An Extensible Framework for Context-aware Communication Management Using Heterogeneous Sensor Networks

Technical Report TR-KOM-2008-08

Johannes Schmitt, Matthias Kropff, Andreas Reinhardt, Matthias Hollick,
Christian Schäfer, Frank Remetter, Ralf Steinmetz

Technische Universität Darmstadt
Department of Electrical Engineering and Information Technology
Department of Computer Science (Adjunct Professor)
Multimedia Communications Lab
Prof. Dr.-Ing. Ralf Steinmetz

An Extensible Framework for Context-aware Communication Management Using Heterogeneous Sensor Networks

Technical Report KOM-TR-2008-08

By
Johannes Schmitt, Matthias Kropff, Andreas Reinhardt, Matthias Hollick,
Christian Schäfer, Frank Remetter, Ralf Steinmetz

Contact
johannes.schmitt@kom.tu-darmstadt.de
http://www.kom.tu-darmstadt.de

First published: November 15, 2008
Last revision: December 9, 2008

For the most recent version of this report see
ftp://ftp.kom.tu-darmstadt.de/pub/TR/KOM-TR-2008-08.pdf

This work has been partly supported by the German Research Foundation (DFG) within the Research Training Group 1362 "Cooperative, Adaptive and Responsive Monitoring in Mixed Mode Environments".

Front page photo source: http://www.sxc.hu

Abstract

This work presents the communication aspects of the *Context Framework*, which is an extensible framework featuring client and server functions building a platform for distributed provisioning, aggregation, evaluation and usage of information in heterogeneous systems. The main intention behind the evaluation of the gathered information about the context of an entity is to improve communication processes like call routing.

Every device, which is equipped with some kind of network interface can be potentially employed as information source. Very good examples are mobile phones, computers or home automation systems. These devices can be considered as *sensors* for the information they provide (e.g. calendar, position, movement, activity, address book). Using these sensors, the framework can autonomously determine a person's situation in order to enable *Context-aware Communication Management*. Since the system parts for the context evaluation were presented previously in [SHS07],[SKHR07], this work focusses on the information acquisition from the sensors.

Chapter 1 explains our vision of context-aware communication and the envisioned application scenario. This part extracts the resulting requirements for the framework, defines the terminology and presents an overview of the key elements of the framework. Finally, this part surveys related research projects.

In Chapter 2, the key issues of this work are presented in detail. This includes the middleware to facilitate services communication, service discovery and semantic sensor lookup. For each of these topics a problem description is given, followed by a presentation of existing and feasible approaches.

Chapter 3 contains the presentation of the chosen system design and its implementation. For each topic presented in Chapter II, this part explains its implementation and its operation in detail.

The performance of the communication towards the sensors is a crucial element of the overall performance of the system. For this purpose, we have tested our implementation using a specific request generator. In Chapter 4, the results of these performance tests are presented.

Chapter 5 finalizes this work with a summary and gives an overview about future work.

This work has be partly supported by the German Research Foundation (DFG) within the Research Training Group 1362 "Cooperative, Adaptive and Responsive Monitoring in Mixed Mode Environments".

Contents

1 Introduction

Electronic devices have entered our surrounding since half a century and devices of common use such as radios, televisions, telephones or hi-fi systems have evolved to small computers. Sometimes the computers are barely noticeable, for instance: mobile phones, computers, PDAs, car electronics, computer-equipped fridges or toasters, automatic electronic shutters, home meteorological stations, and many more.

Also network technology has evolved tremendously in the recent years. Today it is neither a challenge, nor costly to enhance devices with network interfaces. Particularly medium to short range wireless technology gained popularity, as it enables easy data exchange independent of any wired connection. Mobile phones and PDAs have build-in Bluetooth interfaces to synchronize with a PC or to exchange files. The current trend is to equip these devices also with WiFi interfaces for high speed data transmission.

Having a plethora network devices available, many advantages can be gained by letting them collaborate together forming a heterogeneous sensor network. In the addressed scenario sensor network consists of many distributed devices that are capable of acquiring informations about the user's current situation (his *context*), like his location, environment, condition, or activity.

Recently the work presented in [SHS07],[SKHR07] introduces the *Context Service*. This Context Service determines the context on the basis of a learning algorithm and collected sensor data. If, for example, someone tries to call a person the Context Service uses information from a sensor network to determine the user's context in order to decide how the call is handled. Therefore the system is able to learn from feedback of the user. Knowledge from previous, similar situations are used to support the current decision.

Figure 1.1 depicts the processing phases from sensor data acquisition to the usage of context within applications. This work focuses on the acquisition phase, whereas the already presented Context Service provides the evaluation task. The Context Framework provides sensor acquisition, preprocessing and evaluation such that several applications can make use of the systems's features and the derived context information.

In our focused scenario one of the main applications will be a communication system that makes use of the Context Framework. By using context information in a communication system's call flow, the communication system is able to provide *context-aware communication management*.

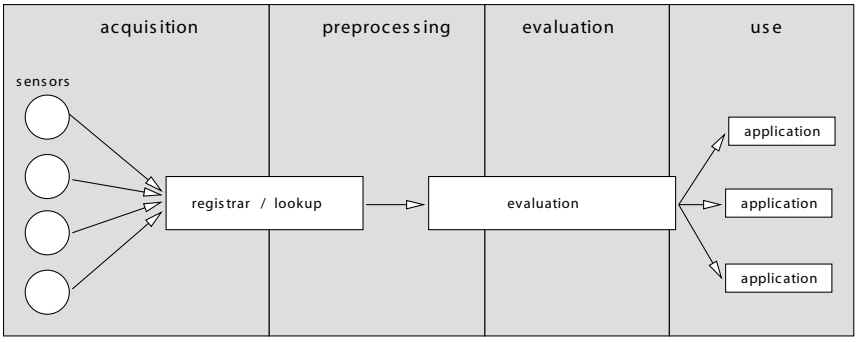


Figure 1.1: Processing phases of the context determination process

1.1 Scenario: Context-Aware Communication

Imagine the envisioned context-aware communication system in a company setting, it decides how an incoming call is handled. An employee usually has a phone at the workplace, a mobile phone and an answering machine. Text messages can be delivered by email, fax or on the mobile. The communication system uses the Context Service, which features a learning algorithm, which adapts the evaluation model and can be trained to reach the employee according to the preferences of the user. Information gathered from the sensors within the company's support network are feed into the system. For instance an employee is in the conference room in a meeting. A location sensor detects the location of the

person, and from the room schedule, the system knows that there is a meeting. Thus, incoming calls are forwarded to the answering machine and the user is notified by a text message that is sent to the mobile. The evaluation model can be trained by using simple feedback messages given by a user. Furthermore, one could also assume an interaction between the mobile phone and the location sensor. When the person enters the conference room, it automatically switches to silent mode. Another rule could be that when the location (and movement) sensor detects no other person in the room, the mobile does not go in silent mode and the phone calls are forwarded to the mobile.

The possible application scenarios of such a system are large. The Context Framework (which is used by the communication system) can involve a wide variety of sensors, such as location, movement, light, temperature, appointment calendars, address books, phone usage or computer activity. The system will allow every user to train its own evaluation model, thus individually desired behavior can be achieved. The central aspect of this work is the retrieval of information from the network; especially crucial information for the Context Service to make decisions. Please note that our system is not limited to the communication system scenario, but can also be applied in other scenarios.

1.2 Requirements

The task is to design and implement a system for information discovery in sensor networks. One key issue is to find suitable devices offering the information or service currently relevant or desired. A number of challenges have to be solved towards this end:

- **Discovery of Sensors.** The central issue is discovery of information suppliers in a network. Approaches in wired networks usually use a central directory to manage the services. But also broadcasting is used, as long as the network is small and it involves a single network segment. Recent proposals include solutions, originating from distributed systems, peer-to-peer networks and ad-hoc networks. A wide variety of protocols have been developed using distributed hash tables and heuristic approaches. This work will not propose a new discovery algorithm, but rather choose an approach from the existing protocols.

- **Communication / Heterogeneity.** When several technologies are combined in one network, problems arise as they are sometimes not compatible to each other. One network may rely on a different communication protocol than the others, hence one has to translate or agree on a third protocol to overcome the inconsistencies. Some networks provide good mechanisms to interact with others. For example Ethernet and WiFi networks, which results from the fact that WiFi was designed to work in combination with Ethernet. RFID badges on the other side are not able to be integrated into an IP network directly. One needs access servers with an RFID transmitter and an access to the Ethernet. The task therefore was to extend the network for a broader set of sensors and especially for embedded devices, which may not be connected via Ethernet. Varying communication protocols are only one aspect for heterogeneity. Problems can also occur due to differing hardware capabilities. For instance, resource-limited devices can not store a large amount of measurements, but some application may require the data of a longer period of time (a query of the kind: “when did it rain for the last time”).

- **Performance and Quality.** QoS (Quality of Service), QoI (Quality of Information) and QoC (Quality of Context) are major issues in this work. Of paramount interest are a short processing duration and a time-to-live for search queries. These issues arise from the communication system scenario, as the system must make a decision within a few seconds. But also for other scenarios this time constrains exist, as a sensor reply is usually required within a certain time. Besides timing issues, QoS in this project also addresses scaling, as the system hast to work with hundreds of sensors. The QoI metric describes the expressiveness of the aggregated information. The challenge here is to find “enough” information in limited time to allow the evaluation algorithm to make decisions with high accuracy, providing a sufficient degree of QoC.

- **Semantics.** Semantics are meta informations that describe the meaning of an entity. Semantics will be used to classify sensors, data types and search objectives in order to determine related sensors and to enable automatic data processing. The system should support search queries of the type: *“Give me all information about person X.”*. This requires a listing and interpretation of the capabilities and attributes of devices in the network. Similar research challenges are discussed in the setting of the *Semantic Internet*, where web-content (home pages, services) is arranged with semantic attributes to ease search queries and automatic processing for machines. The listing of attributes and capabilities is usually performed using an ontology, which is a structured classification using classes and derivation. A common standard for ontology definition is OWL, which is described in Section 2.3.

- **Open System.** Another challenge here is to make new data types applicable for existing applications. The idea is to extend the ontology automatically when unknown sensor types come up. The applications rely on the data types stored in the Ontology and new sensors define their data types as equivalent types or as subtypes of the

existing types. This enables that the system's settings or search request do not have to be adapted but nevertheless the server returns the new sensor values based on the semantic context.

1.3 Terminology

In order to provide a common terminology, some definitions and explanations are given below:

- **Context.** A Context is an abstract and meaningful description of the relationship between objects and their environment. A context is a rich object consisting of context features and can be approximated by a characteristic function. A context label is assigned to each context [Gör05].

- **Context Dimension.** One example for a context dimension is the user's presence, which is a state like being available, being busy or being completely away. Other dimensions are the user's emotional context, the proximity to someone, the social context or the location. Each of these dimensions is treated individually as for each of them a separate set of context labels exists, each of them relates to other sets of sensors, and for each dimension other evaluation models exists.

- **Heterogeneous Sensor Network.** The aspired system should be capable of retrieving information from any kind of sensor network. In terms of technologies the system should work across different interfaces, such as Ethernet, USB, WiMAX, WiFi, Bluetooth, ZigBee, Infrared, RFID; just to name a few. These technologies can form a heterogeneous network, for instance a computer in a LAN wants to query a Bluetooth temperature sensor outside the building. The computer is not directly linked to the sensor, thus it forwards the query to another computer, which has a Bluetooth interface and is in contact to the sensor. The second computer retrieves the temperature from the sensor to forward it back to the originating computer.

- **Semantic Sensor Network.** Sensor networks are more and more evolving to contribute to the vision of the semantic web. Different types of sensors, from tiny RFIDs to high resolution video cameras and other information sources like complex databases will be connected and will cooperate in one network that can be used by different applications. The *agents* in Tim Berners-Lee's semantic web [BLHL01] require various kinds of information, retrieved from many different sources, to fulfill their tasks. This information will be delivered by such a semantic sensor network. These semantic sensor networks will be used in many different fields, i.e. environmental monitoring, smart facilities or communication systems. Sensors and applications will be developed by different manufacturers, which makes it difficult for an application designer to integrate all existing and upcoming variations of a sensor type into his implementation. This requires a binding process that matches sensor output type A to application input type A*. E.g. a luminance sensor that might deliver `lumValue=24` has to be interpreted, so that the lighting system that follows the rule `(movement==true and room==dark) -> exec(lightOn)` can use this sensor. A common approach is to build machine readable ontologies that contain semantic side information about sensor types, data types and applications. The knowledge stored in these ontologies is often used to interpret a data value for another context. This can be done by simple keyword matching like `lumValue` is equal to `lightValue` or by translations to a higher semantical level like `room.lumValue<50` means `room=dark`.

- **Embedded Systems.** Usually an embedded system is a small special purpose computer that is part (embedded) of a larger technical unit. These systems are equipped with an energy efficient CPU that is less powerful to provide a larger lifetime in case it is battery powered. Due to the fact of high integration, those systems only have a limited amount of memory and a small flash memory for non volatile storage.

- **Motes.** A so called *mote* is a low power wireless sensor node. In general it consists of a low power microcontroller, at least one integrated sensor and a radio module. A mote can differ in size, type and amount of sensors or used microcontroller. Emerging single chip systems and target costs of less than 10 cents per unit enables networks with several hundreds to potentially tens of thousands of motes. The one used in this work is a *tmote sky* from the Moteiv Corporation (now Sentilla). The communication between motes is realized via the IEEE 802.15.4 / *ZigBee* protocol [Zig06]. The Moteiv Corporation provides with its motes a GCC compiler tool chain, TinyOS libraries and example applications.

- **Gumstix.** Another embedded System used during this work is a Gumstix (400 to 600 MHz Intel Xscale CPU, 64 to 128 MB RAM and 16 to 64 MB Flash). The Gumstix is running a BusyBox environment on top of a Linux kernel. The Gumstix's CPU is of Intel Xscale microarchitecture, [Cor07], which is based on the StrongARM-architecture.

1.4 Main Context Framework Elements

Figure 1.2 illustrates a sample assembly of the Context Framework's different parts that are described in the following.

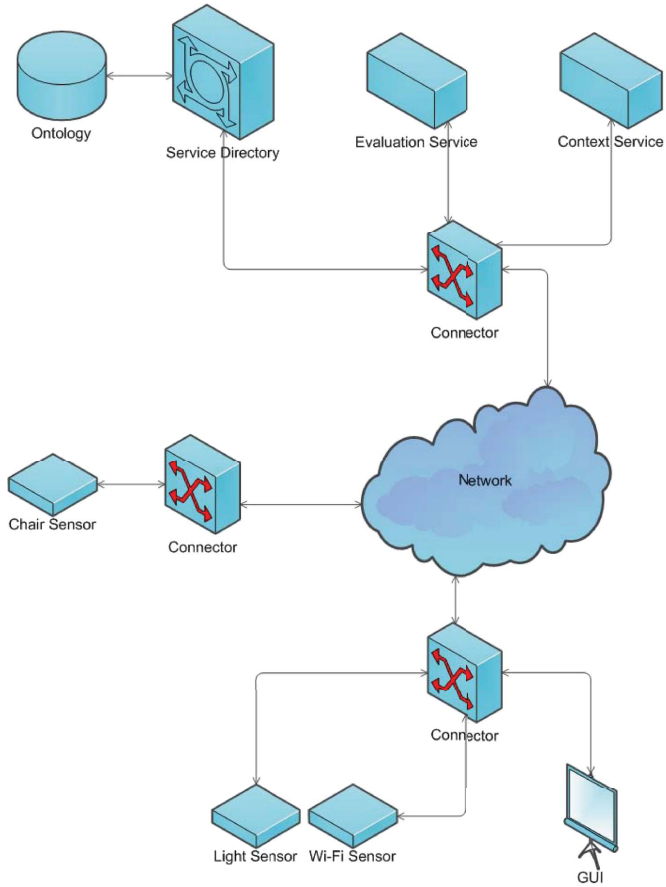


Figure 1.2: Overview - Framework Elements

- **Sensors** are devices that provide information about the surrounding. Sensors are for example sensors for light, movement, vibration, humidity, temperature, sound, open doors etc. These sensors or also sometimes referred to as physical sensors. On the other side, one can also widen the term and look at it in an abstract manner: Everything that provides information can be considered sensors. In this case also computers, providing a users address book are sensors. A sensor has individual attributes and relationships, such as location or ownership.

- **Service Directory (SD).** The Service Directory is responsible for managing the sensors in the network and performs search queries. Whenever the Service Directory discovers a new sensor, it looks for that sensors's semantic description. With the help of this description, the Service Directory then feeds an Ontology model with the data from the service and the corresponding semantic description.

- **Ontology.** The Ontology is a data structure, which is used to define relations between entities based on their semantic description.

- **Context Service (CS).** Generally, the Context Service manages the contexts of all users for its specific dimension of context. The Context Service is, thus, the layer between the framework's logic and the user interface. To be able to return the context of a user quickly, the Context Service periodically updates the context for each user it has in its internal list, by calling the Evaluation Service. Similarly, whenever the Context Service is asked to set the state for the user, it does not do this on its own, but calls the Evaluation Service. There is not only one Context Service, but many, with each of them focusing on its own dimension of context.

- **Evaluation Service (ES).** The Evaluation Service [SHS07],[SKHR07] is responsible for analyzing the context a user is in, as well as updating the user's context upon such a request. The Evaluation Service trains the evaluation

model whenever the user sets his context manually and uses it to evaluate the context of a given user. The Evaluation Service specifies a set of sensors to be used for the evaluation, so a Context Service that is only responsible for the user's emotional context, does not need to include a sensor that provides location data, but should only do so, if this increases the quality of the results.

- **Connector** To allow transparent procedure calls, the services are connected through Connectors. A Connector provides a common set of interfaces to enable communication between services. A Connector abstracts from the programming language, the network type and the used platform.

1.5 Related Projects

This section will provide on overview on comparable approaches and their development in general. Subsequently related sensor network projects are discussed and compared to our approach.

Multi-Sensor Context-Awareness in Mobile Devices and Smart Artifacts

Gellsersen et al. focus in [BGS02] on the use of multiple sensors for context awareness of mobile devices and so called "smart artifacts". They describe various aspects of multi-sensor context awareness for three of their projects. The TEA context awareness module is a small hardware board that contains some sensors and a context aware application and can be added to a mobile phone. The general aim of TEA is to get a "generic solution for making devices smarter" by integrating context awareness. The TEA module introduces an architecture to abstract from raw sensor data. The context is not inferred from the actual sensor data but from so-called "cues", which are abstractions of the raw sensor data. There is an algorithm for each sensor that produces cues out of the raw sensor data. Each sensor can produce multiple cues. For example, the microphone sensor does not provide raw audio data, but produces cues like average volume level and the main frequencies. The sensors are decoupled from the application, because the cues are abstract and not application specific.

Amigo - Ambient Intelligence for the networked home environment

Vallée et al. describe a network for modern homes in [VRV05]. Such a network will be composed of heterogeneous devices and has to be adaptive to the continuously changing resources and users' context. To provide a user-friendly environment, they suggest *Dynamic Service Composition* by groups of agents that interact for a service and adapt to changing context or upcoming activities. The network system is designed by combining the Service-Oriented Architecture (SOA) paradigm with a Multi-Agent System approach. Semantic service descriptions and contextual information is used to describe the service's functionalities and the context of its use. In [VRV05], no prototype implementation is described, but on the project website[1] it can be seen that Amigo is using OWL and Amigo-S, which is supposed to be an extension on OWL-S.

MyConnector

MyConnector is a computer service that tries to connect people intelligently [DKS06]. It is designed to adapt to the user and learn their availability from context cues. The overall goal is very similar to that of the Context Framework. The aim behind MyConnector is to "connect people at the right place, the right time, and with the best possible medium for socially appropriate communication"[DKS06]. MyConnector suggests an architecture where contextual information is registered at a central server, which is called "Core Connector". The Core Connector uses the context cues is obtained to calculate an availability level for each user. MyConnector uses several pieces of contextual information for this calculation. It uses personal calendar information, which it gets from a Microsoft Exchange server. It gets usage data from the clients, like overall activity data, number of mouse clicks and keyboard events, the windows switching frequency and a list of active programs. Additionally, the Skype [2] activity (whether Skype is running, an ongoing instant messaging conversation, an ongoing Skype call, etc.) is also measured. These information sources are predefined and the architecture of MyConnector does not support easily adding or removing certain sources, which is a big difference to the Context Framework that aims to be able to add and remove sensors at runtime. The Core Connector makes use of a Bayesian learning approach to learn a person's availability based on the acquired data. To train the system, a supervised learning method was used, which means the users had to report their availability manually. Danninger et al. point out that this method has the drawback of being very time-consuming.

SUMO

The Suggested Upper Merged Ontology (SUMO) [PNL02] aims to work as upper level Ontology that can be referenced by other ontologies. It was created by merging existing ontologies. The goal is to improve the communication among

[1] www.amigo-project.org

[2] A common client for IP telephony

different ontologies. Instead of having several translations between different ontologies that need to be processed each time data is transferred from one Ontology to another, the idea is that these ontologies reference the same element from a standard upper Ontology. Therefore SUMO presents a hierarchy of general classes and properties. In our Ontology, we did not reference any SUMO element, because they are far too general, and do not fit well to our domain of interest. E.g. we could define our class *sensor* as *SUMO:Measuring Device* but then we have no fitting SUMO element to reference our super class *sensor*.

OntoSensor

OntoSensor by Russomanno et al. [RCKT05] tries to close the gap between the top level Ontology SUMO and the different sensor network ontologies of specific domains. "OntoSensor can be viewed as middle-level Ontology that extends the concepts from a high-level Ontology (SUMO) and whose concepts can be used by more specialized ontologies that model specific domain sensors." [RCKT05] The Ontology provides a sensor classification and a sensor context describing model. Unfortunately, regarding our requirements, the application field of this Ontology is focused on environmental monitoring and the sensors are supposed to be physical types like chemical, mechanical and electromagnetic.

Russomanno et al. present a proof-of-concept network prototype [CG06] that makes use of their OntoSensor Ontology. The network includes wireless sensor nodes of the Crossbow's Mica family that are similar to our motes from Moteiv. The onboard sensing capabilities include physical measurements like ambient light, barometric pressure, GPS and acoustic measurements. Custom software running on base stations collects the raw sensor data periodically and generates OntoSensor referenced OWL repositories. "Successful reference of OntoSensor requires *a priori* knowledge of the sensor platform class associated with a given node identification number." [CG06] In our approach we try to overcome this circumstance by describing the sensor using OWL-S and extending the Ontology. A software agent is used to query the OWL repositories. E.g. the agent can find motes according to a given location or find the average pressure in a region of interest.

Data Fusion in PADRES network

Similar to our Service Directory, the PADRES[3] system uses a Broker that handles the data flow between sensor and application. But instead of requesting the sensors, they use a content-based Publish/Subscribe model (CPS). The main topic in [WPJ07] is the semantic data fusion. The broker incorporates a *semantic engine* that uses mapping functions to abstract from raw sensor data to a higher semantical level. E.g. (light,250) will be translated to (weather, cloudy). These mapping functions are included in ontologies of different domains and can additionally be supplied by applications. In our project, we plan as well to integrate some kind of semantic data fusion or aggregation, thus the semantic engine is a point of interest to us.

Sensor Web Agent Platform (SWAP)

Moodley and Simonis describe a framework [MS06] for building Sensor Web applications over the internet. The framework is a multi agent system and based on a three-tier architecture. In the *Sensor Layer* Sensor Agents access physical or virtual sensors or existing services. The Sensor Agents represent data access or control of actuators. The *Knowledge Layer* consists of Workflow Agents, Tool Agents and Modeling Agents, which work together based on expert knowledge to process the data from the sensor agents. The *Application Layer* provides user interfaces to interact with the system. Moodley and Simonis point out that the crucial point of the performance of an agent based system is the communication between agents, which depends on a powerful Ontology framework. Such an Ontology infrastructure "must support thousands of ontologies across multiple end-user applications and in many different domains." [MS06] They differentiate between different levels of ontologies (from top-level to application) and use OWL as Ontology language.

[3] http://padres.msrg.utoronto.ca

2 Key Issues

Key issues for this work are service communication, service discovery and service description. Before describing the implementation of the Context Framework, various possible approaches are considered for each issue.

This section explains the key issues for the design of the Context Framwork's architecture and structures the considered approaches in order to classify the approach which is used in this work.

2.1 Service Communication

Today organizations employ a variety of heterogeneous hardware with several processor architectures, different operating systems and different communication technologies. This leads to enormous programming effort for developers of information systems. To reduce the amount of work for application developers standard programming interfaces and protocols are necessary. Middleware provides acts as a layer. The API of the Middleware should be platform independent to assure that applications run on multiple platforms. Its high-level interfaces hide the underlying details like distribution and network protocols and allow focusing on the actual application-writing [Ber96].

Bernstein [Ber93] introduces the so called 'pervasive attributes' to measure middleware: usability, distributability, integration, conformance to standards, extensibility, internationalizability, manageability, performance, portability, reliability, scalability and security. These attributes are very general and do not differentiate between distributed and mobile systems. Therefore the attributes will not be discussed in detail, but instead an abstraction about distributed and mobile systems will be given to divide middleware into several types based on the kind of system afterwards.

The following categories of middleware suite best to the most approaches for fixed and mobile distributed systems:

- **Transactional Middleware.** The focus of transactional middleware lies in transactions across multiple hosts. Transactions are meant to be atomic process, which means either the operation fully succeeded or did not even happen. To accomplish this, a two-phase commit protocol is used. This capability is helpful, if several hosts must be kept in consistent state. This kind of middleware supports synchronous and asynchronous communication between hosts.

- **Message-Oriented Middleware.** Message oriented middleware's communication is based on message exchange. Asynchronous communication is the nature of this type, because a message is sent and as soon as it is delivered the host can continue operation without being blocked while waiting for the answer. The server side might answer with another message at a convenient time. Since the received message needs to be stored in a queue, the host requires memory accordingly. Message oriented middleware is particularly suited for event based and publish/-subscribe systems. In a publish/subscribe system a host can be producer (publisher) and/or consumer (subscriber) of a message. The producer sends a message to the content or topic based network. The consumers can subscribe themself to specific contents. The underlying event system then notifies all consumers with a matching subscription. Publish/subscribe systems allow multiple receivers via an indirect addressing scheme.

- **Object and Component Middleware.** Object oriented middleware is a subclass of procedural middleware, which has its main focus on Remote Procedure Calls (RPC). Procedural middleware introduces client and server stubs, which perform the marshalling, unmarshalling and network communication. Distributed objects are the basis of communication in object oriented middleware. A client object requests the execution of an operation from a server object that may reside on another host. Usually client and server stubs are automatically generated therefore. This kind of communication is synchronous, which means that the client is blocked until the server delivers the result.

- **Reflective Middleware.** The principle of reflection [MCE02, chapter 5.2] allows a program to access, reason about and alter its own configuration. According to middleware, this means dynamic (re)configuration. The so called Context-aware Middleware makes use of that principle, since it takes the context of the device into account. Context includes information about the environment (location, brightness, ..) and device characteristics (CPU speed, network interfaces, ..). Context-aware Middleware does not hide that information, but gives the application access to it. This way the application can decide how to react on certain circumstances. The application might be better suited for the decision to make, since it might have additional information. That reduces the complexity of the middleware and thus the required computational power, which is good for less powerful devices. With the knowledge of the application about the execution environment, the application might be able to tune the configuration of a specific device.

- **Data Sharing Middleware.** Data sharing oriented middleware concentrates on the handling of disconnects and data sharing among systems. To accomplish that, replicas of data are used to allow data access also during a disconnect. That requires much effort in effective caching and keeping the replicas in sync. Tuple space-based middleware provides a globally shared data space. That means, that the data is spread across multiple hosts. A tuple space consists of tuples, which contain various data types in a row. The data is created by a process, which computes even if it is disconnected and provides or accesses data, whenever it is connected. So communication is asynchronous. Once tuples are created they are independent from the providing process. Since the tuple space is global, data access is independent from any location.

However, often mixed forms of the outlined conceptual approaches exist in reality. MundoCore (see Section 3.2.2) for example is message oriented in the first place, since it uses the publish/subscribe procedure. But it also uses client and server stubs for remote execution of methods, which implies object orientation. The possibility of configuration with an XML file is of context aware character.

R-OSGI (Section 3.2.2) can not clearly be assigned to a specific class of middleware. It allows remote invocation of methods and thus has a procedural aspect. But the communication is purely message-based. The location of remote methods is hidden by a proxy bundle, which facilitates transparency. On the other side the possibility to tag methods with attributes can be used for awareness purposes.

2.2 Service Discovery

Service discovery is a key element when a loose coupling of many distributed information sources is required. It describes the feature of nodes in a network to offer, find and use services without explicit configuration effort from the user. As networks can have different natures, the generic terms *nodes* and *services* represent a wide range of elements. For example, in a LAN the nodes are computers or laptops, with services such as network printers, file servers, mailboxes, time synchronization, network authentication, DHCP server, etc. On the other side in wireless sensor network the nodes are very resource limited devices with little computational capabilities and the corresponding services are typically sensing or actuation tasks. In both examples the central element is the information, so in a general view service discovery protocols enable the network devices to find information or services from other devices autonomously.

2.2.1 Classification

Despite the variety of network specifications the service discovery protocols can be categorized in three groups: centralized, structured and ad-hoc.

- **Centralized.** Centralized algorithms use a centric instance for managing services. In an underlying server/client architecture service providing nodes publish their service to the server. Service seeking nodes forward their request to the server and get in response the address of a node hosting the appropriate service. Further, communication between the nodes is then transmitted directly in a peer-to-peer manner. In exceptional cases the complete data is directed through the server. This approach is simple and easy to manage. Often it is used in small to medium networks with low traffic and fast data access. However, it has also several downsides: First there is a single point-of-failure at the server. If it fails, the complete service discovery fails too. Also, it does not scale well, especially in multi-hop networks, where the area around the server sees higher bandwidth utilization than the rest of the network. The network has to be extended or optimized in this region, otherwise it gets overloaded. In location based applications, where a device scans for near by services, it can be very inefficient to query a central server, as it can be several hops away.

- **Structured.** In structured protocols the role of a centralized server is distributed to several nodes. Instead of having one server managing the services there is a structure of multiple servers each responsible for a specific part. These can be differentiated by geographical location, service type, resources, network bandwidth, etc.. Furthermore, the structure can be categorized into flat, hierarchical and hybrid: Flat structures use overlay networks similar to those from peer-to-peer networks. They are flat in terms that each node provide the same functionality in the overlay network. Finally there is also the possibility of using both in a hybrid model. The composition of the structures can be static, so that administrators assign them, but also dynamic or autonomous.

  The advantages of structured protocols is the increased robustness. Usually parts of the network can fail, but still the service discovery works in the remaining parts. Therefore, the protocols are more complex and require service directories with a certain degree of capabilities.

- **Ad-hoc.** The last group consists of lose or ad-hoc protocols that have no nodes acting as a server or information lookup node. Usually the discovery works by either flooding the network periodically with advertisements or by sending discovery message for services on demand. Furthermore nodes can cache gathered information to reduce overheard in the network traffic. Recently algorithms from peer-to-peer networks were integrated into discovery protocols. This is achieved by storing the list of services in a distributed hashtable the network. These techniques are often applied in ad-hoc networks where no central directory can be used and in networks with high mobility, where establishing servers is futile, because the information is most likely outdated when another nodes requests it. But also small local area networks use this technique, due to the fact that in a small group a server is not always necessary.

2.2.2 Existing Protocols and Performance Parameters

The problem of service discovery exists for several decades; many approaches and protocols have been developed for different networks [MPHS05, BR00, LH02]. The variations are huge and many dedicated discovery algorithms exist. The protocol performance mainly depends on:

- **Size** is a crucial factor, as it can range from a dozen nodes in Bluetooth scenarios up to millions of nodes in Internet service discovery. The impacts are perceivable in many areas. In small networks flooding of service advertisements and storage of every service on each device is an efficient approach, whereas with world-spanning, large-scale networks this would lead to many unnecessary messages and large databases on each node.

- **Speed** of the network can become an important issue, for example when multiple access to the transmission media is used. In a network with low throughput a node can not transmit frequently large sets of service information to other nodes, because this results in congestion of the network. Other nodes become blocked and can not transmit their information.

- **Dynamics** within the network have fundamental effects on the protocol; it must adapt to new nodes in a reasonable time and also detect disconnected nodes. In static networks this can be minutes or hours, but in mobile scenarios the tables must be updated in short intervals.

- **Devices** do have different resources, such as a PC is more powerful than a wireless sensor node. Some sensor nodes only have 48kb program memory in total, thus they are limited in the memory for the service discovery protocol. In such cases a device can not cache measurement over a longer period.

- **Pathway** addresses the type of interface of the network. A significant example is the difference of wired or wireless networks, as in a wireless network the loss is much higher due to natural interferences. But also issues like multiple access, master/slave relations or data centric have to be considered.

Finally, the various protocols originate from combinations of the five criteria. A protocol for a large wired network is different from a small wired network, but also it would not work for a large wireless network. Often it is the case that depending on the application, which also specifies the network, an individual service discovery protocol is required.

In the following a selection of discovery protocols are introduced. As there are over 30 established protocols, not all of them are described, but only the most popular ones and those, with relevance for this project. A good evaluation can be found in [MPHS05].

SLP, UPnP and Jini are the classical protocols often used in IP networks. They are similar in operation. The other protocols mentioned here are a selection of existing algorithms, each introducing a unique feature.

SLP

The *Service Location Protocol* [VGPK99] is a language independent discovery protocol for services, such as printers, databases or email servers. The language independence comes from the fact that it does not build on Java or XML for data exchange (unlike other protocols). It defines three roles: *User Agents (UA)* are the devices searching for a service, *Service Agents (SA)* provides services and *Directory Agents (DA)* serve as a directory server. SLP can work with or without a Directory Agent. Without it uses multicast messages to locate services. With a Directory Agent it works as a centralized protocol with publish and subscribe methods. It is designed to work in small to enterprise-size networks. The services are addressed in a URL manner, which usually encompass DNS servers.

UPnP

Universal Plug and Play [UPn06b] basically works the same way as SLP does. It supports operation with and without a control point (Directory Agents). In both modes no interaction or configuration from the user is required. In contrast

to SLP, UPnP depends on IP networks, as the IP addresses are the identifier of a device. Also it uses standardized technologies such as TCP, HTTP, XML, SOAP, etc. Originally developed from Microsoft, is now maintained by the *UPnP-Forum*. It declares to be an architecture that "offers pervasive peer-to-peer network connectivity of PCs of all form factors, intelligent appliances, and wireless devices" [UPn06a].

Jini

Java Intelligent Network Infrastructure (Jini) [Sun06a, Jin06] is Java framework for distributed computing, which also includes service discovery. It has multiple mechanisms, but mainly uses a centralized or broadcast approach. The broadcast approach uses reactive multicast messages with a time to live limit to find close-by services. The centralized method uses proactive publish and subscribe methods at a central server. Unique about Jini is its design for Java; it benefits from *Remote Method Invocation (RMI)* and class loading mechanisms. Also service seeking devices can download the required functions for accessing the service from the server and thereby do not need specific drivers a priori.

Salutation

Salutation is a service discovery protocol for networks with a broad set of appliances and applications. It mainly has three components: the *Salutation Managers*, the *Transport Manager* and *Clients/Servers*. Each node can have multiple clients or servers, but only one Salutation Manager. The servers register their services at the Manager. On a request of a client the Manager first checks if there is a local server providing the service. If not, other Salutation Managers are queried. This involves the Transport Manager, which handles the network. The client/server - manager interface (SLM-API) is generic and independent from the network beneath. This means, the clients/servers can use service discovery, without being specified to network interface related issues. These are handled by the Transport Manager, thus for each network interface one Transport Manager is required. This unique design uncouples the discovery from the network technology, as it is the case for Jini and UPnP. The salutation project is no longer maintained. It is referenced in many papers and reports about discovery protocols but the web site[1] is shut down.

Bluetooth SDP

The Bluetooth protocol stack has a service discovery protocol included [Blu04]. It enables devices to find services within the communication range. The requirements for this protocol are unlike to the protocols described so far. It primarily addresses a scenario with a dozen or less devices in close proximity (a few meters) in a wireless network with single hop connections. Requests are issued by sending multicast messages to the devices in proximity, whereupon remote devices offering this service respond. Queries can have three different forms: search by service type, search by service attributes or service browsing (no a priori knowledge). Bluetooth SDP does not have features such as brokers, publish/subscribe mechanisms, advertising services and there is no notification when a service becomes unavailable.

INS/Twine

INS/Twine [BBK02] is a service discovery protocol with very good scaling capabilities. The abbreviation INS stands for Intentional Naming System [AWSBL99] which is a system "where applications describe what they are looking for, not where to find data". This is done by a set of key-value pairs for each device. INS/Twine defines a standard to address devices by its features/attributes, instead of a logical/geographical name like DNS. Twine is a discovery protocol that combines INS with mechanisms from peer-to-peer technology, such as distributed hash tables and overlay networks. This approach leads to a structured design, where so called Resolvers span the overlay network and also store the hash tables. Through the technique called *late-binding* INS/Twine copes well with mobile devices that change their location or logical address frequently.

JXTA

JXTA [Sun06b] is a protocol framework for peer-to-peer applications also including a service discovery protocol. It uses a structured and ad-hoc implementation simultaneously. In JXTA every client is connected to a *Rendezvous Peer*, which is similar to a service broker. A Peer (client) sends a discovery query to all other Peers and Rendezvous Peers. If a regular Peer provides the service, it replies. A Rendezvous Peer checks the query with the cached information and if a match occurs, it sends a message to the service seeking peer and the peer that provides the service. JXTA uses XML for message exchange and thus is independent of programming language or network interface.

SSDS

SSDS [HCZK02] means Secure Service Discovery Service and is a protocol designed with focus on security. It uses a structure of SDS servers, capabilities managers and certificate authorities. SDS servers are similar to the service directories; clients and servers use publish and subscribe methods to enable service discovery. The capabilities managers and certificate authorities are reserved for security related tasks. The language for service description is XML.

[1] www.salutation.org

LANES

A clustering approach is taken in the *LANES* protocol [KKRO03]. It uses a two-dimensional overlay structure, similar to the one proposed in CAN [RFH+01]. The network is arranged in lanes; these are a number of nodes that know a predecessor and successor. Members of the same lane are addressable with an anycast address. Within these lanes services are announces in a proactive manner, thus each node within this lanes knows what services are available. Several lanes are aligned side by side, so each lane has two neighbor lanes (except the border lanes). If a service is not available within a lane, a reactive search is initiated from the searching node by putting a service request to the neighboring lanes. A node from the neighbor lane responds if it knows a node offering the service or forwards the request to the next lane.

2.3 Semantic Sensor Lookup

Parts of the project are tightly coupled with *Web Services*, a currently active field of research. The project addresses among others the problem of efficient utilization of Web Services in the Internet. For now services in the Internet (such as online stores, online reservation service or search engines) have individual interfaces and process procedures. An application for automatic purchase of an item at one web shop will most likely not work for other shops. Web Service research attempts to standardize the description and usage of such services, specially understandable for machines, so that an application can use many different services, even it may not have knowledge of the service a priori. Several standards used in this projects (RDF, OWL, OWL-S) originate from the Web Service research.

In May 2001, Tim Berners-Lee, James Hendler and Ora Lassila published an article called "The Semantic Web" [BLHL01], introducing the term *semantics* in reference to computer networks the first time. Their vision describes a world, where networked devices and their capabilities are associated with keywords or expressions so that other devices can identify them. For example in the event of an incoming call the telephone can instruct all near by devices with a volume control to turn down the sound. Or one can search in the Internet for doctors offering a special treatment not more than 50km away and also covered by the health insurance.

Prerequisite for those tasks are machines that can read and interpret information from other machines. Until now, most information in the Internet is presented for humans, not for machines. Complex searches, like the example with the doctor have to be done by a person in several steps, as a machine does not know how to extract the information from the Internet websites, nor what to do with it. The idea of the semantic web is to provide information processable by machines, in parallel to the information intended for humans.

Since 2001 the idea of a semantic web spreads rapidly in the Internet community. The possibilities this technology holds are amazing, as it can help in every-day situations. Information lookups will become much more comfortable and faster. Also buzzwords such as *Web 2.0* or *Next Generation Internet*, describing the Internet of the near future are often related to semantic features.

To achieve this endeavor two prerequisites have to be fulfilled. First, one has to agree on a uniform knowledge representation. The systems must all know how to structure a document, so that the information is readable by others. This is done by the Resource Description Framework (RDF). Second, rules and a vocabulary must be agreed on to create relations or an Ontology.

2.3.1 Resource Description Framework

Semantics are only possible when a standard to create those semantics is specified. The Resource Description Framework (RDF) is a common applied method to define semantics and knowledge representation.

Like humans, the machines have to agree on a language with structures for sentences, vocabulary and so on. RDF does not specify the complete language, but the syntax. The core statement of RDF denotes that descriptions have the form:

$$Subject \longrightarrow Predicate \longrightarrow Object$$

Subject and object are information, things, properties, etc. The predicate tells how these two are connected, for example if one is a subset of the other. These three components are also called RDF triple.

RDF specifies two ways to notate the description: Either in a graph or in a XML document. A RDF graph is a set of RDF triples. Figure 2.1 illustrates an example. As RDF uses URI technology (Uniform Resource Identifier) for all kind of names, the three components are often notated in form of an URI. An object can also be a literal. Literals are concrete values of a subject and often have no pursuing nodes. Finally subjects and objects can also be a *blank node*, which are nodes without a name. Still they are distinguishable, as they have unique relations to other subjects and objects.

The example in Figure 2.1 has a top node, a subject, called RDF-syntax-grammar, identified with the URI `http://www.w3.org/TR/rdf-syntax-grammar`. This subjects has two objects. On the right side is a predicate `http://purl.org/dc/elements/1.1/title`, referring to an object that is a literal with the value *RDF/XML Syntax Specification (Revised)*. On the left side one can see a blank node followed by an URI object and a literal object.

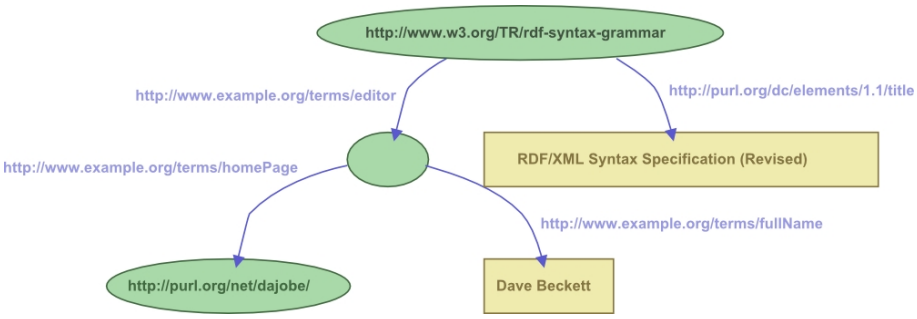


Figure 2.1: RDF Graph - source http://www.w3.org

The second way to notate the description is an XML document. This is actually the common notation and wraps the graph into an XML document. The following XML text represents the RDF graph above:

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns:dc="http://purl.org/dc/elements/1.1/"
        xmlns:ex="http://example.org/stuff/1.0/">
   <rdf:Description rdf:about="http://www.w3.org/TR/rdf-syntax-grammar"
              dc:title="RDF/XML Syntax Specification (Revised)">
       <ex:editor>
           <rdf:Description ex:fullName="Dave Beckett">
             <ex:homePage rdf:resource="http://purl.org/net/dajobe/" />
           </rdf:Description>
       </ex:editor>
   </rdf:Description>
</rdf:RDF>
```

The first line specifies the XLM version. `<rdf:RDF` opens the RDF section of the document. The first three entries starting with `xmlns` specify a name-space, as it is known from programming languages. For example the line `xmlns:rdf="http://..."` expresses that in the rest of the section a XML tag or attribute starting with “rdf:” refers to http://www.w3.org/1999/02/22-rdf-syntax-ns#. `rdf:Description` refers to a subject and `rdf:about` to the name of the subject. The objects are written in the form of values known from the XML style. The predicates of this example start with `dc:title`, `ex:editor` and `ex:homePage`. Because RDF only provides the syntax but not the vocabulary, no RDF predicates exist. As seen from the name-spaces they do all come from non-RDF references. One can either specify own predicates, like `ex` in the example, or use predicates already defined and public, such as Dublin Core (DC) [Dub07] to describe documents and Internet objects.

So far the RDF Model was described. The syntax for RDF descriptions in XML documents is specified in the RDF-Scheme (RDF-S) [W3C07]. It defines a vocabulary and constructs for representing information. The two base elements are classes and properties. Classes are the objects that are described, for example *people* or *cars*. Properties are used to give further details of the classes. These can be user specified, such as *age* or *brand*. When the description of the resources is done, instances of the classes can be created to construct the knowledge representation one desires. Besides the base elements RDF-S also defines containers, collections and utilities. A full discussion of RDF-S is left out at this place, as it is not of major importance for this work.

2.3.2 Ontologies

An Ontology represents knowledge of the real world in form of a collection of statements and expressions. These are ordered in a hierarchy and are linked to each other by defined relations or properties. The field of knowledge that is represented by an Ontology is also called the domain of the Ontology. [Die03]

The ontologies are necessary to interpret the metadata of sensors. E.g., if the sensor’s metadata contains a room value entry, the Ontology will provide the meaning that this entry belongs to a location information. Or in the simplest case, ontologies can state that an expression is the "same as" another. E.g. "conference room" is the same as "meeting room". Ontologies are written in formal languages, which provide a strict syntax and are intended to be readable and interpretable by humans and by machines as well. The Web Ontology Language OWL is the de-facto standard for this purpose. It is used in this work and by almost all related projects.

Web Ontology Language (OWL)

"The OWL Web Ontology Language is designed for use by applications that need to process the content of information instead of just presenting information to humans. OWL facilitates greater machine interpretability of Web content than that supported by XML, RDF, and RDF Schema (RDF-S) by providing additional vocabulary along with a formal semantics." [W3C03]

OWL builds a layer on top of RDF-S and XML. In this work OWL is used to describe an Ontology that contains a hierarchy of sensor and data types. Further the sensors are described in OWL-S which is based on OWL.

The Ontology is build of classes and properties, the latter are further subdivided into:

- *DatatypeProperty* - relates objects to data type values,
- *ObjectProperty* - relates objects to other objects,
- *AnnotationProperty* - describes objects and
- *OntologyProperty* - relates ontologies to ontologies.

The following code fragment will give a tiny example of the use.

```
<!-- Definition of new ObjectProperty -->
<owl:ObjectProperty rdf:ID="daughterOf"/>

<!-- Definition of new DatatypeProperty -->
<owl:DatatypeProperty rdf:ID="wasBorn">
  <rdfs:domain rdf:resource="#Person"/>
  <rdfs:range rdfs:resource="&xsd;date">
</owl:DatatypeProperty>

<Person rdf:ID="Person1">
  <!-- Usage of predefined Annotation Property -->
  <rdfs:comment>the object representing a parent</rdfs:comment>
</Person>

<Person rdf:ID="Person2">
  <daughterOf rdf:resource="#Person1"/> <!-- ObjectProperty -->
  <wasBorn>2000-4-12</wasBorn> <!-- DatatypeProperty -->
</Person>
```

The properties can be predefined OWL or RDF-S properties or user defined. The properties itself can have characteristics that are defined in OWL. E.g. `owl:FunctionalProperty` defines that only one value can be associated to the object by this property. The property `wasBorn` could be defined as `FunctionalProperty` because one person only has one day of birth. Another important element in the use of properties is the OWL class `owl:Restriction`. This class permits to restrict a property that only applies to a particular class. Therefore the restriction references the property via `owl:OnProperty` and the target class via the value restriction `allValuesFrom` or `someValuesFrom`. For example to restrict a property `hasChild` for objects of type `Person` that only "persons" are referenced is written as:

```
<owl:Class rdf:ID="Person">
     <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="#hasChild"/>
            <owl:allValuesFrom rdf:resource="#Person"/>
        </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>
```

In this way OWL permits to use the same properties locally restricted for one class. The class `Pet` could use the same property in the same way, and nevertheless it is clear that the children of Persons can only be other Persons. More information about OWL can be obtained from documents provided by the W3C pages like [W3C03] or in [Lac05].

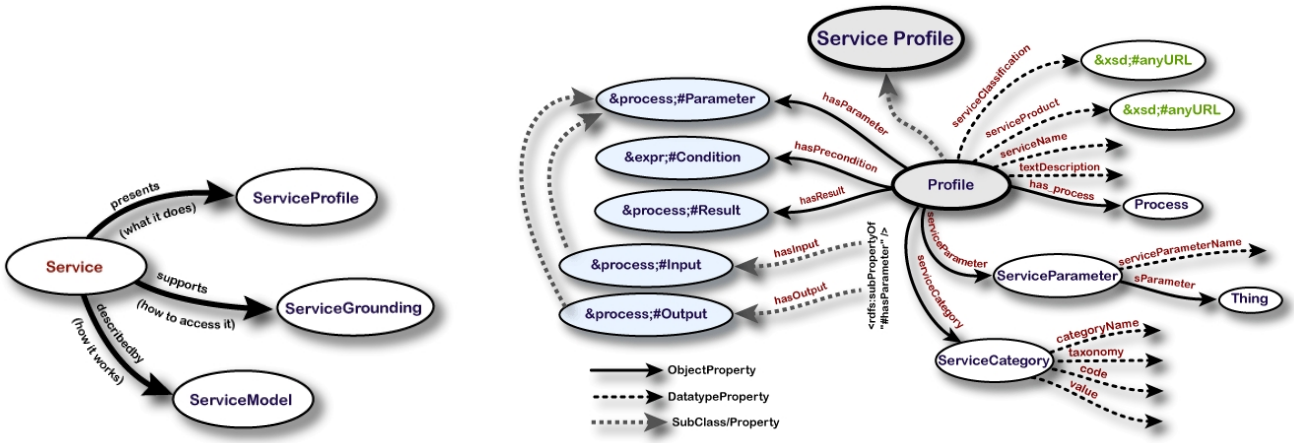
2.3.3 Semantic Service Description

A (web) service is a service accessible via a network, usually the Internet. The term is often related to business services, for example online shops, online bank transactions, dictionaries, etc, but it also refers to more basic services, like the

information from networked sensors. To use these web services, one has to discover them first (see Section 2.2.2) and then access them. If the client has the pre-shared knowledge of how to use the service, the access is no problem. Which data has to be sent when and how to retrieve the desired information or service? For instance a person wants to book a flight. Today we would look at several web sites to find a cheap offer and use the reservation form to complete the booking. However, one use case of the semantic web is to automate this process. The user orders his computer to book the cheapest flight to a certain destination at a given time. With a semantic Internet, the computer can now obtain offers on its own and make a booking. But for now the problem is that each company has its own form and reservation process. The web sites are written for humans and can hardly be interpreted by computers. So the computer does not know how to read flight plans or if the address must be sent before the billing details or the other way around.

OWL-S [OWL04] provides a solution to this problem. It uses an OWL Ontology to describe web services. The OWL-S specification of a service lists the input and output parameters including the datatype, conditions that must be fulfilled, service name, classification, textual description, and several more. A service can also consist of several linked services or processes. Clients using this service can parse the OWL-S specification of the service and thereby know the requested datatypes and sequence for the procedure.

An OWL-S specification consists of four OWL documents: *Service, Service Profile, Process Model and Grounding*. The *Service* is only a container linking the other three documents, see Figure 2.3(a).



(a) OWL-S Service

(b) OWL-S Profile

Figure 2.2: OWL-S Service - source `http://www.daml.org`

The *Service Profile* is a short description of a service. It tells what the service does and what is required to use it. A service seeking machine can verify if the found service matches the requirements, by the interpretation of the Service Model. Figure 2.3(b) shows a graph of the Service Profile. The most important fields are the input and output parameters. The result describes internal state changes of the service and is different to the output.

Process Models give a detailed description of the service. Usually a service consists of several available processes. For instance in an automated flight reservation service one has to check for flights, get available seats and then make the reservation.

While the Service Profile does not show which processes belong to a service, the Process Model does and furthermore it can give the sequence in which the process must be used. The so called "Atomic" Processes are the most important ones as they are the only direct invocable processes.

Finally the *Grounding* links the described service to a specific protocol. The Service Profile and Process Model were only an abstract description of the service. The two documents do not provide information on how to reach a service (for example which DNS name or TCP port) or what protocol is used. The Grounding is bridging the abstract description and the concrete implementation in a protocol.

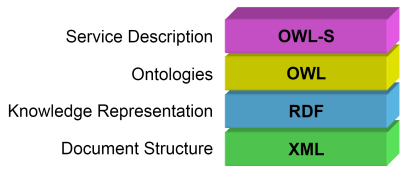


Figure 2.3: XML, RDF, OWL and OWL-S correlation

Until now the only Grounding for OWL-S is implemented in WSDL. WSDL (Web Services Description Language) is a description language for web services. A WSDL service description has different services methods (identified by names and parameters), similar to the OWL-S processes. The WSDL Grounding is linking OWL-S Atomic Processes to WSDL service methods, so that the abstract description can be mapped to a WSDL implementation. However it is based on Internet features which may not be available in sensor networks. Figure 2.3 shows the stack-wise correlation of XML, RDF, OWL and OWL-S. One can see that all these standards build on top of each other. XML, RDF and OWL define a language for ontologies, where OWL-S uses this language to define service interfaces.

3 Design and Implementation

This chapter introduces and explains the major design issues, which were considiered during the implementation of the framework.

3.1 Service Oriented Architecture / OSGI

A service oriented system architectur provides multiple advantages. Since software has become more and more complex and not all the functionality is always required for a product, the demand for non-monolithic software arises. Since integration of existing code absorbs a lot of time from developers, integration standards could help reduce costs when existing components are reused.

The OSGi Alliance, formerly known as the 'Open Services Gateway initiative', formulates the specification of the OSGi technology. It is also called 'the dynamic module system for Java'. Java is the language of choice, because it grants the portability to many different platforms. The OSGi service platform provides the functionality to ease software integration. It allows exchanging components during runtime. The service-oriented architecture (SOA) allows dynamically using other components seamlessly. By using an OSGI based architecture, services and applications can be integrated and maintained via the network.

OSGi is a specification in the first place. Different vendors/distributions like Knopflerfish [Mak07], Oscar [Con04] or Concierge [Rel07a] implement the specification. For the project, the Concierge distribution was chosen, since it has a small file and memory footprint. It is designed for mobile and embedded systems with limited resources in mind and is therefore ideal for the employed Gumstix sensor platform.

3.2 Implementation of the Service Communication

As introduced in Section 7, the individual services have to communicate to each other. Services are connected via a Connector. Any data that is sent to or requested from another service passes through the Connector, because the Connector is the only way to call methods of other services. The Connector chooses the right way to forward each request, preventing the services from having to worry about where another service is running.

3.2.1 Connector Interface Methods

There are different Connectors in the Context Framework, which all share the common ConnectorInterface. The following list introduces the most important methods of the ConnectorInterface:[1]

- **register** This method registers a new service at the Connector. It should be called by every new service to inform the Connector about its existence. For example, a sensor uses this method to tell the Connector that it is now operational. Only then the Service Directory can discover the sensor and use it.
- **unregister** This method can be used to tell the Connector that a certain service is no longer available for operation.
- **registerListener** To be able to know about the existence of other services, each service has to be informed about their existence. To save overhead, services that want to be informed must register themselves as a listener. The method allows specifying a filter, so the service is only informed about other services matching this filter.
- **getFoundServices** Once a service has registered itself as a listener, it can ask the Connector with this method to return all the services matching the filter. However, the returned result is only a list of identifiers, not of the actual services.
- **getRemoteObject** To get a remote reference to the actual service, this method has to be used. It takes an identifier and returns a local reference to the remote object.
- **invokeRemoteService** If a service finally wants to invoke a method of another service, it has to use this method. It takes a local reference, a method name and the parameters for this method and returns the method's result.

[1] Parameters for the methods have been omitted here, because they are not necessary to understand what these methods do.

3.2.2 Connector Types

R-OSGi

R-OSGI [Rel07b] is a bundle, which provides the possibility to remotely access services from distributed platforms. Service URLs are automatically generated on the basis of package name, interface name, IP address, TCP/IP port and the number of the service in the registry of the framework. The URLs are always prefixed with the type 'service:osgi'. R-OSGI messages for remote invocations does not rely on heavyweight frameworks like Jini and RMI. Besides other features there is the possibility to set properties in the announcement messages.

On the service provider side, all that needs to be done is to register the service with the `RemoteOSGiService` using the `RemoteOSGiService.R_OSGi_REGISTRATION` property. On the client side, a `DiscoveryListener` can be registered, which notifies the application on discovery of a service. The listener can be customized via LDAP-style filters [How96], which allow specific selection. After discovery the service can be fetched. If the proxy policy was chosen upon registration, this results in building a proxy bundle on the fly. The remote service can then be accessed transparently, since service method calls are send to the service provider seamlessly.

XML-RPC

Another Connector is called "XML-RPC". As the name does already suggest, it is a Connector that uses XML for remote procedure calls. In contrast to the R-OSGI Connector, this Connector does not know about an object's location and treats all references as a remote reference. The XML-RPC Connector itself can only serialize Java primitive types and needs a library called XStream for serialization of complex objects. Since the R-OSGI Connector does only run with the R-OSGI framework, the XML-RPC Connector must be used if one wants to connect to a non-Java application. For example, this could be an application running on Windows Mobile, programmed in C# using the .NET Framework. Of course, any other language can be used too, as long as it uses the protocol specified by the XML-RPC Connector.

MundoCore

MundoCore is a communication middleware, which was specifically designed for the requirements in mobile and ubiquitous systems [Ait07]. The main goal was to develop a middleware that supports various mobile devices like notebooks, PDAs and smartphones.

These platforms have different resources and communication interfaces available and use different operating systems. Services running on the devices shall be discovered regardless of the connection type. They also shall be able to handle remote communication.

Therefore MundoCore is based on a modular microkernel design and supports dynamic reconfiguration. It is available in the programming languages Java, C++ and Python. The Java edition is meant to work with Java 2 Micro Edition and Java 2 Standard Edition since version 1.1.

Services are the most important part of MundoCore. They communicate via channels. A service that wants to send a message publishes it on a channel. All services that are subscribed to that channel receive the message. The kernel handles the transport of local messages.

To extend the functionality of a node, the plug-in concept of MundoCore permits to load new services during runtime. For that purpose a plug-in directory is watched for new jar archives.

Miscellaneous Connectors

Also other Connectors are implemented for the Context Framework:

- **RMIConnector:** A Connector using the Java's RMI API for remote method invocation.
- **MultiConnector:** A Connector for combining multiple Connectors into a single Connector.
- **LocalConnector:** Fast and simple Connector for local-only communication.
- **PlaybackConnector:** Stores messages in a database. Can also playback (simulate) recent events into a running system.
- **OfflineConnector:** Stores messages during offline phases. Transmits selected messages after reconnection.

All Connectors can be cascaded or combined using the MultiConnector.

3.3 Implementation of Service Discovery

Mundocore provides its own Service Discovery Method which is usually based on broadcast messages. For other approaches like R-OSGI or XML-RPC, a Discovery Method has to be applied. Both Connectors are able to use a broker based service discovery as basic method. For a broker based method, a Connector has to determine a common known network address for the broker, whereas each system running a Connector may be used as broker. If no broker addresses are known, a SLP based discovery method is used.

3.3.1 jSLP

jSLP is a Java implementation of the Service Location Protocol (SLP), for service discovery. SLP provides a scalable framework for the discovery and selection of network services. Using this protocol, computers using the Internet need little or no static configuration of network services for network based applications. [GPVD99] The bundle provides two services to the framework `ch.ethz.iks.slp.Locator` and `ch.ethz.iks.slp.Advertiser`. `Advertiser` can be used to announce services via SLP and `Locator` is able to find such services again.

In some network environments multicast might cause forwarding problems at gateways and bridges. To circumvent that, the specification of SLP in RFC 2608 allows using broadcasts in place of multicast. It is generally allowed to use broadcast, even if multicast is not supported.

3.4 Common Sensor Implementation

A sensor has to provide/implement the following Java classes/interfaces:

- **ServiceImpl** This class contains the actual implementation of the service.
- **ServiceActivator** This abstract class manages service startup, configuration and registration
- **ServiceInterfaceImpl** This class holds common sensor methods like getIndividual and others for retrieving string representations of the owl-s files (details in ServiceInterfaceImpl). This interface has a central role within the sensor lookup, as it provides the sensor description. Details about this interface are shown in Section 3.4

  This class contains common service methods all services need to be able to integrate in the Service Directory. Since the Service Directory is meant to deal with sensors without closer knowledge of their capabilities, an Ontology description is required. This description contains information about the methods the sensor provides for remote execution, the name of the method and the input (if there are any) and output types. The types are specified in the sense of Java types and in the sense of the network Ontology's attributes.

  The description is assembled by five functions, which return a special string. The language of this string makes use of the Resource Description Framework (RDF), Web Ontology Language (OWL) and OWL-S for semantic service description as explained in Section 2.3.

  - **getService()** provides the service document, which is the super document, that fits the other three together.
  - **getProfile()** contains a rough description of the input and output parameters of the service with their respective type. The following code fragment shows how this description can look like. Given is an extract of the PhoneBookService:

    ```
    <profile:hasInput>
       <process:Input rdf:ID="PhonebookService_getPhoneNumberById_in0_IN">
       <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
                &komssn;#userValue
       </process:parameterType>
       </process:Input>
    </profile:hasInput>
    ```

  - **getProcess()** contains the processes with their attribute and related input and output parameters. The most important process attribute in this context is the 'CurrentAttribute'. It marks methods, which can be used by the Service Directory for information gathering. Every method of one sensor is defined as `AtomicProcess` since this is the only process type that is invocable. Again the input and output parameter are defined, but also more details like the `processAttribute` and the `expirationTime` are declared in this file. The latter is needed by the cache and defines how long the output information can be considered as valid.

    ```
    <process:AtomicProcess rdf:ID="OutlookSensor_hasAppointmentNow">
       <process:hasOutput rdf:resource="#OutlookSensor_hasAppointmentNowReturn_OUT"/>
       <komssnowls:processAttribute rdf:datatype="&xsd;#anyURI">
    ```

							Remarks
BT-Sensor	L	X		X			Legend:
PC Task	W						X=Ready
ComputerActivity	X		X	X	(X)		L=Linux,
Light				X		X	W=Windows
Battery				X		X	(X) = In work
GPS			X	X	(X)		(...) = Open
Database/Mapping	X	X	X	X	(X)		
Addressbook	W		X		(X)		
Calendar	W		X				
Skype	W						
Temperature				X		X	
Wlan-AP	L	X	X	X			
Orientation					(X)	X	
Location / Neighboorhood						X	
Chair / Heartbeat						X	
Webcam	X	X	(...)	(...)			
Gateway / Zigbee	X	X		(X)			

Figure 3.1: Overview - Implemented Sensors

```
      &komssn;#CurrentAttribute
   </komssnowls:processAttribute>
   <komssnowls:expirationTime rdf:datatype="&xsd;#long">
      60000
   </komssnowls:expirationTime>
</process:AtomicProcess>
```

- **getGrounding()** contains the information, that put the two previous parts together and specifies the Java method names, that are used to use the previous described processes and their respective input and output parameters.
- **getIndividual()** contains the individual attributes of the sensor like e.g. locationValue or userValue.
- **getSensorURI()** contains the URI of the sensor type.

3.4.1 Implemented Sensors

Several sensors where built for usage within Context Framework and for testing its functionalities. Figure 3.1 shows an overview of all sensors. The column shows the platform for which a sensor is implemented:

1. Sensors for PCs
2. Embedded Linux platforms like Gumstix or network routers.
3. Windows mobile based phones
4. Nokia's N810 Linux based PDA
5. Apple's IPhone
6. Motes (SunSpot / tmote sky)

Figure 3.1 shows:

1. **Bluetooth Sensor (BT-Sensor)** The Bluetooth sensor was developed to detect Bluetooth devices in the neighborhood. It periodically scans for Bluetooth devices in range. It can be used to determine the location or active devices in the environment of a user. For example: Whenever the a mobile phone of a user can "see" the video projector, this is a very good indicator that the user is in the presentation room.

2. **PC Task Sensor** This sensor provides several informations about what a user is currently doing on his computer. It gives information about the program in the foreground, the mouse movement and the rate of keys pressed within a certain time frame.

3. **Computer Activity** This is a rather simple sensor, which measures when the user last used the device. The sensor does neither analyze what was used (e.g. turning off the media player vs. writing an email) nor does it analyze the intensity. These are to be implemented in the PC Task Sensor. Furthermore, the sensor monitors mouse activity to know when the user was active. This is done by using the AWT library of Java.

4. **Light** An uncommon sensor for a mobile device is the Nokias 810's light sensor. It measures ambient light and provides a value in lux as a result. Even though the information is quite simple it can help to detect if the user has the N810 in his pocket. The light sensor is also sensitive enough to distinguish even little changes in luminance. For example, the sensor will measure different values in a dimly lit presentation room, a bright-lit conference room and in a park flooded with sunshine.

5. **Battery** The status of the battery charge can be used as a sensor. This is especially useful because a device reports if the battery is being charged up. In such a case, it usually can be assumed that at least the device is not moving around.

6. **GPS** Many devices come with a Global Positioning System (GPS) [2] receiver. Originally intended to be used with the mapping application, the GPS data can also be read by other applications. As GPS was originally designed to be used by naval vessels, which usually have only limited resources available, the data format already makes sure that the data is very compact. Since the sensor only delivers the longitude and latitude values, this data is already minimal for the information it provides. This information is very good to determine where the user actually is. However, GPS does not work in buildings and the sensor will most likely fail to provide any information there. Another problem is that received GPS positions are not completely accurate. There are multiple sources of error like propagation delay, multipath fading or selective availability [BRA02]. Thus, GPS positioning can only give an estimate where someone is, but is for example not suited to determine the exact room someone is in.

7. **Database Mapping** This sensor enables simple mapping between sensor data. It acts like a database with sensor data as input and output. Using the mapping sensor it is possible to map for example from userId to workplace or from bluetooth bssid to device type.

8. **Addressbook / Calendar** These sensors wraps Microsoft's API for Outlook. They provide information about current and future events or provides information from a user's address book like a mapping from userId to phoneId.

9. **Skype** The idea of this sensor is to read the user's Skype status. The Skype sensor wraps some of the Skype API methods in order to make them usable within the Context Framework.

10. **Temperature** Some devices have a temperature sensor (often for its internal temperature). Such sensors do only measure the internal temperature, which means that outer temperature changes are not immediately detected. The sensor is also dependent on the CPU load, as a constantly working CPU will heat up the device. However, the sensor can still be used for example to distinguish between being carried around in the pocket and laying on the desk of an air-conditioned office. This is possible because the sensor is sensitive to very small temperature changes.

11. **Wlan-Access Point Sensor (Wlan-AP)** In order to detect available access points, a WLAN sensor was developed. The latest revision returns the ESSID, the BSSID and the signal level of the access point found to be the one with the best signal level.

12. **Orientation** Some newer devices like the IPhone have a built-in acceleration sensor. This sensor can be used to determine if a device is currently carried around or if it is laying on a table.

[2] http://www.gps.gov

13. **Location / Neighborhood** By scanning the environment of a wireless network interface, the found identifier of the network interface and the signal strength can be used to determine the location or devices in the environment. By mapping from identifier to user the encounter time between to persons can be measured, which is then used to determine social relationships.

14. **Chair / Heartbeat** Some special sensors were built by using the serial interface on a mote. For example a chair sensor determines if a person is sitting on it (using a pressure sensor) or a heartbeat sensor which senses the heartbeat of a person by connecting a receiver for a heartbeat sensor (available as sports equipment) to a mote.

15. **Webcam** A webcam could be used for analyzing if the user is currently in front of the device, or even for motion tracking.

16. **Gateway** A gateway is needed in order to connect a mote to an Ethernet based network. The gateway enables transparent access to sensors on a mote. Similar (but more lightweight) service interface descriptions are used on the mote side.

3.5 Semantic Sensor Lookup

A simple sensor lookup algorithm may iterate over all sensors and their methods no matter what kind of information they will provide. With an increasing number of sensors this results in a huge amount of requests and information that is not related the original search request, depending on the application performing this search. Imagine an application requires the information about the user's computer status (e.g. idle, active or offline). A search for indications about the user's presence state should for example result in the user's computer status, its location, all other users in the same room, their computer status, etc.. This does not only need a lot of effort to filter the desired information afterwards, it also increases the search execution time.

The challenge was to create a search algorithm that is adaptive to different applications and to a dynamically changing sensor network topology without reprogramming it manually. The idea is to use the semantics stored in our Ontology description. This Ontology named KOMSSN contains three top level classes: *Data*, *Entity* and *Search*. *Data* organizes all data types used in our network in a hierarchy, *Entity* covers an organization of sensors or sensor and the subclasses of *Search* will be used to define a search according to the requirements of an application.

3.5.1 Semantic Annotations for Data Types

The first decision was to make the search algorithm dependent from the output data types of the service methods rather than from the sensor type. One sensor could provide service methods of different types that are not all relevant to a certain application. The ComputerActivity sensor mentioned above provides one service method that has the output type `isActive` which can be true or false. Next to this method there is static information stored in the *Individuals* of this sensor. An Individual is the description of the relation of a sensor to other entities e.g. the Individuals provides `userIdValue`="myName", `roomValue`="Room123" and `roomFunctionValue`="Conference Room". These data value pairs are called setup properties and are used to select this sensor if one matches the search objects in the result tree. Additionally they provide information that can be used. Here it is possible see that room 123 is a conference room. But this information is not desired if the task is to obtain the computer status. Thus defining which output data types will be searched for, results in retrieving the required information only. The setup properties defined in the Individuals are not output data types in the original context of method and input or output parameter, but can be considered as static output of a sensor and therefore will be treated in the same way.

3.5.2 Usage of an Ontology for Hierarchical Data Type Structure

The use of the Ontology enables to select groups of data types that belong to a certain category. E.g. *ComputerActivity* is a subclass of *Activity*. Under this superclass the Ontology describes a further class *PhoneActivity* and *ComputerActivity* is again subdivided into a last and current activity. Figure 3.2 shows this class hierarchy. Thus, by searching for *Activity*, we address several data types of the same category and do not need to know which sensor is available and has which output data type. Additionally when future sensors extend our network, e.g. a Skype sensor, its output data type (e.g. SkypeActivity) will be added as further subclass, and the search algorithm will consider this sensor as well. So it adapts to the dynamically changing Ontology without any manual effort.

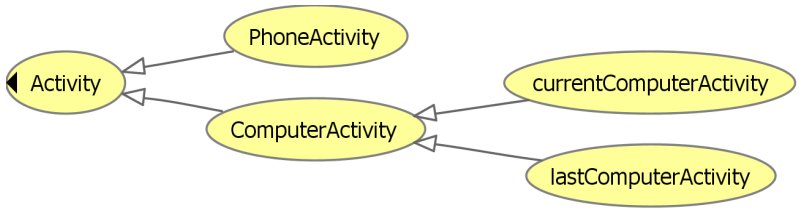


Figure 3.2: Activity Class Hierarchy

3.5.3 Defining an Application Specific Search

The section above showed that the search algorithm is focused on output data types when performing a search. But somehow we have to define which data types or data type classes we search for, and in case we want to iterate over some values we have to define these, too. Since all data types are already defined in the Ontology description we make use of this description and link data types to "Search classes". Therefore we introduce the top level class *Search* and two object properties: *iteratesOver* and *searchesFor*. To define a concrete search, we create a subclass of *Search*. The data type classes that will be considered during a search run are linked via the two object properties. The *searchesFor* property points to data types that will only be considered as result, while the *iteratesOver* property points to data types that are used as intermediate results that will be used as input parameter again. These objects properties can be used multiple times to define as many data types as the application demands. For example to define a search that will be used to determine the presence status for all users in one room, the search process should iterate over `roomValue` and `userIdValue`. So these data types are linked via the *iteratesOver* property. For a presence status we might consider data types of group *Activity* but also appointment entries of the user's Outlook Calendar. These data types are therefore linked via the *searchesFor* property. Figure 3.3 shows how this example will look like. Note that if we point to one class, all subclasses will be

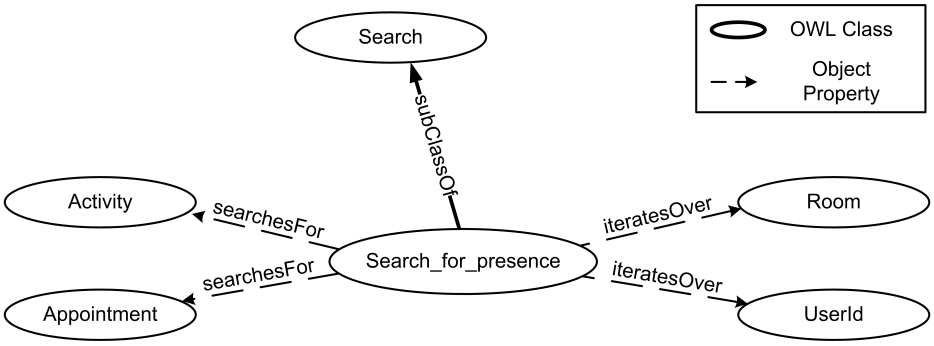


Figure 3.3: Search Definition in the Ontology

considered as well. If we do not want to iterate over some data types we omit the *iteratesOver* property and get results to the initial search entry only. Another advantage of having the search defined in the Ontology is that we do not need to implement anything, if a further application requires a new specific search. The designer can use an editor like Protégé [3] or any other OWL Editor to create a new search definition within a few "clicks".

3.5.4 Ontology-based Search Algorithm

This new search algorithm with an object of `Search` and the containing initial search object. Additionally the name of a search subclass must be declared by the attribute `ontBasedSearchName`. During the initialization phase of the search process, the designated search class in the Ontology is read. The data types that were linked by the two object properties are written to a list each. Then these data types are checked for possible subclasses and in case of existence also written to these lists. If any of these classes has a data type property, this will be added too. Finally we have two groups with data type representatives, that will be compared with the output parameters of available methods.

Now the actual search process starts. It is divided in an iterative run, that will be repeated each time a new result adds itself to the result tree, and a final search run that will be processed only once. The search algorithm separates this because in the iterative run only methods with output data types of the *iteratesOver* group and in the final run

[3] Online available at: http://protege.stanford.edu

of *searchesFor* will be considered. During a search run we differentiate between *sensors* and *non-sensors*. A *sensor* has service methods that have no input parameter. Therefore the connection between this sensor and its context is drawn in its *Individuals*. This context can e.g. be a user it belongs to, or a location. So before a sensor's method will be invoked, it is checked if there exists a connection to an object of the ongoing search, which requires that a setup parameter of the *Individuals* matches one search object in the result tree. Additionally a sensor's method will be invoked only once during a complete search process, because it will provide the same information independent from the matching setup property (in case the *Individuals* contain more than one). On the other hand a sensor may also use input parameter. It will deliver its information according to the input value. Therefore it does not necessarily need setup properties in the Individuals. To invoke a method of this type, it is mandatory to have a search object of the same data type as the input parameter of the method. Methods of this type will be invoked multiple times for each individual search object of that data type. One particular search run includes the following steps:

1. A loop over all sensor types that are registered at the SD.

2. A further inner loop over the service methods of the particular sensor type.

3. Invocation of this method for all sensor instances if
   (a) the output data type matches and
   (b) the setup property or the input data type matches and
   (c) this method was not invoked in the same context before (see difference between *sensors* and *non-sensors*).

4. Adding all setup properties of relevant data type contained in Individuals of matching sensors.

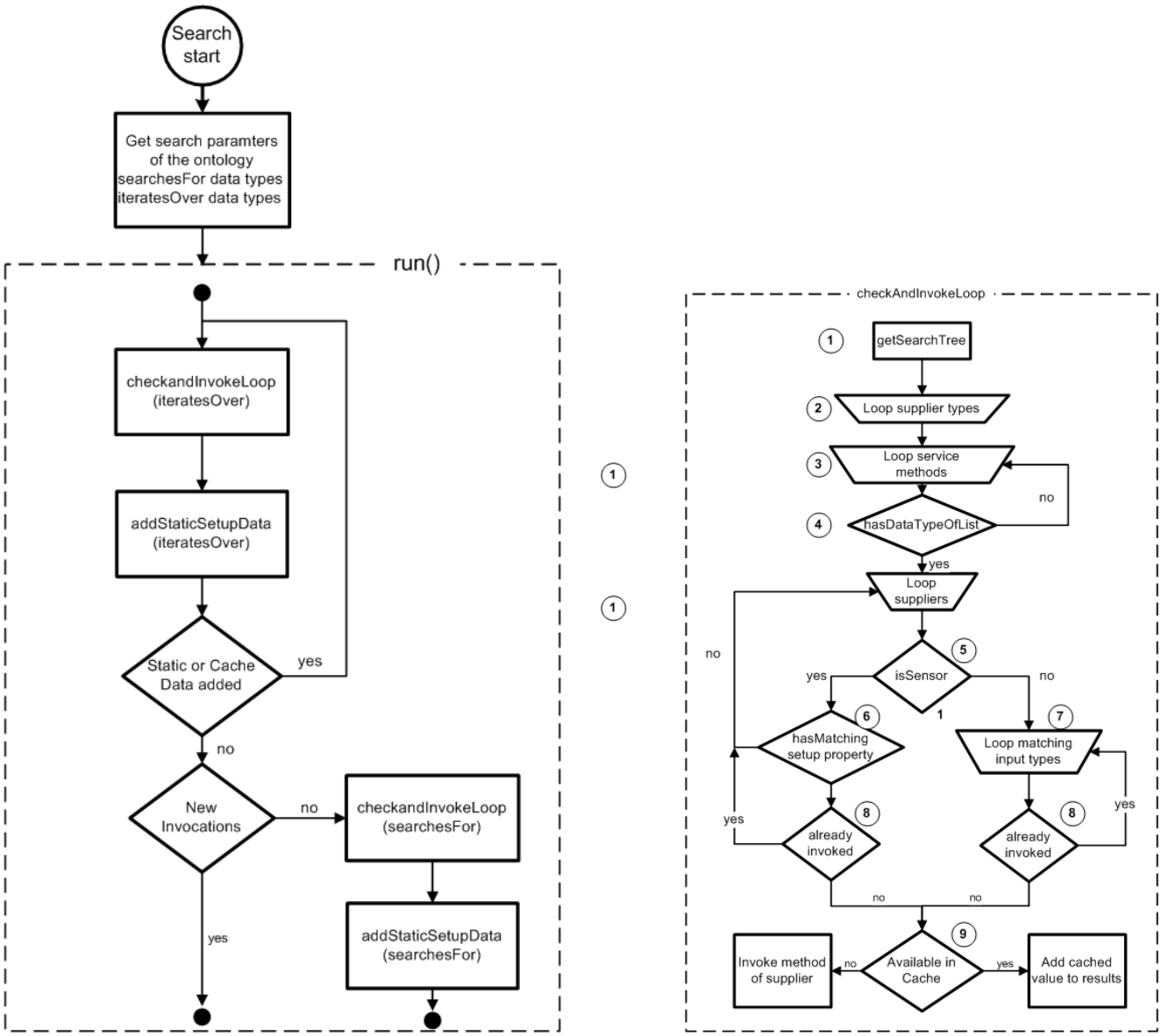
This run will be repeated each time a new answer is added to the result tree, because this answer could be used as input value again. When all invoked methods have returned their results and the search tree is not growing anymore, the iteration is finished. A last search run will performed now over the so far obtained results, but this time the setup properties and service methods according to the *searchFor* data types are considered. When no further methods are invoked and the results of all invoked methods have arrived, the search is finished and will be returned to the calling application. The whole search process is covered by a timeout restriction. If the search has not finished until this timeout because one sensor is left to answer, the search is closed nevertheless and returned to the application with ther esults obtained so far.

After the Ontology is read and both groups of data types are set, the actual search algorithm starts with the first call of `runThread.start()`. `runThread` is an instance of the nested class `RunClass` that extends `Thread` and only overrides the `run()` method. See Figure 3.5(a) that represents this `run()` method.

The setup of a search query is explained in Section . The main arguments of the search query are two lists of data types. The first list *iteratesOver* determines the data types which should be used in further iteration steps and the other list names *searchesFor* determines the data types which sould only be used as result. It is used to call the method `checkAndInvokeLoop` and the method `addStaticSensorData`. It starts with one run over all sensors in `checkAndInvokeLoop` and then adds static setup properties in `addStaticSensorData`. Both methods get as parameter the HashSet containing the selected data types to iterate over. This iteration run will be repeated if new results add to the search tree. This can occur in three cases: (1) if static setup properties are added during `addStaticSensorData`; (2) or during `checkAndInvokeLoop`, when instead of invoking a method the cached value is added to the results. Then the flag `cacheObjectUsed` is set to `true` and the loop will be repeated. (3) When the loop is finished and invoked methods return their results to the SD, this run method is called again. This is done in the method `incomingSensorResults`, which is called by each invoked remote method.

- `addStaticSensorData(HashSet<String> dataTypeList)` This method works on the setup properties in the Individuals of sensors. It loops over these properties and checks if anyone matches with objects in the result tree. If a match is found, the other setup properties are added to the results in case the type of the property is listed in the `dataTypeList` and not already contained in the search results. If at least one object is added to the search results, this method returns `true`, otherwise `false`.

- `checkAndInvokeLoop(HashSet<String> dataTypeList)` This is the main loop of the search algorithm that will be performed multiple times during a complete search process. Figure 3.5(b) shows a graphical representation of the `checkAndInvokeLoop`, which is included again in Figure 3.5(a). In the first step (1) it requests a flat representation of the results in the search tree in form of a Hashtable with the data type of a search object as key and a vector of corresponding search objects as object entry. This Hashtable will be used during the search run to check if any of its entries can be used as input parameter of a method, or matches a sensor's setup parameter. In the next step the method performs a loop over all sensor types (2) and over their service methods (3). In the loop, the attribute `owlsOutputParameter` of the service method is compared with the data types in `dataTypeList` (4). If no match is found, this service method is skipped. Otherwise, the next step depends on the method type (5). If the method



(a) run method (b) check and invoke koop

Figure 3.4: Search Algorithm

has no input parameter (a sensor), the Individuals of this sensor are checked for matching setup properties (6). Otherwise, the process will loop over all objects in the result list (7) that fits to the input parameter. E.g. if the current service method will provide phone numbers to user IDs, it will be called with every user ID entry in the result list. Before the service method is finally invoked, the process checks if it has not been invoked before (8) and if the result is not stored in the cache (9). Steps 5 through 9 will be repeated for all sensor instances of that type.

- `incomingSensorResults` This method handles all results that arrive from a sensor. When a sensor's service method is invoked, a parallel thread is created. This is necessary to prevent the search algorithm to stop and wait for each sensor and increases the search performance. In the `run()` methods of these threads, the result of the service method and an invocation ID is passed to the search engine by calling `incomingSensorResults` with `(Object ret, String invokeID)` as corresponding parameters. For this reason, we have multiple instances of `incomingSensorResults` running in parallel. As mentioned above, this method triggers a new `run()` to process the newly arrived intermediate results. And multiple instances of `run()` would invoke the same service methods multiple times, which would result in duplicated sensor results. To prevent this and synchronize the iteration loop, the `incomingSensorResults` method performs a `join()` on the run thread before calling a new `run()` and thereby waits for one particular iteration loop to be finished. To avoid that, two instances are waiting for the thread to

end, and then simultaneously creating a new one, the sequence is made atomic by a lock. The reason to put the `run()` method in a nested class instead of making the surrounding class runnable, is that Java does not permit to start a `run()` method twice. Therefore, the process creates a new instance of `RunClass` each time a new iteration is required. Figure 3.5 shows the interaction between sensor, search and run methods.

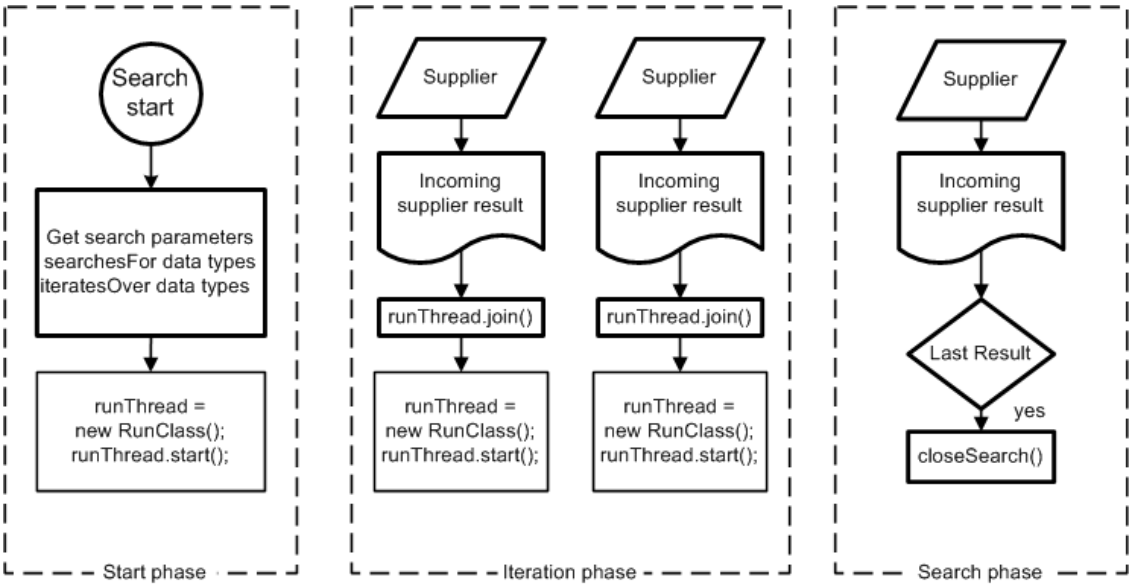


Figure 3.5: Search Phases

Next to triggering new iterations, the second task of this method is to create search objects of the returning data and add them to the result tree.

3.6 Dynamic Ontology Extension

Using a static Ontology, a designer of a new sensor has to add at least[4] the sensor's URI to the Ontology. With the new Ontology-based Search, it becomes also absolutely necessary that the input and output data types of the sensor's method are defined in the Ontology, and even more in the correct position in the data type hierarchy based on their semantic context. In this scenario, an external designer who has no access to the main Ontology has no chance to connect a new well-designed sensor to the KOMSSN network. Additionally the question was why the sensor has to be described once in the OWL-S files and a second time in the Ontology. The idea to overcome this obstacles, was to use the information stored in the OWL-S files for extending the main Ontology dynamically at registration of the new unknown sensor.

The third-party designer knows the current KOMSSN Ontology. In his OWL-S files, he describes his sensor with its service method and its input and output parameters. Additional to these parameters, he defines a reference to the existing KOMSSN data types. He will choose a KOMSSN data type where his new type belongs to, according to the semantic context. E.g., the new sensor could be a software sensor that will check the PC for a Skype client and will provide this client's status (offline, online, away, not available). The designer might decide that this Skype sensor might fit best in the context of the KOMSSN data type *Activity* or even more specialized as subclass of *PhoneActivity*. In this case he will define this semantic reference in his OWL-S description. In case the new sensor is totally different to all data types in the existing Ontology, he will just give it the top level data type *Data*.

The designer has to define these additional references only if he wants to use new data types. If the data type already exists in the Ontology, then he references the existing data type. E.g., a new type of ComputerActivity sensor that does not give the computer status according to active mouse movement, but depends on keyboard usage or CPU activity. In this case no new data type has to be added to the Ontology.

At registration of this new sensor, the SD checks its current Ontology for the URI of this sensor. But instead of rejecting this sensor because the URI is unknown, this time it requests the OWL-S files. Either the URI is an URL and the OWL-S files can be downloaded from the web, or the sensor has to deliver them locally. When the OWL-S files are obtained, a process is started that tries to retrieve all relevant information. This process succeeds if all parameters are known,

[4] If the data types are not contained in the Ontology, they are considered as *Object* data.

or unknown parameter come along with known references. Then, in case of success, this information is written to the Ontology, and the sensor will be registered. The next sensor of that new type will be identified by the URI as usual and the OWL-S parsing process can be skipped.

addUnknownSensor

This method uses the Jena OWL API [Hew07] and additionally the OWL-S API from mindswap [Sir04]. The latter one is used to parse the OWL-S files because it features (in contrast to Jena) dedicated OWL-S support that eases parsing standard OWL-S elements. The Jena API is then used to write the extensions to the Ontology because the Service Directory uses a Jena Ontology model of the KOMSSN Ontology. Therefore, this Ontology model is extended in first place and later written back to a file.

Using the Skype sensor as example this results in exactly one output. The next step depends on the data type of the output (`Output:parameterType`), in this case it is `SkypeActivity`. If a KOMSSN data type is given in this field, the value starts with `&kommssn;`. Then the process rechecks if this data type is contained in the KOMSSN Ontology. If so, nothing has to be done, if not the process aborts. If the data type is a new one, like in the example, the process tries to get the reference to the KOMSSN data types:

```
String propUri=SNConstants.KOMSSNOWLS+"#KOMSSN_Datatype";
OWLDataProperty KOM_Data_prop=myOWLSOnt.getDataProperty(URI.create(propUri));
OWLDataValue value=outputObj.getProperty(KOM_Data_prop);
```

The above code fragment retrieves the KOMSSN data type. If it is defined (`value!=null`), then the process will define the new data type as datatype property of the data class designated by `value`. In the example this is *PhoneActivity*.

```
if (value!=null) {
   String newPropURI=SNConstants.KOMSSN_URL+"#"+type.getURI().getFragment();
   // name of new data type property
   String KOM_Datatype_URI=value.getLexicalValue();
   // name of the referenced data type class
   OntClass superClass=model.getOntClass(KOM_Datatype_URI);
   // OntClass representation of daty type class
   DatatypeProperty newDataProp=model.createDatatypeProperty(newPropURI);
   // creates the new data type property
   newDataProp.addRange(XSD.xstring);
   newDataProp.addDomain(superClass);
   // adds the new property to the super class
   model.createCardinalityRestriction(null,newDataProp,1).setSubClass(superClass);
   // creates a new Restriction with this property for the given class
}
```

Finally, when all fields are processed and the Ontology has been extended for a new sensor class and new data type properties, the extended Ontology model is written back to an OWL file. This file can be used to see the results of this process graphically in Protégé and, if necessary, to maintain the Ontology and reorder data type classes.

Ontology maintenance

Nevertheless we assume that the sensor's designer knows best about the semantic context of his new service, the extensions to the Ontology are made locally on the SD where the new sensor registered. The extended Ontology will be written back to an OWL file, which allows maintaining the extended Ontology by a KOMSSN administrator. The reason for this restrictive procedure is that a lot of new sensor types and new data types can result in an unordered hierarchy because intermediate classes are missing. To prevent this, the administrator can take corrective action, insert intermediate classes and reorder the Ontology. This updated Ontology can then be published in the web and thereby also to other SDs. New sensor development can then refer to this updated Ontology. The existing ones remain working because they are known to the Ontology and will not provoke a re-extension of the Ontology.

3.7 Cache

To increase the performance and reduce the communication, caching and prefetching mechanisms are needed. The challenge here was to have one cache for all the different types of sensors, with a different changing rate of their sensor value. Especially for battery powered sensors, the reduction of communication is important because each message between sensor and server reduces the sensor's lifetime.

The Jena Framework [Hew07] is used to extract the expiry value out of the OWL-S files. When the Service Directory needs to invoke a sensor's method, it first checks the cache for an entry. If a matching, non-expired entry is available, the Service Directory uses the stored result object. If not the Service Directory will invoke the method of the sensor. When a sensor answer arrives at the Service Directory, the result will be written to the cache and is available there for the next invocation.

3.8 Summary

The dynamic extension of the main Ontology is our approach to merge a sensor's description with our Ontology. With this first approach we only use the resulting class hierarchy of the service method data types that will be used by the search algorithm. Despite this relatively simple approach of using a class hierarchy, the description by the OWL-S files has to follow a strict syntax, otherwise the used OWL-S parsers in combination with the processing algorithm might not be able to retrieve all necessary information. Which information the process requires to extend the Ontology, how this information is defined in the OWL-S files and how the process finally parse these files and extend the Ontology is described in the implementation part in Section 3.6.

4 Performance Analysis

The performance of the communication towards the sensors is crucial factor of the overall performance of the system. Due to the communication system scenario and the time constraints within a communication system, the sensor lookup has to be performed within a very limited amount of time. A slow communication towards the sensors and a slow sensor lookup may end up in search queries which have to been terminated although not all related sensors could be queried. The amount of aggregated information strongly affects the desired QoI (described in Section 1.2). For this purpose we tested our implementation using a specific request generator. This part presents the setup and the results of these performance tests.

4.1 Service Communication Performance

[And04] is about monitoring Linux in terms of CPU, memory, disk devices, disk controllers and network devices as metrics to measure. The paper differentiates between monitoring in real time and in snapshot values of the parameters. Our tool uses a mixture of both. To minimize the influence of the tool on the system, we only monitor values in real time, where necessary. Most of the presented tools are special purpose, like 'uptime' for load or 'free' for system's memory usage. We are more interested in a single tool that does not require an additional terminal to retrieve the values and can monitor the memory usage of one desired application. The sysstat project, which is introduced at the end, is more long term system monitoring. These tools provide a quite complete suite, but they require write operations and to differentiate the measurements afterwards one would have to separate them manually. Our development is small and perfectly customized to fit our needs with result presentation after execution.

[MM98] declares three stages for a performance tool: The instrumentation stage is for gathering the raw data. The analysis stage is responsible for e.g. the calculation of statistics. The presentation stage produces e.g. graphs and charts. Our request generator (Section 4.1.1) is the first stage. It retrieves the call time values and saves them in a file. Pidstatm is applied in the second stage, it gathers values but also does averaging. A python script is the third stage. It generates the required output (statistics) for gnuplot. Gnuplot and the respective helper scripts are the fourth stage in our setup. They generate graphs for presentation.

The paper also formulates various metrics to measure and therefore divide the system in three levels. We mainly operate on the first (system as complete unit) and the third (client application specific) level. Level one metrics we measure are total response time (delay) and middleware throughput (requests per second). But we also examine client system values like load average, CPU utilization, memory usage.

The effects of server threads mentioned in [LGL+98] will be also acknowledged in Section 4.1.1.

Since our evaluation is based on the Gumstix as embedded system and the service directory as core infrastructure for querying middleware-based sensors, none of the existing papers really matches our setup.

[Ait06, chapter 9] also evaluates the performance of MundoCore, but most tests are executed on powerful machines. The software running is a small program developed mainly for the evaluation and that way not comparable to our setup. The Java and C++ comparison is done on the same host, which is a special case for MundoCore, since the communication is shared memory-based. The mentioned tests on embedded systems use the C++ version of MundoCore, which is expected to perform better than the Java version.

[RAR07] evaluates the service binding and the service invocation time with the help of the JavaParty and WSTest benchmark for R-OSGI, RMI and UPnP.

4.1.1 Request Generator

For the evaluation of the middleware-based sensors on an embedded system, at first a request generator is required to cause load on the Gumstix with one sensor running.

The values retrieved by the generator shall help to determine:

- Maximum number of requests.
- Response time of a request (delay).

Requirements for the generator:

- Send requests at a specific rate.
- Flood the sensor with requests for a specified time.
- Ability to query any sensor without sensor-specific adaptations at the generator.
- Work without modifications on the sensor side.
- Be able to measure the delay of a request.

Since the sensors provide their Ontology description on registration at the Service Directory the generator is developed as additional part of the Service Directory. So the existing infrastructure can be used and every sensor registered at the Service Directory can be queried.

The request generator uses thread pools. The number of threads has influence on the measurement result. If the number of threads is too low, the tested system will not reach its limits. So a single thread is not sufficient. Too many threads put heavy load on the requesting system. Thread pools use a defined number of threads, which can be reused to restrict the load on the inquiring system.

Reasonable modes for the request generator:

- **single** One call after another (a single thread) - for determining the response time of a single request.
- **burst** Flood (multiple threads) the sensor - for determining the maximum number of requests possible.
- **rps** Send at a specific rate - for rate visualization (graph)

4.1.2 System Monitoring Tools

System monitoring tools are required to examine the load on the Gumstix caused by the requests. The following values are of interest for the evaluation:

- CPU utilization
- Load average
- Memory usage of the application
- Network traffic

The first three parameters need to be recorded by a tool running on the Gumstix. In our setup, pidstatm [1] performs this task. The network traffic can be captured on the requesting system. In order to keep away any unnecessary load from the embedded system traffic counting is done on the requesting system (see Figure 4.1 by procnet).

Procnet

Procnet is a network tool that displays the amount of traffic on a specific interface over a defined time. Just like pidstatm it uses the proc filesystem on a Linux host to gather the desired data. The program code is based on bwbar[2] (bandwidth monitor). It is a breakdown of the original code to the needed functionality with some modifications. On start-up the tool determines the time and reads `/proc/net/dev` to get the amount of bytes send and the amount of bytes received on the chosen interface. Then it sleeps for a certain time (by default 60 s). After that it gathers the same type of data again and calculates the difference. As result it is printing the monitored time, the amount of bytes in total on the interface (plus received and send separately) and the calculated average bandwidth used (again total plus received and send separately) based on the collected data.

Procnet output example of `./procnet -t 1800 eth0`:

```
time: 1800.00 s
TOTAL_BYTES: 12288814
TOTAL_BYTES_IN: 7207895
TOTAL_BYTES_OUT: 5080919
BWU: 6827.12 bytes/s
BWU_IN: 4004.39 bytes/s
BWU_OUT: 2822.73 bytes/s
```

[1] http://linuxwiki.de/proc/pid
[2] http://www.eu.kernel.org/pub/software/web/bwbar

Pidstatm

Pidstatm is the tool, which was developed for that purpose. It reads information about a process the Linux kernel provides through the proc filesystem [Ker05].

Especially `/proc/<pid>/statm` is of interest in this context. It holds information about a process's memory status. The tool accesses the size and the RSS field. Size stands for the total program size. RSS is the resident set size, which equals the occupied real/physical memory. Since both values are in pages, they need to be multiplied with the page size. The pagesize can be obtained from 'include/asm/param.h' and is 4096 bytes in most cases. In order to get the physical memory used by a process in KB, compute:

$$RSS[KB] = RSS * pagesize/1024 = RSS * 4 \tag{4.1}$$

In order to be able to compare that to the overall memory consumption of the system, the tool also parses `/proc/meminfo`. This file holds information about the memory status of the system.

`/proc/<pid>/stat` contains among other things the number of jiffies a process has been scheduled in user mode and the number of jiffies in system mode. In general jiffies [Mis07] represent the system 'uptime' in ticks, which equals the number of timer interrupts since boot. Jiffies increase at HZ (system dependent constant) rate. HZ usually is $100 * 1/s$ on x86-architecture. This is also valid for the Gumstix, because the kernel configuration variable CONFIG_HZ of the Gumstix is 100. That means, that the counter increases every 10 ms. The number of jiffies in combination with the elapsed time is used for calculating the CPU utilization by a process in percent:

$$CPU\% = \frac{(user\_jiffies + system\_jiffies) * \frac{1}{HZ}}{elapsed\_time} \tag{4.2}$$

Further on `/proc/loadavg` is polled to get information about the system load. The first three fields of this file depict the load average, whereas the order corresponds to the average of 1, 5 and 15 minutes. Load average [Gun07] is an exponentially weighted, moving average of processes in the run queue marked running/runnable (state R) or uninterruptible (state D, which usually means waiting for I/O). The kernel gives a process a specific amount of time, which is usually 10 ms (determined by the HZ variable), for demanding the CPU. If the process did not give back control after that time, the previously mentioned hardware interrupt occurs and the kernel takes back control. Now the jiffies counter is increased by one and a check, whether the load average should be calculated, is made. The function `calc_load` is invoked every $500 * HZ \mathrel{\widehat{=}} 5s$. The formula used is an exponential decay function:

$$e.g.\ the\ 60s\ load\ value = (load(t-1) * 1884) + n(t) * (1 - 1884) \tag{4.3}$$

$load(t-1)$ is the previous load, 1884 is the multiplier for the moving average and $n(t)$ is the number of running or uninterruptible processes. So load average represents the CPU usage and the demand for the CPU. A value of one implies CPU utilization (assuming one CPU) such that one process is permanently in the run queue (using the CPU) and no others are waiting. If the load is higher than one, the CPU is used 'more' and the demand for it is higher. [Wal06]

Load average is more suited to display the load of a system than the bare CPU utilization in percent, because it has no upper limit, and a statement about the demand in more CPU power can be made more accurately.

The remaining problem to solve is, how to get the process id (PID), which is required to determine the patch `/proc/<pid>/...` Since the PID is unknown before a program is started and this information is only available during the process exists in the system, a way how to retrieve the PID needs to be found. The most obvious way is to let the tool start the desired program by forking a process and thus get the PID as return value.

```
child.pid = fork();
```

The fork executes the desired task. Then the PID is known and can be integrated into the path variable for the statm (and stat) file of the process, which is read periodically until the fork exits.

Stress

Stress is a small tool to put load on a host. It was used to gain a load value for comparison with the values determined later on. It allows to spawn a specified number of forks running `sqrt()`, `sync()` and/or `malloc()/free()`. The amount of memory to be allocated can also be specified. We used half of the available RAM.

`$ stress --cpu 2 --io 1 --vm 1 --vm-bytes 32M --timeout 60s` caused a load average of 2.54 over 1 minute on the Gumstix.

Iperf

Iperf allows measuring the maximum TCP bandwidth between two hosts. It was deployed to examine the maximum rate the Gumstix can send and receive.

send [from] →[to]	$MBits/sec$
Notebook → Gumstix	46.3
Gumstix → Notebook	33.6
PC → Notebook	92.9

The requesting system is notebook is a core duo with 2GHz each and 2GB RAM. The PC is an Athlon with 1.4GHz and 256MB RAM. The connection is based on fast Ethernet with 100MBit/s full duplex. The platform for the client is the Gumstix introduced in Section 1.3. Both are interconnected via a 100 MBit/s switch (see Figure 4.1).

These values allow to investigate whether the maximum bandwidth is responsible for a stagnating number of requests.

4.1.3 Measurement

The `RequestService`, which is representative for the generator is responsible for retrieving the response times and thus marked yellow in Figure 4.1. Pidstatm collects system data and also does averaging. That allows displaying all desired values at the end of a measurement without the need of storing any values on the Gumstix for later calculations, because long measurements can produce a lot of data. Procnet measures (stage 1) the amount of traffic. Stats.py calculates (stage 2) the statistics required for gnuplot. The gnuplot scripts generate the graphs for presentation (stage 3).

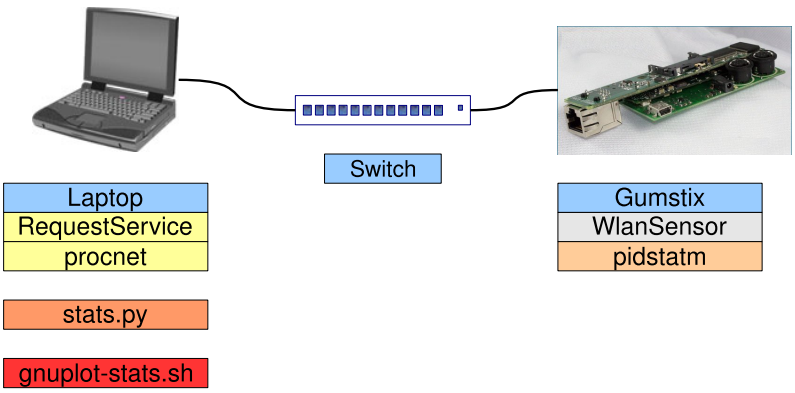


Figure 4.1: Evaluation Architecture.

All tests are based on the WLAN sensor. Because of time constraints only one sensor was compared. During the tests the execution of the binary was disabled and the period for regenerating the internal accesspoint database was increased to disable its influence on the measurement. This also assures that the returned value of a request is the same each time. MundoCore and R-OSGI used TCP as transport protocol.

MundoCore

MundoCore allows choosing between two formats for the send messages, a datatype-oriented XML format or a binary format.

That section compares the performance of the Java and the C++ version. The tests were performed on an Athlon 64 system with 2 GHz with client and server on the same system:

language and protocol	mean call time [ms]
Java BIN	0.72
Java XML	2.67
C++ BIN	0.5
C++ XML	0.59

Transports on the same host are shared memory based, which speed things up. C++ XML is much faster than Java XML in comparison to its binary protocol, because Java XML uses the Apache Xerces XML parser while C++ uses a custom build, speed-optimized XML parser.

MundoCore-Bin

At first a network protocol analyzer (Wireshark) was used to capture a MundoCore session with the binary protocol. By isolating some single requests it was possible to determine the number of packets and their size caused by a single method call and its answer:

middleware	packetcount	packetsize (incl. IP & TCP header) [bytes]
MundoCore (bin)	4	398 + 52 + 530 + 52 = 1032

In the column for packetsize the size of all individual packets are summed up.

The single mode of the request generator delivered a mean response time of 42.8 ms for a single request running 1000 requests in sequence.

The burst mode delivers the maximum number of requests the device can handle. Averaged over 10 minutes, the test resulted in 20.5 requests per second. Several tests over one minute gave close to the same value (20.4 rps).

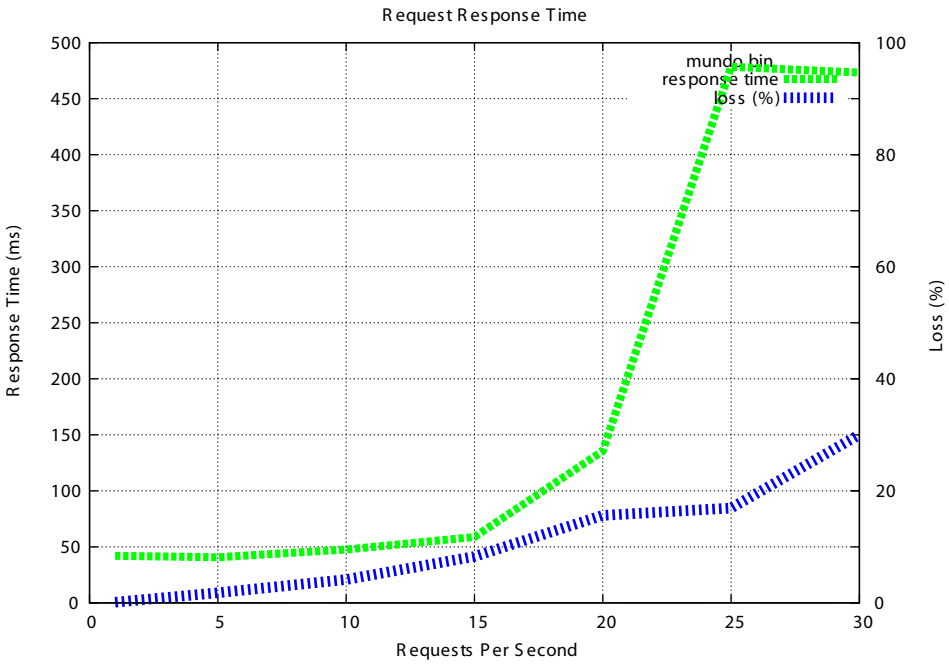


Figure 4.2: Request Response Time (WlanSensor Mundo binary).

Figure 4.2 shows a graph of the request response time at rates from 1 to 30 requests per second (rps). When issuing 15 requests per second (rps) the average call time started to increase while the loss of messages is neglectable until 20 rps. The performance of this sensor is relatively good until 15 rps with MundoCore and the binary format. Above 20 rps, the performance is getting worse since the response time increases significantly and the loss is 16.9% at 25 rps.

Table 4.1 shows some of the collected statistics of requests at different rates over 30 minutes. The first line (0rps) depicts the statistics with the sensor started but without querying it. That means this line represents the traffic caused by discovery and keep-alive messages from MundoCore and the `SensorSurveillance` from the `ServiceDirectory`.

The traffic rises with the rate until 20 rps. Then it is close to the same level for higher rates. That again shows that 20 is the maximum number of requests the device can handle. The average load crosses a value of 1.0 at 20 rps. Since from this rate on more than one process is in the run queue all the time, the response time is expected to increase. This is proven by the numbers in the table and comes particularly clear at 25 rps when the load passes over two and the response time is more than tree times as high as at 20 rps. The CPU utilization is conform to the load and reaches 99% at 25 rps. The memory's resident set size of the application rises about 17% from 1 rps with 10.5 MB to 20 rps with 12.3 MB. The total memory size does not show any clear trend. It is on a stable niveau for all rates.

Table 4.1: Statistics (WlanSensor Mundo bin).

mode	response time [ms]	loss [%]	traffic [kB]	load avg avg (max)	CPU [%]	mem rss [kB]	mem total [kB]
0rps1800	-	-	700	0.04 (0.31)	1	11436	29676
1rps1800	42.0	0.1	2422	0.06 (0.35)	5	10528	29664
5rps1800	40.7	1.8	10039	0.28 (1.08)	21	10500	29504
10rps1800	47.6	4.2	19890	0.49 (1.19)	43	10552	29592
15rps1800	58.8	8.3	29549	0.87 (1.65)	62	11540	29676
20rps1800	135.6	15.6	38728	1.39 (2.45)	84	12332	29572
25rps1800	478.5	16.9	37797	2.48 (3.11)	99	12404	29624
30rps1800	473.3	30.1	38303	2.41 (2.93)	99	11404	29516

4.1.4 MundoCore-XML

Wireshark[3] was used to capture a MundoCore session, but this time XML is used as protocol. Isolating some single requests showed the following number of packets and size:

middleware	packetcount	packetsize (incl. IP & TCP header) [bytes]
MundoCore (XML)	4	708 + 52 + 1076 + 52 = 1888

The single mode of the request generator delivered a mean response time of 165.6 ms for a single request running 1000 requests in sequence.

The burst mode delivers the maximum number of requests the device can handle. Averaged over 10 minutes the test resulted in 4.7 requests per second. Several tests over one minute returned 2.9 rps.

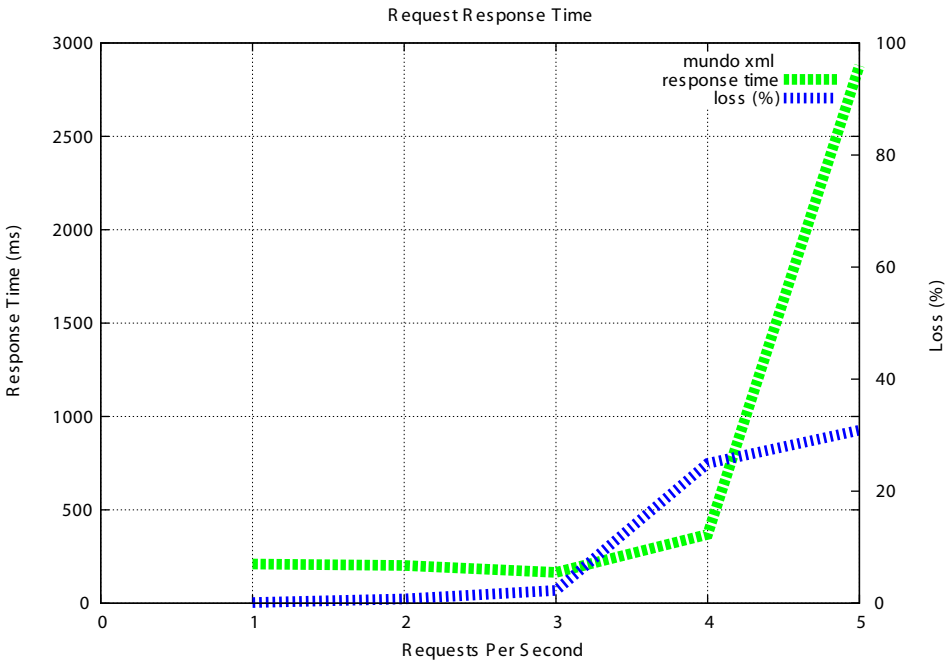


Figure 4.3: Request Response Time (WlanSensor Mundo XML).

Figure 4.3 shows the request response time at rates from 1 to 5 requests per second. The average call time highly rises by a factor of 7.8 from 4 to 5 rps. The loss already increases from 3 to 4 rps to a level of 24.9%. Thus MundoCore with XML is suitable only until 3 rps. This is conform to the calculated maximum number of requests per seconds from the generator's burst mode.

[3] http://www.wireshark.org

Table 4.2: Statistics (WlanSensor Mundo XML).

mode	response time [ms]	loss [%]	traffic [kB]	load avg avg (max)	CPU [%]	mem rss [kB]	mem total [kB]
0rps1800	-	-	1532	0.30 (0.77)	14	11976	29792
1rps1800	210.0	0.1	3772.6	0.08 (0.41)	24	10564	29572
2rps1800	202.1	0.7	7229	0.33 (1.81)	41	10720	29604
3rps1800	164.8	2.3	10692	0.45 (1.69)	50	13592	29804
4rps1800	368.4	24.9	14454	1.35 (2.73)	77	13744	30116
5rps1800	2879.5	30.9	12586	2.16 (2.91)	100	13084	30064

Table 4.2 shows some of the collected statistics of requests at different rates over 30 minutes. The first line (0rps) again stands for the traffic caused by discovery and maintenance.

The traffic rises until 4 rps. The average load average passes over one at 4 rps, which also results in a higher increase of the response time. At 5 rps, the load crosses two and the call time rises by a factor of 7.8. The CPU utilization is at 100% at this point. The memory's resident set size significantly increases by 26.8% to 13.6 MB from 2 to 3 rps. The total memory size of the application shows the trend to increase slightly. The performance of this communication is poor and should not exceed 3 requests per second.

R-OSGI

The capture of a session with R-OSGI shows the following values for the number of packets and their size for a single request and its answer:

middleware	packetcount	packetsize (incl. IP & TCP header) [bytes]
R-OSGI	3	173 + 268 + 52 = 493

The single mode of the request generator delivered a mean response time of 4.7 ms for a single request running 1000 requests in sequence.

The burst mode delivers the maximum number of requests the device can handle. Averaged over 10 minutes the test gave 198.6 requests per second. Several tests over one minute gave 199.6 rps.

Figure 4.4 shows the request response time at rates from 1 to 300 requests per second. At 200 requests per second the average call time increases to 35.8 ms and the loss is the first time above one with 3.9%. Until that point the performance of the sensor with R-OSGI is really good. Above 200 rps the response time increases to a stable level of 49 ms and the loss rises above 19%. That confirms the value of around 199 rps as the maximum for R-OSGI.

Table 4.3 shows some of the collected statistics of requests at different rates over 30 minutes. The first line (0rps) represents the basic traffic caused by discovery and service maintenance.

Table 4.3: Statistics (WlanSensor R-OSGI).

mode	response time [ms]	loss [%]	traffic [kB]	load avg avg (max)	CPU [%]	mem rss [kB]	mem total [kB]
0rps1800	-	-	358	0.02 (0.13)	0	10864	29512
1rps1800	5.2	0.2	1316	0.02 (0.19)	1	10864	29512
50rps1800	21.6	0.9	42104	0.25 (0.60)	23	10864	29480
100rps1800	12.8	0.9	83859	0.54 (0.87)	46	10864	29504
150rps1800	12.6	0.9	125620	0.86 (1.25)	72	10864	29512
200rps1800	35.8	3.9	167223	1.01 (1.18)	93	10864	29508
250rps1800	49.2	19.6	168325	1.02 (1.13)	95	10864	29512
300rps1800	49.1	32.8	168831	1.02 (1.23)	95	10864	29512

The traffic increases until 200 rps. Then it is close to the same level for higher rates. That approves 200 as the maximum number of requests possible with low loss and moderate response time. The average load average also crosses one at this point, but never reaches two, even whith higher request rates. The average resident set size is the same (10.9 MB) for all tested rates, while the application's total memory consumption is at a stable level of 29.5 MB.

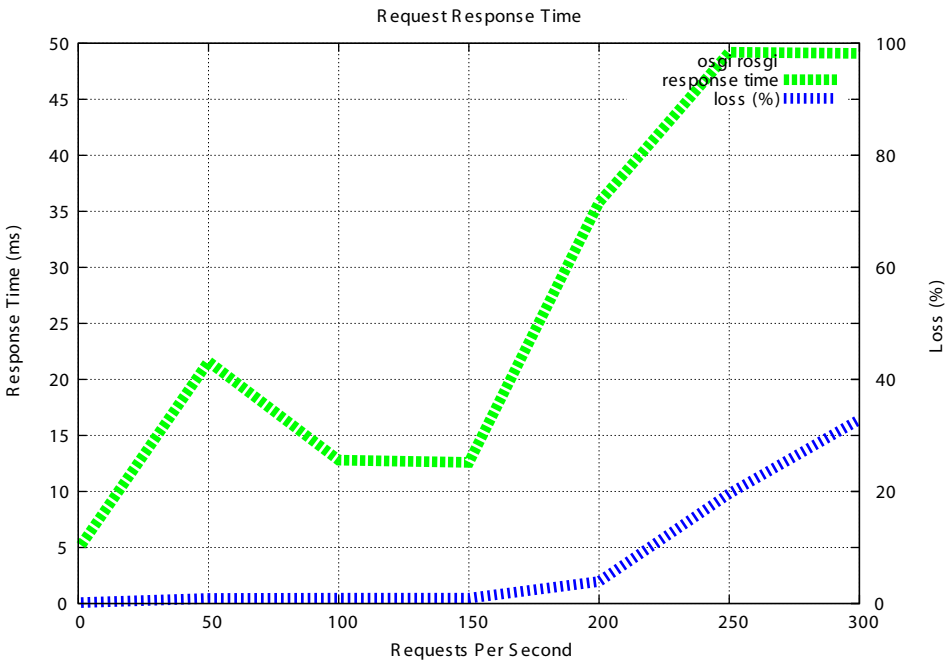


Figure 4.4: Request Response Time (WlanSensor R-OSGI).

4.1.5 Discussion of Results

The performance of the Gumstix heavily depends on the used middleware. The most efficient approach is R-OSGI, which can handle up to 199 requests per second. The response time for a single request is 4.7 ms to 35.8 ms for the recommended maximum rate. R-OSGI also has, with 493 bytes, the lowest amount of traffic for a complete request with the used data. The memory usage is very stable over different request rates at 10.9 MB resident set size (RSS). The load and CPU utilization is, with 23%, still moderate for "lower" rates up to 50 rps. So the power consumption is assumed to be relatively low in normal operation.

Second rank is MundoCore with the binary protocol, which can accomplish up to 20 requests per second. The response time for a single request ranges from 42.8 ms to 135.6 ms for the recommended maximum rate. The amount of traffic for one request with the testdata is 1032 bytes. The memory usage rises to a level of 12.3 MB RSS. The CPU utilization already reaches 21% at a rate of 5 rps.

MundoCore with the XML protocol has the lowest performance, which can only cope with 3 requests per second. The response time for a single request is 164.8 ms for the recommended maximum rate. MundoCore's XML protocol causes, with 1888 bytes, the most traffic for a single request (with our data) including the answer. The memory usage ascends up to 13.6 MB RSS. The CPU utilization achieves 24% with only one request per second.

As already mentioned in Section 3.3.1, SLP, which is used for service discovery, uses UDP multicast per default. MundoCore in contrast, sends UDP broadcasts for node discovery. Multicast can be problematic with routers and switches.

The differences between MundoCore and Concierge with SLP and R-OSGI in setup are that SLP requires a default gateway in the routing table, which is in the same subnet as the first IP address of the network interface. Both require valid resolution of the local hostname via DNS or /etc/hosts.

R-OSGI keeps alive a connection once established. MundoCore on the other hand closes its connections as the varying port numbers prove.

Each OSGi framework instance is autonomous, hence each one needs to be configured with a different port for R-OSGI and jSLP if multiple frameworks should run.

The life cycle management layer (see Section 3.1) of OSGi allows to dynamically load/unload and start/stop bundles during runtime. That is part of the OSGi architecture and therefore already fully functional without the need of the implementation of an extra plug-in handler. That can be used to easily separate functionality in modules (bundles), which are only loaded if necessary. The `RequestService` for example could be separated in a bundle and started only on purpose if it is used.

Thus the complete Service Directory requires common service classes for MundoCore's plug-in mechanism, these are obsolete for the OSGi implementation.

MundoCore uses stubs for remote invocation. To successfully generate them, special annotations in the Java code are necessary, which are parsed by the MundoCore precompiler.

The difference in 'session/packet' size between MundoCore bin (Section 4.1.3) and R-OSGI (4.1.4) is caused by the fact that the MundoCore packets contain 'Mundo overhead' like abstraction by Mundo-`TypedMap` and passivate information for the stubs.

4.2 Search Performance

In this section, the performance influence of several components is discussed. In detail this concerns the communication scheme between gateway and motes, the impact of caching on the overall network performance, and the influence of the selective search algorithm on the overall delay between application request and answer. Section 3.7 described the caching mechanism. The sensor data that arrives at the SD is stored in the cache and is used for future requests of this data until the data is again removed from the cache after a certain time. The performance gain of caching is mainly determined by three elements:

1. **Validity Period** The time period in which the sensor's value is considered as not changing. This period is described as *expiration time* in Section 3.7. The value varies a lot between different types of sensor. E.g. an email address entry has a longer Validity Period than the position output of a mobile GPS sensor. Table 4.4 gives an overview of our sensor nodes and their estimated Validity Periods. A longer Validity Period increases the use of cached values but decreases the correctness probability.

2. **Request interval** The time between two requests of the same sensor data. If only one application is requesting this data, this interval is equal to the interval in which the application starts search requests. In case of two or more applications requesting this data, the request interval is determined by the shortest time between the requests of two applications.

3. **Difference in delay between cached and requested data** The value that actually speeds up the performance is the time difference in obtaining the sensor value either from the sensor or from the cache. Since the cache is located in the SD server, this value is close to zero. The delay between sensor and SD will vary depending on the operations the sensor node has to perform before answering the request. The influence of the physical medium and the MAC layer can be neglected, because the measured delays are in the range of several milliseconds. Table 4.4 shows the measured delay times for each sensor node individually.

Table 4.4: Sensor Validity Period and Delay

Sensor	Validity Period	Δdelay [ms]
Chair sensor	30 sec	25
Outlook appointments	5 min	93
Outlook contacts	1 hour	408
Activity sensor	30 sec	23
PersonalBadge sensor	1 min	23

The usage of the cache is dependent on the relation between *Validity period* and *Request interval*. If the Request interval is longer than the Validity Period, the cached value expires before a new request arrives. In this case the cache is useless. If the Validity Period is longer, the ratio of cached values to newly requested values is calculated by:

$$Ratio = \left\lfloor \frac{ValidityPeriod}{RequestInterval} \right\rfloor$$

Let's assume a sensor has a Validity Period of 50 seconds and the application requests this sensor data each 20 seconds. Together with the first request the sensor value is written to the cache. Then the second and third request can use the cached value. After 50 seconds the cached value expires and for the fourth request of the application, the sensor has to be invoked again. In this example the cached value is used twice and the sensor is requested once before the whole process repeats. Thus the Ratio is two, according to

$$\left\lfloor \frac{50sec}{20sec} \right\rfloor = \lfloor 2.5 \rfloor = 2$$

Table 4.5: Message Reduction by applying Caching Methods and Expiry Information

Sensor	Ratio (% cache hits [20 sec]	[60 sec]	Message reduction per hour [20 sec]⇒180 req	[60 sec]⇒60 req
Chair sensor	1 (50 %)	0 (0 %)	90	0
Outlook appointments	15 (94 %)	5 (83 %)	168	5
Outlook contacts	180 (99 %)	60 (98 %)	179	59
Activity sensor	1 (50 %)	0 (0 %)	90	0
Personal badge sensor	3 (75 %)	1 (50 %)	135	30

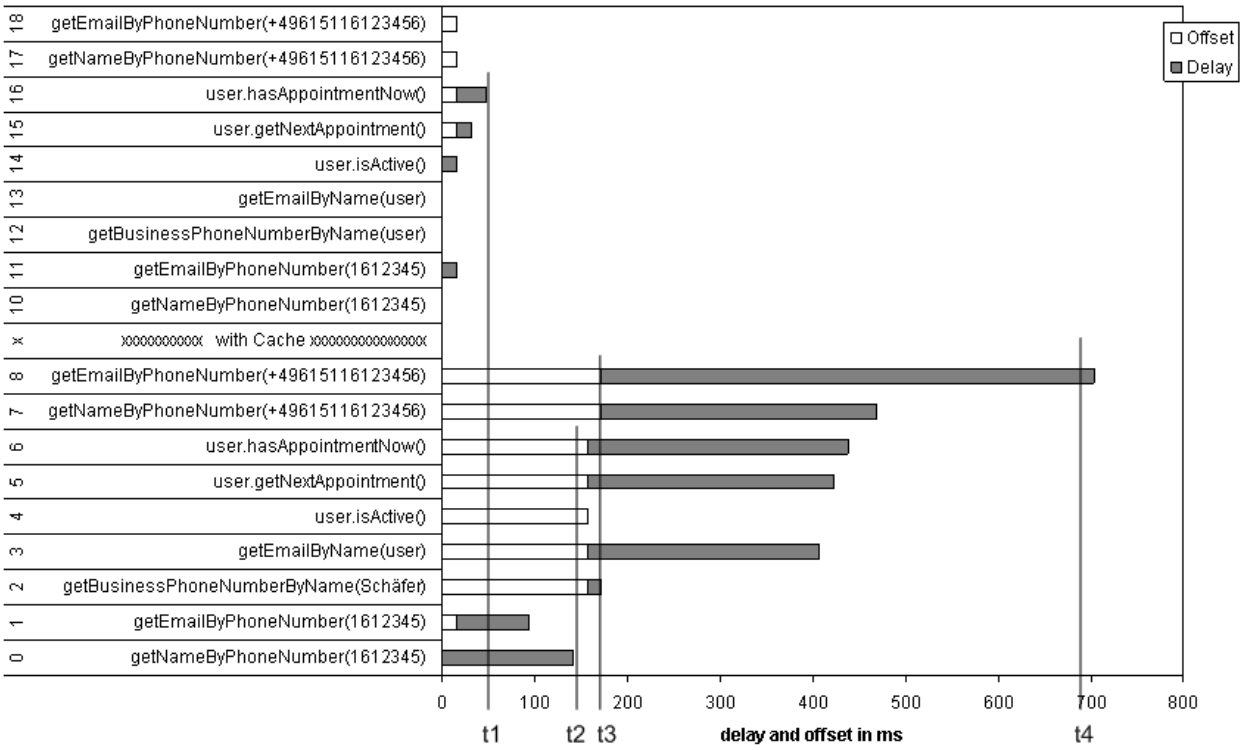


Figure 4.5: Difference between delays and offsets due to the cache

The ratio also gives a percentage of the amount of reduced messages to all messages. If the ratio is two, the use of the cache reduces the amount of messages between SD and sensor for 66%. Table 4.2 shows some calculated ratio values (and the equivalent percentage in brackets) for a request each 20 seconds and again each 60 seconds. In the second column, the amount of messages that can be reduced by the use of the cache in one hour is given. That means if the request period is 20 seconds, there are 180 requests in one hour. If the ratio is equal to one (50%), it means that 90 of the 180 requests are omitted by the use of the cache.

Subsequent an example is described to show the influence of the cache on the end-to-end delay for one entire search process. Involved in this example are the Outlook appointments sensor, the Outlook contacts database and the activity sensor. The search starts with one initial phone number. The Outlook contacts database provides a name to this number. The Outlook appointments sensor and the activity sensor are invoked with this name in the next step. One minute after the search results are returned, the search is started a second time in order to use also cached information. During the second search process the Outlook contact data is taken from the cache. The data of the activity sensor and the Outlook appointments sensor are expired and have to be requested again. Figure 4.5 shows the individual delays and timing for both search processes. The lower half of the figure (rows 0 to 8) shows the first search process. All sensors have to be requested since no cached values are available. The search process is started a second time after 1 minute. The upper half of the figure shows the second search process (rows 10 to 18). The data of the Outlook database is present in the cache. Hence these values are available immediately and the corresponding requests (rows 10, 12, 13, 17, 18) have a delay of 0 ms. As a result, the activity sensor and the Outlook appointment sensor are requested much earlier than during the initial

search process. The delay of the first invoked method (row 0) leads to an offset time of about 150 ms at t2, which can be reduced to zero in the second search run. The search process that makes use of cached values is finished at t1 after 47 ms while the initial search process needs 700 ms to complete. Two additional points regarding the cache can be drawn of this comparison. First, it could be useful to cache empty values that are returned from sensor that can not provide any data to the request. The method `getEmailByPhoneNumber` returns email addresses that are linked to a phone number. But in this example the method can not find an entry to the given number and returns `null`. The cache does not save `null` values and therefore this method is invoked in the second search process again and causes some delay. The second point to mention is the difference between the delays of the Outlook appointment sensor in the initial search process and in the second one. In both cases, the sensor has to be requested because the appointment entries have expired. But in the second search run, the delay is about ten times shorter. A possible explanation for this difference can be that both the Outlook contacts database and the appointments sensor access the Outlook application. But in the second search run only the appointments sensor is accessing the Outlook data sets and less parallel running threads presumably lead to this significant shorter delay.

4.2.1 Discussion of Results

The measurements clearly indicate the cache has an influence on the amount of messages that need to be sent between Service Directory server and sensor but also on the delay for a single request and on the delay for an entire search process. Regarding wireless and battery powered sensor like the badge sensor system and the chair mote application, a reduction of messages is desirable to save battery lifetime. Unfortunately these sensor types provide data types that have a short Validity Period and therefore the use of the cache shows a minor effect in comparison to other sensors like databases. On the other hand, regarding sensors like databases where reduction of messages are of minor concern because bandwidth and energy are almost unlimited, the use of the cache shows advantages on the search end-to-end delay when large data sets cause a longer particular delay. In general the efficiency of the cache increases when more applications are accessing the same sensor and thereby the number of requests per time period to one sensor rises. Future work should include long term analysis of a real world deployment. A more powerful statement about the usage and efficiency of the cache can be given when more sensors and several applications that could send requests in different intervals or at random time instances are involved in a simulation.

Aspects of the adaptive search algorithm

The adaptive search algorithm speeds up the search process by defining the application relevant sensor data types. The search process will not request all available sensors and therefore the search process stops early. Imagine an application that just requires the location of a particular person. In this case the adaptive search algorithm will only request the badge sensor and the chair sensor. The search engine would go on and request the contact information and appointments of the given person, and with the obtained location maybe further persons that are connected. The additional iteration will prolong the total search time although the actual desired information is already present. Subsequently we present a search example and give the individual delays of each invocation. Figure 4.6 shows the individual delays and starting times of each sensor invocation. Involved in this example are the Outlook appointment sensor (row 8 and 9), the Outlook contacts database (rows 1, 2 and 6, 7 and 10 - 13), the badge sensor (row 0), the chair sensor (row 3) and the computer activity sensor (row 4). The initial search entry is userId "userA". With this entry the Outlook database, the badge sensor and the chair sensor are requested. The badge sensor returns the roomValue "MobileLab" at t1, which provokes the invocation of the Computer Activity sensor. This sensor is registered on the one hand to the roomValue but also to the userId "userB", which can be used as intermediate result immediately. Therefore still at t1, the Outlook database and the appointment sensor are requested with this userId. At t2, the Outlook database returns the phone number of "userA" (row 1) which is used to request the Outlook database again (rows 10 and 11). The outlook database that was invoked in t1 with "userB" returns the phone number at t3 which leads to a new invocation. Finally, at t4 all invoked services have returned their results and can be passed to the calling application. If the application requires some information about the location of one particular user, the relevant information is present when the badge sensor and the chair sensor have returned their results, in this case after 156 ms. Without the selectively working search algorithm, all sensors are requested and the entire result will not return before 1761 ms. When more sensors are connected in this network, this difference will even increase. Thus a search algorithm that can be parameterized with the elements that it will search for and that are allowed as intermediate results for iteration is a necessity for the network information processing.

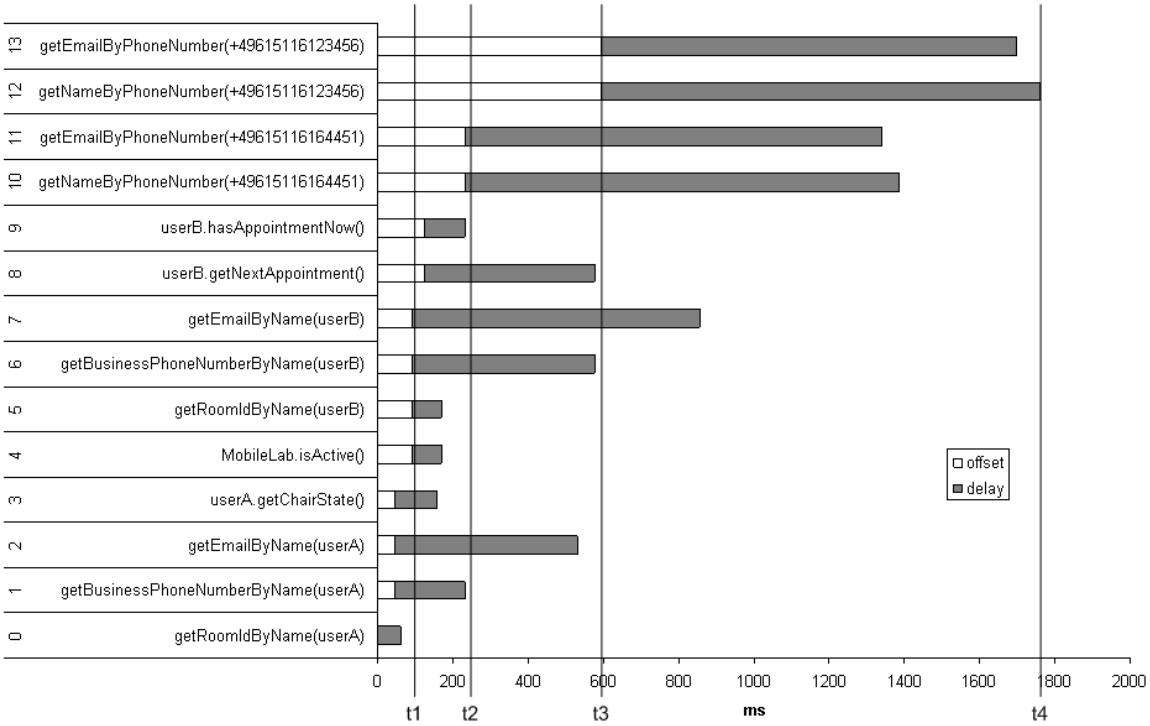


Figure 4.6: Individual delays and offsets of sensor services

5 Conclusion

This work presented the communication aspects of the Context Framework. We introduced our vision of context-aware communication and the envisioned application scenario. The requirements and key issues of the framework resulting from the desired scenario where determined and related work was identified and compared. The resulting architecture was introduced and feasible approaches where pointed out for each issue. The major issues where explained in detail, which includes the topics Services Communication, Service Discovery and Semantic Sensor Lookup. This work gave some insights into some details of the implementation and operation of the Context Framework. Due to the crucial impact of the communication's performance to the overall performance of the system, we performed some performance tests. The test setup and results were also presented.

The Context Framework aims to connect all types of sensors, which are different in delay or size of output data, and also is intended to serve different applications, that have different characteristics in delay constraints and request frequencies.

From the requirement of the sensor data classification in the Ontology and the requirement of a dynamic and growing sensor network, the need for an automatic and dynamic extension of ontologies arises. In contrast to related work, our approach is to use the OWL-S descriptions of the sensors and extend our main Ontology while other projects try to merge different ontologies or translate from one to another. The desired interoperability between all kinds of sensors or information sources, the different network implementations with ontologies of different domains and the end applications will be the most challenging task for future research. *SUMO* and *OntoSensor* are good starting points in the field of standardized sensor network ontologies. *OWL-S*, even with its Web Service background, is a well qualified approach to make sensors accessible for other applications. Nevertheless, to the best of our knowledge, there is no common way to describe semantics that are interpretable by machines. Description Languages like OWL provide syntax and some predefined elements and properties, but there is no guideline how to use OWL to express a certain meaning. By creating user-defined properties and classes, it is possible to write syntactically correct OWL ontologies that will still be useless to other implementations.

The proposed system has a number of positive features making it unique. It does not use some proprietary definitions, but uses OWL. This makes the project interesting for a very active community. The same applies for OWL-S. The Profile and Process documents can theoretically be used by other OWL-S software to identify services, so that sensors and service directories can be included in more applications.

The ontologies and semantics created for the network makes it easy for the search algorithm to find possible sensors for queries. Thereby the search can gather as many information as possible for a given data. By updating the network Ontology it is also possible to add new kinds of sensors to the network without having to modify the service directories source code.

The performance aspect of the network leads to the implementation of the cache. The cache reduces delays, communication traffic and, in case of motes, battery power consumption. The heterogeneity of the cached data is indicated by the sensor dependent caching time. QoS was address briefly in this project, with a timeout for search queries and scalability. Other QoS features can be implemented, such as robustness, security or availability to enhance the system.

The search algorithm deals with the heterogeneity of sensors and different applications. The search algorithm can be parameterized with application specific fields of interest so that only sensors of appropriate types will be included in the search procedure. The sensor data ordered and classified in a data class hierarchy in the KOMSSN Ontology. The search algorithm is able to iterate over intermediate sensor values to get to the actual desired sensor values. This facilitates that sensors of interest, e.g. that are linked to a person, can be requested even if the application only has the phone number of this person as search argument and there is an intermediate sensor that translates from phone number to person's ID.

During our investigations, we identified that the following areas have to be considered in future activities more intensively:

- *Search algorithm.* For several sensor types an event based data transmission or a publish/subscribe mechanism is more efficient than the request based approach. Notifications about state changes require constant polling of the involved sensor, which also interferes with the caching mechanism. On the other hand, the request approach does not have the problem of integrating e.g. a data base in an event-based or publish/subscribe system. Future work will have to find a combination that takes the advantages of both models. If a publish/subscribe or a event based messaging system will be integrated into the Context Framework, two or more search methods might exist simultaneously, where the selection has to be made according to the desired optimization criterion.

- *Sensor data representation and application interface to the network.* In the current implementation, the resulting sensor data set of a search query is stored in a tree of Java objects. An application needs to establish a connection to the Context Framework, starts a search and finally retrieves the Java tree containing the desired sensor values. This interface to the network and the result representation is not optimal regarding interoperability between several networks and applications. There should be a standardized interface to the network, i.e. as a Web Service, enabling different applications to access the networks data without knowledge about the implementation details.

- *Aggregation and data fusion.* Currently the data is passed to the application as provided by the sensor. A desirable feature would be to transform the representation to another format or a higher semantical level, e.g. like the *Semantic Engine* of [WPJ07] does. Moreover it could combine several sensor readings to a more powerful statement or to a conclusion that can be drawn of the single sensor information. In [MS06] so called *Modeling agents* are considered to fulfill this task, while our idea is to use intermediate or virtual sensors that will be accessed like any other sensor. This special sensor will process its output according to a given description from the input it retrieves from other "real" sensors.

- *Distributed Service Directory.* A future bottleneck of the current system might arise from the fact that each Service Directory holds a list of all sensors available within the network segment. For a large number of nodes this will cause high memory demands at the Service Directory. Additionally, the search gets more complex. An alternative would be to balance the load caused by the sensors to several Service Directories.

- *Search Query.* The search service could be used for other purposes than the communication system scenario. Therefore, more detailed search queries must be supported. For now the search handles requests such as *"Get all information on a number"*. If the system shall be applicable to other scenarios also queries like *"Get all lights switched on in the first floor."* must be supported. Therefore a query language is required. The Jena Framework already supports such a language, named "SPARQL". A new search engine supporting this search queries would enhance the system to use in other scenarios.

Bibliography

[Ait06] Erwin Aitenbichler. *System Support for Ubiquitous Computing*. Shaker Verlag, Aachen, 2006.

[Ait07] Erwin Aitenbichler. MundoCore. Website, 2007. http://www.tk.informatik.tu-darmstadt.de/index.php?id=mundocore. Last visit: 09.12.2008.

[And04] Robert Andresen. Monitoring Linux with native tools. Speech, 2004. 30th Annual International Conference of the Computer Measurement Group, Inc.

[AWSBL99] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The Design and Implementation of an Intentional Naming System. In *17th ACM Symposium on Operating Systems Principles*, Charleston, SC, December 1999.

[BBK02] M. Balazinska, H. Balakrishnan, and D. Karger. INS/Twine: A Scalable Peer-to-Peer Architecture for IntentionalResource Discovery. In *International Conference on Pervasive Computing 2002*, Zurich, Switzerland, August 2002.

[Ber93] Philip Bernstein. Middleware - An Architecture for Distributed System Services. Technical report, Cambridge Research Lab, 1993.

[Ber96] Philip Bernstein. Middleware - A Model for Distributed System Services. *Communications of the ACM*, 39(2), February 1996.

[BGS02] Michael Beigl, Hans W. Gellersen, and Albrecht Schmidt. Multi-sensor context awareness in mobile devices and smart artifacts. *Mobile Networks and Applications*, 7(5):341–351, 2002.

[BLHL01] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 2001.

[Blu04] Bluetooth Consortium. Bluetooth Core Specification v2.0 - Volume 3 Part B - Service DiscoveryProtocol (SDP), November 2004.

[BR00] C. Bettstetter and C. Renner. A comparison of service discovery protocols and implementation ofthe service location protocol, 2000.

[BRA02] R. Bajaj, S.L. Ranaweera, and D.P. Agrawal. GPS: Location-Tracking Technology. *Computer*, 35(4):92–94, 2002.

[CG06] David J. Russomanno Caleb Goodwin. An Ontology-Based Sensor Network Prototype Environment. *Fifth International Conference on Information Processing in SensorNetworks*, pages 1–2, 2006.

[Con04] OW2 Consortium. Project Info - Oscar - OSGi Framework . Website, 2004. http://forge.objectweb.org/projects/oscar/. Last visit: 09.12.2008.

[Cor07] Intel Corporation. Intel Xscale Technolopy. Website, 2007. http://developer.intel.com/design/intelxscale/. Last visit: 09.12.2008.

[Die03] Jörg Dieckmann. DAML+OIL und OWL: XML-Sprachen für Ontologien, February 2003.

[DKS06] Maria Danninger, Tobias Kluge, and Rainer Stiefelhagen. MyConnector: Analysis of Context Cues to Predict Human Availabilityfor Communication. In *Proceedings of the 8th International Conference on Multimodal Interfaces*, pages 12–19, 2006.

[Dub07] Dublic Core. The Dublin Core Metadata Initiative, February 2007. http://dublincore.org. Last visit: 09.12.2008.

[GPVD99] Erik Guttman, Charles Perkins, John Veizades, and Michael Day.
RFC 2608 - Service Location Protocol, Version 2. RFC, 1999.

[Gun07] Neil Gunther. Leistungsdiagnostik. *Linux Magazin*, 08/07:84–91, 2007.

[Gör05] Manuel Görtz. *Effiziente Echtzeit-Kommunikationsdienste durch Einbeziehung von Kontexten*. PhD thesis, Technische Universität Darmstadt, 2005. http://elib.tu-darmstadt.de/diss/000592/.

[HCZK02] T. D. Hodes, S. E. Czerwinski, B. Y. Zhao, and A. D. Josephand R. H. Katz. An Architecture for Secure Wide-Area Service Discovery. *Wireless Networks*, 8(2/3):213–230, 2002.

[Hew07] Hewlett-Packard Development Company. Jena - A Semantic Web Framework for Java, 2007. http://jena.sourceforge.net/. Last visit: 09.12.2008.

[How96] Tim Howes. A String Representation of LDAP Search Filters. RFC, 1996. ftp://ftp.rfc-editor.org/in-notes/rfc1960.txt.

[Jin06] Jini.org. The Community Resource for Jini Technology, September 2006.

[Ker05] Michael Kerrisk. PROC(5) Linux Programmer's Manual. manpage, 2005.

[KKRO03] M. Klein, B. Konig-Ries, and P. Obreiter. Lanes - A lightweight Overlay for Service Discovery in Mobile AdHoc Networks. Technical Report Technical Report 2003-6, Technical University Karlsruhe, May 2003.

[Lac05] Lee W. Lacy. *OWL: Representing Information Using the Web Ontology Language*. Trafford Publishing, 2005.

[LGL+98] Yan Liu, Ian Gorton, Anna Liu, Ning Jiang, and Shiping Chen. Designing a Test Suite for Empirically-based Middlware Performance Prediction. In *40th International Conference on Technology of Object Oriented Languages and Systems*, 1998.

[LH02] C. Lee and S. Helal. Protocols for discovery in dynamic and mobile networks. *International Journal of Computer Research*, 11(1):1–12, 2002.

[Mak07] Makewave. Knopflerfish osgi. Website, 2007. http://www.knopflerfish.org/.

[MCE02] Cecilia Mascolo, Licia Capra, and Wolfgang Emmerich. Mobile computing middleware. Technical report, Dept. of Computer Science University College London, 2002.

[Mis07] Miscellaneous. KernelGlossary. Website, 2007. http://kernelnewbies.org/KernelGlossary.

[MM98] Julie McCann and Keith Manning. Tool to Evaluate Performance in Distributed Heterogeneous Processing. In *6th Euromicro Workshop on Parallel and Distributed Processing, Spain,*, pages 180–185, January 1998.

[MPHS05] R. S. Marin-Perianu, P. H. Hartel, and J. Scholten. A Classification of Service Discovery Protocols. Technical Report TR-CTIT-05-25, Univ. of Twente, Enschede, June 2005.

[MS06] Deshendran Moodley and Ingo Simonis. A New architecture for the Sensor Seb: the SWAP Framework. In *ISWC 2006: 5th International semantic web conference*, 2006.

[OWL04] OWL-S Coaltion. OWL-S: Semantic Markup for Web Services, November 2004. OWL-S 1.1 http://www.daml.org/services/owl-s/1.1/overview/.

[PNL02] Adam Pease, Ian Niles, and John Li. The Suggested Upper Merged Ontology: A Large Ontology for the SemanticWeb and its Applications. In *Working Notes of the AAAI-2002 Workshop on Ontologies and the SemanticWeb*, 2002.

[RAR07] Jan Rellermeyer, Gustavo Alonso, and Timothy Roscoe. R-OSGi Distributed Applications through Software Modularization. Technical report, Department of Computer Science, ETH Zürich, 2007.

[RCKT05] Russomano, Cartik, Kothari, and Thomas. Building a Sensor Ontology: A Practical Approach Leveraging ISOand OGC Models. *The 2005 International Conference on Artificial Intelligence*, pages 637–643, 2005.

[Rel07a] Jan Rellermeyer. concierge. Website, 2007. http://concierge.sourceforge.net/.

[Rel07b] Jan Rellermeyer. R-OSGi. Website, 2007. http://r-osgi.sourceforge.net/.

[RFH+01] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A Scalable Content-Addressable Network. In *SIGCOMM '01: Proceedings of the 2001 Conference on Applications,Technologies, Architectures and Protocols for Computer Communications*, pages 161–172, 2001.

[SHS07] Johannes Schmitt, Matthias Hollick, and Ralf Steinmetz. Der Assistent im Hintergrund: Adaptives Kommunikationsmanagement durch Lernen vom Nutzer. *PIK II/2007 - Current Trends in Network and Service Management*, Apr 2007.

[Sir04] Evren Sirin. Mindswap OWL-S API. http://www.mindswap.org/2004/owl-s/api/, 2004.

[SKHR07] Johannes Schmitt, Matthias Kropff, Matthias Hollick, and RalfSteinmetz. The Virtual Assistant: Framework and Algorithms for User-AdaptiveCommunication Management. Technical Report KOM-TR-2007-08, TU-Darmstadt, Multimedia Communications Lab (KOM), Germany, 2007.

[Sun06a] Sun Microsystems. Jini Network Technology, Specifications v2.1, September 2006. http://www.sun.com/software/jini.

[Sun06b] Sun Microsystems. JXTA Technology, September 2006. http://www.sun.com/software/jxta.

[UPn06a] UPnP. Universal Plug and Play Forum, September 2006. http://www.upnp.org.

[UPn06b] UPnP. Universal Plug and Play Forum - UPnP Resources, September 2006. http://www.upnp.org/resources.

[VGPK99] J. Veizades, E. Guttman, C. Perkins, and S. Kaplan. Service Location Protocol, Version 2, June 1999. RFC 2608.

[VRV05] Matthieu Vallée, Fano Ramparany, and Laurent Vercouter. Dynamic Service Composition in Ambient Intelligence Environments:a Multi-Agent Approach. In *Proceeding of the First European Young Researcher Workshop on Service-OrientedComputing*, 2005.

[W3C03] World Wide Web Consortium W3C. Web Ontology Language (OWL) - Overview, February 2003. http://www.w3.org/TR/owl-features/.

[W3C07] W3C Recommendation. RDF Schema, February 2007. http://www.w3.org/TR/rdf-schema.

[Wal06] Ray Walker. Examining Load Average. *Linux Journal*, December 2006. http://www.linuxjournal.com/article/9001.

[WPJ07] Alex Wun, Milenko Petrovi, and Hans-Arno Jacobsen. A system for semantic data fusion in sensor networks. In *DEBS '07: Proceedings of the 2007 inaugural international conferenceon Distributed event-based systems*, pages 75–79, New York, NY, USA, 2007. ACM.

[Zig06] ZigBee Alliance. Zigbee specification, December 2006. www.zigbee.org.