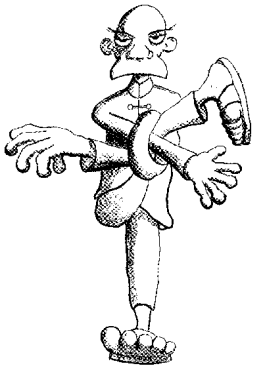
# Coordination Model for Pervasive Computing

## A model to create and design applications using a pervasive middleware



Master Thesis


Benjamin Hadorn

March 2010

**Thesis supervisor**:

Prof. Dr. Béat Hirsbrunner
PAI Research Group

University of Fribourg (Switzerland)

"We make a living by what we get, we make a life by what we give."

- *Sir Winston Churchill*

# Acknowledgements

I would like to thank Pascal Bruegger for his support and contribution to the coordination model for pervasive computing and for sharing his ideas to realize the case study about entity tracking. His work contributed to create the contextual and activity based application in a reliable way. I am grateful to Reto König for his good advice on the Android platform. To Agnes Lisowska for giving me good advise and ideas how to write this document. I'd like also to thank Prof. Beat Hirsbrunner for his support and mentoring during this master thesis. I enjoyed working for the PAI research group and to be part of it. A special thank also to Wolfgang Labus from Studer AG, who made it possible to spend half an year on this research and to reduce my workload at Studer AG. Last by not least many thanks to my wife Manuela who gave me encouragement and strength during this intensive and time consuming phase of my study.

# Abstract

Pervasive computing deals with computation embedded into regularly used objects. The focus of this field is to support the user with all those devices by eliminating the need for explicit interaction when doing his task. The system should interact with the human in a rather implicit and non-intrusive way. Since the number of devices and their current context used in human activity is variable the system must be based on a flexible and reliable coordination framework. Good coordination minimizes the explicit and intrusive interaction between humans and computational devices.

The goal of this master thesis is to create a generic coordination model, which can be used in various scientific fields like computer science, economics and social science. Generally it defines how the world is composed and what dependencies occur between entities. The definition of activities and communication are essential parts of the model and help to describe the dynamic aspects. The second part of the master thesis concerns a more specific coordination model for pervasive computing, which is derived from the generic coordination model. A coordination language is developed to integrate the model into a pervasive middleware. At the end a prototype is developed to validate the model and the language. It is based on the pervasive platform uMove of the PAI research group and shows how entities are tracked in a smart environment.

**Keywords:** Coordination model, coordination language, context awareness, activity awareness, pervasive services, pervasive coordination

# Table of Contents

# List of Figures

# List of Tables



# List of Definitions

# Listings

# 1 Introduction

Pervasive computing is a relatively new discipline in computer science which aims to bring computational power closer to the physical world. In contrast with desktop interaction, which is based on explicit interaction, pervasive computing tries to support human activity in an implicit way. To achieve this goal a pervasive environment can be considered as a physical environment enriched with computing and communication devices. Therefore the system is mostly hidden to human but still present everywhere. Even though mobile networks are regularly used for computing nowadays and mobile applications have been developed to meet many human needs, we still face the lack of calm computing proposed by M. Weiser [Wei91]. Paul Dourish [BD07] identified some of the reasons, like messy infrastructure and incompatible protocols, which are responsible for different developments of pervasive computing in the last few years. Other reasons might be missing frameworks and the huge gap between the operation systems and the pervasive applications. There exist only a few prototypes which allow to create pervasive applications, and most of them are designed for a specific application field.

Our research group is working on a middleware for pervasive computing which fills this gap between operation system and pervasive applications. The middleware provides a uniform and general infrastructure. It helps to handle the heterogeneity of the physical world and provides a standardized homogenous interface to applications. Several publications are already available on different aspects of this middleware such as uMove in [BH09] and Human Computer Interaction (HCI) in [BLLH10].

## 1.1. Motivation

One of the missing pieces of the pervasive middleware is an uniform coordination language, which is dedicated for pervasive computing. The motivation of this thesis was to find and develop a model which fits into our group's middleware. Based on the previous work of Gelernter and Carriero [GC92], Malone and Crowston [MC94] and of our research group [Sch01] we show how coordination for pervasive systems can be realized. We chose a holistic approach to express that the world is made up of interacting entities building a whole. This includes the aspects of the physical and the virtual structure, the activities and the relative point of view. The activity describes any state changes over a certain time, where as the relative point of view expresses the relativity and subjectivity of observers.

## 1.2. Goals

The goal of this thesis is to show how coordination can be integrated into a middleware for pervasive computing. The latest research results on the holistic approach of our middleware must be considered. This especially includes the relativity of the point of view. Instead of directly creating a coordination model for pervasive computing we intend to define a generic model first. This model should be valid for all scientific fields such as social, economic and computer studies. We think that the integration of pervasive applications into the daily life of humans can be done more smoothly if our framework respects the generic model because it will reflect the natural behavior and follow the general rules of the real world. Finally a coordination library for Java will be defined. The goal is to have a tiny but extensible application interface used to define the coordination part of an application.

## 1.3. Outline

This document is structured as follows: Chapter 2 outlines the state of the art. It gives an overview of how coordination is defined in theory and how it influences the performance of a system. A physical meta model is introduced using a holistic approach. Other selected topics like context awareness, activity theory and dependency are briefly explained.

In chapter 3 we propose generic coordination model which can be applied to several scientific fields (figure 1.1). The model helps to explain and define the different parts of a system at a very high level of abstraction. The model includes physical and virtual structures, the definition of tasks (activities) and the relativity of observation. We show how the physical and virtual entities are defined and related to each other. Special attention is given to the definition of tasks and activities. The chapter also explains the relativity of observers, their perception and point of view and includes relative entity classification based on a simple taxonomy. At the end communication is treated. Communication plays a major role in coordination and can be seen as a main activity to exchange information between entities. We show how communication can be classified using simple criteria.

In chapter 4 we show how the generic coordination model is adapted for computing systems and specially for pervasive systems. Only a few selected topics of computer science are treated in this chapter like hardware composition, networking and programming languages. An immense amount of aspects were left out since from our point of view they were not important for pervasive computing. Therefore we call this model *the coordination model for pervasive computing*. In contrast to the traditional models, where the computer was seen as an independent and delimited tool, we put the computer system into a different position. In our model the pervasive system is an observer of the real world rather than being just a tool used for computation. Humans are also treated differently in our model. The user as well as the system engineer influence a system. We state that a human is a central entity to be considered and has to be integrated into the system.

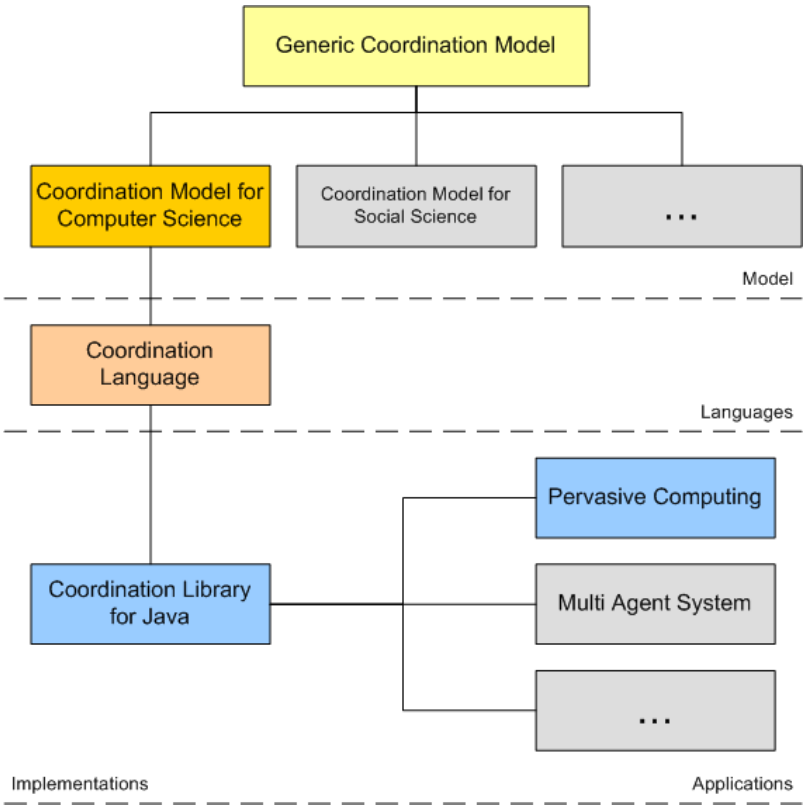



Figure 1.1.: Overview of the coordination model and its implementations

Based on a pervasive coordination model a coordination language is defined in chapter 5. The language is split into two main parts: 1) the representation of the world using entities, their contextual information and relations and 2) the coordination component managing the communication between the entities using rules. The first part is just briefly explained since it is part of the uMove framework and described in [BH09]. The second part contains pure coordination issues and is explained together with the communication channels and rule architecture.

Using a concrete case in chapter 6 we show how coordination of pervasive computing works. The case study shows how a mobile device can be tracked by a smart environment. As a smart environment we use a server application providing pervasive services and a WiFi network to provide the services in a physical environment. Depending on the human's context and activity different services are provided to them.

Some conclusions and possible future work complete this paper.

# 2
# State of the Art



## 2.1. Introduction

This chapter outlines related work and theories which our coordination model is based on. Coordination concerns *who is doing what* and *who is getting which information*. There are many definitions and descriptions of "*coordination*". One of the most commonly used definitions is given by Malone and Crowston, in [MC94]:

**Definition 2.1** *Coordination is managing dependencies between activities.*

In a static world having no necessity for change no coordination will take place. But as soon as someone has needs it will require an activity to satisfy them. For instance two atoms react with each other in order to be in a more stable state. The need is a stable state and the activity is a chemical reaction. All atoms are in concurrency with each other and some implicit coordination process takes care of it, for instance *"first come - first served"*.

A more technical view about coordination was given by Gelernter and Carriero in [GC92]. Mainly, any system can be split into two parts, *the computation* and *the coordination*. Coordination can be seen as the glue fitting computation parts together. It is a very important aspect of the system but mostly unseen and unrecognized by users, as long as it does what it is supposed to do. We mostly notice coordination if it does not work properly. For instance if we get the letters of our neighbor or if we wait at the train station for a delayed train. Gelernter and Carriero also state that a coordination language

is orthogonal to a computing language, meaning it extends the computational language by managing interdependencies between different computational parts. They proposed a language called LINDA which is probably the most prominent coordination language. It is based on a *tuple space abstraction* coordination model (blackboard communication).

Michael Schumacher and Oliver Krone proposed a different coordination model for multi agent systems (MAS) [Sch01], [KCDH98]. This model already defined agents, ports and communication patterns. STL and STL++ were introduced as coordination languages based on C and C++ respectively. Sergio Maffioletti in [MKMH04] defined a coordination model to describe and manage the dynamics of an environment called UBIDEV. This framework allows to model an environment in terms of resources and services manipulating those resources. The framework is structured into several layers. The lower layers have to deal with the heterogeneity of physical entities, where as the top layers provide a homogenous API to applications.

The coordination model proposed in this thesis is based on a generic coordination model, XCM [TCH05]. In XCM everything is an entity. The entities are composed of entities. The root entity is called *the universe*, and an undecomposable entity is called *an atom*. Ports are dedicated to the communication between entities. The ports are the fundamental mechanism in XCM to coordinate entities.

The rest of this chapter addresses important theories and models which are connected to our holistic coordination model, such as the physical model, activities and dependencies.

## 2.2. Physical Model

Our coordination model is based on the physical world. Therefore it is necessary to lean on a physical model. The most used physical model nowadays is based on the mechanist paradigm which states that the physical world is made of material objects moving in space and time. Their movements are based on physical laws. As explained by Eric Schwarz in [Sch02] this model is insufficient to describe partially autonomous biological and social entities. Systems like social, political, ecological and other living systems are hard to situate in the mechanist framework. He proposes a new framework which is more general than the Cartesian-Newtonian mechanist approach. This framework should help to understand real life and complex systems with self-organizing and non-linear behavior. It is neither dualistic nor deterministic, but includes the holistic nature of living systems.

The framework of Eric Schwarz [Sch02] does not only take the material structures (matter and engery) into account, but also considers an immanent network of virtual relations. This relation network affects the future states of the system. The concepts of whole, of existence or of being are part of the framework and denote that no single parts of the whole can be analyzed in order to understand the whole.

The simplest configuration of things is a system made of two components in relation. It either represents two interacting entities or a subject observed by an observer (figure 2.1 left side). The model is made up of three main layers. The first layer is called the *energy plane*. It corresponds to the world of physics and describes the physical world of things. The second layer contains *the cybernetical world of the potential relations immanent in*

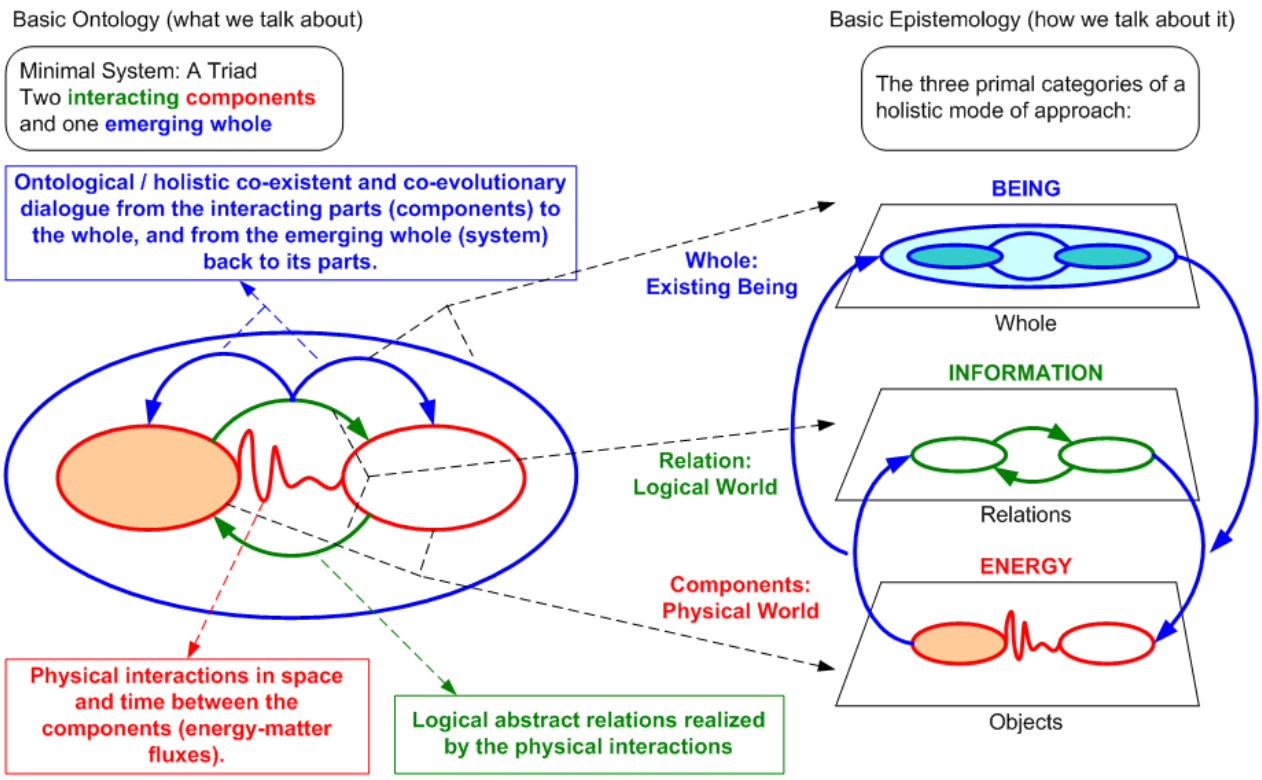



Figure 2.1.: Physical model proposed by Eric Schwarz.

*the system* [Sch02]. The last layer is the plane of being which contains the system as an existing whole.

## 2.3. Activity and Motivation

Activity can be understood as a list of actions changing a particular part of the world in order to achieve a goal. If we take a picture of the world we would not be able to detect any activities. We have to take a second or third picture later to detect what changed since the first snapshot. This example shows that activities occur in an environment over time and define the evolution of the system.

But having no idea about the motivations of the entities changing the world, it is hard to analyze their activities and almost impossible to coordinate them. Activity and motivation go along with each other. Kuutti showed how activities and goals are connected [Kuu95] (figure 2.2). He used the word *motive* rather than *goal* to express why an activity happens, and we will use the same name in future explanations of our model. He showed that activities become decomposed into actions, and actions into operations. The difference between actions and operations is merely that actions are performed consciously where as operations are done unconsciously. We will use the word action for both types in order to simplify the problem.

According to activity theory [Kuu95],[Nar95] an activity always contains various artifacts such as instruments, signs, methods and laws. Each artifact plays a mediating role

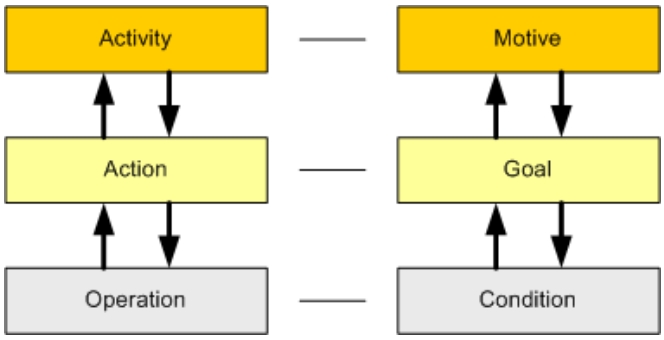



Figure 2.2.: Kuutti defines activities as a chain of actions.

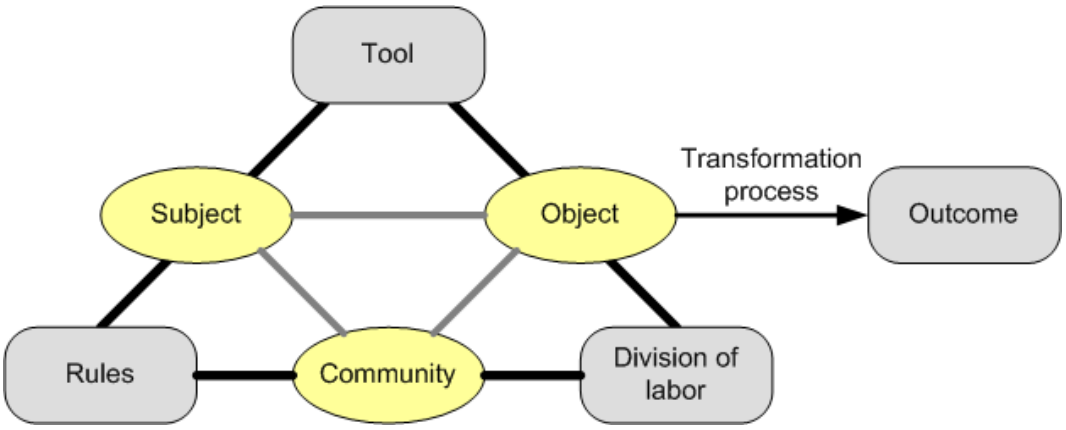



Figure 2.3.: Structure of an activity

within the activity. Figure 2.3 shows the structure of an activity and the mediating roles of artifacts.

The relationship between object and subject is mediated by tools. The tools are used by the subject to transform the object into an outcome. A third main component, called community, is added to the activity framework. The community shares the same object with the subject. The new relation between subject and community is mediated by rules. They also include norms, conventions and social relations. The other relationship called division of labor refers to the organization of the community which is used to transform the object into the outcome.

## 2.4. Dependencies

As recommended by Kevin Crowston in [3] the elements of Malone and Crowston's framework [MC94], namely motives, activities, actors and resources can be grouped into 2 categories:

- Entities defining the world. These include the actors, the environment (location) and the resources used in the activities.
- Tasks which include motives (desired states of the word) and the activities (list of action to be performed to achieve a particular state)

Between entities and tasks there exist various dependencies which are managed by coordination. A dependency is always dynamic and can change over time. Therefore all dependencies are regarded as temporal dependencies. The dependencies can be split into different types:

- ***Task dependency***: describes the dependency between resources and activities using them. For instance two mechanics work together on a car. The common motivation is to fix the problem whereas the activities of the two mechanics might be different from each other. All entities participating in an activity (mechanics, tools and car) depend on each other.
- ***Social dependency***: this dependency describes how entities are related to each other. For instance the neighborhood describes a common social dependency.
- ***Spatial dependency***: the spatial dependency represents the dependency within the physical world. The behavior of a hiker depends on whether he walks inside a valley or climbs up a steep hill.

### 2.4.1. Coordination Process Handling Task Dependencies

The main problem of task dependency is to solve the problem of resource assignment.

**Definition 2.2** *A resource is any physical or virtual entity of limited availability, which can be used to accomplish an activity.*

As proposed by Malone and Crowston in [MC94] different kinds of task dependencies can be characterized. For each dependency one or many associated coordination processes can be identified for handling the dependencies. The following list gives just a short overview of the most common task dependencies (figure 2.4):

- ***Flow***: occurs if an activity produces a resource which is needed by another activity (producer-consumer problem). Possible coordination processes are notification, sequencing and tracking.
- ***Sharing***: occurs if two or more activities need the same resource. First come - first served, priority order or bidding are coordination processes to manage this dependency.
- ***Fit***: two or more activities produce the same resource. Goal selection and task decomposition are possible coordination processes to manage the fit dependency.

After identifying the type of resource needed to perform the activity, the resource has to be found and assigned to the task. Kevin Crowston [3] identified the following steps in order to assign resources to tasks:

1. Identifying the required resource
2. Identifying what resources are available
3. Choosing the particular set of resources
4. Assigning the resource to the task

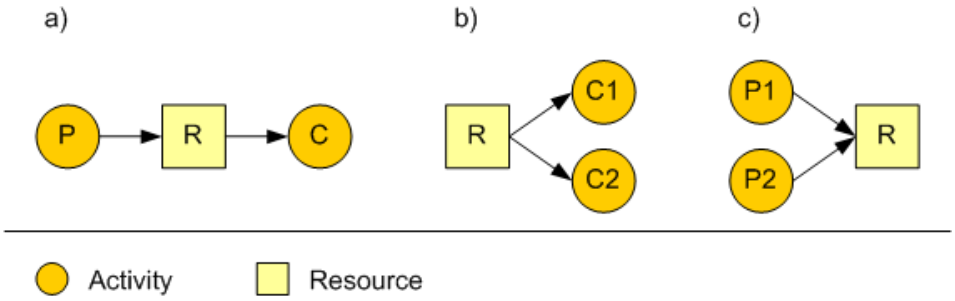



Figure 2.4.: a) flow- b) share- and c) fit-coordination process

### 2.4.2. Social Dependency

Social dependency describes how entities are related to each other. According to Giddens structural theory [Gid84] the dependence of humans on a social structure affects their activity. The social structure is defined as a composition of rules and resources (humans, artifacts). He argues that it is created as a medium for practical activity. But the activity also changes and evolves the social structure. Social structure and the activity depend on each other but also constrain each other.

In our generic coordination model we split Gidden's social structure into the following components:

- ***Social Law***: rules which govern the interaction and activities of the entities participating in the social structure
- ***Social Dependency***: relation of the entities forming the social structure

### 2.4.3. Spatial Dependency

Spatial dependency describes where an entity exists within the physical world. Each entity is placed within an environment. The environment will more or less influence the context of the entity.

# 3

# Generic Coordination Model

## 3.1. Introduction

The generic coordination model allows for the definition of a model for coordination at a very abstract level and therefore can be used in various scientific fields, like computer science, social and economic studies. It is based on previous work by Amine Tafat [TCH05]. A few aspects have changed since the last publication of the former model. The main difference between the model described in [TCH05] and the one we propose in this thesis is the new structure of activities and tasks. This model also contains the concepts of observers and their relative perception.

## 3.2. Modeling Coordination

Before discussing coordination we have to clarify how a system is defined. There exist different types of systems such as inert or dead, living or evolutionary, closed or open. The coordination model holds for living and evolutionary systems but not for static dead systems. For static dead systems no coordination is needed, since coordination manages dependencies between activities and static dead systems do not have them.

The model holds for open and closed systems. Within an open system external influences and dependencies must be considered when coordinating the entities of the system, where as in closed systems no external dependencies exist. To model coordination for an open system we propose to enlarge the open system to include all influencing components in order to achieve a closed system at the end. This will help to avoid the situations where influences from outside are not recognized within the coordination. A more precise definition about what a system is, what it is made of and how it is treated within coordination is given in chapter 3.9. The explaination of the observers and their relative perception are also presented in chapter 3.9.

As proposed by Michael Schumacher in [Sch01] we distinguish between two main types of dependencies: subjective and objective dependencies. Since coordination is managing those dependencies, we split coordination into two groups.

- ***Objective Coordination***: manages the objective dependencies. The main consideration is the organization of the world and the communication between the entities of the world.
- ***Subjective Coordination***: manages the subjective dependencies, which result from the individual point of view of each entity (observer of a system). Subjective coordination can be split into two sub groups:
  - *Explicit Subjective Coordination*: concentrates on coordination techniques which explicitly manage dependencies such as negotiation, planning or organization techniques.
  - *Implicit Subjective Coordination*: concentrates on coordination patterns which implicitly coordinate the dependencies between entities. Each entity is not aware of the coordination but fulfills the tasks following simple rules (techniques such as stigmergic coordination belong to implicit subjective coordination).

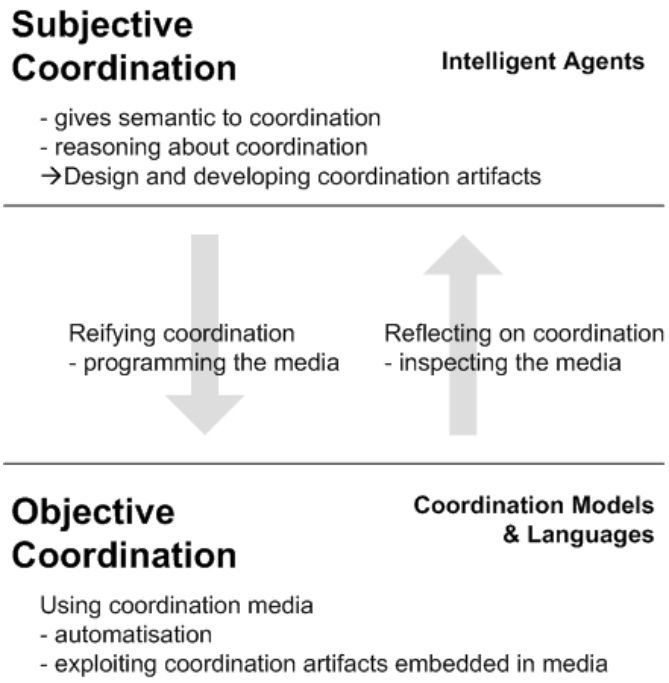



Figure 3.1.: Shows how subjective and objective coordination are linked together.

There exist some dynamics between objective and subjective coordination as A. Omicini described in [ORRV03] (figure 3.1).

- ***Reflecting on coordination***: intelligent agents analyze and measure how well coordination is performed. On the level of subjective coordination the agents try to understand the problems and search for possible solutions.
- ***Reifying coordination***[1]: once a solution is available the agents modify the objective coordination by changing the rules and the structure.

### 3.2.1. Objective Coordination

The first part of the model describes mainly the objective dependencies.

- ***Spatial dependency***: where is the entity located?
- ***Social dependency***: who or what does the entity belong to?
- ***Task dependency***: where is the entity used or what is the entity doing?

These dependencies lead to the objective part of the generic coordination model, which is organized in the physical, virtual and task structure (figure 3.2). The physical structure of entities helps to manage their spatial dependency. It gives a structure to the world of entities and puts them into physical relation. The virtual structure is mainly needed to manage the social dependency. Complex social networks can be modeled using this structure. But it also helps to manage the spatial dependency using virtual places. Finally the task dependency is managed by the task structure which brings the activities and the resources into relation. Each relationship is temporal, because the relation might change during the lifetime of an entity.

[1]Reification: making a data model for a previously abstract concept

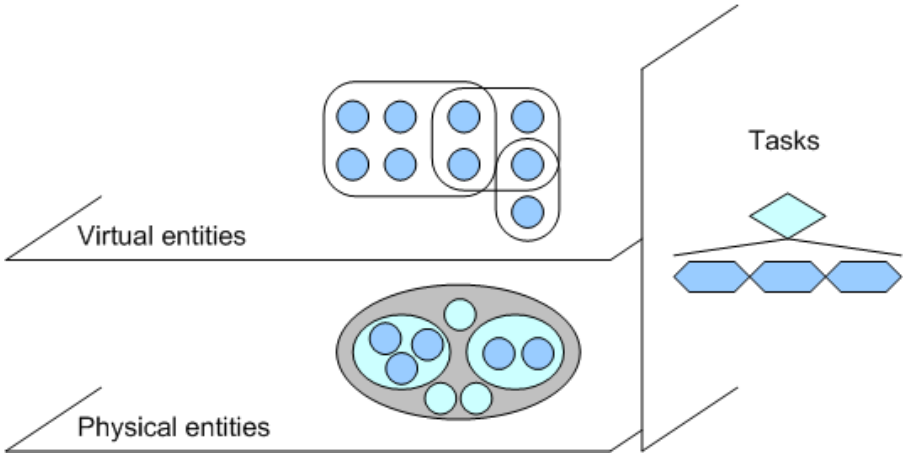



Figure 3.2.: The different components of the coordination model

### 3.2.2. Subjective Coordination

Subjective coordination is a process by which the entity behaves according to some internal logic in order to ensure that the community acts in a consistent manner. This can be performed explicitly, meaning the entity is aware of the coordination process or implicitly, just following some simple rules. For instance some internal algorithm might lead to swarm intelligence known from the study of bees and ants.

The subjective part of the coordination model helps to manage subjective coordination which includes the following aspects

- Negotiating, planning and organizing tasks
- Implicitly marking good solutions which are followed by others (stigmergy)
- Modifiying the social rules in order to optimize the interaction between entities

## 3.3. Definition of an Entity

The key element of the coordination model is the notation of an entity. An entity describes any object which belongs in the system.

**Definition 3.1** *An entity is anything physical or virtual that has a distinguished, separate existence.*

The distinction between physical and virtual entities is:

- ***Physical entity***: is an object that exists throughout a particular trajectory in space over a particular duration of time, such as a planet, human etc.
- ***Virtual entity***: an object that exists in the domain of the immaterial world and does not need any place or time to exist, for instance a community, a social network. This includes all abstract and logical objects.

A decomposable entity is called *a composition* which means it has a structure. An entity can be composed of other entities. For instance a building is composed of floors and

| | ***physical entity*** | ***virtual entity*** | ***task*** |
|---|---|---|---|
| ***physical entity*** | physical / logical relation | logical relation | logical relation |
| ***virtual entity*** | logical relation | virtual/logical relation | logical relation |
| ***task*** | logical relation | logical relation | logical relation |

Table 3.1.: Entity relation types

rooms. The largest composition is called *the universe*. No entities can exist outside of the universe. The universe is the border of a closed system. It is possible to create new entities or to destroy entities within an universe. Entities are also capable of moving from one composition to another, modifying their spatial dependency. The smallest entity is called *an atom*. Atomic entities can not be decomposed.

## 3.4. Definition of a Relation

Two or more entities might share some common properties. We call the sharing *a relation*. For instance if two humans share the same last name, they are related by that name.

A dependency is a special type of relation between entities. Dependency means that the relation influences all entities which are part of the relation. In the example of two people sharing the same last name no dependency exists. They might not know each other. But if they belong to the same family, things are different. They might meet and exchange information, share their history. Dependency is much stronger than a relation.

A relation can exist between entities and tasks. They are in physical, virtual or logical-relations as shown in table 3.1. Physical relations exist only between physical entities and are managed by the physical structure only. Logical relations can exist between physical entities, virtual entities or even between tasks.

As a case, the virtual relation can occur only between virtual entities. If the virtual structure (figure 3.2) represents parts of the physical world, the virtual entities of this structure are also virtually related to each other. Logical relations are used to express mathematical dependencies between entities, whereas virtual relations also represent any other kinds of relations.

## 3.5. Physical Entity Structure and Relations

The physical structure deals with physical entities and their physical relations. As defined in [BH09] there are different types of spacial relations, such as inside, next-to or joined.

- ***Inside***: the inside-relation is the main physical relation. An entity can be only inside one parent at the time.
- ***Next-To***: two entities are close to each other or within a defined range

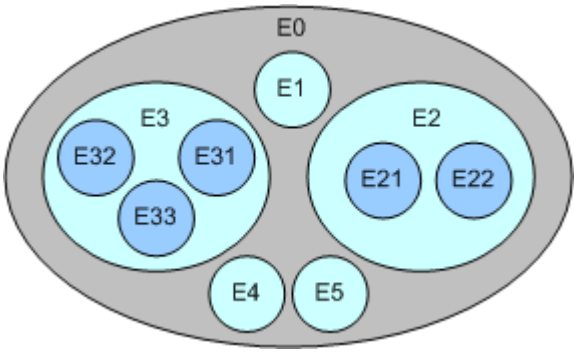



Figure 3.3.: Representation of a physical entity structure.

- ***Joined***: two entities are joined if they are in the same place and move in the same direction

The physical structure helps to localize entities and to compute their environmental context. For instance, if the temperature changes within a room the context will change for all humans remaining there. The environment of the entity concerns the external context, like room temperature and humidity. It also includes the surroundings (other entities).

**Definition 3.2** *The environment is the external context and surrounding of an entity.*

### 3.5.1. Inside Relation

Physical entities can be grouped into larger compositions. A physical entity can exist only in one physical composition at the same time (as we follow the paradigm of classical physics). This has a serious impact on how physical structure is modeled. For instance overlapping of physical entities is not possible (figure 3.3).

An entity which can be decomposed is called a physical location and we use the notation of a rectangle to express it. We use the notation of a circle to express an atomic entity (figure 3.4a).

The inside-relation can be represented using a tree, called *an entity tree*. The entity tree represents the exact same structure and is equivalent to the previous described block notation. Compositions are represented by any nodes containing child nodes, whereas atoms are leaf nodes of the tree (figure 3.4b).

### 3.5.2. Next-To Relation

The next-to relation is defined by a radius around the entity. If another entity stays in the same location and is within this range it has a next-to relationship (in [2]) with the first entity.

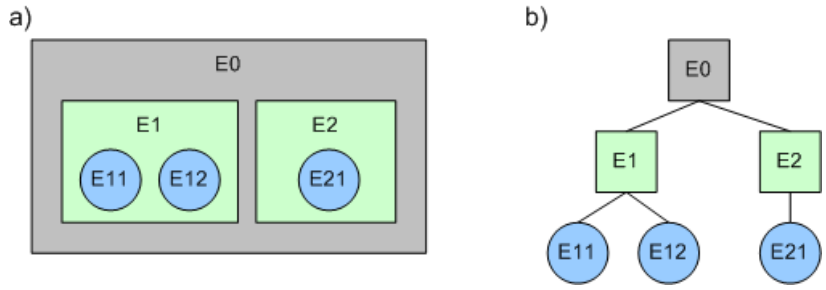



Figure 3.4.: Entities are structured within the universe E0. The physical structure can be represented as a block notation (left) or as an entity tree (right).

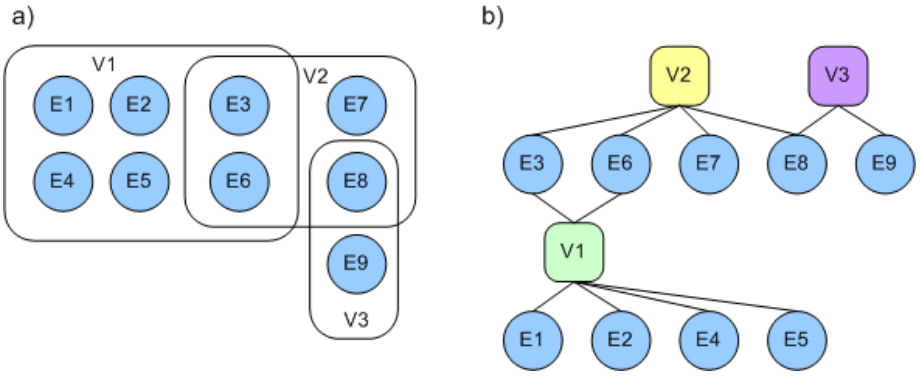



Figure 3.5.: a) A virtual structure can overlap with other virtual structures. b) virtual structure can also be modeled as a relation graph between virtual structures and entities.

### 3.5.3. Joined Relation

The joined relation is stronger than the next-to relation. The two entities in a joined-relation fulfill the next-to relation, but they also move together in the same direction and with the same speed. The joined relation is dynamic. For instance, if two persons travel in the same car, they have a joined relation to each other. But as soon as one person leaves the car, the relation is broken.

## 3.6. Virtual Entity Structure

The virtual entity structure contains virtual and physical entities. It represents all non physical entity relations, for instance the hierarchy of a company or a neighborhood of residents. We use the notation of rounded rectangles to express virtual structures and entities (figure 3.5).

There exist many different types of virtual structures. For instance we have:

- ***Community***: it is a set of entities. The structure of a community is flat.
- ***Organization***: an organization is a complex structure like a company hierarchy or a political system represented for instance by a tree.

- ***Network***: the structure of a network is very generic. Typical instances of network structures are social networks, neuronal networks and computer networks. It is a graph.
- ***Virtual Location***: a virtual location is a special virtual structure. A physical location can be split into several virtual locations overlapping each other. For instance a room can be split into 3 virtual locations: the back, the center and the front. There are no sharp borders between them. Statements like *"more in the back"* or *"almost in the center"* show that virtual locations have fuzzy borders.

The virtual structure helps to define social dependencies between entities. Who is associated with whom can clearly help to coordinate the communication of entities. For instance, a neighborhood automatically receives a message if a cat has disappeared from it.

## 3.7. Task Structure

It is not enough to stay with the physical structure (who is where) and with the virtual structure (who is associated to whom) in order to model optimal coordination. Since tasks have their own structure and describe how the system structure evolves over time, we added the notation of tasks to the generic model as a separate structure. In general, a task can be understood as a transformation of the system over time. Each task describes how this transformation is done, what the motivation is and what resources are needed.

The task structure is needed in the generic coordination model to manage the task dependencies of entities.

### 3.7.1. Decomposing Tasks

A task can be decomposed into sub-tasks. Sub-tasks can run at the same time (in parallel). The main task ends if all sub-tasks have ended. For instance the main task is *"building a car"*. It can be decomposed into *"building the motor"*, *"building the seats"*, *"assembling"* seats and motor into the car etc. (figure 3.6).

Some of those tasks can be done in parallel such as *"building seats"* and *"building motor"*. Some can start after others are done, such as putting seats and motor into the prepared car. The management of tasks is naturally defined by the used resources and the fact that some tasks need resources produced by other tasks. Nevertheless the task management can be very complex and lead to deadlocks, starvation and other problems.

### 3.7.2. Motivation

Every task has its motivation. There exists no task without a motivation. Motivation is not always obvious. Sometimes it is hidden and only known to the entities performing the task. If a task is decomposed into several sub tasks, each sub task has a motivation which follows the main motivation.

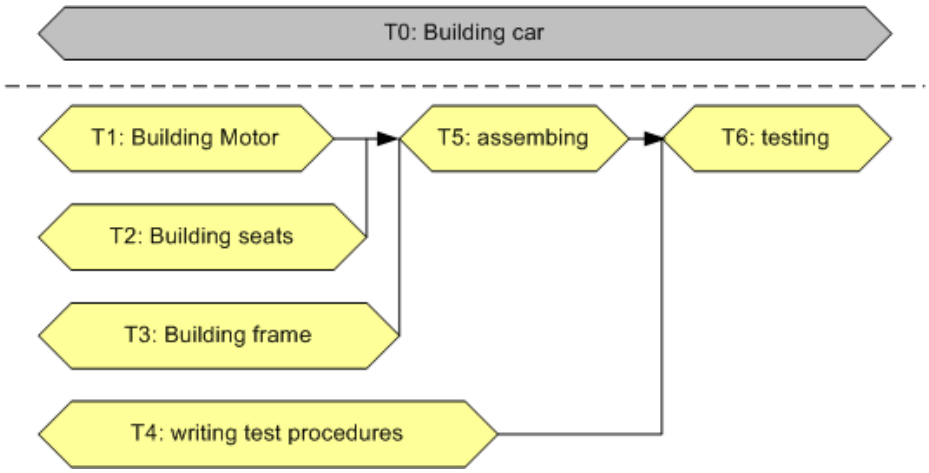



Figure 3.6.: The example shows how a task can be decomposed into several sub-tasks.

### 3.7.3. Activity

An atomic task is called an activity. An activity can contain several actions which are executed sequentially (figure 3.7). For instance the activity *"writing a text"* contains *"take pen"* and *"write title"* as actions.

Each action has its own goal. It is some kind of post-condition of an action. The goal does not change during the action whereas the motive of a task can change and evolve over time. If a motive changes, it might happen that the goals of executed actions don't fit the motive anymore. Then the actions are useless or interfering in the current activity. For instance if a team manager changes his opinion about an ongoing process, it is expected that some work done by the team will become obsolete. An activity has always an outcome. The following list shows some of the possible outcomes:

- Creation of new entities
- Removal of old entities
- Internal state changes of entities
- Starting of new tasks
- Subjective coordination (This includes internal optimizing of coordination processes)

### 3.7.4. Role

Activity uses different entities during its execution. This generates several relations between entities. A relation is called *a role* and the entity involved in the activity is called *a resource*. Each resource participating in an activity plays a specific role. The following list is not exhaustive but gives an overview of the most important roles that an entity might play:

- ***Actor***: role of the entity who performs the activity (figure 3.7)
- ***Place***: the activity is always done within a place. In this case the entity is used as an environment.

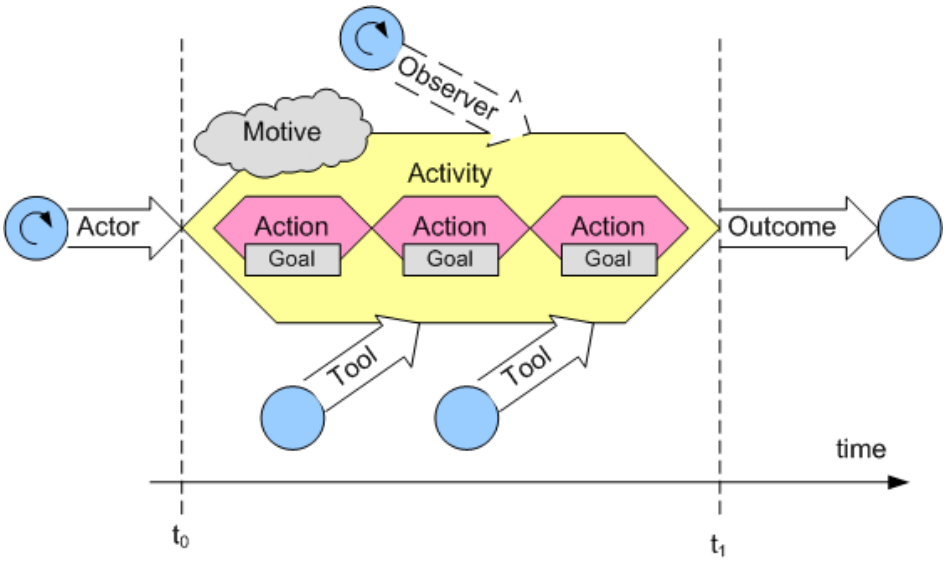



Figure 3.7.: Shows the composition of an activity and several relations to entities taking part in this activity.

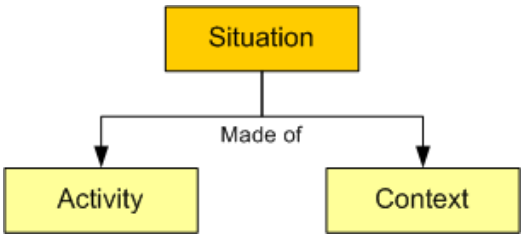



Figure 3.8.: Model of situation

- ***Tool***: an actor uses a tool to perform the activity
- ***Observer***: the observer is looking at the entities carrying out an activity. He evaluates the current situation of the setup and reacts upon it, for instance using subjective coordination.
- ***Port***: a port is an entity used to receive and send information to other entities
- ***Not-Classified***: all entities which are not classified or don't participate in the activity

### 3.7.5. Situation

Finally there is the definition of a situation. Situations are related to the notion of context but are situated on a higher semantic level as proposed by S. W. Loke in [Lok04]. A situation also includes activities. As proposed by Y. Li and J. Landay in [LL08] *an activity evolves every time it is carried out in a particular situation.* Here the situation is understood as a set of activities or tasks performed under certain circumstances. Our model integrates those two visions [BH09] (figure 3.8).

In general, situations are used to evaluate an activity within the context of entities. Situation analysis helps to check if the rules are followed by entities.

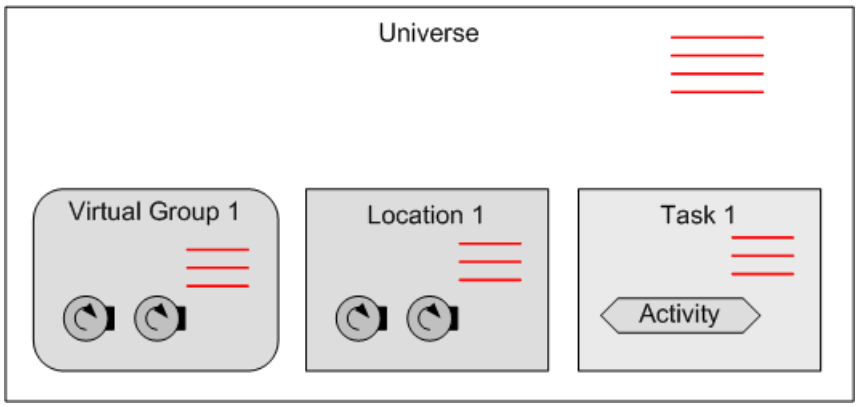



Figure 3.9.: A law defined in the universe affects the physical, virtual and task structure.

**Definition 3.3** *A situation is an activity performed within a context [BH09].*

It is temporal and changes immediately if the context or the activity changes. The notation is a very important construct for objective coordination. Together with the social rules it can be used as an input for the coordination management, since it includes all objective coordination elements such as contextual information (spatial and social dependencies) and the activity (task dependency).

## 3.8. Physical and Social Laws

Laws are used to guide and control the behavior of a system. We distinguish between physical and social laws. A law is made of a set of rules. Each rule treats a specific part of the law.

A physical law is static and can only be changed by the creator of the closed system. Laws mostly describe physical behavior in the physical world. No entities living within the physical world (like humans) can change them. Also laws hold equally for all entities and exceptions are not possible.

In contrast, social laws are changeable by the entities of the system. For instance the laws of a country are changeable by its government and there might be many exceptions for certain situations.

**Definition 3.4** *A Social Law defines how entities interact with each other and how social structures are created.*

A social law can be understood as a guideline for activities and interaction between entities. It can be defined for physical and virtual structures as well as for tasks (figure 3.9). Sub-entities always inherit social laws from their parent. A social law can be modified or overwritten by the sub-entity. For instance the laws of a country also applies to its states, but a state might extend some of the laws.

Each social law might also declare what must be done if an entity violates it. The so called *penalty actions* are executed by entities responsible for keeping the system stable. The law has to be checked during the coordination management and governs the objective coordination. After each check, a list of executable actions is produced. For instance, in a democracy the judiciary checks if a human violates a law. After the checking some actions might be taken. A different authority takes care of this, the executive authority. Social law can evolve over time. Some entities might be authorized to modify or add new rules in order to optimize the coordination and the interaction behavior (subjective coordination).

## 3.9. Observer

So far we described how the system is made. For closed systems using only objective coordination this might work well. But for open systems or subjective coordination, we have to change our point of view. For instance, the system engineer of a pervasive computing system is part of the system itself. He acts as an observer if he is maintaining the system, but he is also an actor and is observed by the system.

The view of an observer onto the system is not absolute, it is relative. He has a relative perception onto the part of the world he observes. His point of view is reported to other entities and used to coordinate the system. We think that it might lead to undesirable behavior, especially in coordination, if the relativity is ignored. This section explains how the relative point of view is treated in our model and influences the system.

### 3.9.1. Point of View

The perception of structures, activities and roles is done by observers. As described in KUI[2] by Pascal Bruegger in [BH09] entities are observed by an observer from a specific point of view. We consider a view as a filtered and a specific representation of the world, since it contains only part of this world.

To have a view the observer must first chose a viewer and a reference in time and space. The viewer is a physical entity used by the observer to receive, preprocess and filter the information. A viewer can be seen as a port or as a tool (e.g glasses, microscope, ears and eyes). On the other hand the view is a virtual entity. The properties of the viewer and the chosen physical references influence the view:

- ***Focus***: the focus is a property of the viewer, which includes the direction, the range at which the observation starts and the level of depth to which the observation goes. For instance an astronomer uses a telescope as a viewer to focus on a part of the universe. The range goes up to the border of the universe. But a biologist observes the world differently. He might focus on cells using a microscope. The range will go down to the level of chromosomes.
- ***Location***: the location is part of the chosen physical reference. It influences what an observer might see in his view. According to Albert Einstein's general relativity theory, even the time depends on the location (field of gravity).

[2]Kinetic User Interface

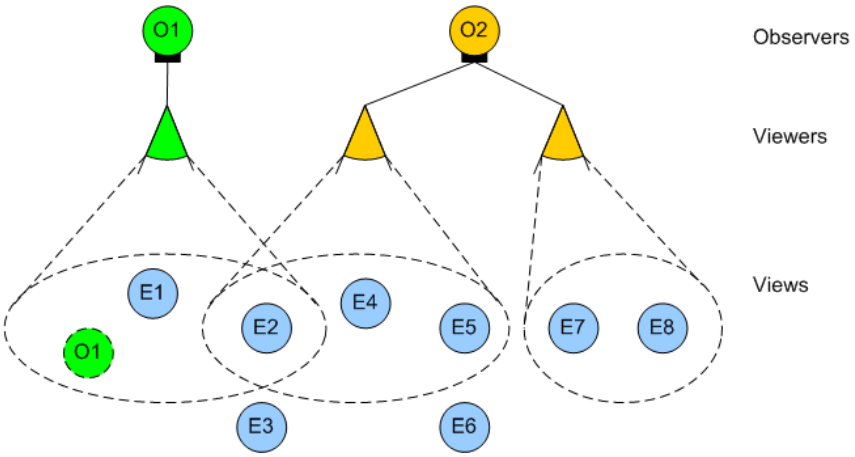



Figure 3.10.: Observers are using views to observe entities

- ***Motion***: the observer gets a different view of a situation if he moves. The motion depends on the chosen reference. For instance if an observer is traveling in a train, to him it looks like the landscape is moving. The observer knows only by experience and knowledge that he is moving, not the landscape. And again speed influences the time clock of the observer according to the theory of general relativity.

This shows that the view always gives a relative perception to the observer. He tries to interpret the information he collects through the viewers used. The interpretation of the observer depends on several internal properties of the observer such as intelligence, knowledge and experience.

An observer can have many different views at the same time (figure 3.10). The eyes of a human provide the visual part of the world, whereas the ears might provide a completly different view of the world. Both are interpreted by the human brain and together form *a picture* of physical world. Some viewers allow to observe the observer himself in a reflective manner. A typical example are mirrors, used by humans to look at themselves.

An observer can detect many different aspects of a system using a viewer. The following list explains the most important ones for coordination:

- ***Relative structure***: Each observer recognizes atoms in their environment (other entities) using a different level of granularity.
- ***Relative entity classification***: The observer categorizes the observed entities according to entity classes and kinds into taxonomy.

### 3.9.2. Relative Structure

The perception of the entity structure depends mainly on the focus which includes the direction, the level and the range of the observation (figure 3.11).

The observer is able to sense atoms and the environment.Atoms are undecomposable for the observer. The observer might recognize them as compositions using another view. For instance, in traditional chemistry neutrons, protons and electrons are atomic entities.

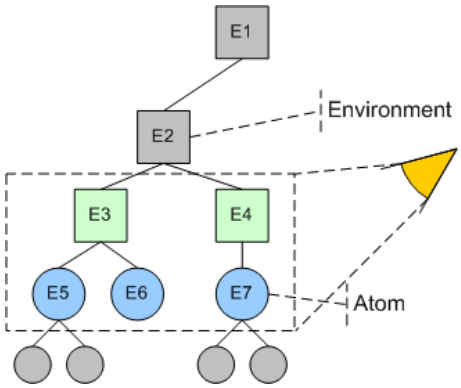



Figure 3.11.: Shows the observation of a structure.

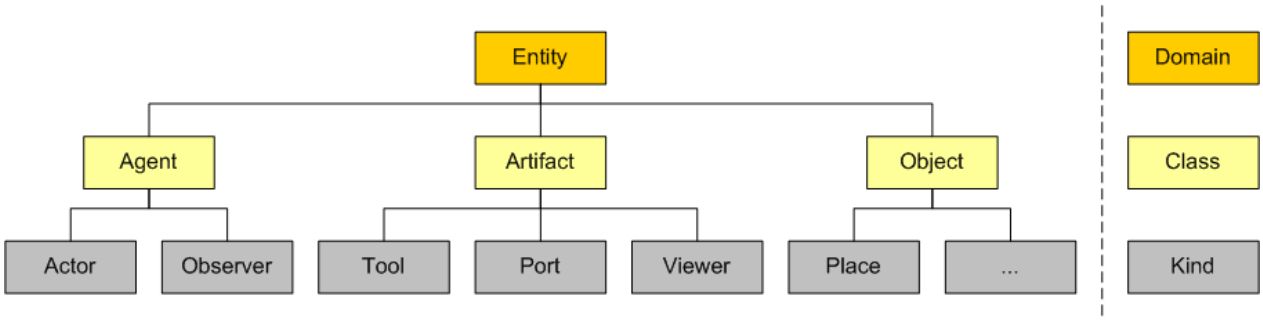



Figure 3.12.: Shows the relative classification of entities using the taxonomy Domain-Class-Kind

For physicists at CERN, these nuclear particles are just compositions. This awareness leads to the following list of structural elements:

- ***Relative atom***: entity recognized as an atom
- ***Relative composition***: entity which can be decomposed into sub-entities
- ***Relative environment***: the observer observes entities within this entity. The environment is outside of the observation but it influences its sub-entities.

Observation can be done on physical and virtual structures. Computers are able to observe virtual structures whereas humans normally observe parts of the physical world.

### 3.9.3. Relative Entity Classification

The classification depends on the point of view and internal knowledge of an observer. An observer is capable of classifying entities into classes and kinds. The following taxonomy is not exhaustive and can be easily extended with new kinds for any specific model (figure 3.12).

- ***Agent***: entity able to sense the environment and act upon it. It is also capable of changing the environment autonomously [Sch01].
  - *Actor*: an agent performing an activity
  - *Observer*: an agent observing other entities and evaluating their situation

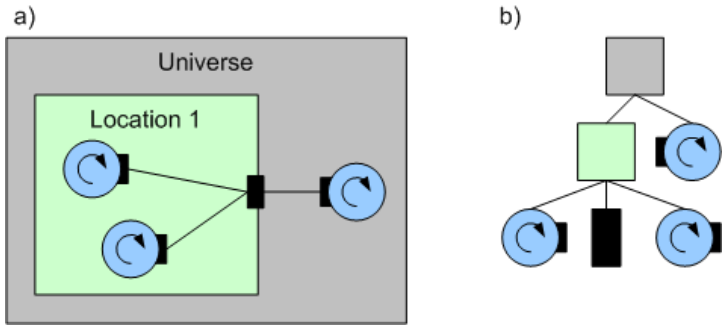



Figure 3.13.: a) 3 agents connected with each other through ports. b) shown in the entity tree

- ***Artifact***: entity which is made or changed by an agent. It is regarded as dead or passive entity, mostly used by agents to perform an activity.
  - *Tool*: an artifact used to by an actor to perform the activity
  - *Port*: an artifact used to exchange information
  - *Viewer*: an artifact used by an observer to look at other entities
- ***Object***: any unspecified entity, like natural resources
  - *Place*: an entity used as a place to perform an activity

Entity classification does not distinguish between virtual and physical entity. This is defined in the entity ontology.

### Agent

An agent has the capacity to modify its internal state or its relations to the surrounding environment. Commonly the agent senses the environment using sensors and reacts depending on the sensed information using its effectors. For instance a human uses the senses: sight, hearing, touch, taste and smell to get information from the environment. Depending on the result of the processed information the human can act within the environment using several effectors such as limbs, fingers, head and speech. We use the notation of a circle with a rotating arrow to express agents (figure 3.13).

### Port

A port is special artifact used for communication. An entity can send or receive information through a port. A port is represented by a black bar (figure 3.14).

A port can be modeled as a sub-entity of any other entity. It is always an atom. An entity can receive information through a port only if the port is coupled to that entity (figure 3.14b). If uncoupled ports can be coupled dynamically. We call these ports *removable ports*, whereas static coupled ports are *irremovable ports* [TCH05]. Section 3.10 decribes ports in more detail.

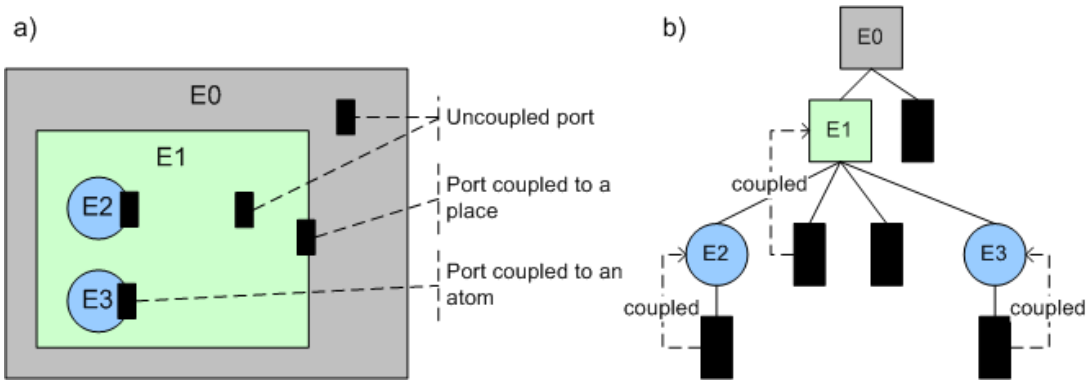



Figure 3.14.: a) Shows the notation of ports in block notation b) or tree notation

### 3.9.4. Relativity of a System

How a system is structured and who belongs to the system depends on the observer point of view. It exists only one closed system, which is called the universe. By definition there no entities exist outside of the universe and therefore there is no interdependence with the outside of the universe. All other systems are open systems, even if the interdependence is minimal (there is at least one spatial temporal relation to other entities).

**Definition 3.5** *A system contains two or more entities which are in relation to each other. It has a clear border which defines which entities belong to the system.*

A entity can be considered as a system if it contains other entities. For instance a human can be considered as a complex system with many sub-entities. The border is normally defined physically, but in some models the system of a human includes also the aura. In this case the system is a virtually entity overlapping other entities.

If the system contains an actor then the system is a live system. On the other hand if the system only contains artifacts or objects the system is a dead or static system.

## 3.10. Communication

The coordination within a system influences the communication. Coordination helps to bring the right entities into relation and provides them possibilities to communicate. We can state that communication is a sign-mediated interaction which includes all types of information exchange between two or more entities. It is mostly done as a two-way process, where both sides interact with each other. Nevertheless, one-way communication is possible too.

**Definition 3.6** *Communication is the process of transferring information from one entity to one or more entities.*

In a more technical approach communication can be modeled using different levels and protocols to control information exchange. The generic model distinguishes 3 layers:

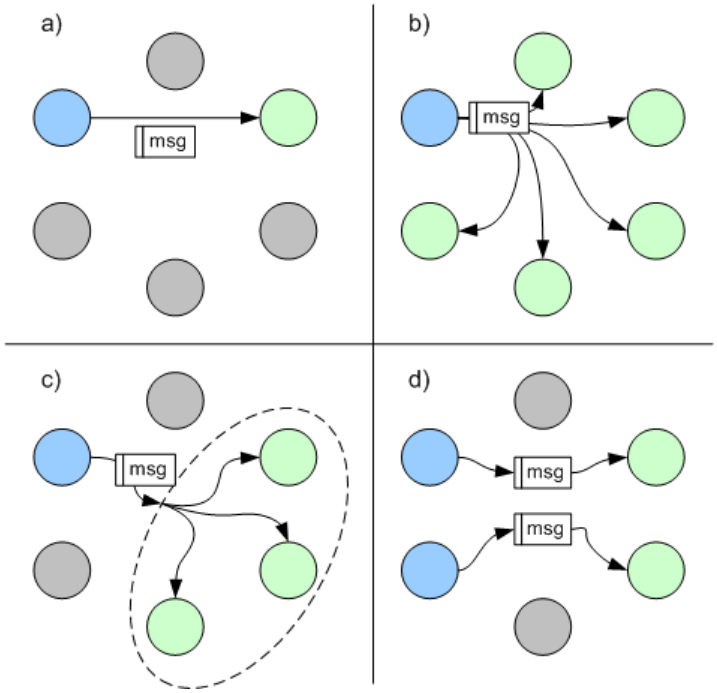



Figure 3.15.: basic communication paradigms: a) peer-to-peer b) broadcast c) multicast and d) generative communication

- ***Pattern***: how and where is information transferred?
- ***Paradigm***: what paradigms exist in order to communicate?
- ***Protocol***: what protocols are used?

## 3.10.1. Communication Pattern

A communication pattern describes how messages are sent between entities. M. Schumacher proposed several communication patterns in [Sch01] (figure 3.15).

- ***Peer-to-peer communication***: the messages are sent directly to an entity
- ***Broadcast communication***: the message is sent to all entities in the system. Entities interested in the message evaluate it; all others drop the message.
- ***Multicast communication***: the message is sent to a specific group of entities. The broadcast is a special type of multicast communication where as the group is the universe.
- ***Generative communication***: the message is sent to a blackboard or a pool. Entities read the message any time after transmission (asynchronously).

In order to implement a communication pattern, ports and communication channels must be properly defined.

### Communication Channel

Once two ports are connected they communicate over a media called *a communication channel*. The channels are mediated by the coordination. The communication channel

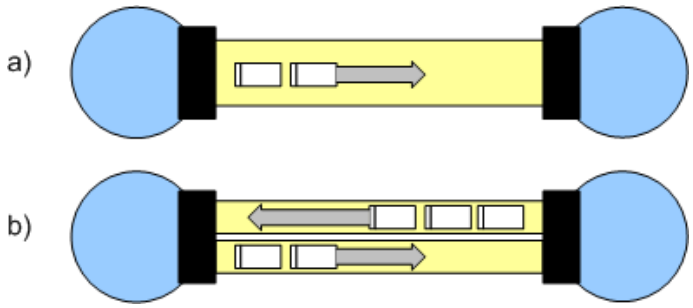



Figure 3.16.: a) unidirectional communication between two entities b) bidirectional communication realized with two inverse directed communication channels

is defined very generically using following definition given by Shannon and Weaver in [SW49]:

**Definition 3.7** *A communication channel is the medium used to transmit a signal from transmitter to receiver.*

It's important to note that the communication channel is always unidirectional. Information can be transmitted only from one entity to another. To setup bidirectional communication, two inverse channels are needed (figure 3.16).

Because it is possible to have multiple receivers we use a modified definition from the one above:

**Definition 3.8** *A communication channel is a medium used to transmit information from one entity to one or more entities.*

This allows us to implement patterns sending one message to multiple receivers over the same channel.

### Communication Ports

Ports help to separate the entity from other entities when communicating. Two types of communication can be setup: *identified* and *anonymous*. In identified communication the participating entities know each other. On the other hand in anonymous communication the entities have no knowledge about their communication partners.

Before two entities can communicate with each other, their ports must be connected. The connection process first checks if the ports match. This can be done either explicitly by an entity or implicitly if the context of an entity changes. For instance all people hear the radio when they are close to it. Their ears are *connected* to the speaker of the radio in a implicit way.

The port matching depends on the port features. The following list was defined by Oliver Krone in [KCDH98] and shows some possible basic features for port matching:

- ***Primary features***: they define the semantics of a port

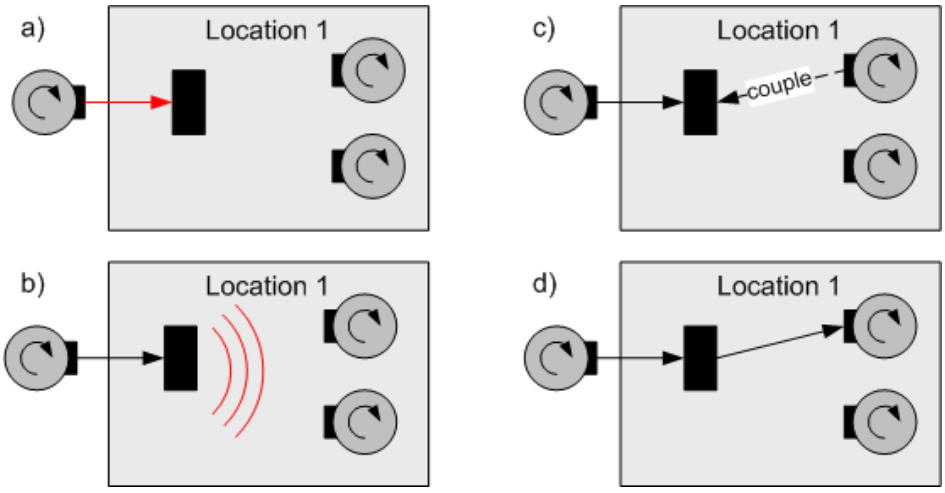



Figure 3.17.: Shows how communication between an uncoupled port works. a) caller connects to the port. b) port rings. Ringing is a broadcast to all entities nearby. c) one of the entities gets coupled (connected) to the port. d) caller finally communicates with an entity.

  - Communication structure, like peer-to-peer, broadcast, generative communication
  - Synchronization type, like synchronous or asynchronous communication
  - Orientation, like input, output or in-out
- ***Secondary features***: describe the characteristics of a port
  - Saturation: denotes how many connections at once are allowed for communication (from 1 to infinite)
  - Lifetime: declares how many messages can be passed to the port (from 1 to infinite)
  - Protocol: describes the accepted communication protocols

If a port is coupled to an entity the port is *active* otherwise it is *inactive*. The coupling between an entity and a port is called an interface. There are removable ports and unremovable ports. For instance a telephone is removable whereas the human's ear is unremovable. In our model we allow the connection to an inactive (uncoupled) port. For example, somebody calls a mobile phone, the mobile phone will ring until somebody is coupled to the phone port and answers the call (figure 3.17).

### Port Types

So far we have seen ports as an entity concerned with receiving or sending information. Ports can be devided into 3 main groups:

- ***Source and Slot***: ports of this type are used by entities to send and receive information
- ***Mirror***: a mirror reflects the information about an entity. It can be used as a relay to indirectly communicate with entities. For instance a driver uses the back mirror to see whats behind a car. Mirrors can be used also as a direct reflection

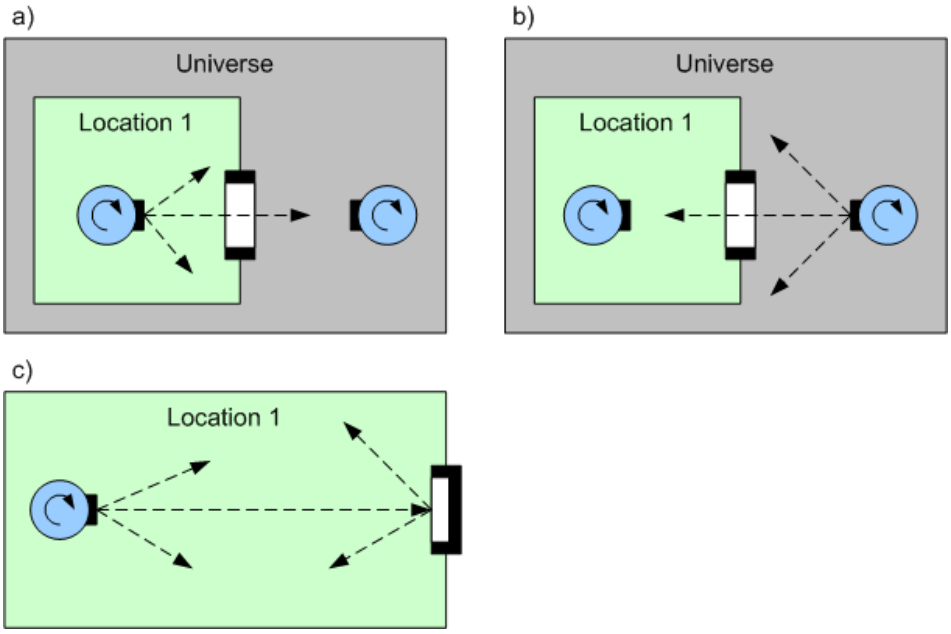



Figure 3.18.: Shows different port types. a) passing information from inside a room to the outer world. b) passing information from outside into the room. c) A mirror reflects information inside a location

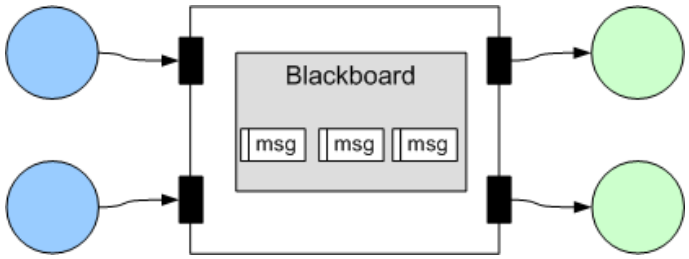



Figure 3.19.: A blackboard is not just a port, but an entity implementing a generative communication paradigm

of information. For instance people look directly into a mirror to check if their hairstyle is okay.

- ***Window***: people staying inside a room can see the outside world through a window. The window passes the information from outside into the room (figure 3.18c). A window can be transparent in one or both directions. It can even reject some types of information. A window is therefore also called a *filter*.

### Blackboard Communication

Blackboard communication is an implementation of a generative communication pattern. Many different communication paradigms can be implemented using a black board.

A blackboard is an entity with ports (see figure 3.19) which takes messages and puts them on its board. Any entity can read the message as long as it stays on the board. Once an entity removes a message, it is deleted and no longer available for other entities. The

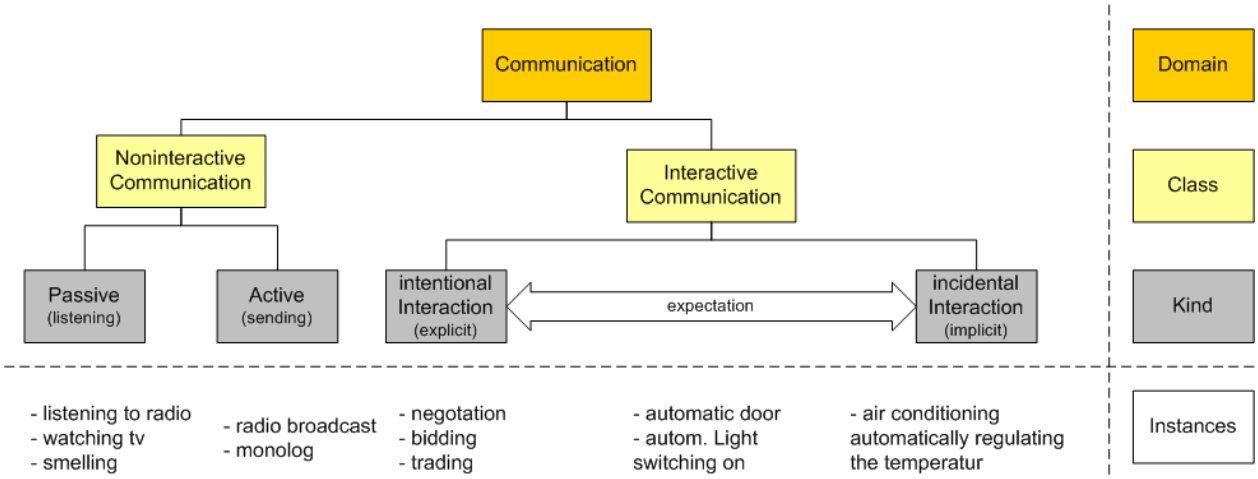



Figure 3.20.: Taxonomy of communication classification

blackboard can event support privacy in the sense of hiding some messages from other entities. Only some dedicated entities might have access to private messages.

## 3.10.2. Communication Paradigm

In contrast to communication patterns the communication paradigms are situated on a higher level. They deal with the communication semantics and the effects they have on entities rather than the syntax. Generally there exist two main paradigms: *Interaction* and *Non-Interaction*. By interaction we mean any kind of action, e. g. communication or physical forces, that occurs as two or more entities have an effect one another. Since we are dealing with communication two communication paradigm classes can be defined: *interactive* and *non-interactive* (figure 3.20).

To distinguish between the two communication paradigms the message and the content flow must be formalized. In classical distributed systems a *happened-before* relation, denoted by $\rightarrow$, is defined [CD88]:

1. If $\exists$ process (entity) $p_j$ : $e \rightarrow_j e'$, then $e \rightarrow e'$
2. For any message $m$, *send(m)* $\rightarrow$*receive(m)*, where *send(m)* is the event of sending the message, and *receive(m)* is the event of receiving it.
3. If e, e´and e˝are events such that $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$

The Lamport clock is a simple implementation by which the happened-before ordering can be captured numerically. Each process maintains a software counter which is monotonically incremented at each event. An other implementation are the vector clocks, which help to generate a total order of events [CD88].

To model interaction we have to define a new relation called *has-an-effect-on*, denoted by $\rightsquigarrow$. This relation includes the *happened-before* relation and takes the semantic dependency between messages into account. In other words, if the content of a message $m_2$ depends on the content of a message $m_1$, we denote $m_1 \rightsquigarrow m_2$.

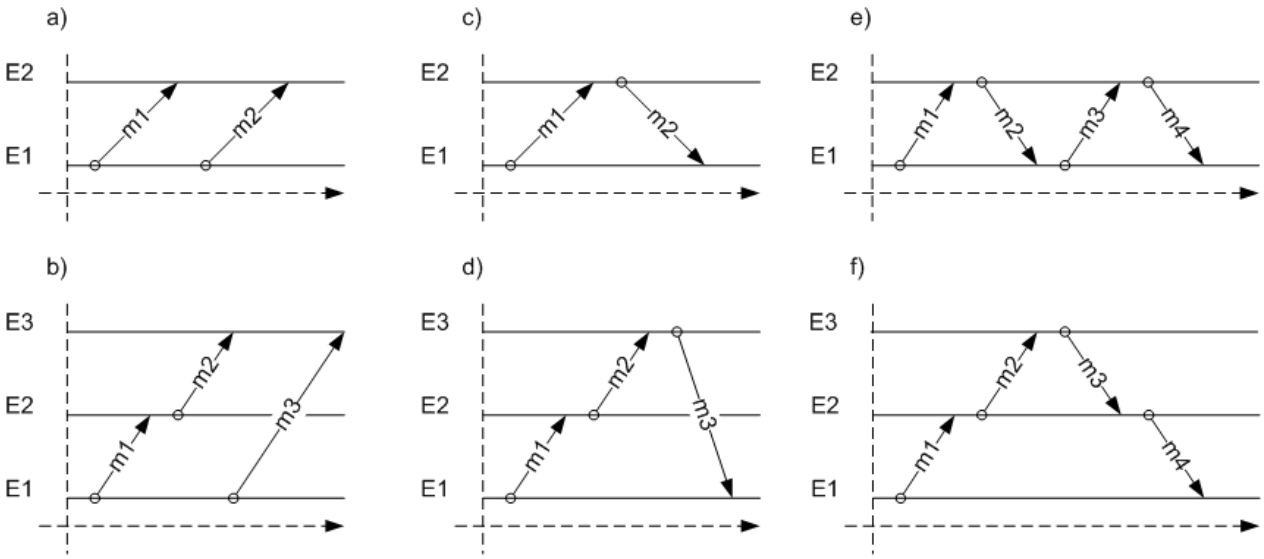



Figure 3.21.: Examples of communication paradigms

An interaction involves two or more entities and at least two messages which depend on each other. Generally there are 3 types of entities involved when communicating:

- ***Proactive Entity***: an entity which initiates a communication
- ***Reactive Entity***: an entity which responds to a received message
- ***Passive Entity***: an entity which is receiving messages but is not responding at all

Figure 3.21 shows some examples of the communication paradigm using the interactive and non-interactive paradigms. a) and b) show non-interactive communication. b) m1 and m2 have a causal dependency. c), d), e) and f) show interactive communication patterns. In d) the interaction occurs between E1 and E3, where as in f) the interaction occurs between E1 and E2, and between E2 and E3.

At the beginning of an interaction an entity has to act proactively by sending a message to another entity and this entity reacts to this message by sending the response. The interaction will stop as soon as an interacting entity receives the last message and does not respond to it.

Non-interactive communication can be split into two different kinds, active and passive message exchange:

- ***Passive***: an entity is receiving messages, but is not responding to them (no feedback or acknowledgment)
- ***Active***: an entity is sending messages but not receiving any acknowledgments

Interactive communication can be split into two extreme, intentional interaction and incidental interaction. Alan Dix in [Dix02] defined a continuum for interaction based on the expected feedback. On one end we have intentional interaction. Feedback is expected by the interacting entity. On the other end incidental interaction is defined. The entity is not aware of or does not expect any feedback. Between the two extremes the differences are often not clear. They depend on the level of intention and the expectations of the entity involved in the interaction.

- ***Intentional interaction***: an entity explicitly interacts with his communication partner and expects feedback
  - *Negotiation*: the negotiation process between entities generates temporary agreements. The agreements can be re-negotiated in case of need (e.g change of context).
    - Bilateral negotiation: two entities (or parties) are participating in the negotiation
    - Multilateral negotiation: more than two entities (or parties) are negotiating on one subject
  - *Trading*: an entity tries to buy or sell some resources. The law of demand and supply takes control over this process.
  - *Bidding*: a special type of buying. The entity with the highest bid gets the resource (or task).
- ***Incidental interaction***: an entity implicitly interacts with another entity (unintended). Feedback is not expected.

### 3.10.3. Communication Protocol

Protocols help to structure and control the communication. They govern the way in which entities communicate with each other using a set of rules. Without protocols the entities would not understand each other.

**Definition 3.9** *A protocol is a set of rules governing the communication between entities.*

Communication can be organized in many ways. Based on the work of Ch. W. Morris in [Mor71] communication can be seen as a process controlled by three levels of semionic rules:

- ***Syntactic rules***: relations among signs in formal structures
- ***Semantic rules***: relations between signs and the things they represent
- ***Pragmatic rules***: relations between signs and their effects on those who use them

Whenever a protocol is defined these three levels of semionic rules have to be considered. Not all levels are properly defined. For example, pragmatic rules are often implicitly or informally defined within a protocol specification.

# 4

# Coordination for Pervasive Computing

## 4.1. Introduction

This chapter focuses on a coordination model for computer science which is based on the generic coordination model. We show how the model can be adapted for pervasive systems, but do not apply it to any other computing systems suchas multi-agent-systems or grid computing. Therefore we call this model *the pervasive coordination model*, PCM.

In pervasive computing the information processing moves out of a user's focus. The user concentrates more on their task rather than on a tool like a computer. The purpose of such a system is to support the user in their task. Pervasive computing has been defined as context aware applications which use sensors to observe the activity and the context of participating users.

Pervasive computing systems can be extremly complex to setup and to manage due to the heterogeneous nature of the devices (smart phones, sensors, networks). Weiser in [Wei91] has introduced the concept of calm technology. In his concept, the user is increasingly surrounded by computing devices and sensors. In order to avoid an unneeded cognitive load for the user to interact with those devices, it becomes a nessecity to limit the direct interaction. Coordination is an important issue to reduce unnecessary interaction with computing devices. It helps to provide useful and expected information in time at the right place. This can be achieved for instance by context and activity awareness. The computing system has an impression of what the user is doing (activity) and under which circumstances (context).

The rest of this chapter is organized as follows: First, we apply the generic coordination model to computing systems and explain the different coordination structures suchas the physical, virtual and task structure. The explanations are valid for all computing models. For pervasive computing we add a few extensions and explain how they fit into the model. The last part of the chapter concerns the communication types in a computing system, such as human computer interaction and networks.

## 4.2. Computer System

Traditional computer systems were and still are considered as closed systems. They are used as input-process-output devices. The users (humans or other computer systems) were situated outside of the system but interacting with it through well defined devices, such as keyboards, screens, mice etc. The focus of coordination was mainly in managing hardware devices and software components. To perform the needed coordination processes the computer was equipped with an operating system handling all the dependencies between running tasks and resources. For instance if two tasks open a file both are able to read the content, but typically only one task is allowed to change the content.

In distributed systems the point of view changed slightly towards coordination of remote devices and distributed resources. Computers were wired together using a network which allowed them to interact with each other.

With pervasive computing, which is a sub-discipline of distributed computing, the user has become an important element of the system. They are considered and observed by the system and provide contextual information to get the expected services.

Before going into the coordination, the computing system must be structured. A computing system is built with the following layers:

- ***Hardware***: physical entities such as the processor, input-output devices, sensors, actuators, bus systems
- ***Operating System (OS)***: software managing the hardware and providing an API to access its functionality
- ***Middleware***: framework providing services and functions on a higher level of abstraction. The middleware typically fills the gap between the operation system and the applications.
- ***Application***: user and problem specific implementations

In earlier models the user was put on top of the application level, showing the direct interaction User-Application (figure 4.1). Nowadays the user is put at the hardware level because the user interacts directly with the IO devices, such as keyboards, mice and screens. The interaction between human and computer or between humans happens on the same level. For instance when a human presses a key on the keyboard, he interacts with the keyboard and gets a first feedback through his touch senses. The keyboard sends a message to the internal computer hardware. The message is treated by the different software levels until it reaches the word processing application. The logic puts the corresponding letter at the cursor position and generates a feedback message which travels the opposite way up to the screen.

The proposed separation of coordination considers the fact that today a computer system is built up of hardware, an operating system and applications (figure 4.1):

- ***Coordination at the hardware level***: the coordination is mostly hardwired and done using simple communication protocols. For instance if two computers try to send data over an Ethernet cable, the hardware coordinates the access. If one computer detects a communication conflict it will wait for a random time and then try again to access the connection.
- ***Coordination at the OS level***: the operating system manages the dependencies between processes (activities) and system resources such as hardware devices and system objects. Modern systems also implement security management to coordinate user access.
- ***Coordination at the the middleware level***: coordination in the middleware manages many different dependencies within a system. For instance it manages the interdependence of processes by mediating communication spaces and channels, or it might help to manage the dependencies between human activities for pervasive applications.

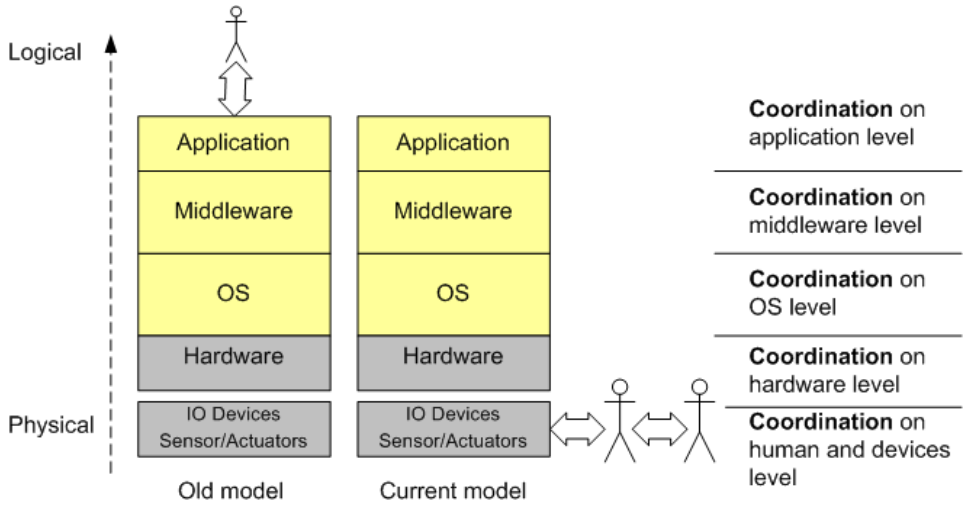



Figure 4.1.: The old model puts the user on the top. Current models put the user at the hardware level interacting with the IO devices and sensors.

## 4.3. Physical Entity Structure

According to the generic coordination model each entity can exist only once in space and time and overlapping is not possible. The following physical entity classes exist in computer science:

- Hardware devices
- Network topologies
- Users
- Elements of programming languages

### 4.3.1. Hardware Devices

The name *hardware* denotes that the object it refers to exists in the physical world of things. This implies that all hardware devices are physical entities. A computer is a composition of different devices and can be split into several sub-compositions suchas the screen, a keyboard, a hard drive, a processor etc. Each of these devices can be split again into several sub-entities. For instance the processor is made of registers, the ALU, clocks and internal buses (figure 4.2).

In this document, we define computers as all hardware devices used to compute data, such as personal computer, main frames, mobile devices or other micro-controller based devices.

A computer has a physical location which gives the environmental context. For traditional computing this context is often not relevant and is ignored. But, for pervasive computing the location and the environmental context gives useful information about how the device is used. Also, mobile devices often contain various sensors which might be usable to observe the human user of the device.

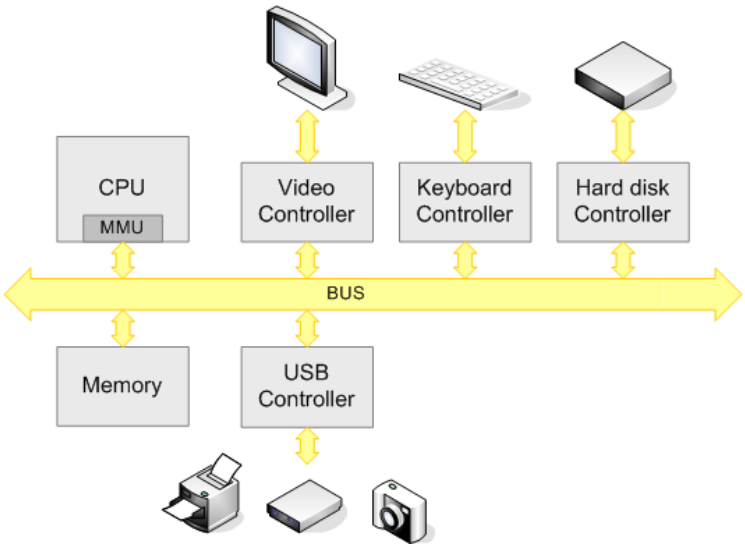



Figure 4.2.: Computer hardware can be separated into several hardware entities.

The following list is not exhaustive, but gives an overview of which physical entities are considered in pervasive computing in addition to those normally used in computer science:

- Location of the computing device
- Any sensors attached to the device
- Humans

### 4.3.2. Networks

The different hardware components are often connected with each other by a network. The network allows them to exchange information. There exist many different network topologies like:

- ***Bus network***: all components are connected to one communication line. Two main techniques are common to manage bus access.
  - A master is controlling the bus and grants access to other devices. The master entity is the coordinator of the communication. Examples are PCI bus, Profibus, ASI Bus.
  - A protocol (rule) is used to detect if the bus is free and to handle data collisions. The coordination is more implicitly given by the rule each device has to follow. The most famous example is the Ethernet protocol using coaxial cables.
- ***Star network***: all components are connected to a central device called switch. All messages arriving at the switch are sent from there to the target stations. The switch uses an addressing protocol to locate the target station. For instance an Ethernet switch processes the data on the network layer of the OSI model. The switch is responsible for sender-to-target package delivery including the routing through intermediate hosts.
- ***Ring network***: each host connects to exactly two other hosts forming a closed loop. The data travel through all nodes, which read and handle the data package. The ring network performs very well for heavy network loads and does not require

a special server to manage the connectivity. Note that the famous token ring uses a ring topology, but the ring is not wired. It uses a wired star topology and a special software protocol.

- ***Tree network***: a network is split into several sub networks forming a tree. Each sub-network has its own address range and is managed independently as long as local devices are addressed. The parent node is managing all messages which have to be passed to a device in a different branch of the tree. A typical example is the Internet using IP addresses.
- ***Mesh network***: the devices are directly connected to each other. Each connected pair has its own connection line. A fully connected mesh network has a complexity of $O(n^2)$ and is very costly for larger networks. Often partially connected mesh networks are built to create redundancy within a network.

To summarize the coordination at the hardware level, the management is often hardwired or at least uses static protocols. Many networks use software layers to realize a more complex coordination.

### 4.3.3. Human Users

The human user is the central entity of a computer system. Former models often gave a secondary role to the user. This often led to bad or unsatisfactory design of the hardware and software components. Nowadays the user plays a central role. With mobile computing and pervasive computing the user is *the* entity to be considered and observed.

There are other humans to be considered in a computing system suchas system architects and engineers who design and build the system and later, the system engineers maintain it.

Both human groups are important to the system and often the same human plays different roles. Within the coordination model for pervasive computing we propose that all participating humans are entities of the system and must be considered at all times. By doing this we believe that the system will evolve in a smooth and practical way to accommodate the needs of the human users.

### 4.3.4. Elements of Programming Languages

A programming language can be represented by an abstract syntax tree (AST). The nodes of the tree can be seen as physical entities even though they describe the behavior of pure virtual entities. All leafs of the tree represent terminals of the language whereas intermediate nodes simply represent a composition of terminals and other composite literals.

There might be more than one tree representing the same programming code. How the tree is built up is relative to the interpreter (observer) of the programming language. But once the interpreter has parsed the literals, each terminal belongs unambiguously to one parent node of the tree. Therefore the elements of the programming language meet the

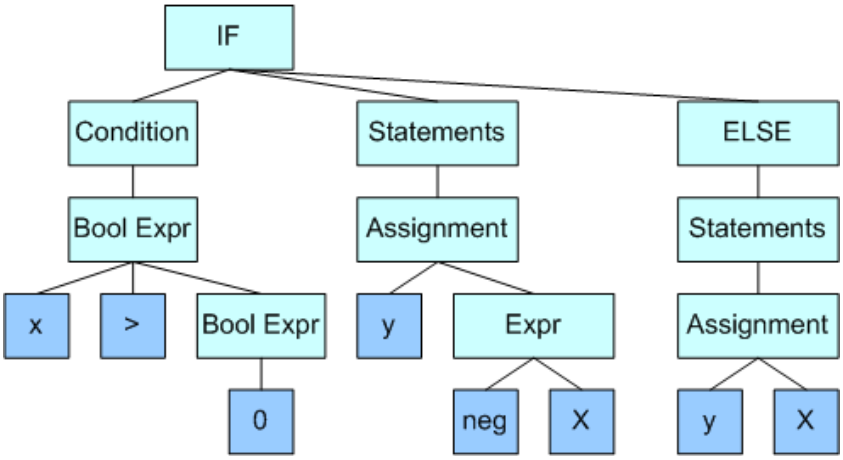



Figure 4.3.: Parse tree representing a program code

conditions of the physical structure. The example in figure 4.3 shows a simple syntax tree of the programming code

```
1 if (x > 0) then
    y:= -x;
3 else
    y:= x;
```

Listing 4.1: Example code to show the abstract syntax tree (AST)

## 4.4. Smart Environment

The pervasive coordination language helps to manage the dependencies of activities in an physical environmen enriched with sensors, actuators and mobile devices. The environment is not only observing the humans but is also able to interact with them, for instance using services. We define a service as:

**Definition 4.1** *A Service is the performance of any duty or work for another entity [9].*

An enriched environment is called *a Smart Environment*. The goal of smart environments is to make the life of humans more confortable by replacing their hazardous work, physical labor, and repetitive tasks with intelligent and pervasive agents.

**Definition 4.2** *A Smart Environment is a physical world that is richly and invisibly interwoven with sensors, actuators, displays, and computational elements, embedded seamlessly in the everyday objects of our lives, and connected through a continuous network [Lew04].*

The smartness of the environment is influenced by two major aspects: 1) how well coordination is performed and 2) how intelligent the provided services are. For instance a location-based internet search is an intelligent service providing useful information. But, if the service is never offered to the humans (or is offered at the wrong time), it becomes useless. The human users will experience a smart environment only if the service is proposed or available in the correct situation.

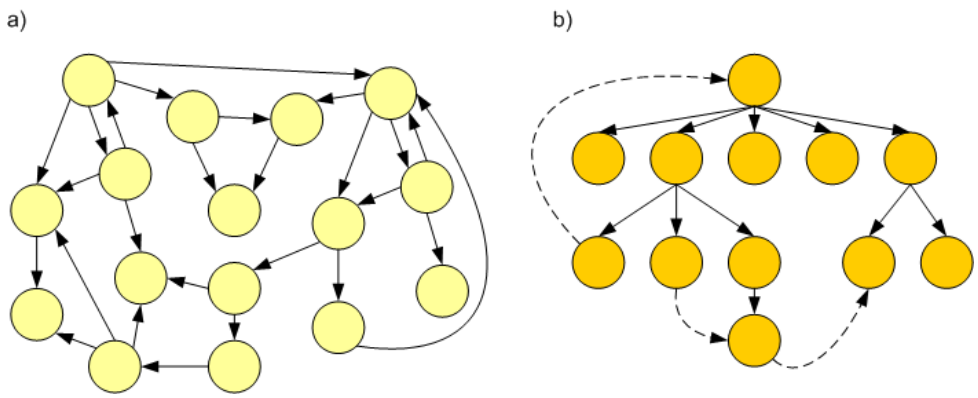



Figure 4.4.: a) software structure is normally a graph. b) tree organized software structure. Software components are able to reference each other (dotted line).

## 4.5. Virtual Entity Structure

A computer system is typically designed to treat different aspects of the physical world at a higher level of abstraction using logical and mathematical terms. Whereas the gathering of information through hardware devices is done on the physical structure, the treatment of information is done in the logical or virtual structure. The following components of the computing system belong to logical or virtual structure:

- ***Operating System***: the operating system (OS) deals directly with the hardware devices (physical entities) of a computer system. Often the OS abstracts the hardware devices using drivers and the coordination is done using more abstract system objects. It manages the dependencies between those objects and the processes.
- ***Semantics of the program code***:
  - Software components
  - Applications and services
- ***Virtual Machines***: a virtual machine is able to run on distributed hardware platforms
- ***Virtual Memory***: the virtual memory is an abstraction of physical memory. A virtual memory page can exist in RAM, in the cache and on the hard drive at the same time. Typically the operating system takes care of their management. A virtual address is mapped to a physical RAM address by the memory mapping unit (MMU) of the processor.

### 4.5.1. Software Components

All software components and objects are virtual entities. Depending on the intention of the system some of them can be treated as physical entities. For instance the software representation of the physical world can be designed to have a similar behavior as physical entities (virtual reality, simulation).

In most cases the structure of the software is a graph (figure 4.4a). Software entities reference other entities and form a tight net of relations, the virtual structure of the software. In order to model the physical world (or similar behavior) special software framework

are used. They help to simplify the software structure using a tree of ownership. Every entity has an owner except the root. An entity might be decomposable into several sub entities. This tree is able to represent a physical structure. All other relations are strictly defined as references and represent the virtual structure (figure 4.4b). Examples are:

- Graphical User Interface designs such as VCL components of Embarcadero[1] or QT library [BS06]
- XML document structure which is a tree and is used to structure the data [11]
- The Entity Space of the uMove framework, which represents observed entities [BH09]

## 4.6. Task Structure

The task structure manages the dependencies between tasks. Within a computer system two main groups can be distinguished:

- Computer tasks
- Human tasks

### 4.6.1. Computer Tasks

In modern operating systems, computer tasks are defined as processes. A process is the activity of a micro processor and its context (state) is stored within the registers of the micro controller. While processing the program code (description of the activity) the processor builds up relations to different system objects. For instance it allocates parts of the memory to store the processed information, or it opens a file on the hard drive for reading. Those dependencies are stored in registers or private memory pages.

Operating systems often implement a multitasking feature. This allows to sequentially assign different processes to one processor by interrupting other running processes. This coordination process is called scheduling and allows to parallelize processes. The context of each process (including all dependencies) is stored in a process table entry and must be updated and restored when switching from one process to another. Such an entry typically contains a process ID, values of the processor registers and a program counter.

Multiprocessor systems work in the same way, but involve the additional task of coordinating the access of entities. For instance the problem of mutexes and memory caching involves advanced coordination and communication techniques.

There exist different scheduling strategies. The following list gives a brief overview of the most common strategies:

- ***First-Come First-Served***: simplest non-preemptive scheduling algorithm using a FIFO queue (First In, First Out)
- ***Round-Robin Scheduling***: each process receives a time interval (quantum). The process is switched if it voluntary releases the processor (sleeping) or if the time interval exceeded. If the process is not terminated it is put at the end of the waiting process queue.

- ***Priority Scheduling***: similar to round-robin but uses a priority queue for scheduling. Each process has a priority to be considered. The process with the highest priority is in the front of the queue and treated next.

### 4.6.2. Human Tasks

Today computers help to organize and plan the work of humans. Human tasks are often managed by highly specialized applications (for instance MS Project or MS Outlook[1]). The tasks are normally predefined by name, duration and the resources needed. Smart tools help to detect conflicts and work overload.

In pervasive computing human tasks are not predefined, but observed and perceived. The dependencies between tasks are often very hard to detect. In order to simplify their detection, smaller pieces of a complex task, called activities, are observed. One of the main goals of pervasive computing is to provide suitable information about the current user's activity and context to different interacting objects, application and the user itself.

## 4.7. Physical and Social Laws

The generic coordination model defines physical law as static and unchangeable, whereas social laws are dynamic.

### 4.7.1. Physical Laws

Physical laws give strong restrictions. One of the main restrictions is that a computer can not do more than what a Turing Machine is able to do [Koh07]. For instance there exist problems which no computer can solve, because it is proven that these problems can't be solved by a Turing Machine. There also exist problems which can only be solved with Turing Complete languages (using while or goto). Some languages which are not Turing Complete are not able to solve these problems. Therefore the computing system and the programming language restrict the domain of solvable problems. This restriction is given by a physical law which holds for all computers.

Another example of physical restriction is the speed of light, which is used to transmit information between two systems. If a system queries data and expects a responce within a fixed timeinterval, the target system must be in range, meaning the time for sending, processing and sending back should not exceed the expected timeinterval. For instance the rate of the processor clock limits how far the internal processing registers can be located in order to receive the stored information before the next clock interval.

For pervasive computing other physical restriction must be considered. Observing the physical world is limited by the restrictions on measuring physical quantities, such as temperature, gravitation, pressure, etc.

[1] Products of The Microsoft Corporation®

### 4.7.2. Social Laws

Social laws are defined and changeable within the system. In traditional computer science many operating systems and applications use security settings. Security settings are used to grant access to information and the use of devices for different user groups. In general there are two types of security settings: permissions and restrictions. Permissions are used to grant access to a system in which access is blocked by default. Restrictions do the opposite. They are used to limit access to a system where access is open by default.

On the other hand, there exists unwritten social laws on how to use and work with computers. For instance how an employee writes an e-mail to his customers is culturally and socially dependent.

### 4.7.3. Social Laws for Pervasive Systems

For pervasive systems a more flexible way is needed to govern the interaction and connectivity between the different human users and services. Instead of having static security settings we propose context related security settings. For instance, before establishing a connection between two entities the social laws of the environment must be evaluated.

## 4.8. Observer

There exist different observers within a computer system suchas humans, sensors or pure software observers. A human observer mostly observes the system in order to improve the performance and to find bugs. The human uses his senses to observe the output devices such as screens, printers etc. This view can be extended by special software tools which help to monitor and capture data. Examples are the program debugger, network monitor and virus scanner.

The sensors help to get some states of the computing system, such as temperature and power level or consumption. Current personal computers are equipped with special power safe monitoring which helps to regulate the processor ventilation fan or the screen backlight. Sensors are also used by industrial computing to observe the state of a machine and to react upon changes using actuators. Typically a PLC (programmable logic controller) is used to logically combine the states of the sensors and to control the output signals.

The last group of computing observers are software observers. They help to observe software components and their state changes in the virtual structure. There exist many different kinds of software observers. The following list gives a short overview of the most important ones:

- ***Observer-Subject pattern***: the pattern is proposed by Gamma et Al. [GHJV95]. It is one of the most used in object oriented software engineering. The pattern helps to logically separate the observer from the subject (observed entity). Some implementations (like Java) use the name *Listener* instead of *Observer*.
- ***Hooks***: hooks are used to observe the communication between two entities. For instance the Windows operating system offers them on the API. The hook is in-

stalled between the sender and the receiver to observe and monitor messages (man in the middle).

### 4.8.1. Entity Classification

In traditional computer systems observers use a more or less static entity classification. The classification is given by the design of the system and contains actor, software agent and artifact.

#### Actor

Usually we define the human user as an actor in the computing system because he is capable of doing and acting in his environment. The type of design of such system is called *user centered design*. Most applications are designed this way, like desktop applications and internet services.

The computer often plays the role of actor for technical application designs. The focus is the work and the activity of the computer. Examples for technical application designs are grid computing and multi agent systems (MAS).

#### Software Agent

A software component which is able to sense its environment and act upon the sensed information is called a software agent. Software agents usually run in threads and processes. As we have seen, the process (or thread) is the activity of a software component. The thread allows an agent to change the internal state autonomously and to exchange information with the environment of the component. The behavior of an agent is typically written in a programming language [Sch01].

#### Artifact

All data generated by humans or computer systems are artifacts (e.g. files, print outs). Artifacts are subdivided into tools, ports and viewers. Instead of adding a new kind of entity for information, we define information as a virtual tool to perform an activity. Since this classification is relative, the following examples give an idea, on how entities can be classified.

- ***Tools***: the computer generated data is used as a tool in for further activities. Also all IO devices can be seen as tools since they are used as such by humans.
- ***Port***: all interfaces between humans and other computing systems are ports(screens, sensors and network interfaces). Software ports are used to access the functionality of software components. Typical software ports are interfaces used in Java or CORBA. Also the methods of a Java class can be seen as a port to communicate with an instance of the class.
- ***Viewer***: the viewer is a filter used to capture information. A viewer could be a network filter or a graphical user interface showing only relevant information.

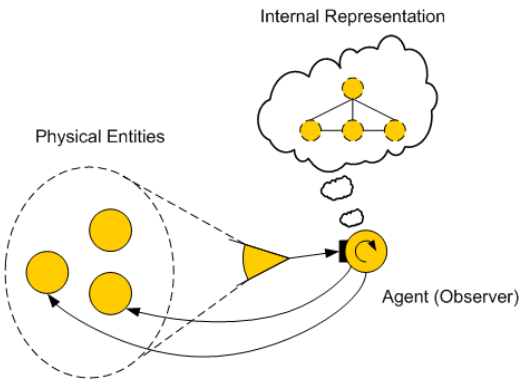



Figure 4.5.: A pervasive computing system

## 4.9. Observer in a Pervasive System

A pervasive system observes the physical world in order to get the context and the activity of physical entities and provides contextualized services to them. In our holistic approach the pervasive system includes not only the computing system (e.g. sensors, networks, software) but also all physical entities observed (e.g. humans, cars).

As we have mentioned in the generic coordination model the perception of an observer is relative. The observer is not able to understand the perceived world completely, but rather creates an abstraction, a model to simplify the world for his needs. This model is called *an internal representation* (figure 4.5).

In this sense we can describe a pervasive computing system as follows:

- A pervasive computing system observes entities in the physical world. Depending on the observation the system models an internal and relative representation of the observed situation and behaves according to rules (programmed logic).
- In the literature the internal representation is called context aware if it includes the observed environmental context of the physical entities [ADB+99].
- If the internal representation also takes the activities of the physical entities into consideration, it is called activity aware or motion aware system [BH09].

When we deal with observation, we face a duality between the reality and its internal representation. This fact must be considered when designing a pervasive system in order to analyze and improve its behavior.

### 4.9.1. Sensor

A sensor is a device which transforms a physical quantity into electrical signal. This transformation process is also called *sensing* and is a primitive level of observation. Nowadays sensors become smarter and also pre-process the raw data and add some semantic to it and in this case we use better the word *observing* rather than sensing.

**Definition 4.3** *A method of data collection in which the situation of interest is watched and the relevant facts, actions and behaviors are recorded [4].*

The world is observed by the sensors connected to a pervasive system. Each sensor allow to have a view onto a part of the world. For instance a temperature sensor gets the temperature of an entity. The range and the sensitivity of the temperature sensor is its view. If the temperature is changing less than the sensitivity, the change is not captured by the sensor.

Sensors are able to capture different aspects of the pervasive system.

- ***Context***: one of their main goals is to capture the contextual information of the observed entities and their environment.
- ***Activities***: for activity aware and motion aware application, sensors also help to capture the motions of the tracked entities.

There exist two groups of sensor usage:

- ***Endogenous***: a sensor which is attached to an entity and is able to sense data from that entity. Typical examples are temperature sensors mounted in a room to observe its temperature or sensors attached to a human observing his heartbeat.
- ***Exogenous***: a sensor which is attached to an entity but is observing the environment around the entity. Examples are temperature and light sensors attached to a mobile device. They observe the environment of the user of this device.

This distinction is necessary in order to treat and store the contextual information in the right place. For endogenous captured data the contextual information belongs to the entity wearing that sensor. For exogenous captured data, the contextual data belongs to the environment of the entity (its parent).

### 4.9.2. Internal Representation

The internal representation of the pervasive system is done with software entities. It is a rough mapping of the physical world enriched with additional information about the captured entities. It considers the physical structure of the entities, their context information and relations with other entities.

We have shown in the generic model that the entities are organized in three structures (figure 3.2). We suggest that the same three structural elements should be used to model the internal representation. This will help to give a more accurate representation of the physical world.

- The physical structure is used to put the observed entities into spatial relation (spatial dependency). This structure is organized as an entity tree (entity containing other entities having themselves children and so on).
- The virtual structure is used to bring physical entities into social and logical relations (social dependency). This structure is organized as a graph containing physical entities and other virtual entities.
- The task structure is used to put entities (physical or virtual) into relation if they are working or used in an activity (task dependency). This structure can be organized as a set of activities.

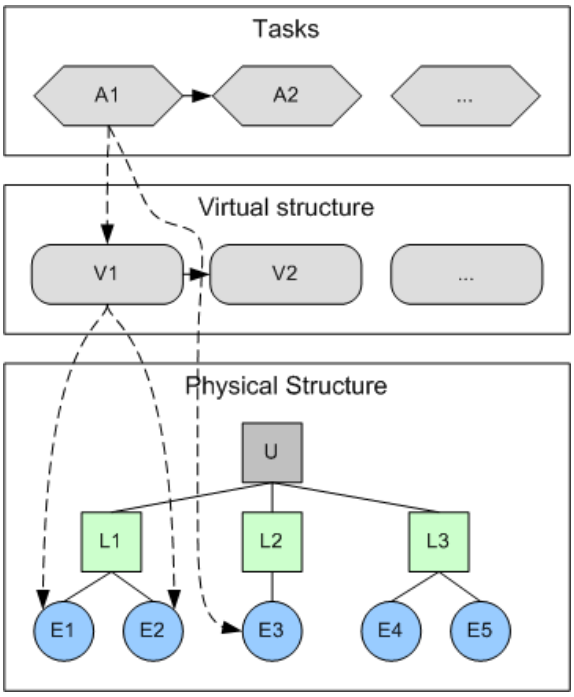



Figure 4.6.: Internal representation of entities and tasks.

The figure 4.6 shows how a virtual entity references other physical and virtual entities. The referencing denotes that the virtual entity V1 is composed of physical entities E1 and E2 and the virtual entity V2. An activity is able to reference entities. For instance, A1 references the physical entity E3 and the virtual entity V1. The referencing denotes that the entities play a certain role within the activity (such as actor, place, tool). An activity might have relations to other activities, meaning that an activity depends on other activities.

### 4.9.3. Relative Entity Classification

We have seen that entity classification on traditional computing systems is done more or less statically at the design phase. To be more flexible the pervasive system should classify the entities in a more dynamic way depending on the point of view on the system and the needs of the pervasive application.

Since the entity classification decides how the context of the entity is used and interpreted we propose a new way to set up the contextual settings for entities. Pascal Bruegger, in [BH09], defined several contexts for entities like identity, location, structure and relations. The activity of an entity is not considered as context, because activity is influenced by the contexts of the participating entities. Therefore only the role, which an entity plays in an activity, belongs to the context. The role describes how the entity participates in the activity. The context can be split into the following main groups (figure 4.7):

- ***Entity context***: the entity itself can have many attributes describing the contextual behavior, like identity, room temperature, humidity etc.
  - *Location context*: the location context is naturally given by the entity tree. The parent of an entity represents its location.

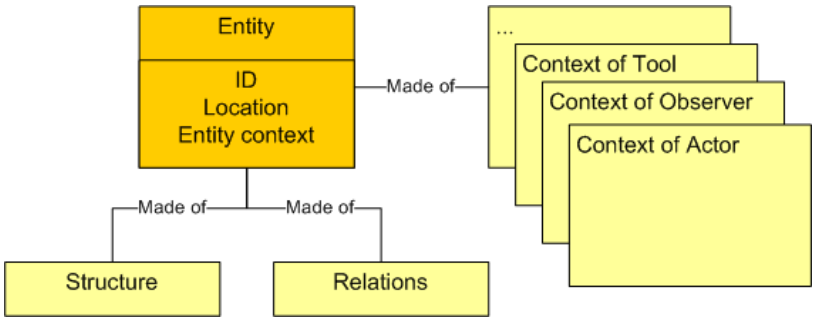



Figure 4.7.: Organization of the entity context

- ***Structure context***: the structure context tells if an entity is simple or complex. Simple means that an entity is an atom, complex means that an entity is composed of at least one sub-entity. This information is also given by the entity tree. The geometry information of an entity also belongs to the structure context.
- ***Relation context***: the relation context contains all memberships to virtual structures, such as communities, conversations, interactions. The social context is included in the relation context. Physical relations (suchas inside, next-to or joined relations) are handled by the structure context.
- ***Entity kind context***: each classification kind has its own context storage. For instance the role of an activity is relative and might change depending on the point of view.

The splitting allows two different concepts to be followed: First, an observer which is capable of entity classification can classify them and create a new role-specific context, without influencing another observer with its own point of view. Second, an observer can query the context of an entity in order to see how it can be classified.

### 4.9.4. Internal Observers

The pervasive system uses internal software observers to react to context and activity changes. If a change happens the observer will reevaluate the situation and each new situation will be reported at the application level (figure 4.8).

Dedicated observers also report situation changes to the rule evaluation component. Rule evaluation is part of the coordination model and produces actions to coordinate the system.

## 4.10. Communication

Communication in a computer system is very essential. There exist many levels of communication, such as:

- Human-Computer-Interaction (HCI)
- Network communication
- Interprocess communication

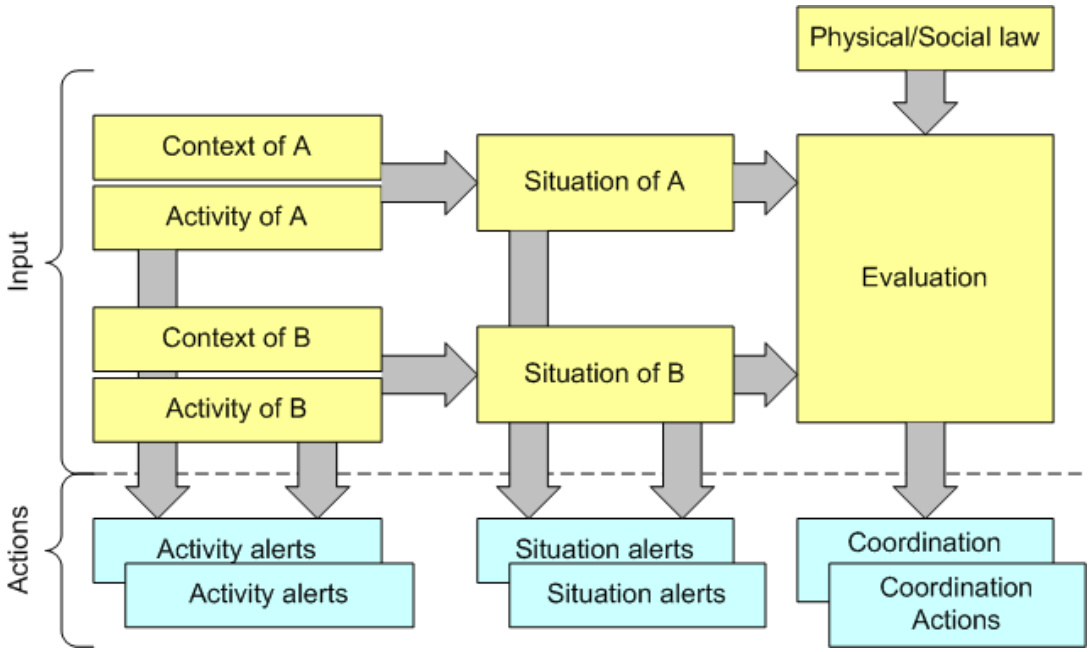



Figure 4.8.: Shows the chain of context and activity evaluation.

- Programming languages

### 4.10.1. Human Computer Interaction

Human computer interaction is an important research field in computer science. It analyzes how humans interact with a computing system. Interaction can be split into two main fields, intentional interaction (explicit interaction) and incidental interaction (implicit interaction). Traditionally, humans use input devices such as mouse, keyboard and touch panels to input data into a computing system. The feedback is mostly given by the screen or by audio devices. This kind of interaction clearly belongs to the intentional interaction paradigm.

In pervasive computing new ways of human-computer interaction are explored. Most of them are intended to be implicit. For instance the uMove framework [BH09] uses the motion and activity of a human actor as a primary input modality. In a concrete case study called Smart Heating System in [BPH09] the activities within a room control the heating system.

### 4.10.2. Network Communication

Computers are able to communicate together through a computer network. There exist many different network types each one having its special field of operation. Networks are characterized by the hardware protocols and the type of communication media. A few examples are:

- ***Bus***: hardwired parallel or serial communication. Mostly used for internal communication between the processor and peripheral devices.

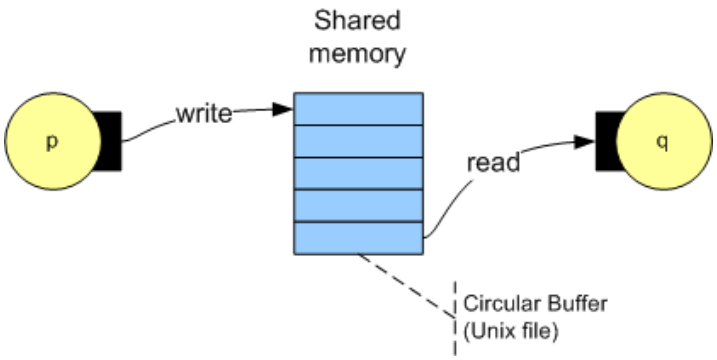



Figure 4.9.: Unix pipe as a communication media between two processes p and q.

- ***Ethernet***: wired serial communication. Used to link different personal computers and server stations together.
- ***WiFi and Bluetooth***: wireless serial communication. Used by mobile devices (phones, PDA, Laptop) to access the network through the air.

Software protocols are put on top of the hardware protocols. They implement different needs in order to communicate. Some of the needs are listed below:

- Addressing and look-up mechanism
- Security
- Fault tolerance
- Type of connection
  - Messaging like UDP
  - Transmission control like TCP

All communication patterns can be realized by network communication. Hardwired networks normally use the peer-to-peer pattern. Multicast and generative communication are implemented by software layers, like for instance the UDP/IP allows multicast by using special multicast addresses 224.x.x.x.

Networks using the air as a communication medium use the broadcast pattern. All listening stations receive the packages. The target device is able to read the information package using special security and addressing protocols. All other devices drop them.

### 4.10.3. Interprocess Communication

An operating system often offers different techniques which allow two or more processes to communicate with each other. These techniques are called *interprocess communication*. The following list gives a non-exhaustive overview:

- ***Shared Memory***: the operating system allows access to a shared page within the memory. There are different techniques for using this memory for information exchange, like:
  - Circular FIFO buffer
  - Blackboard (generative communication)

| *Notation* | *Meaning* |
|---|---|
| x=y; | syntactic: y communicates with x<br>semantic: y sends the value to x. The value of x equals the value of y afterward. |
| x *= y; | syntactic: y communicates with x<br>semantic: x is multiplied by the value of y and contains the result afterward. |
| x = x * y; | syntactic: x and y produce a result r which communicates with x.<br>semantic: x is multiplied by the value of y and contains the result afterward. |

Table 4.1.: Communication using the assignment of variables

- ***Message Passing***: the most commonly used type of message passing is the MPI standard (message passing interface). MPI was invented for parallel computing. Typically all processes working on the same problem use MPI to explicitly exchange task relevant information.
- ***Tuple Space***: the tuple space allows generative communication in computer science. It is an associative memory used in parallel and distributed computing to store and read information tuples concurrently. The tuple space is also called a blackboard. The service to access such a space provides methods like read and write. Commonly known examples are Linda and JavaSpace.
- ***Shared Files***: some operating systems offer the possibility to use a shared file to exchange information. Unix for instance uses the notation of pipes "|" (figure 4.9). It mediates a shared file which can be accessed by a process using the FIFO protocol. The pipe implements the producer-consumer coordination process. One process produces an output whereas another process consumes the produced information.

### 4.10.4. Programming Languages

We have seen that the elements of a programming language are entities. The programming language also helps to express communication between different entities. The simplest way is to use the assign operator [=]. For instance x=y; denotes that y communicates with x. In C and Java the semantic meaning is sending the value of y to x (Table 4.1).

Another way to communicate is by calling functions. If a function $f$ is called within a function $g$, then $g$ communicates with $f$ (Table 4.2).

A higher level of communication in programming languages handles events. Events are often sent by the operating system or by the middleware and are handled by program modules. For instance the Java platform offers various action listeners to capture mouse and keyboard events. Unix implements a similar approach using signals. Before handling an event or signal a specific handler (function or object) must be installed.

Some programming platforms such as QT by Trolltech [BS06], use Event-Source and Slot techniques. For instance the value source of a trackbar component can be graphically

| ***Notation*** | ***Meaning*** |
|---|---|
| f(x↓); | syntactic: call of function f and passing x as input parameter.<br>semantic: f receives the content of x. |
| f(x↓, y↑) | syntactic: call of function f and passing x as input parameter and receives y as result.<br>semantic: f receives the content of x and returns the context for y. |
| f(x↓ ↑); | syntactic: call of function f and passing x as input/output parameter<br>semantic: f receives the content of x and returns the context for x. The semantic might change depending on the language implementation. |

Table 4.2.: Communication by calling functions

linked with a value slot of a label. The trackbar then communicates directly with the label by sending its current state.

# 5 Coordination Language



## 5.1. Introduction

A coordination language is the materialization or the linguistic embodiment of a coordination model [GC92]. It offers syntactical meaning to implement an application using that coordination model. Gelernter and Carriero stated in [GC92] that a coordination language is orthogonal to the computational language by extending the coordination part of the application. They developed a framework called LINDA. It is an extension to various programming languages intended to be used for process creation, communication and

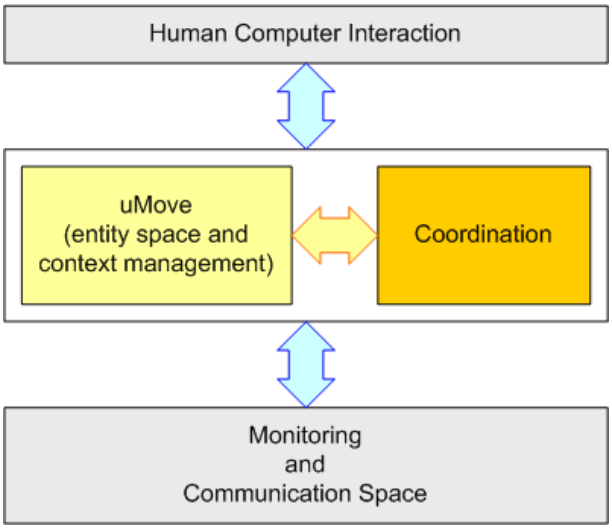



Figure 5.1.: The coordination and context management are essential components in the middleware for pervasive computing.

synchronization. LINDA is very generic and can be used for parallel computing as well as for distributed computing.

The coordination language we propose is based on the pervasive coordination model. This language provides an extension to the Java programming language in form of a Java based library. The main difference between it and LINDA is that the coordination is based on the context and activities of the participating entities and the given rules of their environment. In other words the concept of LINDA is part of the pervasive coordination language which is enriched by the new paradigm of pervasive computing.

This chapter is organized as follows: First a pervasive middleware is presented and an overview on the different software components of the middleware is given. Then the internal representation of the observed entities is explained. This includes the definition of entities, ports and observers. Afterwards, we focus on the coordination of communication. A generic message passing protocol is presented together with a light version of the pervasive services. This chapter concludes with the definition and programming of rules and actions.

## 5.2. Pervasive Computing Middleware

Since our main focus is the scientific field of pervasive computing, we will first introduce the architecture of a pervasive middleware. It is used as a framework to develop pervasive applications. Figure 5.1 shows the different components of the middleware and how they are connected to each other.

- ***HCI***: the human computer interaction components define patterns for interacting with a pervasive system
- ***uMove***: the uMove framework helps to organize and store contextual information for observed entities

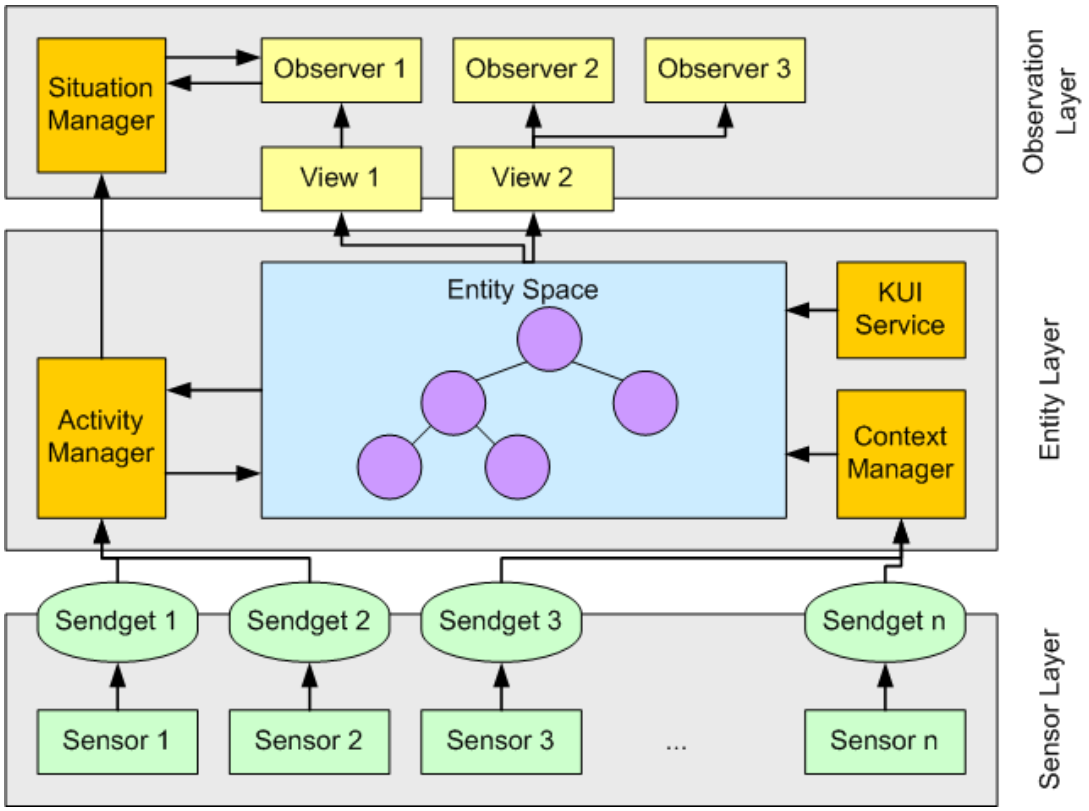



Figure 5.2.: The uMove Framework is devided in three main layers: sensor, entity and observation.

- ***Coordination***: the coordination component manages the communication between different entities. It creates communication channels and provides services to them according the context and the activity.
- ***Monitoring***: the monitoring helps to locate devices in a pervasive environment
- ***Communication Space***: the communication space offers a space to exchange information between different systems

uMove and the coordination component together provide the complete coordination language. The architecture of the coordination language is explained in the following sections.

## 5.3. Representation of the World

The representation of the world by entities (physical and virtual) is a part of the coordination language. We call this representation the entity space. In our implementation we separate the *entity space* from rest of the coordination. The idea is that the coordination component can be put next to any entity space. For pervasive computing we use the entity space implementation of the uMove framework [BH09].

The framework uses a layered architecture to separate sensors, entities and observation processing (figure 5.2). The sensor layer implements the view of the pervasive system in

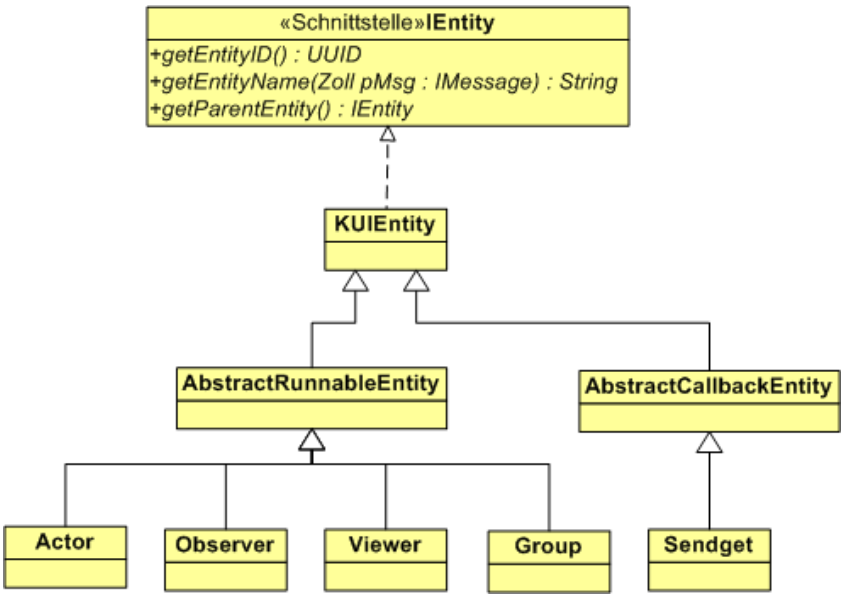



Figure 5.3.: Entity classes available in uMove framework.

the physical world. Through this view the system gathers information about the physical entities. The sensor gadget (Sendget) treats the raw data retrieved from the sensors and converts it into contextual information for the associated entities. The Sendgets help to hide the heterogeneity of the sensors. The entity layer on top becomes independent of the different sensor implementations.

The main component of the entity layer is the entity space. The entity space is used as an internal storage for all observed entities. Different managers help to create and update the entity space. For instance the activity management helps to detect the activity from the motion. The KUI Service is used to integrate and update mobile entities. It allows to connect distributed entity spaces together.

The observation layer contains various internal observers. Those observers are used by the software application on top of the middleware to observe parts of the entity space. Each observer has a specific view of the entity tree. If a context or an activity change has been detected the observer reevaluates the situation of the entities and reports the new situation to the upper level.

### 5.3.1. Entity

In the current version of uMove there exist several entity classes. The system engineer can extend the actual entity classes if needed.

The figure 5.3 shows the entity class diagram:

- ***IEntity***: the IEntity defines the generic entity interface
  - *ID*: an entity must provide a unique ID. The universally unique identifier (UUID) of Java is used to be sure the ID exists only once in the universe.
  - *Parent*: the entity parent gives the location of the entity. For physical entities the location is always a physical entity. The universe is represented by *null*.

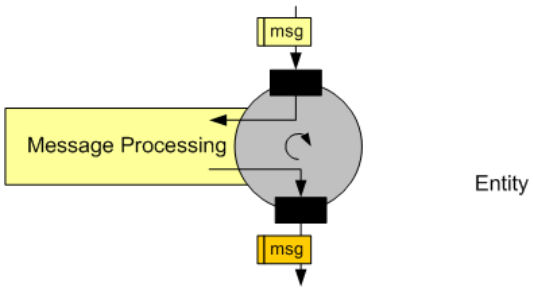



Figure 5.4.: The message processor treats the incoming messages and generates new messages for other entities.

  - *Name*: the name is a descriptive parameter only, used by humans to identify an entity
- ***KUIEntity, AbstractRunnableEntity and AbstractCallbackEntity***: are abstract classes implementing the basic behavior of the different entity types. The abstract runnable entity is also called *an agent*.
  - *MessageProcessor*: the message processor contains the computational part of an entity
- ***Actor***: the actor represents an observed physical entity e.g. place, human, computer
  - *Identity*: used to identify the actors. It includes entity ID, RFID tags etc.
  - *Context*: storage of the contextual information
  - *Activity*: current activity of the actor
  - *Location*: physical and logical locations
- ***Observer***: the observer entity is used to observe parts of the world. It is capable of evaluating situations of entities.
  - *SituationManager*: the situation manager treats the context and activity changes of actors and reevaluates their situation status
- ***Viewer***: the observer entity uses a viewer to observe parts of the world. The viewer is used as a filter.
- ***Sendget***: the sensor gadget class used to integrate sensors into uMove

A runnable entity has two major ports in general to communicate with other entities (an input and an output port). A message processor can be attached to each runnable entity and it contains the algorithms to treat the incomming messages (figure 5.4). New messages are generated and handed back to the entity, which sends them over the output port.

uMove comes with standard message processing, but the message processor can be easily extended or replaced by specifically developped message processing.

### 5.3.2. KUI System

The uMove framework provides an extensive API to handle the system components. The KUI class provides methods for:

- Creating actors, observers, viewers and physical or logical zones
- Creating sensors and sendgets for various purposes like location, contextual information or motion detection
- Starting and stopping a KUI service

```
{
  KUISystem pSystem = new KUISystem();
  Actor pPlace1 = pSystem.createPhysicalActor("Place1", null);
  Actor pActor1 = pSystem.createPhysicalActor("ctor", pPlace);

  Viewer pView1 = pSystem.createViewer(pPlace1, 4);
  Observer pObserver = pSystem.createObserver(pView1,
      new MySituationManager());
  ...
}
```

Listing 5.1: How to create entities using the KUI system

### 5.3.3. Ports

The concept of port is used for communication between entities. As explained in the generic model ports are the interface between the computational and the coordination parts of the programming language. They are described by port descriptors. The descriptor contains all relevant properties of a port such as:

- ***ID***: a unique identifier
- ***Name***: the name of the port used for port matching
- ***Orientation***: input, output or in-out
- ***Synchronization type***: asynchronous or synchronous communication
- ***Delivery protocol***: FIFO, causal or total ordering of messages
- ***Publicity***: flag if the port is public or private
- ***Address***: some of the public ports need an additional address e.g. TCP-based ports

Attributes are used to match two ports and create couplings between them. Once a coupling is created a communication channel is implicitly set and the connected entities are able to exchange information.

The basic communication concept implements the writing and reading of messages (figure 5.5). The reading and writing depend on the port orientation: input, output or in-out. The input only allows reading of messages sent by other entities. The output restricts the port behavior in the opposite way. The port is only able to send messages to others. The in-out orientation allows communication in both directions.

The coordination language provides several interfaces and classes to create, access and manipulate ports.

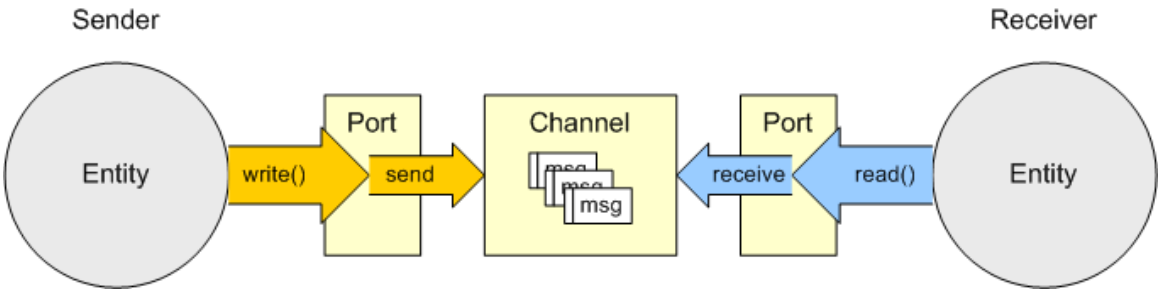



Figure 5.5.: Shows the sending and receiving part of the communication using ports.

### Interface - IPort

The interface *IPort* is used to hide the different port implementations from the computation. Any computation thread can use this interface to exchange information with other entities. The interface defines the following methods:

- ***peek()***: reads a message from the communication channel without removing it. The method does not block and returns null if no message is available.
- ***read()***: reads and removes a message from the communication channel. The method does not block and returns null if no message is available.
- ***readBlocked()***: reads and removes a message from the communication channel. The blocked reading will wait until a message becomes available or aborts if the channel is closed.
- ***write()***: sends a message to another entity
- ***open()***: opens a port explicitly. In general ports are opened implicitly by the coordination manager. The open method takes the communication channel as an input parameter.
- ***close()***: explicitly closes the connected channels. This method is implicitly called by the coordination manager if the communication between two ports ends.

### Port Classes

The class diagram (figure 5.6) shows how the ports are defined. The current coordination language provides only asynchronous ports. There exist several concrete port classes:

- ***InputPort***: port to only read messages
- ***OutputPort***: port to only write messages
- ***InOutPort***: port to read and write messages

The existing set of ports can be extended by adding new ports with different behaviors. For instance, synchronous ports can be implemented by extending the abstract port class *TAbstractPort*. The following list shows how the interface can be used by the computational part of the application.

```
1 class MyAgent extends Thread
  {
3   private InputPort m_pInPort = new InputPort();
```

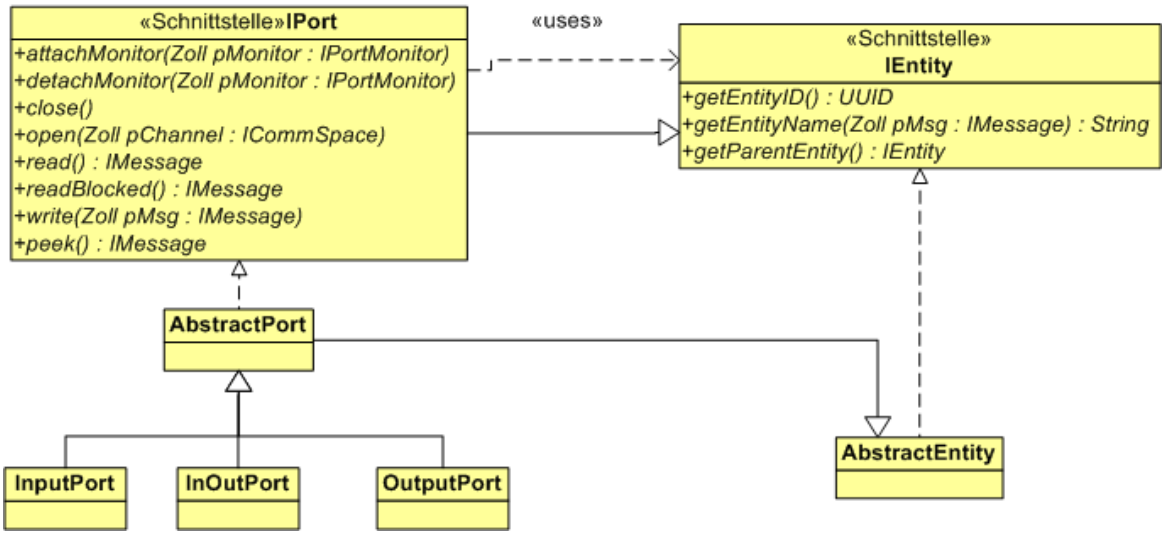



Figure 5.6.: Port implementation of the coordination library.

```
  private OutputPort m_pOutPort = new OutputPort();
5
  public void run()
7 {
    IMessage pMessage = pInPort.read();
9   ...
    pOutPort.write(new TextMessage("Hello World"));
11 }
}
```

Listing 5.2: Creating ports within an entity.

## 5.4. Coordination Component

Communication is the main activity between software entities. Therefore the goal of the coordination component is to manage the communication between entities. It takes care of the mediation between communication channels. Channels are the managed dependency and they are created by matching the ports of entities. The coordination component implements the different coordination processes needed for the desired information exchange between the entities.

The central object of the coordination component is the *coordination manager* (CM) (figure 5.7). The manager implements the main API of the coordination layer. The API allows the coordination to be used in an explicit way. The API is used to:

- find ports and entities within the system
- create port couplings using the matching algorithm
- remove and clean unused port couplings
- start and stop services
- evaluate rules

Three databases are used inside the coordination component. The first is an *entity-registry* used to find an entity by its ID. The entities automatically register themselves here when

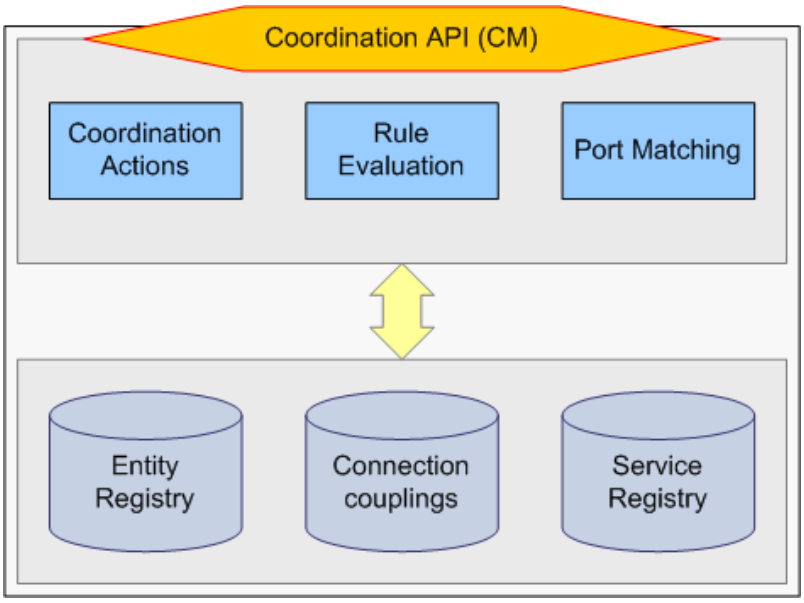



Figure 5.7.: Architecture of the coordination component.

they are created by the system. The second database stores the *connection couplings*, which includes the two ports connected to each other and the mediated communication channel. The last database is used to manage the public ports (services). It keeps track of which services are enabled or disabled for specific entities.

In addition to the database layer three components help to manage and coordinate the data stored in the databases.

- Port Matching helps to match ports and to create new port couplings
- Rule Evaluation is used to check the rules and generate actions (section 5.6)
- A set of predefined coordination actions to coordinate the system (section 5.6)

### 5.4.1. Port Matching

The port matching process checks if two ports are compatible for communication. As described in the generic coordination model (section 10) the matching is based on primary and secondary port features.

The following matching criteria are implemented in the current version of the coordination language:

- The orientations must be inverse for *in* and *out*, or equals to *in-out*
- The names must be equal
- The communication structures (peer-to-peer, broadcast, etc.) and the synchronization types (synchronous or asynchronous) must match exactly
- There must be a free connection to the port. Some ports allow only one channel to be connected, others might allow multiple channels.
- The protocols must be compatible

The port matching also decides the type of channel to be created in order to meet the communication goals. The current version provides only two different types of communication channels:

- ***LocalQueueSpace***: a locally installed FIFO queue. Typically a shared memory or file is used to store the messages.
- ***RemoteQueuedSpace***: socket based communication channel using the Transmission Control Protocol TCP. Since TCP is reliable in terms of out-of-order delivery, it is used as a FIFO queue between remote systems.

In the future the coordination language can be extended with other communication channels, implementing other delivery protocols like priority sending, causal order or total order.

### 5.4.2. Message Passing Protocol

We have seen that the communication between entities is realized through channels. Each channel implements its own strategy to pass messages and to organize the communication. This section explains the message passing protocol and the different strategies for passing messages between entities.

The message passing protocol is a generic communication protocol used to exchange messages between two or more entities. There exist two main classes of message passing:

- ***Local message passing***: local passing is very effective for parallel and local computing because it minimizes the thread synchronization and uses local memory to store messages
- ***Remote message passing***: remote message passing allows the exchange of messages between distributed computing systems through a computer network

Remote and local message passing are fully transparent when communicating through ports.

#### Message

The message is the basic object to transmit information. A message contains a header and a body. The header includes various fields, like:

- ***Sender ID***: identity of the message sender
- ***Recipient ID***: list of the recipient ID's used for identified communication
- ***Time stamp***: a logical time stamp to check which message is sent before another one. Only the partial ordering is implemented using Lamport clocks.
- ***Properties***: list of customer properties. A property is made up of a key-value pair. The number of pairs is unlimited.

The body is variable and is defined by the concrete message classes. The coordination library provides the interface *IMessage* and an *AbstractMessage* class which implements the header part (figure 5.8). A few message classes are offered by the library:

- ***TextMessage***: the body is a single string containing textual information

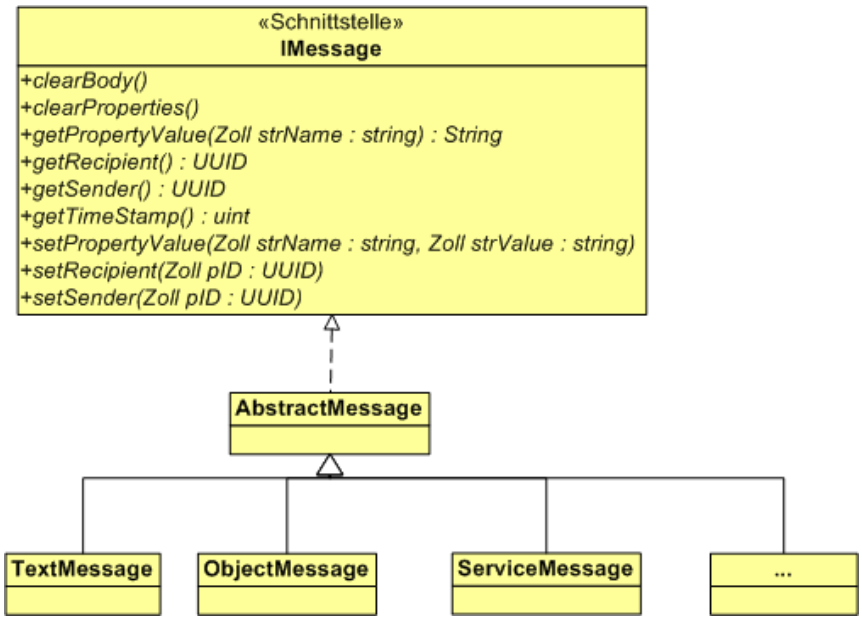



Figure 5.8.: Messages classes of the message passing protocol.

- ***ObjectMessage***: the body contains a streamable object

The object message must be streamable (including all objects added to the body). This ensures that the message can be sent through all types of channels, in particular channels to remote systems which often only support byte streaming.

### Port Coupling

There are two main strategies for passing messages between entities:

- Anonymous message passing
- Identified message passing

For anonymous message passing the entity sends its message throughout the output port not knowing who is getting the message. The simplest way to do this is to install a permanent coupling. The strategy is to create and manage a communication infrastructure using the computational part or by evaluation rules. The recipient header field of the message is not needed and can be set to null. The big advantage is that the entities don't have to know each other.

Identified message passing is more dynamic. Before sending a message the entity puts all recipients ID's into the recipient header field of the message. As soon as the message is sent through the output port a communication structure is created temporarily. The necessary couplings are created immediately between the output port of the sender and the input ports of recipients. After sending the message the coupling is released and the communication channel removed.

The current version supports only permanent and temporary communication channels. In future versions the notion of expiration could give more flexibility to the use of the communication infrastructure.

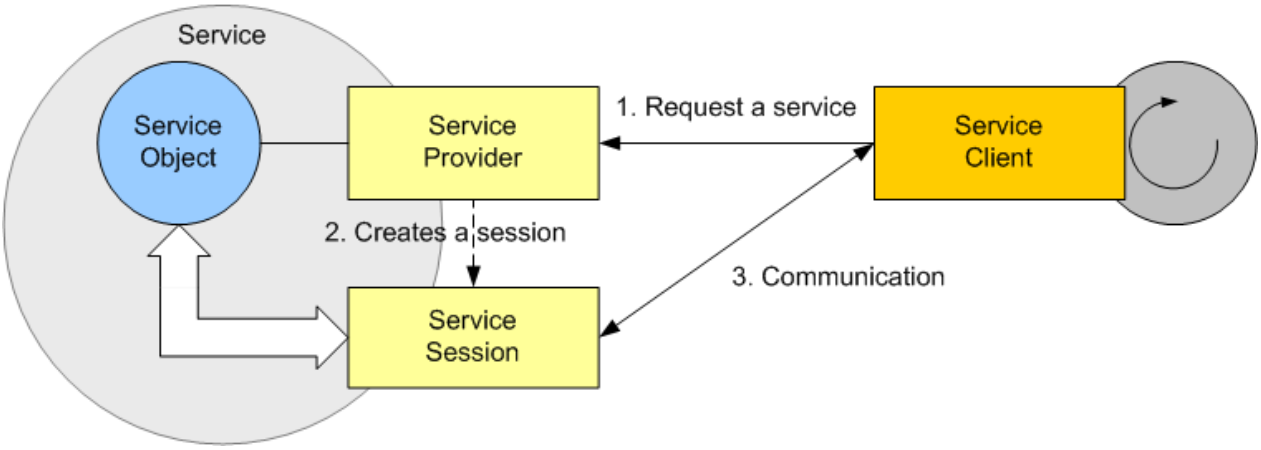



Figure 5.9.: Layout of a public service provider.

## 5.5. Service

Services are realized as a composition of a *service object* and a public port which provides the service to outstanding entities. This public port is called *a service provider* and allows remote entities to communicate with the service.

### 5.5.1. Service Provider

A service provider is a specialized public port which is dedicated to providing services to other entities. The service provider listens to a public port for incoming requests (figure 5.9). If a client requests a service (1) and is accepted by the service provider port, a new session is created (2). The service session is private and controls the communication between the service client and the service object (3). The current coordination language defines only TCP-based service provider ports.

### 5.5.2. Service Client

The *service client port* is the counterpart of the service provider. A client port must be attached as a public port to an actor. The actor puts the client port into its port list. Once the matching process of the coordination manager matches all of the actors available ports, the clients are implicitly connected to the available services. The actor is normally not used to exchange data through the client port. Dedicated applications on top of the middleware access the client port and exchange their data and requests with the service (figure 5.10).

But how can a client find services in a smart environment? There exist different techniques for finding a service within an environment. The pervasive middleware provides a monitoring component (figure 5.1). The monitoring helps to detect mobile entities within an environment. Once a mobile device is detected and accepted it will periodically receive a list of available services. Each time a new service is provided to the mobile device the coordination manager tries to find a matching client. If a service disappears the client is automatically disconnected.

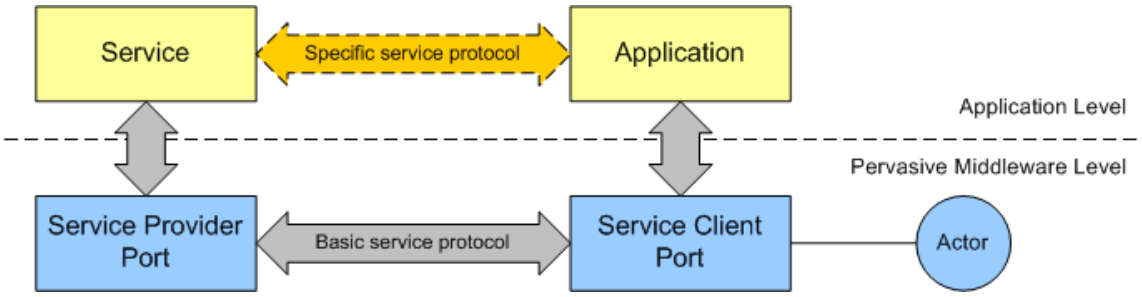



Figure 5.10.: Service architecture

The list of available services for each entity is received from the service registry. The registry keeps track of all enabled services for each entity. Services are enabled and disabled using the methods *enableService()* and *disableService()* of the ServiceRegistry singleton.

A lean monitoring example is explained through the case study in the chapter 6.

### 5.5.3. Service Message Protocol

Each service can use its own message protocol. The protocol governs the data exchange between the service object and a service client application. We distinguish between two different protocols. The first is called *the basic service protocol*. It implements the basic rules for service communication. More details about the insight of the basic protocol can be found in the appendix of this document. The second and more interesting protocol is called *the specific service protocol*. This protocol must be defined by the programmer of the service and controls the data exchange between the serverside service and the client application (figure 5.10).

### 5.5.4. Creating a Service

A service object must implement the interface *IService*. The interface defines the method *process()*. The incomming message and the querying type are passed as arguments. Following querying types are defined:

- ***etRead***: reads data from the service object
- ***etReadBlocked***: reads data from the service object. The call should be blocked until data is available.
- ***etPeek***: peek of data instead of reading
- ***etWrite***: writes data to the service object
- ***etWriteRead***: writes data to the service first and then reads a result

The following example shows the simple HelloService.

```
import ch.unifr.coordination.structure.interfaces.IService;
import ch.unifr.coordination.structure.interfaces.EServiceQueryType;

public class HelloService implements IService
{
  @Override
```

```
  public IMessage process(IMessage pMessage, EServiceQueryType eQueryType)
  {
    return new TextMessage("Hello world!");
  }
}
```

Listing 5.3: Creating a service

A service can be started using the coordination manager. It is automatically registered by the coordination manager.

```
import ch.unifr.coordination.CoordinationManager;
...
{
  CoordinationManager.getInstance().startTCPBasedServer("MeetingService",
                                                        9901,
                                                        new MeetingService());
  ...
}
```

Listing 5.4: Starting a TCP based server

On the other side a matching client must be developed. The client must implement the *IServiceClient* interface. A service client must be attached to a physical entity.

```
...
{
  MeetingServiceClient pMeetingClient = new MeetingServiceClient(new PortDescriptor(
    UUID.randomUUID(),
    "MeetingService", ""));

  m_pActorEntity.attachPort(pMenuClient);
  ...
}
```

Listing 5.5: Create a service client port attached to an entity

It is also possible to manually connect a client to a service. For instance the *TCBBased-ServiceClient* class provides a static method to create and connect a client directly to the service:

```
...
{
  PortDescriptor pDescriptor = new PortDescriptor(UUID.randomUUID(), "MyClient", "
      10.1.0.1:9000");

  TCPBasedServiceClient pClient =
    TCPBasedServiceClient.createConnectedClient(pDescriptor, pActor);
}
```

Listing 5.6: Connect a service client manually

More details about the service architecture and the implementation are put in the appendix section A.1 of this document.

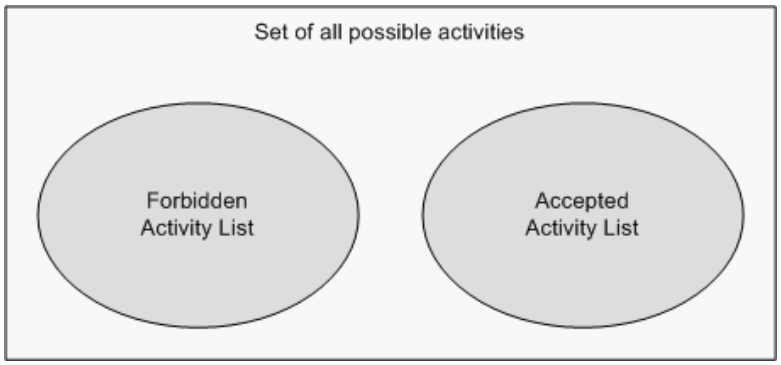



Figure 5.11.: Forbidden and accepted list concept of activity rules.

## 5.6. Rule

Rules are an important concept in coordination. They often logically describe certain circumstances and conditions and help to govern the system by generating actions. Rules are needed for various tasks. For instance they help to check if an activity is allowed or forbidden within an environment. We call this type of rule *an activity rule*. Rules also help to to analyze the situations of entities. *Situation rules* help to generate complex situations out of the context and the activity of an entity. For coordination the *social rules* are the most important ones. They help to govern the social and interactional behavior of entities.

### 5.6.1. Activity Rule

Activities are evaluated using rules. The activity of an actor is checked by the activity rules of the actor's parent (place).

There exist three statuses for activities: accepted, forbidden and negotiable activities [BH09]. The status is stored within the recognized activity. The activity rules are defined at the level of the actors. Each actor contains a list of all accepted activities and a list of all forbidden activities (figure 5.11). All activities which are neither one of the two lists are stated as negotiable activities.

### 5.6.2. Situation Rule

Situation rules check the activity and the context of an actor and evaluate its situation. Since the situation depends on the point of view of an observer the situation status is evaluated by the observer. A situation manager can be attached to each observer. Each application observing the entity space must provide a dedicated situation manager implementing the reasoning on situations [Lok04]. The manager takes care of the evaluation and contains all situation rules necessary to produce the situation status.

Situation rules are not directly used by the coordination management, but help the pervasive application react to critical states. The situation status helps a pervasive application be as unobtrusive as possible. This behavoir is also called calm computing [Wei91]. The current language knows three situation states:

- ***Normal***: the situation of an entity is normal. No necessity to bother the entity (calm and unobtrusive).
- ***Potentially Dangerous***: the situation of an entity might be dangerous. The pervasive system might warn the entity.
- ***Critical***: the situation of an entity is critical. The pervasive system notifies the entity and might also inform other entities.

### Situation Manager

A situation manager must implement the interface *SituationManager* which defines one method called *checkSituation()*. This method must return the situation status of an entity. The situation status is an object containing the level (normal, critical or dangerous) and description of the situation.

```
{
import ch.unifr.umove.observation.situation.SituationManager;
import ch.unifr.umove.observation.situation.SituationStatus;
...
public class MySituationManager implements SituationManager
{
  @Override
  public SituationStatus checkSituation(Actor pActor, Actor pPlace)
  {
    SituationStatus pRetval;

    //Implement the rules here
    if (...)
    {
      pRetval = new SituationStatus(SituationLevel.CRITICAL, "Is in danger");
    }
    ...

    return pRetval;
  }
}
```

Listing 5.7: Creating a situation manager

The situation manager must be directly attached to the observer.

```
{
  KUISystem pSystem = new KUISystem();
  ...
  Observer pObserver = pSystem.createObserver(pView1,
      new MySituationManager());
  ...
}
```

Listing 5.8: Creating an observer and attaching the situation manager

## 5.6.3. Social Rules

The social rules control the interaction and social relations between entities. The rules are checked by the entity if the context or activity of the entity has been changed.

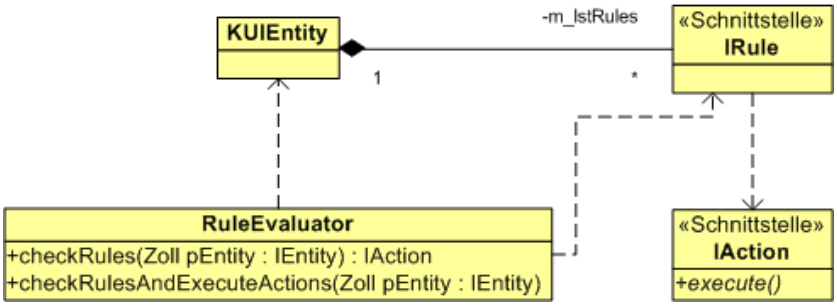



Figure 5.12.: Rule evaluation using rule and action interfaces.

The coordination language implements the social rule model proposed by the generic coordination model:

- Sub-entities inherit the rules from their parent entities
- Sub-entities can overwrite and extend the rules given by parent entities
- The evaluation checks the rules and controls the system

The rules are evaluated by the *RuleEvaluator*. The evaluation generates a list of actions which are the consequences of violated rules (figure 5.12). The actions typically perform coordination work within the system such as:

- ***Controlling the interaction***: for instance if a human enters a building equipped with a smart environment the human automatically gets access to provided services. The interaction can also be terminated. For instance if an employee enters a meeting room, depending on the rules some services like SMS are blocked.
- ***Managing social groups***: social rules also help to create or change social groups

The current version of the coordination language evaluates the rules on the level of actors and directly executes the actions afterwards. In the future the rules must be evaluated on the observer level. The actions should not directly be executed because the observer might negotiate the actions with other observers (chapter 7). This could lead to a more consistent system if two or more observers are evaluating rules and produce contradictory actions.

The concept of rules and actions is very powerfull. It allows to extend and customize the system for any needs. How rules and actions are defined and added to the environment is shown in the following sections.

### Rule Classes

The rules must be defined by the system engineer. The rule classes have to implement the IRule interface, which provides the following methods:

- ***check()***: checks if the rule is respected. It must return an action object if some changes must be done.
- ***getName()***: returns the name of the rule. The name is used to identify the rule.
- ***isFinal()***: if a rule is final it cannot be overwritten by sub-entities

The inheritance of rules is done via their names (id). If an entity $e_{n-1}$ is the parent entity of $e_n$, the rules for the sub-entity $e_n$ are merged in the following way:

- Let $R_{n-1}$ be the set of rules of $e_{n-1}$ and $R_n$ be the set of rules of $e_n$.
- Then the rule merging $\uplus$ of the two sets $R_{n-1}$ and $R_n$ is defined as

$$R_{n-1} \uplus R_n = R_{xy}(R_{n-1}, R_n) \cup R_x(R_{n-1}, R_n) \cup R_y(R_{n-1}, R_n) \tag{5.1}$$

where as $R_{xy}$ returns the set of all rules which occur in the set $R_{n-1}$ and $R_n$ but don not have the same identity:

$$R_{xy}(R_{n-1}, R_n) = \{x \in R_{n-1}; y \in R_n : id(x) \neq id(y)\} \tag{5.2}$$

$R_x$ returns all rules from the parent entity $e_{n-1}$ which are final and can not be overwritten by the sub-entity:

$$R_x(R_{n-1}, R_n) = \{x \in R_{n-1} : \exists y \in R_n : id(x) = id(y) \wedge isFinal(x)\} \tag{5.3}$$

and $R_y$ returns all not final rules which are overwritten by the sub-entity $e_n$

$$R_y(R_{n-1}, R_n) = \{y \in R_n : \exists x \in R_{n-1} : id(x) = id(y) \wedge \neg isFinal(x)\} \tag{5.4}$$

The merging operator $\uplus$ is applied to all ancestors of $e_n$. The merging starts with the root entity $e_0$ and the first ancestor $e_1$. The result is then merged again with the second ancestor $e_2$ and so on, until the level of $e_n$ is reached. At the end a completly merged rule set $R_{merged}$ to evaluate $e_n$ is received. This set is then passed to the rule evaluator.

$$R_{merged} = (((R_0 \uplus R_1) \uplus R_2) ... \uplus R_n) \tag{5.5}$$

### Action Classes

All actions must implement the *IAction* interface. The interface provides only one method called execute(). The coordination languages proposes few actions, like:

- ***EnableServiceAction***: enables a service for the entity. The entity is capable of connecting to a service provider port.
- ***DisableServiceAction***: the entity is disconnected from a service

Nevertheless the system engineer can design new actions in order to coordinate the system. The coordination language can be extended and customized easily in this way.

### Implementing a customized Rule

The following example illustrates how a customized rule is implemented. The rule checks if the temperature of an entity is lower than zero. If this is the case the service *MyService* is disabled. Otherwise the service is enabled. The two actions are provided by the coordination component and dedicated to enable or disable services.

```
{
2 import ch.unifr.coordination.rules.interfaces.IRule;
```

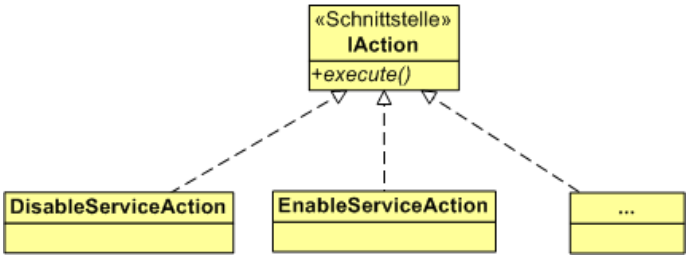



Figure 5.13.: Action classes

```
import ch.unifr.coordination.actions.DisableServiceAction;
import ch.unifr.coordination.actions.EnableServiceAction;
import ch.unifr.umove.espace.actor.Actor;
import ch.unifr.umove.espace.contexts.Temperature;
...

public class MyRule implements IRule
{
  @Override
  public IAction check(IEntity pEntity)
  {
    IAction pRetval = null;
    if (pEntity instanceof Actor)
    {
      //Evaluate the context and activity and generate a rule
      Actor pParent = (Actor) pActor.getParentEntity();
      Temperature pTemperature = (Temperature) pActor.getContexts().getContext(
        "Temperature");

      if (pTemperature == null || pTemperature.getTemperature() < 0)
      {
        pRetval = new DisableServiceAction(pEntity, "MyService");
      }
      else
      {
        pRetval = new EnableServiceAction(pEntity, "MyService");
      }
    }
    return pRetval;
  }

  @Override
  public String getName()
  {
    return "MyRule";
  }

  @Override
  public boolean isFinal()
  {
    return false;
  }
}
}
```

Listing 5.9: Creating a customized rule

The rule is attached to the environment. All entities staying within this environment have to follow this rule afterwards.

```
{
  ...
  KUISystem pSystem = new KUISystem();
  Actor pPlace1 = pSystem.createPhysicalActor("Place1", null);
```

```
5
   //Add all rules here
7  pPlace1.addRule(new MyRule());
   ...
9 }
```

Listing 5.10: Attaching a rule to an environment

### Implementing a customized Action

The collection of actions can be customized too. The next example shows how an action can be implemented. The action class *MyAction* must implement the the *IAction* interface.

```
   {
2 import ch.unifr.coordination.actions.interfaces.IAction;
   import ch.unifr.coordination.structure.interfaces.IEntity;
4 ...

6 public class MyAction implements IAction
   {
8    public MyAction(IEntity pEntity)
     {
10     m_pEntity = pEntity;
     }
12
     @Override
14   public Retval execute()
     {
16     Retval retValue = Retval.RET_REQUEST_FAILED;
       //TODO: place the coordination code here
18     ...
       return retValue;
20   }

22   private IEntity m_pEntity;
   }
24 }
```

Listing 5.11: Creating a customized action

# 6

# Case Study: Tracking of Entities



## 6.1. Introduction

In pervasive computing it is essential to track mobile entities in order to update their contextual information and to exchange information between server and mobile applications. This case study shows how the tracking can be done using the uMove framework together with the coordination module. The focus is to integrate mobile entities into a server based entity space which provides pervasive services to them (smart environment). A mobile entity is typically a human equipped with a mobile device.

In a concrete example we show that a mobile device is able to get two services from the smart environment:

- ***Meeting service***: the meeting room is equipped with a meeting service which provides information about current and future meetings. As soon as a human enters the meeting room he will receive the information on his mobile device.
- ***Menu service***: the menu service provides information about the current menus served in the cafeteria. If a human is entering the cafeteria he will receive the menu, but only if he is not running.

Those two services are coordinated using simple rules. Whereas the meeting service depends only on the location of a human, the menu service takes also activities into consideration.

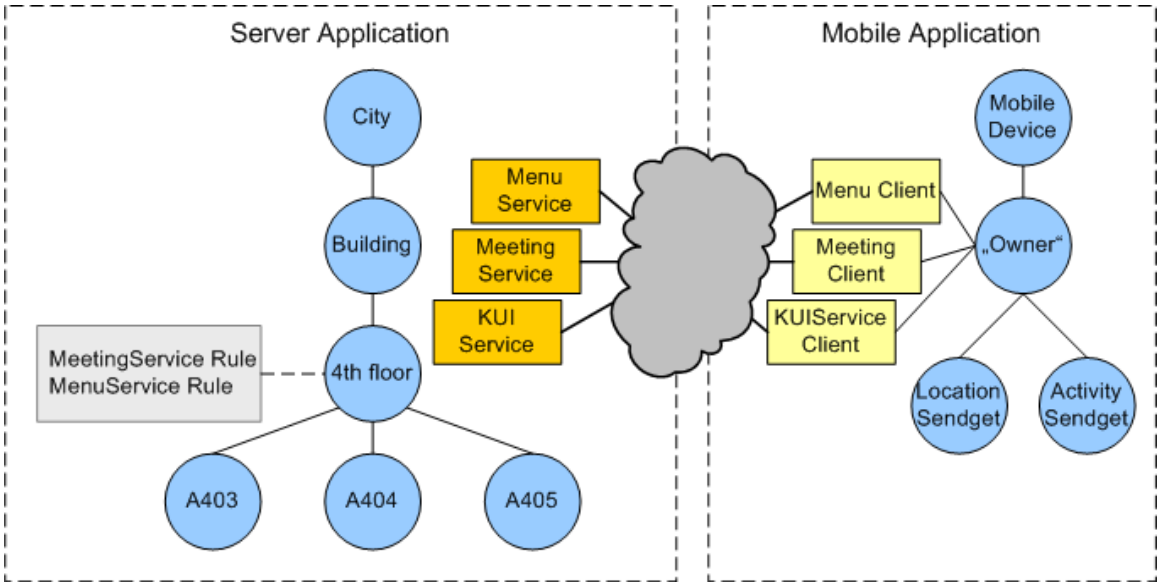



Figure 6.1.: Setup of server and mobile application

## 6.2. Layout of the System

The implementation of the tracking system is split into two applications, the server and the mobile application (figure 6.1).

- ***Server application***: the server system enriches an environment with pervasive services. This environment is called *a smart environment*.
- ***Mobile application***: the mobile application runs on a mobile device, using embedded sensors. It observes the human and provides contextual information to the smart environment.

Both systems are based on the pervasive middleware framework, especially the uMove framework to represent the world, the coordination component to manage the communication and the monitoring to detect mobile entities (human, mobile devices).

### 6.2.1. Server Application

We extended an existing pervasive application, called Robin [BLLH10] as a server application which was a case study to show how uMove is used to build activity based pervasive applications. The main extension is the implementation of the concept of services (figure 6.1). The server provides three services to mobile entities:

- ***KUIService***: is part of the uMove framework and allows mobile entities to be identified and integrated in the KUI system.
- ***MenuService***: the menu service is an application-specific service. It provides menu information from a cafeteria.
- ***MeetingService***: the meeting service is an application-specific service too. It provides information about the meeting schedule of a meeting room.

Two rules are attached to the floor to coordinate the services:

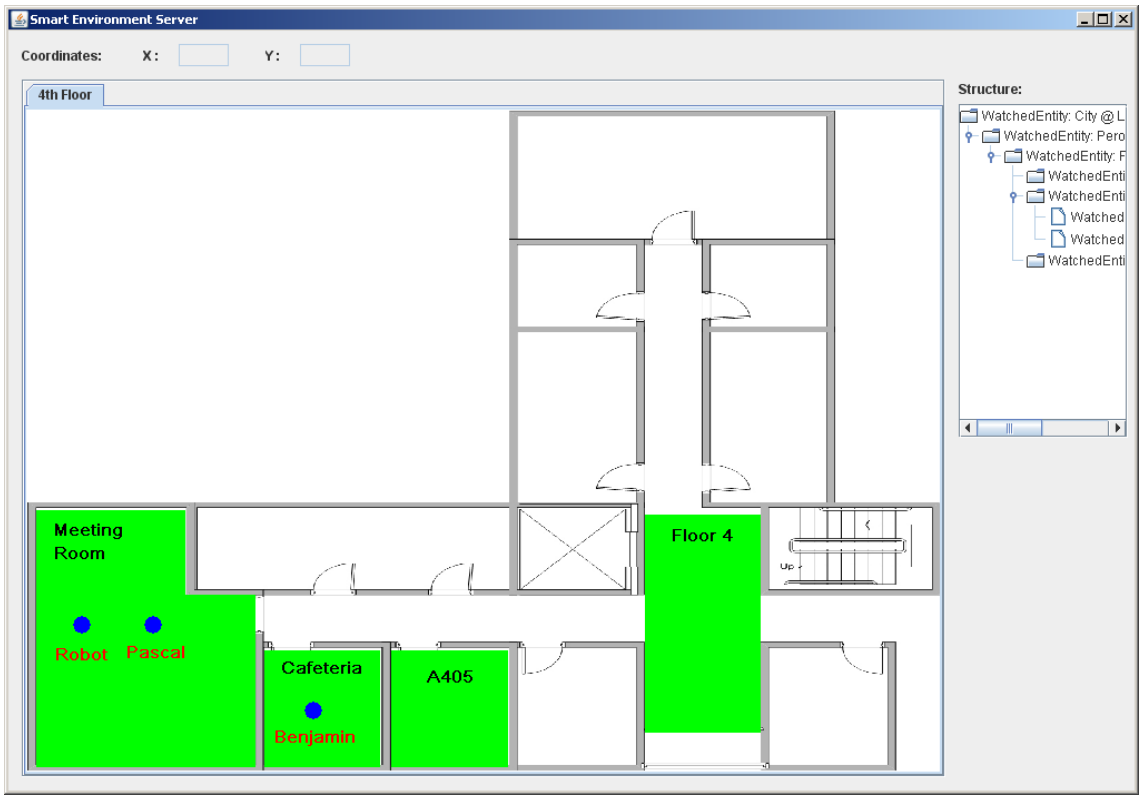


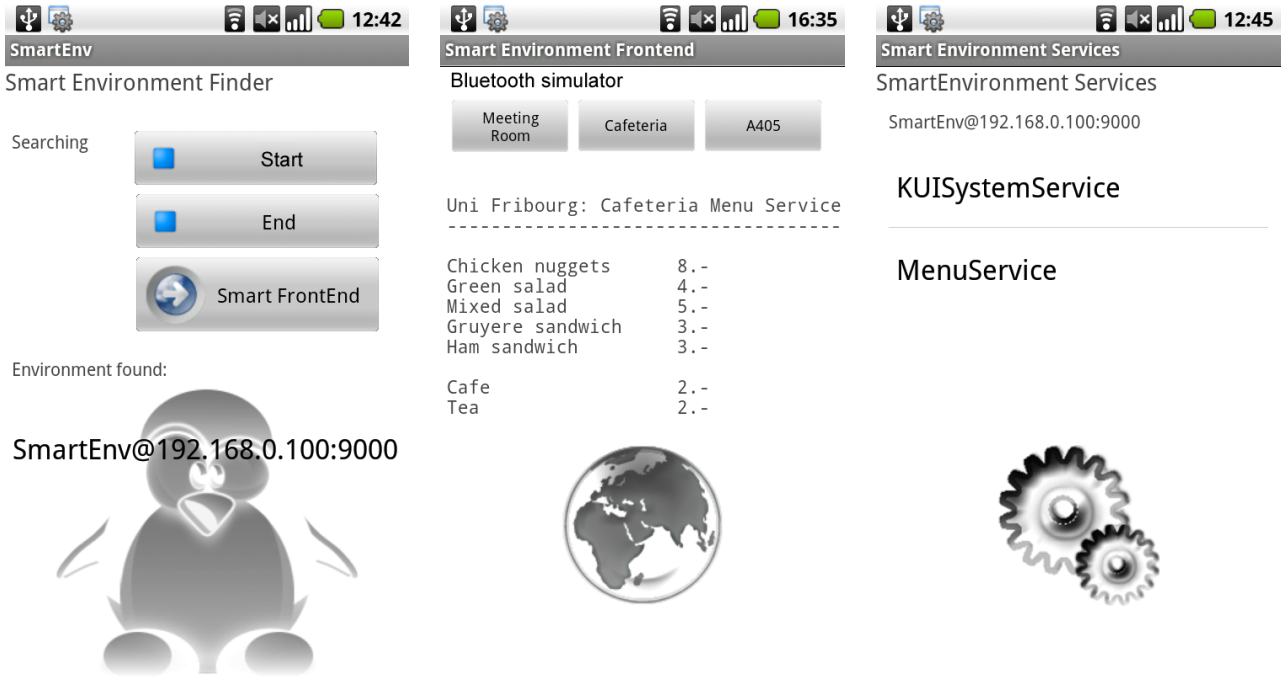

Figure 6.2.: Server application. The blue dots show currently tracked entities. The figure is modified to improve the printout.

- ***Meeting Service Rule***: this rule governs the meeting service. As soon as a mobile entity enters the meeting room, the service client is automatically connected to the meeting service. The mobile entity will receive the current list of meetings.
- ***Menu Service Rule***: this rule governs the menu service. The rule considers not only the location of a mobile entity but also heir activity. The mobile entity will receive the menu if it is inside the cafeteria and the activity of the human is walking. If the human is in a hurry (running) he will not receive any menu information.

No rules are needed for the KUI Service because this service should be always available. The server provides a graphical user interface which shows the layout of a building (figure 6.2). Three rooms are configured on the 4th floor: a Meeting Room (A403), a Cafeteria (A404) and an office A405. All tracked mobile entities are represented as dots at their current location.

### 6.2.2. Mobile Application

The mobile application runs on the mobile device. If the human waering the mobile device enters a smart environment, the application automatically connects to that environment. To be able to identify the mobile device in the server application, a special attention has been given to connection algorithm. It must be able to connect or reconnect the device when a server is retected and disconnect properly when the device is out of range (e.g. leaving the building).




a: b: c:

Figure 6.3.: The three main screens of the mobile application. The figures are modified to improve the printout.

We use the Motorola Milestone as mobile device. It is equipped with the Android OS and a Android Java Platform which is compatible with the Java Standard Edition. This allows us to develop a mobile application using the same pervasive middleware as we use for the server application. How to setup a mobile device with the pervasive middleware is explained in the appendix B.

We apply a two level scanning to handle the connectivity. The first level is the lookup for a smart environment using the monitoring component (section 6.3). Once a smart environment is found the client and server exchange service availability information. Coordination is then used to connect to new services becoming available or to disconnect services if they disappear. The tracking starts when a mobile entity enters the area covered by a smart environment. A login process starts. The mobile device first sends the login information such as the user name and the password (if needed). If the smart environment accepts the mobile device it sends the available services to the device. As long as the mobile device stays within the covered area it is tracked by the smart environment and gets updates of the available services.

A small graphical user interface has been also developed. The main screen shows the available smart environments (figure 6.3a). When the Start button is pressed the device starts scanning for smart environments. The Stop button can be used to explicitly disconnect from all smart environments. In regular operation the device disconnects implicitly from the environment if it is out of range, for instance when leaving the building.

The Smart FrontEnd button switches the screen to the front end used to receive information from services (figure 6.3b). Since the rooms are not equipped with Bluetooth dongles or RFID's we added a small Bluetooth simulator. The three buttons can be used to change the location between the three rooms. Below the small simulator a service message appears depending on the situation of the actor (context, location and activity).

Once a smart environment is detected it is shown on the main screen. When a smart environment is selected the service screen appears (figure 6.3c). This screen is used to check the availability of services and is mostly used for debugging or defining rules.

## 6.3. Monitoring

For the entity tracking a lean monitoring component had been created. The monitoring helps to connect mobile devices to the smart environment implicitly.

The scanning is the most essential part of the lean monitoring implementation. The mobile device broadcasts a ping message over a WiFi interface (figure 6.4a). The ping message contains the open listening port for any echos. If a smart environment is close by it responds to the ping (figure 6.4b). The echo message contains the public port address of the main service from the smart environment. This service is used to log in and integrate the mobile device into the smart environment. Once the device is accepted a permanent communication channel is installed (figure 6.4c). Through this channel the service provides information about the environment such as other available public services. On the other hand the mobile device can provide its mobile services to the smart environment as well enriching the environment with new functionalities. If the mobile device is out of range or disconnected on purpose, the connection is reset (figure 6.4d). Because the scanning of the mobile device is running permanently the mobile device is automatically reintegrated as soon as it is in the range of the smart environment again.

The mobile device is able to connect itself to multiple smart environments at the same time. This happens if two environment networks overlap in a certain area, like for instance if a city is equipped with a smart city environment and the user is close to a train station, which also has its own smart environment. In this case the mobile device uses some services provided by the city and some services provided by the train company.

The implementation and class diagrams are put in the appendix section A.2 of this document.

## 6.4. KUI Service

The KUI Service manages the integration of a mobile device into the entity space of the smart environment. This integration is realized by the creation of a stub entity which represents the actor of the mobile device in the server (smart environment) (figure 6.5).

If a new mobile entity enters the smart environment and connects to the KUI Service all relevant information is sent to the server. According the the service implementation

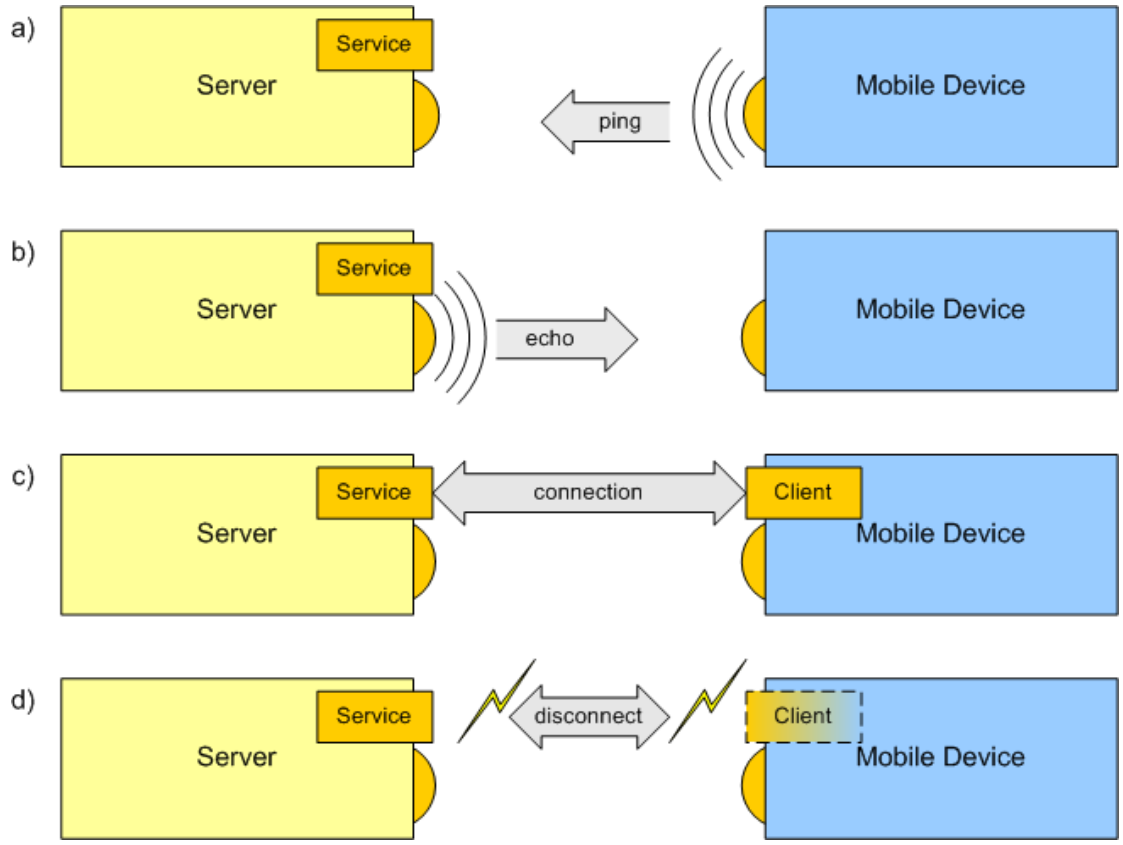



Figure 6.4.: Monitoring of mobile devices.

of the coordination library a private KUI Service session takes care of the client-server communication. Each time the actor receives new contextual information or changes activities the messages are also passed to the stub through the client-server connection.

If the mobile device leaves the smart environment the stub entity is removed from the entity space of the smart environment and all communication channels are disconnected.

This architecture allows the mobile device and the designated mobile applications to be independent of the smart environment. On the other hand the server applications can provide context-aware services to the mobile device. The stub also helps to reduce the communication between the smart environment and the mobile device. If a server application is querying information about the actor it communicates only with the stub and gets the context of the stub. The stub is fully transparent to the server application.

## 6.5. Message Flow

This section explains the two major message flow implementations between a mobile device and the server. The first example shows how a simple context change (e.g. temperature, motion) is managed. The second example illustrates the location change of a mobile entity within a smart environment.

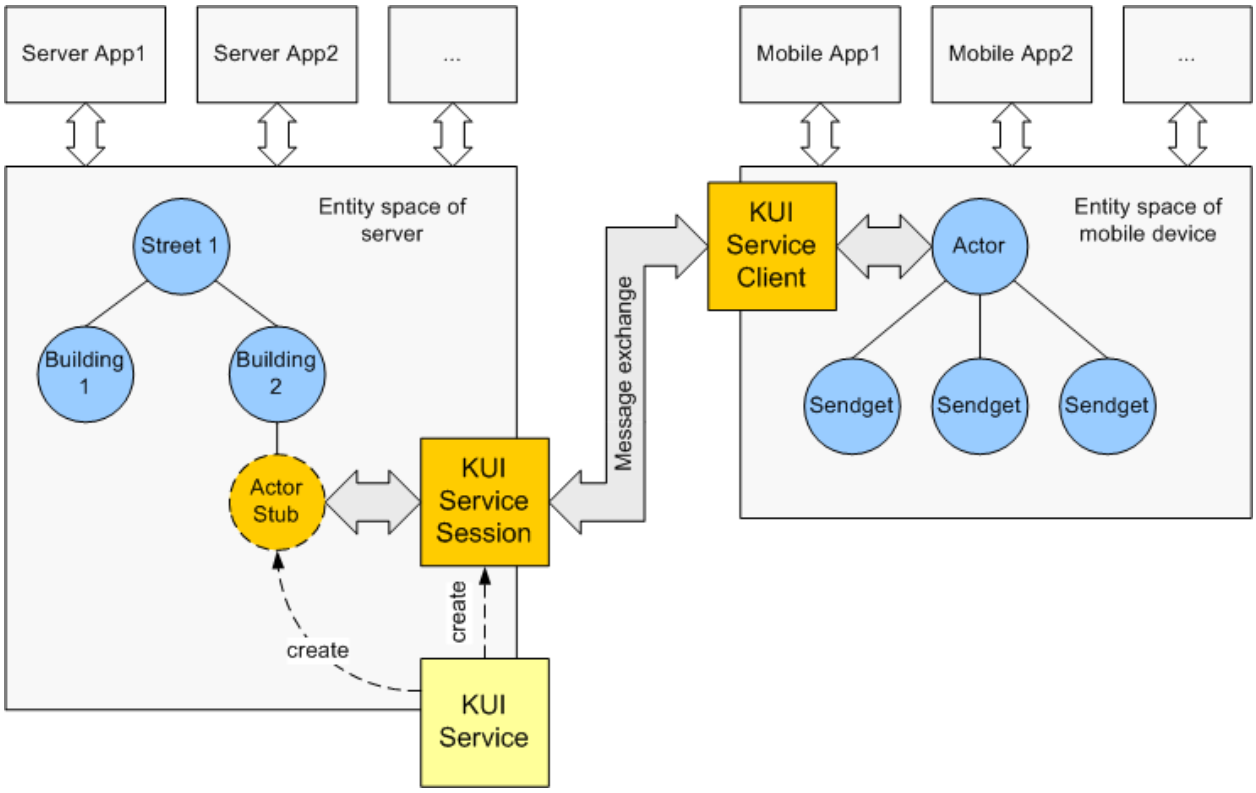



Figure 6.5.: Mobile device integration into entity space of the server using the KUI Service

### 6.5.1. Context Change

As shown in figure 6.6 any context change of an entity starts at the sensor level. For instance the temperature sensor reads a new temperature value and sends the it to the actor (1) which stores it in his context. Then the actor generates two messages. The first message is sent to the observer which observes the actor on the mobile applications (2.1), (3.1) and (4.1). The second message is sent to the KUI Service client (2.2) which passes the message through the network to the KUI Service session (3.2). For any context change the message is directly passed to the stub of the actor (4.2) which actualizes his context too. Afterwards the change is reported to the viewer, the observer and finally to the server application (5.2), (6.2) and (7.2). In parallel the attached rules are evaluated (5.3) and the necessary actions are executed affecting the service registry (6.3). For instance a service might be not available anymore for mobile devices. The smart environment client on the monitoring level queries periodically for service changes. The smart environment session reads the current state of the service registry (7.3) and sends it to the client (8.3). The client itself will recheck and update the availability of services locally and if a service becomes available tries to connect to it. On the other hand if a service disappears the client will disconnect the associated service client (9.3). Finally the application gets the state changes of the connected services (10).

### 6.5.2. Location Change

The uMove implements two strategies to detect the location of an entity:

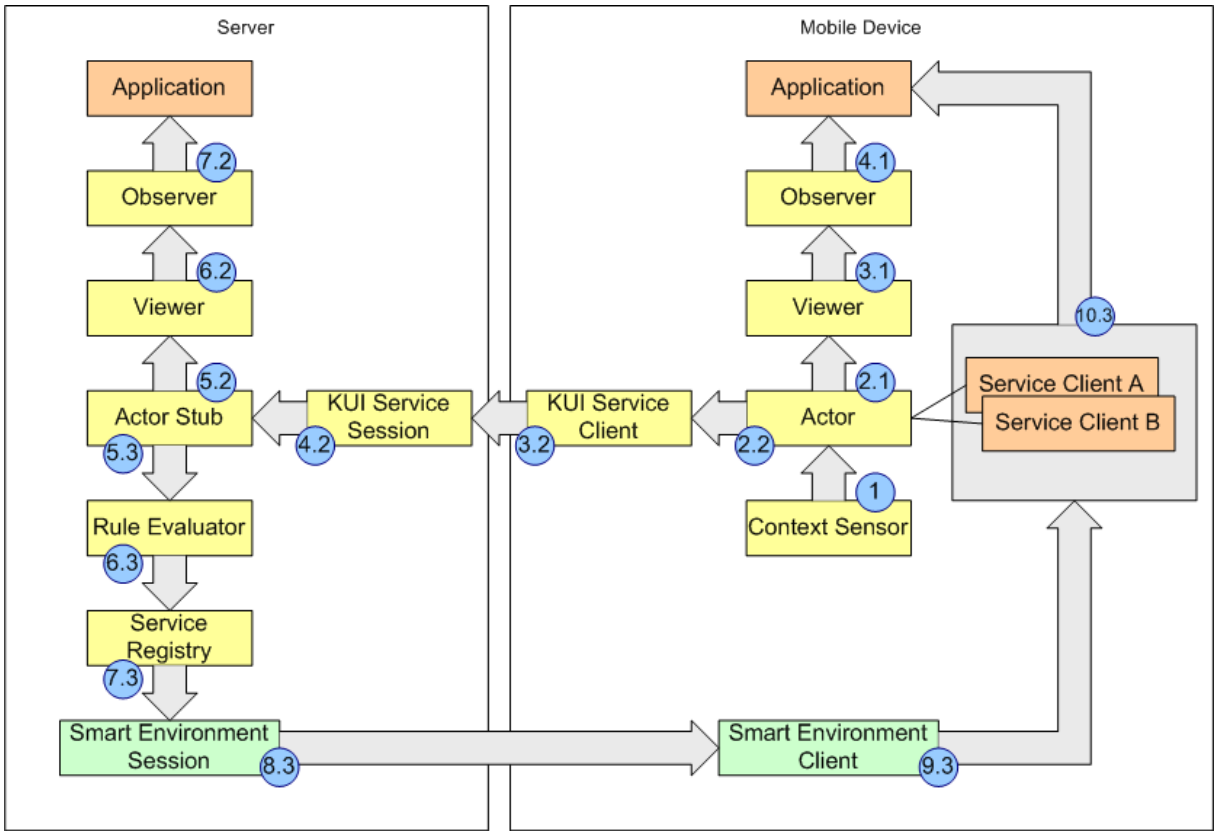



Figure 6.6.: Message flow of a simple context change

- ***Active Detection***: the entity (for instance a mobile device) is capable to localize himself. A sensor attached to the entity detects and processes a location change. Examples are GPS, local positioning using WiFi or bluetooth. In the case of bluetooth tag for instance, it is the zones which are tagged and the mobile entity processes the tag number and gets the corresponding location.
- ***Passive Detection***: the entity wears a tag only. The position of the entity is detected by sensors placed within the environment. Examples are the tracking of letters, goods using RFID or the opening of doors using a badge.

The case study uses the active detection which is more complex and must be solved using a proper entity-stub architecture. The location change starts in the same way as the simple context change. Since the structure is unknown on the mobile device, a location change can not be treated completly and must be handled as a simple context change (figure 6.7). An unknown location change message is sent to the KUI Service session (3.2). Here the message can not be passed directly to the stub-entity, because the message contains only raw information like a GPS position or a RFID-tag. The corresponding location must first be found by a virtual location sendget (4.2) using the entity ID registry (5.2). Once the location is found a location change message is sent to the stub-entity (6.2) which manages the location change. Three entities are involved in this process, the stub-entity representing the actor in the server side, the previous parent entity (old position) and the new parent entity (new position). Finally the following steps are done for each of this three entity:

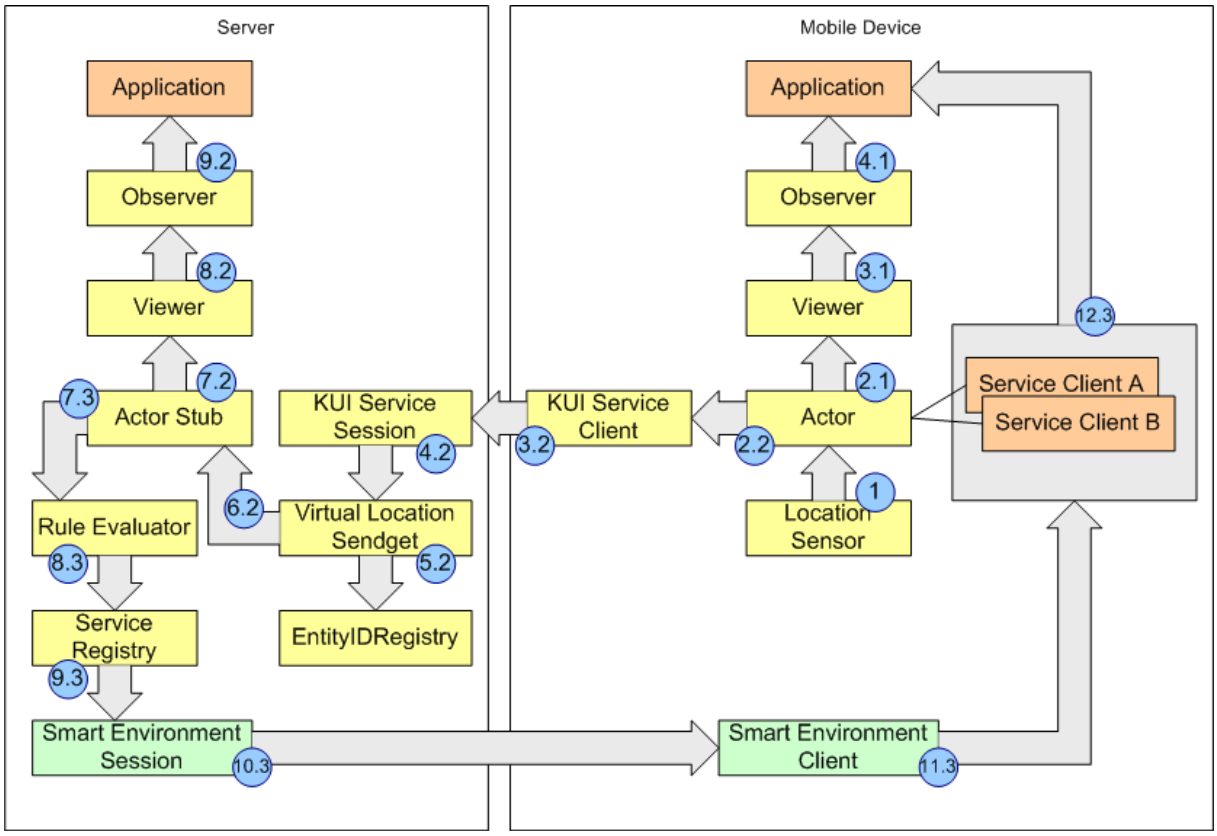



Figure 6.7.: Message flow of a location change

- Updating structure, notification of viewer, observer and application (7.2), (8.2) and (9.2)
- Evaluation of attached rules (7.3) and update of the service registry if needed (8.3)
- If the entity is mobile and is integrated into the smart environment dynamically, the client side will get an update of the availability of services (9.3) upto (12.3)

# 7 Conclusion

## 7.1. Contribution

Coordination is already a wildly discussed topic on which many paper have been published. As a whole this thesis addresses the coordination model for pervasive systems. We chose the systemic point of view to create our model to meet the complexity of the real world. We have introduced several topics connected to coordination, like a holistic physical model, activity theory and dependencies between entities. We have drawn a picture of a coordination model which includes the coordination aspects of the generic coordination model XCM [TCH05], but based on the connected topics we have unearthed new aspects like:

- Activity and task structure
- Observation of entities by observers using viewers
- Relativity of observation in time and space
- Subjective entity classification

These aspects were integrated into our holistic and generic coordination model. The systems we coordinate are based on the model of systems developed by Bruegger et al. [BH09]. They consider a system as an integrate whole, living and open, made of actors, observers and viewers. They evolve over time. In such dynamic systems, activies and tasks of the actors are taken into consideration and require an important level of coordination. To adequately do this coordination, it is required to have a reliable activity recognition.

The concept of observation is an other major extension of the XCM. Coordination is related to an observer perceiving information about the current state of a system. We showed that the perception is always relative to the point of view of an observer. This comprehension led to concept of relative observation (depending of the point of view).

We have also introduced a more specific coordination model for pervasive computing based on the generic coordination model. We showed how coordination is realized in computer science in general and more specifically for pervasive computing systems.

Based on the coordination model for pervasive systems we came up with a coordination language. The language is implemented as a Java library providing coordination functions. It is part of our pervasive middleware made of uMove and coordination libraries

which fills the gap between the different operating systems participating in a smart environment and the pervasive applications. The language is used to define and coordinate the physical and virtual structure, rules, actions and the communication between entities.

The case study was developed with uMove and the Coordination library and demonstrate how entities are tracked inside a smart environment. It allows the integration of a mobile entity into a smart environment and provides pervasive functionality to the mobile entity. Special attention has been given to the connectivity and the implementation of the concept of services. The coordination is illustrated using two pervasive application services, the menu service providing a menu of the cafeteria and the meeting service providing a meeting schedule. These services are coordinated by the smart environment and become available or not to mobile entities depending on their contexts and activities.

## 7.2. Future work

The project of the coordination model for pervasive computing does not end with this thesis. This thesis uncovered many open research fields which the PAI research group intends to answer in a future master thesis or doctoral dissertations. We have identified some of them:

**Rules** The concept of rules is an important instrument to govern and control coordination. In our model we briefly explained how rules can be defined. Observers evaluate the rules and take actions to coordinate activities. A system can have more than one observer producing actions for coordination. We experienced that actions from different observers can create a conflict with other actions (one is undoing something the other just did). This question must be studied in order to avoid such conflit and garanty always a smooth coordination of coherent actions.

The current language implements the rules as Java classes. We need a more flexible way to adapt and allow rules to evolve. Observer should be able to change them as described in subjective coordination.

**Activities** In this thesis we concentrate on communication as the major activity between entities and our coordination is limited to manage the communication between them. As mentioned in chapter 4, human activities can be coordinated using specialized tools. Nowadays these tools are not pervasive. All activities must be entered and planed by human users (e.g. using project management application). A top level pervasive coordination could help to dynamically and implicitly coordinate the work of humans working together. Our system already detects activities by analyzing the motion of entities. This together with the existing planning tools would help to support humans in their daily collaboration with others. Further, the system could be help to improve and control business processes in a more implicit way.

**Pervasive Services** Pervasive services are an essential part of a smart environment. Our coordination language provides just a simple service architecture. Sergio Maffioletti in [Maf04] proposed a pervasive service framework called UBIDEV. Because many aspects

of the Sergio Maffioletti's dissertation have not been reevaluated in the new coordination model, we believe that a comparison between these two models and possibly the integration of Maffioletti's concepts into our new model could be interesting and useful.

# A

# Implementation of Coordination Language

## A.1. Service Architecture

This section explains the architecture and implementation of services. The service architecture is defined in several layers (figure A.1). The first layer defines the ports and the message passing between them. The second layer implements the basic service protocol which controls the communication on the service level. The third layer implements the TCP based communication using the TCP/IP protocol. Finally the last layer implements the application specific service protocol and the functionality of a service.

The major classes are briefly introduced:

- ***IServiceClient***: the interface defines the methods of the service client
- ***IServiceSession***: the interface defines a communication session for services
- ***AbstractServiceClient***: the abstract service client class implements the basic service protocol of the client
- ***AbstractServiceSession***: the abstract server class implements the basic service protocol of the server
- ***AbstractServiceProvider***: the abstract service provider port implements the management of all connected service sessions
- ***TCPBasedServiceClient, TCPBasedServiceSession, TCPBasedServiceProvider***: these classes implement TCP/IP based communication between server and client

The communication between client and server is controlled by the basic service protocol. This protocol implements the login and data exchange between client and server.

- ***Login Process***: implements the login process of a client-server connection
- ***Data Exchange***: implements the data exchange between client and server

The client implements the port behavior and can be handled as a regular port. The port specific methods are overwritten in order to map the call to the basic service protocol. The server has a passive or reactive behavior. Each call is handled and then replayed to the client.

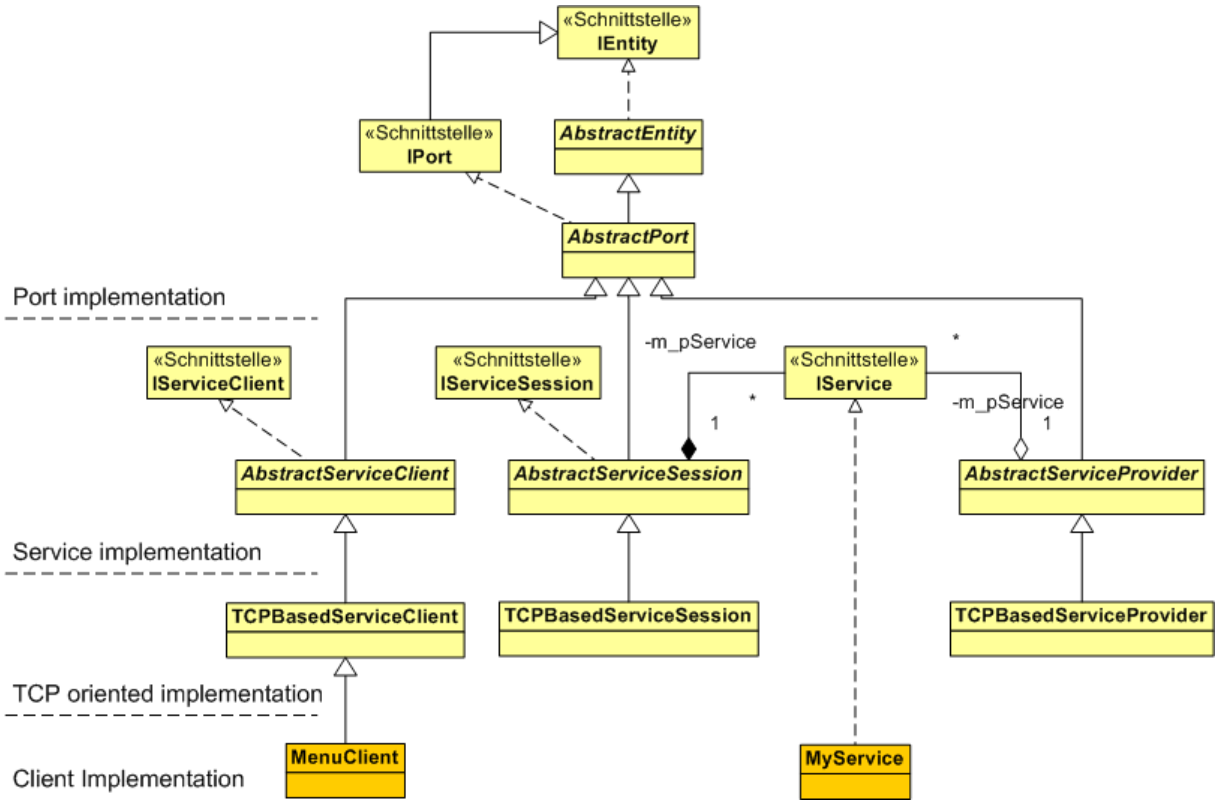



Figure A.1.: Architecture of services

## A.1.1. Login Process

The login is done after a client connects to the server (figure A.2). First a service session object is created and then the login information are sent. If the session object accepts the login then a confirmation is sent back to the client. As a login the client sends:

- ***Parent ID***: the ID of the parent entity or the client port. This expresses to who the client port belongs. Normally the parent is an actor.
- ***Port Descriptor***: the port descriptor of the client. The descriptor contains all information about the client port.

A specialized message type called *LoginMessage* is used to log into the server. This message wraps all login information.

## A.1.2. Data Exchange

The client also uses the basic service protocol to exchange the data with the server. Since the client extends the port class, all port methods are mapped to a query type which is transparently passed to the process() method of the service object on the server side (subsection 5.5.4). The table A.1 shows all port method calls and their corresponding query type:

The *exchange()* is not defined within the port interface but is implemented by the *AbstractServiceClient*. The exchange is a two way communication like read-write.

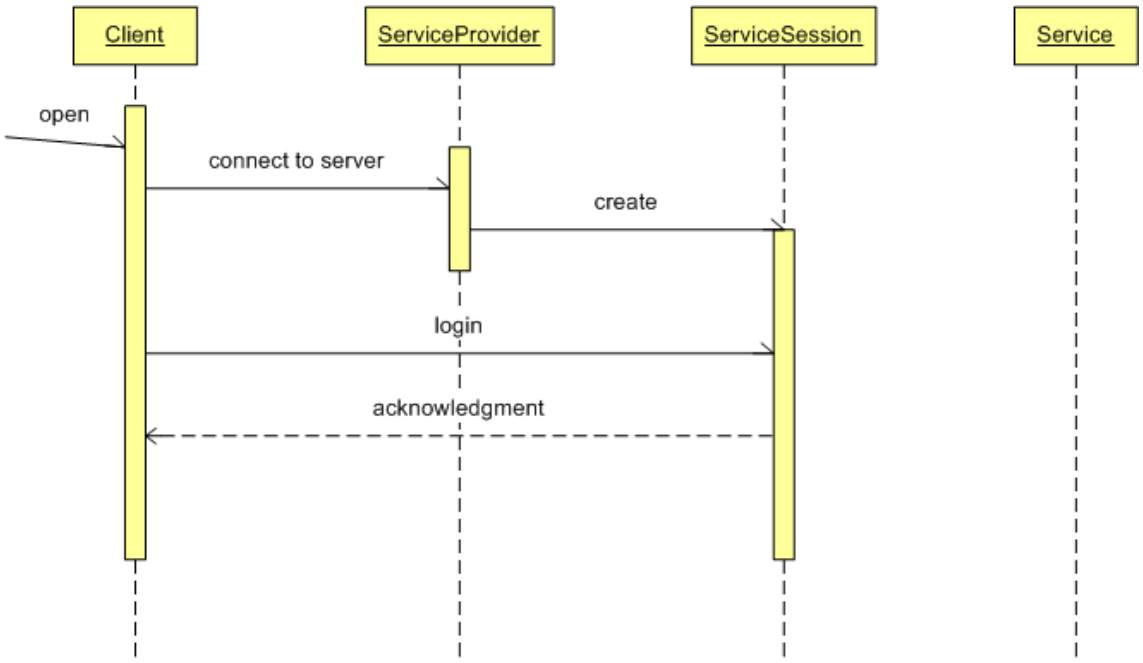



Figure A.2.: Login process of services

| Port method | EServiceQueryType | Service Message Protocol |
|---|---|---|
| read() | etRead | In: Query Type<br>Out: Confirmation + Data Message |
| readBlocked() | etReadBlocked | In: Query Type<br>Out: Confirmation + Data Message |
| peek() | etPeek | In: Query Type<br>Out: Confirmation + Data Message |
| write() | etWrite | In: Query Type + Data Message<br>Out: Confirmation |
| exchange() | etReadWrite | In: Query Type + Data Message<br>Out: Confirmation + Data Message |

Table A.1.: Service Message Protocol

The basic service protocol is implemented by the *ServiceMessage* class. The service message contains fields for querying and confirmation type and for an optional data message. Figure A.3 shows the sequence diagram how data is read from a service.

The current service implementation is intended to be used for querying data. This means that the client is the active part and start the communication. The server is passive and reponses to the incomming requests only. As consequence of this behavoir the client must periodically query the server for information. A server can not directly contact the clients. But the client can also use the readBlocked() call to avoid a constant polling. This blocks the client until the server has data to send to the client.

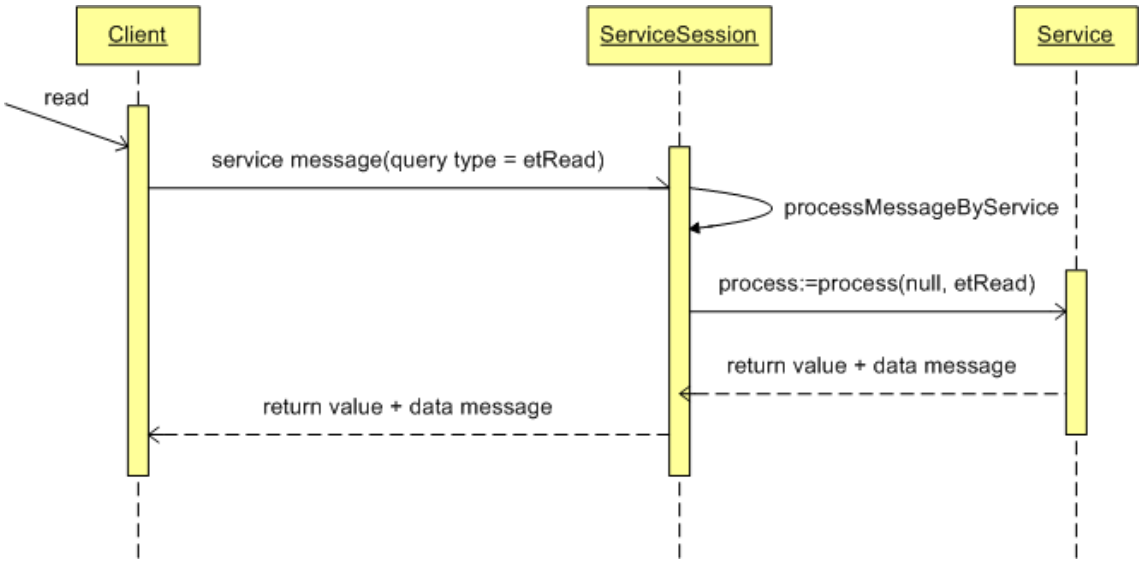



Figure A.3.: Reading a message from the service

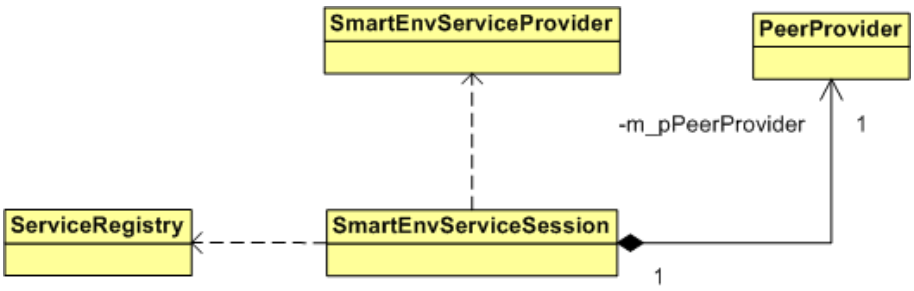



Figure A.4.: Classdiagram of the smart environment service

## A.2. Monitoring

This section explains the implementation of the monitoring used in the case study. The monitoring is realized with two components: the smart environment service provider and the smart environment client.

Figure A.4 shows the monitoring classes of the server. The *PeerProvider* class is used to replay searching signals from mobile devices. The *SmartEnvServiceProvider* provides pervasive services to mobile entities. All services are registered in the *ServiceRegistry*. A session is created for each client entering the environment which handles communication and the exchange of the availability of services.

Figure A.5 shows the classdiagram of a mobile device. The mobile device sends signals out to find a smart environment using the *PeerFinder*. Once the mobile devices finds an environment a *SmartEnvServiceClient* object is created. This object takes care about the data exchange with the environment, mainly to get the availability of other public services. The smart environment client manages the connections between application service clients and the application services of the smart environment. As soon as a new application service becomes available the corresponding application client is connected to it. On the other hand if an application service disappears, the corresponding client is

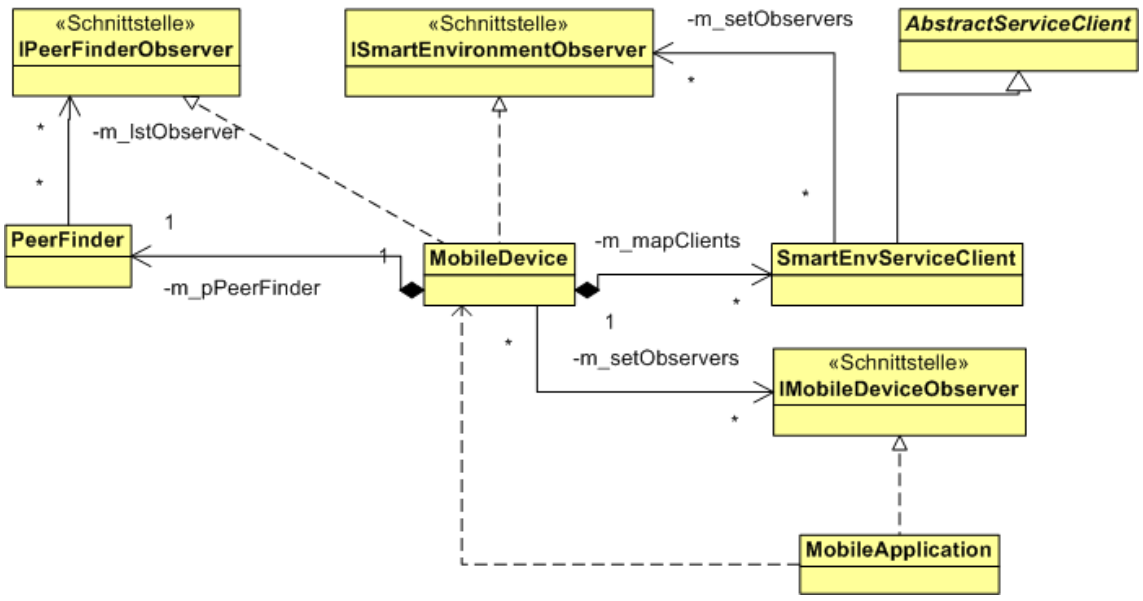



Figure A.5.: Structure of the mobile device

disconnected immediately. A mobile application receives the connecting or disconnecting states through the *IMobileDeviceObserver* interface:

- ***onEnvironmentFound()***: notification if a new environment has been found
- ***onEnvironmentLost()***: notification if the mobile device has been disconnected from an environment
- ***onServiceFound()***: notification if a service becomes available. The matching application client port is returned if it has been connected to the application service automatically.
- ***onServiceLost()***: notification if the application service disappears

# B

# Installation of Pervasive Middleware

## B.1. Motorola Milestone

The Motorola Milestone is equipped with the Android operation system. The Android provides a JVM compatible with Java SE. Few modification on the system must be done in order to install the pervasive middleware of PAI research group.

The proposed setup procedure is explained using the *cmd* shell of Windows. For Linux and Mac their shell applications might use other commands.

### B.1.1. Setup your PC

- Motorola provides a CD which includes the software and drivers. This must be installed on your computer. It's needed in order to connect to your phone using the *adb* tool later in the setup process.
- Installation of the Android. For future setup you have add the Android tools directory to the *PATH* environment variable (Windows).
  - Set the environment variables to "{installation of android-sdk-windows}\tools\"
  - Test if *adb* tool is recognized by the console

### B.1.2. Change Phone Setting

Before starting with the application development few settings must be changed on the phone. Open the settings menu of the mobile phone:

- *Applications* →*Development*: Enable *"USB debugging"*, *"Stay awake"* and *"Allow mock locations"*
- *Sound & Display* →*Display settings* (scroll to the bottom):
  - For GUI testing the *"Orientation"* option might be disabled
  - Increase the *"Screen Timeout"* (depending on your testing purposes)

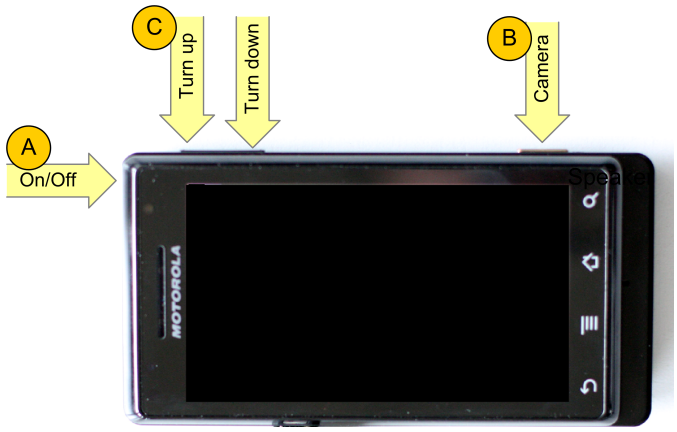



Figure B.1.: Motorola MileStone

## B.1.3. Getting Root Access

The Milestone comes with Android 2.0 User Build, which does not include the Linux root access. With this version we are only capable to install pure APK Applications, but to JAR libraries. So first a special update must be installed to get root access to the phone. According to [8] the version 2.0 **should not be updated**, otherwise rooting the system will probably not work anymore. The following procedure was proposed by [12]. Figure B.1[1] shows the different buttons used in the installing process.

1. Download first the file *update.zip*[2] to the SD-card of the mobile phone.

   Use the *adb* command: *> adb -d push update.zip /sdcard/update.zip*
2. Turn off the mobile phone and **restart holding the camera button** (figure B.1B) until the boot screen appears (figure B.2 left).
   To get to the boot menu **hold the speaker key for turning up the sound (C)** and then **press the camera key again (B)**. The boot menu will appear (figure B.2 middle).
3. Select the *update.zip* and press enter. You might open the phone using the navigation keys (figure B.2 right). There will be an error on the screen saying that it can not find the update_binary. This can be ignored. Root should work after rebooting the phone.

To test if root is working connect the mobile phone with your PC using USB and start the *adb* application.

```
> adb -d shell <ENTER>
\$ su <ENTER>
\#
```

On the phone a dialog will appear which has to be confirmed in order to get root access. There is no password needed though (figure B.3).

[1] The quality of the figures is bad but helpful for the installation procedure
[2] included on CD-Rom: \03_Implementation\Motorola_Milestone\update.zip

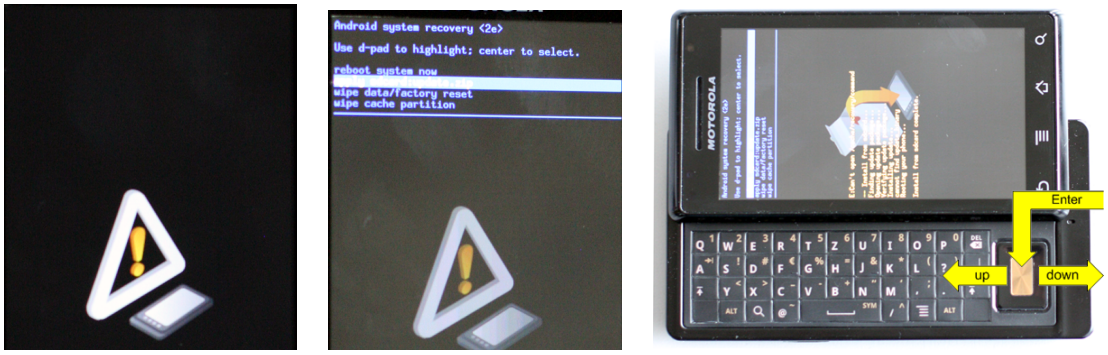



Figure B.2.: Boot procedure to install an update on the Motorola Milestone

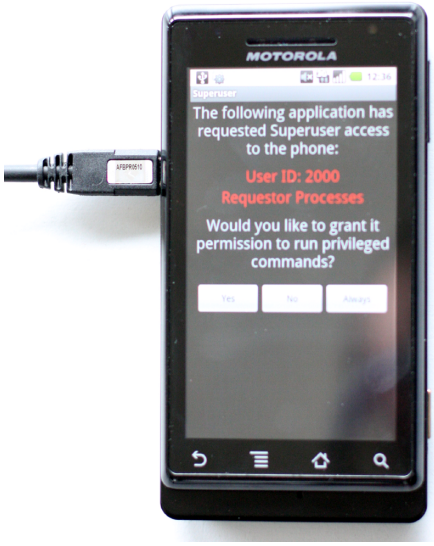



Figure B.3.: Screen showing the root request

## B.1.4. Installation of JAR-Libraries

JAR libraries are not accepted by default on Android 2.0. They have to be created in a special manner first. Following instructions must be repeated for all JARs of the pervasive middleware from unifr[3]:

- ch.unifr.coordination.jar
- ch.unifr.monitoring.jar
- ch.unifr.umove.jar

Follow the instruction for each library listed above:

1. Compile the JAR using Netbeans or Eclipse IDE. Make sure it is compiled against the *android.jar* provided by Android SDK.
2. Generate the file *classes.dex* and add it to the JAR-file using the *dx* application. You find this application within the android platform folder *.\platforms\android-x.x\tools*. Some other examples are given by [6].

```
>dx ––dex ––output=path\classes.dex path\coordination.jar
>cd path
```

[3]Namespace of the University of Fribourg

```
>aapt add coordination.jar classes.dex
```

3. Use *zipalign* tool to gain more performance on the mobile phone. Since the phone is 32bit the align is 4 bytes.

```
>zipalign -f -v 4 coordination.jar ch.unifr.coordination.jar
```

Once the JAR is prepared we have to download and install it on the mobile device. An example is given by [10] or [7]. Use following instructions for the Motorola Milestone:

1. Create an XML for permissions called *"ch.unifr.coordination.xml"* containing the name of the library and the location on the android system:

```
<?xml version="1.0" encoding="utf-8"?>
<permissions>
  <library name="ch.unifr.coordination"
           file="/system/framework/ch.unifr.coordination.jar"/>
</permissions>
```

2. Extend the *platform.xml* of the Android platform
   - Upload the *platform.xml* first

   ```
   >adb -d pull /etc/permissions/platform.xml myfolder\platform.xml
   ```

   - Add the *permissions*-node to the XML:

   ```
   ...
     <library name="ch.unifr.coordination"
              file="/system/framework/ch.unifr.coordination.jar"/>
   ...
   ```

3. Download the JAR-file and both XML files to the SD-card of the phone

```
>adb push ch.unifr.coordination.jar /sdcard
>adb push ch.unifr.coordination.xml /sdcard
>adb push platform.xml /sdcard
```

4. Now install the files on the SD-card to correct folder on the Android platform:
   - Login to the console as root

   ```
   >adb -d shell
   \$ su
   #
   ```

   - Since the folders */system/framework* and */etc/permissions system* are write protected the file system must be remounted with read-write permissions. Note that the remounted block might be different for other phones. Type the *mount* command without parameters to receive the current mounting.

   ```
   #mount
   rootfs / rootfs ro 0 0
   tmpfs /dev tmpfs rw,mode=755 0 0
   devpts /dev/pts devpts rw,mode=600 0 0
   proc /proc proc rw 0 0
   sysfs /sys sysfs rw 0 0
   tmpfs /sqlite_stmt_journals tmpfs rw,size=4096k 0 0
   none /dev/cpuctl cgroup rw,cpu 0 0
   /dev/block/mtdblock6 /system yaffs2 ro 0 0
   /dev/block/mtdblock8 /data yaffs2 rw,nosuid,nodev 0 0
   /dev/block/mtdblock7 /cache yaffs2 rw,nosuid,nodev 0 0
   ```

   - The ***/system*** must be remounted:

```
# mount -o remount,rw -t yaffs2 /dev/block/mtdblock6 /system
```

- Now we can copy the files from the SD-card to the system directories:

```
# dd if=/sdcard/ch.unifr.coordination.jar
 of=/system/framework/ch.unifr.coordination.jar
# dd if=/sdcard/ch.unifr.coordination.xml
 of=/etc/permissions/ch.unifr.coordination.xml
# dd if=/sdcard/platform.xml of=/etc/permissions/platform.xml
```

- Because we copied the data as root, the file rights must be changed to **user access rights**.

```
# chmod 644 /system/framework/ch.unifr.coordination.jar
# chmod 644 /etc/permissions/ch.unifr.coordination.xml
```

5. Remount the system with *read-only* access and reboot the phone.

```
# mount -o ro,remount -t yaffs2 /dev/block/mtdblock6 /system
# sync
# reboot
```

6. If you work on the PAI projects using Netbeans (such as coordination, umove or monitoring), download also the *install.sh* to the phone. The *install.sh* is used to automatically update the JARs on the phone.
   a) Download *install.sh*:

```
> adb -d push install.sh /sdcard
```

   b) The *build.xml* of the projects are adapted to automatically generate the DEX-file, create a JAR for Android and download it to the SD-card of the phone
   c) To update the JARs run *install-jars.bat* on your computer (windows)

### B.1.5. Using a Library in a APK Project

First the JAR libraries must be added to the project. For netbeans click on the *Library* folder of the project and use *"add JAR/Folder"* in the context menu. The added JAR libraries must be also declared in the *androidmanifest.xml* of the project. Add following line within the *application*-node:

```
<application android:debuggable="true">
  <uses-library android:name="ch.unifr.coordination"/>
...
</application>
```

If this line is missing the android system can not resolve the classes and methods from the jar library and will throw an exception of type *java.lang.NoClassDefFoundError*.

NOW YOU ARE READY TO RUN YOUR PERVASIVE APPLICATION. Have Fun!!

# C
# Common Acronyms

**ALU** Arithmetic and Logical Unit
**API** Application Programming Interface
**ASI** Actuator-Sensor-Interface
**AST** Abstract Syntax Tree
**CD** Compact Disk
**CERN** European Organization for Nuclear Research
**CM** Coordination Manager
**CORBA** Common Object Request Broker Architecture
**CPU** Central Processing Unit
**FIFO** First In - First Out
**FTP** File Transfer Protocol
**GPS** Global Positioning System
**HTML** Hypertext Markup Language
**HCI** Human Computer Interface
**KUI** Kinetic User Interface
**ID** Identifier
**IDE** Integrated Development Environment
**IO** Input-Output
**IP** Internet Protocol
**JVM** Java Virtual Machine
**MAS** Multi agent system
**MMU** Memory Management Unit
**MPI** Message Passing Interface
**MS** Microsoft
**OS** Operation System
**OSI** Open Systems Interconnection Reference Model
**PAI** Pervasive and Artificial Intelligence research group
**PC** Personal Computer

**PCI** Peripheral Component Interconnect

**PCM** Pervasive Coordination Model

**PDA** Personal Digital Assistant (electronic handheld information device)

**PLC** Programmable Logic Controller

**Qt** C++ library of Quasar Technologies

**RAM** Random Access Memory

**RFID** Radio Frequency Identification

**SMS** Short Message Service

**STL** Simple Thread Language for C and C++ (STL++)

**TCP** Transmission Control Protocol

**UDP** User Datagram Protocol

**USB** Universal Serial Bus

**UUID** Universally Unique Identifier

**VCL** Visual component library from Borland

**WECA** Wireless Ethernet Compatibility Alliance

**WiFi** Trademark of the Wi-Fi Alliance (former WECA. Used with certified products that belong to the class of WLAN devices.

**WLAN** Wireless local area network

**XCM** Generic Coordination Model for pervasive computing

**XML** Extensible Markup Language

# D License of the Documentation

# E
# Deliverable Products

This chapter gives an overview of the products delivered with this Master Thesis, like:

- A printout of the Master Thesis
- A poster for exhibition
- A CD-ROM Figure E.2 containing the binaries and documents of the Master Thesis
  - The source code, compiled binaries and API documents of:
    - The coordination component
    - The lean monitoring (smart environment monitoring)
    - The uMove Framework (Version 2)
  - The binaries and sources of this document (LaTeX)
  - Coordination Models
  - Referenced documents and readings used during this Master Thesis

Figure E.1 provides a tree view of the CD-ROM.

```
|-- 01_Documentation            // Msc Thesis, Appendix, Java Doc
|   |-- latex                   // Latex source of the thesis
|   |-- graphics                // Graphics used in the documents
|   |-- graphics_high_res       // Printing version of graphics
|   |-- JavaDocCoordinationLib   // API of the coordination component
|   |-- JavaDocMonitoringLib    // API of the lean monitoring
|   `-- JavaDocuMoveLib         //API of the uMove framework
|
|-- 02_Presentation             // presentations of the Msc Thesis
|
|-- 03_Implementation           // Implementation of the thesis
|   |-- Coordination            // Coordination language
|   |-- CoordinationMonitor     // Coordination monitoring (debugging)
|   |-- MobileMonitoring        // Lean monitoring implementation
|   |-- robin                   // Server application (robin)
|   |-- SmartAppForAndroid      // Client application for android
|   |-- uMove2                  // uMove framework
|   `-- Motorola_Milestone      // installer for mobile phone
|
|-- 04_References               // reference documents
|-- 05_ProjectPlan              // Project plan and dead lines
|-- 06_GuideLines               // Guideline of the thesis
`-- 08_Images                   // Various images used in the documents
```

Figure E.1.: Tree view of the content of the CD-ROM

Figure E.2.: The CD-ROM of this project

# References


[ADB+99] Gregory D. Abowd, Anind K. Dey, Peter J. Brown, Nigel Davies, Mark Smith, and Pete Steggles. Towards a better understanding of context and context-awareness. In *HUC '99: Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing*, pages 304–307, London, UK, 1999. Springer-Verlag.

[BD07] Genevieve Bell and Paul Dourish. Yesterday's tomorrows: notes on ubiquitous computing's dominant vision. *Personal Ubiquitous Comput.*, 11(2):133–143, 2007. 2

[BH09] Pascal Bruegger and Béat Hirsbrunner. Kinetic user interface: Interaction through motion for pervasive computing systems. In *Parallel session "Designing for Mobile Computing"*, pages 297–306. HCI international conference 2009, Springer, July 2009. 2, 4

[BLLH10] Pascal Bruegger, Agnes Lisowska, Denis Lalanne, and Béat Hirsbrunner. Enriching the design and prototyping loop: A set of tools to support the creation of activity-based pervasive applications. March 2010. 2

[BPH09] Pascal Bruegger, Vincenzo Pallotta, and Béat Hirsbrunner. Optimizing heating system management using an activity-based pervasive application. *Journal of Digital Information Management*, July 2009.

[BS06] Jasmin Blanchette and Mark Summerfield. *C++ GUI Programming with Qt 4*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.

[CD88] George F. Coulouris and Jean Dollimore. *Distributed systems: concepts and design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[Dix02] Alan Dix. Beyond intention pushing boundaries with incidentional interaction. pages 1–5, Lancaster, UK, 2002. Lancaster University.

[GC92] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):97–107, 1992. 2, 6

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Gid84] A Giddens. *The Constitution of Society: Outline of the Theory of Structure*, volume 1. Berkeley, CA: University of California Press., 1984.

[KCDH98] Olivier Krone, F. Chantemargue, T. Dagaeff, and Béat Hirsbrunner. Coordinating autonomous agents. In *Proceedings of the ACM Symposium on Applied Computing. Special Track on Coordination, Languages and Applications*, 1998. 7

[Koh07] Jürg Kohlas. Script notes: Theory of computation, computability and complexity, 2007.

[Kuu95] Kari Kuutti. Activity theory as a potential framework for human-computer interaction research. pages 17–44, 1995. 8

[Lew04] Frank L. Lewis. *Smart Environments: Technology, Protocols and Applications*, chapter 4: Wireless Sensor Networks. Parallel and Distributed Computing. Wiley-Interscience, November 2004.

[LL08] Yang Li and James A. Landay. Activity-based prototyping of ubicomp applications for long-lived, everyday human activities. In *CHI '08: Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, pages 1303–1312, New York, NY, USA, 2008. ACM.

[Lok04] Seng W. Loke. Representing and reasoning with situations for context-aware pervasive computing: a logic programming perspective. *Knowl. Eng. Rev.*, 19(3):213–233, 2004.

[Maf04] Sergio Maffioletti. *UbiDev: A Middleware for Ubiquitous Computing*. phdthesis, 2004.

[MC94] Thomas W. Malone and Kevin Crowston. The interdisciplinary study of coordination. *ACM Comput. Surv.*, 26(1):87–119, 1994. 2, 6

[MKMH04] Sergio Maffioletti, Soraya Kouadri Mostefaoui, and Béat Hirsbrunner. Automatic resource and service management for ubiquitous computing environments. In *Middleware Support for Pervasive Computing Workshop (at PerCom '04)*, pages 219–223. PerWare, March 2004. 7

[Mor71] Charles W. Morris. *Writings on the general theory of signs*. The Hague, Mouton, 1971.

[Nar95] Bonnie A. Nardi, editor. *Context and consciousness: activity theory and human-computer interaction*. Massachusetts Institute of Technology, Cambridge, MA, USA, 1995. 8

[ORRV03] Andrea Omicini, Alessandro Ricci, Giovanni Rimassa, and Mirko Viroli. Integrating objective & subjective coordination in fipa: A roadmap to tucson. In *WOA*, pages 85–91, 2003.

[Sch01] Michael Schumacher. *Objective Coordination in Multi-Agent System Engineering: Design and Implementation*, volume 2039. Springer-Verlag, 2001. 2, 7

[Sch02] Eric Schwarz. Can real life complex systems be interpreted with the usual dualist physicalist epistemology - or is a holistic approach necessary? pages 1–5, March 2002. 7, 8

[SW49] C. E. Shannon and W. Weaver. *The mathematical theory of communication*. The University of Illinois Press, Urbana, IL, 1949.

[TCH05] Amine Tafat, Michèle Courant, and Béat Hirsbrunner. Implicit environment-based coordination in pervasive computing. In *SAC '05: Proceedings of the*

*2005 ACM symposium on Applied computing*, pages 457–461, New York, NY, USA, 2005. ACM. 7

[Wei91] Mark Weiser. The computer for the 21st century. *Scientific American*, 02/1991 1991. 2

# Referenced Web Ressources


[1] Embarcadero rad studio. http://www.embarcadero.com (accessed February 22, 2010).

[2] Pascal Bruegger. ubiglide. an application for gliding activities based on a motion-aware middleware architecture. pages 47–48. University of Fribourg, 2007. http://diuf.unifr.ch/pai/education/studentProjects/PascalBruegger/mscubiglide.pdf (accessed February 22, 2010).

[3] Kevin Crowston. A taxonomy of organisational dependancies and coordination mechnaism. pages 2–14, 1994. http://ccs.mit.edu/papers/CCSWP174.html (accessed February 22, 2010).

[4] Online Encyclopedia ENCYCLO. Definition of observation. http://www.encyclo.co.uk/define/observation (accessed March 10, 2010).

[5] Free Documentation Licence (GNU FDL). http://www.gnu.org/licenses/fdl.txt (accessed February 22, 2010).

[6] felix.apache.org. Apache felix framework and google android. http://felix.apache.org/site/apache-felix-framework-and-google-android.html (accessed February 22, 2010).

[7] messenger13. Multitouch gallery. http://androidforums.com/support-troubleshooting/34622-multitouch-gallery.html (accessed February 22, 2010).

[8] nodch. Motorola milestone android 2.0.1 update. http://www.nodch.de/motorola-milestone-android-2-0-1-update/1633 (accessed February 22, 2010).

[9] Fourth Edition The American Heritage® Dictionary of the English Language. Definition of service. http://dictionary.reference.com/browse/service (accessed February 22, 2010).

[10] verdebreuk. Google maps working on openmoko freerunner. http://verdebreuk.blogspot.com/2009/08/google-maps-working-on-openmoko.html (accessed February 22, 2010).

[11] World Wide Web Consortium (W3C). Xml technology. http://www.w3.org/standards/xml/ (accessed February 22, 2010).

[12] www.machackpc.com. How to root motorola milestone? http://www.machackpc.com/featured/how-to-root-motorola-milestone/ (accessed February 22, 2010).

# Index