Vrije Universiteit Brussel

Faculty of Science
Department of Computer Science
Programming Technology Lab

Environment queries: Service Discovery For Open Mobile Systems

Thesis submitted in fulfillment of the requirements for the degree of Master of Applied Computer Science

Frederik Geerts

Academic Year 2005-2006

Promotor: Prof. Dr. Theo D'Hondt
Supervisor: Stijn Mostinckx

August 2006

Vrije Universiteit Brussel

Faculty of Science
Department of Computer Science
Programming Technology Lab

Environment queries: Service Discovery For Open Mobile Systems

Thesis submitted in fulfillment of the requirements for the degree of Master of Applied Computer Science

Frederik Geerts

Academic Year 2005-2006

Promotor: Prof. Dr. Theo D'Hondt
Supervisor: Stijn Mostinckx

August 2006

You can't use what you can't find. – Peter Morville

Abstract

The proliferation of mobile devices and of the open mobile networks they bring forth, complicates the advertisement and discovery of services. The hardware characteristics of such mobile networks issue severe consequences for software development. In response to these consequences the Ambient-Oriented Programming paradigm was introduced. Service discovery protocols for open mobile networks must adhere to the same hardware characteristics. The current state-of-the-art protocols fail to cope with these restrictions. This led to the identification of a number of criteria to which an Ambient-Oriented service discovery protocol must comply with. The most crucial of those criteria is: rich service descriptions.

This dissertation attempts to resolve the absence of rich service descriptions. The solution we propose, is to enrich AmbientTalk in such a way that it can describe services in terms of logic. A logic programming language is chosen as a service description language, because this makes reasoning about the services easier and more expressive. This way we gain the advantages of current protocols, without the disadvantages for resource-poor devices. Our solution, environment queries, provide a concise way of describing services where processing power, memory and bandwidth, are limited or vary profoundly and where expressiveness is a critical factor to cope with the (future) proliferation of services. AmbientTalks enhanced service discovery protocol is extensively validated by using our implementation in various cases.

Samenvatting

De proliferatie van mobiele toestellen en van de open mobiele netwerken, die ze voortbrengen, maken het aanbieden en ontdekken van bepaalde diensten moeilijker. De hardware kenmerken van zulke mobiele netwerken hebben ernstige gevolgen op de software ontwikkeling. Als een antwoord op deze gevolgen werd het 'Ambient-Oriented Programming' paradigma geïntroduceerd. 'Service Discovery' protocollen voor open mobiele netwerken moeten zich houden aan diezelfde hardware restricties. De huidige protocollen slagen er echter niet in om te gaan met deze beperkingen. Dit heeft geleid tot de identificatie van een aantal criteria aan de welke het 'Ambient-Oriented Service Discovery' protocol moet voldoen. Het meest cruciale kenmerk is hierbij het expressief omschrijven van bepaalde diensten.

Deze eindverhandeling moet gezien worden als een poging om de afwezigheid van dit bovengenoemde kenmerk op te lossen. De oplossing die wij voorstellen, is het verrijken van AmbientTalk in die zin dat het diensten logisch kan omschrijven. We hebben gekozen voor een logische programmeertaal omdat dit het redeneren over de diensten vereenvoudigt en tevens expressiever maakt. Op die manier behouden we de voordelen van de huidige protocollen en hebben we niet te maken met de nadelen voor wat betreft toestellen met beperkte middelen. Onze oplossing, zijnde Environment Queries, biedt een beknopte manier aan om diensten te omschrijven waarbij de verwerkingscapaciteit, het geheugen en de bandbreedte beperkt zijn of ernstig kunnen schommelen en waarbij de expressiviteit een kritieke rol speelt in het omgaan met de (toekomstige) proliferatie van diensten. Het verbeterde AmbientTalk Service Discovery protocol wordt uitgebreid gevalideerd door onze implementatie in verschillende scenarios aan te wenden.

Acknowledgements

First of all I like to thank Prof. Dr. Theo D'Hondt for promoting this dissertation.
Special thanks goes to my supervisor Stijn Mostinckx for his advice, his help whenever I encountered problems during this research and the programming of my proof-of-concept implementation.
I would also like to thank all the members and students of the Programming Technology Lab for their advice and opinions on the subject, which led to many discussions that contributed to a better understanding of the matter.
I also express my thanks to Frans Govaerts and Sofie Demeyer for proof-reading this dissertation and correcting my spelling, grammar or other inconsistencies.
Finally, many thanks to my parents for giving me this opportunity and to my girlfriend and friends for their support and distraction.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

In traditional (non-ad-hoc) networks, system administration is so time-consuming that most companies charge an entire in house department for that task or even outsource the whole problem. In case we want to deploy a new service on our network, we have to assign it a network address and publish that address to all users who want to take advantage of the service it provides. When the service fails or loses its connectivity, clients can't automatically be redirected to a substitute or backup service that is up and running, nor are they informed when the component is available again. In many cases, different services force different drivers on every client that wants to use them. In such a network, hosting a temporary guest who might want to utilize printers, beamers or internet connection, is so complicated that it is usually avoided unless absolutely indispensable. [Dyr03]

So, even now one of the toughest tasks in computer maintenance is the location and configuration of networked services such as printers, mail or database servers. It poses a number of logistical and technical challenges and as the number of network services increases, so does the need for a service discovery mechanism.

Another problem is the rapid increase in numbers of mobile devices such as cellular phones, mp3-players, palm-sized computers, UMPC's and other portable gadgets. These devices revolve around suitable form factors and low power consumption instead of functionality. For this reason they are "peripheral-poor" and need other devices in the vicinity for services like: storage, faxing, printing, scanning and internet access. All these devices together form open mobile (ad-hoc) networks, which are the subject of this dissertation.

Assuming these existing trends will continue, we will have to support true mobility of users in open networks, and focus on interactions between different mobile devices and devices embedded in their surroundings. This vision has been researched for a few years and is known as Ambient Intelligence [DBS+03], which can only be supported if we change the way services are advertised and discovered. As a result, there has recently been a considerable amount of research into the service discovery field, which we will review in chapter three. Basing ourselves on this research we will extend the service discovery protocol of AmbientTalk with a new service description language that incorporates the advantages of current protocols, while keeping in mind the restrictions of an open mobile network and even introducing more expressiveness in service advertisements and queries.

1.1 Proposed solution in the Dissertation

The solution proposed in this dissertation is to enrich AmbientTalk in such a way that it can describe services in terms of logic. A logic programming language is chosen as a service description language, because this makes reasoning about the services easier and more expressive. Additionally, a symbiosis will be constructed between the logic language and the base language.

Language symbiosis between two languages [Gyb03] means that both languages can use each other's functionality. For instance, this can enable two object-oriented languages to send messages to one another's objects, or enable an object-oriented programming language to execute queries in a declarative language.

In our approach a symbiosis between a logic service description language

and a prototype-based base language is constructed, in order to provide the base language AmbientTalk with the ability to manage service descriptions.

AmbientTalk is a prototype-based ambient-oriented programming language developed at the Vrije Universiteit Brussel, and will be used as base language in our experiment. AmbientTalk is based on Pic%, which itself is an extension of Pico, a very simple and expressive language originally intended to teach programming concepts to non-computer science students. AmbientTalk strives to hold on to the expressiveness and simple syntax featured in Pico.
Loco is the logic programming language that will be used as description language to describe services in AmbientTalk. Like AmbientTalk, Loco is also developed at the Vrije Universiteit Brussel. Loco syntactically resembles Pico (and AmbientTalk), which will prove to be very useful when constructing the language symbiosis.

In our experiments, Loco will be triggered when we require or provide services in AmbientTalk. Loco will reason about them and send back the results to AmbientTalk. This proof-of-concept implementation enables us to discuss the benefits of using a logic programming language as a service description language for service discovery in open mobile systems.

1.2 Roadmap to the Dissertation

In the next chapter we will describe the context of this dissertation. We will address the hardware phenomena that become apparent in open mobile networks and how languages of the ambient-oriented programming paradigm handle them. At the end of the chapter a prototype-based language, following the paradigm, is discussed in more detail.
Chapter three will explain what service discovery is, and the concepts that compose a service discovery protocol are defined. Different strategies and design options will be discussed. A couple of scenarios will illustrate the use of discovery, and they are followed by an in depth analysis of nine of the most important service discovery protocols including AmbientTalk. A comparison is made between them and we identify related research in the same area. The chapter will then be concluded with a summary and outlook

to the future.
Chapter four revolves around logic programming and language symbiosis. We describe the declarative programming paradigm and take a closer look at Loco, the logic programming language that will be used in the symbiosis. We briefly look at language symbiosis and at a concrete implementation: Symbioco.
In the fifth chapter we finally introduce environment queries and we put forward a proof-of-concept implementation supporting the thesis. Hereafter a more technical discussion will be presented concerning a number of practical issues and the chapter will end by several validating examples.
The sixth and final chapter presents our conclusions and identifies areas for future work.

2
Context

In this chapter an overview is given of the different concepts that shape the context in which this dissertation is to be situated. Each of the concepts reflects on the work presented in the remainder of this thesis as will be indicated in this chapter.

2.1 Ambient Intelligence

Ambient Intelligence (AmI) is a vision of the future where electronic devices are integrated into our everyday environments, surrounding users with a so called processor cloud, that is sensitive as well as responsive to both the presence and actions of people. Such an AmI environment is characterized by the following key attributes: context aware, adaptive, personalized, ubiquitous, immersive, transparent and intelligent. [DBS+03]

In addition to the hardware embedded in the environment, the AmI vision also foresees that people will carry around a Personal Area Network (PAN): a network of small consumer electronics (mp3-players, mobile phones, pdas, ...) which interact with both the embedded devices as well as devices belonging to the personal area networks of others in their immediate surroundings.

The networks that connect these devices are dynamically defined because devices may enter and leave at any time, usually as a result of users moving about.

The vision on AmI seeks to integrate computing technology into everyday life much like electricity has pervaded everyday life: it is ever-present and widely used, but we do not think about it, and most of the time we are not even aware of it. AmI intends to provide design criteria for establishing a similar intelligent infrastructure of computing devices; intelligent, not only because it can interpret our actions and intentions, but also because it can change, more or less interactively, our environment to help us with transparent solutions. [PRDIRC05]

In the context of this dissertation, we will consider a small subset of the problem of integrating computing power transparently into everyday life, namely the selection of which service(s) to interact with. The motivating example we will use throughout this dissertation: Imagine yourself walking through a computer lab, in your hand a PDA and on it a document you would like to have a hardcopy of. You are in a hurry and just want a quick draft, so you select a dpi [1] of minimum 300. Meanwhile your PDA has discovered all the matching nearby printers, even arranged them by your previous preferences like speed, distance from your position, size of the print queue and so on. The most suitable one has already been selected as default, you just press the print-button and promptly you are shown directions to the printer where you will find your document. But it doesn't stop here, the possibilities are nearly endless, and more elaborated scenarios have been written by ISTAG. [DBS+03]

2.2 Hardware phenomena

Mobile devices are becoming widely available and almost all of them can be classified as "resource-poor". They have limited processing power, memory and battery capacity compared to traditional hardware. Although things are changing in the last years, e.g. smart phones and pda's are sharing more

[1] Dots Per Inch/Pixels Per Inch. The resolution of an image or how many pixels are defined in the boundary of a square inch. From Cadmus Professional Communications

and more functionality with "resource-rich" devices such as laptop. We have come to the point were the mobile devices have enough resources (up to a certain point) to run meaningful services for others to use. This is also the reason why we believe the capabilities and resources of mobile devices will keep growing. In contrast to these hardware restrictions, which will be gradually resolved as technology progresses, there are more fundamental issues in an ambient environment, that separate mobile networks from existing technology. [DCM+05]

2.2.1 Ambient Resources

In static networks references to remote services are often obtained by encoding the existence of the service into the application. This technique is no longer applicable in a context populated by mobile devices as remote resources may appear or disappear dynamically in the environment. The availability of a resource at that point relies solely on the location of the requesting device.

2.2.2 Autonomy

For static networks the client-server paradigm remains the dominant strategy for developing distributed applications. Generally the server coordinates the various interactions between its clients, but in case of (ad-hoc) mobile networks a connection to such a coordinator is not always feasible or even possible. The connection may disconnect as the location of any device can change at any moment. Therefore each device has to act as autonomous computing component.

2.2.3 Connection volatility

We also have to deal with the fact that wireless communication is restricted by a perimeter and a user can move out of range, breaking connections. Therefore we can never assume stable connections between two collaborating devices. When a task is interrupted as result of a broken connection, users typically expect the program to resume the job at hand, should the connection be restored (within a reasonable amount of time). So users assume their tasks to be carried out in spite of the presence of volatile connections.

2.2.4 Natural Concurrency

Distribution and concurrency are not the same phenomena in theory, but they tend to intertwine in practice. In theory a client can wait for a server to return results before resuming its task, yielding a form of distribution with concurrency. However, both to optimally exploit the available processing power in the network of devices and to maximise the autonomy of the involved devices, concurrency is a natural phenomenon for distributed software. A testament to this is the tendency of developing more and more multi-threaded software. Before long this will also arise on mobile devices, as the devices evolve from single-purpose to multi-purpose devices.

In response to these characteristics the AmOP paradigm was introduced [DCM+05]. This will be explained next and is the paradigm that will be used for this dissertation. We will also use these criteria to evaluate the existing service discovery protocols in section 3.3. The current service discovery protocols don't solve all the challenges of open mobile environments. Hence we deem new solutions are necessary and they will come from a new paradigm: Ambient-Oriented Programming.

2.3 Ambient-Oriented Programming

To be able to realize the AmI vision, advances in both hardware and appropriate software support are needed. Supporting the development of such applications, is a task for which the current generation of programming languages were not designed. Therefore it is useful to explore a new branch of programming languages that are equipped with built-in features to handle the volatile connections and the openness of the network.
More dedicated abstractions are needed to deal with all the consequences of mobile hardware in order to alleviate the burden on software developers. This observation justifies the need for a new Ambient-Oriented Programming paradigm (AmOP for short) that consists of programming languages that explicitly incorporate potential network failures in the very heart of their basic computational steps. [OOP05]

A key characteristic of these languages is that they are based on dynamically typed prototype-based programming languages that have the neces-

sary built-in provisions to deal with networking, partial-failure, distribution, mobility, persistence and so on.

The AmOP paradigm can be described more formally by a set of four characteristics, which are directly inspired by the hardware phenomena described in section 2.2. These hardware phenomena effectively issue boundary conditions for the AmOP-paradigm, which we will describe in this section. Up to the present the object-oriented paradigm has been the most successful approach for coping with distribution and its induced concurrency, because there is a natural alignment between encapsulated objects and concurrently running distributed software entities. [BY87] Hence, Ambient-Oriented Programming languages are a member of the distributed concurrent object-oriented programming language family but they also share some other characteristics that differentiate them from this group, which will be addressed in the following sections.

2.3.1 Non-blocking Communication Primitives

There are two main reasons why non-blocking communication primitives are necessary in an ambient environment: the first is the volatility of the employed connections and the second reason is that every hardware device is autonomous and therefore induces natural concurrency. Blocking communication primitives often give rise to (distributed) deadlocks. [VA98] In local networks we can deal with these (distributed) deadlocks through contemporary remote debugging environments, but more harm is bound to happen in mobile networks, where not all parties are necessarily available for communication and thus vastly complicating the solution of deadlocks in these networks.
An additional and more crucial issue for a concurrency model running on mobile networks is that we should minimize the duration of locks on resources. In these networks we have to cope with severe high latency of communication over volatile connections, otherwise the availability of the resources will die down. If we nonetheless choose to have blocking communication, a program or device would block the moment it stumbles across unstable connections or the (temporary) unavailability of a device. These reasonings have already been mentioned in literature [MCE02] [CNP01] [MPR01] and reinforce the choice for a concurrency model without blocking communication primitives

for Ambient-Oriented Programming languages.

It's also important to stress that non-blocking communication is not the same thing as asynchronous communication. Asynchronous message sending represents only half of the issue: the send operation is non-blocking, but asynchronous communication doesn't assert anything of the receive operation. A typical example of asynchronous send operations combined with blocking receive operations is found in the tuple-space based middleware , which provide explicit, blocking receive operations on the tuple-space [Gel85].

2.3.2 Reified Communication Traces

Seeing that we have non-blocking communication, both the senders and receivers will continue their execution regardless of any events after the message send. For that reason it may lead up to an inconsistent state concerning the task at hand, which they are trying to solve. They must be capable of restoring their state to a consistent one, so they can decide what to do next. Examples of the latter could be overruling one of the two computations or could be deciding together on a new state with which both parties can resume their computation. Hence, a programming language in the ambient-oriented paradigm will have to provide us with reversibility provisions giving programmers a way to manipulate their execution state based on an explicit representation (i.e. a reification) of the communication details that led to the inconsistent state. This explicit representation permits them to take proper actions to reverse (part of) the computation.

We can identify various levels of delivery guarantees for non-blocking communication. A first one is build into the many-to-many invocations library [KB02]: using asynchronous messages for all communication and no delivery guarantees. This paradigm is very light on resources and fitting if delivery guarantees are not mandatory. When there is no process listening, all the messages we send will vanish. A second opposite approach, the actor model, demands that all asynchronous messages that are send, are also received. [Agh86] It's apparent that this will be much more resource intensive and not sensible or even possible in mobile networks.
We can deduce that no ideal message delivery guarantee policy exists and a

trade-off will have to be made between available resources and necessary requirements of the application at hand. Programming languages classified by the Ambient-Oriented paradigm should make this trade-off possible instead of forcing a single strategy. Explicit control over the communication traces allows one to make the trade-off between different delivery guarantees.

2.3.3 Ambient Acquaintance Management

Contrary to the client-server communication models we have no need for a third party to interact between hardware devices. They are autonomous and resources are dynamically detected while they wander, which conveys that they share the same capabilities without mediation of a server. In order to abandon the use of explicit references to each other (whether directly or via a server) we need what is known as distributed naming. [Gel85] One example of an implementation can be found in tuple-based middleware, where a process can publish data in a tuple space, which subsequently can be read out by another process based on pattern matching basis. Another style is incorporated by many-to-many invocations [KB02], where we can broadcast to all objects implementing a certain interface. Distributed naming is especially important in the context of ad hoc distributed systems, because it delivers a means to communicate when the addresses of the processes are not known ahead of time.

Although it is totally feasible to setup a server (just) for a special application, an Ambient-Oriented Programming language should permit applications to depend on distributed naming if the situation calls for it. In short: the acquaintances of an object must be dynamically manageable.

2.3.4 Conclusion of the discussion

The three properties established above define what constitutes an Ambient-Oriented Programming language. As they are derived from the hardware phenomena introduced in 2.2, it is possible to conclude that given the scope of applications targeted for interacting mobile devices, they are clearly necessary. Furthermore we can advocate for their sufficiency by the fact that in order to send a message, one must a) establish an acquaintance relation between communicating parties, b) have primitives to exchange the message, and c) have primitives to manipulate the message in order to deal with

results, or failure to deliver the message. These aspects are all embraced by the above considerations. Therefore the three properties are arguably necessary and sufficient to support future ambient applications, although it remains hard to prove.

The current state of the art in distributed languages does not comply with the characteristics of AmOP. The non-blocking communication characteristic is violated by languages for local area networks. Languages for open networks (typically the internet) tend to fulfill the non-blocking communication characteristic. They often do not support ambient acquaintance management and never provide a full reification of the communication as a basis for reversibility provisions.

2.4 AmbientTalk

AmbientTalk, which is the artifact we have explored and extended for this dissertation, is a first scion of the AmOP programming language family. AmbientTalk is a small expressive language allowing the programmer to deal with the conceptual problems of writing software for devices connected by wireless networks without having to meddle with their low-level technicalities [Amb06]. The language is based on the actor model of concurrency [Agh90]. Such actors (see figure 2.1) are objects with an associated thread that communicate and synchronize by sending each other asynchronous methods. Such incoming (respectively outgoing) messages are stored in mailboxes (FIFO queues) and are processed (respectively transmitted) whenever possible. Due to these mailboxes, the sending and receiving actor can be easily distributed (even on networks with unpredictable connections) since messages can be sent (although not transmitted) to actors even if they are not available.

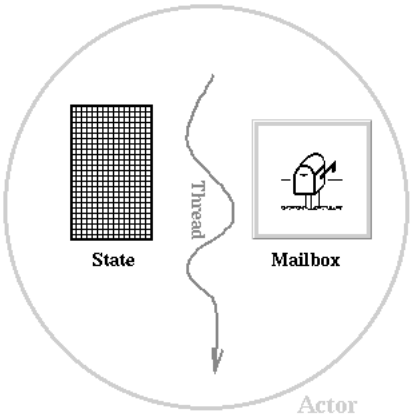


Figure 2.1: A graphical presentation of what constitutes an actor.

2.4.1 Ambient Actor Model

The Actor Model, first proposed by Hewitt [Hew77] and expanded by Agha [Agh90], complies with non-blocking communication characteristic of the paradigm, because all communication is asynchronous between actors and there is no explicit receive operation in the model. The Actor Model is thus a potentially valuable starting point for constructing an AmOP language. However the model has its limitations with regards to the remaining two characteristics of an AmOP language.

Ambient Acquaintance Management is not supported since an actor depends on other actors to acquire acquaintances. This implies that the entire configuration of actors needs to be set up beforehand, which is impractical in open and especially mobile networks. An extension to the actor model, namely the ActorSpace Model [AC94] enables distributed naming by a grouping mechanism called spaces. However these spaces are managed by centralized authorities and are therefore unable to cope with network partitions. The dependency on centralised authorities violates the autonomy characteristic and therefore makes the ActorSpace Model not suitable for mobile networks.

Reified Communication Traces are not supported by actors or even ActorSpaces, as a consequence of the guarantee stated by the model that

all messages eventually will be delivered. In response to a network partition it may be necessary to retract messages that were sent, but not yet transmitted (cf. section 2.3.2). Due to the delivery guarantees imposed by the actor model, such reflective access to the communication of an actor (as prescribed by the AmOP paradigm) is impossible.

2.4.1.1 Ambient Actor Model

The AmbientTalk language in fact uses an extension to the Actor Model called the Ambient Actor Model [Ded04], which addresses the lack of ambient acquaintance management and reified communication traces with the introduction of explicit mailboxes.

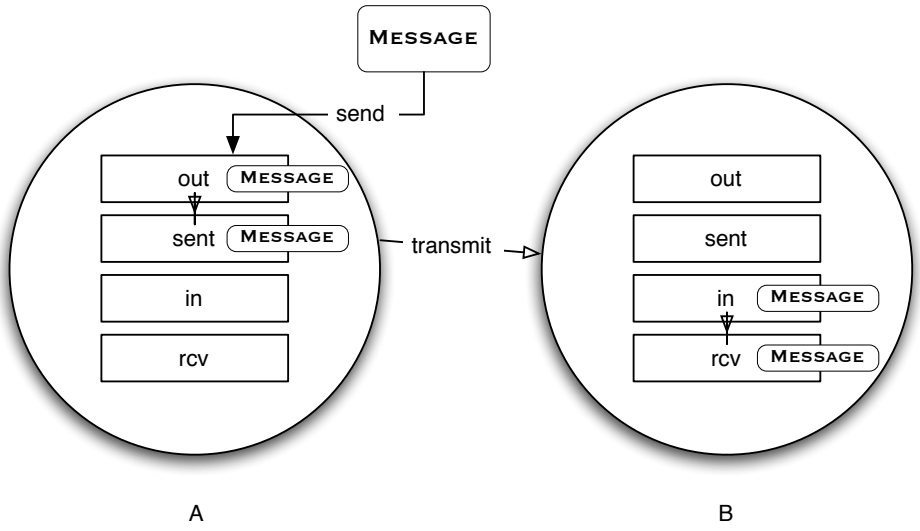


Figure 2.2: Illustrates the communication state of an actor.

Communication State: The Ambient Actor Model offers explicit control over an actor's communication state. If we examine the communication between two actors, we can differentiate four types of messages and each will be put in another mailbox (shown in Figure 2.2). The first are the messages, the actor has send, but are still waiting to be transmitted in the mailbox "out". A second type are the ones that an actor has send and that are transmitted in the "sent" mailbox.

Third are the messages that are received, processed and remaining in the "rcv" mailbox. And the last kind are those that an actor has received but still needs to process. They will end up in the mailbox "in".

The distinction with the regular Actor Model lies in the fact that these four mailboxes are not implicit but reified and accessible. Whereas the "in" and "out "mailboxes are implicitly present in the Actor Model to facilitate non-blocking communication primitives, their content cannot be modified, making it impossible to define one's own delivery guarantees and reversible computing support as argued for in the Ambient-Oriented Programming paradigm in section 2.3.2. Since they are fully reified in the language, the "in" and "out" mailboxes describe the continuation of an actor, as these two mailboxes contain the messages that will be processed and transmitted in the future. The remaining mailboxes "sent" and "rcv" reveal the communication history of an actor and are essential for undoing the effects of certain messages. The combination of all four produces a gateway into both the past and future computation of the actor.

Aside from all those standard mailboxes, each actor can create custom mailboxes. A message can occupy numerous mailboxes at the same time. Its delivery status can be monitored and altered, e.g. if we remove a message from the mailbox "out", it will never be transmitted. This way the "in" and "out" do not only represent the future of processing or communication, they even allow us to manipulate it.

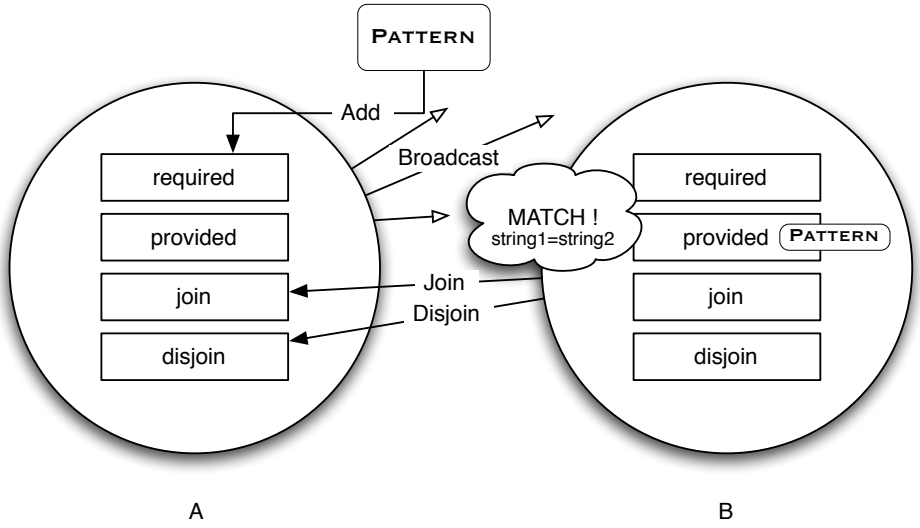


Figure 2.3: Illustrates the discovery process with mailboxes when the patterns match.

Ambient Acquaintance Management: The concept of mailboxes also introduces a service discovery mechanism to allow actors to get acquainted with one another (illustrated in figure 2.3). Whenever an actor (A) requires a particular service, it will add a pattern describing the service to its "required"-mailbox. The interpreter will then ensure that the request is (periodically) broadcasted such that an interpreter hosting another actor (B), which has the same pattern (a string), in its "provided"-mailbox will match, so that the actors can be introduced to one another. Whenever a match occurs the actor requiring the service, will receive a join message and the required pattern along with the provider will be added to the "join"-mailbox until a disjoin-message arrives, signaling that the service is no longer provided by the actor or the connection failed. At the arrival of the disjoin-message the pattern along with the provider will be moved to the "disjoin"-mailbox. Collectively these four mailboxes also provide us a detailed view on all the requested and provided services and their history.

2.4.2 The Object System

Within the scope of a single actor, AmbientTalk introduces objects, which appear to exist in a purely sequential language. The object model of AmbientTalk is based on the one of Pic% [DMWJ03] (pronounced as Pic-oh-oh), a small multi-paradigm (functional and object-oriented) programming language. It's a dynamically typed prototype-based language, which is exemplified below by the implementation of the prototypical boolean objects:

```
true: object( {
    new():: { this() };
    ifTrue(code()):: { code() };
    not():: { false };
    and(exp()):: { exp() };
    or(exp()):: { this() }
});

false: object({
    new():: { this() };
    ifTrue(code()):: { void };
    not():: { true };
    and(exp()):: { this() };
    or(exp()):: { exp() }
});

test: random(true, false);
test.ifTrue({
    display("The value is true.", eoln)
})
```

Figure 2.4: Implementation of the prototypical boolean objects in Pic%

Methods are invoked using the dot-operator. Names are declared using either ':' or '::' . The former declares variables, while the latter declares constants. These two types of declarations are also aligned with the visibility rules of the slots in an object [DMWJ03].
Methods can have two kinds of parameters: normal parameters and functional parameters . The actual kind, of a parameter, is syntactically visible in the definition of the function (the use of '()' behind the argument). In the case of normal parameters, the argument is evaluated and the associated value is bound to the formal parameter before the body of the function is evaluated. In case of functional parameters their arguments will be delayed.

One of the key features that makes AmbientTalk an extensible language is its special call-by-name parameter passing technique [DMWJ04], but again the details for this would lead us too far for the purpose of this dissertation.

2.4.3 Integrating the Actor Model

On top of the object model of Pic% we require three more pieces to make AmbientTalk an actor based language: one construct for creation of actors, another one for message sending and last but not least a construct for manipulation of actors. We will illustrate through the following example:

```
counter: actor( {
    n: void;
    new(aNumber):: { copy(n:=aNumber) };
    increment():: { become(counter.new(n+1)) };
    decrement():: { become(counter.new(n-1)) };
    get(customer):: { customer<-result(n) };
    init():: { display("initialized as actor") }
});

mycounter: actor(counter.new(5));
mycounter<-increment();
mycounter<-decrement()
```

Figure 2.5: Implementation of a counter in AmbientTalk

2.4.3.1 Creating new actors

Creating an actor in AmbientTalk is done with the actor primitive, which takes one argument: the actor's behavior. The resulting actor will receive an init-message, which we will use later on in the examples of chapter five, to initialize an actor. The passing of the object to the primitive actor is done by copy in order to avoid data sharing between two actors. The expression' actor(counter.new(5))' from the example, wraps a clone of the counter object (instead of a reference) in an actor entity.

2.4.3.2 Message sends

Message sending to an actor is done by the operator $<-$ (syntactic sugar for #). For example, the expression mycounter$<-$increment() sends the asynchronous message increment to the actor referred by the variable mycounter. The return value of a message is void as function calls are asynchronous. If a result is expected, this should be sent back by means of a callback method. When passing arguments, all objects are passed by copy to avoid sharing

them between different actors which would introduce race conditions on their internal state. Actors can be passed by reference since they are by default shielded from concurrent accesses since they process incoming messages (which are buffered in their inbox) one by one.

2.4.3.3 Changing the state and behavior

The become statement is used in AmbientTalk to change the state and behavior of an actor. It takes one argument: an object that will process all future messages. In the example the state is updated via the become primitive after an increment or decrement message. A clone of the counter, with an updated state, is used to replace the previous one. [2]
Since the 'become'-statement may be used to install new behavior inside an actor, messages that arrive at an actor that the current behavior can't handle will remain in the inbox . The moment become is used to alter the behavior (instead of the state) and the new behavior supports that message, it will get processed.

2.5 Service Discovery

Service discovery protocols are network protocols which allow automatic detection of devices and services offered by these devices on a computer network. It is the goal we aimed for in section 2.1, where we wanted to find a printer. AmbientTalk already has its ambient acquaintance management as we explained in 2.4.1.1, but it completely relies on two patterns to be equal. Since a pattern consists of merely a string, the protocol only permits text-based matching. There is no support for preferences, the use of history, logical operators or even subtype-matching, and this is exactly the demand we are attending in this dissertation.

2.6 Conclusion

This chapter has started with an outline of the Ambient Intelligence vision along with a concrete scenario that we will explore in the remainder of this

[2] Note that when only the state needs to be updated it is not necessary to use the become primitive since the underlying object model allows for imperative programming. In the example become was used rather than plain assignment for didactic purposes.

thesis. Subsequently we have introduced the essential characteristics of the involved hardware as well as their repercussion on the software side. This has culminated in the Ambient-Oriented Programming paradigm which was embodied in the AmbientTalk programming language.

In the remainder of this dissertation we will explore various existing mechanisms for service discovery. The goal of this study is to be able to enhance the service discovery mechanism of the language AmbientTalk which is based on a text-based matching of patterns in two mailboxes.

3

Service Discovery

In our present-day network environments we face two main problems: the first is the expansion of computing environments in homes and offices through the ever growing numbers of printers, scanners, digital cameras, and other peripherals, integrated into networked environments. The second problem is the proliferation of mobile devices such as laptop and palm-sized computers, cellular phones and other portable gizmos. These devices all trade functionality for suitable form factors and low power consumption; they are therefore "peripheral-poor and as a result they must connect to proximate machines for storage, faxing, printing, scanning and internet access. [Ric00]

Due to these changes mobility and modularity have become the modern goals of system development to enable Ambient Intelligence (see section 2.1). The classical client server paradigm is hardly applicable to present networks anymore and is increasingly displaced by peer-to-peer approaches, allowing endless changes in network topology and turning the use of fixed infrastructure in to old-fashioned customs. [SGF02]

Network resources and application software do not follow the mobile users when they leave their offices or homes, or when they relocate to another

temporary office or home. Supporting true mobility of users will therefore require changing the way application software is advertised and discovered. [LH02]

To face the management of all these devices we are in need of service advertisement and discovery technologies which will enable yet richer ways of interacting with our environments, services and devices. For that, different consortiums and labs have developed a range of service discovery protocols that are the subject of this thesis.

First we will give a definition and explanation of what constitutes such a protocol, then we will expand on the basics underlying the process. Thereafter three small scenarios will illustrate the context, followed by an overview of the most important competing protocols with a comparison. That leaves us only to touch some issues that are being addressed in the research field, and to end with a summary and outlook to the future.

3.1 What is Service Discovery ?

Service discovery protocols are network protocols which allow automatic detection of devices and services offered by these devices. They are supposed to possess the following three properties:

A Discovery Mechanism to detect services and service users, to find each other on a network. Here we can identify two types of service discovery architectures: registry-based (e.g. Jini) and peer-to-peer (e.g. UPnP). A registry-based architecture has a third entity, called the "registry". A"manageR" registers its services at it and "users" discover the services through unicast queries to the registry. In the peer-to-peer architecture there are no registries, and users discover managers through broadcast or multicast queries. The registry-based architecture reduces network traffic and makes a network more manageable by allowing registries to keep track of arriving and departing services. [SHdH$^+$05] The peer-to-peer architecture avoids single point of failures and bottleneck problems, that may surface in the registry-based architecture, but on the other hand increases network traffic.

Mobile and open so we have to include techniques to detect changes in component availability and maintain a consistent view of components in a network. This can be achieved by either monitoring periodic announcements (a heartbeat algorithm) or by persistently resending unacknowledged messages up to some bound, issuing a remote exception if the bound is exceeded (bounded retries). In AmbientTalk the former solution is implemented as well as in Jini and UPnP.

Description language to provide the means to describe services, so that the service user can determine if a discovered service matches his/her requirements. There are three major approaches to describe resources: the first is XML, which is widely used for webservices but is bandwidth intensive (cf. UPnP). The second is the use of interfaces (cf. M2MI), which is the most effortless approach but also the least expressive. The third way to describe a service is the use of attribute-value pairs, like the functional units in Salutation. We will combine the benefits of all three and propose a new description language for AmbientTalk, that will enable us to write richer descriptions without the performance drawbacks, in a straightforward manner.

In the following sections we will expand on the properties, we just introduced.

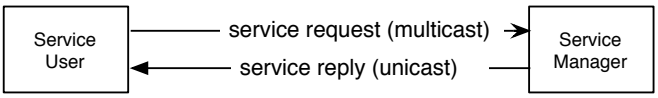
3.1.1 Discovery mechanism

For services to be able to find each other a discovery process has to take place, and as stated before we can differentiate between two underlying architectures: registry based (three-party) and Peer-to-Peer based (two-party). [DM01] Service discovery protocols define three entities, one of which is optional. All entities are software components that are distributed on a network:

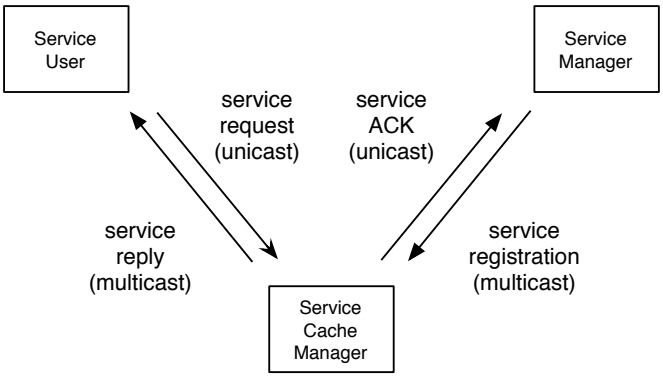
1. The service manager holds information about one or more services or devices, together with their attributes and interfaces.
2. The service user queries for a certain service or device it wants to use, and selects the most appropriate one found in the results.
3. Optionally, a service cache manager can act as a broker between service managers and service users, in order to reduce network traffic and

to increase performance. However, the design should ensure that all discovery activities are possible in the absence of service cache managers, because not all of the existing protocols support service cache managers (e.g. AmbientTalk).

In two-party-architecture multicasts are necessary for every discovery process, whereas in three-party-architecture multicasts from service users and service managers are only necessary for initial discovery of the service cache manager. That is why the latter case is desirable for larger networks. A graphical presentation of the two possible architectures is given in the figure 3.1.



(a) Two-party



(b) Three-party

Figure 3.1: Discovery architectures

Three different kinds of discovery can be distinguished: aggressive, lazy and directed. In aggressive discovery the service user sends out multicast requests, either a fixed number in fixed intervals or until it has discovered enough service managers or service cache managers.

The service managers or service cache managers send a unicast reply to the requestor if they can provide the requested service, with the given attributes. Aggressive discovery is normally used when a node has just joined the network, in order to discover the existing services for the first time.
In lazy discovery, service managers and service cache managers advertise their services in fixed intervals by multicast communication. Service users and service cache managers can store the received data for later use of the advertised services. Lazy discovery is useful to detect changes in component topology during operation. In directed discovery a service user contacts a service manager or service cache manager directly in order to probe for a previously advertised or discovered service. [Dyr03]

3.1.1.1 Registry-based

In a registry-based discovery process every service manager holds a collection of service descriptions. Each of these descriptions must be registered with all the discovered service cache managers (registries). The manager and cache manager have to negotiate about the lease time, the time after which the registration is removed if there was no renewal. From the moment a service is registered any service user can discover it, or can register its interest in receiving notifications about changes concerning the registered service. The reception of these notifications is also tied to a negotiated period of time. The registry-based architecture reduces network traffic and makes a network more manageable by allowing registries to keep track of arriving and departing services.

3.1.1.2 Peer-to-peer based

A two-party architecture consists of two component types: a service manager, a service user (and no service cache manager). Users discover managers either through broadcasts or multicast queries. The peer-to-peer architecture avoids single point of failure problems, as may exist in registry based architectures, but increases network traffic. It is also the preferred choice for ambient networks composed out of resource-poor devices. A limited battery for example favors against service cache managers, because you don't want to waste your own power on caching results for other people's queries that don't concern you. This is also the main reason why AmbientTalk is in the peer-to-peer based camp.

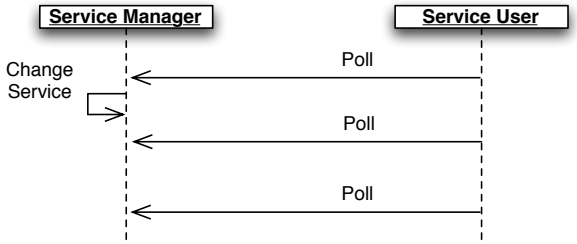
3.1.2 Mobile and open

Since we are dealing with distributed systems, new services can be deployed, obsolete ones can be removed, nodes and links may appear, disappear or fail. [DM02] Thus, a consistent view of all services on the network can not be guaranteed. It doesn't matter if we are tackling the discovery with a registry or a peer-to-peer based solution, consistency is an important issue, and we also have to address failures in a volatile network.

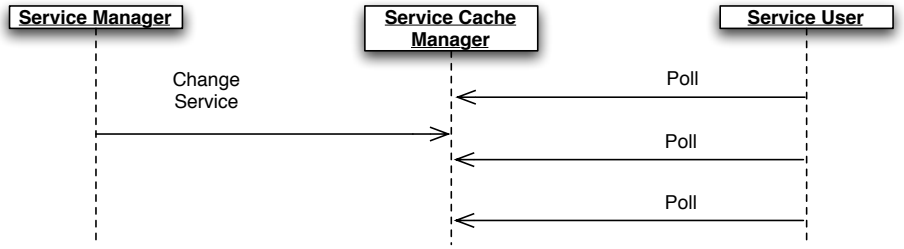
3.1.2.1 Consistency Maintenance

After logging onto a network and the initial discovery of all available services, a service user has to ensure that his knowledge about existing services stays consistent with the actual distributed state. There are two basic mechanisms for that: polling and notification. With polling, the service user initiates receiving updates, whereas with notification the service managers propagate changes as they occur. [Dyr03]

Polling With polling, a service user sends queries to obtain up-to-date information about a service that was previously discovered. In a two-party-architecture (shown in figure 3.2) it sends these queries directly to the previously discovered service managers and receives the responses via unicast. In UPnP for example we use the "HTTP Get request" mechanism to poll the service manager to retrieve a service description associated with a specific URL. In response, the manager provides a list of all supported services, including their relevant attributes. In a three-party (shown in figure 3.2) architecture polling consists of two processes. Service managers propagate changes concerning provided services regularly to the service cache managers and each service user queries its relevant service cache manager.



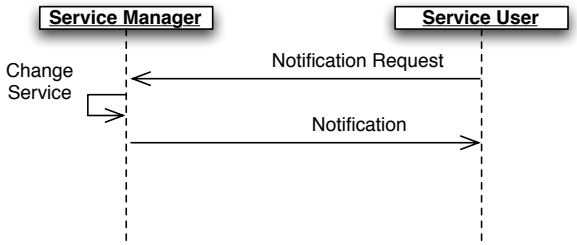
(a) Two-party



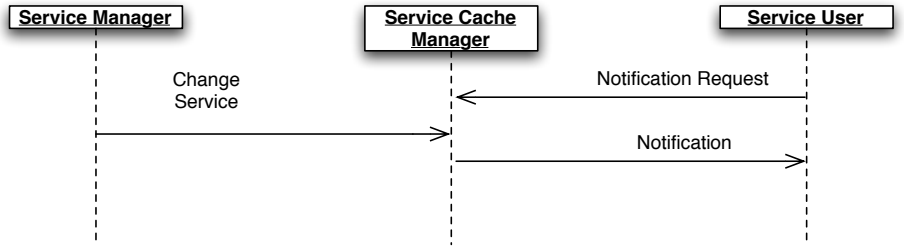
(b) Three-party

Figure 3.2: Consistency maintenance by polling

Notification When using the notification mechanism (illustrated in figure 3.3), changes in service descriptions are propagated from the service managers to the service users immediately after an update occurs. In a two-party architecture a service user has to register with the service manager. This registration is only valid for a negotiated time and needs to be renewed regularly. The service manager then subsequently announces changes to all service users that registered with it. In UPnP this is called event-subscription, the service user sends a subscribe request, and the service manager responds by either accepting the subscription, or denying the request. The subscription, if accepted, is retained for a TTL (time to live), which may be refreshed with subsequent subscribe requests from the user. In three-party-architecture service managers announce changes to the service cache managers. The cache managers do not need to register for that purpose. The service users register with the service cache manager in order to be notified about changes as soon as the cache manager receives them.



(a) Two-party



(b) Three-party

Figure 3.3: Consistency maintenance by notification

3.1.2.2 Failure Detection and Recovery

Up until now we have only considered topology changes that happen consciously and intentionally, and components are able to signal their sign-off. But when we reflect on the volatility of ad-hoc networks we also have to cope with changes due to failures. Hosts, processes and network links may fail, packets can get lost on the network, transmission can be jammed, hosts may move out of range and so on.
To deal with these failures correctly, a service discovery protocol must provide means to detect them and to recover after their ending. For that reason, two basic mechanisms exist: soft state persistence by monitoring periodic announcements and application-level persistence by bounded retries and remote exceptions. [Dyr03]

Soft State Persistence with Periodic Announcements Discovery protocols define key messages that components send out in fixed intervals to periodically announce their current state. Monitoring these key messages

empowers other components (listeners) to cache almost real-time states, i.e. they can store the information and overwrite it with every update. To detect communication failure, i.e. if a component does not receive such a heartbeat message from a remote component within the given interval, it may assume that the communication path or the remote component itself has failed. In order to keep a consistent view of reachable components, the listener deletes the cached information after non-appearance of an expected status update. As soon as the remote component or the communication link to it is back up and a new heartbeat message is received, the listener regards the remote component as available again.

Application-level Persistence with Bounded Retries Another means to detect a failure are bounded retries. This mechanism is widespread in networking. If a component does not answer a request, the request is resent several times. If the number of trials exceeds a given bound, the client assumes that the remote component has failed and throws a remote exception to the application layer (which is the application that wants to use a service). This mechanism is reasonable in systems where discovery is normally initiated by applications. It is automatically given in discovery protocols that presume reliable network communication, since the transport layer reports the inability to send data by definition.

The application has several possibilities to deal with such a remote exception:

1. It can ignore it. This is reasonable for polls and notifications since they recur periodically.

2. It can retry the operation after a certain period of time and thus recover from the failure as soon as the next communication attempt is successful. Until then it must assume that the remote component is not reachable.

3. It can discard knowledge about the remote component. If the remote component is a service manager (two-party architecture or poll from service cache manager in three-party architecture), knowledge about its service descriptions is discarded. This corresponds to the soft state persistence mentioned in 2.1 and expects the peer entity to send a

notification when it is back up and reachable. If the remote component is a service cache manager (three-party architecture), knowledge about its existence is discarded, possibly making it necessary to discover other cache managers.

3.1.3 Description language

To facilitate the discovery process, services are semantically described following a certain description language. Service requests are also expressed using the same description language. So protocols provide a data scheme to represent a service, this is called the service description.
A description is composed by an identity, a type and possibly more refining attributes. The identity is mandatory, required to be unique and has to contain a location of the service, i.e. its network address. The service type, also mandatory, explains what type of service is being described. The description can include attributes that characterise the service more precisely, like a user interface for example.

3.1.3.1 Ontology

A description language on itself is not enough, we also need an interpretation for our language. Concretely this means that for a client to be able to use a service, it expects to get a reply pointing to a suitable service after it poses a query for that service. For example, the client who launches a query for "printer" should get a reply that points to a printer service. This requires that both the service and the client agree on the meaning of "printer". As computers can do nothing but symbol computing, this agreement is actually one between humans who create the query and those who create the service description.

First, a human forms a conceptual model about a real world object or an object type (a set of objects sharing common features). Second, shared symbols are created among humans to represent the conceptual model. We call this set of shared symbols with agreed meanings a shared ontology. The ontology reflects the shared conceptual model of the service, which includes what a service is capable of doing (e.g. the functional interface of the service), the terms in which the service is described (e.g. the data types for describing the service) and the meanings of the terms (e.g. what they stand for and

what operations are allowed on them). This shared ontology is passed to software through the efforts of programmers or software users. Therefore, the software produced will show behaviors consistent with human's conceptual model. For the service discovery to work, the client of a service and the service itself share a common ontology on the service representation, which is ultimately shared between humans who create the service description and the query.[Yan01]

There are three main approaches for service representation:

3.1.3.2 XML

Service descriptions and queries can be specified in eXtensible Markup Language (XML), by which we gain the flexibility and semantic rich content of this self-describing syntax. XML allows the encoding of arbitrary structures of hierarchical named values; this flexibility allows service managers to create descriptions that are made for their type of service, while additionally enabling "subtyping" via nesting of tags.
Valid service descriptions have a few required standard parameters, while allowing service providers to add service specific information (e.g. a printer service might have a color tag that specifies whether or not the printer is capable of printing in color). An important advantage of XML is the ability to validate service descriptions against a set schema, in the form of Document Type Definitions (DTDs). Unlike a database schema, DTDs provide flexibility by allowing optional validation on a per tag granularity. This allows DTDs to evolve to support new tags while maintaining backwards compatibility with older XML documents.
Services encode their service metadata as XML documents and in a registry-based architecture register them with the service cache manager. Typical meta-data fields include location, required capabilities, timeout period, and Java RMI address. Clients specify their queries using an XML template to match against, which can include service-specific tags. A sample query for a color postscript printer and its matching service description are presented in figure 3.4.

Figure 3.4: An example XML query (A), matching service description (B), and failed match (C).

Alas XML also has some major downsides:

Need to process XML: due to the very limited processing power and restricted memory of mobile devices this itself can pose a problem.

Verboseness of XML: XML and SOAP are far more verbose than some formats in use. Their verboseness can pose a crucial problem over mobile networks where bandwidth can be limited or vary greatly.

Lack of mature standards: some of the Web Services standards lack maturity and various other needed domain-specific standards do not yet exist.

3.1.3.3 Interfaces

A service can also be described by its interface type and possibly other attributes. This is the most straightforward approach and we can structure services with the inheritance of the programming language used to write the interface. In M2MI for example, service discovery is done by interface

(shown in 3.1) matching, in Jini we have Java attribute matching in addition to its interface matching lookup protocol.

The downside of the interface approach:

Lack of expressiveness: we can only convey so much with interface. In an environment with a substantial amount of services we need more precise queries/advertisements, to find what we are looking for.

Communicate with java interface: that it is completely based on the Java programming language, thus locking out all devices that are not able to provide the power and resources required to host a virtual machine (and the JINI application itself).

An example:

Listing 1

```
public interface StorageService extends Remote {
    public boolean open(String username, String password, boolean newAccount)
          throws RemoteException;
    public boolean close(String username, String password)
          throws RemoteException;
    public boolean shutdown(String username, String password)
          throws RemoteException;
    public boolean store(String username, String password, byte[] contents,
          String pathname) throws RemoteException;
    public byte[] retrieve(String username, String password, String pathname)
          throws RemoteException;
    public boolean delete(String username, String password, String pathname)
          throws RemoteException;
    public String[] listFiles(String username, String password)
          throws RemoteException;
    public String name() throws RemoteException;
}
```

Listing 3.1: A remote file storage facility interface

3.1.3.4 Attribute-value tuples

Attribute-value pairs are a fundamental data representation in many computing systems and applications. Designers often desire an open-ended data structure that allows for future extension without modifying existing code or data. In such situations, all or part of the data model may be expressed as a collection of tuples <attribute name, value> ; each element is an attribute-value pair. Depending on the particular application and the implementation chosen by programmers, attribute names may or may not be unique.

This system also has its limitations:

No subtyping: in the first two techniques we could have a relation between the descriptions by subtyping, but attribute-value tuples lack this feature.

Lack of expressiveness: attribute-value descriptions also have their boundaries. We need a richer language to label services.

As an example we have the functional unit, a basic building block in the Salutation [Con] architecture. It is the minimal meaningful function to constitute a client or service. A collection of functional units defines a service record. For example, the functional units [Print], [Scan], and [Fax Data Send] can define a fax service. Each functional unit is composed of a descriptive attribute record, specified in ISO 8824 ASN.1

3.1.3.5 Problems with the existing service descriptions

Absence rich descriptions: These services are defined in terms of the their functionalities and capabilities. The functionality and capability descriptions of these services are used by the service clients to discover the desired services. Attribute matching is a very important component for finding out the proper services in such an environment. The existing service discovery infrastructures lack expressive languages, representations and tools that are good at representing a broad range of service descriptions and are good for reasoning about the functionalities and the capabilities of the services.

Absence inexact matching: In the Jini architecture, service functionalities and capabilities are described in Java object interface types. Service

capability matchings are processed in the object-level and syntax-level only. [Yan01] For instance, the generic Jini Lookup and other discovery protocols allow a service client to find a printing service that supports color printing, but the protocols are not powerful enough to find the geographically closest printing service that has the shortest print queue. The protocols do exact semantic matching while finding out a service. Thus they lack the power to give a 'close match' even if it was available.

3.1.3.6 Environment Queries

Environment queries, the language construct proposed in this dissertation, address the absence of rich descriptions. By employing a logical programming language to describe and query services we gain the advantages of XML with its flexible and expressive syntax, without the disadvantages for resource-poor devices.

No existing protocol allows both a service description with arbitrary complex attribute types and a set of meaningful comparison operations based on the semantics of those attributes. We need those capabilities to make our printer scenario possible:

Imagine yourself walking through a computer lab, in your hand a PDA and on it a document you would like to have a hardcopy of. You are in a hurry and just want a quick draft, so you select a dpi [1] of minimum 300. Meanwhile your PDA has discovered all the matching nearby printers, even arranged them by your previous preferences like speed, distance from your position, size of the print queue and so on. The most suitable one has already been selected as default, you just press the print-button and promptly you are shown directions to the printer where you will find your document.

3.2 Scenarios

In this section we showcase a few more scenarios to illustrate the need for service discovery protocols.

[1] Dots Per Inch/Pixels Per Inch. The resolution of an image or how many pixels are defined in the boundary of a square inch. From Cadmus Professional Communications

Scenario 1: Imagine finding yourself in a taxi cab without your wallet. Fortunately, you have a Jini technology-enabled cellular screen phone, and your cellular provider uses Jini technology to deliver network-based services tailored to your community. On your phone screen, you see a service for the City Cab Company, so you download the electronic payment application to authorize payment of your cab fare. The cab company's payment system instantly recognizes the transaction and sends a receipt to the printer in the taxi. You take the receipt and you're on your way.

Scenario 2: Consider an insurance salesman who visits a clients office. He wants to brief new products and their options to the client which are stored in his Windows CE Handheld PC. Since his handheld PC has wireless network and supports UPnP, it automatically discovers and uses an Ethernet-connected printer there without any network configuration and setup. He can print whatever in his H/PC or from computers in his main office and promote the new products.

Scenario 3: Consider an intelligent, on-line overhead projector with a library client. After identification to the system, the user may select a set of electronically stored charts or other document(s) for viewing. Rather than bringing foils to a meeting, the user accesses them through the LAN server in the library.

Scenario 1 is a Jini demo scenario told by Sun and scenario 2 is a UPnP scenario by Microsoft. The last one is from Salutation. [LH02] When we consider these scenarios for the first time, they all seem to share the same plot line: mobile devices, zero configuration and impromptu community enabled by discovery protocols, and cooperation of the ambient resources. Without trademarks such as Jini and UPnP, we could hardly know which scenario is told by whom. Even though they are direct competitors , these three rivals have different origins, underlying technologies, flavors, and audiences. They take different approaches and see the problem from different angles, therefore all protocols have pros and cons, especially compared to others. In the next section we will line them up and compare them with open mobile networks in mind.

3.3 Protocols

Now that we have attained a thorough insight into the general design of a service discovery protocol, we will have a look at the existing protocols. In those protocols different strategies are being pursued by different people, and none of them are widely employed so far. Examples of such platforms include SLP, Jxta, UPnP, Jini and many more. While all these share a number of common traits, they each have their own distinguishing features. However, careful examination of these protocols reveal shortcomings that we believe will inhibit the development of applications that exploit ambient environments.

In this section we will discuss these shortcomings and propose a new solution that brings together the advantages of current service discovery and interaction technologies and provide a new discovery protocol that we consider to be better suited for the development of ambient oriented applications. This review of protocols is specifically targeted towards mobile and open environments, where applications will be required to interact with a wide range of services and devices, that can disappear and reappear at any time.

A short overview:

M2MI: Makes use of a omnihandle, that enables us to send message calls to every object implementing the interface of the call. It is not a very demanding protocol, it also has dynamic proxy synthesis at runtime, but only features interface matching and you have to write your own network administration (addresses, routing). [KB02]

Jini: Jini works with a central service repository, handles requests with a proxy object and lets you download the rest if necessary. It supports remote events and leasing of services. The downside is that it requires a JVM and it only gives us an interface and attribute equality matching. [Inca]

JXTA: Jxta is a set of open protocols allowing us many choices. The discovery can be done by broadcast, invitation, cascading or rendezvous. We can group peers and create a virtual network of peers. It's language and network independent but only supports Java for the

moment. Another disadvantage, for performance reasons, remains the use of XML for descriptions.[Incb]

SLP: SLP divides its functionality between user agents(request), service agents(advertise) and directory agents (cache). It can assure integrity and authenticity of messages and supports logical operators in queries as well as substring matching. It lacks event notification and the directory agent could impose a single point of failure if we don't replicate. [SLP]

UPnP: UPnP features its own HTTPMU and HTTPU protocols and provides us with self configuration (cf. AutoIp, DHCP, Ipv6) and remote notification. But it lacks a service repository and makes a lot of use of bloated XML descriptions and SOAP calls. [UPn]

UDDI: UDDI is a web-based distributed directory made for the web and dividing advertisements between white(basic), yellow(categorization) and green(technical) pages. It is platform independent but is only useful for webservices, and also utilizes xml. [UDD]

Salutation: Salutation features rerouted advertisements and makes use of functional units instead of XML. It has a transport interface [SLM-TI] for achieving service discovery protocol independence in applications. There is also the "lite" version for PDA's and other small devices. It lacks lifetime of services and there is no discovery event system. [Con]

SDP: The protocol provides some consistency support, it is the only one that supports browsing of services, but lacks expressiveness and scalability. [SDP]

AmbientTalk: AmbientTalk has a lot of support for open mobile networks (e.g. notification, the heartbeat algorithm), but in the current version only uses strings to describe services. [Amb06]

In the following sections we will describe the nine protocols in more detail. First up are the registry based (three-party) protocols, thereafter we will turn to Peer-to-Peer based (two-party) protocols.

3.3.1 Registry-based

3.3.1.1 UDDI

The Universal Description, Discovery and Integration (UDDI) protocol is a cross-industry effort driven by major platform and software providers, as well as marketplace operators and e-business leaders within the OASIS standards consortium, including Microsoft, IBM and Ariba. It's platform-independent, XML-based distributed registry for businesses worldwide to list themselves on the Internet. UDDI enables businesses to publish service listings and to discover each other and to define how the services or software applications interact over the Internet. A UDDI business registration consists of three components:

1. White Pages: address, contact, and known identifiers
2. Yellow Pages: industrial categorizations based on standard taxonomies
3. Green Pages: technical information about services exposed by the business

Discovery mechanism: UDDI is one of the most popular Web Services standards. It is designed to be interrogated by SOAP[2] messages and to provide access to Web Services Description Language (WSDL) documents describing the protocol bindings and message formats required to interact with the web services listed in its directory. So discovery is done by using SOAP API's to query and to publish information to a UDDI registry as we explained in section 3.1.1.1

Mobile and open: This protocol is designed for webservices and thus has very little or no support for the problems an ambient environment poses, as it wasn't designed for this purpose.

Description language: The services are presented by XML and the queries are done in SOAP, which both have the performance problems we described earlier in section 3.1.3.2, the moment we use them on resource-poor mobile devices.

[2] Simple Object Access Protocol SOAP is a lightweight XML based protocol for exchange of information in a decentralized, distributed environment.

UDDI	**Advantages**	**Disadvantages**
	Platform independent	No support mobile open networks
	Rich Descriptions	Performance XML and SOAP

Table 3.1: Advantages and disadvantages UDDI

3.3.1.2 Salutation

Salutation was developed by a consortium of more than thirty companies, the most important of which are IBM, HP, Sun and Cisco. Salutation focuses on platform and network independency. Therefore it does not require TCP/IP[3] but works on top of any transport layer protocol. Other than managing discovery and advertising of services, it also handles access to components by providing a transparent communication pipe. There isn't that much information left about Salutation since the consortium dissolved in 2005 and the official website is down.

Discovery mechanism Salutations service cache manager is called a 'Salutation Manager' and is mandatory. Other than broking requests, it handles all communication between clients and servers, network independently. To realize that, it relies on one or more Transport Managers, at least one for every network protocol. So the Transport Manager is an abstract communication layer and a Salutation Manager can act as a proxy between components on different network types.
Services register with one Salutation Manager; clients request services only from one Salutation Manager. If a Manager does not offer a requested service by itself, it asks other Managers for the service.
The Salutation Consortium has elaborated Functional Units, which are classes of devices and applications provided by server components, e.g. print service or scan service. For every Functional Unit, the consortium has also defined a fixed set of attributes.

The Salutation Manager is involved with four kinds of processes: Service

[3] The Internet protocol suite is the set of communications protocols that implement the protocol stack on which the Internet runs. It is sometimes called the TCP/IP protocol suite, after the two most important protocols in it: the Transmission Control Protocol (TCP) and the Internet Protocol (IP), which were also the first two defined.

Registration, Discovery and Availability are familiar from the earlier description of registry based service discovery. The main difference here is that every client and every server communicates only with one assigned Manager. Service Session Management is the process of handling communication between one client and one server. A session can be established in one out of three modes: In Salutation Mode, the Manager does not only forward packets but also defines formats that are used in the session. In Emulated Mode the Manager just forwards packets, whereas in Native Mode client and server communicate directly (which is only possible if they operate on the same transport protocol) over a proprietary application protocol.

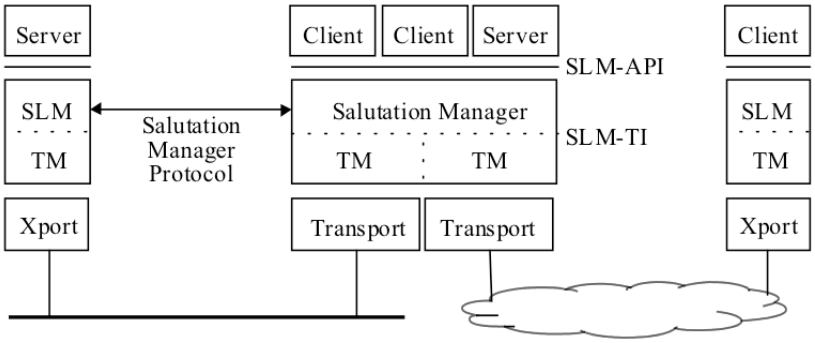


Figure 3.5: The Salutation Architecture

Mobile and open: To handle service availability, a client application can ask the local Salutation Manager to periodically check the availability of services. This checking is done between the local manager and the corresponding manager.
Salutation-Lite is a scaled down version of the Salutation architecture targeted at devices with small footprints. Salutation-Lite adapts well to low bandwidth network such as IR and Bluetooth. The Salutation Consortium envisioned that Salutation-Lite would have tremendous applicability to small information appliances such as palm-size and handheld computers (i.e. Palm and WinCE devices), but as the consortium is dissolved this is no longer the case.

Description language: Functional units define the services being advertised and discovered, such as printing ,faxing, and document storage. They

specify a set of attributes which characterizes the service provided, such as printing capabilities, faxing speeds, storagelimits, and cost of use. The functional unit is the minimal meaningful function to constitute a client or service. A collection of functional units defines a service record. For example, the functional units [Print], [Scan], and [Fax Data Send] can define a fax service. Each functional unit is composed of a descriptive attribute record, specified in ISO 8824 ASN.1 .

Salutation	**Advantages**	**Disadvantages**
	Network and protocol independent	Consortium is dissolved
	Rerouting advertisements	No lifetime services
	Lightweight descriptions	Too simple descriptions

Table 3.2: Advantages and disadvantages Salutation

3.3.1.3 Jini

Jini is an extension of Java and has been developed by a consortium lead by Sun Microsystems. Other members are AOL and many mobile equipment vendors. It enables devices supporting Java to connect with and provide services to each other. Jini is an open standard, the only prerequisite is a Java Virtual Machine running on the device. Sun provides a Jini Technology Starter Kit on its web site [Inca].

Discovery mechanism: The design is very similar to the one described earlier in 3.1.1.1 . Although Jini does not make a difference between service users and service managers, any component can lookup and invoke services from any component. The Lookup Server, just another name for the service cache manager, holds a Lookup Table and is optional. In case of absence of the Lookup Server, components operate their own lookup table and we retain a peer-to-peer based protocol.
The lookup table contains pointers to services and Java-based mobile programme code. Thus, the result of a discovery process is not only a URL to a service, but a service proxy (a Java interface) that can be accessed directly, much like a driver in traditional architectures. The three operations supported by the Lookup Table are ‘store’, ‘match’ and ‘fetch’, which

correspond to service registration, service lookup and the download of the service proxy, respectively.
Accessing remote components is done by Java Remote Method Invocations (RMI). This is very specific for Jini and distinguishes it from all other service discovery protocols presented in this section. The lookup service maps interfaces seen by clients to set of service proxy objects. A client downloads the service proxy, which is actually RMI stub that can communicate back with the server. This proxy object enables the client to use the service without knowing anything about it. Hence, there is no need for device drivers.
Jini also introduces the concept of a group, to subdivide networks with large numbers of components into administrative scopes. In order to avoid conflicts, Sun recommends using domain name style (e.g. printers.vub.ac.be). Components can be part of zero or more groups.
The process of discovering a service cache manager is called discovery. The registration process is called join. They do not work any differently than described in 3.1.1.1.

Mobile and open: The protocol has a lot of support for a dynamic network with volatile connections:

Leasing: a form of soft-state service registration and service usage. Here a service must periodically advertise itself to the lookup service in order to keep its registration alive. Likewise, when a client is using a service, it must periodically advertise to the lookup service that it still requires the service. This covers two forms of failure, one where the service becomes unavailable without informing the lookup service, or where a client using a service goes down, without first informing the service provider and releasing the service. In the second case, the service is released after a certain period and becomes available for use by another client.

Remote events: These are used by both clients and services to signal a change in their state. A simple example is a PDA and a networked printer. When the PDA connects to the system, there may not be any available printers. When a printer joins the Jini system, the client is notified using a remote event, to signal that the PDA can now use a print service.

Transactions: One of the most difficult problems to address in distributed systems, is the handling of system crashes involving non-idempotent operations. Transactions provide a solution to this by allowing operations to be grouped, where either all operations pass, or all the operations fail.

Description language: Services in Jini are represented by serialized objects encapsulated to form an entry stored in the lookup service or JavaSpace. Each entry provides a simple comparison functionality to allow the service to be matched against a search template. Attributes for the service and search template are simply specified as object member variables.
One benefit of the Jini approach is that it permits matching against subtypes, which is analogous to matching subtrees in XML. A disadvantage of the model is that it requires a Java interface object to be sent over the network to the lookup service to act as the template, such representations cannot be stored or transported as efficiently as other approaches.
Because we compare interfaces, the discovery is liable to false negatives due to class versioning problems. It is also an unscalable technique of using Java and Java objects: any service that wishes to participate in the Jini system must have an according Java-coded proxy object.

Jini	**Advantages**	**Disadvantages**
	Mobile and open support	Class versioning problems
	No need for drivers	Unscalable java
	Fairly rich descriptions	Not expressive enough

Table 3.3: Advantages and disadvantages Jini

3.3.1.4 SLP

The Service Location Protocol (SLP) is the widest spread and one of the most lightweight of the presented protocols and manages not only the discovery but also gives access to services. It was developed by the IETF (Internet Engineering Task Force) SvrLoc working group, the most important members are SUN, HP, Novell, IBM and Apple. This working group also provides two reference implementations.

SLP is vendor and platform independent, it is built on top of TCP/IP as network protocol. For most communication the unreliable and packet oriented protocol UDP[4] is used. TCP is only used where data does not fit in one datagram. Protocol messages are mixed binary and string-based, whereas binary representation is mostly used for headers and string representation for service descriptions. SLP is a very scalable discovery protocol, intended to serve enterprise networks.

Discovery mechanism: The protocol consists of the same entities as described in 3.1.1 and supports both the two- and three-party-architecture. It names the entities User Agent (UA), Service Agent (SA) and Directory Agent (DA). The DA can be announced via DHCP (Dynamic Host Configuration Protocol) or configured statically at the client side. SLP2 names aggressive and lazy discovery: active and passive. There may be no DA in small networks. In this case the UA's service request message is directly sent to SA's. A service identity SLP2 defines a Service URL containing a service type, host address, port number and path. For example a printer service at the VUB: 'service:printer:lpr://prog2.vub.ac.be:515/lpr02'. The set of service attributes is called Service Template here and also consists of attribute-value-pairs. Service Templates have to be registered with IANA (Internet Assigned Numbers Authority). SLP is said to be a solution to the intranet service discovery needs, but it scales well to larger networks. The scalability is supported by various features such as the minimal use of multicast messages, scope concept, and multiple DA's. Services may be placed into administratively assigned scopes. A scope is not more than a string that groups a number of services into a collection to aid scalability. We can assign one or more scope identifiers to each client, they will act as filters, restricting the client to detect only services within those scopes. Here is an example:

```
Attributes = (Name=Igore),(Description=For developers only),
             (Protocol=LPR),(location-description=12th floor),
             (Operator=James Dornan \3cdornan@monster\3e),
             (media-size=na-letter),(resolution=res-600),x-OK
```

[4] A connectionless protocol that, like TCP, runs on top of IP networks. Unlike TCP/IP, UDP/IP provides very few error recovery services, offering instead a direct way to send and receive datagrams over an IP network. It's used primarily for broadcasting messages over a network.

Mobile and open: This protocol also supports a simple service registration leasing mechanism: SLP includes a leasing concept with a lifetime that defines how long DA will store a service registration.
The protocol also scales very well to larger networks, which makes it ideal for coping with the rapid increase of numbers of (mobile) devices.

Description language: One of the most interesting aspects of SLP is its structure for describing service information. Services are organized into service types, and each type is associated with a service template that defines the required attributes that a service description for that service type must contain. The functionality and expressiveness of this framework is almost an exact mapping onto the functionality of XML: each template in SLP provides the same functionality as an XML DTD. Queries in SLP return a service URL. SLP supports service browsing and string-based query for service attributes which allow UA to select the most appropriate service from among services on the network. The UA can request query operators such as AND, OR, comparators (=, ¡, ¡=, ¿, ¿=), and substring matching. This is more powerful than all the other description languages. For example, in Jini, service attribute matching can be done only against equality.

SLP	**Advantages**	**Disadvantages**
	Scalable	Possible single point of failure
	Rich Descriptions	No event notification
	Lifetime	

Table 3.4: Advantages and disadvantages SLP

3.3.2 Peer-to-Peer based

We have addressed all the registry-based protocols now and will continue with the Peer-to-Peer alternative service discovery protocols.

3.3.2.1 M2MI

Many-to-Many Invocation (M2MI) is a paradigm devised at the Rochester Institute of Technology. M2MI provides an object oriented-method call abstraction based on broadcasting: objects send multicasted messages to other

listening objects.

Discovery mechanism: M2MI is an extension for Java, that allows objects to implement a particular interface. When an M2MI invocation is called on that interface, then every object that implements the interface will receive a broadcast message to call this method. M2MI allows for messages to be sent via handles. A handle denotes an abstract set of objects to which a message will be broadcast. [KB02] M2MI messages are routed to other M2MI-aware objects using M2MP (Many-To-Many Protocol) which broadcasts the message via the wireless network. The discovery mechanism is designed for wireless networks, in which broadcasting is more natural than routing messages from device to device.
The M2MI layer synthesizes remote method invocation proxies dynamically at run time, which will handle M2MI messages, eliminating the need to compile and deploy proxies ahead of time (cf. RMI[5]). Objects wanting to be exposed to external M2MI messages are exported to the M2MI layer. Therefore we don't need central servers, nor network administration, even complicated resource-consuming ad-hoc routing protocols are not required, and system development as well as deployment are simplified. There are three varieties of handles: omnihandles, unihandles, and multihandles. We will illustrate the use of a unihandle and a omnihandle.

Listing 2

```
// Exporting Player object into the M2MI layer
M2Ml.export(player_l, Player.class);

// Creating handles
Player allPlayers = M2MI.getOmnihandle(Player.class);
Player aPlayer = M2Ml.getUnihandle (aPlayerObject, Player.class);

// calling the method move on all Players and once more on aPlayer
allPlayers.move();
aPlayer.move();
```

[5] RMI is a mechanism that is part of the Java programming language. It allows Java objects to invoke methods on objects from another JVM.

Listing 3.2: Example unihandle and omnihandle

Mobile and open With M2MI no server must be set up, or system properties provided. Devices can come and go at any given moment and they will become part of the ad-hoc network. But M2MI is only a low-level framework, if we want consistency management or failure detection and recovery, we have to write it ourselves.[Gup05]

Description language An M2MI-based application is built by defining one or more interfaces, creating objects that implement those interfaces in all the participating devices, and broadcasting method invocations to all the objects on all the devices. We can only describe so much with interfaces, more precise queries and descriptions are necessary when the number of provided services expands.

M2MI	**Advantages**	**Disadvantages**
	Not a very demanding protocol	Write own network administration
	No centralized server	Only interface matching
	No network administration	

Table 3.5: Advantages and disadvantages M2MI

3.3.2.2 JXTA

Sun Microsystems has founded an open community-project called JXTA (abbreviation for Juxtapose) [Incb]. The objective of the project is to try to create a common platform that helps developers in building distributed P2P services and applications in which every device and software component is a peer and can easily interact with other peers. JXTA technology is basically a framework on top of which developers can build their own applications, without worrying about low-level details that are instead provided by the underlying JXTA layer.
JXTA defines a set of simple, small, and flexible mechanisms that can support peer-to-peer computing on many platforms. The platform consists out of the following entities:

Peer Groups: These allow grouping of peers in any useful fashion. They are deliberately not clearly specified, and could represent e.g. a collection of services, a geographical group and so on.

Messages: These are datagram style messages, so they can be used on unreliable, asynchronous and unidirectional transports such as IP. They can also be used on reliable, synchronous and bidirectional transports such as TCP.

Pipes: These connect between peers and are used to send messages. They can be one-to-one, many-to-many, etc. They are also bound at runtime, allowing the possibility of being rebound if errors occur. They are used as the only communication mechanism.

The JXTA platform is defined by the following six protocols: Peer Resolver Protocol (PRP), Peer Discovery Protocol (PDP), Endpoint Router Protocol (ERP), Pipe Binding Protocol (PBP), Rendezvous Protocol (RVP) and Peer Information Protocol (PIP).

1. PRP: allows a peer to send a search query to another peer.
2. PDP: allows a peer to discover other advertisements.
3. ERP: allows a peer to query for routing information to route messages through the network.
4. PBP: allows a peer to bind a pipe endpoint to a physical peer.
5. RVP: is the mechanism by which services are bootstrapped within the network.
6. PIP: allows a peer to query for the current status of another peer.

JXTA advertisements are XML encoded resource descriptions. Although several predefined types of advertisements exist, extended or new types of arbitrarily nested advertisements may be introduced. Advertisements can be discovered on all or certain nodes of a peer group, depending on attribute-value pairs that must exist in matching advertisements. Peer groups are clusters of certain JXTA services provided by network nodes. By default only search queries using predefined attributes, like Type or ID, are possible. JXTA implements a DHT (Distributed Hash Tables) algorithm that

takes effect beyond a peer's local (multicasting based) neighbourhood. Additionally JXTA offers relays and the ERP to increase the number of peers efficiently. Relays can be used to bridge different physical or logical networks by forwarding messages on behalf of peers that can not directly address each other. The multi-hop ERP connects peers within the virtual JXTA network, even across firewalls.

Discovery mechanism: Peer discovery can be done in a variety of ways, first of all by multicast. This can be used to bootstrap a peer and inform it of other local peers, an other way is by unicast connection to a repository of peers. This can be used to bootstrap a peer and inform it of world-wide peers. The current implementation has a hard-coded set of repositories, but this will be fixed in later versions. There is also the possibility of requesting peer lists that are known by another peer or offering a peer list to another peer. All these peer discovery methods allow a dynamic set of peer relationships to be built up at the expense of network traffic, particularly at startup of a new peer.
The lowest level of searching for services is by the Peer Discovery Protocol. Peers are distinguished by being "ordinary" peers or by being "rendezvous" peers. Ordinary peers keep information about the services they offer. Rendezvous peers cache service adverts like service cache manager, so that they act as proxies for service adverts (just for the adverts, not the services themselves). Discovery works as follows: ask all the peers one hop away if they have the service, then ask the known rendezvous servers if they know of the service.The rendezvous servers may ask other rendezvous servers if they know of the service and the peer asking for the service must backoff for a certain time before making a repeat search.

Mobile and open: A peer is assigned a network address (this is called an Endpoint) during the initialization phase of the platform. JXTA does not provide any means to modify that address at runtime. If the peer leaves for a while for any reason and tries to reconnect to the network through another access point, it will not be able to join the peer community, since there is no way to update its Endpoint and the other peers cannot see it. As a consequence, JXTA is not recommended in highly dynamic environments where peers frequently log off, move and log on again.
In JXTA a Discovery Service is available in order to help peers to discover

each other. This service has been designed bearing in mind that it has to deal with fixed networks and with stable connections, available for most of the task execution time. The JXTA Discovery Service is based on the mechanism of publishing/discovery of advertisements containing the information that each peer wants to share (e.g. addresses, services, etc.). The advertisements are saved in a temporary cache and managed by a Service Cache Manager. Each advertisement will be automatically removed from the cache when its own lifetime has elapsed. Since JXTA relies on fixed and stable connections, in its existing implementation the timeout is set to a high value and there is no way for a user application to configure it. This is not acceptable in ad hoc network environments where frequent connections cause a lot of discovery requests.

Description language Services are described using WSDL. These can be used to describe any resource such as nodes or services peers. These are nodes of the JXTA network, and can speak the JXTA peer protocols. These can be used to describe any resource such as nodes or services. The protocol is specified as a set of XML messages. This means heterogeneous devices with completely different software can interoperate with the JXTA protocols. The way a service is invoked is not prescribed except that they are invoked using JXTA pipes. The invocation mechanism through a pipe could be by SOAP, for example. But the same disadvantages of the use of XML remain as explained in section 3.1.3.2.

JXTA	**Advantages**	**Disadvantages**
	Cross platform	XML
	Many discovery strategies	No volatile support

Table 3.6: Advantages and disadvantages JXTA

3.3.2.3 UPnP

The consortium that developed UPnP was founded and is lead by Microsoft. Other important members are Intel, Compaq and Cisco. UPnP is published under a free license. It's the most well-known commercial product with an implementation in Windows XP. As a consequence this is currently the widest spread service discovery protocol. Because of its simplicity and its

supporters on both the software and hardware market, we can expect UPnP to gain importance over the other service protocols in the future but only on the wired consumer market. UPnP targets home and small office computing environments.

UPnP is basically an extension of the existing Windows Plug and Play mechanism where components don't have to reside on the same host but rather have to be reachable via a TCP/IP network. HTTP-over-UDP is used for discovery and advertising, and SOAP is used for transactions.
The entities of the protocol are called Control Point, Device and Service. A Service is the smallest unit of control in the UPnP system. It represents any singular particular service being offered, for example printing. A device represents a collection of services and/or embedded devices. A control point is a controller, which is capable of discovering and controlling other devices. A client thus interacts with a service through their control point. For communication between devices, it uses the protocol stack shown in figure 3.6.

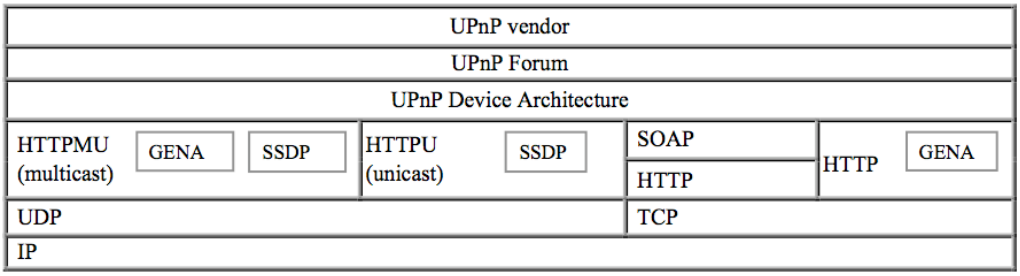


Figure 3.6: The UPnP Protocol Stack

Another important feature of UPnP is the automatic configuration of IP addresses being plugged in. Being introduced for this purpose, AutoIP enables a device to join the network without any explicit administration. When a device is connected to the network, it tries to acquire an IP address from a DHCP server on the network. But in the absence of a DHCP server, an IP address is claimed automatically from a reserved range for the local network use. So, named as AutoIP. The device claims an address by randomly choosing an address in the reserved range and then making an ARP request

to see if anyone else has already claimed that address.

Discovery mechanism: The Simple Service Discovery Protocol (SSDP) is the protocol used by UPnP for service discovery. This protocol is used for announcing a device's presence to others as well as discovering other devices or services. Therefore, SSDP is analogous to the trio of protocols in Jini: discovery, join, and lookup. SSDP is built on HTTP over multicast (HTTPMU) and HTTP over unicast UDP. A joining device sends out an advertisement (ssdp:alive) multicast message to advertise its services to control points. They are the potential clients of services embedded into the device. The other message of SSDP is search (ssdp:discover) multicast message sent when a new control point is added to the network. Any services, which match the request specification, will unicast a response.
The design only supports a two-party architecture, there is no caching entity and all components have to use multicast to advertise or discover services. That is the reason UPnP does not scale well on large networks.

Mobile and open: There are no mechanisms for consistency maintenance, which limits its usage to networks with reliable network communication. Thus the protocol is not adequate for open mobile networks.

Description language: As UPnP uses XML to describe its devices and services, it provides resource-rich description architecture, using an open, commonly used Internet standard. It allows for both UPnP standardised device types, along with non-standard device types. This allows for a broad range of services. However, before the description can be used, it must first be downloaded. While XML is the standard of choice for object description, it is verbose; thus, the client must perform a large download before it can view a services description set. The aforementioned messages contain a URL that points to an XML file in the network, describing the UPnP device's capability. Hence other devices, by retrieving this XML file, can inspect the features of this device and decide whether it fits their purposes. This XML description allows complex, powerful description of device capability.

UPnP	**Advantages**	**Disadvantages**
	Rich Descriptions	XML
	AutoIP	Not Scalable

Table 3.7: Advantages and disadvantages UPnP

3.3.2.4 SDP

The Bluetooth specification was developed by Microsoft, Intel and the most important mobile equipment manufacturers. The Bluetooth wireless technology is a relatively new short-range communication system designed for robustness, low power consumption and low cost. Its architecture describes all network layers from physical (radio transmission around 2,4 GHz) up to application layer specific topics like defining so called profiles (e.g. data synchronisation profile or telephony control profile) out of which applications may choose.
Service discovery is part of the Bluetooth protocol stack and forms an own sub-layer. Every device has an SDP server and an SDP client (which correspond to the service user and service manager in 3.1.1). Bluetooth networks are pico nets with a maximum of 256 devices, only eight of which can be active, i.e. can communicate, at the same time. Several pico nets can overlap but the core specification does not define any routing mechanism, so that neither discovering nor using services in a neighbouring pico net is possible. Because devices discover each other when joining the network, no service cache manager is necessary. So we have a simple request/response discovery in SDP, which allows searching for service type (Service Search Request) and attributes (Service Attribute Transaction) and browsing (ServiceSerachAttributeTransaction) all services available. The latter application is only reasonable in Bluetooth because of the limited number of devices in one network.
Bluetooth was designed to operate as small area network, with distances of approximately 10m. As a result, SDP was specifically designed to work with this. While attempts are undertaken to extend the range of Bluetooth, it is still far from the implementation stage.

Discovery mechanism: A client searches for a particular service with a service search request and service attribute request calls, known as SDP-PDUs. Any device participating in service discovery assumes one of the following roles. The first role is Local Device (service user) which implements the service discovery application and client portion of the SDP layer. The client also initiates SDP transactions. The second role is Remote Device (service manager). A Local Device using SDP transactions contacts this device. It implements the server context of the SDP layer, which replies to the SDP transactions. Only devices in the vicinity of the Local Device are able to participate in service discovery that are within a ten-metre radius.

Mobile and open: Notification about new devices, and thus new SDP servers becoming available or disappearing, is provided by other means of the Bluetooth architecture. Since Bluetooth is designed for ad-hoc use, all consistency maintenance is delegated to lower network layers. Bluetooth networks only support a maximum of 256 devices which is probably too low for future ambient environments.

Description language: All of the information about a service that is maintained by an SDP server is contained within a single service record. The service record consists entirely of a list of service attributes. Each service attribute describes a single characteristic of a service. Some examples of service attributes are ServiceClassID-List and the Provider Name. Some attribute definitions are common to all service records , but service providers can also define their own service attributes. A service attribute itself consists out of two components: an attribute ID and an attribute value.
Each service is an instance of a service class. The service class definition provides the definitions of all attributes contained in service records that represent instances of that class. Each attribute definition specifies the numeric value of the attribute ID, the intended use of the attribute value, and the format of the attribute value. A service record contains attributes that are specific to a service class as well as universal attributes that are common to all services.
Each service class is assigned a unique identifier ,this service class identifier is contained in the attribute value for the ServiceClassIDList attribute, and is represented as a UUID. A UUID is a universally unique identifier that is guaranteed to be unique across all space and all time. UUIDs can be inde-

pendently created in a distributed fashion. No central registry of assigned UUIDs is required.

An example of a service record:

```
Service Name: OBEX Object Push
Service RecHandle: 0x10000
Service Class ID List: "OBEX Object Push" (0x1105)
Protocol Descriptor List: "L2CAP" (0x0100)  "RFCOMM" (0x0003)
Channel: 9  "OBEX" (0x0008)
Language Base Attr List: code_ISO639: 0x454e encoding: 0x6a base_offset: 0x100
Profile Descriptor List:  "OBEX Object Push" (0x1105) Version: 0x0100
```

SDP	**Advantages**	**Disadvantages**
	Consistency support	No expressive descriptions
	Browsing services	Not scalable
		No advertising

Table 3.8: Advantages and disadvantages SDP

3.3.2.5 AmbientTalk

The service discovery protocol of AmbientTalk has already been described in 2.4.1.1. Figure 3.7 illustrates the protocol to refresh your memory.

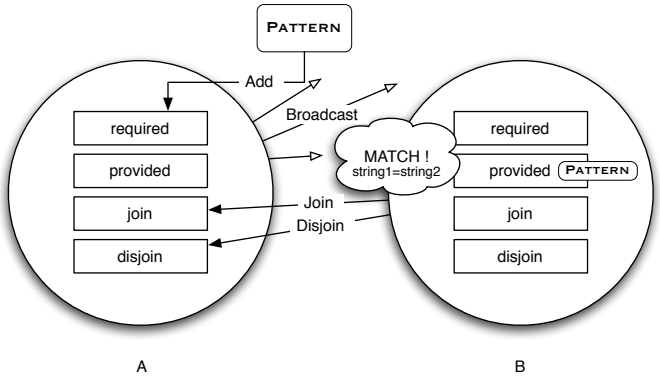


Figure 3.7: Illustrates the discovery process with mailboxes when the patterns match.

AmbientTalk	**Advantages**	**Disadvantages**
	Heartbeat Notification	String descriptions

Table 3.9: Advantages and disadvantages AmbientTalk Service Discovery

3.4 Comparison

All discovery protocols are fairly new and none of them has acquired extensive proliferation up till now. Each one of them is supported by important companies that are already implementing their protocol in products now. More and more mobile phones and PDA's support Bluetooth and Microsoft Windows XP implements UPnP. All protocols have different strengths and problems in different environments, so that probably not only one protocol will survive. However, different environments will favour one or a few service discovery architectures.

A lot of comparisons have been made [Dyr03] [FDW+04] [MPHS05] [Gee05] [Liv03] [Hel02] [Ric00] [CL02] and table 3.10 summarises the characteristics of the nine service discovery protocols presented before. Their suitability for mobile ad-hoc networks will be discussed subsequently.

Protocol	**Discovery mechanism**	**Mobile/open**	**Description language**
UDDI	registry	no support	xml
Salutation	registry	leasing	functional unit
Jini	registry/peer-to-peer	leases/events/transactions	interface
SLP	registry/peer-to-peer	leases	xml
M2MI	peer-to-peer	events	interface
JXTA	peer-to-peer	leases	xml
UPnP	peer-to-peer	no support	xml
SDP	peer-to-peer	no support	service record
AT	peer-to-peer	heartbeat/events	string

Table 3.10: Summary of all protocols

3.4.1 Discovery mechanism

Our protocols can be divided into two main techniques: registry-based and peer to peer based. UDDI and Salutation are the ones that only support the first system and are consequently not suited for ambient environments. In open mobile networks the composition of the network is too dynamic and the registry is a potential single point of failure.
Peer-to-peer is the most applied approach and handles the constant changing of the composition of an environment very well. We don't need central nodes and can form networks spontaneously. These are the same reasons why the service discovery protocol of AmbientTalk makes use of the peer-to-peer technique like Jini, SLP, M2MI, JXTA, SDP and UPnP.

3.4.2 Mobile and open

The three architectures that come from discovery approaches on top of traditional (hardwired) networks (TCP/IP) are SLP2, UPnP, UDDI. Since

TCP/IP itself is not designed for ad-hoc networks, these two discovery protocols might not be satisfying in ambient environments, where high mobility is likely e.g. a car network.
The other six protocols, Jini, JXTA, Salutation, AmbientTalk, M2MI and Bluetooths SDP, were particularly developed for the purpose of ad-hoc networking. Characteristics necessary for consistency maintenance and failure handling are becoming very distinctive in these protocols. Jini has the best support with its implementation of leases, events, and transactions.
In AmbientTalk (see 2.4) we already have e.g. a heartbeat algorithm, a notification mechanism and support for open mobile networks is growing. There is even some recent research into conversations for handling exceptions [MSW05] as counter part of the transactions in Jini.

3.4.3 Description language

There are three candidate types as a means to advertise and query services: XML, interface, attribute-value pairs. The first one brings forward the most rich descriptions for UDDI, SLP, JXTA and UPnP, but the verboseness of XML is not acceptable in ambient networks where bandwidth is scarce and/or fluctuates, and processing power of the devices is limited and battery capacity is precious.
Interfaces in Jini and M2MI provide more simple advertisements, which limits the detail of descriptions but they make querying and providing a service very straightforward in the programming code.
The last approach, the attribute-value tuples of SDP and Salutation, is also the most elementary. They give us a lot of freedom and ease of use but lack a way to structure hierarchies.
AmbientTalk originally represented a service only as a string, but in chapter five we will justify and explain my extension to the AmbientTalk's service discovery protocol: Environment Queries.

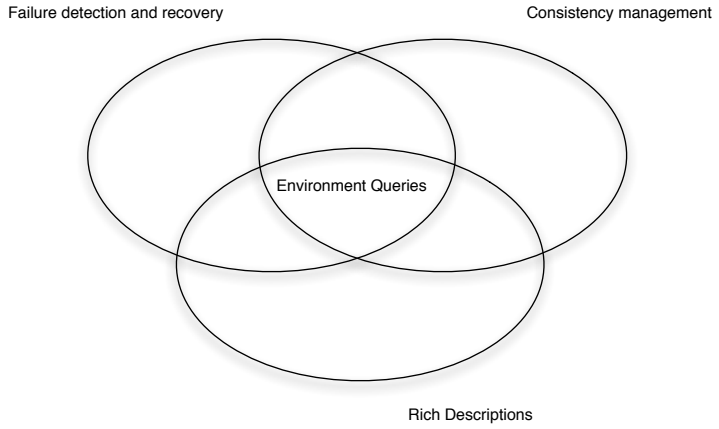


Figure 3.8: Venn diagram of environment queries characteristics

3.4.4 Conclusion for AmbientTalk

The service discovery protocol of AmbientTalk already supports most of the functional features that are proposed by all the other protocols, but one major aspect is absent: rich service descriptions. The concrete goal of my dissertation is to extend the protocol of AmbientTalk with a new description language that incorporates the advantages of current alternatives while keeping the restrictions of an open mobile network in mind and even introducing more expressiveness in service advertisements and queries (see figure 3.8).

3.5 Related Research

3.5.1 Personalization through the use of Meta-data, Reputation and History

One of the essential elements in personalized service discovery [JP04] is the ability to augment service descriptions with additional information, i.e. metadata. Service providers may adopt various ways of describing their services, access polices, contract negotiation details, etc. However, many resource consumers also introduce their own selection policies on the services they prefer to use, such as quality of service and reputation metrics [Ric]. It would make searching a lot more personal if we allow third parties to add

metadata to service descriptions, so that information about a service can be built up rapidly and used in discovery. Furthermore, it is useful to add such metadata not only to service descriptions, but also to any other concept that may influence the discovery process, such as supported operations, types of arguments, and businesses.

If we consider the way in which many services are used in real environments, we see that people personalise their services, e.g. I find this printer the fastest, this projector belongs to my research group and so on. These annotations are often personal or role based and may well change over time(e.g. a printer will always print on a certain size of paper which is part of its service description, but maywell move between rooms or be replaced by a faster model). We believe that personalised, group and public metadata will be important to the utility of service location and interaction protocols. We do not intend that everyone should be able to modify the service description of a device, rather that service location platforms provide hooks for linking into meta-information databases. This would enable users and client applications to search for services that have been recommended by their colleagues. [FDW+04]

Support for attributing meta-data concerning the usage history, access profile (history) and user experiences of a given service is not provided by any of the current service discovery protocols. We believe that this is a track that deserves more attention as it promises to help us cope with countless services that will be provided in an environment.

3.5.2 Caching

We have to consider where information about service managers is available. If available only at the service managers themselves, a high overhead in searching for this information is incurred. If information is cached at other points in the network, this raises questions regarding cache lifetime and mechanisms for (more or less aggressively) maintaining cache consistency. In addition, the volatility of service manager availability has to be taken into account when designing such a system.[FK04]
Cache consistency is usually maintained either by push-based (cf. notification) or by pull-based (cf. polling) approaches. In the former approach, the

data source is responsible for updating the content cached by service users, whereas in the latter approach, the service users must periodically ask for updates. [CD]

We have to make sure the time during which a the network stores inconsistent information is as short as possible. In the reactive case (with caching), a node learns about a new service manager only after it has overheard or processed a request/reply pair to this node. This happens soon at high request frequencies.
Periodic announcements, on the other hand, limit the time until the new service manager is known to the announcement interval. To match the reactive protocols performance in high-load cases, frequent announcements would be necessary, which are just wasted overhead in other cases. Additionally, the period of inconsistency for service managers that have become unavailable remains. Hence, the benefits of periodic announcements would depend on a careful tuning of announcement intervals.

When a service manager becomes unavailable, every cached entry on this service manager may cause incorrect replies. Here, the disadvantages of caching are clear, both for reactive and for periodically announcing systems. To limit incorrect answers, short cache lifetimes would be necessary, nullifying its effectiveness. Therefore, explicitly removing cached entries by negative announcements would allow to maintain caching benefits without, hopefully, imposing a too large administrative burden.

3.5.3 Semantic Discovery

Existing mechanisms for service discovery are not well suited for MANET's[6], since they either rely on central directory servers or produce a huge message overhead. More sophisticated approaches analyze the content of the service requests to route them semantically. Typically, they support rather primitive service descriptions only. We believe that for an efficient usage of services in MANET's (as well as in any network), service discovery based on

[6] A mobile ad-hoc network (MANET) is a self-configuring network of mobile routers (and associated hosts) connected by wireless links, the union of which form an arbitrary topology. The routers are free to move randomly and organise themselves arbitrarily; thus, the network's wireless topology may change rapidly and unpredictably.

semantically rich, ontology based service descriptions needs to be supported. [KoR02] [KKRO03]

To discover a service, the infrastructure should be able to describe its capabilities based on the functionalities that it provides. This description must be well defined, machine understandable and processable with minimal or no intervention from programmers, specifying not only the format of the information (using XML) to be exchanged but also its meaning. For example, two identical XML descriptions may mean very different things depending on the context of their use. A client and a service manager may use differing markups to describe the same thing. So, capability matching between both descriptions become unrealistic to expect advertisements and requests to be equivalent. The discovery mechanism may return a service that doesn't fulfill the needs of the requester or it may not return a service although it matches the constraints of the request. This limitation of capability matching is surpassed by performing matching at a semantic level.

Current web service technology based on UDDI and WSDL does not make any use of semantic information failing therefore to address the problem of locating web services. [Yu06] The solution of this problem requires a semantically rich language to express capabilities of services based on ontology such as DAML-S. Also in service discovery protocols designed for ad-hoc networks no semantic information is used, but research in this area is being done [TES05] [AT05] [CPAJ01].

3.6 Summary and Outlook

Since future networks will be much more dynamic than traditional (wired) ones, service discovery will gain more and more importance. The goal of service discovery mechanisms is to enable software components to find each other on such highly dynamic networks, like mobile open networks. Other than orchestrating advertising and discovery, consistency maintenance and failure handling are important challenges to service discovery.

At present there exists a variety of service discovery protocols, most important SLP, Jini, Salutation, SDP, UDDI, M2MI, JXTA, UPnP and off course

our own in AmbientTalk. We have compared these approaches and listed their advantages and disadvantages. Each of these protocols approaches the vision of service discovery from a different perspective and every one of them turns out to be suitable in different environments. We expect that most of them will continue to co-exist, and a lot more will rise up to the challenge of open mobile networks.

Mobility is an issue that has not yet been solved in a totally satisfactory manner in any of the protocols we described. It is thus expected that future versions will include improvements here. Another problem that needs more attention remains the description of services. In the time to come we will have to deal with a myriad of services and this can only be handled with personalization, semantic information and more expressive advertisements and queries. Our answer to the latter is environment queries, and these will be discussed in chapter five.

4

Logic Programming and Language Symbiosis

In the previous chapter we gave an in-depth analysis of the state of the art of service discovery protocols and we concluded that no solution tackles all the needs of open mobile networks. We particularly emphasized the need for a service description language that accommodates the proliferation of devices in open mobile networks. Such a description language needs to be equipped to cater for both expressive queries and service descriptions. In this dissertation, we propose the use of a logic programming language to model and query for services. The use of such a language yields shorter service descriptions augmented with a variety of new possibilities, which we will explain in this chapter. Loco is our logic programming language of choice and will be briefly introduced in section 4.1.4. After this we will discuss the symbiotic facilities of Symbioco [Leu05] , a symbiosis layer between Pic% (introduced in section 2.4.2) and Loco, which was used as an experimental platform to handle the link between AmbientTalk actors and Loco service descriptions.

4.1 The declarative programming paradigm

The declarative programming (also known as logic programming) is the name of a programming paradigm which was developed in the 70's. As an alternative to step-by-step description of an algorithm, we design the program as a set of logical facts and rules. Each procedure call is considered as a theorem of which we need to determine the truth, and this way executing a program means searching for a proof. In traditional imperative programming languages, the program specifies *how* a problem should be solved, the program is a list of steps to solve a problem. Contrary to the approach in logic programming where the program is a declarative specification of *what* the problem is, which is quite a different way of thinking. [PF94]

4.1.1 Queries on knowledge

In search of a proof for theorems (or procedure calls) the logic engine needs to be presented with a certain amount of knowledge about the world the problem is situated in. Consider the following example: suppose a logic engine is asked to prove that 4 is greater than 2. In order to prove this, the system needs to know what 'greater than' means. This knowledge needs to be specified in a rule base, which can be queried to prove theorems. As a result the equivalent of procedure calls in an imperative programming language are given the name 'queries' in a logic programming language, hence the title of my dissertation: Environment Queries. When we carry out a query, the system will search through all its knowledge for any information concerning the query. The result of this search may be twofold: facts and rules.

4.1.2 Facts and Rules

Fact: A fact is an unconditional truth, for example 9 is greater than 6'. If this fact is available in the rule base when the logic interpreter is asked whether 9 is greater than 6, it can see straightaway that this is the case. However if we ask if 7 is greater than 6, the rule base provides insufficient knowledge and the system is unable to prove this theorem. Having only facts at our disposal, we would be obliged to put an infinite amount of facts into the system before we can do simple number comparison. This problem is dealt with through the induction

of another kind of knowledge, rules.

Rule: A rule is a conditional truth, a conclusion which can only be drawn when its premises are known to be true. For example a number x is greater than a number y if x - y is positive, only when x - y is positive we can deduce that x is greater than y. If this rule is part of the rule base we can compare any two numbers we desire (presuming similar rules describing how to subtract and how to check whether a number is positive).

Programming with rules and facts requires quite a different way of thinking for people familiar with imperative programming. One of the differences emerges when one looks at the concept of a program variable. In imperative languages it is a name for a memory location which can store data of certain types. The contents of that location may vary over time, the variable itself forever points to the same location and is always well-defined. Contrary to a logic program, where it is a placeholder that can change to any value (as in the mathematical sense of a variable). For example if we ask the system whether X is greater than 6, it will try to find a value that makes this query true (e.g. X = 7). This procedure for finding a value for a variable is called unification.

4.1.3 Unification

Unification essentially is a binary operation whose purpose is to attempt to make its two operands the same. The process is similar to pattern matching (and we will use it in this dissertation to match service descriptions and service queries). If we carry out the query 'X is greater than 6' in a logical programming language, the system will first look up everything it knows about 'greater than'. If we presume it finds (or can derive) the fact '7 is greater than 6', in that case it will unify the operands in our query (x and 6) with the operands (7 and 6) of the fact we found. The unification of two values (6 and 6) means comparing them and in case they are equal the unification is successful. To be able to unify a value with a variable, we first have to check if it isn't already bound (ground). If it is so we unify the two values as described above, otherwise the unbound variable is assigned the value we started out with. When a variable is given a value through unification, that variable is then said to become instantiated or

bound to that value. It will never change anymore, during the remainder of this attempt to prove the theorem. The last case is the one where we need to unify two variables. When the first is still free, it will be assigned to the second variable. In the other case the value of the first variable will be unified with the second variable.

4.1.4 Loco

Loco is a declarative logic programming language. It was developed by Prof. Dr. Theo D"Hondt and was originally written in Pico, meant for educational use. Syntactically Loco looks a lot like Pico and Pic% , which has proven to be very advantageous for the symbiosis with Pic% [Leu05] and thus by extension for AmbientTalk as well. First of all, this section shows how the paradigm introduced in the previous section applies to Loco, while at the same time introducing Loco's syntax. We will therefore adapt the example presented in [PF94], which has a small part of the London underground as its universe of discourse (shown in figure 4.1 [1]).

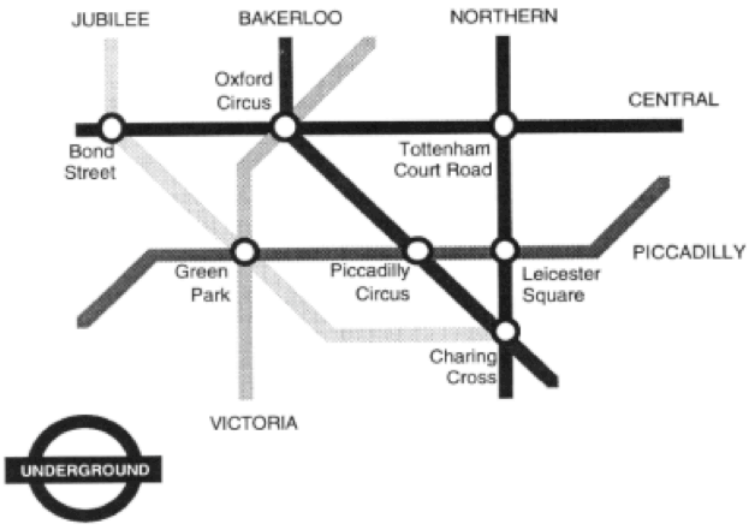


Figure 4.1: A map of the London Underground

This picture incorporates a lot of information including lines, stations, rel-

[1] Picture taken from 'Simply Logical: Intelligent Reasoning by example', by Peter Flach

ative distances and transit stations. Devising logic statements that capture the knowledge in the above picture can be done in a straightforward manner. The map can be represented by describing all the lines with 11 facts as shown in listing 4.3.

Listing 3

```
{
   connected("bondstreet", "oxfordcircus", "central");
   connected("oxfordcircus", "tottenham_court_road", "central");
   connected("bondstreet ", "green_park", "jubilee");
   connected("green_park", "charing_cross", "jubilee");
   connected("green_park", "piccadilly_circus", "piccadilly");
   connected("piccadilly_circus", "leicester_square", "piccadilly");
   connected(""green_park", "oxford_circus", "victoria");
   connected("oxford_circus", "piccadilly_circus", "bakerloo");
   connected("piccadilly_circus", "charing_cross", "bakerloo");
   connected("tottenham_court_road", "leicester_square", "northern");
   connected("leicestersquare", "charing_cross", "northern")
}
```

Listing 4.3: Connections of the London Underground

We can define two stations to be nearby if they are on the same line with at most one station in between. This relation can represented by the following additional facts in listing 4.4.

Listing 4

```
{
   nearby("bondstreet", "oxfordcircus"");
   nearby("oxfordcircus", "tottenham_court_road");
   nearby("bondstreet", "leicester_square");
   etc.
}
```

Listing 4.4: Connections of the London Underground

In total 16 new facts can be derived. Nevertheless we can also derive these nearby facts from the previous 11 formula in a systematic way. If two stations X and Y are directly connected via some line L, then X and Y are nearby. Alternatively, if there is some Z in between, such that X and Z are directly connected via L, and that same line L also connects Y. This can be formulated in logic as follows:

Listing 5

```
{
   nearby(X, Y) : connected(X, Y, L) ;
   nearby(X, Y) : connected(X, Y, L)  & connected(X, Z, L) ;
}
```

Listing 4.5: Rules for describing nearby.

In these formulas in listing 4.5, the colon (:) should be read as 'if', and the ampersand (&) between connected(X, Y, L) and connected(X, Z, L), should be read as 'and'. Other basic valid symbols are '!' for 'not', and 'l' for 'or'.

Native	Interpretation		Native	Interpretation
:	IF		get	$\geq$
&	AND		let	$\leq$
|	OR		minus	-
!	NOT		times	$\times$
plus	+		findall	find all matches
max	maximum		equals	test equal values
min	minimum		append	append to end
gt	$>$		member	check if member
lt	$<$		elementat	return element at position

Table 4.1: Basic syntax of the Loco programming language

Now we have two definitions of the nearby-relation, one (listing 4.4) which simply lists all pairs of stations that are nearby, and one (listing 4.5) in terms of direct connections. Logical formulas of the first type are called facts and the second type will be called rules. For each possible query, both give exactly the same answer. As we explained previously, they both represent the knowledge we need to handle queries. In the following listings a couple of queries and their results are listed.

Listing 6

```
> connected("greenpark", ""charingcross", "jubilee");
: ok
```

Listing 4.6: No free variables

This first query (listing 4.6) contains no free variables and the system will try to find a match, ultimately finding the fact 'connected("green park","charing cross","jubilee")' while searching through all the knowledge about connected. All the terms in the query match with those of the fact we found, so unification succeeds and 'ok' signals the success of the query.

Listing 7

```
> connected(""picadilly_circus"" , otherstation, line);
: ok
otherstation = "leiceister_square"
line = "picadilly"

> #
: ok
otherstation = "charingcross"
line = "bakerloo"
```

Listing 4.7: Two free variables

In the next query (listing 4.7) we have two free variables and we can interpret the query as: Find all stations connected to "picadilly circus" and the according lines connecting them. Again the system will try to find a match and unify the query with a fact in our knowledge base. We will find a fact connecting "picadilly circus", so in this case otherstation variable will be unified with "leiceister square" value and line will be bound to "picadilly". Again the query is successful and values for the two free variables are displayed.
Although there are more solutions for our query, only one will be shown. The reason for this being that the number of solutions may be infinite (e.g. a query asking for all the numbers greater than 3), so printing all of the solutions is very risky. To see more solutions we can enter # as a query, and the system will continue its search where it stopped before.

Listing 8

```
> connected("greenpark"", "tottenham_court_road", "unexistingline");
: No match found
```

Listing 4.8: No result

This is the first query (listing 4.8) that isn't successful. There is no connection between "greenpark" and "tottenham court road" station and certainly not by "unexistingline". The system will signal a failure by displaying: "No match found".

Listing 9

```
> nearby( "oxford_circus"", nearbystation);
: ok
nearbystation = "tottenham_court_road"
```

Listing 4.9: Unify with a rule

This last query looks similar to the second (listing 4.7), but unlike that second one it will unify with a rule instead of a fact. It will unify with the first rule for nearby (listing 4.5). First "oxford circus" will bound to 'X', then nearbystation will bound to 'Y', following the rule we need 'connected("oxford circus", nearbystation, L)' to be successful and to have the query signaling success. So the system will try to unify this new query with a fact and it will find 'connected("oxford circus", "tottenham court road", "central")' , which will match. As a result our last query also turns out to be a success.

In this section Loco was briefly described. First of all the declarative programming paradigm was introduced, after which Loco's syntax has been illustrated. Now that Loco has been introduced, we will move on to Symbioco, a symbiosis between Pic% (described in section 2.4.2) and Loco.

4.2 Language Symbiosis

In biology symbiosis is defined as "A close, prolonged association between two or more different organisms of different species that may, but does not necessarily, benefit each member" [LPG]. Mutualism is a symbiotic relation

in which the association is advantageous to each species, and outside the field of biology the most common symbiotic relation. Language symbiosis is a relationship between two programming languages, that allows them to access and use one another's concepts in a transparent way. In practice this means two (or more) languages are able to utilize each other's functionality. As concrete example we could have an object-oriented language which syntactically sends a message to an object, causing it to execute a query in a declarative language it's symbiotically related to. To express environment queries, we have extended Symbioco, an existing symbiosis layer between Loco and Pic% (AmbientTalk's object model) [Leu05].

4.2.1 Reason for symbiosis

Every so often it is easier to express a piece of program in another language or paradigm, which is exactly the opportunity symbiosis offers. It provides additional expressiveness, and this is precisely the same kind of richness we want for service descriptions and queries in service discovery protocol.

4.2.2 Symbioco

Symbioco is a symbiosis between Pic% and Loco, which was developed by Tom Leuse to explore aspect-oriented logic meta programming for a prototype-based language [Leu05]. His implementation exploits the fact that both the Loco-interpreter and the Pic% interpreter are written in a common language (Java), it contains a layer that can evaluate both Loco and Pic% expressions, using the respective interpreters. Aside from being able to start interpreters, the Symbioco layer will also capture all errors and output operations that arise in both interpreters. Therefore we are able to redirect output and take appropriate actions when certain errors occur. Responding to those errors is where the real symbiosis lies: when a function or logical formula is not found by the current interpreter, Symbioco will catch the error and start evaluating the same expression in the other interpreter.

To allow AmbientTalk to make use of Loco to express service descriptions and queries, the Symbioco layer needs to be extended with knowledge on how to interact with the AmbientTalk interpreter (figure 4.2) as well as extend the AmbientTalk interpreter with the proper hooks signalling when to switch between AmbientTalk en Loco. The precise mechanisms for doing so

are at present limited only to the service discovery algorithm and will be explained in the next chapter.

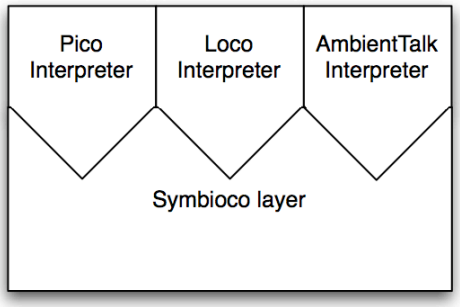


Figure 4.2: Graphical representation of the Symbioco layer

4.3 Summary

In this chapter we gave a brief introduction to Loco and to the declarative programming paradigm of which Loco is an example, after which we described symbioses as it is used in Symbioco. In chapter five it will turn out that loco is ideal for solving our need for a description language that handles the proliferation of devices in open mobile networks. In the next chapter we will also extend Symbioco to handle symbiosis between AmbientTalk and Loco.

5 Environment Queries

Now that we know what service discovery means for open mobile networks and we have identified the current solutions (section 3.3) for tackling this issue, we can discuss the benefits of using a logic programming language as a service description language for service discovery in open mobile systems. It has already been mentioned (section 3.4) that none of the current protocols cope sufficiently with the hardware phenomena in section 2.2, but now we will present a new protocol for AmbientTalk following the ambient-oriented programming paradigm (section 2.3). This new approach is based on a symbiosis with the logic programming language Loco (section 4.1.4) supporting very rich service descriptions and queries.

5.1 Conceptual Design

In this section we will explain the notion of Environment Queries and how it differs from the approach that was taken for the previous service discovery protocol of AmbientTalk.

5.1.1 Service Discovery For Open Mobile Networks

The Discovery Mechanism in our new implementation remains the same as in AmbientTalk's original service discovery protocol. It's peer-to-peer based with aggressive device discovery: multicasts are send in fixed intervals to all devices in range. Services are discovered through observers on required and provided mailboxes as described in section 2.4.1.1.

Mobile and open characteristics are supported by AmbientTalk's choice for peer-to peer, since a limited battery for example favors against service cache managers, and you don't want to waste your own power on caching results for other people's queries, that don't concern you. To maintain a consistent view of components in a network we need to detect changes in component availability, and this is achieved in AmbientTalk by monitoring periodic announcements (heartbeat algorithm).

Description language of AmbientTalk with only basic string-matching as explained in 2.4.1.1, is replaced by Environment Queries. These will address the absence of rich descriptions in the description languages of current service discovery protocols. By employing a logical programming language to describe and query services we gain the advantages of XML with its flexible and expressive syntax, without the disadvantages for resource-poor devices.

5.1.2 Strings vs. Environment Queries

In the old protocol we could only describe a service with a single string and as a consequence a query also consisted of only one string. These strings were compared and in case they were equal (no substring matching) we would have a match, and the requester a provider.
The new protocol on the other hand makes use of Loco syntax for service descriptions. Each actor(A, B, C, D) has its own Loco process for interpreting logic statements as shown in figure 5.1.

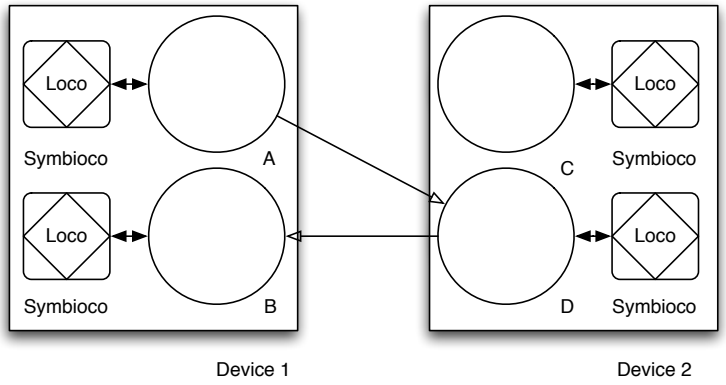


Figure 5.1: Incorporation of Loco process for every actor

We interact with the Loco process when we provide or require service in one of the actors. Let's take a closer look at these two phases.

5.1.2.1 Phase one: Providing a service

Providing a service materializes in adding a fact to the knowledge base of an actor (2). We encapsulate the fact together with an optional rule, in a new structure called a pattern. That pattern is then added to the provided mailbox(1), as it was done in the earlier protocol. Next all actors are notified (3) of the addition of a new pattern, so then can act accordingly.

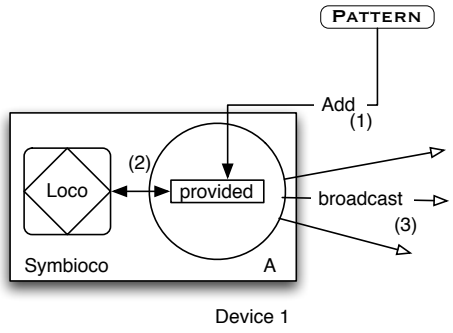


Figure 5.2: Providing a service

5.1.2.2 Phase two: Requiring a service

When we require a service, again we will construct a pattern, but this time it will consist out of a rule and a query. The pattern is added to the required mailbox (1) and its addition notified to all actors (2).

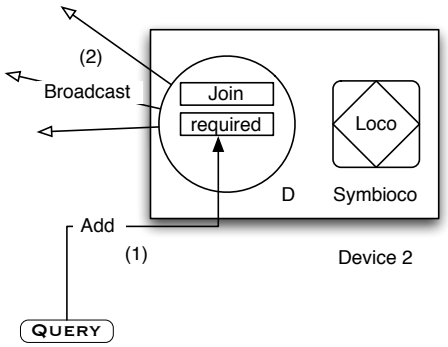


Figure 5.3: Requiring a service

There, the rule will be asserted (1) and if the query returns results, it will forward them back to the requesting actor through a join message (2). If a pattern is no longer provided, the pattern will be added to the disjoin mailbox.

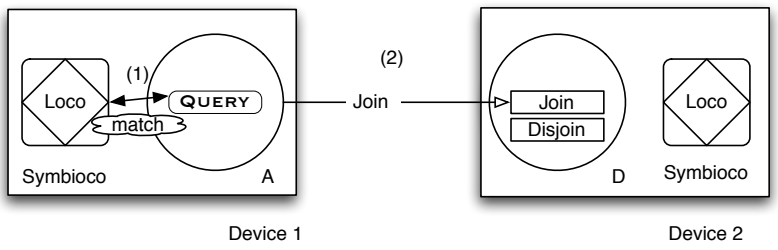


Figure 5.4: Sending back providers

5.2 Syntax and Examples

The syntax of AmbientTalk is extended with three new functions: require, provide and loco, which we will explain subsequently. Of course the full

syntax of Loco is also supported at this moment, but only as an argument for the new require and provide functions. Next we will show how the concept, introduced in the previous section, applies to AmbientTalk while at the same time introducing the new syntax.

It is time to go back to the example of the printer in section 2.1. The challenge was not simply finding a printer, but finding one that meets all the requirements. The concrete goal was to find a printer with a minimum of 300 dpi for a quick draft of our document, other previous preferences would be used to arrange the matches. In the remainder of this section we will introduce the required functionality to implement the printer scenario.

In all the following examples an actor 'a' and 'b' is created with its behavior as an argument. The init function is always called after creation, triggering the provide function in (a) and the require function in (b). Calling startNetwork() prompt devices to discover each-other, this is necessary since the two pieces of code are evaluated in two separate virtual machines. As illustration of success we have a simple test-function that displays the string: "The test worked!". The join function will send the test-message to all providers passed as an argument in the join messages, and display 'msg sent' to signal this event to the transcript.

5.2.1 Basic example

As a first example we will provide and require a printer with a dpi (= dots per inch) of 800 as depicted in figure 5.5.

```
a: actor({
    init()::provide("printer(dpi(300));
                     printer(dpi(800))");

    startNetwork();

    test()::display("The test worked!", eoln);

    loco(result)::display("Loco:", result, eoln)
})
```

(a) Provider

```
b: actor({
    init()::require("printer(dpi(800))");

    startNetwork();

    join()::{ joined: messages("joined");
        for(i:1,i<=size(joined),i:=i+1, {
            provider(joined[i])#test();
            delete("joined", joined[i]);
            display("msg sent", eoln)
        })
    };

    disjoin()::display("Disjoin detected: ", disjoined);

    loco(result)::display("Loco:", result, eoln)
})
```

(b) Requester

Figure 5.5: Basic example

The provide function takes one or two arguments, depending on the optional usage of rules. The first argument is string of rules written in Loco syntax, separated by comma-dots (;). The second and mandatory argument is string of facts, again separated by comma-dots (;). Together the rules and facts represent all the services that will be provided.
In (a) we add no rules but two facts, representing two services: a printer with a dpi of 300 and one of 800.

The loco function is merely a signaling function for mainly debugging purposes. Its default behavior is to write events of the loco process to the transcript, but this can be overridden by the programmer.

Require has almost the same syntax as the provide function, except for the second argument that is not the declarative description of a fact but of a query also written in the Loco programming language. This syntax looks very natural to programmers as it is identical to the way SQL[1] is integrated in all the common programming languages.
In (b) the query is identical to the fact in (a), because there are no free variables, and we show here that we can simulate the previous (string-matching) service discovery protocol.

[1] SQL (Structured Query Language) is a standard interactive and programming language for getting information from and updating a database.

In listing 5.10 we show the transcript of the virtual machine interpreting the behavior of the 'a' actor. Listing 5.11 shows the output of the other virtual machine for the 'b' actor. These show the displays of the loco function, and illustrate the success of the unification process and use of the 'test'-service discovered though environment queries.

Listing 10

```
AmbientTalk initialized.
:<actor>
msg sent
```

Listing 5.10: Requester transcript basic example

First the provide and require functions are called after creation of both actors. Actor 'a' will receive a message from the Loco process, signaling that the service is added (asserted: <pattern>). On the other side (b) a pattern containing the query is broadcasted to all actors. Then 'a' will receive this message and find a match, the success of unification is signaled through a loco message (match <pattern>) and join message is send back to 'b'. On arrival 'b' will call all the services, that were discovered and for each display a message: 'msg sent'. And finally 'a' will display 'The test worked!' as a result of the call.

Listing 11

```
AmbientTalk initialized.
:<actor>
Loco: asserted: <pattern>
Loco: match: <pattern>
The test worked!
```

Listing 5.11: Provider transcript basic example

5.2.2 Logic operator example

The next example will show the use of logical operators, as means to describe more accurately what service we are looking for (shown in figure 5.6).

```
a: actor({
    init()::provide("printer(dpi(900))");

    startNetwork();

    test()::display("The test worked!", eoln);

    loco(result)::display("Loco:", result, eoln)
})
```

(a) Provider

```
b: actor({
    init()::require("printer(dpi(X)) & gt(X,800)");

    startNetwork();

    join()::{ joined: messages("joined");
        for(i:1,i<=size(joined),i:=i+1, {
            provider(joined[i])#test();
            delete("joined", joined[i]);
            display("msg sent", eoln)
        })
    };

    disjoin()::display("Disjoin detected: ", disjoined);

    loco(result)::display("Loco:", result, eoln)
})
```

(b) Requester

Figure 5.6: Logic operator example

This example is very similar to the basic example, but now we have another query passed to the require function in (b). It contains one free variable X, in both terms of the logical 'and relation' (&). The query is true if both ends are true.

During unification we will find all values for X that unify with the left term, and if the same values also unify with the right term we have found a match. As 'gt' stands for 'greater then' we will find all printers with a dpi of more then 800. Some logical languages (not yet Loco) support syntactic sugar for this common statements and our query would like "printer(dpi(X>800))'. This all gives us a short and more detailed query.
The use of logical operators is not restricted to the requiring side, we could just as well employ it at the provider side with the same effect.

5.2.3 Rule example

In our third example we will illustrate the use of rules as an addition to the logical queries and facts. The example in figure 5.7 shows us how we can add rules to a query.

```
a: actor({
    init()::provide("fax(bw)");

    startNetwork();

    test()::display("The test worked!", eoln);

    loco(result)::display("Loco:", result, eoln)
})
```

(a) Provider

```
b: actor({
    init()::require("printer(dpi(X),bw):fax(bw) & equals(X,300)",
                    "printer(dpi(Z),bw) & gt(Z,200)");

    startNetwork();

    join()::{ joined: messages("joined");
        for(i:1,i<=size(joined),i:=i+1, {
            provider(joined[i])#test();
            delete("joined", joined[i]);
            display("msg sent", eoln)
        })
    };

    disjoin()::display("Disjoin detected: ", disjoined);

    loco(result)::display("Loco:", result, eoln)
})
```

(b) Requester

Figure 5.7: Rule example

The difference with the previous example is that now two arguments are passed to the require function: a rule and almost the same query as before. Hence, we are not limited to the use of all the logical rules that are packed with Loco, we can create are own custom ones.

The unification process as described in section 4.1.3 does not only consider facts but also rules. All values unifying with X in the head, will be checked with the body of the rule. So if we have a black and white fax, and we are looking for a black and white printer with only 300 dpi, the fax will be offered as printer service. Again this can be simplified with syntactic sugar: 'printer(dpi(X), bw): fax(bw) & X=300'.

As shown in this example we can use rules to encode arbitrary structures of hierarchical name values, even on the fly, as an easy way to introduce sybtyping in our descriptions.

This rule could just as well be added at the provider side with the same effect. It would remove the necessity of typing the rule every time we require the same service.

5.2.4 Recursion example

Our last example will explain how we we can use recursion as tool to write even more expressive environment queries like in figure 5.8.

```
a: actor({
    init()::provide("unlocked(cafetaria,park)
                     unlocked(park,hallway);
                     unlocked(hallway,computerlab)",
                    "printer(printer1,computerlab");

    startNetwork();

    test()::display("The test worked! , eoln);

    loco(result)::display("Loco:", result, eoln)
})
```

(a) Provider

```
b: actor({
    init()::require("reachable(X,Z): unlocked(X,Z);
                     reachable(X,Z): unlocked(X,Y)
                                     & reachable(Y,Z)",
                    "printer(X,Y) & reachable(cafetaria,Y)");

    startNetwork();

    join()::{ joined: messages("joined");
        for(i:1,i<=size(joined),i:=i+1, {
            provider(joined[i])#test();
            delete("joined", joined[i]);
            display("msg sent", eoln)
        })
    };

    disjoin()::display("Disjoin detected: ", disjoined);

    loco(result)::display("Loco:", result, eoln)
})
```

(b) Requester

Figure 5.8: Recursion Example

Until now we have encountered two types of logical formulas: facts and rules. There is a special kind of rule which deserves special attention: the rule which defines a relation in terms of itself.
In (b) we define a relation of reachability in our printer example , where a location is reachable from another location if they are connected by one or more unlocked doors. The first rule speaks for itself: a location Z is reachable from a location X if the connection between X and Z is unlocked. The reading of the second rule is as follows: Z is reachable from X if Y is connected by an unlocked door with X, and Z is reachable from Y.
We can now prove, based on the facts from (a), that the computerlab, where printer1 resides, is connected to the cafetaria without locked doors. So we have a match and the service will be send back through a join message.

Of course this can be translated into rules without recursion but then we would need a lot more, and much longer rules. Recursion is a much more convenient and natural way to define such chains of arbitrary length.

5.3 Technical Issues

As we now grasp the concept of environment queries, we will take a closer look at some technical issues that emerge in the implementation of the service discovery protocol.

5.3.1 Multiple matches

An actor can provide multiple services that match a certain query and in theory this can lead to an infinite loop. On the level of Loco this is handled by not returning all results at once, but one at a time. To the AmbientTalk process we send a join message back for every match. After a certain threshold (currently 100) of matches we stop sending join messages to prevent flooding of the network and message buffers.

5.3.2 Loco processes

Every actor has its own Loco process to achieve the most parallelism and to avoid having actors waiting for each other. A single process for every device would bring forward a single point of failure if we corrupt the knowledge base, and we would need to implement namespaces (e.g. using prefixes) to separate the services.

5.3.3 Rules

Rules are always asserted on the side of the provider as it is there where the matching will take place. If rules would have been asserted on the requester side, we would be forced to transfer the rule base of the provider. The order in which the rules are traversed during the unification process is very significant, as it is in all other logic programming languages. This becomes even more important because both provider and requester can add rules. Rules asserted through the provide-statement, precede those declared through the request-statement.

We have to be careful with rules on the provider-side because they can give rise to unexpected conflicts in interpretation. A provider could cause a lot of false positives or false negatives by deliberate or accidental erroneous rules that a requester can not expect as shown in figure 5.9.

```
a: actor({
    init()::provide("printer(dpi(X),bw):fax(bw) & equals(X,300)",
                    "fax(bw)");

    startNetwork();

    test()::display("The test worked!", eoln);

    loco(result)::display("Loco:", result, eoln)
})
```

(a) Provider

```
b: actor({
    init()::require("printer(dpi(X),bw):fax(bw) & equals(X,200)",
                    "printer(dpi(300),bw)");

    startNetwork();

    join()::{ joined: messages("joined");
        for(i:1,i<=size(joined),i:=i+1, {
            provider(joined[i])#test();
            delete("joined", joined[i]);
            display("msg sent", eoln)
        })
    };

    disjoin()::display("Disjoin detected: ", disjoined);

    loco(result)::display("Loco:", result, eoln)
})
```

(b) Requester

Figure 5.9: Rule causing unexpected matches

5.3.4 Symbiosis

To allow AmbientTalk to make use of Loco to express service descriptions and queries, the Symbioco layer was extended (figure 5.10) to support AmbientTalk as well by adding its interpreter (conveniently written in Java too) to the framework and implementing a callback mechanism to return the results. The error-capturing (described in 4.2.2) functionality isn't yet fully supported in the implementation of our environment queries, because the focus of this dissertation is on the service discovery and not on symbiosis, although it would be a nice extension for future work.

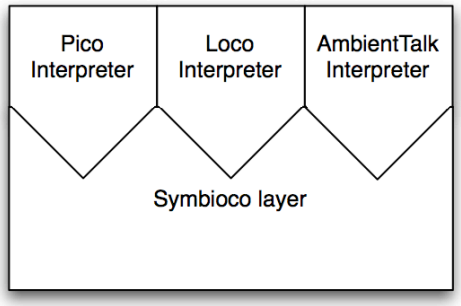


Figure 5.10: Graphical representation of the Symbioco layer

The hooks signaling when to switch between AmbientTalk and Loco are situated in provide and require native functions.

5.3.4.1 Providing a pattern

When an actor provides a service, the pattern (rules + facts) describing its properties, is added to the required-mailbox but the pattern is also passed to the Symbioco layer along with the running AmbientTalk process. Symbioco uses the original Process classes from both interpreters, and manipulates the stack through them. In order to manage these processes, a wrapper is built around them. Symbioco uses its own stack to manage these process wrappers. Symbioco will create a new process wrapper for the AmbientTalk process, and will push it on its process stack. Next a Loco process will be created, also pushed on the process stack and then start this process. It will interpret the pattern and signal the Symbioco layer when the evaluation is done.
When the process finishes execution, Symbioco will give control back to the wrapped AmbientTalk process to send back the results, based on the return value. Because originally the call came from provide-statement, the only action left to do is sending a Loco message back to the actor signaling the successful assertion of the pattern (asserted: <pattern>).

5.3.4.2 Requiring a pattern

The other point where we enter the Symbioco layer, is when an actor requires a service. The pattern is added to ”required”-mailbox and the device

broadcasts its addition to all neighboring devices as described in section 2.4.1.1. All the receiving devices will loop over their local actors and initiate a query on the actor's corresponding rule base. The query will again be passed to a Loco process, through the Symbioco layer.
On completion of evaluation in Loco, the Symbioco layer will capture the results. It will send the results back to the originating actors under the form of join messages (containing pattern and provider). A Loco message is also send to the actors with a matching service pattern: 'match: <pattern>' .

We minimized syntactic changes to AmbientTalk and chose a syntax that looks very natural to programmers as it is identical to the manner SQL is integrated in all the common programming languages. Another option would be: making the symbiosis even more transparent by removing the quotes from the arguments for provide and require. This would also be a nice extension for future work.

5.4 Conclusion

Our proof-of-concept implementation lead to three key stages that we set as our goals in section 3.4.4, and that we will evaluate now:

5.4.1 Incorporating advantages of existing alternatives

After researching current service discovery protocols we identified the strong points of the other description languages. One of those assets were logical operators that enhance the expressiveness and shorten the descriptions. Another was the ability to construct hierarchies and introducing subtyping in our service advertisements and queries. Both of them are now supported in AmbientTalk through symbiosis with Loco. Logical operators are part of Loco's syntax and hierarchies can be built with rules (see previous examples).

5.4.2 Minding restrictions of open mobile networks

One has to pay heed to the hardware restrictions and phenomena discussed in section 2.2 and in section 3.1.3, when designing a service discovery protocol for finding services provided by ambient resources. As autonomy is required, service cache managers are out of the question and not part of

environment queries. The volatility of connections demands for consistency maintenance, failure detection and recovery. Therefore the protocol has the same notification mechanism and heartbeat algorithm as in the original implementation of AmbientTalk. For concurrency reasons we have equipped every actor with his own Loco process to manage services and queries on them.
XML was the most expressive description language in section 3.1.3 but its verboseness can pose a crucial problem over mobile networks where bandwidth can be limited or vary greatly. Even processing XML, due to the very limited processing power and restricted memory of mobile devices, itself can pose a problem. Environment queries on the other hand are very compact without losing expressiveness, resulting in low overhead on the network. However, there is no such thing as free lunch and more computation is needed due to recursion for unification and because of the language symbiosis.

5.4.3 Introducing more expressiveness

The proliferation of devices discussed in chapter three, will give rise to an exponential growth of services in open mobile networks. As a result we need precise service descriptions, otherwise there would be too much matches. Users usually have specific demands as in the printer example in section 2.1 for the services they require. Environment queries provide the tool to meet those demands: logic operators, rules and recursion.

Hence, one can argue that we reduced the complexity of finding a fitting service, which will become paramount on a larger scale. In our conclusions in the next chapter we will point to future work to further enrich service descriptions.

5.5 Summary

In this chapter we have extended the original service discovery protocol of AmbientTalk and showcased a new description language: environment queries. It differs from the string-matching algorithm by expressing service descriptions and queries in logic. Each actor's knowledge base, of services, is transparently updated and queried by it's own Loco process. The protocol

supports logical operators, rules and recursion.

Though the implemented examples are rather basic, the power and practical use of environment queries was clearly shown.

6

Conclusion and Future Work

The objective of this dissertation was to investigate the benefits of using a logic programming language as a service description language for service discovery in open mobile systems. This chapter presents our conclusions on this matter.

6.1 Summary

In our present-day network environments we face two main problems: the first is the expansion of computing environments in homes and offices through the ever growing numbers of printers, scanners, digital cameras, and other peripherals, integrated into networked environments. The second problem is the proliferation of mobile devices such as laptop and palm-sized computers, cellular phones and other portable gizmos. These devices all trade functionality for suitable form factors and low power consumption; they are therefore "peripheral-poor" and as a result they must connect to proximate machines for storage, faxing, printing, scanning and internet access. [Ric00] Due to these changes mobility and modularity have become the modern goals of system development to enable Ambient Intelligence.

Network resources and application software do not follow the mobile users when they leave their offices or homes, or when they relocate to another temporary office or home. Supporting true mobility of users will therefore require changing the way application software is advertised and discovered.

This dissertation consists of two parts. In the first part we presented the necessary background concerning ambient intelligence, service discovery, logic programming and language symbiosis.
In the second chapter we described the context of this dissertation. We addressed the hardware phenomena that become apparent in open mobile networks and how languages of the ambient-oriented programming paradigm handle them. At the end of the chapter a prototype-based language, following the paradigm, was discussed in more detail.
Chapter three explained what service discovery is, and the concepts that compose a service discovery protocol were defined. Different strategies and design options were discussed. A couple of scenarios illustrated the use of discovery, and they were followed by an in depth analysis of nine of the most important service discovery protocols including AmbientTalk. A comparison was made between them and we identified related research in the same area. A summary and outlook to the future concluded this chapter.
Chapter four revolved around logic programming and language symbiosis. We described the declarative programming paradigm and took a closer look at Loco, the logic programming language that was used in the symbiosis. We briefly looked at language symbiosis and at a concrete implementation: Symbioco.
The second part of this thesis starts with chapter five, where we finally introduced a new language construct called "environment queries" and we put forward a proof-of-concept implementation supporting the thesis. Thereafter a more technical discussion was presented concerning a number of practical issues and the chapter ends with several validating examples.

6.2 Contributions

Chapter three of this dissertation serves as a survey of the state of the art in service discovery protocols, and lead us to reveal shortcomings that we believe will inhibit the development of applications that exploit open mobile

networks. We particularly focused on the absence of rich service descriptions.
The main technical contribution of this thesis is the design and implementation of a new service discovery protocol for open mobile networks, based on AmbientTalk's previous ambient-oriented service discovery protocol. The new protocol has a rich description language with the following properties:

- The syntax of AmbientTalk is extended with three new functions: require, provide and loco. The first two are used for querying neighboring actors for a service and providing a service, respectively. The Loco function is merely a signaling function for mainly debugging purposes. Its default behavior is to write events from the Loco process to the transcript, but this can be overridden by the programmer.
- Service descriptions now support the use of logical operators like: and, or, if, not, equal, greater than, less than.
- The addition of custom rules permits encoding arbitrary structures of hierarchical name values, even on the fly, as an easy way to support subtyping in service descriptions.
- The possibility to utilize recursion to reduce the size of service descriptions in a natural way, through recursive rules.

We have evaluated these properties by using the environment queries construct in various examples which showed that environment queries provide a concise way of describing services where processing power, memory and bandwidth, are limited or vary profoundly and where expressiveness is a critical factor to cope with the (future) proliferation of services.

6.3 Limitations and Future Work

6.3.1 Scale of network

In order to evaluate our new service discovery protocol in an ambient oriented context we used the environment queries in a number of basic examples. However, the full potential and possible problems following adaptation of the protocol will, most likely, only be uncovered when it is used on a larger scale, since one of the primary goals of our description language is to reduce

the complexity of finding a fitting service in large networks. Therefore, future work will have to focus on the development of larger ambient oriented applications using the protocol to fully understand its applicability in this context.

6.3.2 Personalization through the use of Meta-data, Reputation and History

If we go back to the printer example in section 2.1, we see that we solved the main part of the problem: describing the properties of the printer. Now the second part consists of taking the user's (or even other users') service history into account. The previous preferences of a user are valuable for his/her future requests and could help categorize or order services. Useful data is already present in the "require" , "join" and "disjoin"-mailboxes, but more language support would be a nice extension.

Another approach for more personalization could manifest itself through social environment queries, by enriching the service descriptions with additional information, i.e. metadata. For example, users would be able to raise or lower the reputation score of a service, and even the reputation of the user could be considered in the calculation. We could also be able to see all the services a trusted friend has vowed for, or personal comments of other users on the service to help us make decisions on what service to use.
Social data is becoming more useful and popular on the web as can be seen by the use of the much hyped phrase Web 2.0 [1], and this approach may become valuable in organizing the large amount of services in the future.

6.3.3 Inexact Matching and Load Balancing

Our service discovery protocol does only exact semantic matching while finding out a service. Thus it lacks the power to give a 'close match' even if it was available. The solution for this could lie with fuzzy logic. Fuzzy logic is derived from fuzzy set theory dealing with reasoning that is approximate rather than precisely deduced from classical predicate logic. It can

[1] The phrase Web 2.0, which was created by O'Reilly Media to refer to a supposed second generation of services available on the World Wide Web that lets people collaborate and share information online in a new way - such as social networking sites, wikis and folksonomies. From Wikipedia, the free encyclopedia

be thought of as the application side of fuzzy set theory dealing with well thought out real world expert values for a complex problem.

A user may not be able to specify the exact values of interested characteristics of a service. For example, if a service has an attribute that specifies its 3-dimension GPS[2] location, it is difficult for a client to pose a query with the exact value of the attribute. A client usually wants to find a service that is close to it. The notation of physical proximity is best captured by comparing the distance between two locations, instead of syntactically comparing two location expressions, or comparing the values independently at each dimension. [Yan01]

The service discovery protocol doesn't use any performance parameters for the existing services. They are satisfied with finding out a service only. It doesn't consider whether the service would be able to serve the requester. Future work is therefore needed on load-balancing in order to more evenly distribute the data processing across available service providers. Service advertisements could be retracted based on the load.

[2] Global Positioning System: A worldwide radio-navigation system that was developed by the US. Department of Defense. In addition to military purposes it is widely used in marine, terrestrial navigation and location based services.

Bibliography

[AC94] G. Agha and C. J. Callsen. Open heterogeneous computing in actorspace. *Journal of Parallel and Distributed Computing*, 21:289–300, 1994.

[Agh86] Gul Agha. *Actors: a model of concurrent computation in distributed systems.* MIT Press, Cambridge, MA, USA, 1986.

[Agh90] G. Agha. Concurrent object-oriented programming. *Concurrent object-oriented programming*, 33(9):125–140, 1990.

[Amb06] AmbientTalk. Ambient-oriented programming and ambienttalk http://prog.vub.ac.be/amop/, 2006.

[AT05] Rebecca Montanari Alessandra Toninelli, Antonio Corradi. Semantic discovery for context-aware service provisioning in mobile environments. Dipartimento di Elettronica, Informatica e Sistemistica Universitat di Bologna, 2005.

[BY87] Jean-Pierre Briot and Akinori Yonezawa. Inheritance and Synchronization in Concurrent OOP. In J. Bézivin, J-M. Hullot, P. Cointe, and H. Lieberman, editors, *Proceedings of the ECOOP '87 European Conference on Object-oriented Programming*, pages 32–40, Paris, France, 1987. Springer Verlag.

[CD] Michalis Vazirgiannis Christos Doulkeridis, Vassilis Zafeiris. The role of caching and context-awareness in p2p service discovery.

[CL02] Sumi Helal Choonwah Lee. Protocols for service discovery in dynamic and mobile networks. *The International Journal of Computer Research, Special issue on Wireless Systems and Mobile Computing*, 11(1), September 2002.

[CNP01] G. Cugola, E. Di Nitre, and G. Picco. Content-based dispatching in a mobile environment, 2001.

[Con] Salutation Consortium. Salutation architecture for service discovery. http://www.salutation.org/.

[CPAJ01] D. Chakraborty, F. Perich, S. Avancha, and A. Joshi. Dreggie: Semantic service discovery for m-commerce applications, 2001.

[DBS+03] K. Ducatel, M. Bogdanowicz, F. Scapolo, J. Leijten, and J. C. Burgelman. Istag scenarios for ambient intelligence in 2010. Technical report, ISTAG, 2003.

[DCM+05] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Theo D'Hondt, and Wolfgang De Meuter. Ambient-oriented programming. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 31–40, New York, NY, USA, 2005. ACM Press.

[Ded04] Van Belle Dedecker. Formal foundations of the ambient actor model, 2004.

[DM01] C. Dabrowski and K. Mills. Analyzing properties and behavior of service discovery protocols using an architecture-based approach, 2001.

[DM02] C. Dabrowski and K. Mills. Understanding self-healing in service-discovery systems. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 15–20, New York, NY, USA, 2002. ACM Press.

[DMWJ03] D'Hondt T. De Meuter W. and Dedecker J. Intersecting classes and prototypes. In *In Ershov Memorial Conference (2003), M. Broy and A. V. Zamulin, Eds., vol. 2890 of Lecture Notes in Computer Science, Springer. Perspectives of Systems Informatics, 5th International Andrei Ershov Memorial Conference, PSI 2003, Akademgorodok, Novosibirsk, Russia*, July 9-12 2003.

[DMWJ04] D'Hondt T. De Meuter W. and Dedecker J. Pico: Scheme for mere mortals. In 1st European Lisp and Scheme Workshop, Oslo, Norway, 2004.

[Dyr03] Michael Dyrna. Peer2peer network service discovery for ad hoc networks. Hauptseminar im Wintersemester 2003 / 2004, December 2003.

[FDW+04] Adrian Friday, Nigel Davies, Nat Wallbank, Elaine Catterall, and Stephen Pink. Supporting service discovery, querying and interaction in ubiquitous computing environments. *Wirel. Netw.*, 10(6):631–641, 2004.

[FK04] Christian Frank and Holger Karl. Consistency challenges of service discovery in mobile ad hoc networks. In *Proceedings of the 7th ACM International Symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM)*, pages 105–114, Venice, Italy, October 2004.

[Gee05] Frederik Geerts. Overview of service discovery protocols. Presentation, 2005.

[Gel85] David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.

[Gup05] Gautam Umesh Gupta. Service discovery in ad-hoc networks. Master's thesis, Rochester Institute of Technology, 2005. Proposal of a thesis.

[Gyb03] Kris Gybels. Soul and smalltalk - just married: Evolution of the interaction between a logic and an object-oriented language towards symbiosis. In *Proceedings of the Workshop on Declarative Programming in the Context of Ob ject-Oriented Languages*, 2003.

[Hel02] Sumi Helal. Standards for service discovery and delivery. *IEEE Pervasive Computing*, 01(3):95–100, 2002.

[Hew77] C. E. Hewitt. Viewing control structures as pattern of passing messages. *Atrificial Intelligence: an International Journal*, 8, 3:323–364, June 1977.

[Inca] Sun Microsystems Inc. Jini network technology: an open architecture that enables developers to create network-centric services. http://www.sun.com/software/jini/.

[Incb] Sun Microsystems Inc. Jxta technology enables developers to create innovative distributed services and applications. http://www.sun.com/software/jxta/.

[JP04] Micheal Luck Luc Moreau Terry Payne Juri Papay, Simon Miles. Principles of personalisation of service discovery, http://www.citebase.org/abstract?id=oai:eprints.ecs.soton.ac.uk:10221, 2004. School of Electronics and Computer Science University of Southampton.

[KB02] Alan Kaminsky and Hans-Peter Bischof. Many-to-many invocation: a new object oriented paradigm for ad hoc collaborative systems. In *OOPSLA '02: Companion of the 17th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 72–73, New York, NY, USA, 2002. ACM Press.

[KKRO03] M. Klein, B. Konig-Ries, and P. Obreiter. Lanes – a lightweight overlay for service discovery in mobile ad hoc networks, 2003.

[KoR02] M. Klein and K. onig Ries. Multi-layer clusters in ad-hoc networks - an approach to service discovery, 2002.

[Leu05] Tom Leuse. A symbiotic approach to aspect-oriented logic meta programming in a prototype-based language. Master's thesis, Vrije Universiteit Brussel, Faculty of Sciences , Department of Computer Science and ,Applied Computer Science, 2005.

[LH02] Choonwa Lee and Sumi Helal. Protocols for service discovery in dynamic and mobile networks. *International Journal of Computing Research*, 22 number 1:1–12, 2002.

[Liv03] Steven R. Livingstone. Service discovery in pervasive systems. Master's thesis, The School of Information Technology and Electrical Engineering The University of Queensland, 2003.

[LPG] LLC. Lexico Publishing Group. Dicitionary http://www.dictionary.com.

[MCE02] C. Mascolo, L. Capra, and W. Emmerich. Mobile computing middleware, 2002.

[MPHS05] R. Marin-Perianu, P. H. Hartel, and J. Scholten. A classification of service discovery protocols. http://eprints.eemcs.utwente.nl/735/, June 2005.

[MPR01] A. Murphy, G. Picco, and G.-C. Roman. Lime: A middleware for physical and logical mobility. In *In Proceedings of the 21 st International Conference on Distributed Computing Systems*, pages 524–536, May 2001.

[MSW05] Van Cutsem Tom Mostinckx Stijn, Dedecker Jessie and De Meuter Wolfgang. Conversations for ambient intelligence. *Workshop on Exception Handling for Object-Oriented Systems, ECOOP 2005*, 2005.

[OOP05] OOPSLA. Ambient-oriented programming in ambienttalk, http://oopsla.acm.org/2005/showevent.do?id=643, 2005.

[PF94] The Netherlands Peter Flach, Tilburg University. *Simply Logical: Intelligent Reasoning by example*. Wiley Professional computing, 1994.

[PRDIRC05] Gian Luca Foresti Paolo Remagnino Digital Imaging Research Centre, Kingston University, editor. *Ambient Intelligence: A Novel Paradigm*. Springer, 2005.

[Pro] Prog. Programming technology lab, http://prog.vub.ac.be.

[Ric] Ryan Wishart Ricky. Superstringrep: Reputation-enhanced service discovery.

[Ric00] Golden G. Richard. Service advertisement and discovery: Enabling universal device cooperation. *IEEE Internet Computing*, 4(5):18–26, 2000.

[SDP] SDP. The offical bluetooth web site: http://www.bluetooth.com/bluetooth/learn/works/.

[SGF02] Rüdiger Schollmeier, Ingo Gruber, and Michael Finkenzeller. Routing in mobile ad hoc and peer-to-peer networks. a comparison. Technical report, Int.Workshop on Peer-to-Peer Computing Technische Universiteit Mnchen, 2002.

[SHdH+05] V. Sundramoorthy, P. H. Hartel, J. I. den Hartog, J. Scholten, and C. Tan. Functional principles of registry-based service discovery. In *LCN '05: Proceedings of the The IEEE Conference on Local Computer Networks 30th Anniversary*, pages 209–217, Washington, DC, USA, 2005. IEEE Computer Society.

[SLP] SLP. Service location protocol project. http://srvloc.sourceforge.net/.

[TES05] Neilze Dorta Tarek Essafi and Dominique Seret. A scalable peer-to-peer approach to service discovery using ontology, www.math-info.univ-paris5.fr/ seret/artgd.pdf, May 15 2005.

[UDD] UDDI. Uddi technical white paper. http://uddi.org/pubs/uddi-tech-wp.pdf.

[UPn] UPnP. The upnp forum is an industry initiative designed to enable simple and robust connectivity among stand-alone devices and pcs from many different vendors. http://www.upnp.org/.

[VA98] Carlos A. Varela and Gul A. Agha. What after java? from objects to actors. In *WWW7: Proceedings of the seventh international conference on World Wide Web 7*, pages 573–577, Amsterdam, The Netherlands, The Netherlands, 1998. Elsevier Science Publishers B. V.

[Vin03] Steve Vinoski. A steve vinoski toward integration column from ieee internet computing. Technical report, IEEE, February 2003.

[Yan01] Xiaowei Yang. A framework for semantic service discovery, 2001. MIT Laboratory of Computer Science, 200 Technology Square, Cambridge, MA 02142,, USA.

[Yu06] Xinqi Wang Xueli Yu. A owl-based semantic web service discovery framework. *AICT-ICIW '06. International Conference on Internet and Web Applications and Services/Advanced International Conference on Telecommunications*, page 125, 2006.