy around a service is called service decoration. Thus, the proxy decorates the service, or the service is decorated by a proxy. The proxy is the authorised representative of the decorated service, i.e. the proxy is the interaction point between the decorated service and its clients. The objective is to intercept all client initiated communication to the decorated service. In object oriented programming, this communication is the invocation of operations of the service by the clients. The objective of transparent wrapping with the proxy pattern is to ensure that neither the decorated service nor its clients are aware of the proxy.

Using the proxy pattern, features such as logging for audit and monitoring purposes, billing to charge for service use, security to control service access, persistence to pertain the service state, transaction control for defined workflow, etc., can be wrapped around a service. The situation of the service determines the features required. Thus, in order to preserve resources it does not make sense to have the functionalities above being an integral part of services. Instead, flexible service composition can be applied to add those functionalities on demand and on the fly. Transparent extension is feasible, because the functionality of the service does not need to change to have the features above extended.

A client accesses a service through a reference of that service (see Figure 4-1). A reference is a link to software, e.g. a service, a client, or a proxy. The reference uniquely addresses the software by the

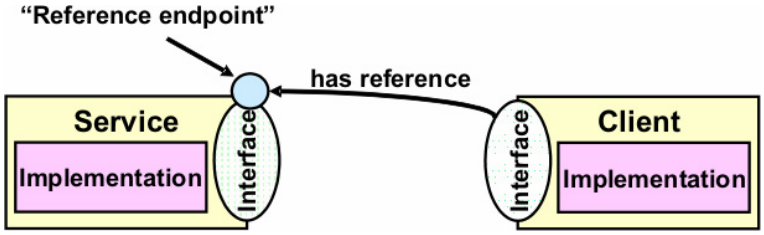

Figure 4-1: A client accesses a service through the reference of the service

software's reference endpoint. By definition, references are immutable and unique. In other words, the reference pointing to a reference endpoint cannot be changed to point to another reference endpoint. Also, the reference endpoint is inseparable bound to its software. Furthermore, a reference may not be divided, i.e. point to different software at the same time. In the remainder of this chapter, this reference type is also called traditional reference.

When creating a proxy around a service, the references of the proxy and the decorated service are

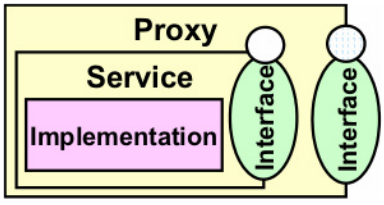

Figure 4-2: Decorating a service with a proxy

different, because references are unique. Both references point to different software elements (see

Figure 4-2). It is not possible to change the reference of the service to point to the proxy because references are immutable.
Thus, the resulting figure of a client using a decorated service looks like in Figure 4-3.
The immutability of references is still an unsolved problem if the proxy around a service is created

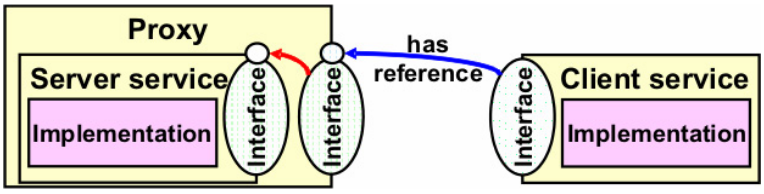

Figure 4-3: A client access a service through a proxy around the service

after clients have obtained the reference to the decorated service. For a transparent extension with a proxy neither the decorated service nor the clients of the service should be aware of the proxy's existence. Consequently, there is no reason for the service and its clients to expose information about each other, i.e. which clients use the decorated service and which services are used by a client. Furthermore, the clients have no reason to support changing the reference from their used service to the new proxy around the service.

4.2 Publish/subscribe

The communication mechanism used between loosely coupled services needs to support flexible communication links. Publish/subscribe is such a communication mechanism [20, 21]. It follows the message or event oriented coordination model.
In publish/subscribe, the subscriber registers with publishers to receive published events. The registration is done by passing the reference of the subscriber to the publisher. The publisher sends

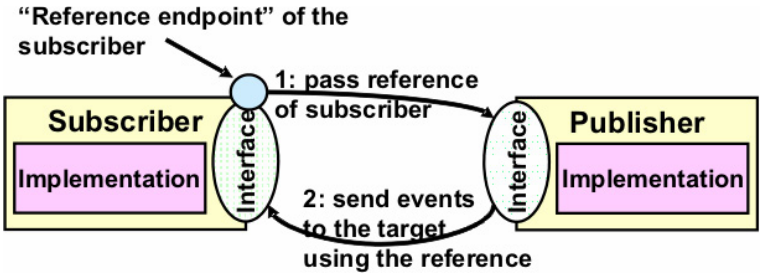

Figure 4-4: Operation of publish/subscribe

events to the subscriber by using the subscriber's reference. The major advantage of publish/subscribe is that publishers do not need to address the subscribers. The process of registration and event-sending is shown in Figure 4-4.

4.3 The concept of mutable reference endpoint

A problem occurs if a transparent extension of the subscriber using the proxy pattern is implemented after the subscriber passed its reference to the publisher. As explained in the previous section, references are immutable and unique. Subscribers have no obligation to update their references in their subscribed publishers, and publishers do not keep track with the changes or validity of their subscribers' references. A subscriber registers at a publisher using its reference and the publisher sends events to the subscriber using the reference. However, if the subscriber is extended with a proxy after a successful registration, the publisher still has the reference of the subscriber and not of the proxy. In this case, the events from the publisher are directly sent to the subscriber, and the proxy is bypassed. As a result, the intention of transparently wrapping the subscriber with a proxy will fail because there are possibilities that clients (publishers) will directly communicate with the proxied service (subscriber), and will bypass the proxy.

To solve the outlined problem, caused by the use of unique immutable references in the implementation of publish/subscribe and the proxy pattern together, the concept of mutable reference endpoint is introduced. Mutable references, which are the references to mutable reference endpoints, are unique, and they point to or link to software. However, contrary to traditional references they are mutable, which means that the reference may change the software it points to. Every software, e.g. service, client, proxy, etc., has its own unique mutable reference.
The mutable reference is a substitute for immutable references. To use the mutable references effectively, the usage of immutable references must be avoided. Both reference types, mutable and immutable, cannot coexist.
If necessary, mutable references can change their references. This is necessary when one intends to use the publish/subscribe communication mechanism before applying the proxy pattern to the subscriber. If a subscriber is transparently decorated after a subscription at a publisher, the mutable reference of the subscriber allows the proxy to redirect the reference to it, ensuring that published events are intercepted by the proxy, and not received directly by the subscriber.
Two additional operations for each object type are proposed to introduce mutable references into object oriented programming languages. The first operation is the *get* method that retrieves the

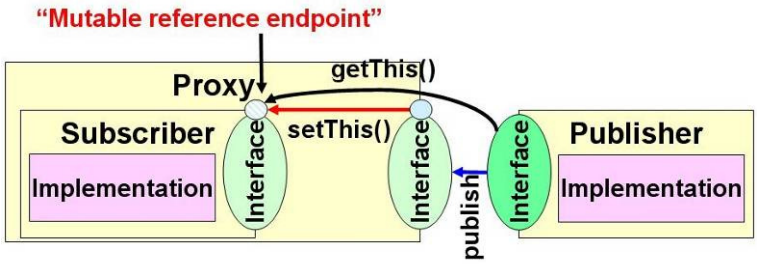

Figure 4-5: The concept of mutable reference endpoint

mutable reference. In Java or C++ this operation would be called *getThis()*. The other operation is the *set* method that updates the mutable reference endpoint, so that the mutable reference will point to another object. This operation is called *setThis(…)*. Figure 4-5 shows the concept of mutable references. In this illustration a publisher uses the *getThis()* operation to obtain the mutable reference of a subscriber. The proxy will use *setThis(…)* to update the mutable reference endpoint. This will provide publishers with the up-to-date reference of the decorated subscriber.
Whenever the reference to an object, like a subscriber, should be stored for some purposes, like sending events, the get method for returning the mutable reference would be used instead of the built-in operations of object oriented programming languages.
With the use of mutable reference endpoints and mutable reference, the proxy pattern can be applied for online-update of services while the services are allowed to be subscribers in a publish/subscribe communication.
Using mutable references has two limitations: (1) Using mutable references, the reference of a decorated service is updated. When a service is decorated with more than one proxy at the same level, i.e. wrapping a proxy around a service and the wrapping a second proxy around the same service, and not the proxy around the service, would cause a collision. There is no way to tell which proxy is the real representative of the decorated service. However, a proxy may be decorated by another proxy without causing collisions. (2) When mutable and immutable references are used at the same time, the whole concept of mutable references becomes obsolete and the problems discussed above, i.e. bypassing the proxy, remain. Once a service exposes its immutable reference and this immutable reference is used to register at any publisher, any proxy around this service would be bypassed.

4.4 Implementation of the mutable reference endpoint in FAME²

The FAME² update manager (UM) realises online-update of FAME² services even when these services may communicate using events, i.e. publish/subscribe. The UM adds a mutable reference

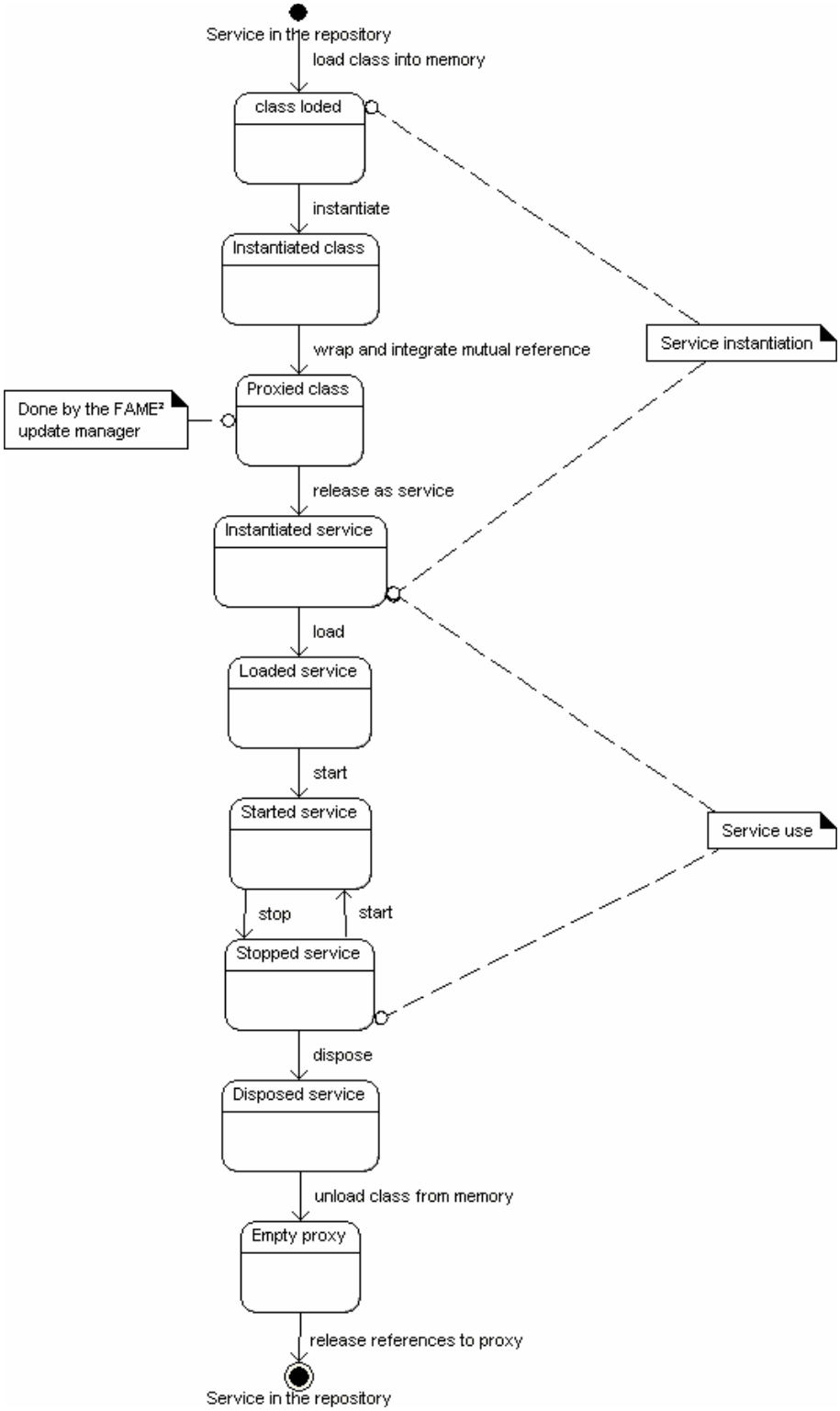

Figure 4-6: Life cycle of a service with implementation of the mutable reference

endpoint to every service instance once they are instantiated. Actually, the mutable reference endpoint is implemented as a proxy around a service. This proxy is created before the service can expose its own reference. Service developers should use a placeholder for the mutual reference that is defined in the service shell, instead of the *traditional reference*. Additionally, the proxy

implements part of the update functionality, i.e. the logic to replace a service with another one. The life cycle of a service with the implementation of the mutable reference is shown in Figure 4-6.
The UM enriches every FAME² service with the two additional operations, *getThis()* and *setThis(…)*. The operation *getThis()* returns the current instance of the service. If there is no proxy around the subscriber, the *getThis()* operation returns the value that the *traditional reference* would do. If there is a proxy around the subscriber, the *getThis()* operation returns the reference to the proxy that is around the subscriber. The *setThis(…)* operation updates the mutable reference.
Whenever a proxy is created around a subscriber, the proxy is, by contract, aware of the *setThis(…)* operation. Therefore, when the proxy is created, it invokes the *setThis(…)* operation on the subscriber, passing itself as parameter. Whenever the subscriber registers itself with, or is registered at a publisher, the *getThis()* operation is used to get a reference to the mutable reference endpoint. By using the mutable reference the reference at the publisher always refers to the most up to date proxy around the subscriber, or the subscriber itself.

Program 4-1: Source code of an implementation of mutable reference endpoint

```
1   private Subscriber mutableReference;
2   public Constructor() {
3     mutableReference = ProxyOfThis;
4   }
5   public Subscriber getThis() {
6     return mutableReference;
7   }
8   public void setThis(Subscriber s) {
9     mutableReference.setThis(s);
10  }
```

The source code in Program 4-1 is an example of an implementation for the *getThis()* and *setThis(…)* operations. The instance that is returned by *getThis()* is the mutable reference of the subscriber. This is done to take into consideration that the result of the *getThis()* operation is used for registration with publishers. Thus, the source code in Program 4-1 is a bridge pattern implementation of the subscriber (see [16] for a definition of the bridge pattern). The *ProxyOfThis* creates a proxy of the subscriber with the immutable reference to the subscriber as the initial object reference for the created bridge.
Figure 4-7 shows a sequence chart of interaction with a service based on publish/subscribe. Starting with step 1, the update manager of the FAME² service execution environment (*FAME² SEE*), which instantiates a FAME² service (*Service*), wraps the *Service* with a proxy (*Proxy 0*). The request to the update manager to wrap the service with a proxy is sent by the *Service* itself and is implemented as part of the FAME² service shell. During instantiation, the *Service* subscribes for events at a *Publisher*. The *Publisher* sends events to all its subscribers, including the *Service*. First, the *Publisher* retrieves the current reference to the *Service* by invoking the *getThis()* operation (step 2). The operation would return the reference to *Proxy 0*. Next, the *Publisher* sends an event to the reference endpoint, which is the *Proxy 0* created by the UM (step 2.1). *Proxy 0* forwards the event to the *Service* which had subscribed at the *Publisher* (step 2.1.1). Some time later the *FAME² SEE* is requested to wrap the *Service* with another proxy (*Proxy 1*) (step 3).
The *Proxy 1* updates the mutable reference endpoint by invoking the *setThis(…)* operation with itself as the parameter (step 3.1). The management of updates to the mutable reference is handled by *Proxy 0*. If the *Publisher* wants to send an event to the *Service*, then it retrieves the current reference to the *Service* by invoking the *getThis()* operation (step 4). This time, the operation would return the reference to *Proxy 1*. Next, the *Publisher* sends an event to the reference endpoint, which is the *Proxy 1* (step 4.1), which forwards the event to *Proxy 0* (step 4.1.1) which actually forwards the event to the *Service* which had subscribed at the *Publisher* (step 4.1.1.1). As it can be seen from this sequence diagram, wrapping a service with a proxy happens invisible to a service and its clients, or in other words, happens invisible to subscribers and publishers.

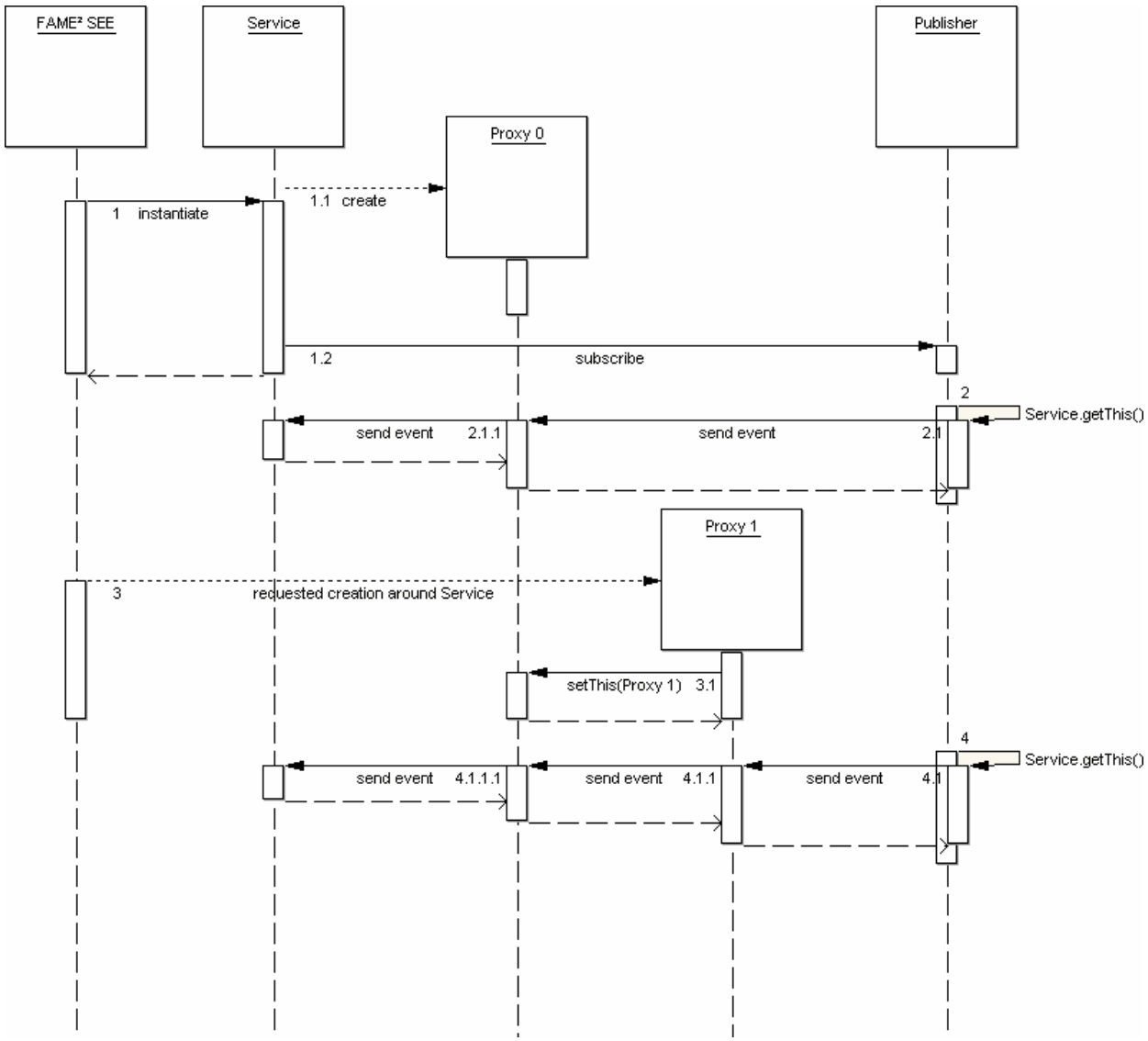

Figure 4-7: Sequence chart of the use of the mutable reference endpoint concept and publish/subscribe communication

Figure 4-8 shows the sequence chart for an update for the above situation. The communication is based on publish/subscribe. In step 1, the *Publisher* retrieves the current mutable reference endpoint, which is *Proxy 0*, of the *Service* and sends its event. *Proxy 0* forwards the event to the *Service*. In step 2, an update request is sent to replace the *Service* with a new, *Updated Service*. The update manager, as part of the implementation of *Proxy 0*, releases the binding to *Service* and establishes a new binding to the *Updated Service*. Events are sent then to the *Updated Service* (step 3). The update of the service is invisible for the *Publisher*, i.e. no termination and registration for the subscription needs to be done. Furthermore, the update is invisible for the *Service* and the *Updated Service* because no information about subscription needs to be exchanged between the *Service* and the *Updated Service*.

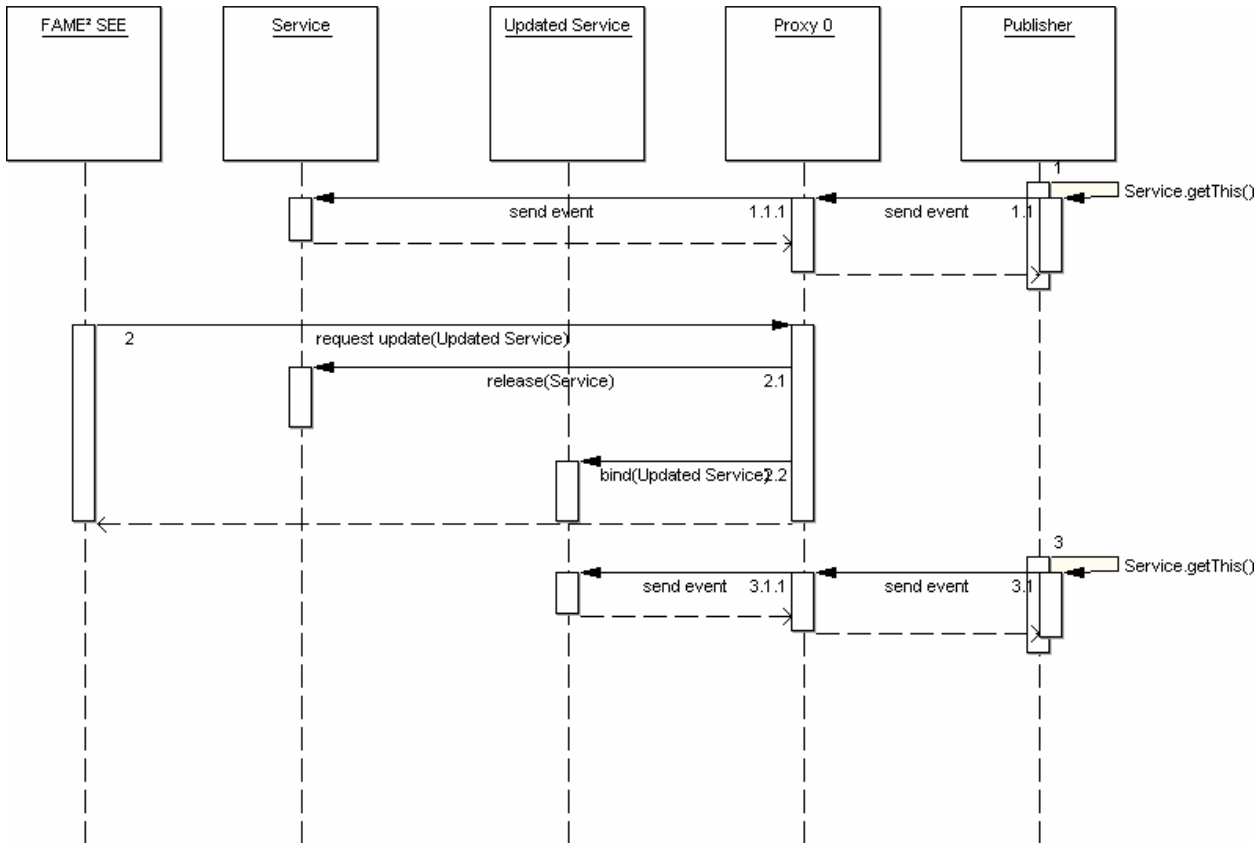

Figure 4-8: Sequence chart of an update and publish/subscribe communication

4.5 Summary

The proxy structural design pattern enables online-update of services. This pattern does not require any change of already existing software. Upon update, the proxy remains while the wrapped object is updated. Objects have references to the proxy only. All communication to a wrapped object passes through its proxy. However, the use of the proxy pattern and publish/subscribe as communication between objects is problematic. Publish/subscribe requires that subscribers expose their reference to publishers, often allowing inbound communication, i.e. events, to a proxied subscriber bypassing the proxy. When the proxied subscriber is then updated, the publisher cannot deliver the events because the addressed subscriber is replaced by an updated one.

This chapter has introduced a new concept for object oriented programming languages, the *mutable reference endpoint*. *Traditional references* in object oriented programming languages are, by definition, immutable and unique links to objects. A mutable reference endpoint is a unique endpoint for a reference to an object, but it can be changed, i.e. can be bound to another object. Mutable references avoid bypassing of proxies around subscribers in publish/subscribe architecture.

The FAME² update manager implements the logic to wrap every service with a proxy upon instantiation. This proxy adds a mutable reference endpoint. When the service is updated, the mutable reference endpoint will be bound to the updated service, and instantly updates the mutable references.

5 Extensible constraint framework

One of the primary objectives of creating software from services is reusability. Services are discoverable, self-contained software elements providing functionality for specific, well-defined tasks. Reusability of services promises a reduction of development cost, i.e. time, money, and human resources, resulting in cheaper software and shorter time-to-market. Furthermore, reuse promises increased software quality because the number of sources of programming errors, i.e. the number of services to be reimplemented, reduces with the number of services that are reused [263, 268-271]. Reuse of services is based on the economic principle of *economies of scale* [24, 272], which means that *doubling the resources results in more than double output* [273, 274]. But instead of doubling the resources for a current project, reuse enables utilisation of resources spent for past projects.

In practice, reuse is limited by the uncertainty of suitability [22-24], which also led to serious and costly software errors in the past [275]. A major source for this uncertainty is incomplete specification of the service interfaces [276, 277]. Usually, the service interfaces are the only contract between the service and its clients. Formulating and specifying explicit limitations between services and its clients is a solution to reduce uncertainty [25, 26]. The formulation of specification of such explicit limitations is called *Design by Contract* (DbC). The contract then defines the interface, but also the behaviour of software. Services with explicit limitations as a part of their contract are called *Trusted Services* [24], because they have guaranteed properties and behaviour.

DbC competes with *defensive programming* [278, 279]. As Meyer pointed out in his paper on *Applying "Design by Contract"* [25], defensive programming is about

> *to include as many checks as possible, even if they are redundant with checks made by callers. Include them anyway, the advice goes: if they do not help, at least they will not harm. […] This technique, however, often defeats its own purposes. Adding possibly redundant code "just in case" only contributes to the software's complexity – the single worst obstacle to software quality in general, and to reliability in particular. The result of such blind checking is simply to introduce more software, hence more sources of things that could go wrong at execution time, hence the need for more checks, and so on ad infinitum. Such blind and often redundant checking causes much of the complexity and unwieldiness that often characterizes software. – cited from [25]*

Following Beugnard et al [22], there are four types of contracts between services and their clients:

- Basic contracts
- Behavioural contracts
- Synchronisation contracts
- Quality-of-service contracts

Basic contracts specify the operations a service can perform, i.e. its operations, the input and output parameters each operation requires, and the possible exceptions, i.e. error conditions, which might be signalled by the operation when being executed. Behavioural contracts specify an operation's behaviour using conditions on input and output parameters. These conditions, called pre- and post-condition or –constraint, are Boolean assertions that ensure that operations receive proper input and return valid results. Synchronisation contracts specify the global behaviour of services, i.e. they express dependencies of timing and sequence for invocations of the service operations. Quality-of-service contracts specify qualities of the service behaviour, e.g. maximum response delay, quality and precision of the result, number of responses for a given number of queries, and so on.

Regardless of the type of a contract, a contract aims to protect both contractors, e.g. a service and its client. A contract protects the client by specifying how much should be done to be entitled to receive a certain result. A contract protects the service by specifying how little is acceptable to be not liable for failing to carry out tasks outside of the specified scope [25]. This idea goes back to Floyd-Hoare-Logic [280] and the Hoare triple that provides a simple mathematical notation for reasoning about the correctness of software: $\{P\}S\{R\}$. In this notation, P and R are predicates and S is a statement of the software. The Hoare triple says that if a pre-condition P is hold for a statement

S, then this statement *S* will produce a result that is compliant to a post-condition *R*. Beside pre- and post-conditions there are invariants, which ensure that a software is never in an inconsistent state.
While DbC is a powerful tool to increase reliability and quality of service use, existing approaches do not take into consideration the dynamic and loose coupled nature of software systems created from services. This dynamic and loose coupled nature requires that contracts are dynamically negotiable, i.e. contracts adapt to the situations and contexts [22].
This chapter describes an approach to dynamically formulate and negotiate pre-conditions for behavioural and quality-of-service contracts. This approach, called the *Extensible Constraint Framework* (ECF), also supports contract enforcement and monitoring, which is identified as critical element for contract management [281, 282]. The next section reviews existing solutions for the different approaches for DbC. Section 5.2 describes the ECF. A discussion and evaluation of the ECF is given in Section 5.3.

5.1 Review of existing solutions for Design by Contract

There are numerous examples that demonstrate the usefulness of DbC in different software. Sicilia and Sánchez-Alonso [271] use DbC to formulate dependencies between *learning objects*. Learning objects are *independent and self-standing units of learning content predisposed to reuse in multiple instructional contexts* [283], e.g. courses at a university. Pre-conditions are the requirements that a student is required to follow the course material. Post-conditions are the knowledge the student has gained when successfully completing the course.
Liu and Cunningham [23] demonstrate the use of DbC for service discovery. In service discovery, clients may receive multiple responses to one query. Selecting the most appropriate discovered service includes an exploration of its interface. Contracts help in this selection process by providing additional information about the service capabilities.
The authors of [268, 270] use contracts not only in the implementation of software elements, e.g. services, but also in their architecture and design. Pre-conditions express the requirements of the service architecture and design, e.g. other services or a specific architectural style [84, 259]. Post-conditions express to what services and architectural styles the service architecture and design are compatible. The intention of [268, 270] is to enable reuse of architecture and design in different software projects. They also indicate limitations of DbC when services do evolve, i.e. extending or changing their provided functionality. These limitations result from the explicit formulation of post-conditions. While post-conditions are required for a formal proof of the correctness of software, they are often a limiting factor when services might become more powerful. The reason is that post-conditions can only be strengthened to avoid that clients of services will receive a result from an evolved service that is outside of the previous specification of the service.
In [284, 285] it is described how DbC can simplify emerging programming styles, e.g. extreme programming and aspect oriented programming. The idea of extreme programming is to implement functionality without previous design but only a specification. This specification documents the inputs to accept, and the expected outputs for different inputs. Programmers then implement the functionality and run a test on their implementation, testing if every specified input returns the specified output. If this test is not successful, then the functionality is implemented again. The intention of extreme programming is to avoid the time required for a design, because in practice every design does not cover some of the requirements the software should solve. Design by Contract is used to simplify the testing of functionality because the accepted inputs and the resulting outputs are no more documented in an external specification, i.e. a printed text document, but rather being explicitly formulated as a part of the source code of the implementation. Aspect oriented programming (AOP) improves separation of concerns, easing software development by supporting modularisation. Services are designed and coded separately from code that cross-cuts their functionality [286, 287]. The latter is code which implements a non-functionality feature, e.g. logging, access control, and pre- and post-conditions. Some of the cross-cutting features which are expressed in aspects (thus the name AOP) cannot be woven with other features into the same

service since two features could be mutually exclusive. In [284] it is described how to use DbC to explicitly specify the compatibility between aspects.
The authors of [269, 275] show how DbC increases the quality of software and allows using software reuse and the object-oriented programming paradigm in safety-critical software. In [275] it is analysed how the cause of the Ariane 5 rocket explosion during its maiden flight could have been avoided. The explosion, resulting in an economically loss of a half-billion US-dollar, was caused by reusing a software function from the previous Ariane 4 rocket. This software function expected that the flight's horizontal bias is represented as 16-bit signed integer value, whereas the software of the Ariane 5 rocket was designed to represent this value as a 64-bit floating point value. In [275] it is showed that it was nearly impossible to detect this mistake because even this incompatibility was documented it was too difficult to find this information in the whole amount of available documentation. Furthermore, all tests were passed successfully, resulting in less motivation to further study the software and its documentation to detect any incompatibility. Nevertheless, they demonstrated how DbC would have avoided this software bug without any testing, but just by compiling the software. Crocker [269] discusses on how to use the object-oriented programming paradigm, still considered as unsafe, in safety-critical software for airplanes and nuclear power plants. He demonstrates this by a detailed example of flight-instrument software that displays the most appropriate flight instrument on a screen for the pilot of an airplane. His primary pillar to ensure the correctness of the software, which is a must for safety-critical software, is the use of DbC to ensure that the different parts of the software, i.e. objects, fit together and operate in the expected way.
Another motivation for Design by Contract is provided in [288] that states that blaming the source of mistakes, e.g. passing invalid parameters to software, increases the overall quality of software systems. Existing approaches for DbC do a static modelling of contracts by using pre-compiler directives, new keywords in programming languages or totally new programming languages, runtime instrumentation, wizards, or aspect oriented programming.

5.1.1 Pre-compilers

Pre-compilers are tools that modify source code of software before it is compiled. Such modification has often the purpose of replacing short directives with longer language constructs that serve a specific purpose, like contract enforcement. Using pre-compilers does not translate source code into executable code, but rather is a step before compiling source code.
In [276, 289] it is described the *Java Modelling Language* (JML) to enrich the specification of source code in the Java programming language with assertions, i.e. pre- and post-conditions. JML is an unobtrusive extension to Java. The specification of source code is annotated with the assertions in the form of comments. Existing compilers will ignore these comments, whereas a JML pre-compiler translates the assertions into Java source code. Using the pre-compiler, and then the compiler, activates assertion checking, while not using the pre-compiler disables the checking. The JML assertions are runtime checks, i.e. they are checked while the Java software executes. JML does not allow refining and negotiating contracts between clients and services. This makes JML less suitable for software dynamically composed form services.
The iContract tool, described in [290], is another pre-compiler that translates assertions, described in comments of the Java programming language, into Java source code. In comparison to JML, iContract supports recursive contract checks. A recursive contract check is used in inheritance to evaluate the assertions defined in all super-classes a particular class is derived from. Like JML, iContract does not allow refining and negotiating contracts.
Another pre-compiler is the *Java with Assertions* (Jass) described in [291]. Jass annotates Java software with specifications in the form of assertions, i.e. pre- and post-conditions. In addition to JML and iContract, Jass supports *trace assertions* and *refinement checks*. Trace assertions are used to monitor the dynamic behaviour of objects in time. Thus, they are especially useful for concurrent software, i.e. software where its parts are executed concurrently. Refinement checks are used to determine whether a class is related with another class, e.g. it is a derived class. In a subtype

(derived class), method pre-conditions can be weakened and post-conditions can be strengthened. Like in JML and iContract, assertions are checked during runtime of the software. However, Jass is not suitable for software dynamically composed from services because it lacks a negotiation and refinement concepts for contracts after software is compiled and deployed.
All approaches using pre-compilers specify assertions in an unobtrusive manner. Existing compilers are able to compile source code annotated with assertions because the annotations are done in comments. To include the assertions into the software, the source code has to be pre-compiled before it is compiled. As a result, the assertions are always checked at runtime of the software. A disadvantage that all approaches based on a pre-compiler share is the lack of negotiation and refinement of assertions, i.e. contracts, after the compilation of software. This limitation makes these approaches unsuitable for software that is dynamically composed from services.

5.1.2 Special compilers

Special compilers have specific features that are not available in standard compilers. Special compilers are replacements for standard compilers. While translating source code of software into executable code, the special compiler adds language constructors to serve different purposes, like contract enforcement.
Using pre-compilers to compile contracts into source code of programming languages, as described above, results in contracts being checked during execution time of the software. To enable consistency checking of contracts, the authors of [292] argue that they need to be an integral part of the programming language. As a consequence, they introduced new keywords into the Java programming language to express pre- and post-conditions and use one compiler to translate the source code into binary code. They further argue that their approach simplifies debugging and incremental compilation because there is no need for a separate compilation for the contracts (there are no pre-compilers). The disadvantage of this approach to introduce new keywords into a programming language is the requirement for a special compiler. Especially open-source projects will suffer from such approach because it requires everyone who wants to participate in the project to use a non-standard compliant compiler. Also, the lack of negotiation and refinement of contracts is not supported.
Nunes introduce another keyword into programming languages, i.e. the Java programming language, to access the previous state of objects [293]. The use of the previous state of objects is helpful when defining contracts for transactions, where the previous transaction influences the next transaction. Like in the work of [292], the approach of Nunes lacks the capability of negotiation and refinement of contracts after the source code is compiled.
Allen and Cartwright strengthened the type checking for containers in the Java programming language via a form or pre-condition [294]. Similar approaches were done by the authors of [295-298].

5.1.3 Runtime instrumentation

Modifying executable code while it is being executed is the purpose of runtime instrumentation. Directives are replaced or augmented by additional directives just at the moment or slightly before the software is being executed.
jContractor is a tool that uses runtime instrumentation to add contracts to Java software [299-301]. Contracts are written in the programming language and follow a rigid naming scheme. jContractor analyses the software, using Java reflection technology [139, 141], and modifies the compiled software to insert contract checking at runtime. The advantage of jContractor, compared to solutions like JML [289], Jass [291], or special compilers as described in [292, 293], is the unobtrusive and common process of software development. There is no need to use any special compiler or pre-compiler. jContractor supports the inclusion of previous states of objects into the evaluation of contracts, similarly to the work presented in [293]. jContractor allows negotiation and refinement of contracts after software is compiled. The limitations of jContractor are the use of signed code, i.e. when the authenticity and integrity of the software is secured with a digital

signature. Modifying such software would result in a negative evaluation of the digital signature. Furthermore, instrumentation requires low-level access to the software and can take place only before the software is executed. Once the software runs, no more modification, e.g. negotiation and refinement of contracts, is possible. Another problem is the rigid naming scheme, being a source of errors when a contract is not added to an operation because of typos. Such errors are not detected and not reported by jContractor, resulting in wrong assumptions about the running software.

5.1.4 Web Services Policy

The *Web Services Policy* (WS-Policy) *Framework* describes the non-functional aspects of Web Services. It provides the grammar for defining attributes that represent the non-functional aspects. An attribute is part of an assertion. Assertions represent a requirement or capability of a Web Service. Assertions are grouped in alternatives. A policy is a collection of alternatives [302]. While WS-Policy was designed to support synchronisation and quality-of-service contracts, it supports behavioural contracts only [303]. Web Services policies need to be matched, there is no enforcement concept. Policy matching can be done on syntactical level, comparing attributes for equality, or using semantic interpreted information [304]. Policy definitions in WS-Policy can be refined at runtime of the software the policy document is written for, because they are formulated in XML documents. Modifying these XML documents does not require the software to be in a specific state. However, modifications do not apply to ongoing interactions between clients and the software. WS-Agreement allows for negotiation of policies [305]. WS-Agreement is a grammar to exchange and compare WS-Policy documents.
An open question for WS-Policy is the lack of enforcement of policies. As WS-Policy depends on clients who evaluate and respect the policies in the WS-Policy document of software, there are no penalties when clients disobey the specified policies.

5.1.5 Wizards

Wizards are, in this context, tools that help developers to specify functionality for the software they program. Usually, wizards have a graphical interface, and abstract from the source code by letting developers intuitively select the functionality they want to include in their software.
Defining contracts requires programmers to express them in the form of pre- and post-conditions in the source code (see above for examples). Designers of software already define constraints that the implementation must follow. A natural step would be to allow designers to define the contracts that the implementation must keep. Wizards address this requirement and provide tools that abstract from the programming language and allow designers to define their contracts.
Such wizards are described in [306] for the Microsoft .NET. The designer can use this tool to formulate constraints independently from a programming language. The wizard integrates well into the standard development environment for Microsoft .NET, the Visual Studio .NET. The wizard supports pre- and post-conditions. However, there is no support for previous object state, as e.g. in jContractor [299] or the work presented in [293]. Also, there is no support for contract negotiation and refinement after the software is compiled.
Another wizard, designed for the Python programming language, is presented in [307]. Python is a programming language available for a number of computing devices, including personal computers and mobile phones. However, the same restrictions and limitations that apply to the .NET wizard also apply to this wizard.

5.1.6 Aspect oriented programming

In aspect oriented programming, source code is augmented by directives that are treated special by compilers. These directives advice compilers to include available functionality like contract enforcement.
The objective of aspect oriented programming (AOP) is to weave cross-cutting functionality into software during its compilation. Such cross-cutting functionality is usually tangled over numerous parts of the software. Examples are logging, access control, and pre- and post-conditions. The cross-

cutting functionality is called aspects. The advantage of AOP is that the developer concentrates on the functional part of the software, e.g. numerical calculations, data manipulation, etc., while the aspects are woven into the software by the compiler.
In [308], aspects for checking pre- and post-conditions are defined. The implementation of the aspect is an ordinary function implemented in the same programming language as the software is programmed in, and woven into the software by the compiler. A similar approach is described in [309].
The authors of [310] extend the JWAM framework. JWAM is a framework for interactive business applications and is structured into various business related layers [311]. Part of JWAM is a layer realising DbC. Using the DbC functionality of JWAM requires an explicit invocation of the functionality. AOP is used to make this invocation implicit by analysing comments in the source code and creating proper pre- and post-conditions based on these comments.
Like the approaches based on pre-compilers, special keywords for programming languages, and wizards, the approaches based on AOP do not support negotiation and refinement of contracts after compilation of the software. This limitation makes the approaches based on AOP unsuitable for software that is dynamically composed from services.

5.1.7 Contract violation

In [22], different possibilities of actions that can be taken in the case of a contract violation are listed. The first possibility is to ignore the contract violation, averting any effect. In some situations it may be acceptable to ignore a contract violation, e.g. when it the redeeming of the contract is desired but not critical. An example for such contract would be email delivery with a guaranteed delivery time. When the email is delivered too late, only a part of the contract is violated, and the sender may decide to ignore the violation because the email was at least received. The second possibility is to reject the request for an operation. For example, a calculator service that multiplies two positive integer numbers, but requires that the values of both terms are below 100, may reject the calculation request if any of the terms has a value above 100 or is negative. Typically, such reject is signalled with an exception, indicating that at least a part of the contract was violated. The third action is to wait. Waiting might be feasible when dependencies are not resolved, but it is expected that they might be in the future. An example is an email service that waits with the delivery of an email until network connectivity is available. The fourth action is to negotiate. If a part of the contract cannot be hold in the current situation, then the contractors may change the contract so it will be satisfied. An example would be a limitation of the number of invocations of service operations in a defined period of time. Clients may request from a service to increase the number of allowed invocations, possibly in exchange of a decreased number of allowed invocations at another time. What can be seen from the different possibilities of actions is that they all apply on pre-conditions, i.e. to react on a contract violation before an operation is executed. Furthermore, the different examples motivate that services should be able to react with all four different actions to contract violations. Ignoring and waiting does not require any special feedback from the service to the client. Rejection and negotiation requires a feedback from the service to the client. This feedback can be given in the form of exceptions, requesting the client to react to an exception with either accepting it, or starting a negotiation process.

5.1.8 Comparison of existing solutions

Services may depend on other services. When a dependant service disappears and is replaced by another, similar service, the pre- and post-conditions of services may change. As a result, composing software from services requires negotiation and refinement of contracts on services. A solution for Design by Contract for use in software based on the concept of service oriented architecture requires changing conditions after service deployment and during service execution.
Conditions should be formulated in the same programming language as services. A dedicated or modified programming language for expressing conditions requires that developers have to learn another programming language, potentially reducing the willingness to define pre- and post-

conditions. This includes the use of tools as well. Optimally, developers do not need to learn another programming language to express conditions, and do not require to use any additional or modified tool. Finally, there must be no requirement for a special runtime environment for contract enforcement, because this will hinder the widespread use of contract-aware services. Optimally, a solution for DbC has to support all four different contract types identified by Beugnard et al [22]: basic contracts, behavioural contracts, synchronisation contracts, and quality-of-service contracts. Finally, optimally all types of contract violation are supported.
Table 5-1 compares the reviewed existing approaches and lists their capabilities.

Table 5-1: Comparison of existing approaches for design by contract

	Pre-compilers			New keywords		Instru-mentation	WS-Policy	Wizards		Aspect oriented programming	
	JML	iContract	Jass	[292]	[293]	jContractor		.NET	Python	[308]	[310]
Supported conditions											
Pre	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Post	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes
Invariant	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	No	No
Previous state	Yes	No	No	No	Yes	Yes	No	No	No	No	No
Supported contracts											
Basic	Yes[1]	Yes[1]	Yes[1]	Yes[1]	Yes[1]	Yes[1]	No	Yes[1]	Yes[1]	Yes[1]	Yes[1]
Behavioural	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Synchronisation	No	No	No	No	No	No	No	No	No	Yes	Yes
Quality-of-service	No	No	No	No	No	No	No	No	No	Yes	Yes
Supported features											
Unobtrusive	Yes	Yes	Yes	No	No	Yes	Yes	Yes	Yes	No	Yes
Negotiation	No	No	No	No	No	Yes[2]	Yes	No	No	No	No
Refinement	No	No	Yes[2]	No	No	Yes[2]	Yes	No	No	No	No
Proof of contract	Yes	No	No	No	No	No	No	No	No	No	No
Requesting conditions	No	No	No	No	No	No	Yes	No	No	No	No
Supported contract violations											
Ignore	No	No	No	No	No	Yes	No	Yes	Yes	Yes	Yes
Reject	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes
Wait	No	No	No	No	No	Yes	No	Yes	Yes	Yes	Yes
Negotiate	No	No	No	No	No	No	No	No	No	No	No
Does not require learning / using …											
New language	No	No	No	No	No	Yes	No	Yes	Yes	Yes	Yes
New tools	No	No	No	No	No	Yes	Yes	Yes	Yes	No	No
Modified runtime environment	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes

[1] : support is provided by the interface specification (operation name, parameter signatures, etc.) of the service
[2] : Negotiation and refinement are only possible while the service is not running

Summarising, there is no solution that supports all contract types and all contract violations. Furthermore, all existing solutions require to learn new languages or tools or to use a modified runtime environment when executing services.

5.2 Description of the ECF

This section describes the Extensible Constraint Framework (ECF), which is an optional part of FAME². It is a new approach to dynamically define, refine and negotiate all different types of contracts on services and their interactions.
Existing approaches for Design by Contract integrate (also known as *hard-wire*) pre- and post-conditions into the software either at compile time or when the software starts. Hard-wiring means that the assertions are an integral part of the software, and there is no possibility to negotiate or

refine contracts while the software is running. However, such negotiation and refinement is desirable for behavioural and quality-of-service contracts. Behavioural contracts might be changed when dependencies of software elements change, e.g. a dependant service disappears and is replaced with another service that has slightly different, harder or weaker, limitations. Quality-of-service contracts may be negotiated to cover times of heavy load, or different context, e.g. when a computer hosting a service changes from being battery powering to being line operated, or the service moves to another computer with different hardware like a more powerful CPU. Existing approaches do not provide such support of negotiation and refinement of contracts.

The *extensible constraint framework* (ECF) defines pre- and post-conditions, also called constraints, as software artefacts, i.e. classes in object oriented programming. These software artefacts can be added and removed from specially prepared service interfaces. The preparation of service interfaces is part of the FAME² update manager, but is also realisable via an ECF service (described below). The proxy that is created around each service has a special container to hold pre- and post-conditions. When an operation of a service is invoked, the invocation is intercepted by the proxy around this service. Before the operation itself is invoked, the passed parameters are passed to the assigned pre-conditions. The result of the operation is checked against all assigned post-conditions before it is returned to the invoker of the operation. The action of contract violation, i.e. ignoring, rejecting, waiting, or negotiation, depends on the implementation of the pre-condition. Post-conditions always ignore contract violations.

The ECF supports four operations to model any kind of Boolean relationship between conditions: and, or, not, and parenthesis. Conditions are activatable or deactivatable depending on the current situation or context, e.g. to reflect fluctuations in available resources, or heavy service use. Violations of conditions are not only indicated by Boolean values, as it is the case in JML, iContract, jContractor, etc., but by expressive exceptions that contain details what condition was violated and why. On the one hand this helps in debugging software; on the other hand this information is useful to adapt software at runtime to meet the capabilities of current situations and contexts. The ECF comes with a service that allows querying the pre- and post-conditions assigned to operations of services. This information can then be used for other services, like service discovery for more accurate service selection. Negotiation of contracts is also possible via the ECF service.

The ECF supports the following features:

- **Seamless integration** into existing software development tools. There is no requirement for compiler support.
- **Extensibility** by defining new constraints which seamlessly integrate with the ECF. New constraints are assigned to services without any source code modification, recompilation and redeployment of the service.
- **Dynamic constraints** by allowing adding and removing of constraints from a service at runtime. No recompilation or restart of the service is required for the new constraints to be activated.
- **Grouping of constraints** into constraint functions. Constraint functions behave like normal constraints. For grouping of constraints Boolean logic is used. The Boolean logic operations of and, or and not are supported. Constraint functions are always evaluated at once, such that a constraint function is comparable to the mathematical construct of parenthesis.

5.2.1 Example

Program 5-1 is an example of a service that stores objects of unspecified type. The ECF is used to limit the types that might be stored to be of type *java.lang.Integer* and *java.lang.String*. Furthermore, the ECF is used to limit the number of stored objects, and demonstrates how the size of the storage, i.e. the number of stored objects, can be changed. The Java source code of the essential parts of the example is shown in Program 5-1.

Program 5-1: Example of using the extensible constraint framework

```
1   public interface IStorageService extends IService {
2     public void add(ISecurityPrincipal Principal, Object Entry) throws Throwable;
3     public void remove(ISecurityPrincipal Principal, Object Entry) throws Throwable;
4     public int getSize(ISecurityPrincipal Principal) throws Throwable;
5   }


6   public class CStorageService extends AService implements IStorageService {
7     private java.util.Collection m_Storage = new java.util.LinkedList();
8     public void add(ISecurityPrincipal Principal, Object Entry) throws Throwable {
        m_Storage.add(Entry); }
9     public void remove(ISecurityPrincipal Principal, Object Entry) throws Throwable {
        m_Storage.remove(Entry); }
10    public int getSize(ISecurityPrincipal Principal) throws Throwable { return
        m_Storage.size(); }
11  }


12  public class CTypeConstraint implements IConstraint {
13    private Class[] m_AllowedTypes;
14    public CTypeConstraint(Class[] AllowedTypes) { m_AllowedTypes = AllowedTypes; }
15    public void check(Object[] Values) throws EConstraintViolation {
16     for (int i = 0; i<Values.length; i++) {
17      boolean valid = false;
18      for (int j = 0; j<m_AllowedTypes.length; j++) {
19        if (Values[i].getClass().equals(m_AllowedTypes[j])) {
20        valid = true;
21        break;
22        }
23      }
24      if (!valid) { throw new EConstraintViolation("The parameter number " + i + " has a type
          that is not allowed for this operation."); }
25     }
26    }
27  }


28  public class CUpperLimitConstraint implements IConstraint {
29    private java.lang.reflect.Method m_Method;
30    private Object m_Object;
31    private ISecurityPrincipal m_Principal;
32    private int m_Limit;
33    public CUpperLimitConstraints(java.lang.reflect.Method M, Object O, ISecurityPrincipal
        P, int L) {
34     m_Method = M;
35     m_Object = O;
36     m_Principal = P;
37     m_Limit = L;
38    }
39    public void check(Object[] Values) throws EConstraintViolation {
40     try {
41      Object result = m_Method.invoke(m_Object, new Object[]{m_Principal});
42      if (result instanceof Integer) {
43        if (((Integer)result).intValue() > m_Limit) { throw new EConstraintViolation("The
          object already stores the maximum number of objects. Please remove objects before
          adding new ones."); }
44      } else { throw new EConstraintViolation("The given method \"" + m_Method + "\" does not
          support the upper limit constraint."); }
45    }
46    public void setLimit(int NewLimit) { m_Limit = NewLimit; }
47  }


48  public interface IECFService implements IService {
49    public IService constrainService(ISecurityPrincipal Principal, IService Service) throws
        Throwable;
50    public void addPreCondition(ISecurityPrincipal Principal, IService Service,
        java.lang.reflect.Method Method, IConstraint Constraint) throws Throwable;
51  }
```

The interface declaration of the storage service is shown in the lines 1 to 5. The implementation of the service is in the lines 6 to 11. The constraint that limits the storage service to store objects of certain types is shown in the lines 12 to 27. Lines 28 to 47 show the implementation of a constraint that limits the number of elements stored. A part of the interface of the ECF service is shown in the lines 48 to 51.

Constraining the storage service to limit the types of objects to be stored within is realised by invoking the *constrainService(…)* operation of the ECF service with the instance of the storage service as a parameter. The result will be a storage service where conditions can be assigned to. To limit the types that this storage service is allowed to store, the operation *addPreCondition(…)* is invoked with the storage service as a parameter and a properly configured instance of the *CTypeConstraint* class. The *Method* parameter of the *addPreCondition(…)* operation can remain empty (=*null*) because this parameter is not required by the *CTypeConstraint*. After the instance of *CTypeConstraint* is assigned to the storage service, no other types than configured in the constraint may be stored in the storage service. The constraint does not, however, affect already stored objects. Thus, the integrity of the storage service, which may depend on the containment of objects of other types, remains intact. The use of the *CTypeConstraint* is also more powerful than using templates. Templates are an element of generic programming, to specialise generic algorithms to a specific data type [312]. Templates would restrict the storage service to accept exactly one type, but not both (*java.lang.Integer* and *java.lang.String*). The *CTypeConstraint* supports to restrict the storage service to accept several different types. One might also imagine other constraints to limit the value range of accepted integers, or the length of strings to store.
Limiting the number of objects to be stored in the storage service is realised by the *CUpperLimitConstraint*. It is assigned to the storage service via the *addPreCondition(…)* operation of the ECF service (*IECFService*), where the *Method* parameter is a reference to the *getSize(…)* operation of the storage service. This reference is required so the *CUpperLimitConstraint* can ascertain the number of objects currently stored. Changing the maximum number of objects to be stored is configurable via the *setLimit(…)* operation in the *CUpperLimitConstraint*.
All the constraints are assignable to the storage service while the service is running. Also, there is no requirement to prepare the storage service to be constrained.

5.3 Evaluation and discussion

The extensible constraint framework (ECF) supports the formulation of pre- and post-conditions and previous states. It does not support invariants, because the objective of ECF is to control the interaction between services, and not the validity of service execution. With ECF, it is possible to define all four types of contracts: basic, behavioural, synchronisation, and quality-of-service contracts. Furthermore, ECF is the only solution that supports all four types of contract violation. ECF does not require the developer to learn any new tools, language, or use any modifid runtime environment when executing services. Except proof of contracts, ECF also has support for the different features listed in Table 5-1 above. ECF is the only approach that supports negotiation and refinement of conditions while a service is running.

6 Future work

The *Framework for Applications in Mobile Environment* (FAME²) is a framework for developing middleware based on the concept of service oriented architecture. It is a collection of features considered to be useful and mandatory for middleware that is used in distributed computing systems in ubiquitous computing: service life cycle, service discovery, online-update, refinement and negotiation of contracts, support for access control, and tagging interfaces to model cross-cutting aspects of services like logging and billing.
While FAME² provides a set of useful and reasonable features and guidelines, there are features that would increase the usability of FAME² for middleware developers. Among these features are:

- extended and autonomic service life cycle support with dependency resolution
- semantic service descriptions and service modelling
- smart and meaningful selection of available service discovery solutions for optimal resource preservation
- autonomic service deployment for optimal resource use in a distributed computing system
- service validation

In this chapter, some thoughts for future work on these identified topics are given.

6.1 Service life cycle and dependency resolution

In current FAME², there is no formulation of dependencies between services. When a service depends on another service, this other service must be available, where availability means that the depending service is running either locally or remotely and if it runs remotely, then a communication protocol for remote communication must be available. There is no mechanism in FAME² that runs depending services automatically. This complicates management and administration of FAME² based software run in distributed systems, because users have to resolve dependencies manually.
The PCOM middleware includes a partial solution by explicitly formulating dependencies between services [127]. A similar approach of explicit dependency formulation and resolution for the Open Services Gateway initiative is presented in [180]. Yet, PCOM and the work presented by Cervantes and Hall are not able to automatically run services when they are demanded. The *Information Society Technology* (IST) *Future and Emerging Technologies* (FET) project CASCADAS (*Component-ware for Autonomic Situation-aware Communications, and Dynamically Adaptable Services*) will tackle this question when investigating life cycle for services distributed in a self-organising network [313].
To integrate dependency resolution, and thus an automated service life cycle, the FAME² *life cycle manager* (LCM) can be improved. As the LCM is separated from services, its modification has no negative effects on already existing services.

6.2 Semantic service description and modelling

The features and capabilities of services are described in their description. Existing service description languages, e.g. the *Interface Description Language* (IDL) of CORBA and the *Web Services Description Language* (WSDL) of Web Services do describe the interface specification of services, i.e. method names, types of parameters, return types, and signalled exceptions. Such descriptions are machine processible, that is they can be analysed, interpreted, and modified by computers, but they provide little information about compatibility, requirements, and behaviour that would help developers when creating service ensembles.
Using service description with semantic information enables the introduction of fuzzy concepts into features like semantic service discovery, service choreography, service interaction modelling, etc. Instead of searching for services with precise request and requiring exact matches, natural language queries are used, e.g. looking for a “service that provides the current location” and imprecise matches would be sufficient. Similarly, the realisation of choreography of services based on the

semantic description of their features, mediated by a communication middleware and binding framework, and not on the interface specification, would enable a more dynamic interaction between services. Developers are supported when modelling service ensembles by designing the flow between services based on semantic information like compatibility, requirements, and behaviour.
Several scenarios for semantic service description and modelling in their work on data management, and Web Services choreography are presented in [314-316, 355]. The approach described in [317] distributes the semantic service description in a knowledge network and utilises this knowledge network for composing services. Using the information on requirements specified in semantic service descriptions to optimise the resource use in GRID computing systems is presented in [318]. A comparison of different approaches for semantic service descriptions and their matchmaking in service discovery is given in [319] with the result that existing semantic service descriptions fail to describe aspects like behaviour of services unambiguously. Altogether, the existing approaches show the benefits of using semantic service descriptions, while specialising their description languages to particular domains, e.g. mobile data management [314], GRID computing [315, 318], and service composition [317]. However, the objective should be a general semantic service description language.
The repository realised as part of the FAME² service execution environment supports the management of different service descriptions. However, FAME² does not specify a service description that could be used for automatic dependency resolution, semantic description, choreography, etc.

6.3 Service discovery optimisations

The *Open Service Discovery Interface* (OSDI) is an abstraction from service discovery solutions that are based on the request/response schema, i.e. where clients send requests querying about services, and responses are returned from service discovery solutions with information about found services. Similar approaches are the service discovery abstraction in the *Support for Service Discovery and Interaction* (SSDI) project [12, 13], the *Open Service Discovery Architecture* (OSDA) [14], and the *service discovery framework* in the *Reflective Middleware for Mobile Computing* (ReMMoC) [15]. However, only the OSDI permits clients to formulate their requests in any query language, integrates new service discovery solutions as services allowing for their own discovery (self-similarity), is flexible by allowing clients to interact with service discovery solutions directly or via the abstraction, and supports continuous operation
Future work should investigate optimisations for service discovery, for example study mechanisms to determine the *most efficient* network (UMTS, Bluetooth, W-LAN, etc.) to use when many of them are available. *Most efficient* refers to aspects of energy consumption, bandwidth, cost, response time, accuracy, reliability, availability, etc. Current research in virtual network interfaces and network abstraction might provide such mechanisms [320-324].
The OSDI achieves access transparency for service discovery. The selection algorithm implemented in the event dispatcher of the OSDI, which decides where events are sent to, is based on the inefficient *cycle and see philosophy* [15]. Smarter selection algorithms may increase efficiency of the OSDI by selecting only the relevant out of the available service discovery solutions, and thus limit the number of destinations where events are sent to. The design of OSDI supports the replacement of the selection algorithm at runtime, making it possible to use optimised selection algorithms for different situations. First hints for alternative selection algorithms may come from network packet scheduling, e.g. Round Robin [325-329] and Fair Queuing [330, 331].
Research on translation of query languages of service discovery would result in a more flexible use of it. Translation means to formulate a query in the language of one service discovery solution, and let it automatically translate into the language of another service discovery solution. The advantage would be then that two otherwise incompatible service discovery solutions can be used by issuing only one query. The architecture of OSDI includes the concept of query translators. However, translating query languages is not realised within this dissertation. For translation, there are two

possible approaches: first, having a direct translation, and second, defining a common language into which all other languages can be translated. In this dissertation, no recommendation on the approach to select is given. The OSDI supports both approaches. Further optimisations can be achieved by choice of filters. Filters reduce the set of discovered services. This reduction simplifies the selection process to be realised within the client of OSDI when numerous services are discovered that match the query of the client. Filtering can be based on aspects of security (like access control), communication cost, availability, reachability, quality of service metrics, behavioural contracts, etc.

6.4 Service deployment

FAME² supports reconfigurability of middleware by adding and removing services at its runtime. As a result, FAME² theoretically supports logical mobility. Optimising resource use, quality of service, and optimising communication between services are the main reasons for logical mobility. Other reasons are security considerations, personal preferences of the service users, and expected unavailability of remotely used services. Logical mobility of services includes service deployment. Service deployment comprises the steps of release, installation, update, activation, deactivation, adaptation, uninstalling, and de-release of services [332]. Additional steps of service deployment are license management, dependency checking (see Section 6.1), requirements checking, compatibility questions, and so on. A comprehensive overview of numerous solutions for deployment is given in [332]. While there is progress in realising the above steps (e.g. [333-337]), there is one question that remains unanswered until today: *"Who deploys what when to where?"*
This question is of particular relevance in ubiquitous computing. Ubiquitous computing envisages unobtrusive distributed systems, i.e. they do not bother the user but just serve him. Due to the heterogeneity and mobility expected in distributed computing systems for ubiquitous computing, logical mobility is a necessary feature to optimise resource use, increase quality of service, and to optimise the communication between services. To implement unobtrusive distributed computing systems, users, including administrative and technical personnel, are not the answer to the *"Who"* in the above question. A possible answer for the *"Who"* could then be: the computers themselves.
The *"what"* in the above question could simply be answered with *"services"*. But this answer is too simple. It has to be clarified what is a service, and what is part of a service. Services may have a persistent state. This state might be deployed, too, to support that a deployed service just continues in the same state as before its deployment. On the other hand, it might be that such persistent state should not be deployed to support that the service does not continue with its previous state, but performs a total restart. Furthermore, the persistent state of a service may be invalid on the computer where the service is deployed to. A possible solution to this problem could be to limit the *"what"* to *"stateless services"*.
The *"when"* and *"where"* part of the question is probably the most challenging. When answering this part from the viewpoint of communication between services, the answer would be *"as soon as possible as close to the communication partner"*. But when answering this part from the viewpoint of computation, the answer would change to *"as soon as possible to the fastest computer available"*. And when answering this part from the viewpoint of cost for moving services, the answer would be *"as late as possible to the cheapest location"*. Cost functions, a mathematical concept used to solve optimisation problems [338-340], could be an answer to the *"when"* and *"where"* part of the question. Depending on the cost function and parameters set, the answer could be one of the above, or any other.
The authors of [341] list requirements that automatic deployment needs to tackle for Enterprise Java Beans and CORBA Components. They came to the result that a minimal requirement is that a hosting environment for components, i.e. a service execution environment, needs to be present on every device where a component will be deployed to. However, the work presents no answers to the above question. A cost function for deployment of service logic in networks is presented in [342]. The cost function is part of a service creation environment for creation of distributed network services. The service creation environment assumes that services are stateless and there is no

request for deployment while services are running. Another work that uses cost functions for planning of deployment is presented in [343]. Yet, this work did not analyse the dependency between the number and quality of resources and their impact on the efficiency of their cost function. The CASCADAS project [313] is going to define cost functions for the deployment and movement of services in autonomic communication systems, which might also be used for ubiquitous computing.

6.5 Service validation

Validation is the process of checking if something satisfies certain criteria. Validation refers to the process of controlling that data inserted into software satisfies pre-determined formats or complies with stated length and character requirements and other defined input criteria, and also refers to the process of ensuring that software is allowed to do something.
Service validation is important in middleware based on the concept of service oriented architecture and that supports reconfigurability and logic mobility of services. As services integrate into a middleware, it needs to be ensured that the integrating service is valid, i.e. complies with the criteria of the middleware. Such criteria are the policies setup for resource use, communication, security, data handling, etc.
There are different forms of service validation, among them are:

- validating the correctness of specifications and algorithms implemented in services
- validating the correctness of data exchange between services
- validating the correctness of service compositions
- validating the correctness of services itself

How the correctness of specifications and algorithms, implemented in services, can be validated is presented in [344-346]. The authors of [344] present a development methodology for service creation for multimedia telecommunication services and introduce a validation cycle after every development step. This validation cycle reverses the development step to check whether the result will lead to the starting conditions or not. Their approach is, however, only applicable to development steps that can be unambiguously reversed. If during reversing the development step and ambiguities may result, then their approach fails. The authors of [345] create an accessibility graph that contains all possibilities of how to use a service. The arcs of the graph represent state changes of the service and indicate how a service may be used after its last use, i.e. which operations of the service are allowed to be invoked after the previous invocation. This graph is then compared with the specification of the service. Identified mismatches can be located and fixed by examining the created accessibility graph. A mismatch is an operation that is not executable but should be. In [346] an approach for service validation is done by formulating properties that services should satisfy (i.e. desired behaviour) and that they should not violate (i.e. undesired behaviour). The properties, i.e. the behaviour of services, is described in a meta-language and validated by an agent that compares the described properties with the current behaviour of services. Similar work is carried out for services in autonomic communication in the area of pervasive supervision, which inspects services and compares the behaviour of the service with a plan that defines its expected behaviour [347-350].
Validating the correctness of data exchange between services is the primary objective of design by contract (DbC). Motivations, related work in this area, and an own approach for runtime validation of contracts can be found in Chapter 5 of this dissertation. Another work in this area is presented in [351]. They extend IDL by the possibility to formulate behavioural contracts. These contracts are then evaluated at runtime of the services.
The authors of [352] and [353] focus on the correctness of service composition. In [353], a formal model to verify, that is to proof, that service compositions are valid, is introduced. For this purpose the interfaces of dependant services are compared with the implementation of depending services. The comparison is based on a formal notion that can be evaluated using mathematical expressions. The evaluation is supported by tools and applicable at runtime of services, including compositions of services formed at runtime of middleware. The authors of [352] describe a similar approach

based on aspect oriented programming to formulate policies for the behaviour of services. Aspects are weaved into composed services and executed. Their approach is demonstrated for *Enterprise JavaBeans*. Further work is required to check if the idea of using aspects to formulate policies can be generalised to be useful for other frameworks.
The validation of correctness of services itself includes checking the integrity of services. This is particularly important for logical mobility, because services moving from one device to another may claim to be something which they are not. Using checksums, certificates, etc. provides a mechanism to ensure that the implementation of a services matches its description, and allows users to judge whether a service is what it claims to be or not. Usually, setup software used for deployment includes such functionality [332].
What is missing is a service validation process encompassing the different forms of service validation, i.e. validating the correctness of specifications, algorithms, data exchange, compositions, and the services itself. Future research activities may seek to integrate existing solutions for the single forms of validation, or to create new forms of service validation.

6.6 Summary

In addition to the capabilities of FAME², this chapter proposes five topics to improve and extend its capabilities and to make it more convenient and usable for middleware developers:

- Service life cycle and dependency resolution
- Semantic service description and modelling
- Service discovery optimisations
- Service deployment
- Service validation

Automated service life cycle and dependency resolution simplifies the administration of middleware. Dependency resolution and automated service life cycle would be implemented in the LCM of FAME². As the LCM is separated from services, its modification has no negative influence on existing services.
A semantic service description may be used for new concepts of semantic service discovery, service choreography, and service interaction modelling. The repository of FAME² supports different types of service descriptions. Features like semantic service discovery or service choreography can be implemented as FAME² services.
OSDI abstracts service discovery solutions. These solutions are implemented as services. The implementation of the OSDI is responsible to publish service discovery queries in an event channel, and collect all responses. More smart solutions, where the OSDI forwards the service discovery queries to selected service discovery solutions only, might improve the performance of service discovery in FAME².
Logical mobility, which includes service deployment, is a possibility to optimise resource use, quality of service, and to optimise communication between services. The question of service deployment is *"Who deploys what when to where?"* Especially in ubiquitous computing the answer to this question is not trivial, because usually users should not be bothered with decisions of when and where something should be deployed. Instead, ideally computers are able to answer this question. However, as it is pointed out, there are different possible answers, some of them are conflicting.
Service validation ensures the correctness of services, service compositions, data exchange between services, and algorithms implemented in services. Service validation is an important feature in middleware supporting logical mobility and reconfigurability, as any middleware based on FAME² does. With service validation, the quality and robustness of middleware based on FAME² is greatly enhanced.
For each of the identified topics, relevant work exists. However, this existing work has to be adapted to fit into FAME² and ubiquitous computing.

7 Conclusion

The focus of this dissertation is on middleware and enabling technologies to be used in distributed computing systems for ubiquitous computing. The dissertation contributes concepts and ideas for developing middleware for these systems.
The vision of ubiquitous computing foresees computers of manifold type seamlessly integrate and support the user in the *background*, meaning that computer systems do not distract the user in his everyday activities. A characteristic of distributed computing systems for ubiquitous computing is the heterogeneity of computers, reaching from small scale devices embedded into coffee cups and carpets, over mobile devices like cellular phones, personal digital assistants and notebook computers, to computers with high performance like servers, mainframes, etc. All these devices interact together to provide services to the users. The heterogeneity is manifested in different operating systems, network protocols, etc., too. Middleware is a software technology concept used to abstract from such heterogeneity and providing a homogeneous layer that supports software implementing the services for the users.

Chapter 2 concentrates on the review, analysis, and finally the creation of a framework that supports development of middleware that will be executed on mobile devices, like mobile phones and personal digital assistants. The characteristics of mobile devices are heterogeneity, limited resources, and wireless connectivity. Different programming models and software technology concepts for the realisation of middleware are described. They are evaluated for their suitability to be used in middleware for ubiquitous computing. Following on from that, existing frameworks and technologies for middleware development are reviewed and their characteristics are compared with the previous description of programming models and software technology concepts, and the identified characteristics of mobile devices. From this comparison it can be seen that RUNES and Web Services are the most promising candidates to implement middleware for distributed computing systems in ubiquitous computing. However, RUNES is in its early specification phase, and it is unclear how its realisation will look like. Web Services are, concerning ubiquitous computing, premature. Various solutions, like WS-Policy, code mobility, etc. are under active research and no standard is available now. This state-of-the-art comparison still motivates the need for complementary work.
The *Framework for Applications in Mobile Environments 2* (FAME²) is specifically designed for developing middleware that is able to be executed on mobile devices. It includes a development process to implement middleware services, and service execution environments (SEEs) that host (i.e. execute) the middleware services. The objective of the development process is to separate the concerns of developers of middleware services, and of developers of SEEs, which is often mixed today, resulting in unnecessary interdependencies between middleware services and their SEEs. FAME² is the first solution to implement middleware for distributed computing systems in ubiquitous computing with all the following features:

- minimal resource use
- flexibility by enabling reconfiguration of middleware and its services at runtime
- a security concept on three different levels of access control: middleware, service, and operation
- an open interface to utilise virtually any existing service discovery that is used in middleware today
- online-update functionality of middleware services, including services that are in use

Service discovery is revisited and handled in detail in Chapter 3. As service discovery is a fundamental technology required for middleware used in distributed computing systems for ubiquitous computing, a review of the state of the art leads to the conclusion that there is a demand for an integration of already existing and future service discovery solutions. Existing solutions for service discovery integration are discussed and their shortcomings are identified. The *Support for*

Service Discovery and Interaction (SSDI) integrates service discovery solutions as plug-ins and clients use service discovery by formulating their queries in a proprietary query language. The shortcomings of SSDI are shown by the requirement to use a proprietary query language, and the missing ability of a direct interaction with the service discovery plug-ins. As a consequence, future service discovery solutions cannot be integrated because the proprietary query language of SSDI does not reflect new features introduced by these future service discovery solutions. The *Open Service Discovery Architecture* (OSDA) creates an overlay on top of different service discovery solutions. Clients query this overlay network with a proprietary query language, and the results are also in a proprietary format. As a consequence, OSDA suffers from similar shortcomings as SSDI do. The *Reflective Middleware for Mobile Computing* (ReMMoC) contains a *Service Discovery Framework* that integrates different service discovery solutions. ReMMoC allows clients to directly interact with the service discovery solutions it integrates, enabling integration of future service discovery solutions. However, ReMMoC requires a modification of the middleware's implementation and a restart upon integration of a new service discovery solution. Such requirement is undesirable for middleware in distributed computing systems for ubiquitous computing, because integration of a new service discovery solution requires human interaction. This violates the idea of unobtrusive user support.
The *Open Service Discovery Interface* (OSDI) incorporates some of the concepts used in existing solutions for integrating service discovery solutions. It is the first solution that enables continuous operation during integration of future service discovery solutions, free choice of query languages, and provides purposeful results to clients using service discovery, i.e. a link to the services they are looking for. Furthermore, the design of the OSDI enables integration of filters and translators. Filters limit the set of discovered services, e.g. by security or cost policies. Translators translate a service discovery query from one language into another, increasing the chances of discovering the searched service.

Chapter 4 points out why the proxy structural design pattern is the only convenient solution for online-update. The reasons are:

- no modification of software that may be updated necessary
- no modification of software that uses software that may be updated necessary
- no modification of the platform, i.e. hardware, operating system, etc., that runs software that may be updated necessary
- no modification of programming languages necessary
- no modification of established communication links between interacting software necessary
- no requirement to use twice as much resources, e.g. physical disk space, memory, CPU, etc., as it would be required by the software that may be updated
- no requirement to resolve any ambiguities of service interfaces upon (semi-) automatic service composition

Yet, the proxy pattern fails when event oriented communication in the form of publish/subscribe is used. Publish/subscribe enables communication between loosely coupled software. Software publishes events in publish/subscribe systems and other software subscribes to them. This subscription process requires that publishers have links to their subscribers, which are unique and immutable, so they can publish the events to their subscribers. However, the proxy pattern requires that software does not have direct links to other software, but instead lets the proxy manage links.
The novel concept of mutable reference endpoints solves this incompatibility issue between the proxy structural design pattern and publish/subscribe. Mutable references are endpoints of links to software elements like objects and behave identical to the endpoints of unique and immutable links, but support changing the endpoint, i.e. the software element they point to. The concept of mutable reference endpoints is integral part of the FAME² SEE. As a result, all services developed with the framework of FAME² are able to utilise the proxy pattern and publish/subscribe without worrying about compatibility.

Chapter 5 presents the *Extensible Constraint Framework* (ECF), which allows developers to define contracts on services. Loosely coupled software systems, composed from software elements like objects, components or services, tend to be composed in an incompatible or suboptimal manner resulting in a decreased quality of the resulting system. Thus, it is necessary to ensure that software elements are compatible with each other when they are composed. *Design by Contract* (DbC) is a recognised technology to increase quality of software systems comprising of different software elements. Pre- and post-conditions and invariants are used to formulate contracts that specify the ensured behaviour of software. As long as all pre-conditions hold, the results will adhere to the post-conditions, and the contracted software element will never violate its invariant, i.e. enter any other state than the ones specified in the invariant. Existing approaches often require software developers to learn new languages, learn and use new tools, or use a modified platform. Other existing approaches do not support negotiation and refinement of contracts after software is deployed, i.e. installed and running.
With the Extensible Constraint Framework, developers define pre- and post-conditions on service operations. The defined contracts are negotiable and refinable. Furthermore, the supported constracts include behavioural, synchronisation, and quality of service contracts. Behavioural contracts are used to specify the data exchange between software elements. Synchronisation contracts specify the parallel execution of software elements, for example in distributed computing systems. Quality of service contracts control the operation of software depending on available resources and given situations, for example reducing the number of calculations executed per second when software is running on battery driven mobile devices, and increasing this number when there is an emergency. ECF is the only solution that supports all these features without the requirement to learn new languages, use new tools, or use modified platforms.

Chapter 6 suggests ideas for possible future work to increase the usability of FAME² and its realisations of enabling technologies, i.e. life cycle of middleware services, service description and modelling, discovery, deployment, and validation. Solutions in these areas will increase the quality and robustness of middleware for distributed computing systems in ubiquitous computing. Research results may also be feed back into the general middleware development that is based on the concept of service oriented architecture.

During the time of carrying out this dissertation, it could be observed that the ideas driving FAME² are also becoming more and more important for industry. Service oriented architecture is one of the major architectural concepts today. Software executed on mobile computers, which connect and interact spontaneously, is a clear trend supported by existing and new wireless network technologies, e.g. Bluetooth, Zigbee, Near-Field-Communication, etc. Activities in the area of Web Services are another indicator. New technologies for Web Services are designed to:

- support different existing solutions, e.g. for service discovery
- integrate different existing solutions, e.g. Web Services Discovery Architecture
- host Web Services on mobile computers, e.g. the lightweight SOAP server

Yet, FAME² is the only solution that features a service execution environment supporting reconfigurability, minimal resource use, a security concept on different levels of access control, a truly open and flexible integration of service discovery solutions, and online-update capability.

8 References

[1] **Philip A. Bernstein:** *Middleware: a model for distributed system services*. In Communications of the ACM, Vol. 39, No. 2, February 1996, pp. 86-98

[2] **Mark Weiser:** *The Computer for the Twenty-First Century*. In Scientific American, September 1991, pp. 94-101, available at: http://www.ubiq.com/hypertext/weiser/SciAmDraft3.html (July 2005), republished in IEEE Pervasive Computing, Vol.1, No.1, January-March 2002, pp. 19-25

[3] **K. Ducatel, M. Bogdanowicz, F. Scapolo, J. Leijten, and J.-C. Burgelman:** *Scenarios for Ambient Intelligence in 2010*. Information Society Technologies Advisory Group, February 2001, available at: ftp://ftp.cordis.lu/pub/ist/docs/istagscenarios2010.pdf (July 2005)

[4] **Andrew S. Tanenbaum and Marteen van Steen:** *Distributed Systems: Principles and Paradigms*. Prentice-Hall Inc., Upper Saddle River, NJ, 2002, ISBN: 0131217860

[5] **Olaf Drögehorn, Olivier Coutand, and Klaus David:** *Platform for service driven networks: Using a general architectural approach*. In Proceedings of the 3rd International Conference on Networking (ICN'04), Guadeloupe, French Caribbean, Feb. 29 - March 4, 2004

[6] **Stefan Arbanowski, Pieter Ballon, Klaus David, Olaf Drögehorn, H. Eertink, Wolfgang Kellerer, H. van Kranenburg, Kimmo Raatikainen, and Radu Popescu-Zeletin:** *I-centric Communications: Personalization, Ambient Awareness, and Adaptability for Future Mobile Services*. In IEEE Communications Magazine, September 2004

[7] **R. Tafazolli (ed.):** *Technologies for the Wireless Future*. WWRF, Wiley, 2005

[8] **Stefan Arbanowski, Stefan Steglich, Olaf Drögehorn, and Ioannis Fikouras:** *Generic Service Elements Advanced Service Creation for 3GB Systems*. In Proceedings of the WPMC'03, Yokosuka, Japan, pp. 53-56, 2003

[9] The Wireless World Research Forum Homepage: http://www.wireless-world-research.org/ (July 2005)

[10] **Christian Becker, Gregor Schiele, Holger Gubbels, and Kurt Rothermel:** *BASE – A Micro-broker-based Middeware For Pervasive Computing*. In Proceedings of the 1st IEEE International Conference on Pervasive Computing and Communications (PerCom 03), pp. 443-451, March 23-26, Fort Worth, USA, 2003

[11] **C. Dabrowski and K. Mills:** *Understanding Self-Healing in Service-Discovery Systems*. In Proceedings of the ACM Workshop on Self-Healing Systems, Charleston, South Carolina, United States, November 2002

[12] **Adrian Friday, Nigel Davies, and Elaine Catterall:** *Supporting Service Discovery, Querying and Interaction in Ubiquitous Computing Environments*. In Proceedings of the 2nd ACM International Workshop on Data Engineering for Wireless and Mobile Access, Santa Barbara, California, United States, 2001

[13] **Adrian Friday, Nigel Davies, Nat Wallbank, Elaine Catterall, and Steve Pink:** *Supporting service discovery, querying and interaction in ubiquitous computing environments*. In ACM Baltzer Wireless Networks (WINET) Special Issue, vol. 10, no. 6, 2004

[14] **Noura Limam, Joanna Ziembicki, Reaz Ahmed, Youssef Iraqi, Dennis Tianshu Li, Raouf Boutaba, and Fernando Cuervo:** *OSDA: Open Service Discovery Architecture for Cross-domain Service Discovery*. In Computer Communications Journal, Special Issue on Emerging Middleware for Next Generation Networks, 2005

[15] **Paul Grace:** *Overcoming Middleware Heterogeneity in Mobile Computing Applications*. Ph.D. Thesis, Lancaster University, March 2004, available at: http://www.lancs.ac.uk/postgrad/gracep/thesisfinal.pdf (July 2005)

[16] **Erich Gamma, Richard Helm, John Vlissides, and Ralph Johnson:** *Design Patterns: Elements of Reusable Object Oriented Software*. Addison Wesley Longman, Inc., October 1994, ISBN: 0201633612

[17] **Gísli Hjálmtýsson and Robert Gray:** *Dynamic C++ Classes: A lightweight mechanism to update code in a running program*. In Proceedings of the USENIX Annual Technical Conference, New Orleans, Louisiana, USA, June 1998, pp. 65-76

[18] **Alessandro Orso, Anup Rao, and Mary Jean Harrold:** *A Technique for Dynamic Updating of Java Software*. In Proceedings of the 18th IEEE International Conference on Software Maintenance (ICSM'02), 2002, pp. 649-658

[19] **Hans Werner Pohl and Jens Gerlach:** *Using the Bridge Design Pattern for OSGi Service Update*. In Proceedings of the 8th European Conference on Pattern Languages of Programs, Irsee, Germany, June 2003

[20] **Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec:** *The Many Faces of Publish/Subscribe*. In ACM Computer Surveys, Vol. 35, No. 2, June 2003, pp. 114-131

[21] **Urs Hölzle:** *Integrating Independently-Developed Components in Object-Oriented Languages*. In Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP '93), Kaiserslautern, Germany, 1993

[22] **Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins:** *Making Components Contract Aware*. In IEEE Computer, Vol. 32, No. 7, July 1999, pp. 38-45

[23] **Yi Liu and H. Conrad Cunningham:** *Software Component Specification Using Design by Contract*. In Proceedings of the SoutEast Software Engineering Conference, Tennessee Valley Chapter, National Defense Industry Association, Huntsville, Alabama, USA, April 2002

[24] **Bertrand Meyer:** *A framework for providing contract-equipped classes*. In Proceedings of the 10th International Workshop on Abstract State Machines – Advances in Theory and Applications, Egon Boerger, Angelo Gargantini, and Elvinia Riccobene (Eds.), Springer-Verlag, Taormina, Italy, March 2003

[25] **Bertram Meyer:** *Applying "Design by Contract"*. In IEEE Computer, Vol. 25, No. 10, October 1992, pp. 40-51

[26] **Bertrand Meyer:** *Design by Contract: The Eiffel Method*. In Proceedings of the Technology of Object-Oriented Languages and Systems, IEEE Computer Society, Washington, Dacota, USA, 1998, pp. 446ff

[27] **Rebecca Parsons:** *Components and the World of Chaos*. In IEEE Software, May/June 2003, pp.83-85

[28] **Hugo Haas and Allen Brown:** *Web Service Glossary*. W3C Working Group Note, 11th February 2004, available at: http://www.w3.org/TR/ws-gloss/ (July 2005)

[29] **Roy Thomas Fielding:** *Architectural Styles and the Design of Network-based Software Architectures*. Dissertation, University of Carolina, Irvine, USA, 2000, available at: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[30] The Common Component Architecture Forum, Terms and Definitions, available at: http://www.cca-forum.org/glossary/index.html (July 2005)

[31] **Eric Armstrong, Jennifer Ball, Stephanie Bodoff, Debbie Bode Carson, Ian Evans, Dale Green, Kim Haase, and Eric Jendrock:** *The J2EE 1.4 Tutorial*. For Sun Java System Application Server Platform Edition 8.1 2005Q1, 16th December 2004, available at: http://java.sun.com/j2ee/1.4/docs/tutorial/doc/ (July 2005)

[32] International Technology Education Association, Awareness Campaigns for Technology, Glossary of Technological Terms, available at: http://www.iteawww.org/ACT/ACT_Pages/ACT_Glossary.html#S (July 2005)

[33] **Ali Arsanjani, Bernhard Borges and Kerrie Holley:** *How to Execute a Sound SOA Design Technique*. Online article, Thursday 14th October 2004, available at: http://www.webservices.org/index.php/ws/content/view/full/44989

[34] **Mark Endrei, Jenny Ang, Ali Arsanjani, Sook Chua, Philippe Comte, PŒl Krogdahl, Dr Min Luo, and Tony Newling:** *Patterns: Service-Oriented Architecture and Web Services*. IBM Redbooks, 2004, available at: http://www.redbooks.ibm.com/redbooks/SG246303/wwhelp/wwhimpl/js/html/wwhelp.htm (July 2005)

[35] **Paul Allen:** *Component-based Development for Enterprise Systems*. Cambridge University Press, 1998, ISBN 0521649994

[36] Sun Microsystems, JiniTM Architecture Specification, white paper, June 2003, available at http://wwws.sun.com/software/jini/specs/jini2_0.pdf (July 2005)

[37] **Jürgen Sellentin:** *Konzepte und Techniken der Datenversorgung für komponentenbasierte Informationssysteme*. Dissertation, Universität Stuttgart, 9th November 1999, available at: http://elib.uni-stuttgart.de/opus/volltexte/2000/612/pdf/diss.pdf (July 2005)

[38] **Clemens Szyperski:** *Component Software Beyond Object-Oriented Programming*. Addison-Wesley Professional, 19th December 1998, ISBN: 0201178885

[39] **Mahadev Satyanarayanan:** *Mobile Computing: Where's the Tofu?*. In ACM SIGMOBILE Mobile Computing and Communications Review, Vol. 1, No. 1, April 1997, pp. 17-21

[40] **Nobert Diehl and Albert Held:** *Mobile Computing*. International Thompson Publishing, Bonn, 1995, ISBN 3-929821-80-X

[41] **Mark Weiser:** *Ubiquitous Computing*. http://www.ubiq.com/hypertext/weiser/UbiHome.html (July 2005)

[42] **Mahadev Satyanarayanan:** *A Catalyst for Mobile and Ubiquitous Computing*. In IEEE Pervasive Computing, Vol.1, No.1, January-March 2002, pp. 2-5

[43] **Wolfgang Emmerich:** *Software engineering and middleware: a roadmap*. In Proceedings of the Conference on The Future of Software Engineering, Anthony Finkelstein (Ed.), ACM Press, 2000, pp. 117-129

[44] **Sacha Krakowiak:** *What is Middleware*. ObjectWeb Open Source Middleware, unpublished, available at: http://middleware.objectweb.org/ (July 2005)

[45] **Paul Grace, Gordon S. Blair, and Sam Samuel:** *A reflective framework for discovery and interaction in heterogeneous mobile environments*. In ACM SIGMOBILE Mobile Computing and Communications Review, Vol. 9, No. 1, January 2005, pp. 2-14, available at: http://doi.acm.org/10.1145/1055959.1055962 (July 2005)

[46] **Licia Capra, Gordon S. Blair, Cecilia Mascolo, Wolfgang Emmerich, and Paul Grace:** *Exploiting Reflection in Mobile Computing Middleware*. In ACM SIGMOBILE Mobile Computing and Communications Review, Vol. 6, No. 4, October 2002, pp.34-44

[47] **Kimmo Raatikainen, Henrik Bærbak Christensen, and Tatsuo Nakajima:** *Application Requirements for Middleware for Mobile and Pervasive Systems*. In ACM SIGMOBILE Mobile Computing and Communications Review, Vol. 6, No. 4, October 2002, pp. 16-24

[48] **Nigel Davis and Hans-Werner Gellersen:** *Beyond Prototypes, Challenges in Deploying Ubiquitous Systems*. In IEEE Pervasive Computing, Vol.1, No.1, IEEE 2002, pp.26-35

[49] **J. Flinn, D. Narayanan, and Mahadev Satyanarayanan:** *Self-Tuned Remote Execution for Pervasive Computing*. In Proceedings of the 8th IEEE HotOs Conference, Elmau/Oberbayern, Germany, May 2001

[50] **J. Flinn and Mahadev Satyanarayanan:** *Energy-aware adaptation for mobile applications*. In Proceedings of the 17th ACM Symposium on Operating Systems Principles, Kiawah Island Resort, SC, December 1999

[51] **Denis Conan, Sophie Chabridon, and Guy Bernard:** *Disconnected Operations in Mobile Environments*. In Proceedings of the 16th International Parallel and Distributed Processing Symposium, IEEE Computer Society, Washington, DC, USA, 2002, pp. 118-125, ISBN: 0-7695-1573-8

[52] **Kurt Geihs:** *Middleware Challenges Ahead*. In IEEE Computer, June 2001, pp.24-31

[53] **Abdulbaset Gaddah and Thomas Kunz:** *A Survey of Middleware Paradigms for Mobile Computing*. Technical Report SCE-06-16, Carleton University, Systems and Computer Engineering, July 2003, available at: http://www.sce.carleton.ca/wmc/middleware/middleware.pdf (July 2005)

[54] **Richard M. Adler:** *Distributed Coordination Models for Client/Server Computing*. In IEEE Computer, April 1995, pp. 14-22

[55] **Scott M. Lewandowski:** *Frameworks for Component-Based Client/Server Computing*. In ACM Computing Surveys, Vol.30, No.1, March 1998, ACM Press, New York, NY, USA, pp. 3-27

[56] **Jin Jing, Abdelsalam Sumi Helal, and Ahmed Elmagarmid:** *Client-server computing in mobile environments*. In ACM Computing Surveys, Vol. 31, No. 2, June 1999, pp. 117-157

[57] **Günther Bengel:** *Verteilte Systeme*. 3. Auflage, Vieweg & Sohn Verlag, Wiesbaden, Germany, 2004

[58] **Dejan S. Milojicic, Vana Kalogeraki, Rajan Lukose, Kiran Nagaraja, Jim Pruyne, Bruno Richard, Sami Rollins, and Zhichen Xu:** *Peer-to-Peer Computing*. HP Laboratories Palo Alto, HPL-2002-57 (R.1), July 2003, available at: http://web.cs.wpi.edu/~goos/Teach/cs4513-d05/papers/p2p-tutorial.pdf (July 2005)

[59] Gnutella Protocol Development, available at: http://www.the-gdf.org/wiki/index.php?title=Main_Page (July 2005)

[60] **Matei Ripeanu:** *Peer-to-Peer Architecture Case Study: Gnutella Network*. In Proceedings of the IEEE 1st International Conference on Peer-to-Peer Computing, Sweden, August 2001

[61] **Mark Weiser:** *The future of ubiquitous computing on campus*. In Communications of the ACM, Vol. 41, No. 1, January 1998, pp. 41-42

[62] **Gerd Kortuem, Jay Schneider, Dustin Preuitt, Thaddeus G.C. Thompson, Stephen Fickas, and Zary Segall:** *When peer-to-peer comes face-to-face: collaborative peer-to-peer computing in mobile ad-hoc networks*. In Proceedings of the IEEE 1st International Conference on Peer-to-Peer Computing, Sweden, August 2001, pp. 75-91

[63] **Thilo Kielmann:** *Designing a Coordination Model for Open Systems*. In Proceedings of the 1st International Conference on Coordination Languages and Models, Lecture Notes In Computer Science, Vol. 1061, 1996, pp. 267-284

[64] **Flaviu Cristian:** *Synchronous and asynchronous*. In Communications of the ACM, Vol. 39, No. 4, April 1996, pp. 88-97

[65] **Andrew D. Birrell and Bruce Jay Nelson:** *Implementing remote procedure calls*. In ACM Transactions on Computer Systems, Vol.2, No.1, ACM Press, 1984, pp. 39-59

[66] **George Coulouris, Jean Dollimore, and Tim Kindberg:** *Distributed Systems: Concepts and Design*. 3. Edition, Pearson Education, Harlow, Essex, England, 2001

[67] **Kohei Honda and Mario Tokoro:** *An Object Calculus for Asynchronous Communication*. In Lecture Notes In Computer Science, Vol. 512, also published In Proceedings of the European Conference on Object-Oriented Programming, 1991, pp. 133-147

[68] **Maarten van Steen, Andrew S. Tanenbaum, Ihor Kuz, and Henk J. Sips:** *A Scalable Middleware Solution for Advanced Wide-Area Web Services*. In Distributed Systems Engineering, Vol.6, No.1, March 1999, pp.34-42

[69] **Philip Homburg, Leendert van Doorn, Maarten van Steen, Andrew S. Tanenbaum, and Wiebren de Jonge:** *An Object Model for Flexible Distributed Systems*. In Proceedings of the 1st ASCI Conference, ASCI, Heijen, The Netherlands, May 1995, 69-78

[70] **Giacomo Cabri, Letizia Leonardi, and Franco Zambonelle:** *Coordination Models for Internet Applications based on Mobile Agents*. In IEEE Computer, 1999

[71] **David Gelernter:** *Generative Communication in Linda*. In ACM Transactions on Programming Languages and Systems (TOPLAS), Vol.7, No.1, January 1985, ACM Press, New York, NY, USA, pp. 80-112

[72] **David Gelernter, Nicholas Carriero, Sarat Chandran, and Silva Chang:** *Parallel Programming in Linda*. In Proceedings of the International Conference on Parallel Processing, St. Charles, Illinois, August 1985, IEEE, pp. 255-263

[73] **Sudhir Ahuja, Nicholas Carriero, and David Gelernter:** *Linda and friends*. In IEEE Computer, Vol.19, No.8, IEEE Computer Society Press, Los Alamitos, CA, USA, 1986, pp. 26-34

[74] **Gian Pietro Picco, Amy L. Murphy, and Gruia-Catalin Roman:** *LIME: Linda meets mobility*. In Proceedings of the 21st International Conference on Software Engineering, Los Angeles, California, United States, IEEE Computer Society Press, Los Alamitos, CA, USA, 1999, ISBN: 1-58113-074-0, pp. 368-377

[75] Sun Microsystems, JavaSpaces Service Specification, Version 1.2.1, April 2002, available at: http://www.sun.com/software/jini/specs/jini1.2html/js-title.html (July 2005)

[76] **Eric Freeman, Susanne Hupfer, and Ken Arnold:** *JavaSpaces Principles, Patterns, and Practice*. Pearson Education, June 1999, available at: http://java.sun.com/developer/Books/JavaSpaces/ (July 2005)

[77] **Marc Shapiro, Antony Rowstron, and Anne-Marie Kermarrec:** *Application-independent reconciliation for nomadic applications*. In Proceedings of the 9th workshop on ACM SIGOPS European workshop: beyond the PC: new challenges for the operating system, Kolding, Denmark, 2000

[78] **R. Nobili and U. Pesavento:** *John von Neumann's Automata Revisited*. In Artificial Worlds and Urban Studies, E.Besussi and A.Cecchini Ed.s, DAEST Publication, Convegni 1, Venezia, November 1994, available at: http://www.pd.infn.it/%7Ernobili/pdf_files/jvnconstr.pdf (July 2005)

[79] **Jim Waldo:** *Mobile Code, Distributed Computing, and Agents*. In IEEE Intelligent Systems, March/April 2001, pp. 10-12

[80] **L. Cardelli and A. D. Gordon:** *Mobile ambients*. In Foundations of Software Science and Computational Structures, M. Nivat (ed.), LNCS 1378, pp. 140-155. Springer-Verlag, 1998

[81] **A. Fuggetta, G. P. Picco, and G. Vigna:** *Understanding code mobility*. In IEEE Transactions on Software Engineering, Vol. 24, No. 5, pp. 342-361, May 1998

[82] **G.C. Roman, G.P.Picco, and A. L. Murphy:** *Software Engineering for Mobility: A Roadmap*. The Future of Software Engineering, Anthony Finkelstein (Ed.), ACM Press 2000

[83] **V. Grassi, R. Mirandola, and A. Sabetta:** *A UML Profile to Model Mobile systems*. In Proceedings of UML 2004 (LNCS 3273), 11-15 October 2004, Lisbon, Portugal

[84] **Philippe B. Kruchten:** *4+1 view model of architecture*. In IEEE Software, Vol. 12, No. 6, 1995, pp. 42-50

[85] **Mike P. Papazoglou:** *Service-Oriented Computing: Concepts, Characteristics and Directions*. In Proceedings of the 4th International Conference on Web Information Systems Engineering, IEEE Computer Society, Washington, DC, USA, 2003

[86] **K. Birman and T. Joseph:** *Reliable Communication in the Presence of Failures*. In ACM Transactions on Computer Systems, 5(1), pp. 47–76, February 1987

[87] **K. Birman, A. Schiper, and P. Stephenson:** *Lightweight Causal and Atomic Group Multicast*. In ACM Transactions on Computer Systems, 9(3), pp. 272–314, August 1991

[88] **C. Malloth:** *Conception and Implementation of a Toolkit for Building Fault-Tolerant Distributed Applications in Large Scale Networks*. PhD thesis, Federal Institute of Technology, Lausanne (EPFL), 1996.

[89] **Marco Conti, Gaia Maselli, Giovanni Turi, and Silvia Giordano:** *Cross-Layering in Mobile Ad Hoc Network Design*. In IEEE Computer, February 2004, pp. 48-51

[90] **Hubert Zimmermann:** *OSI Reference Model – The ISO Model for Architecture for Open Systems Interconnection*. In IEEE Transactions on Mobile Computing, Vol. 28, No. 4, April, 1980, pp. 425-432

[91] **Jeffrey Hightower, Barry Brumitt, and Gaetano Borriello:** *The Location Stack: A Layered Model for Location in Ubiquitous Computing*. In Proceedings of the 4th IEEE Workshop on Mobile Computing Systems and Applications, IEEE Computer Society, Washington, DC, USA, 2002, pp. 22-28

[92] **David Garlan, D. Siewiorek, A. Smailagic, and Peer Steenkiste:** *Project Aura: Toward Distraction-Free Pervasive Computing*. In IEEE Pervasive Computing, April-June 2002, pp.22-31

[93] **J. P. Sousa and D. Garlan:** *Aura: an Architectural Framework for User Mobility in Ubiquitous Computing Environments*. In Proceedings of 3rd Workshop IEEE/IFIP Conference on Software Architecture, Montreal, Canada, 2002, Kluwer Academic Publishers, August 25-31, 2002. pp. 29-43

[94] **Rakesh Agrawal, Roberto J. Bayardo Jr., Daniel Gruhl, and Spiros Papadimitriou:** *Vinci: A Service-Oriented Architecture for Rapid Development of Web Applications*. In Proceedings of the 10th International Conference on World Wide Web, Hong Kong, Hong Kong, 2001, pp.355-365

[95] **Luciano Baresi, Reiko Heckel, Sebastian Thöne, and Dániel Varró:** *Modeling and Validation of Service-Oriented Architectures: Application vs. Style*. In Proceedings of the 9th European software engineering conference, Helsinki, Finland, 2003, pp.68-77

[96] **Michael Champion, Chris Ferris, Eric Newcomer, and David Orchard:** *Web Service Architecture*. W3C Working Draft, 2002, available at: http://www.w3.org/TR/2002/WD-ws-arch-20021114/ (July 2005)

[97] **M. Colombo, E. Di Nitto, M. Di Penta, D. Distante, and M. Zuccalà:** *Speaking a Common Language: A Conceptual Model for Describing Service-Oriented Systems*. In Proceedings of International Conference on Service-Oriented Computing (ICSOC 05), Amsterdam (NL), December 2005, LNCS 3826, pp. 50–62

[98] **A. M. Sassen and C. Macmillan:** *The service engineering area: An overview of its current state and a vision of its future*. European Commission, Directorate D – Network and Communication Technologies, Software Technologies, 2005, available at: ftp://ftp.cordis.lu/pub/ist/docs/directorate_d/st-ds/sota_v1-0.pdf (July 2005)

[99] **Paolo Bellavista, Antonio Corradi, Rebecca Montanari, and Cesare Stefanelli:** *Dynamic Binding in Mobile Applications: A Middleware Approach*. In IEEE Pervasive Computing, Vol.2, No.2, 2003, pp.34-42

[100] **Neil D. Jones and Steven S. Muchnick:** *Binding time optimization in programming languages: Some thoughts toward the design of an ideal language*. In Proceedings of the 3rd ACM SIGACT-SIGPLAN Annual Symposium on Principles of Programming Languages, Atlanta, Georgia, ACM Press, New York, NY, USA, 1976, pp. 77-94

[101] **Bjarne Stroustrup:** *The C++ Programming Language*. Special Edition, Addison-Wesley Pub. Co., May 2000, ISBN 0-201-70073-5

[102] **Tim Lindholm and Frank Yellin:** *The Java™ Virtual Machine Specification*. Second Edition, Addison-Wesley Professional, 1999, ISBN: 0201432943

[103] **Bill Joy, Guy Steele, James Gosling, and Gilad Bracha:** *Java™ Language Specification*. Second Edition, Addison-Wesley Professional, 2000, ISBN: 0201310082

[104] **George A. Papadopoulos and Farhad Arbab:** *Coordination Models and Languages*. Technical Report: SEN-R9834, CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands, 1998, available at: http://scholar.google.com/url?sa=U&q=http://www2.cs.ucy.ac.cy/~george/AdvComp.pdf (July 2005)

[105] **David Gelernter and Nicholas Carriero:** *Coordination Languages and their Significance*. In Communication of the ACM, Vol. 35, No. 2, February 1992, pp. 96-107

[106] **René Meier:** *Communication Paradigms for Mobile Computing*. In ACM SIGMOBILE Mobile Computing and Communications Review, Vol.6, No.4, October 2002, pp. 56-58

[107] **A. D. Joseph, A. F. de Lespinasse, J. A. Tauber, D. K. Gifford, and M. F. Kaashoek:** *Rover: a toolkit for mobile information access*. In Proceedings of the 15th ACM symposium on Operating systems principles, ACM Press, New York, NY, USA, 1995, pp. 156-171

[108] **Marco Mamei, Franco Zambonelli, and Letizia Leonardi:** *Programming Pervasive and Mobile Computing Applications with the TOTA Middleware*. In Proceedings of the 2nd IEEE International Conference on Pervasive Computing and Communications (PerCom'04), IEEE Computer Society, Washington, DC, USA, 2004, pp. 263-275

[109] **A. D. Birrell and B. J. Nelson:** *Implementing remote procedure calls*. In Proceedings of the ACM Symposium on Operating System Principles, October 1983

[110] **Manuel Román, Christopher Hess, Renato Cerqueira, Anand Ranganathan, Roy H. Campbell, and Klara Nahrstedt:** *A Middleware Infrastructure for Active Spaces*. In IEEE Pervasive Computing, October-December 2002, pp. 74-83

[111] **Michael R. Genesereth and Steven P. Ketchpel:** *Software Agents*. In Communications of the ACM, Vol. 37, No. 7, July 1994, pp. 48-59

[112] **Mejdi Kaddour and Laurent Pautet:** *Towards an adaptable message oriented middleware for mobile environments*. In Proceedings of the IEEE 3rd workshop on Applications and Services in Wireless Networks, Bern, Switzerland, July 2003

[113] **Yongqiang Huang and Hector Garcia-Molina:** *Publish/subscribe in a mobile environment*. In Wireless Networks, Vol.10, No.6, 2004, Kluwer Academic Publishers, Hingham, MA, USA, pp. 643-652

[114] **Do-Guen Jung, Kwang-Jin Paek, and Tai-Yun Kim:** *Design of MOBILE MOM: Message Oriented Middleware Service for Mobile Computing*. In Proceedings of the 1999 International Workshops on Parallel Processing, IEEE Computer Society, Washington, DC, USA, 1999

[115] **Pingpeng Yuan and Hai Jin:** *A Composite-Event-Based Message-Oriented Middleware*. In Proceedings of the 2nd International Workshop on Grid and Cooperative Computing, September 2003, pp. 491-498

[116] **Markku Korhonen:** *Message Oriented Middleware (MOM)*. available at: http://www.tml.tkk.fi/Opinnot/Tik-110.551/1997/mqs.htm (July 2005)

[117] **Guruduth Banavar, Tushar Deepak Chandra, Robert E. Strom, and Daniel C. Sturman:** *A Case for Message Oriented Middleware*. In Proceedings of the 13th International Symposium on Distributed Computing, Lecture Notes In Computer Science, Vol. 1693, Springer-Verlag, London, UK, 1999, pp. 1-18

[118] **Brian Oki, Manfred Pfluegl, Alex Siegel, and Dale Skeen:** *The Information Bus®: an architecture for extensible distributed systems*. In Proceedings of the 14th ACM symposium on Operating systems principles, ACM Press, New York, NY, USA, 1994, pp. 58-68

[119] **Christof Bornhoevd, Alejandro P. Buchmann, Mariano Cilia, Ludger Fiege, Felix Freiling, Christoph Liebig, Matthias Meixner and Gero Muehl:** *DREAM: Distributed Reliable Event-BasedApplication Management*. In Web Dynamics: Adapting to Change in Content, Size, Topology and Use by Springer Verlag, 2004, pp. 319-352

[120] **Carles Pairot, Pedro García, and Antonio F. Gómez Skarmeta:** *DERMI: A Decentralized Peer-to-Peer Event-Based Object Middleware*. In Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04), IEEE Computer Society, Washington, DC, USA, 2004, pp. 236-243

[121] **Gianpaolo Cugola, Elisabetta Di Nitto, Alfonso Fuggetta:** *The JEDI event-based infrastructure and its applications to the development of the OPSS WFMS*. In IEEE Transactions on Software Engineering, Vol. 27, No. 9, IEEE Computer Society, Washington, DC, USA, September 2001

[122] **Peter R. Pietzuch and Jean M. Bacon:** *Hermes: A Distributed Event-Based Middleware Architecture*. In Proceedings of the 22nd International Conference on Distributed Computing Systems, IEEE Computer Society, Washington, DC, USA, 2002, pp. 611-618

[123] **Peter R. Pietzuch:** *Event-Based Middleware: A New Paradigm for Wide-Area Distributed Systems?*. In Proceedings of the 6th CaberNet Radicals Workshop, Funchal, Madeira Island, Portugal, February 2002

[124] **René Meier and Vinny Cahill:** *STEAM: Event-Based Middleware for Wireless Ad Hoc Network*. In Proceedings of the 22nd International Conference on Distributed Computing Systems, IEEE Computer Society, Washington, DC, USA, 2002, pp. 639-644

[125] **Andreas Frei, Andrei Popovici, and Gustavo Alonso:** *Event based systems as adaptive middleware platforms*. In Proceedings of the Workshop of the 17th Europeean Conference for Object-Oriented Programming, Darmstadt, Germany, July 2003

[126] **Kerry Raymond:** *Reference Model of Open Distributed Processing: Introduction*. In Proceedings of the 3rd IFIP TC6/WG6.1 International Conference on Open Distributed Processing, Brisbane (Australia), February 1995, pp 3-14

[127] **Christian Becker, Marcus Handte, Gregor Schiele, and Kurt Rothermel:** *PCOM - A Component System for Adaptive Pervasive Computing Applications*. In Proceedings of the 2nd IEEE International Conference on Pervasive Computing and Communications (PerCom 04), Orlando, USA, 2004

[128] **Paolo Costa, Geoff Coulson, Cecilia Mascolo, Gian Pietro Picco, and Stefanos Zachariadis:** *The RUNES Middleware: A Reconfigurable Component-based Approach to Networked Embedded Systems*. In Proceedings of the 16th Annual IEEE International Symposium on Personal Indoor and Mobile Radio Communications (PIMRC), Berlin, Germany, September 2005

[129] Reconfigurable Ubiquitous Networked Embedded Systems, project webpage: http://www.ist-runes.org/ (July 2005)

[130] Open Services Gateway Initiative project web page, available at: http://www.osgi.org/ (July 2005)

[131] **Dave Maples and Peter Kriens:** *The Open Services Gateway initiative: An Introducory Overview*. In IEEE Communications Magazine, December 2001, pp. 2-6

[132] Sun Microsystems: JavaBeans specification, 1997, available at: http://java.sun.com/beans (July 2005)

[134] The Object Management Group, Common Object Request Broker Architecture: Core Specification, Version 3.0.3, March 2004, available at: http://www.omg.org/cgi-bin/apps/doc?formal/04-03-12.pdf (July 2005)

[135] **Mads Haahr, Raymond Cunningham, and Vinny Cahill:** *Supporting CORBA Applications in a Mobile Environment*. In Proceedings of the 5th annual ACM/IEEE International Conference on Mobile Computing and Networking, Seattle, Washington, United States, ACM Press, New York, NY, USA, 1999, ISBN: 1-58113-142-9, pp. 36-47

[136] **Jochen Seitz, Nigel Davies, Michael Ebner, and Adrian Friday:** *A CORBA-based Proxy Architecture for Mobile Multimedia Applications*. In Proceedings of the 2nd IFIP/IEEE International Conference on Management of Multimedia Networks and Services MMNS'98, Versailles, France, 16th-18th November 1998

[137] **Philippe Merle:** *CORBA Component Model Tutorial*. OMG Meeting, Yokohama, Japan, April 24th 2002, available at: http://www.omg.org/cgi-bin/doc?ccm/2002-04-01 (March 2005)

[138] StarCCM project web page, A CCM implementation in C++, http://sourceforge.net/project/showfiles.php?group_id=99026 (July 2005)

[139] **Massimo Ancona and Walter Cazzola:** *Implementing the Essence of Reflection: a Reflective Run-Time Environment*. In Proceedings of ACM Symposium on Applied Computing (SAC), Nicosia, Cyprus, 14th-17th March 2004

[140] **Robert Englander:** *Developing Java Beans*. Java Series, O'Reilly Media, Inc., 1997

[141] **B.C. Smith:** *Reflection and Semantics in a Procedural Programming Language*. PhD thesis, Report MIT-TR-272, Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge, Massachusetts, USA, January 1982

[142] **Werner Vogels:** *Web Services Are Not Distributed Objects*. In IEEE Internet Computing, IEEE Computer Society, Washington, DC, USA, November / December 2003, pp. 59-66

[143] **Francisco Curbera, William A. Nagy, and Sanjiva Weerawarana:** *Web Services: Why and How*. In Proceedings of the Workshop on Object-Oriented Web Services – OOPSLA 2001, Tampa, Florida, USA, October 2001

[144] **Mikio Aoyama, Sanjiva Weerawarana, Hiroshi Maruyama, Clemens Szyperski, Kevin Sullivan, and Doug Lea:** *Web services engineering: promises and challenges*. In Proceedings of the 24th International Conference on Software Engineering, Orlando, Florida, USA, ACM Press, New York, NY, USA, 2002, pp. 647-648

[145] **D. Fensel and C. Bussler:** *The Web Services Modeling Framework WSMF*. Specification, available at: http://www1-c703.uibk.ac.at/users/c70385/wese/wsmf.paper.pdf (July 2005)

[146] **Aphrodite Tsalgatidou and Thomi Pilioura:** *An Overview of Standards and Related Technologies in Web Services*. In Distributed and Parallel Databases, Vol. 12, No. 2-3, Kluwer Academic Publishers, Hingham, MA, USA, September-November 2002, pp. 135-162

[147] **Heather Kreger:** *Web services conceptual architecture WSCA 1.0*. Technical report, IBM, 2001, available at: http://www-4.ibm.com/software/solutions/webservices/ (July 2005)

[148] **Christopher Ferris and Joel Farrell:** *What Are Web Services?*. In Communications of the ACM, Vol. 46, No. 6, June 2003, page 31

[149] **K. Gottschalk, S. Graham, Heather Kreger, and J. Snell:** *Introduction to Web services architecture*. In IBM Systems Journal, Vol. 41, No. 2, 2002, pp. 170-177

[150] **Mike P. Papazogolou and D. Georgakopoulos:** *Service-Oriented Computing*. In Communications of the ACM, Vol. 46, No. 10, October 2003, pp. 25-28

[151] **Francisco Curbera, Matthew Duftler, Rania Khalaf, William Nagy, Nirmal Mukhi, and Sanjiva Weerawarana:** *Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI*. In IEEE Internet Computing, IEEE Computer Society, March / April 2002, pp. 86-93

[152] **Heather Kreger:** *Fulfilling the Web Services Promise*. In Communications of the ACM, Vol. 46, No. 6, June 2003, pp. 29-34

[153] **Michael Stal:** *Web Services: Beyond Component-Based Computing*. In Communications of the ACM, Vol. 45, No. 10, October 2002, pp. 71-76

[154] **Patrick Cauldwell, Rajesh Chawla, Vivek Chopra, Gary Damschen, Chris Dix, Tony Hong, Francis Norton, Uche Ogbuji, Glenn Olander, Mark A. Richman, Kristy Saunders, and Zoran Zaev:** *Professional XML Web Services*. Wrox Press Ltd., Birmingham, UK, 2001, ISBN: 1-861005-09-1

[155] **Steven J. Vaughan-Nichols:** *Web Services: Beyond the Hype*. In IEEE Computer, Vol. 35, No. 2, IEEE Computer Society Press, Los Alamitos, CA, USA, February 2002, pp. 18-21

[156] **Linh Pham and Guide Gehlen:** *Realization and Performance Analysis of a SOAP Server for Mobile Devices*. In Proceedings of the 11th European Wireless Confernce 2005, Vol. 2, Nicosia, Cyprus, April 2005, pp. 791-797

[157] **Michael Ditze, Guido Kämper, Isabell Jahnich, and Reinhard Bernhardi-Grisson:** *Service-based Access to Distributed Embedded Devices through the Open Service Gateway*. In Proceedings of the IEEE Industrial Informatics 2005 conference, August 2005

[158] Open Services Gateway Initiative, OSGi Service Platform, Release 3, IOSPress, Amsterdam, The Netherlands, March 2003

[159] **Choonhwa Lee, David Nordstedt, and Sumi Helal:** *Enabling Smart Spaces with OSGi*. In IEEE Pervasive Computing, July-September 2003, pp. 89-94

[160] **Richard S. Hall and Humberto Cervantes:** *Challenges in Building Service-Oriented Applications for OSGi*. In IEEE Communications Magazine, May 2004, pp. 144-149

[161] **Ryutaro Kawamura and Hiroyuki Maeomichi:** *Standardization Activity of OSGi (Open Services Gateway Initiative)*. In NTT Technical Review, Vol. 2, No. 1, January 2004, pp. 94-97

[162] **Li Gong:** *A Software Architecture for Open Service Gateways*. In IEEE Internet Computing, January / February 2001, pp. 64-70

[163] **Chris Loeser, Wolfgang Müller, Frank Berger, and Heinz-Josef Eikerling:** *Peer to Peer Networks for Virtual Home Environments*. In Proceedings of the 36th Hawaii international Conference on System Sciences (HICSS-36), Big Island, Hawaii, January 2003

[164] **J. Dunlop, R. C. Atkinson, J. Irvine, and D. Pearce:** *A Personal Distributed Environment for Future Mobile Systems*. In Proceedings of the IST Mobile and Wireless Summit 2003, Aveiro, Portugal, July 2003

[165] **Sumi Helal, Bryon Winkler, Choonhwa Lee, Youssef Kaddourah, Lisa Ran, Carlos Giraldo, and William Mann:** *Enabling Location-Aware Pervasive Computing Applications for the Elderly*. In Proceedings of the 1st IEEE Pervasive Computing Conference, Fort Worth, Texas, USA, June 2003

[166] **Sumi Helal, Carlos Giraldo, Youssef Kaddourah, Choonhwa Lee, Hicham Zabadani, and William Mann:** *Smart Phone Based Cognitive Assistant*. In Proceedings of the 2nd International Workshop on Ubiquitous Computing for Pervasive Healthcare Applications (UbiHealth), Seattle, Washington, USA, Springer Verlag. November 2003

[167] **Sumi Helal, William Mann, Hicham El-Zabadani, Jeffrey King, Youssef Kaddourah, and Erwin Jansen:** *The Gator Tech Smart House: A Programmable Pervasive Space*. In IEEE Computer magazine, March 2005, pp. 64-74

[168] **Alois Ferscha, Manfred Hechinger, Rene Mayrhofer, and Roy Oberhauser:** *A Light-Weight Component Model for Peer-to-Peer Applications*. In Proceedings of the 24th International Conference on Distributed Computing Systems Workshops - Workshop 4: MDC, IEEE Computer Society Press, March 2004, pp. 520-527

[169] **Richard S. Hall and Humberto Cervantes:** *An OSGi Implementation and Experience Report*. In Proceedings of the IEEE Consumer Communications and Networking Conference, 2004

[170] **Tao Gu, Hung Keng Pung, and Da Qing Zhang:** *Toward an OSGi-Based Infrastructure for Context-Aware Applications*. In IEEE Pervasive Computing, October-December 2004, pp. 66-74

[171] **Hiroo Ishikawa, Yuuki Ogata, Kazuto Adachi, and Tatsuo Nakajima:** *Building Smart Appliance Integration Middleware on the OSGi Framework*. In Proceedings of the 7th IEEE International Symposium on Object-oriented Realtime Distributed Computing, Vienna, Austria, May 2004

[172] **Andreas Frei and Gustavo Alonso:** *A dynamic lightweight Architecture*. In Proceedings of the 3rd International Conference on Pervasive Computing and Communications (PerCom 2005), March 2005

[173] **Humberto Cervantes and Richard S. Hall:** *Beanome: A Component Model for the OSGi Framework*. In Proceedings of the Workshop on Software Infrastructures for Component-Based Applications on Consumer Devices, Lausanne, Switzerland, September 2002

[174] **Paul Grace, Gordon S. Blair, and Sam Samuel:** *Middleware Awareness in Mobile Computing*. In Proceedings of the 23rd International Conference on Distributed Computing Systems, IEEE Computer Society, Washington, DC, USA, 2003, pp. 382-387

[175] **Paul Grace, Gordon S. Blair, and Sam Samuel:** *A Marriage of Web Services and Reflective Middleware to Solve the Problem of Mobile Client Interoperability*. In Proceedings of the 1st international symposium on Information and communication technologies, Trinity College, Dublin, Ireland, 2003, pp. 506-511

[176] **Paul Grace, Gordon S. Blair, and Sam Samuel:** *A Higher Level Abstraction for Mobile Computing Middleware*. In Proceedings of Workshop on Communication Abstractions for Distributed Systems, Paris, France, November 2003

[177] **Cecilia Mascolo, Stefanos Zachariadis, Gian Pietro Picco, Paolo Costa, Gordon Blair, Nelly Bencomo, Geoff Coulson, Paul Okanda, and Thirunavukkarasu Sivaharan:** *Runes Middleware Architecture*. Deliverable of the IST Project number IST-004536-RUNES, available at: http://www.ist-runes.org/docs/deliverables/D5_02.pdf (July 2005)

[178] **Christian Becker and Gregor Schiele:** *Middleware and Application Adaptation Requirements and their Support in Pervasive Computing*. In Proceedings of the 3rd International Workshop on Distributed Auto-adaptive and Reconfigurable Systems (DARES) at ICDCS, Providence, USA, 19th-22nd May 2003, pp. 98-103

[179] **Marcus Handte, Christian Becker, and Gregor Schiele:** *Experiences - Extensibility and Minimalism in BASE*. In Proceedings of the Workshop on System Support for Ubiquitous Computing (UbiSys) at Ubicomp, Seattle, USA, 2003

[180] **Humberto Cervantes and Richard S. Hall:** *Automating Service Dependency Management in a Service-Oriented Component Model*. In Proceedings of the 6th Workshop on Component Based Software Engineering (CBSE), 2003

[181] **Olaf Drögehorn and Klaus David:** *Design Principles for future Service Platforms*. In Proceedings of the 2005 IEEE/IPSJ International Symposium on Applications and the Internet Workshops (SAINT 2005 Workshops), IEEE Computer Society, Trento, Italy, February 2005, pp. 132-135

[182] **Bjoern Wuest, Olaf Drögehorn, Hilko Hofmann, Renata Guarneri, and Klaus David:** *Personalisation Proxy in Content Delivery Networks*. In Proceedings of the IST Mobile & Wireless Summit 2004, Lyons, France, June 2004

[183] **Walter L. Hürsch and Cristina Videira Lopes:** *Separation of Concerns*. Technical Report, Northeatern University, Boston, USA, February 1995

[184] **Carlo Ghezzi, Mehdi Jazaveri, and Dino Mandrioli:** *Fundamentals of Software Engineering*. Prentice-Hall Inc., Upper Saddle River, NJ, 1991, ISBN: 0133056996

[185] **Marie-Luise Moschgath:** *Kontextabhängige Zugriffskontrolle für Anwendungen im Ubiquitous Computing*. Dissertation, Universität Darmstadt, Fachbereich Informatik, 01st July 2002

[186] **Geetanjali Sampemane, Prasad Naldurg, and Roy H. Campbell:** *Access Control for Active Spaces*. In Proceedings of Annual Computer Security Applications Conference (ACSAC2002), Las Vegas, Nevada, Dec 9-13 2002

[187] **B. W. Lampson:** *Protection*. In Proceedings of 5th Princeton Symposium on Information Science and Systems, 1971, pp.437-443

[188] **David Ferraiolo, D.M. Gilbert, and N. Lynch:** *An Examination of Federal and Commercial Access Control Policy Needs*. In Proceedings of the NIST-NCSC National Computer Security Conference, 1993

[189] **D.E. Denning:** *A lattice model of secure information flow*. In Communications of ACM, Vol.19, No.5, 1976, pp.236-243

[190] **D. Elliot Bell and Leonard J. LaPadula:** *Secure Computer Systems: Mathematical Foundations and Model*. Technical Report M74-244, MITRE Corp., Bedford MA, 1973

[191] **D. Elliot Bell and Leonard J. LaPadula:** *Unified exposition and Multics interpretation*. Technical Report ESD-TR-75-306, MITRE Corp. Bedford MA, March 1975

[192] **David Ferraiolo and Richard Kuhn:** *Role-Based Access Control*. In Proceedings of 15th National Computer Security Conference, NIST/NSA, 1992, pp. 554-563, available at: http://xsun.sdct.itl.nist.gov/rbac/paper/rbac1.html

[193] **David Ferraiolo, J. Cugini, and Richard Kuhn:** *Role Based Access Control: Features and Motivations*. In Proceedings of Annual Computer Security Conference, IEEE Computer Society Press, Baltimore, 1995

[194] **R. Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman:** *Role Based Access Control Models*. In IEEE Computer, Vol.29, No.2, 1996, pp.38-47

[195] **Florian Almenárez, Andrés Marín, Celeste Campo, and Carlos Gracía Rubio:** *TrustAC: Trust-Based Access Control for Pervasive Devices*. In Proceedings of the 2nd International Conference on Security in Pervasive Computing, Boppard, Germany, April 2005

[196] **Florian Almenárez, Andrés Marín, Celeste Campo, and Carlos Gracía Rubio:** *Mobile Devices: Secure Clients or Secure Peers?*. In Proceedings of the IST Mobile & Wireless Summit 2005, Dresden, Germany, June 2005

[197] **D. L. Parnas:** *A technique for the Specification of Software Modules with Examples*. In Communications of the ACM, Vol. 15, No. 5, May 1972, pp. 330-336

[198] **E. W. Dijkstra:** *Notes on Structured Programming*. In APIC Studies in Data Processing, No. 8, Academic Press, New York, NY, USA, 1972

[199] **Friedrich Steimann and Philip Mayer:** *Patterns of Interface-Based Programming*. In the Journal of Object Technology, Vol. 4, No. 5, Bertrand Meyer (Ed.), July/August 2005

[200] **Jens Gößner, Philip Mayer, and Friedrich Steimann:** *Interface Utilization in the Java Development Kit*. In Proceedings of the 2004 ACM symposium on Applied computing, ACM Press, New York, NY, USA, 2004, pp. 1310-1315

[201] **Mariano Ceccato and Paolo Tonella:** *Adding Distribution to Existing Applications by means of Aspect Oriented Programming*. In Proceedings of the Source Code Analysis and Manipulation, 4th IEEE International Workshop on (SCAM'04), IEEE Computer Society, Washington, DC, USA, 2004, pp. 107-116

[202] **Andrew Borg:** *A Real-Time RMI Framework for the RTSJ*. In Proceedings of the 15th Euromicro Conference on Real-Time Systems, July 2003, pp. 238-246

[203] **Yaron Weinsberg and Israel Ben-Shaul:** *A Programming Model and System Support for Disconnected-Aware Applications on Resource-Constrained Devices*. In Proceedings of the 24th International Conference on Software Engineering, ACM Press, New York, NY, USA, 2002, pp. 374-384

[204] **Savitha Srinivasan:** *Design Patterns in Object-Oriented Frameworks*. In IEEE Computer, February 1999, pp.24-32

[205] **Tim Kindberg and Armando Fox:** *System Software for Ubiquitous Computing*. In IEEE Pervasive Computing, Vol. 1, No. 1, January/February 2002, pp.70-81

[206] **Ioannis Fikouras and Friedhelm Ramme:** *Service Orchestration with Generic Service Elements*. In Proceedings of the 6th Internation Symposium on Wireless Personal Multimedia Communication, October 2003, Yokosuka, Japan

[207] **Steve Vinoski:** *Service Discovery 101*. In IEEE Internet Computing, Vol. 7, No. 1, January/February 2003, pp. 69-71

[208] **Kimmo Raatikainen:** *Wireless Internet: Challenges and Solutions*. Helsinki University Printing House, Helsinki, Finland 2004, ISBN 952-10-2073-3, available at http://www.cs.helsinki.fi/u/kraatika/Papers/TenYearsEversion.pdf (July 2005)

[209] **Yérom-David Bromberg and Yalérie Issarny:** *Service Discovery Protocol Interoperability in the Mobile Environment*. In Proceedings of the International Workshop Software Engineering and Middleware, September 2004

[210] **Choonhwa Lee and Sumi Sumi Helal:** *Protocols for Service Discovery in Dynamic and Mobile Networks*. International Journal of Computer Research, vol. 11, no. 1, Nova Science Publishers, 2002, pp. 1-12

[211] **Feng Zhu, Matt W. Murka, and Lionel M. Ni:** *Service Discovery in Pervasive Computing Environments*. In IEEE Pervasive Computing, vol. 3, no. 4, October-December 2005, pp. 81-90

[212] **Rekesh John:** *UPnP, Jini and Salutation – A look at some popular coordination frameworks for future networked devices*. California Software Labs, June 1999, available at: http://www.calsoft.co.in/whitepapers/jini_upnp.html (July 2005)

[213] **Paul Mockapetris:** *Domain Names – Concepts and Facilities*. Request for Comments 822, Network Working Group, Information Sciences Institute, November 1983, available at: http://www.ietf.org/rfc/rfc882.txt (July 2005)

[214] **Paul Mockapetris:** *Domain Names – Concepts and Facilities*. Request for Comments 1034, Network Working Group, Information Sciences Institute, November 1987, available at: http://www.ietf.org/rfc/rfc1034.txt (July 2005)

[215] **Paul Mockapetris:** *Domain Names – Implementation and Specification*. Request for Comments 1035, Network Working Group, Information Sciences Institute, November 1987, available at: http://www.ietf.org/rfc/rfc1035.txt (July 2005)

[216] SUN Microsystems, Java™ Remote Method Invocation Specification, Revision 1.10, 2004, available at: http://java.sun.com/j2se/1.5.0/docs/guide/rmi/spec/rmi-title.html (July 2005)

[217] **Jan Newmarch:** *Jan Newmarch's Guide to JINI Technologies*. Version 3.06, October 2004, available at: http://jan.netcomp.monash.edu.au/java/jini/tutorial/Jini.xml (July 2005)

[218] **Jim Waldo and Ken Arnold:** *The JINI Specifications*. Second Edition, Pearson Education, Upper Saddle River, New York, USA, December 2000

[219] Microsoft Corporation, Understanding Universal Plug and Player, White Paper, available at: http://www.upnp.org/download/UPNP_UnderstandingUPNP.doc (July 2005)

[220] **Yaron Y. Goland, Ting Cai, Paul Leach, Ye Gu, and Shivan Albright:** *Simple Service Discovery Protocol*. Internet Draft, October 1999, available at: http://www.upnp.org/download/draft_cai_ssdp_v1_03.txt (July 2005)

[221] **J. Veizades, E. Guttman, C. Perkins, and S. Kaplan:** *Service Location Protocol*. Request for Comments 2165, Network Working Group, June 1997, available at: http://www.ietf.org/rfc/rfc2165.txt?number=2165 (July 2005)

[222] **E. Guttman, C. Perkins, J. Veizades, and M. Day:** *Service Location Protocol, Version 2*. Request for Comments 2608, Network Working Group, June 1999, available at: http://www.ietf.org/rfc/rfc2608.txt (July 2005)

[223] **Christian Bettstetter and Christoph Renner:** *A Comparison Of Service Discovery Protocols And Impelementation Of The Service Location Protocol*. In Proceedings of EUNICE, September 2000, available at: http://wwwtgs.cs.utwente.nl/eunice/summerschool/papers/paper5-1.pdf (July 2005)

[224] **Rasmus L. Olsen, Homare Murakami, Hans-Peter Schwefel, and Ramjee Prasad:** *User centric Service Discovery in Personal Networks*. In Proceedings of the International Symposium on Wireless Personal Multimedia Communication, Albano Terme, Italy, September 2004

[225] **John Garofalakis, Yannis Panagis, Evangelos Sakkopoulos, and Athanasios Tsakalidis:** *Web Service Discovery Mechanisms: Looking for a Needle in a Haystack?*. In Proceedings of the International Workshop on Web Engineering, in conjunction with ACM Hypertext 2004, August 2004, Santa Cruz, USA

[226] **Luc Clement, Andrew Hately, Claus von Riegen, and Tony Rogers**, *UDDI Version 3.02 – UDDI Spec Technical Committee Draft*, OASIS Open, October 2004, available at: http://uddi.org/pubs/uddi-v3.0.2-20041019.pdf (July 2005)

[227] **Venu Vasudevan:** *A Web Services Primer*. webservices.xml.com, O'Reilly, April 2001, available at: http://webservices.xml.com/pub/a/ws/2001/04/04/webservices/ (July 2005)

[228] **Steve Graham, and Peter Niblett:** *Web Services Base Notification 1.0*. OASIS, May 2004, available at: http://www.oasis-open.org/committees/download.php/6599/WS-BaseNotification-1-0.pdf (July 2005)

[229] **Shuping Ran:** *A model for web services discovery with QoS*. In ACM SIGecom Exchanges, Vol. 4, No. 1, ACM Press, New York, New York, USA, 2003, pp. 1-10

[230] **David Mertz:** *Understanding ebXML – Untangling the business Web of the future*. IBM developerWorks, June 2001, available at: http://www-128.ibm.com/developerworks/xml/library/x-ebxml/ (July 2005)

[231] **Robin Cover**: *IBM and Microsoft Issue Specification and Software for Web Services Inspection Language*. OASIS Cover Pages, November 2001, available at: http://xml.coverpages.org/ni2001-11-02-a.html (July 2005)

[232] **Keith Ballinger, Peter Brittenham, Ashok Malhotra, William A. Nagy, and Stefan Pharies:** *Web Services Inspection Language (WS-Inspection) 1.0*. In IBM developerWorks, November 2001, available at: ftp://www6.software.ibm.com/software/developer/library/ws-wsilspec.pdf (July 2005)

[233] **Robin Cover:** *Microsoft Releases Web Services Dynamic Discovery Specification (WS-Discovery)*. OASIS Cover Pages, February 2004, available at: http://xml.coverpages.org/ni2004-02-17-b.html (July 2005)

[234] **John Beatty, Gopal Kakivaya, Devon Kemp, Thomas Kuehnel, Brad Lovering, Bryan Roe, Christopher St. John, Jeffrey Schlimmer, Guillaume Simonnet, Doug Walter, Jack Weast, Yevgeniy Yarmosh, and Prasad Yendluri:** *Web Services Dynamic Discovery (WS-Discovery)*. Specification, Microsoft Corporation, Jeffrey Schlimmer (Ed.), April 2005, available at: http://xml.coverpages.org/WS-Discovery20050422.pdf (July 2005)

[235] **William A. Nagy, Francisco Curbera, and Sanjiva Weerawarana:** *The Advertisement and Discovery of Services (ADS) protocol for Web services*. In IBM developerWorks, October 2000, available at: http://www-128.ibm.com/developerworks/library/ws-ads.html?dwzone=ws (July 2005)

[236] **Kunal Verma, Kaarthik Sivashanmugam, Amit Sheth, Abhijit Patil, Swapna Oundhakar, and John Miller:** *METEOR-S WSDI: A Scalable P2P Infrastructure of Registries for Semantic Publication and Discovery of Web Services*. 2003, available at: http://lsdis.cs.uga.edu/lib/download/VSS+03-TM06-003-METEOR-S-WSDI.pdf (July 2005)

[238] **Daniel J. Mandell and Sheila A. McIlraith:** *Automating Web Service Discovery, Customization, and Semantic Translation with a Semantic Discovery Service*. In Proceedings of the 12th International World Wide Web Conference, Budapest, Hungary, May 2003

[239] **Matthew Moran and Adrian Mocan:** *WSMX – An Architecture for Semantic Web Service Discovery, Mediation and Invocation*. In Proceedings of the 3rd International Semantic Web Conference, November 2004, Hiroshima, Japan

[240] **M. Zaremba and Matthew Moran:** *Enabling Execution of Semantic Web Services: WSMX Core Platform*. In Proceedings of the 1st WSMO Implementation Workshop (WIW2004), Frankfurt, Germany, 2004

[241] **A. Haller, E. Cimpian, Adrian Mocan, E. Oren, and C. Bussler:** *WSMX - A Semantic Service-Oriented Architecture*. In Proceedings of the International Conference on Web Service (ICWS 2005), Orlando, Florida, 2005

[242] **Emanuele Della Valle, Irene Celino, and Dario Cerizza:** *Semantic Web Activities Homepage*, available at: http://swa.cefriel.it/Glue (July 2005)

[243] The Athena Project Homepage, available at: http://disl.cc.gatech.edu/Athena/ (July 2005)

[244] **James Caverlee, Ling Liu, and Daniel Rocco:** *Discovering and Ranking Web Services with BASIL: A Personalized Approach with Biased Focus*. In Proceedings of the 2nd International Conference on Service Oriented Computing (ICSOC '04). Co-sponsored by ACM SIGWEB and ACM SIGSOFT (ACM Press), November 2004, New York, New York, USA, available at: http://www.cc.gatech.edu/~caverlee/pubs/p89-caverlee.pdf (July 2005)

[245] **Paul Grace, Gordon S. Blair, and Sam Samuel:** *Interoperating with Services in a Mobile Environment*, Technical Report (MPG-03-01), Lancaster University, 2003, available at: http://www.lancs.ac.uk/postgrad/gracep/grace03.pdf (July 2005)

[246] **Paul Grace, Gordon S. Blair, and Sam Samuel:** *ReMMoC: A Reflective Middleware to Support Mobile Client Interoperability*. In Proceedings of International Symposium on Distributed Objects and Applications(DOA), Catania, Sicily, Italy, November 2003

[247] **S. R. Schach:** *Classical and Object-oriented Software Engineering with UML and C++*. McGraw-Hill, 1998

[248] **Yves Vandewoude and Yolande Berbers:** *An Overview and Assessment of Dynamic Update Methods for Component-oriented Embedded Systems*. In Proceedings of The International Conference on Software Engineering Research and Practice, CSREA Press, Las Vegas, Nevada, USA, June 2002, pp. 521-527

[249] **Michael Hicks, Jonathan T. Moore, and Scott Nettles:** *Dynamic Software Updating*. In Proceedings of the ACM SIGPLAN'01 Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah; United States, June 2001, pp. 13-23

[250] **Andrew Baumann and Gernot Heiser:** *Providing Dynamic Update in an Operating System*. In Proceedings of USENIX '05 Annual Technical Confernece, General Track, Anaheim, California, USA, April 2005, pp. 279-291

[251] **Jesper Andersson and Tobias Ritzau:** *Dynamic Code Update in JDrums*. In Procedings of the 1st Workshop on Software Engineering for Wearable and Pervasive Computing (SEWPC) in Conjunction with ICSE'2000, Limerick, June 2000

[252] **Scott Malabarba, Raju Pandey, Jeff Gragg, Earl Barr, and J. Fritz Barnes:** *Runtime Support for Type-Safe Dynamic Java Classes*. In Proceedings of the 14th European Conference on Object-Oriented Programming, Springer-Verlag, London, United Kingdom, 2000, pp. 337-361

[253] **Kuo-Feng Ssu and Hewijin Christine Jiau:** *Online Non-stop Software Updating Using Replicated Execution Blocks*. In Proceedings of the 24th International Computer Software and Applications Conference, IEEE Computer Society, Washington, DC, USA, 2000, pp. 319-324

[254] **F. J. Corbato and V. A. Vyssotsky:** *Introduction and Overview of the Multics System*. In Proceedings of the AFIPS Fall Join Computer Conference, 1965, pp. 185-196

[255] **Huw Evans:** *Dynamic On-line Object Update in the Grumps System*. In Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment, IEEE Computer Society, Washington, DC, USA, 2002, pp. 994-999

[256] **M. M. Gorlick and R. R. Razouk:** *Using weaves for software construction and analysis*. In Proceedings of the 13th International Conference on Software Engineering, IEEE Computer Society Press, May 1991

[257] **M. M. Gorlick and A. Quilici:** *Visual programming-in-the large versus programming-in-the-small*. In Proceedings of the IEEE Symposium on Visual Languages, IEEE Computer Society Press, October 1994

[258] **J. Kramer, and J. Magee:** *The evolving philosophers problem: Dynamic change management*. In IEEE Transactions on Software Engineering, Vol. 16, No. 11, November 1990

[259] **Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor:** *Architecture-Based Runtime Software Evolution*. In Proceedings of the International Conference on Software Engineering 1998 (ICSE'98), Kyoto, Japan, April 1998

[260] **Mitsuhisa Sato, Hidemoto Nakada, Satoshi Sekiguchi, Satoshi Matsuoka, Umpei Nagashima, and Hiromitsu Takagi:** *Ninf: A Network based Information Library for Global World-Wide Computing Infrastructure*. In Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking, Springer-Verlag, London, United Kingdom, 1997, pp. 491-502

[261] **Abdelmadjid Ketfi and Noureddine Belkhatir:** *Dynamic Interface Adaptability in Service Oriented Software*. In Proceedings of the 8th International Workshop on Component-Oriented Programming (WCOP'03), Darmstadt, Germany, July 2003

[262] **František Plášil, Dušan Bálek, and Radovan Janeček:** *SOFA/DCUP: Architecture for Component Trading and Updating*. In Proceedings of the International Conference on Configurable Distributed Systems, IEEE Computer Society, Washington, DC, USA, 1998, pp. 43-51

[263] **Ralph E. Johnson:** *Frameworks = (Components + Patterns)*. In Communications of the ACM, Vol. 40, No. 10, October 1997, pp. 39-42

[264] **Karen Renaud and Huw Evans:** *Engineering Java™ Proxy Objects using Reflection*. In Proceedings of the NET.OBJECT-DAYS 2000, Messekongresszentrum Erfurt, Germany, October 2000

[265] **H. Lieberman:** *Using prototypical objects to implement shared behavior in object-oriented systems*. In Proceedings of OOPSLA ´86, Object-Oriented Programming Systems, Languages and Applications, November 1986, pp. 214-223

[266] **Ian Welch and R. J. Stroud:** *Kava – Using Byte code Rewriting to add Behavioural Reflection to Java*. In Proceedings of the 1st OOPSLA Workshop on Reflection and Software Engineering, 2000, pp. 155—167

[267] **S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S. Kim:** *Composite Events for Active Databases: Semantics*. Contexts and Detection, In the Proceeding of the 20th VLDB Conference, Santiago, 1994

[268] **Richard Lajoie and Rudolf K. Keller:** *Design and Reuse in Object-Oriented Frameworks: Patterns, Contracts, and Motifs in Concert*. In Proceedings of the 62nd Congress of the Association Canadienne Française pour l'Avancement des Sciences (ACFAS), Montreal, Canada, May 1994

[269] **David Crocker:** *Safe Object-Oriented Software: The Verified Design-By-Contract Paradigm*. In Proceedings of the 12th Safety-Critical Systems Symposium, F. Redmill and T. Anderson (Eds.), Springer-Verlag, London, 2004, 19-41

[270] **Andreas Rausch:** *"Design by Contract" + "Componentware" = "Design by Signed Contract"*. In Journal of Object Technology, Vol. 1, No. 3, Special Issue for TOOLS USA 2002, pp. 19-36

[271] **Miguel-Ángel Sicilia and Salvador Sánchez-Alonso:** *On the Concept of Learning Object Design by Contract*. In WSEAS Transactions on Computers, Vol. 2, No. 3, July 2003, pp. 612-617

[272] **Bertram Meyer:** *On To Components*. In IEEE Computer, Vol. 32, No. 1, January 1999, pp. 139-143

[273] **Knut Wicksell:** *Finanztheoretische Untersuchungen: nebst Darstellungen und Kritik des Steuerwesens Schwedens*. Gotlieb Fischer Verlag, Jena, Germany, 1896

[274] **Charles W. Cobb and Paul Howard Douglas:** *A Theory of Production*. In American Economic Review, Vol. 18, No. 1, March 1928, pp. 139-165

[275] **Jean-Marc Jézéquel and Bertrand Meyer:** *Design by Contract: The Lessons of Ariane*. In IEEE Computer, Vol. 30, No. 1, January 1997, pp. 129-130

[276] **Yoonsik Cheon, Gary T. Leavens, Murali Sitaraman, and Stephen Edwards:** *Model Variables: Cleanly Supporting Abstraction in Design By Contract*. In Software - Practice & Experience, Vol. 35, No. 6, May 2005, pp. 583-599

[277] **Torben Weis, Christian Becker, Kurt Geihs, and Noël Plouzeau:** *A UML Meta-model for Contract Aware Components*. In Proceedings of the 4th International Conference on the Unified Modeling Language, Modeling Languages, Concepts, and Tools, Lecture Notes in Computer Science, Springer-Verlag, London, United Kingdom, 2001, pp. 442-456

[278] **James L. Elshoff:** *Defensive Programming*. GMR-1799, Computer Science Department, General Motors Research Labs, Warren, Michigan, Frebruary 1975

[279] **Donald G. Firesmith:** *A Comparison of Defensive Development and Design by Contract*. In Proceedings of the Technology of Object-Oriented Languages and Sytems, IEEE Computer Society, Washington, Dacota, USA, 1999

[280] **C. A. R. Hoare:** *An axiomatic basis for computer programming*. In Communication of the ACM, Vol. 12, No. 10, October 1969, pp. 576-580

[281] **Z. Milosevic, S. Gibson, P. F. Linington, J. Cole, and S. Kulkarni:** *On design and implementation of a contract monitoring facility*. In Proceedings of the 1st IEEE Workshop on Electronic Contracting, July 2004, pp. 62-70

[282] **Mark Grinblatt, and Sheridan Titman:** *Adverse risk incentives and the design of performance-based contracts*. In Management Services, Institute for Operations Research and the Management Sciences, Linthicum, Maryland, USA, 1989, pp. 807-822

[283] **P. R. Polsani:** *Use and Abuse of Reusable Learning Objects*. In Journal of Digital Information, Vol. 3, No. 4, 2003

[284] **Yishai A. Feldman:** *Extrem Design by Contract*. In Proceedings of the 4th International Conference on Extreme Programming and Agile Processes in Software Engineering (XP2003), Springer, Lecture Notes in Computer Science, Genova, Italy, May 2003, pp. 261-270

[285] **Herbert Klaeren, Elke Pulvermüller, Awais Rashid, and Andreas Speck:** *Aspect Composition applying the Design by Contract Principle*. In Proceedings of Generative and Component-based Software-Engineering Second International Symposium GCSE2000, Springer, Lecture Notes in Computer Science, Erfurt, Germany, 2000, pp. 57-69

[286] **Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin:** *Aspect-Oriented Programming*. In Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland, Springer-Verlag LNCS 1241, June 1997

[287] **Eric Wohlstadter, Stoney Jackson, and Premkumar Devanbu:** *DADO: enhancing middleware to support crosscutting features in distributed, heterogeneous systems*. In Proceedings of the 25th International Conference on Software Engineering, Portland, Oregon, IEEE Computer Society, Washington, DC, USA, 2003, ISBN: 0-7695-1877-X, pp. 174-186

[288] **Robert Bruce Findler and Matthias Felleisen:** *Contracts for Higher-Order Functions*. In Proceedings of the 7th ACM SIGPLAN international conference on Functional programming, Pittsburgh, PA, USA, 2002, pp.48-59

[289] **Gary T. Leavens, Albert L. Baker, and Clyde Ruby:** *JML: A Notation for Detailed Design*. In Behavioural Specifications for Businesses and Systems, Haim Kilov, Bernhard Rumpe, and William Harvey (Eds.), Kluwer Academic Publishers, 1999, pp. 175-188

[290] **Reto Kramer:** *iContract – The Java™ Design by Contract™ Tool*. In Proceedings of the Technology of Object-Oriented Languages, Santa Barbara, California, USA, August 1998, pp. 295-307

[291] **Detlef Bartetzko, Clemens Fischer, Michael Möller, and Heike Wehrheim:** *Jass – Java with Assertions*. In Electronic Notes in Theoretical Computer Science, Vol. 55, No. 2, Elsevier Science, 2001

[292] **Martin Lackner, Andreas Krall, and Franz Puntigam:** *Support Design by Contract in Java*. In Journal of Object Technology, Vol. 1, No. 3, Special Issue for TOOLS USA 2002, pp. 57-76

[293] **Isabel Nunes:** *Design by Contract Using Meta-Assertions*. In Journal of Object Technology, Vol. 1, No. 3, Special Issue for TOOLS USA 2002, pp. 37-56

[294] **Eric Allen and Robert Cartwright:** *The case for run-time types in generic Java*. In Proceedings of the Inaugural Conference on the Principles and Practice of Programming, Dublin, Ireland, 2002, pp.19-24

[295] **Mark Day, Robert Gruber, Barbara Liskov, and Andrew C. Meyers:** *Subtypes vs. where clauses: constraining parametric polymorphism*. In Proceedings of the 10th annual conference on Object-oriented programming systems, languages, and applications, Austin, Texas, USA, 1995, pp.156-168

[296] **Boris Bokowski:** *CoffeeStrainer: statically-checked constraints on the definition and use of types in Java*. In Proceedings of the 7th European software engineering conference, Toulouse, France, 1998, pp.355-374

[297] **Jonathan Bachrach and Keith Playford:** *The Java syntactic extender (JSE)*. In Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications, Tampa Bay, Florida, USA, 2001, pp.31-42

[298] **Cormac Flanagan, K. Rustan, M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata:** *Extended static checking for Java*. In Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, Berlin, Germany, 2002, pp.234-245

[299] **Murat Karaorman, Urs Hölzle, and John Bruno:** *jContractor: A Reflective Java Library to Support Design By Contract*. Technical Report 1998-31, Department of Computer Science, University of California Santa Barbara, December 1998, available at: http://www.cs.ucsb.edu/research/tech_reports/reports/1998-31.ps (July 2005)

[300] **Parker Abercrombie and Murat Karaorman:** *jContractor: Bytecode Instrumentation Techniques for Implementing Design by Contract in Java*. In Proceedings of the 2nd Workshop on Runtime Verification (RV'02), Klaus Havelund and Grigore Rosu (Eds.), Elsevier Science, 2002

[301] **Murat Karaorman and Parker Abercrombie:** *jContractor: Introducing Design-by-Contract to Java Using Reflective Bytecode Instrumentation*. In Formal Methods in System Design, Vol. 27, No. 3, Kluwer Academic Publishers, Hingham, Massachusets, USA, November 2005, pp. 275-312

[302] **Siddharth Bajaj, Don Box, Dave Chappell, Francisco Curbera, Glen Daniels, Phillip Hallam-Baker, Maryann Hondo, Chris Kaler, Dave Langworthy, Ashok Malhotra, Anthony Nadalin, Nataraj Nagaratnam, Mark Notthingham, Hemma Prafullchandra, Claus von Riegen, Jeffrey Schlimmer, Chris Sharp, and John Shewchuk:** *Web Services Policy Framework (WS-Policy)*. Specification, September 2004, available at: http://www-128.ibm.com/developerworks/library/specification/ws-polfram/ (July 2005)

[303] **Vladimir Tosic and Bernard Pagurek:** *On Comprehensive Contractual Descriptions of Web Services*. In Proceedings of the 2005 IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE'05) on e-Technology, e-Commerce and e-Service, IEEE Computer Society, Washington, DC, USA, April 2005, pp. 444-449

[304] **Kunal Verma, Rama Akkiraju, and Richard Goodwin:** *Semantic Matching of Web Service Policies*. In Proceedings of the 2nd International Workshop on Semantic and Dynamic Web Processes (SDWP 2005), In Conjunction with the 3rd International Conference on Web Services (ICWS'05), July, 2005, Orlando, Florida, pp. 79-90

[305] **Alain Anrieux, Asit Dan, Kate Keahy, Heiko Ludwig, and John Rofrano:** *Negotiability Constraints in WS-Agreement*. Version 0.1, January 2004, available at: http://www-unix.mcs.anl.gov/~keahey/Meetings/GRAAP/WS-Agreement%20Negotiability%20Constraints.pdf (July 2005)

[306] **Karine Arnout and Raphael Simon:** *The .NET Contract Wizard: Adding Design by Contract to Languages Other than Eiffel*. In Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems, 2001, pp. 14-23

[307] **Reinhold Plösch:** *Tool Support for Design by Contract*. In Proceedings of Technology of Object-Oriented Languages and Systems, 1998, pp. 282-295

[308] **Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold:** *Getting Started with AspectJ*. In Communications of the ACM, Vol. 41, No. 10, October 2001, pp. 59-65

[309] **Therapon Skotiniotis and David H. Lorenz:** *Cona: Aspects for Contracts and Contracts for Aspects*. In Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, Vancouver, British Columbia, Canada, 2004, pp.196-197

[310] **Martin Lippert and Cristina Videira Lopes:** *A Study on Exception Detection and Handling Using Aspect-Oriented Programming*. In Proceedings of the 22nd International Conference on Software Engineering (ICSE'00), 2000, pp. 418-428

[311] **Dirk Bäumer, Guido Gryczan, Rolf Knoll, Carola Lilienthal, Dirk Riehle, and Heinz Züllighoven:** *Framework Development for Large Systems*. In Communications of the ACM, Vol. 40, No. 10, October 1997, pp. 52-59

[312] **J. Goguen:** *Parametrized Programming*. In IEEE Transactions on Software Engineering, Vol. 10, No. 5, September 1984, pp. 528-543

[313] CASCADAS Project Web page, available at: http://www.cascadas-project.org/index.html (July 2005)

[314] **Birgitta König-Ries and Michael Klein:** *Tutorial "Semantic Service Descriptions - Relevance for Mobile Applications, Requirements and State of the Art"*. Tutorial at the 6th International Conference on Mobile Data Management (MDM 2005), May 2005

[315] **Michael Klein, Birgitta König-Ries, and Michael Müssig:** *What is needed for Semantic Service Descriptions - A Proposal for Suitable Language Constructs*. In International Journal on Web and Grid Services 2005 - Vol. 1, No. 3/4, October 2005, pp. 328-364

[316] **Ulrich Küster, Michael Klein, and Birgitta König-Ries:** *Discovery and Mediation using the DIANE Service Description*. In Proceedings of the Semantic Web Services Challenge 2006, Challenge on Automating Web Services Mediation Choreography and Discovery, Palo Alto, CA, USA, March 2006

[317] **L. Chen, N. R. Shadbolt, C. Goble, F. Tao, S. J. Cox, C, Puleston, and P. Smart:** *Towards a Knowledge-based Approach to Semantic Service Composition*. In Lecture Notes in Computer Science LNCS 2870, 2003, pp 319-334

[318] **Simone A. Ludwig and S.M.S. Reyhani:** *Semantic Approach to Service Discovery in a Grid Environment*. In Journal of Web Semantics, Vol. 4, No. 1, July 2005

[319] **Sven Schade, Arnd Sahlmann, Michael Lutz, Florian Probst, and Werner Kuhn:** *Comparing Approaches for Semantic Service Description and Matchmaking*. In Proceedings of the 3rd International Conference on Ontologies, Databases, and Applications of Semantics for Large Scale Information Systems (ODBASE 2004), Larnaca, Cyprus, 2004

[320] **Bruce Lowekamp, Nancy Miller, Dean Sutherland, Thomas Gross, Peter Steenkiste, and Jaspal Subhlok:** *A Resource Query Interface for Network-Aware Applications*. In Proceedings of the 7th IEEE Symposium on High-Performance Distributed Computing, Chicago, Illinois, USA, IEEE Computer Society, July 1998, pp. 189-196

[321] **Even Speight, Hazim Abdel-Shafi, and John K. Bennett:** *Realizing the Performance Potential of the Virtual Interface Architecture*. In Proceedings of the 13th International Conference on Supercomputing, Rhodes, Greece, ACM Press, New York, NY, USA, 1999, pp. 184-192

[322] **Laura Marie Feeney and Martin Nilsson:** *Investigating the Energy Consumption of a Wireless Network Interface in an Ad Hoc Networking Environment*. In Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Society, Anchorage, Arkansas, USA, 2001, pp. 1548-1557

[323] **Thomas Stockhammer, Miska M. Hannuksela, and Stephan Wenger:** *H.26L/JVT Coding Network Abstraction Layer and IP-based Transport*. In Proceedings of the Internation Conference on Image Processing, Vol. 2, 2002, pp. 485-488

[324] **Jari Porras, Petri Hiirsalmi, and Ari Valtaoja:** *Peer-to-peer Communication Approach for a Mobile Environment*. In Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04), IEEE Computer Society, Washington, DC, USA, 2004

[325] **E. L. Hahne and R. G. Gallager:** *Round robin scheduling for fair flow control in data communication networks*. In Proceedings of the IEEE International Conference on Communications, June 1986

[326] **E. L. Hahne:** *Round-robin scheduling for min-max fairness in data networks*. In IEEE Journal on Selected Areas in Communications, Vol. 9, No. 7, September 1991, pp. 1024-1039

[327] **M. Katevenis, S. Sidiropoulus, and C. Courcoubetis:** *Weighted round-robin cell multiplexing in a general-purpose ATMswitch chip*. In IEEE Journal on Selected Areas in Communications, Vol. 9, No. 8, Oktober 1991, pp. 1265-1276

[328] **Richard O. LaMaire and Dimitrios N. Serpanos:** *Two-Dimensional Round-Robin Schedulers for Packet Switches with Multiple Input Queues*. In IEEE/ACM Transactions on Networking, Vol. 2, No. 5, IEEE Press, Piscataway, NJ, USA, October 1994, pp. 471-482

[329] **C. Faisstnauer, D. Schmalstieg, and W. Purgathofer:** *Priority Round–Robin Scheduling for Very Large Virtual Environments*. Vienna University of Technology, Austria 1999

[330] **A. Demers, S. Keshav, and S. Shenker:** *Analysis and Simulation of a Fair Queueing Algorithm*. In Proceedings of the Sigcomm 1989 Symposium on Communications Architectures and Protocols, Vol. 19, No. 4, September 1989, pp. 1-12

[331] **P. E. McKenney:** *Stochastic Fairness Queueing*. In Internetworking: Research and Experience, Vol. 2, January 1991, pp. 113–131,

[332] **Antonio Carzaniga, Alfonso Fuggetta, Richard S. Hall, Dennis Heimbigner, André van der Hoek, and Alexander L. Wolf:** *A Characterization Framework for Software Deployment Technologies*. Technical Report CU-CS-857-98, Department of Computer Science, University of Colorado, April 1998

[333] **Bowen Alpern, Joshua Auerbach, Vasanth Bala, Thomas Frauenhofer, Todd Mummert, and Michael Pigott:** *PDS: A Virtual Execution Environment for Software Deployment*. In Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments, Chicago, USA (2005).

[334] **David E. Lowell, Yasushi Saito, and Eileen J. Samberg:** *Devirtualize virtual machines enabling general, single-node, online-maintenance*. In Proceedings of the 11th international conference on Architectural support for programming languages and operating systems, Boston, Massachusetts, USA, 2004, pp. 211-223

[335] **Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh:** *The design and implementation of Zap: a system for migrating computing environments*. In Proceedings of the 5th symposium on Operating systems design and implementation, ACM SIGOPS Operating Systems Review, Vol. 36, Special Issue, 2002, pp. 361-376

[336] **Cecilia Mascolo, Gian Pietro Picco, and Gruia-Catalin Roman:** *CodeWeave: Exploring Fine-Grained Mobility of Code*. In Automated Software Engineering, Vol. 11, No. 3, Kluwer Academic Publishers, Hingham, Massachusets, USA, June 2004, pp. 207-243

[337] **Ya-Yunn Su and Jason Flinn:** *Slingshot: deploying stateful services in wireless hotspots*. In Proceedings of the 3rd international conference on Mobile systems, applications, and services, Seattle, Washington, USA, 2005, pp. 79-92

[338] **D. A. Pierre:** *Optimization Theory with Applications*. John Wiley & Sons, New York, New York, USA, 1969

[339] **Christos Papadimitriou and Mihalis Yannakakis:** *Optimization, approximation, and complexity classes*. In Proceedings of the 20th annual ACM symposium on Theory of computing, Chicago, Illinois, USA, 1988, pp. 229-234

[340] **Michael A. Gennert and Alan L. Yuille:** *Determining the Optimal Weight in Multiple Objective Function Optimization*. In Proceedings of the 2nd International Conference on Computer Vision, December 1998, pp. 87-89

[341] **Jos Bonnet, Fabrice Dubois, Sofoklis Efremidis, Pedro Leonardo, Nicholas Malavazos, and Daniel Vincent:** *Cooling the Hell of Distributed Application Deployment*. In Proceedings of the 7th International Conference on Intelligence and Services in Networks: Telecommunications and IT Convergence Towards Service E-volution, 2000, pp. 181-190

[342] **Anastasia K. Kaltabani, Evangelia C. Tzifa, Panagiotis P. Demestichas, and Militades E. Anagnostou:** *Cost Effective Deployment of Service Logic in Future Distributed Processing Environments*. In Proceedings of the 10th Mediterranean Electrotechnical Conference, 2000, pp. 144-147

[343] **Tatiana Kichkaylo and Vijay Karamcheti:** *Optimal Resource-Aware Deployment Planning for Component-based Distributed Applications*. In Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing, IEEE Computer Society, Los Alamitos, California, USA, 2004, pp. 150-159

[344] **Dionisis X. Adamopoulos, George Pavlou, and Constantine A. Papandreou:** *Advanced Service Creation Using Distributed Object Technology*. In IEEE Communications Magazine, March 2002, pp. 146-154

[345] **Ana Cavalli, Bruno Defude, Christian Rinderknecht, and Fatiha Zaïdi:** *A Service-Component Testing Method and a Suitable CORBA Architecture*. In Proceedings of the 6th IEEE Symposium on Computers and Communications, Hammamet, Tunesia, July 2001, pp. 655-660

[346] **X. Logean, F. Dietrich, and J.-P. Hubaux:** *TINA Service Validation: The ErnesTINA Project*. In Proceedings of the IEEE International Conference on Communications 1998, Atlanta, Georgia, USA, June 1998, pp. 1150-1154

[347] **A. G. Ganek and T. A. Corbi:** *The Dawning of the Autonomic Computing Era*. In IBM Systems Journal, Vol. 42, No. 1, 2003, pp. 5-18

[348] **J. Kephart and D. Chess:** *The Vision of Autonomic Computing*. In IEEE Computer, Vol. 36, No. 1, 2003

[349] **Luciano Baresi, C. Ghezzi, and S. Guinea:** *Smart Monitors for Composed Services*. In Proceedings of the 2nd International Conference on Service Oriented Computing, 2004

[350] **S. Sen, A. Vardhan, G. Agha, and G. Rosu:** *Efficient Decentralized Monitoring of Safety in Distributed Systems*. In Proceedings of the 26th International Conference on Software Engineering, Edinburgh, Scotland, May 2004, pp. 418-427

[351] **Yan Jin and Jun Han:** *Runtime Validation of Behavioural Contracts for Component Software*. In Proceedings of the 5th International Conference on Quality Software, September 2005

[352] **John Grundy and Guoliang Ding:** *Automatic Validation of Deployed J2EE Components Using Aspects*. In Proceedings of the 17th IEEE International Conference on Automated Software Engineering, San Diego, California, USA, November 2001, pp. 47-56

[353] **Andreas Speck, Elke Pulvermüller, Michael Jerger, and Bogdan Franczyk:** *Component Composition Validation*. In International Journal on Applied Mathemtical Computer Science, Vol. 12, No. 4, April 2002, pp. 101-109

[354] **Stephan Hartwig, Jan-Peter Strömann, and Peter Resch:** *Wireless Microservers*. In IEEE Pervasive Computing, IEEE 2002, Vol.1, No.2, pp.58-66

[355] **P. Lord, P. Alper, C. Wroe, and C. Goble:** *Feta: A light-weight architecture for user oriented semantic service discovery*. In Proceedings of the European Semantic Web Conference, A. Gómez-Pérez and J. Euzenat (Eds.), Springer-Verlag, 2005, pp. 17-31

9 Acronyms

ADS	-	Advertisement and Discovery of Services
AMD	-	Advanced Micro Devices
AmI	-	Ambient Intelligence
AOP	-	Aspect Oriented Programming
API	-	Application Programming Interface
BPEL4WS	-	Business Process Execution Language for Web Services
CASCADAS	-	Component-ware for Autonomic Situation-aware Communications, and Dynamically Adaptable Services
CBAC	-	Context-based Access Control
CCM	-	CORBA Component Model
CLDC	-	Connected Limited Device Configuration
CORBA	-	Common ORB Architecture
CPU	-	Central Processing Unit
DAC	-	Discretionary Access Control
DAML-QL	-	DAML Query Language
DAML-S	-	DARPA Agent Markup Language Semantics
DbC	-	Design by Contract
DNS	-	Domain Name Service
ebXML	-	Electronic Business XML
ECF	-	Extensible Constraint Framework
EDSP	-	Extended Service Discovery Protocol
EJB	-	Enterprise JavaBean
EM	-	FAME² Event Manager
ESOA	-	Extended Service Oriented Architecture
FAME²	-	Framework for Applications in Mobile Environments 2
FET	-	Future and Emerging Technologies
FIFO	-	First in, first out
GIOP	-	General Inter-Operation Protocol
GPRS	-	General Packet Radio System
GSE	-	Generic Service Element
HTML	-	Hypertext Markup Language
HTTP	-	Hypertext Transport Protocol
HTTPMU	-	HTTP over multicast
HTTPU	-	HTTP over UDP
IDL	-	Interface Description Language
IEEE	-	Institute of Electrical and Electronics Engineers
IP	-	Internet Protocol
IST	-	Information Society Technology
J2EE	-	Java 2 Enterprise Edition
Jass	-	Java with Assertions
JINI	-	Jini Network Technology
JML	-	Java Modelling Language
JTP	-	Java Theorem Provider
JVM	-	Java Virtual Machine
LCM	-	FAME² Life Cycle Manager
LIME	-	Linda in Mobile Environments
MAC	-	Mandatory Access Control
MOM	-	Message-oriented Middleware
MWSDI	-	METEOR-S Web Services Discovery Infrastructure
OLE	-	Object Linking and Embedding
ORB	-	Object Request Broker
OS	-	Operating System
OSDA	-	Open Service Discovery Architecture
OSDI	-	Open Service Discovery Interface
OSGi	-	Open Services Gateway initiative
OSI	-	Open Systems Interconnection
P2P	-	Peer-to-peer
PCOM	-	Pervasive Components
PDA	-	Personal Digital Assistant
PDU	-	Protocol Data Unit
QoS	-	Quality of Service
RBAC	-	Role-based Access Control
ReMMoC	-	Reflective Middleware for Mobile Computing
RM	-	FAME² Reference Manager
RMI	-	Remote Method Invocation
RM-ODP	-	Reference Model of Open Distributed Processing
ROI	-	Remote Object Invocation
RPC	-	Remote Procedure Call

RUNES	-	Reconfigurable Ubiquitous Networked Embedded Systems
SD	-	Service Discovery
SDP	-	Service Discovery Protocol
SEE	-	Service Execution Environment
SLP	-	Service Location Protocol
SM	-	FAME² Security Manager
SOA	-	Service Oriented Architecture
SOAP	-	Simple Object Access Protocol
SOFA/DCUP	-	SOFtware Appliances/Dynamic Component UPdating
SOM	-	Service Oriented Middleware
SQL	-	Structured Query Language
SSDI	-	Support for Service Discovery and Interaction
SSDP	-	Simple Service Discovery Protocol
TCP	-	Transmission Control Protocol
TP	-	Transaction processing
TrustAC	-	Trust Access Control
UDDI	-	Universal Description, Discovery and Integration
UDP	-	User Datagram Protocol
UM	-	FAME² Update Manager
UML	-	Unified Modeling Language
UMTS	-	Universal Mobile Telecommunications System
UNICS	-	Uniplex Information and Computing System
UNIX	-	(see UNICS)
UPnP	-	Universal Plug and Play
URI	-	Uniform Resource Identifier
UUID	-	Universally Unique Identifier
WLAN	-	Wireless Local Area Network
WS	-	Web Services
WSDA	-	Web Services Discovery Architecture
WS-Discovery	-	Web Services Dynamic Discovery
WSDL	-	Web Services Description Language
WSIL	-	Web Services Inspection Language
WSMO	-	Web Services Modeling Ontology
WSMX	-	Web Services Modelling Execution
WS-Policy	-	Web Services Policy
WWRF	-	Wireless World Research Forum
WWW	-	World Wide Web
XML	-	Extensible Markup Language