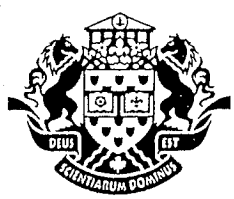


Université d'Ottawa · University of Ottawa

Université d'Ottawa · University of Ottawa

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES

FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

Yong LU

AUTEUR DE LA THÈSE - AUTHOR OF THESIS

M. Sc. (Computer Science)

GRADE - DEGREE

School of Information, Technology and Engineering

FACULTÉ, ÉCOLE, DÉPARTEMENT - FACULTY, SCHOOL, DEPARTMENT

TITRE DE LA THÈSE - TITLE OF THE THESIS

Agent-based Service Discovery in Ad Hoc Communications Environments

A. Karmouch

DIRECTEUR DE LA THÈSE - THESIS SUPERVISOR

CO-DIRECTEUR DE LA THÈSE - THESIS CO-SUPERVISOR

EXAMINATEURS DE LA THÈSE - THESIS EXAMINERS

A. Nayak

J. Oommen

J-M. De Koninck, Ph D

LE DOYEN DE LA FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES

DEAN OF THE FACULTY OF GRADUATE AND POSTODORAL STUDIES

Agent-based Service Discovery in Ad-hoc Communications Environments

By

Ying Lu, B.E.

A thesis submitted to the
Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Computer Science

Ottawa-Carleton Institute for Computer Science
Department of Computer Science
School of Information Technology and Engineering
Faculty of Engineering

University of Ottawa
Ottawa, ON, Canada

June 2004

Your file Votre référence
ISBN: 0-494-01538-1
Our file Notre référence
ISBN: 0-494-01538-1

Abstract

Service Discovery allows for the automatic and spontaneous discovery and configuration of network services. It is especially important in ad-hoc environments where the network topology is unpredictable. We propose the use of Agent Technology, an efficient and convenient technology, in our work to shed new light in this area.

Our first model takes on a centralized architecture. One agent in the network, the Service Discovery Agent (SDA), assumes the responsibility of maintaining the central service repository. Each service and user has their own agents which we name Service Agents (SAs) and Personal Agents (PAs) respectively. SAs are responsible for registering and deregistering their services with the SDA. As service information is received, the SDA determines which users are authorized to access the service and forwards the information to the corresponding PAs. Therefore, users always have a complete list of available services they are authorized to use.

In inter-domain discovery, each domain maintains its own discovery scheme as described above. Interactions between different domains are carried out by the SDAs. The SDAs inform each other of the services offered in their respective domains. Remote service access can be done by cloning and moving SAs from home to destination.

Our second discovery model uses a distributed architecture that can operate in a fully dynamic environment. For this we use an existing probabilistic P2P search algorithm as a basis. The algorithm's merit lies within its efficiency to perform resource searches. Only a small portion of network devices needs to be contacted to find a resource. In the distributed model, each device has its own SDA. An underlying presence awareness mechanism is created so that devices are constantly aware of the presence and absence of other devices. Discovery is performed on demand using the concepts of the P2P search algorithm. The algorithm is enhanced in many ways so that it is adapted to an ad-hoc environment while still keeping the basic efficient operations. As more searches are performed, devices become more knowledgeable and discovery performance would improve. While users are required to initiate discovery, their responsibilities are still made as little as possible due to the efficient and continuous learning nature of the adapted P2P algorithm.

Acknowledgements

I would like to thank my supervisor Dr. Ahmed Karmouch for his constant support and guidance throughout my research. I would like to thank my colleagues at the Multimedia and Mobile Agent Research Laboratory in the University of Ottawa for their cooperation and encouragement. I wish to thank my parents and my friends for providing moral and motivational support during my period of study. And I extend my thanks to NSERC and MITEL for providing important suggestions and monetary support that helped in completing my Masters' program.

Contents

List of Figures

Chapter 1

Introduction

With today's rapidly advancing technology comes the demand for a much wider variety of applications. Users residing on networked environments are seeing an increased number of network services as well as more applications incorporated for network sharing and usage. This increase in the demand and number of services in networks translates into more weight and complexity being put onto network service management. Service Discovery techniques are introduced in an effort to reduce the work and automate the process of managing network services. It has become an ever important part of network operations.

1.1 Motivation

The popularity and the need for ad-hoc communication are on the rise in recent years. More and more users are settling into a busier work style that requires frequent mobile roaming. The highly dynamic nature of ad-hoc environments creates a greater need for Service Discovery as well as an added level of complexity for the discovery mechanism to handle. The use of ad-hoc environments often means users spending limited time within the network. Thus time limit is an important issue and discovery should be performed as quickly as possible. These new challenges brought forth by the use of ad-hoc communications serve as the motivations behind our research into Service Discovery. Another driving force is Agent Technology. Agents, as autonomous and asynchronous entities, have the potential to provide many advantages and conveniences for an effective Service Discovery model in an ad-hoc communication environment. Therefore, the increasing need for ad-hoc communications and the benefits offered by Agent Technology drive us to create new discovery models for today's rapidly advancing networks.

1.2 Service Definition

In the context of Service Discovery, Chakraborty and Chen [1] defined services as "facilities that are available to a computer". Each distinct task that can be done for the user is deemed a service. Services may be confused with devices. Devices are the physical components that build a network. The devices are hardware entities such as

computers, printers and telephones. Certain devices, as McGrath [2] indicated, can offer a number of services. For instance, a printer can have both print and fax capabilities.

In a network, other than the standard network services such as printing, there can also be sharing of regular applications that one would not normally associate with a network, such as media players. A user, for example, may not have a MP3 player on her device, but can request to use one that resides on another device. In summary, services include any network service or application available for use and sharing by network users.

1.3 Service Discovery and Its Benefits

With the absence of a Service Discovery mechanism, all responsibility involved in network service management fall on the shoulders of the administrators. Tasks such as connecting new services and setting access authorizations for different users are done manually. Users must have some prior knowledge of the services offered and perform the necessary steps of configuration themselves to access the services. Service Discovery offers an all-in-one package where it takes on the roles of both the administrator and the user to relieve them of most service-related tasks.

We can therefore perceive Service Discovery as existing on two levels. First, on the lower level, some awareness of all the network services and their status must be present. This also means awareness of changes as new services can be added and existing ones removed. Then on the higher level, with the knowledge of all the network services, access and configuration of services should automatically be done for the users.

As Bettstetter and Renner [3] indicated, Service Discovery lightens the heavy load for the network administrator. With a good discovery mechanism, there should be little or no need for the administrator to update and maintain network services manually. The same applies to users of services. The users would not spend much effort looking for services with the use of an effective Service Discovery protocol. Bettstetter and Renner [3] illustrated that in an ideal scenario, a user can simply walk into a network with a laptop computer, and all authorized services in the network will be connected to the laptop automatically and are ready to be used. Hence we can look at Service Discovery as an automatic and, as McGrath [2] added, spontaneous method for discovery and management of network services at both the administrative and the user levels.

1.4 Ad-hoc Communications and Service Discovery

An ad-hoc network, as described by Tschudin [4], is a *"wireless network without infrastructure"*. Such a network is only a temporary one. It has no fixed topology and is characterized by the dynamic nature of the nodes that form the network. Network devices simply connect and disconnect spontaneously. This ad-hoc nature allows for rapid network construction and deconstruction, thus offering convenience for users that participate in mobile roaming situations.

Because of the spontaneous and dynamic characteristics of ad-hoc networks, no human administrative control can be performed. Instead, network management is done automatically in a distributed manner and it is up to the devices themselves to organize and configure various communications and cooperative operations. Motegi *et al* [5] noted that due to the difficulties of administering ad-hoc networks, services and their

locations cannot be easily determined. Therefore one can imagine that Service Discovery is even more important in such a network. In fact, as a Service Discovery mechanism has already been described as dynamic and spontaneous [2], it is perfectly suited to be adapted for ad hoc environments.

One of the main challenges with performing discovery in ad-hoc environments is to be aware of and maintain dynamic service information at all times since devices can join and leave without warning or notification. Consequently, how this dynamic service information should be translated to the affected users in a timely fashion is another major concern. In the process of looking for and obtaining a service for the user, it is also very important to look at the dynamic picture of the devices instead of a freeze frame of the network. Changes in the devices during this time may have an effect on the outcome of the search.

1.5 Thesis Contributions

The results of the Thesis are -

(i) *Conception and implementation of a centralized agent-based Service Discovery model*

The Thesis proposes a software agent-based Service Discovery model that builds on the common centralized design of Service Discovery. In this design, all authorized available services in the network are made ready and maintained for a user as she enters and stays in the network. The users need not be aware of services and their status.

(ii) *Inter-domain Service Discovery*

It is often beneficial to share certain services between several different administered network domains. The proposed architecture designates the task of communication with other domains to a specific agent within each domain. Agent cloning and traveling are also used to allow for accessing services from other domains.

(iii) *Conception and implementation of a distributed agent-based Service Discovery model*

The distributed Service Discovery model proposed takes one step further in that it is more suited for fully ad-hoc environments. The model makes use of a P2P search algorithm and modifies it into an agent-based design that can adapt to the dynamic nature of ad-hoc environments. The end result is a discovery mechanism that performs service search in an efficient manner and concurrently builds its knowledge base of available services to become more search efficient.

1.6 Frequently Used Keywords and Definitions

Service: Any application available for use and sharing by network users. A device may offer any number of services.

Service Discovery: A mechanism that is capable of the dynamic and spontaneous discovery of services and provides the means for accessing them.

Lookup Service: A passive, directory-oriented process to look for resources in networks.

Ad-hoc Network: A temporary spontaneous wireless network with no fixed infrastructure. It is automatically formed when nodes come together.

Agent: An autonomous software entity that makes decisions and performs services for other entities in an asynchronous and responsive manner.

Inter-domain: Between two or more different administered network domains.

Peer-to-peer (P2P): A communications model in which the communicating entities have equal abilities and conversations can originate from either entity.

1.7 Thesis Outline

This Thesis presents the details of the proposed Service Discovery architectures for ad-hoc environments. Chapter 2 introduces the necessary background information and related projects. Lookup services, as a predecessor to discovery, are discussed followed by a review of some of the existing discovery protocols. The Chapter also introduces the concepts behind Agent Technology and ad-hoc communications.

Chapter 3 outlines the central problem that the proposed Service Discovery models are solving. It discusses the general discovery principles that should be adhered to in the design. Chapters 4 and 5 form the core of the Thesis in that they describe the detailed design of the proposed models. Chapter 4 presents the centralized version of the Service Discovery model while Chapter 5 outlines the distributed model. Details of all the components in the architectures, their interactions, design issues and rationales are discussed in both Chapters. Chapter 6 then presents the prototype implementations of the two Service Discovery models. This will include execution screenshots and important code snippets.

The concluding Chapter outlines the contributions of this Thesis and proposes possible future improvements and research areas in the field of Service Discovery for ad-hoc environments.

Chapter 2

Background and Related Work

This Chapter clarifies the background information and technologies that relate to the contributions of this Thesis. We begin with an overview of lookup services that serve as a basis for Service Discovery. Some existing discovery protocols are then examined. These protocols demonstrate some of the common discovery architectures and techniques that will be closely looked at in the next Chapter. This Chapter will also introduce the fundamental concepts behind Agent Technology and how its features can benefit the computing world in areas such as ad-hoc communications. We will take another look at ad-hoc environments and briefly review some of the current projects that are aimed at developing Service Discovery models for ad-hoc environments.

2.1 Lookup Services

Lookup services provide the means for locating resources in networks. They are directory-based in that resources must register and deregister with the directory. Lookup is done passively. A search is performed only upon receiving a lookup request. McGrath [2] adds that *"lookup may be done in a statically configured environment, the directory need not be writable"*. In addition, static directories and databases are used which in turn must be maintained by human administrators. Lookup services are widely in use. They include some well-known protocols as well as many other registry and directory services. We briefly look at two such lookup protocols here.

2.1.1 Domain Name Service (DNS)

The Domain Name Service (DNS) is an Internet service in which domain names such as www.uottawa.ca are translated into IP addresses. DNS can be used to find addresses of network devices and services. As Shaikh *et al* [6] indicated, the presence of DNS is transparent to the user. Thus it offers users great convenience in that it eliminates the need for memorizing long IP address strings.

One disadvantage of DNS is that it does not support a wide range of services and search options are very limited. Since DNS is a trusted service, its static databases can only be maintained by privileged administrators. Furthermore, the availability of specific domain-IP matches are not guaranteed, and no announcements are made when domains register and deregister. These limitations make DNS a poor candidate to be adapted for spontaneous discovery.

2.1.2 Lightweight Directory Access Protocol (LDAP)

The Lightweight Directory Access Protocol (LDAP) is a standard developed by the Internet Engineering Task Force (IETF). As Severance [7] described, *"LDAP allows a program to perform directory lookups across a wide variety of directories"*. LDAP has a hierarchical structure and lookups can be performed at different hierarchies. Service advertisements can also be done through LDAP, although multicasting of announcements is not supported.

LDAP has the advantage of being extensible. It can store many different service types. In addition, it provides more comprehensive search capabilities than DNS. Although LDAP is still not a discovery protocol in that it does not delve into spontaneous discovery, but with its solid directory and query features, it is well suited to act as the directory server in Service Discovery models. Discovery features can be built onto existing features of LDAP.

2.1.3 Lookup as Basis for Service Discovery

To summarize, lookup services are directory services and usually require their databases to be manually maintained. The lookup process is passive and the actual availability of specific registered services is not guaranteed. The main function of a lookup service is to perform service queries when requested. In that respect, a lookup service with good directory and search capabilities such as LDAP can be used and built upon as part of a Service Discovery mechanism. We can see lookup services as a predecessor to Service Discovery. As technology keeps improving and user needs keep

increasing, lookup services are often not enough anymore. Automatic and spontaneous discovery is needed to offer more convenience to users.

2.2 Existing Service Discovery Protocols

As indicated earlier, Service Discovery should be present at both the network administrative level and the user level. Unlike the passive lookup services, it performs the automatic, dynamic, and spontaneous discovery and configuration of services. In addition, Service Discovery inherits the purpose of lookup services in that it should offer the capability to search for specific types of services when required and provide the means for accessing the services.

Since Service Discovery refers to the process of discovering services for users, the obvious two types of components in any discovery model are therefore service entities and user entities. The detailed technique of performing the discovery and whether other components are used differ from protocol to protocol. We now examine some popular and commonly used Service Discovery protocols.

2.2.1 Service Location Protocol (SLP)

One of the most popular Service Discovery protocols is the Service Location Protocol (SLP) [8]. SLP originates from Sun Microsystems and is an Internet Engineering Task Force (IETF) standard. In addition to having the service and user components in its architecture, it also employs a central element to perform discovery. Thus the three components that make up the SLP architecture are described as follows:

- *Service Agents (SAs)* represent specific services. An SA provides the interface for accessing the service it represents. It may also advertise the service.
- *User Agents (UAs)* act on behalf of specific users or client applications. They initiate discovery when a service is required.
- *Directory Agents (DAs)* maintain service information in a repository. A DA is the central element that facilitates discovery actions.

Note that the word 'agent' here refers to a more general and abstract definition. It does not necessarily mean agents in the sense of Agent Technology which will be discussed later.

Both SAs and UAs communicate with the DA. Therefore, in order to initiate the discovery mechanism, SAs and UAs must first discover the DA. The address of the DA can be manually configured. Although this requires little work, it is not a flexible solution. For dynamic configuration, the Dynamic Host Configuration Protocol (DHCP) [9] is used. DHCP servers are maintained by human administrators and they can provide the address of the DA when requested.

In both the above instances, prior knowledge of where to obtain the DA address is required. Another option would be the use of real discovery to find the DA.

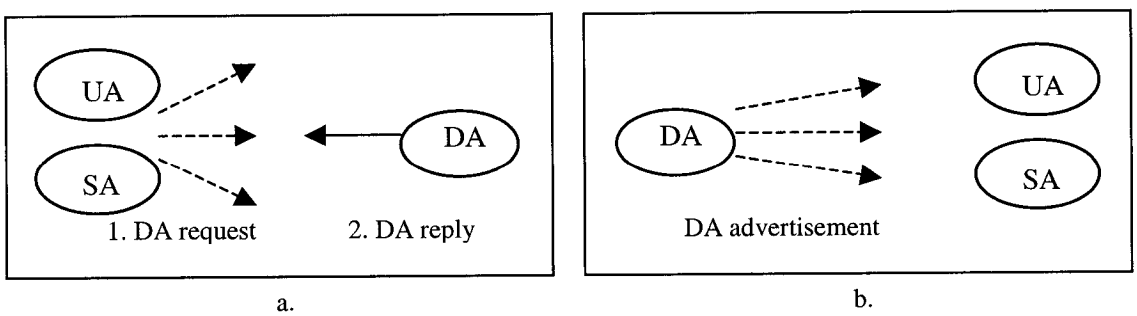


Fig 2.1: DA Discovery

Fig 2.1 demonstrates the two possibilities of DA discovery [10]. The first option (Fig 2.1 a.) is to have the UAs and SAs make their requests to discover the DA. This is done through multicasting to the network. Upon receiving the request, the DA will reply to the sender, thus establishing communication between the two entities. The second option (Fig 2.1 b.) works in the opposite direction. The DA advertises itself by multicasting to the network. UAs and SAs simply need to listen for the advertisement to discover the DA. This multicasting is done periodically to make sure all agents receive the information.

Once SLP is up and running, network service discovery begins with the registration of services.

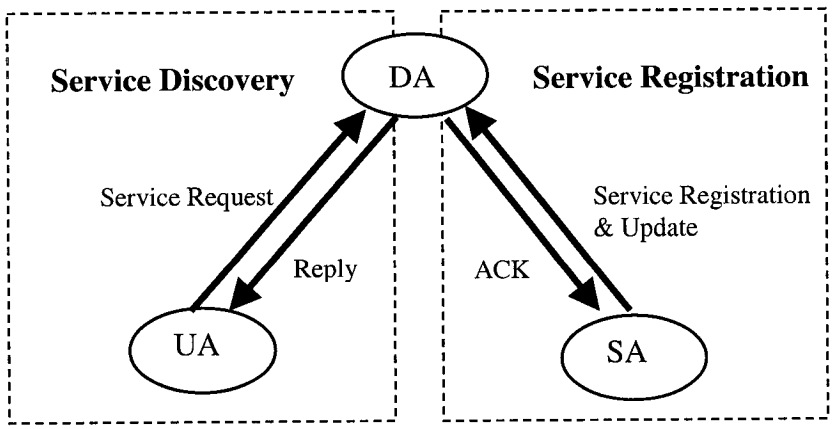


Fig 2.2: SLP Architecture and Operations

Fig 2.2 illustrates the operations of SLP [10]. There are two phases involved. The first one is the Service Registration Phase during which the SAs register the services they represent with the DA. The detailed service information and attributes may be sent along with the registration message. Acknowledgement is sent back by the DA once it

receives the registration and updates its service repository. SAs may also update the DA when changes occur in the services.

The second phase is the Service Discovery Phase. This is when UAs would contact the DA to request a service. UAs only initiate discovery when a service is required. The DA queries its repository to find matching services based on the criteria of the request. This is where a lookup service such as LDAP can be used. In fact, as McGrath [2] mentioned, SLP is designed specifically to work with LDAP although it is not required. Once the DA finishes its search and has a list of matching services, it then replies to the requesting UA sends it the addresses of these matching services. The UA now has access to the type of service it requires. The two phases of SLP can operate concurrently so that the DA always has the most current service information to be queried.

Another technique that SLP uses to maintain current information is *leasing*. Leasing refers to the use of a registration period. After a certain amount of time, SAs must reregister with the DA or its registration will be removed. Leasing offers the added security in that even if a service terminates prematurely, its lease will eventually expire, thus keeping the DA's repository current.

SLP's directory service can offer a rich set of features since the repository stores detailed information on services. Not only is it able to perform powerful queries based on service attributes, it can also provide many other capabilities such as service browsing for the clients. Thus these are bonus features for clients to use in addition to the transparent discovery service.

Large networks with thousands of services can put a heavy strain on the DA. To lighten the load, there can be multiple DAs in a SLP setup. SLP agents can find all the DAs using the methods described previously. The different DAs should all have approximately the same view of the network services. Since UAs may communicate with any DA to perform discovery, the workload is therefore distributed among the DAs.

While SLP supports multiple DAs, it also allows for directoryless operation where no DA is present. In this case UAs would broadcast their service requests to the network. SAs who provide the requested service would unicast replies to the UA. SAs may also broadcast service advertisements. Broadcasting is an expensive communication method that costs significant bandwidth. Therefore, this directoryless operation mode is best suited for small networks with fewer clients and services. Since SLP is flexible with the use of DAs, it is scalable from small networks to large enterprise networks.

SLP is a well-documented, vendor-independent and language-independent protocol. Many influential companies, such as IBM, Apple and Novell, are involved in its development. There also exist many other research implementations [3, 11, 12]. SLP therefore plays an important role in Service Discovery.

2.2.2 Jini

Another well-known Service Discovery protocol is Jini [13] developed by Sun Microsystems. Jini is based on Java technology and thus is tightly bound to the Java environment which provides uniformity across different platforms. It assures the identical behaviour of Java objects and code everywhere. This includes what is called a type system [14]. Java is an object-oriented language and each Java object must have a

type. In terms of the Jini discovery protocol, each service has a service type. The types are organized into tree hierarchies. For example, a specialized photo printer can be of type `PhotoPrinter` which is a subtype of `Printer`. This illustrates the object-oriented properties: inheritance and polymorphism. The discovery of services belonging to a specific type may also return services residing in the subtree of that type.

Jini uses its own lookup service as the central repository in its architecture. Similar to SLP, there may be multiple lookup services. Lookup services may maintain identical service information to make the system more robust. They may also contain different sets of information and thus be distinguished into lookup groups. Lookup groups divide services into logical sets and services can be configured to only register with certain lookup groups.

We can see that Jini has a similar architecture as SLP. The communication pattern is also similar in that services register with lookup services and clients ask lookup services for required services. Multicasting is involved in finding the lookup services initially, although lookup services will announce their presence to the network periodically as well. Leasing is also used extensively to help maintain the most current service information. Thus Jini follows the general operation procedures of SLP.

Jini's object-oriented nature translates into communication being done via the exchange of serialized objects. Each service, whether hardware or software, has a proxy object that implements the defined interface for the type of that service and any additional specialized methods specific to this service. Note that a proxy object can implement more than one interface if a device offers a combination of services (e.g. an all-in-one

printer) [14]. In this case the object represents multiple types and is therefore a candidate for searches of any of these types.

When services register with lookup services, they will upload their proxies to the lookup services. The proxies are downloaded by clients who discover them through the lookup services. Clients can then directly access the services by invoking the methods provided in the proxy. Because of the use of objects, we can potentially encapsulate other code that clients need to utilize the services (e.g. device drivers). Therefore the serialization and encapsulation of objects allow for the quick and direct access of services. It would significantly reduce the amount of messages that need to be sent. This is a powerful feature. It is the main advantage that sets Jini apart from other discovery protocols.

When it comes to querying and matching, Jini provides simple and straightforward capabilities. Matching can be done by using an interface description specified by the client making the search request [15]. If a proxy in the lookup service implements the same methods as those required, then this proxy is a match. Another option is to search by attributes [14]. Clients simply indicate whether certain attributes for the service they require should be present or not. While these methods do not provide as much complexity as other protocols such as SLP, they are sufficient for most discovery purposes and simplicity is one of the main goals of Jini.

Although Jini offers some unique and very useful discovery features, it does have its share of problems. For one, it is not interoperable with language environments other than Java. The benefit of Java's uniform nature also ties Jini to the language. Secondly, unlike SLP, Jini cannot operate without the central lookup services. Thus it may not be

the best option for a small network or a dynamic environment. Furthermore, devices are required to have Java Virtual Machine (JVM) installed in order to participate in discovery. This is a problem for small devices with limited storage space. It can be remedied by having docking stations for small devices or assigning a JVM on another device to service the small device [14]. Although these solutions would allow small devices to use Jini, they create the added problem of inter-dependence between devices. Despite these drawbacks, Jini is still a popular protocol for Java-based clients.

2.2.3 Universal Plug and Play (UPnP)

We have seen two discovery protocols that uses primarily centralized designs where central repositories are needed for storing service information. Universal Plug and Play (UPnP) [16], developed by Microsoft, is a protocol that adopts a distributed approach of discovery. There is no service registry and devices rely on each other to find what they need. UPnP is suited for use in small office or home networks.

UPnP is an extension of the Plug and Play technology. It is specifically designed for TCP/IP networks. It uses many existing protocols and technologies to support its own operations. The UPnP operations are divided into six areas.

Addressing: Every device must have an IP address. When a device enters the network, it obtains an IP address as an initialization step. This address is normally assigned by a DHCP server. It may also be statically configured, obtained through a DNS server, or generated using the Auto-IP protocol.

Discovery: Discovery is carried out using the Simple Service Discovery Protocol (SSDP) [17]. Clients search for services by multicasting requests. Devices who offer the

requested service reply the specific client through unicasting. Included in this reply message is a link to the detailed description and access information of the device and its services. Devices in turn also advertise their services by multicasting. The advertisement message is the same as the reply messages sent to clients directory. Again, leasing is used in UPnP to give clients fresh views of the network services.

Description: This refers to the method that clients can access the detailed descriptions of services. The properties of a device and its services are described using the XML syntax. UPnP defines its own standard XML template for devices and services [18]. The XML file for a device is located at a URL which clients would obtain at the Discovery stage described above. HTTP is used for clients to download the XML device description. XML service descriptions can then be accessed in the same way by using the URLs stated in the device description.

Control: XML service descriptions contain information on how to access the services. Clients issue commands to the services using the Simple Object Access Protocol (SOAP) [19]. SOAP is a HTTP-based protocol that operates over TCP networks. Commands, in the form of XML, will be sent to what is called the control URLs of the services. These control URLs can be found by the client in the XML device description. Results of actions invoked on the service is sent back to the client again using HTTP.

Eventing: Other than specifying access details, the XML service description also contains variables that represent the state of the service. The state of a service may not be fixed and can change from time to time. Clients may choose to subscribe to receive state change information from the services they have access to. A notification message will be

received if any of the state variable changes. Notification messages are sent by devices that offer services and this is done via the General Event Notification Architecture (GENA) [20]. These messages, once again, are in the XML syntax which contains the state variables and their new values. Subscriptions also have leases and clients must renew their leases to keep receiving state change notifications.

Presentation: Devices can optionally have HTML pages for their services. If these pages exist, then their URLs are specified in the XML service descriptions. Clients can access the HTML pages using HTTP request. These pages can be viewed with web browsers and are considered interfaces to the services. They may contain service information for the users to browse and control elements for service access.

The set of protocols and Internet technologies that UPnP employs are all standards that have been widely used across different platforms. These technologies coupled with operation on IP networks ensure successful interoperation between a wide variety of devices that are made by different vendors and that may reside in different networks that support IP traffic [21]. Hence UPnP is characterized by its universality.

XML is a feature that is used extensively in UPnP. It is an extremely powerful language that allows us to document and specify virtually anything about an entity. In UPnP, device and service descriptions are in XML form, as well as certain communications between clients and devices. All useful information regarding services and other events can be conveniently found and translated into appropriate actions. Yet it still opens up opportunities for future enhancements to UPnP operations when more features may be added in the XML descriptions to be used.

Although UPnP is a distributed discovery mechanism, its dependence on the many existing protocols may hinder the true distributed nature. Protocols such as HTTP and DHCP all require their own servers somewhere [2], therefore these servers must be setup before discovery operations can take place. UPnP is still a young protocol compared to other existing protocols such as SLP. However it is a promising Microsoft technology that is being supported by many other companies such as Intel, Compaq and Dell.

2.2.4 Other Discovery Protocols

There are also many other Service Discovery protocols on the market. Salutation [22] is such a protocol from the open industry Salutation Consortium. It uses what it calls the Salutation Managers as the central control element. While Salutation operates in a similar way as SLP and Jini, it is however defined at a higher level (i.e. above the transportation layer), thus making it more technology independent.

Another protocol is the Bluetooth discovery protocol. Bluetooth is a technology used for short range wireless transmission. It uses the Service Discovery Protocol (SDP) [23] to locate services offered by Bluetooth devices. The protocol supports discovery and service browsing, but not service access. Access can be done using other techniques or existing protocols such as Salutation. More examples of discovery protocols include University of California Berkley's Ninja [24] and IBM's Deapspace [25, 26].

2.3 Agent Technology

It is apparent that much research has been done on the topic of Service Discovery. In a continued effort, this Thesis aims to investigate a new approach through the means of agent technology. In doing so we will discover and learn about the benefits that this new approach can bring to the Service Discovery field.

2.3.1 Agent Definition and Characteristics

Agent Technology has been gaining attention in recent years. Agents, as individual autonomous and intelligent entities, can offer great convenience and new features for many computing applications and networks. In order to understand why this is so, we need to have a good understanding of what agents are.

There are various definitions for agents given by different scholars. But the concept of agents was first used in the field of Artificial Intelligence (AI). Here are a couple of definitions that emerged from the AI community.

"An autonomous agent is a system situated within a part of an environment, that senses that environment and acts on it, in pursuit of its own agenda and so as to effect what it senses in the future." [27]

"Autonomous agents are computational systems that inhabit some complex, dynamic environment, sense and act autonomously in this environment, and by doing so realize a set of goals or tasks for which they are designed." [28]

The emphasis of agents in the AI context, as can be seen in the above definitions, is on the agents' autonomous, reactive and intelligent nature. First of all, an agent exists for a reason. Each agent has its own goal or set of goals to accomplish. Agents are not controlled. They make intelligent decisions and act on their own accord. They are smart enough to be aware of and react to their environment, and they may learn from past experiences to improve on their future actions. Agents are also social entities in that they are capable of communicating and negotiating with other agents in order to carry out their tasks. The AI definitions focus mainly on what agents can be capable of and how intelligent they can be. Here is a more general definition that also touches the role of agents.

"Intelligent agents are software entities that carry out some set of operations on behalf of a user or another program with some degree of independence or autonomy, and in so doing, employ some knowledge or representation of the user's goals or desires." [29]

This general definition states the common goal of all agents which is to serve other entities. The most important intelligence features of agents here are independence, autonomy and the ability to achieve client goals. Intelligence in the sense of AI (reactivity to environment changes and learning) is not a strict requirement since it is a broad and difficult area in AI. This agent definition is widely accepted in the computing society.

A subcategory of agents is the mobile agent. These agents may move within a system or from host to host. This kind of mobility allows for many advantages. For example, an agent residing on a heavily loaded system can move to a lighter loaded one while still be able to carry out its tasks. Mobility further enhances an agent's independence. It provides more options and flexibility for the agent to achieve its goal(s).

2.3.2 Agent Standards and Platforms

Since agents are social entities that may be mobile, it raises the issue of interoperability between different agents, and between agents and different hosts. Therefore a common standard is needed to specify the communication pattern for agents and to organize the agents. Organization refers to the existence of agent execution environments or agent platforms. In an agent-driven environment, each host should have an agent platform to manage its own agents and any foreign agent that comes from other platforms. Such a platform is commonly implemented as an application running on the operating system.

Two interoperability standards for agents are the Mobile Agent System Interoperability Facility (MASIF) [30] specified by the Object Management Group (OMG), and the specification from the Foundation for Intelligent Physical Agents (FIPA) [31]. MASIF defines interoperation between mobile agent platforms while FIPA focuses on using generic agent technologies that maximize interoperability between agents within and across different platforms [32].

There are numerous platforms implemented based on the two standards. One such platform is the Lightweight Extensible Agent Platform (LEAP) [33] which is developed based on JADE [34]. LEAP, as its name states, is a lightweight platform mainly for use on small devices. It is FIPA-compliant and therefore can interoperate with other FIPA-compliant platforms. It is also extensible in that it offers additional features for devices that have the capability to run them.

Another popular platform is the FIPA Open Source (FIPA-OS) agent platform [31] originally developed by Nortel Networks. FIPA-OS is a component-based platform that has much more complexity than LEAP. However there also exists a lightweight version called Micro FIPA-OS for the smaller devices. FIPA-OS is the chosen platform for the realization of the agent-based Service Discovery model proposed in this Thesis.

2.3.3 Agent Technology as a New Paradigm

In a traditional client server model, communication is carried out in sessions and messages are exchanged in a synchronous manner. In an agent environment, because of the nature of agents, operations are done asynchronously and need not be session based. A client may instruct an agent on its goals and move on to other tasks while the agent performs its duties. This is much more efficient since a greater amount of tasks can be done not to mention that much bandwidth can also be saved. One can imagine an overall picture of agents moving around in a network or in between networks, associating with each other to get their jobs done, creating a large agent community.

Existing legacy systems can adapt to Agent Technology by using wrapper agents that serve as the agent representatives of these existing applications. This allows any

application to be incorporated into the agent world. With all of these benefits and convenience that Agent Technology brings, it is easy to be considered a new application paradigm, possibly even the next widespread standard. It is not there yet since it is still a very young technology and many issues such as security remain to be properly resolved.

The enthusiasm in Agent Technology and its promising future serve as strong motivations for us to investigate how it can contribute to Service Discovery. Advantages such as agent mobility bring great convenience in certain aspects of our design. Agent Technology is easy to adapt to Service Discovery. As we have seen from the existing protocols, especially SLP, the abstract concept of agents has already been used. It is therefore only fitting to use Agent Technology in Service Discovery.

2.4 Ad-hoc Communications and Agents

We have discussed the basic concepts of ad-hoc communications in the previous Chapter. Since ad-hoc usually refers to wireless devices, it follows that these devices are mobile and not stable fixtures in the network. In this Section we introduce the types of ad-hoc communication environments our discovery models aim at serving.

2.4.1 Ad-hoc Communication Environments

Ad-hoc communications automatically leads us to think of ad-hoc networks. The common type of ad-hoc network is the Mobile Ad-hoc Network (MANET) [35]. All devices (or nodes) in a MANET operate autonomously and independently. One of the main issues of MANET is routing which is very difficult when the network topology

changes constantly. There are two general approaches for routing: proactive and reactive routing. Proactive routing requires nodes to know of the topology before receiving routing requests. Some proposed protocols for proactive routing include Destination Sequenced Distance-Vector (DSDV) [36] and Wireless Routing Protocol (WRP) [37]. Reactive routing (also called on-demand routing) on the other hand, only becomes active when routing is needed. Examples of reactive routing protocols include Dynamic Source Routing (DSR) [38] and Ad-hoc On-demand Distance Vector (AODV) [39].

A MANET is a typical example of an ad-hoc communications environment. However, the possibilities are not narrowed down to this one instance. Whenever mobile devices are part of a network, opportunities for ad-hoc communications would exist. Even non-mobile devices can be set up in a network at any time, and without the presence of an administrator, ad-hoc communications become necessary.

Our proposed discovery models are aimed at two general categories of ad-hoc communications environments. The first of these is a hybrid environment where there is a fixed portion of the network always present. Such a network can be very common. For example, users carrying mobile devices can attend a meeting in a conference room already equipped with fixed devices such as telephones and projectors. The second type of ad-hoc communications environment would be a fully dynamic environment. This is the more general scenario. A discovery model suited for a fully dynamic network will also function in a hybrid environment. Thus the hybrid and the fully dynamic networks are the types of ad-hoc communications environments we will be constructing the discovery models for.

2.4.2 Agents in Ad-hoc Environments

Agent Technology, as described earlier, offers great efficiency, something that is essential in ad-hoc environments. Power can be conserved if devices delegate agents to travel around the network instead of establishing communications themselves. Furthermore, agents can keep carrying out their duties even if their owner devices disconnect from the network.

Issues such as routing and security will fall in the hands of the agents. An agent may choose to perform its tasks during opportune moments (i.e. when connection is more stable and sufficient bandwidth is available) therefore efficiently utilizing network resources. Also security measures are carried out on a higher level between communicating agents. This reduces the likelihood of network devices being threatened. We can see that unlike the traditional synchronous and connection-oriented model, Agent Technology is much more suited in today's mobile and limited bandwidth network environments.

2.5 Related Work

One effort in bringing Service Discovery into ad-hoc networks is the Konark method proposed by Helal *et al* [40] from the University of Florida. Konark is an approach that is designed specifically for ad-hoc networks. It puts more emphasis on device independent software services, in particular mobile commerce software services. Konark assumes IP level connectivity and is thus independent of the network layer.

Konark uses a distributed discovery approach in a peer-to-peer topology. Every device can act as both server and client. A main discovery component called the Konark SDP Manager is present on each device. The SDP Manager is responsible for discovering services, registering local services, and advertising the services. It maintains a service registry that stores all local services and services that the device has discovered. The registry is a tree structure. Each node in the tree represents a generic service type and each leaf represents a specific service. Any node can have leaves and nodes classified immediately under it. The tree therefore is a hierarchy of levels that represent service classification. Discovery and advertisement can be done at any level of the tree.

A client requesting a service will multicasts a discovery message. The message contains either a service path (for all services under a generic type) or a keyword (for a specific service). Devices who offer services that meet the client's requirement will create and send one reply message for every matching service they find. Service advertisements are also actively multicast to the network periodically. For a client to access a service it has discovered, it must first retrieve the detailed service description of the service from its URL. This URL is specified in the reply or advertisement message sent by the server device. The service description is an XML file that contains functions offered by the service and access details. The client can then send commands to the service through SOAP.

Konark is, in many ways, very similar to UPnP. Our proposed distributed discovery model employs an agent platform that manages all the communication between devices. Thus, without having to deal directly with various different protocols such as HTTP and SOAP, our model aims to achieve more simplicity for the devices.

Another discovery project being developed for ad-hoc environments is the Mobile Agent Runtime Environment (MARE) proposed by Storey *et al* [41] from Lancaster University. MARE is aimed at supporting operations within a mobile ad-hoc environment. In this approach Agent Technology is explored. Agents move around the network and communication is done via distributed tuple spaces. A tuple space is similar to shared memory where data is stored and can be shared and updated by different processes. The use of tuple spaces allows anonymous communication between entities while still getting the job done. The idea here is that, in addition to anonymity in the network, agents can communicate with one another even when they are on the move.

Agents in the MARE system are used to bring services to where they are needed so that there will be no extra transmission for clients to remotely access the services. Therefore MARE is a bandwidth-efficient approach. The design models proposed in this Thesis are focused on encapsulating all services and other control elements into agents governed in an entirely agent-based environment. Also we explore the area of inter-domain discovery which has not been greatly researched in the current discovery field.

2.6 Summary

In this Chapter we have discussed some of the technologies that form the basis and support for the Service Discovery work proposed in this Thesis. The concept of lookup services paves the way for automatic discovery as lookup services can be used to act as directory servers in a centralized discovery model. There are numerous discovery protocols on the market of both the centralized (SLP, Jini) and the distributed (SLP, UPnP) type. We have seen that Agent Technology is a promising new technology.

Agents, due to their autonomous and independent nature, can bring great convenience and efficiency. This is especially realized in MANET where device power and bandwidth are limited. Currently there are several projects working on discovery in ad-hoc environments (Konark, MARE). While these projects all use various technologies to carry out discovery operations, the discovery models proposed in this Thesis fully employ agents to encapsulate operations and the use of other technologies.

Chapter 3

Service Discovery Principles & Requirements

Any act of discovery always falls into one or a combination of several general categories of operation modes. In order to carry out the design of a discovery model, a clear understanding of these operation modes is needed. This Chapter examines these Service Discovery principles in detail. Their advantages and disadvantages will be discussed along with what types of network each is suitable for. Other frequently used techniques such as leasing and service access methods will also be reviewed. Finally we will look at the target environment for a Service Discovery model, specifically the environment description laid out for our discovery models proposed in this Thesis. The intention of this Chapter is therefore to provide essential information on project requirements and common discovery principles that will be used in our work.

3.1 Operation Modes

The main questions in performing Service Discovery are how to become automatically aware of other services and how to look for services. Assuming that initially none of the devices knows of any service provided by the other devices in the network, how can discovery be carried out? We have already seen how discovery is done in some of the existing protocols. Here we organize and examine the techniques in detail. There are several possibilities of action. We can categorize them into different views of operation modes.

3.1.1 Pull vs. Push

This view deals with which party would initiate the discovery process. The *pull* mode refers to user-side initiation, meaning the device in need of a service will make its request. When it receives a reply of the service it seeks, it can then contact the host device to access this service. Pulling can be seen as active discovery on the seeker's part and on-demand or reactive discovery based on user's needs. Clients make service requests that are heard by servers who can then respond appropriately with matching services.

The *push* mode operates in exactly the reverse direction as the *pull* mode. Devices that offer services are the ones actively advertising or pushing service information to clients (i.e. other devices in the network) periodically. Clients may store and update the service information they receive for potential future use. Pushing is a passive method of discovery for clients in that they are receiving service information

instead of seeking for them. With respect to servers, pushing is a proactive discovery method due to the active advertisement of services.

The *pull* mode is more bandwidth-efficient since communication only happens when services are needed. However this bandwidth efficiency comes at the price of slower response time. There would be more delay for discovering services. On the other hand, the *push* mode greatly improves on the response time. Because the clients are constantly being fed service information offered by other devices, it is highly likely that no discovery needs to be done when a service is actually required. The actual discovery process has already been performed by devices pushing service advertisements to the clients. Thus, discovery is carried out *before* the client is even aware of what services it will require. Of course all the repeated service advertisements can put serious strain on the network bandwidth. Also many clients would not need all the services being advertised, thus bandwidth is being wasted as well. We can therefore see a tradeoff between bandwidth efficiency and response time. It is important to determine which factor is more crucial and employ the *pull* and *push* modes accordingly.

3.1.2 Centralized vs. Distributed

With the *pull-push* view, the focus is on the behaviour of discovery. Now we examine the possible structures of Service Discovery models. This can be categorized into *centralized* and *distributed.*

A *centralized* discovery architecture generally means the use of a repository. This is a central database that stores network service information. It is where a lookup service may be employed. Services register and advertise through this repository, and clients

may use it to discover services. As Service Discovery is aiming to eliminate human administrative duties, the repository should be kept automatically updated to always have the most recent service information. Using a dedicated service repository means that maintenance work is required, but it also points at the capability of performing complex search queries that may result in finding better service matches. It is also possible to federate repositories or organized them into hierarchies that may provide wider range and more flexibility for querying.

A *distributed* discovery architecture refers to the absence of a central service repository. Devices inevitably depend on each other to perform Service Discovery. There is less maintenance and configuration overhead. Service advertisements and requests are made through multicasting or broadcasting. Since there is no specialized component to store and maintain service information, the querying capabilities are consequently simpler. However, complex queries are less likely needed in a distributed design since it should be relatively simple for individual devices to check whether they offer a required service or not.

In a *centralized* discovery design, a large amount of effort is put into the maintenance of the central repository which is the core of the design structure. Since all discovery-related communications are done via the repository, mostly only unicast messages are necessary and this conserves bandwidth. *Distributed* architectures do not concern with service repositories and are much simpler to configure and execute. However, without a central component to direct discovery-related communications, multicasting is required. Frequent use of multicasting can hurt the bandwidth significantly. Furthermore, a large portion of messages sent are wasted since not all

devices would offer a required service and a client may receive many replies to its service request but may only need one.

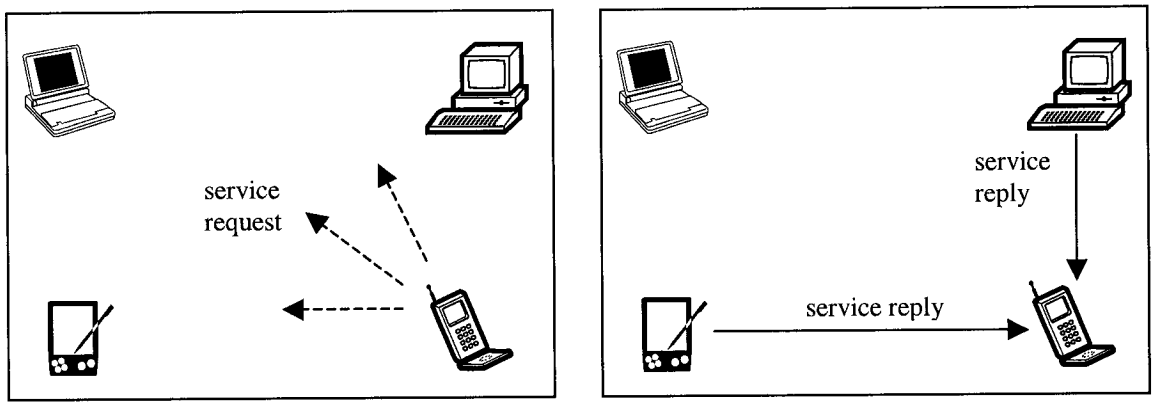
Based on these characteristics, we can say that a *centralized* discovery design is more suited for static networks, or for hybrid networks that have a static portion with fixed nodes (i.e. devices). In such a network, the existence of a central repository can be guaranteed. And because of the fixed resources, the amount of maintenance and configuration required for *centralized* discovery models can also be more easily carried out. The *distributed* design, on the other hand, is more suited for small networks with fewer services, or for dynamic environments such as ad-hoc networks. Devices communicate with each other for discovery, but they are not dependent on each other or any fixed network component. Thus changes to the network topology will not have any significant effect on the discovery process, therefore making the *distributed* design suitable for dynamic environments.

3.1.3 A Combined View

The *pull-push* modes give us an outside view of how discovery is achieved for the users, whether it is client or server initiated. The *centralized-distributed* view deals with the internal discovery structure. If we put these two dimensions together, then four design possibilities can be formed.

Distributed Pull

Clients broadcast service requests to other devices.



a. Broadcast Service Request

b. Service Reply

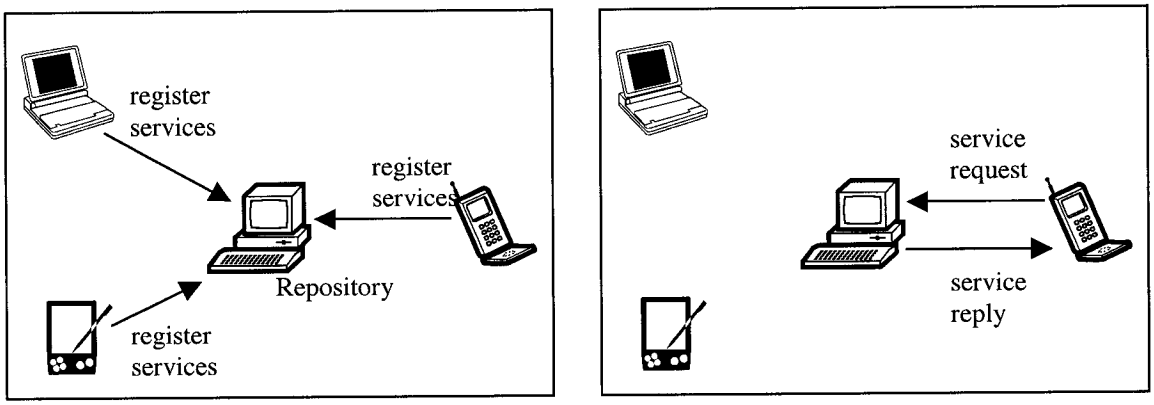
Fig 3.1: Distributed Pull

With distributed pull, there is no central repository and clients initiate discovery by broadcasting service requests (Fig 3.1 a.). When a device receives such a request, it proceeds to check if itself offers a matching service. A service reply is sent to the client if the device does provide the service required (Fig 3.1 b.).

It is possible for a client to receive more than one result since different devices may all offer the same type of service. In this situation it is up to the client to decide what to do with the services. One possibility is to only use the service that is received first and disregard the rest. Another solution is to find the best match within the results. The best path to take depends on factors such as client device capability, specific requirement details of the service and network characteristics.

Centralized Pull

Clients initiate discovery with the control of a central repository.



a. Service Registration b. Discovery

Fig 3.2: Centralized Pull

As illustrated in the above figure, there is a central repository where devices would register their services (Fig 3.2 a.). When a device needs a specific service, it initiates the discovery process by sending a service request to the central repository. The repository queries its service database for a suitable match and returns a service reply to the requesting device (Fig 3.2 b.).

While this is a very organized and bandwidth-efficient approach, it does have some drawbacks. First of all, the centralized design exposes a single point of failure. The discovery system will not function if the central repository is down. This may be less of a problem if there is a federation of several repositories. The issue of locating the repository must also be considered. The address of the repository can be statically configured into all the devices if the network itself is of a static nature. But ideally it is

more beneficial for the devices to be able to locate the repository automatically. An example of this will be seen in one of the existing discovery protocols to be discussed.

Distributed Push

Devices broadcast service advertisements to clients (i.e. other devices).

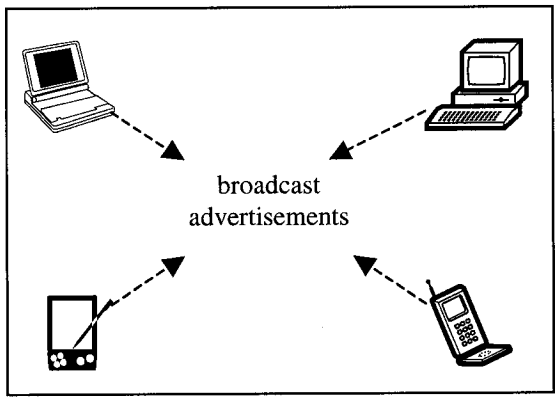


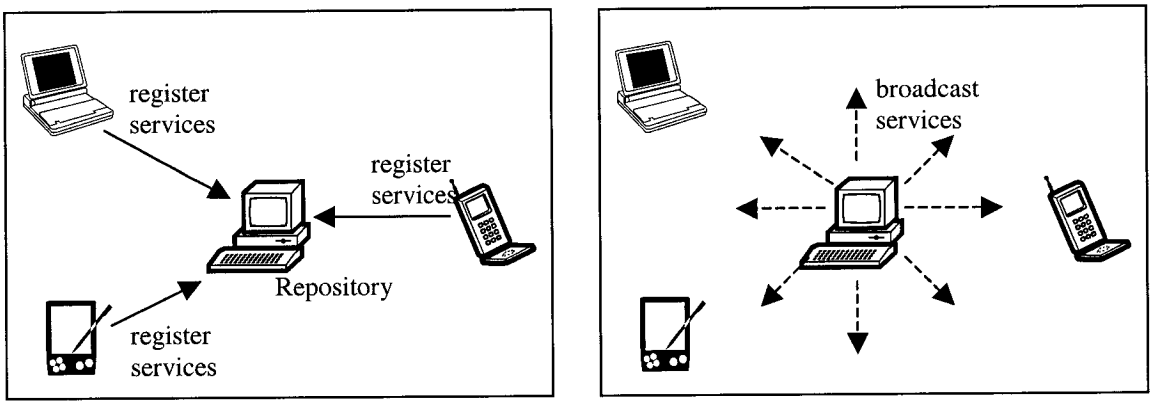
Fig 3.3: Distributed Push

In distributed push, devices periodically broadcast their service advertisements to the network (Fig 3.3). Therefore, each device would store these service information. When a service is needed, the client device simply needs to find a match within the information it already has and then contact the device that offers the service.

Distributed push allows devices to always have the most up-to-date view of available network services. One issue though is determining the length of the broadcast interval. With bandwidth concerns in mind, this interval cannot be too long. It should be dependent on the network size. Broadcast can also be initiated upon request. In this case, a device in need of a service that cannot be found yet may trigger other devices in the network to rebroadcast, therefore combining the *push* and *pull* modes.

Centralized Push

The meaning of *push* is to proactively provide all clients with service information whether they need it or not. Thus *push* is always done by multicasting or broadcasting. Centralized push in this case loses its meaning of being centralized.



a. Service Registration b. Pushing

Fig 3.4: Distributed Push

The central repository will perform the broadcast of service information (Fig 2.4 b.). Devices can then contact other devices directly to access services. There is no longer a streamlined unicast communication pattern that is the key advantage in a centralized design. Therefore centralized push is not a useful design with distinguishing characteristics.

The *push* and *pull* modes, as described in the previous design options, have their own merits and drawbacks. *Push* enables quick discovery and constant service awareness, but tends to waste and use too much bandwidth. *Pull*, on the other hand, is more bandwidth-efficient since communication is established only when needed, but it

causes extra delay for discovery. These two modes may therefore be used together to combine the advantages and reduce the drawbacks. It is possible for a Service Discovery design, either centralized or distributed, to employ both *push* and *pull* models. In fact, we have already seen examples of this in some existing discovery techniques (i.e. SLP, UPnP). The combined used of these two operation modes will also be explored in the discovery designs proposed by this Thesis.

3.2 Common Techniques

The different operation modes form a very general picture of how a discovery model operates. Detailed operation processes and architecture differ from model to model. However, there are common areas of concern. Many discovery models would also employ the same type of elements and techniques in certain areas of their design. This Section looks at some of these elements.

3.2.1 Architectural Components

Most Service Discovery models have a similar high-level structure. They all contain two or three types of main components as we have seen in the different operation modes. Service Discovery is centered around clients discovering services. Thus, first there are the network services that include both hardware-based services (e.g. printer, fax machine) and software applications (e.g. e-mail). Then we have the clients that access the services. Software entities are used to represent the services and clients. These are the two necessary types of components in all discovery models. If a design is directory-

based or centralized, then there will be another component type: a main control element that maintains the services in registries and controls interactions between clients and services. This high-level architecture serves as the skeleton for different discovery models including the ones proposed in this Thesis.

3.2.2 Self-Healing

One of the main concerns of a discovery model is to constantly keep a consistent and current view of the network services. Sometimes services may become inactive without warning (e.g. a system crash). It is important to put in place a mechanism that can detect such failures so as to eliminate incorrect service information from the current view. One method is to use bounded retries [42]. This approach requires clients or central repositories to constantly test the servers and find out the continued availability of the services. If the testing message is unacknowledged for a number of times, then the service is deemed unavailable and removed from the current view.

The far more popular and widely adopted method however, is to use leasing. The concept of leasing was introduced in the previous Chapter. Leasing refers to the lease time of a service. When a service is registered with a repository or made known to a client, the clock starts ticking for this service. If the lease time is reached and the service has not renewed its registration, then it no longer exists and is removed from the current view. Leasing requires less transmissions than bounded retries and therefore conserves bandwidth. However, the downside is that we can still have incorrect views of services between the time that a service becomes unavailable and the expiration of its lease time. Therefore it is important to find a good lease time that gives a very short period of

inconsistency. The lease time is very difficult to determine. It should be dependent on the nature of the service and the network characteristics. Thus different services may have different length for their lease times.

3.2.3 Service Access

Thus far we have concentrated on the discovery side of a design. Many discovery models also provide the means for accessing services after finding them. Some provide detailed service descriptions that inform clients of how to access the services (UPnP, Konark). If a network has many specialized services then this method would certainly be very useful for clients to access all the special service features. Other discovery models may just offer the locations of discovered services and leave the details of interaction to the clients and the servers themselves. This is simple and effective for smaller networks with standard services. Clients are assumed to know how to access the services. Other access methods also exist such as bring services to the clients (MARE). In our proposed models we shall explore the access possibilities with agents.

3.3 Target Environment

Before designing a discovery model, it is important to have a clear knowledge of the type of environment the model is targeted at. The same goes for the designs proposed in this Thesis. Our discovery model is part of a collaborative effort toward building an ad-hoc communications system for a conference room setting. Besides Service Discovery, the ad-hoc system also involves other aspects such as Context Awareness and

Conferencing. Although the design of Service Discovery is independent from the design of the other areas, some cooperative efforts are still needed especially for the implementation and final integration of the different components.

The target network environment in our case is a physical conference room. Users (equipped with mobile devices) attending a conference or a meeting can come in and out of the room as they wish, thus creating an ad-hoc environment in the conference room. The devices that users bring to the room may provide one or more services for other users in the room. Because the environment resides in a physical room, it is a small network that will not have a large number of services. The room network however, may be connected with other similar conference rooms and therefore services may be shared between different rooms.

3.4 Summary

At the very high level of a Service Discovery design, we can discern two levels of the design: one being the general architecture of the model (centralized or distributed), and the other being the general behaviour of the model (pull or push). These two dimensions make up the template for a discovery design where detailed operations and features are then mapped out. Some areas of special interest include self-healing and service access where common practices are used by many existing protocols. For the particular models proposed in this Thesis, we will be looking at these design elements as a guide. Our targeted network type is an ad-hoc environment within one or more physical conference room settings, and Service Discovery is but one of several

components that creates an ad-hoc communications system for the conference room network.

Chapter 4

Centralized Service Discovery Design

In this Chapter we present our first approach of agent-based Service Discovery for the conference room environment. It is a centralized design that has the similar high-level architecture as other centralized protocols. This Chapter holds part of the central theme that represents this Thesis' major contribution. The Chapter is organized as follows. First the basic version of the discovery design is examined. This version incorporates the main idea behind our design. We then move on to make enhancements to this basic version so as to improve on aspects such as functionality and organization. Components involved in the design architecture will each be discussed. Finally we delve into inter-domain discovery where several conference room networks can share services.

4.1 System Goals

While users carrying mobile devices can enter and leave the conference room as they wish, we also know that the room is already equipped with servers and may have existing services such as printers and projectors. Therefore we have a hybrid network environment made up of a fixed and an ad-hoc portion. Thus the challenge resides in discovery for mobile devices and discovery of services offered by mobile devices.

Due to the ad-hoc nature caused by mobile devices, one of the primary goals of our design is to always keep track of the current available network services and their status. Updates must be promptly discovered and used to generate the most current service information. While it is critical to ensure that the network is aware of its active services at all times, our other goal is to create as little discovery work on the user side as possible. When several conference rooms are connected and sharing resources, discovery should be scaled to bring services from all rooms to any user.

4.2 Basic Architecture

We first look at two options of operation. The first is to use *pull* and have the user side search for a service when it is needed. The second option is *push* and it eliminates the need for searching by providing the user side with all the available services regardless of whether they will be required by the user or not. Note that either option should keep the process of obtaining services transparent to the actual user as implied by the nature of Service Discovery. For our research we choose to use the *push* option since it reduces delay for the user when initiating services. Also our goal is to create little work

for the user, and it can be achieved by providing users with all the services. Because this discovery model is intended for use in the small network within a conference room, thus the amount of traffic generated by constantly updating the user side is not a concern.

The decision to use a centralized design comes from the knowledge that there is a fixed network portion in the conference room. Therefore the central repository can be hosted on a server residing permanently in the room. With a fully ad-hoc network, it would be difficult to achieve this.

By using Agent Technology we are in fact creating an agent environment with our Service Discovery architecture and thus must carefully regard the rules of such an environment when designing our work. One of the most important characteristics of an agent environment is the exclusive presence of agents. Every independent entity within this environment is an agent, whether it be a management entity, or service provider, or something else. Thus to adapt legacy software (traditional services such as drivers for hardware) into this environment, agents must be created to communicate with and represent such software. We therefore develop such an environment for our discovery mechanism in which all tasks are carried out by agents who communicate and interact with each other.

The natural place to start our design is the general architecture that is used by most existing discovery protocols (i.e. SLP, Jini, Salutation). SLP as a representative of these has used the abstract concept of agents in the general architecture, and thus we now take it one step further by transforming the three components in the architecture into real software agents according to Agent Technology. Hence we derive our agents as service agents (SAs) that represent the actual services, personal agents (PAs) that represent users,

and a central service discovery agent (SDA) that manages services and operates according to the needs of both SAs and PAs (See Fig 4.1).

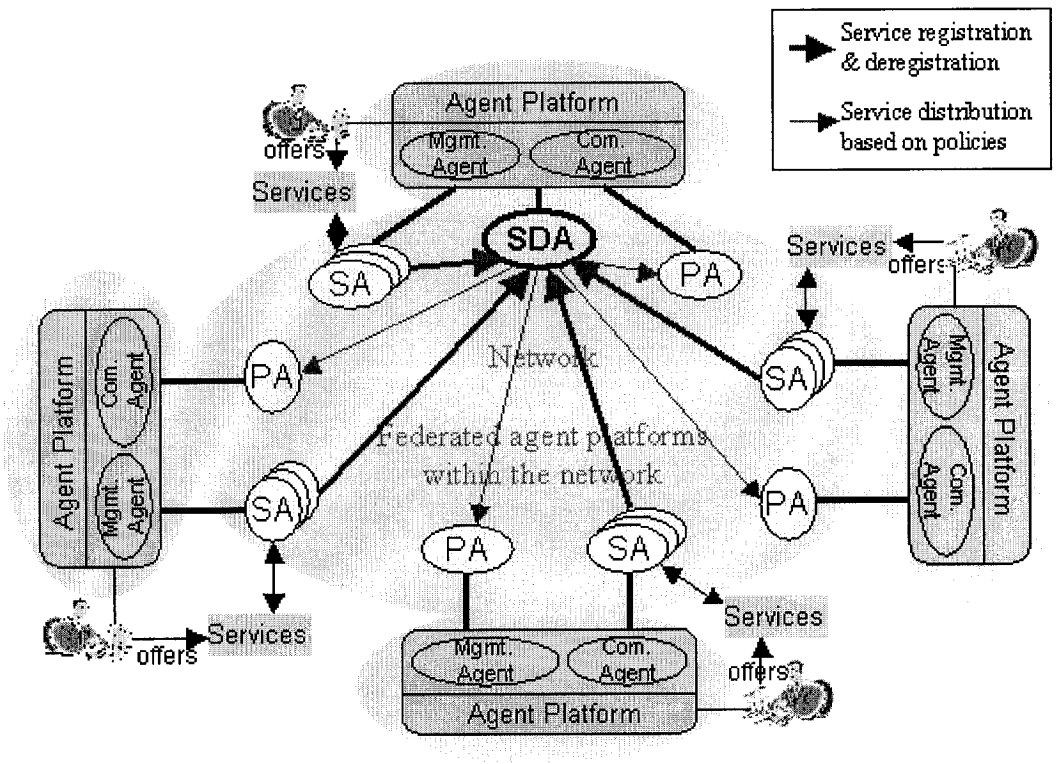


Fig 4.1: Basic Agent-based Service Discovery Architecture

Fig 4.1 illustrates the overview of the centralized discovery model. Each device runs its own agent platform. Such a platform would contain control entities. These include a management agent and a communications agent. More on agent platforms will be discussed later. Within each platform, there reside SAs for all the shared services the parent device offers, and a PA to act for the user of the parent device. One of the devices will host the SDA in its agent platform. Together, the platforms on each device form a federation. This allows for easy communication between agents on different devices.

While SLP serves as a basic template for the architecture of our design, the operations are however different. Our discovery mechanism uses mainly the *push* mode

to ensure that the users are automatically informed of all services available to them at any time. SLP is mostly based on the *pull* mode where the user agents must initiate the discovery of services upon request. Also, SLP as well as other existing protocols do not go into the territory of inter-domain discovery which we will be looking at later.

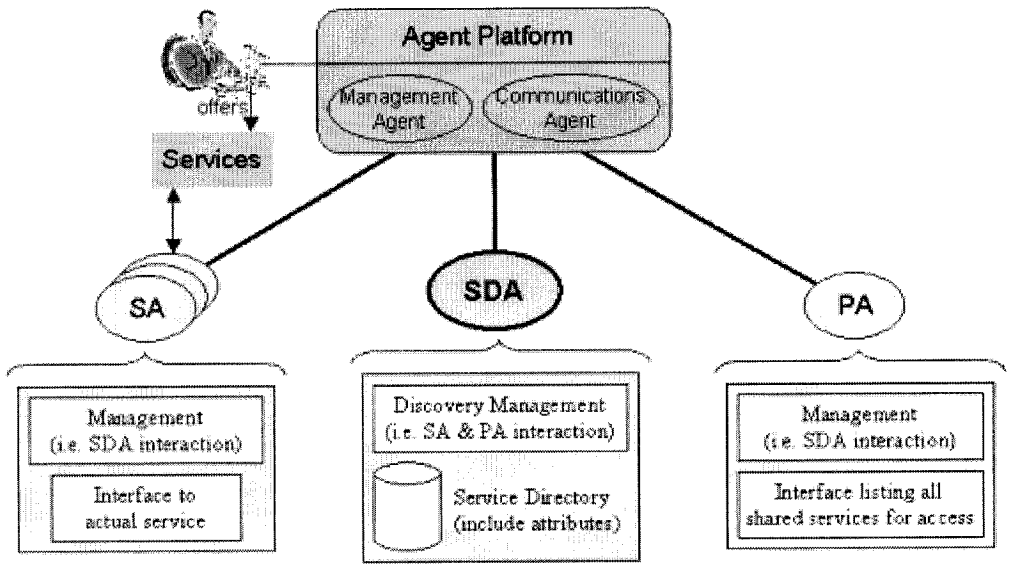


Fig 4.2: Agent Details

Fig 4.2 shows the basic make-up of the three types of agents in our centralized design. Note that as discussed of the nature of agent environments, SAs are needed to facilitate interactions with such services as printers and projectors. In this case an SA only needs to maintain an interface to the real service so as to access all its functions and properties. SAs also keep communications open with the SDA to inform it of service updates. Likewise, PAs provide their users with an interface for accessing the services, and they listen for service updates sent by the SDA. The SDA maintains a database of current services and their attributes. This directory is built based on the information

provided by SAs. The SDA also performs discovery by managing interactions between SAs and PAs. The details of operation are discussed in the following Sections.

4.2.1 Service Registration

Recall that our first concern is to have the network be constantly aware of the status of all available services. This is dependent on the interaction between a centralized discovery management entity and the services themselves. Translated to our agent architecture, the centralized entity is the SDA and the services are represented by the SAs. Therefore, following the method in the general discovery architecture, the SAs are obliged to register themselves with the SDA who keeps a record of all currently active services in the network.

In an ad-hoc environment, services can come and go without warning. Thus SAs must in correspondence be spawned and destroyed on the go. Let us look at how a typical service registration works. When a service (e.g. a printer) is brought into the network, its detection will trigger the creation of an SA for this new service. SAs are designed in such a way that once an SA is created, it will send a registration message along with all its service attributes to the SDA. Note that the SA can obtain the address of the SDA by performing a search in the agent platform for the agent with type "Service Discovery". The SDA then adds this new service as a record in its list of active services.

We also incorporate the leasing concept by having the SA re-register periodically. This is to let the SDA know of any unexpected service failure or departure so that it can remove the service as promptly as possible, thus maintaining the accuracy of its service list. SAs may also send updates such as changes in their service status (e.g. busy,

inactive, etc.) to the SDA. And when possible, an SA will inform the SDA of its exit indicating the departure of the service, therefore ending the SA's lifecycle and the registration scenario. The SDA is constantly and promptly kept informed of all relevant information that it needs to maintain the most up-to-date service list. Therefore our first goal is achieved.

4.2.2 Service Distribution

The next concern is to update the users of all the services they can access. This involves communication between the SDA and the PAs. PAs act on behalf of the users and they are aware of their users' conditions and environments. Part of their jobs is to obtain services for their users. Different users have different credentials and therefore may or may not be allowed to use certain services. These rules can be recorded in a separate policy engine represented by a policy agent for convenience.

To keep the PAs informed of services they can use, whenever the SDA receives service change information, it will consult with the policy agent to obtain a list of PAs that are authorized to access the particular service so that it can notify those PAs of the updated service information. For example, when the SDA learns of a new service, it will query the policy engine and find out which PAs are allowed to use this service. The SDA then informs these PAs about the new service so that they can now add this service to the list of services they have access to in the network. The same is done when a service becomes unavailable. By doing this we ensure that all users will have access to all the network services they are authorized to use at any time. No unnecessary querying needs to be done by the PAs.

It is clear that this basic architecture complies with our requirements. First it deals with the issue of keeping track of the most current service information at all times. Secondly it frees the user side from having to search for services in the network since they should already have the complete list. We can see that this architecture is based on the *push* mode in which services are actively discovered and pushed to the user. It also goes further than just pushing or advertising since it also distributes the services to the appropriate users according to their credentials.

4.3 Design Enhancements

Having thoroughly laid out our initial design, we then follow to examine it and identify places for improvements and additions. There are a few areas where more functionalities can be added and the structure can be better organized.

4.3.1 Service Search

Even though our basic design frees the users from having to search for network services, it overlooks the case where a user might need a service that is not currently available in the network. In this case we would like to have our discovery mechanism attempt to search outside the network if it is possible for such a service to be used remotely. Note that we are now in fact adding the *pull* mode of Service Discovery in which the user will be initiating a service search when needed.

To avoid burdening the SDA with more tasks, we create a separate agent called Search Agent to carry out the duty of searching for external services. Another reason for

creating a new agent is that searching is an independent task. We can actually think of the Search Agent as a special type of SA (Service Agent) that performs a service for other entities. Users shall have a search element in their graphical user interfaces (GUI) that allows for manual service discovery. The search request will come to the Search Agent who then performs a search outside the network for the service. Ideally it would search the Internet but that is not a feasible option since the Internet is vast and made up of countless sub-networks that use different protocols. We therefore must find another suitable way, which will be described in a moment.

4.3.2 SDA Breakdown

Another aspect to improve is the structure of the basic architecture. It is fairly simple with only three components. However, in order to make our design more extensible so that it can accommodate future revisions and enhancements, it is beneficial to break down the SDA into smaller and modular agents that perform different sub-functions. The change will make the architecture more organized and easier to maintain and update. This is not a difficult task since we already have the separate Search Agent to begin with. The main SDA is still present but the function that maintains the service list is moved into a new agent called Description Agent. We now have a much better architecture in which the functions of the SDA component are clearer and easy to see. The SDA acts as the commanding entity of the other agents who will carry out the actual discovery duties (See Fig 4.3).

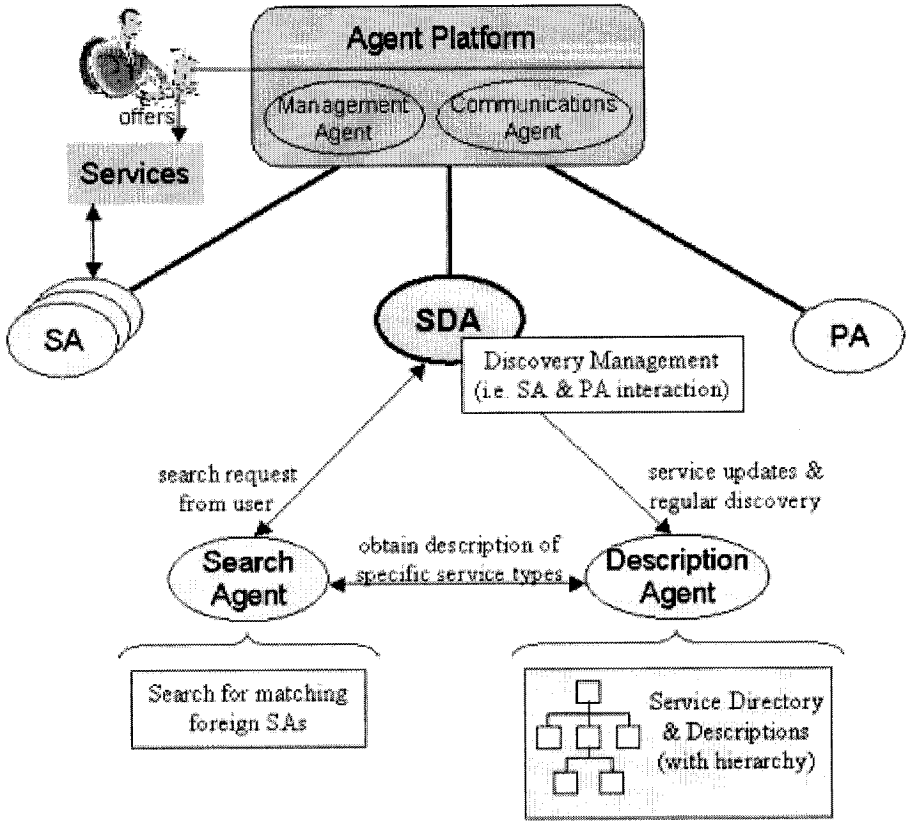


Fig 4.3: Modified SDA Architecture

Since now there is a dedicated agent for maintaining the service list, we have more flexibility to expand the properties of the list and add additional useful features to it. Instead of just storing all the currently available network services, the list can be expanded and modified into a formal service representation. All network service types, whether currently available in the network or not, can be recorded and their possible attributes formally presented. Such an extensive service representation will come in handy for any searching and matching tasks.

4.3.3 Service Search Revisited

We can now come back to our previous discussion of the search function. Since we are using Agent Technology, and for the purposes of our implementation, the Search Agent can perform the external search by looking to see if there is any foreign agent offering the required service. Foreign agents are mobile agents and are not originated in the current agent environment. They have traveled from other agent platforms into ours. Our discovery mechanism is agent-based and therefore can receive foreign agents as it is one of the social properties of agent environments such as FIPA-OS platform. Thus we see agent mobility as an important advantage in using agent technology.

The Search Agent obtains the service information the user has entered in the GUI. It then communicates this information with the Description Agent who can go into the service ontology and retrieve the possible service attributes of the looked-for service. Once the Service Agent gets these attributes it can use them to query the foreign agents and possibly find a suitable agent that offers the service. Therefore we see that the Description Agent not only maintains the list of currently available services, it also acts as a general service information repository for others who need such information. The architecture is now better than the basic design. It has a more organized structure and by using both the *push* and *pull* modes we increase the possibility that users will be able to find and thus access as many services as they require.

4.4 Inter-domain Service Discovery

There are many situations where a number of networks would share resources amongst each other. For instance, a company may have branch offices in different locations and thus would want to have these offices connected and certain resources shared. Thanks to Agent Technology, the concept of mobile agents will make this an easier task as we will see later. Thus far our design has been limited for single network Service Discovery. Therefore our next aim would be focused on scaling our discovery function from intra-domain to inter-domain.

The goal of inter-domain Service Discovery is to have each network share some of its resources with the other networks in the group of domains. A user would then be able to access authorized services either from his/her own network or from the other networks. Each network will have the single network discovery mechanism that we have proposed to perform Service Discovery within their own networks. Thus each network has its own SAs, Description Agent, Search Agent, and SDA. Modifications must be made to the mechanism to perform discovery in other networks. Since the SDA is the Service Discovery control entity for a network, it follows that in order to achieve multi-network Service Discovery, the SDAs from all the participating networks should communicate and negotiate with each other.

Communication between SDAs can take place whenever there is a service update (e.g. new service registration, service departure) in a network. In this case the SDA from this network will notify the SDAs in the other networks of this information. Fig 4.4 illustrates a scenario with three participating networks. Let us assume that there is a new

service in Network A. The SDA in A will receive a registration message from the SA that is created for the new service.

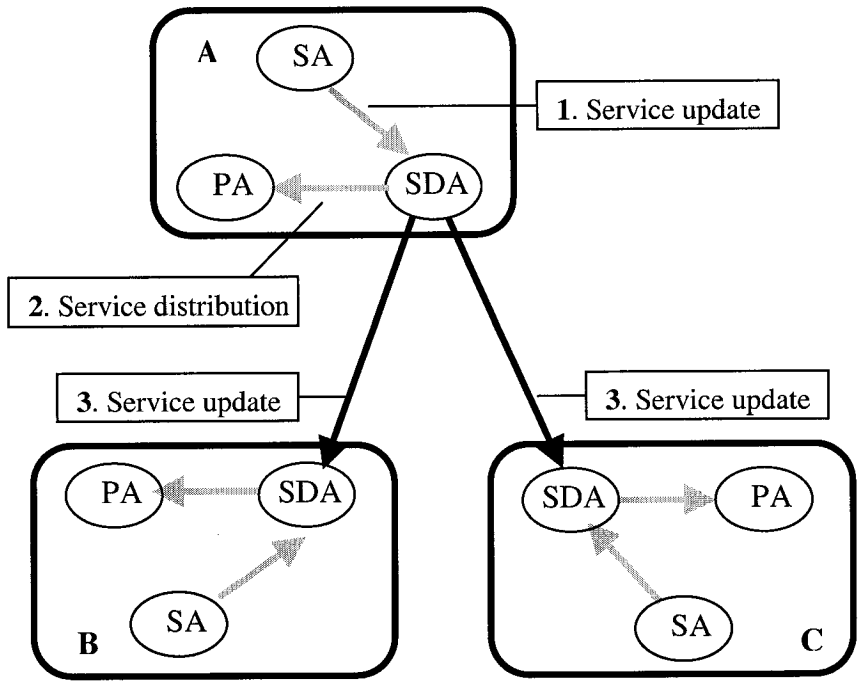


Fig 4.4: Inter-domain Service Discovery Scenario

In addition to recording this and informing the appropriate PAs, the SDA would now also notify the SDAs in networks B and C. It is up to each of those two SDAs to decide whether or not it wants this new service from Network A to be available to its own network. This decision is again based on policies pre-configured in the policy agent. For example, such a policy can be that a printer is not allowed to be used remotely from other networks. It is logical to have the source SDA notify the other SDAs of possible service sharing information and then leave the other SDAs to make their own decisions about what to do. For a network, only the SDA that belongs to this network is able to judge if an outside service is needed in this network.

Assuming that the SDA in Network B has made the decision to accept the new service, it will inform the SDA in A (where the new service was originated) of this decision prompting it to initiate the process of making the new service available to Network B. We prefer to create as little additional work on the user side (i.e. the PAs) as possible so that the location of the new external service remains transparent as if it is just another ordinary network service. Therefore we should create a proxy SA for the new service in Network B so that it will then follow the regular procedure to register with the local SDA.

We must keep in mind that the actual service still resides in Network A and will not be moved. However, to create a new SA we can simply clone the original SA in Network A and then move the cloned SA to Network B (agent mobility). There will be a link between the two SAs. Thus any service request made in Network B to the cloned SA will then be forwarded to the original SA in Network A. This describes our proposal as to how inter-domain Service Discovery is done.

4.5 Summary

In the centralized Service Discovery model for the ad-hoc conference room environment, agents are used to act on behalf of the different entities in the architecture. Services are represented by SAs and users are represented by PAs. The central component that controls discovery actions is headed by the SDA with the help of the search agent and the description agent to maintain the service repository. This centralized design is characterized by freeing the users from performing discovery for network services. Since users will always have access to all current active services they

are authorized to use, they need not concern with searching for services unless it is not available in the network. Inter-domain discovery is also covered in this design where SDAs from each conference room communicate to deliver service information for the users. Here we take advantage of agent mobility and cloning to conveniently achieve inter-domain service sharing.

Chapter 5

Distributed Service Discovery Design

In this Chapter we present our distributed approach of agent-based Service Discovery for the conference room environment. We consider the case where we have a fully ad-hoc environment. This Chapter holds the remaining part of the central theme that represents this Thesis. The goals that the distributed model aims to achieve are presented first. We will next look at a peer-to-peer search algorithm that is adopted for our design. The strength and weakness of this algorithm will be discussed and how it is enhanced and adapted to our distributed discovery design will then be explained. The details of the design architecture, the operations, and the role of each component are therefore interlaced within the discussion of the search algorithm.

5.1 System Goals and Overview

For the centralized design, it is made clear that the conference room will have a fixed network portion, thus making it capable to support a centralized discovery model. However, while this model is well suited for our target environment, it may not be a good solution for other network structures especially for fully ad-hoc environments. The next step in our Service Discovery research is therefore to examine a distributed discovery approach for the conference room setting, assuming it is an entirely ad-hoc setting. Such an approach would indeed be a universal approach that may function in any type of network structure.

The goals of the distributed discovery system are similar to those of the centralized system. Essentially there should be a sense of network service awareness. The entrance and exit of any service should be tracked to maintain correct service information. This is now especially important in a fully ad-hoc environment. And as before, users should be spared of performing discovery as much as possible. In a conference environment, the users would do better to concentrate more on the conference at hand than on looking for services.

In our distributed design, agents are placed in each device to carry out discovery on behalf of the device user. Discovery is initiated on demand. However, with the use of an enhanced peer-to-peer search algorithm, a discovery session will also push the discovered service to those clients who participate in propagating the search. Therefore the chance of users needing to look for services is reduced. We now examine our design in detail.

5.2 Peer-to-peer Search

Since the target environment is now fully ad-hoc, it is only logical to shift the discovery design into a distributed model. As we have seen in some existing distributed protocols (e.g. UPnP), service announcements are broadcast to the entire network. It reduces discovery delay and creates less work for users, something our design aims to achieve. Although this is suitable for our case since we are dealing with a relatively small network with fewer services, it is still better to use a more efficient method and reduce the amount of broadcasting. Also, now that all the devices are mobile, we must keep in mind that they may all have limited power and capacity. Therefore pushing all network services to each device may overload it.

The *pull* operation mode, on the other hand, will only acquire needed services, but it introduces more delay and work for the users. Thus we need to find a compromise between distributed push and distributed pull. In this Section we discuss an existing search algorithm that meets our requirements and that we will be enhancing for our design.

5.2.1 Peer-to-peer Setup

In our research we discovered a peer-to-peer (P2P) search algorithm proposed by Menascé [43]. The algorithm presents an efficient method of looking for resources in a network. As will be seen, a careful combination of both *push* and *pull* is used to achieve progressively simplified searches for users. However this algorithm assumes a fixed network topology and thus must be modified and enhanced for a dynamic environment

such as in the case we have. Its general search principles though, serve as a basis for our distributed discovery model.

A P2P system has a distributed topology containing what are called peer nodes. Network operations and computations rely on individual nodes' capabilities. Each peer node can act as both server and client. In this sense the nodes are all equal in a P2P network. Ad-hoc networks have the P2P characteristics, although the nodes are mobile and unstable.

In Menascé's P2P search algorithm, a network is assumed to be fully connected with an arbitrary topology.

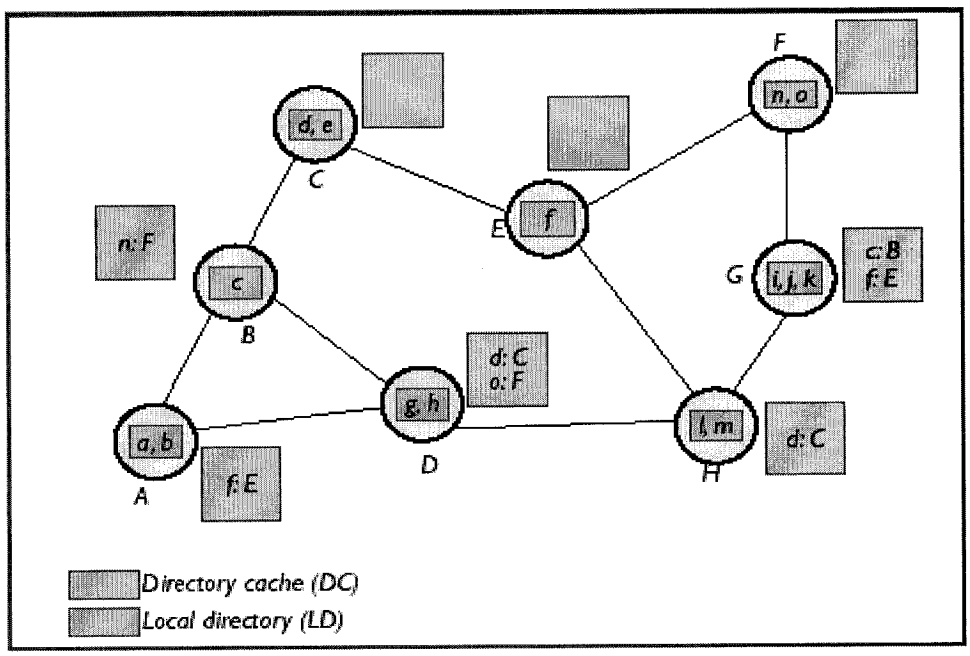


Fig 5.1: P2P Computing System [43]

Fig 5.1 illustrates an example of the P2P system. Each node is directly connected to a number of other nodes which are the neighbours of that node. Also there are two

resource directories kept by each node. One is the local directory (LD) that stores pointers to the resources offered locally by this node. The other is the directory cache (DC) which contains links to all the presumed foreign resources to the node's knowledge. For example, in Fig 5.1, node A's LD indicates that it has resources *a* and *b* in its possession, and its DC shows that it knows that resource *f* is located at node E. Neither of the two directories needs to be populated though, as we can see in the diagram that nodes C, E and F all have empty DCs, meaning they do not know of any foreign resources.

LDs are always under the complete control of their perspective nodes. Any changes to the local resources are not actively made known to other nodes in the network. Therefore the resource links in the DC does not guarantee the existence of the foreign resources.

5.2.2 Search Operations

The P2P search algorithm aims to be an efficient algorithm by reducing the amount of nodes involved in a search. If all nodes are contacted during a search then the chance of finding the resource is maximized, but it comes at the high cost of bandwidth if we have a large network. To achieve efficiency, Menascé proposes a probabilistic search method. Instead of involving all nodes, only a portion of the nodes would be contacted using probability. Furthermore, each resource discovery would benefit not only the initiating node, but also all the nodes involved in the search, therefore making the search process even more efficient and less necessary over time.

The search algorithm involves two stages which translates into the two types of messages a node can receive: SearchRequest and ResourceFound. We look at both of these actions in detail.

SearchRequest

The SearchRequest message is initiated by a node in need of a resource. This node is called the source node. The message contains four pieces of information.

- *Source* - This is the location of the source node. It is used to reply back to the source node of the search result.
- *Resource* - This is a description of the required resource.
- *Reverse Path* - This is a list of node locations that reflects the path this SearchRequest message has traversed so far.
- *Time to Live (TTL)* - This is an integer value that will be decremented every time the SearchRequest message advances to another node. It is used to stop the search after a number of unsuccessful hops.

The SearchRequest message is generated at the source node and it propagates through the P2P network in search of the required resource. When a node receives a SearchRequest message, the first thing it does is to check its LD to see if the required resource can be found locally on this node. If yes, then this search path has come to an end and the node prepares to send back a ResourceFound reply (discussed in the next Section).

If however, the node does not have the resource locally, it will then look into its DC to see if it knows of any other node who might have the resource. Note that the information in the DC may not be correct as explained previously. A node's resources can change without notifying other nodes. Therefore, if the resource is found in the DC, it does not mean the foreign node is guaranteed to still have this resource. Thus, instead of directly sending a ResourceFound reply back, the current node will forward the SearchRequest to the node that presumably has the service. And when this node receives this SearchRquest message, it will behave as usual when such a message is received.

If, after querying both the LD and the DC, the resource is still not found, the node will then check the value of TTL in the SearchRequest message. A TTL value of 0 means this is the last node in the search path and the process ends here. Otherwise TTL is decremented by 1 and the node adds its own address to the Reverse Path of this message. It then continues the search by performing a probabilistic forward that is characteristic of this search algorithm.

For each of its neighbours, the node will apply a pre-determined probability to decide whether or not to forward the SearchRequest to that neighbour. This probability is a fixed value for all the nodes. The logic of using this type of probabilistic forward is to reduce the number of nodes contacted in a search while still maintaining a good chance of finding the resource. Fig 5.2 shows an example of a SearchRequest propagation.

A SearchRequest (SR) for resource *n* first originates at node C. Node C does not have *n* in its LD and its DC is empty. Therefore it applies the probability on both its neighbours B and E. Only node B passes the probability, thus the SR is forwarded to node B with TTL decremented and Reverse Path updated. After receiving the SR, node

B proceeds to check its LD for resource *n*. It is not offered locally thus the DC is then queried. At this point node B finds that it knows resource *n* is presumably offered by node F. Therefore node B forwards the SR to node F with TTL and Reverse Path properly updated. Finally node F confirms that it indeed still offers resource *n* by checking its LD, and the search ends successfully. If, however, node F does not host the resource anymore, then it will continue the search as normal.

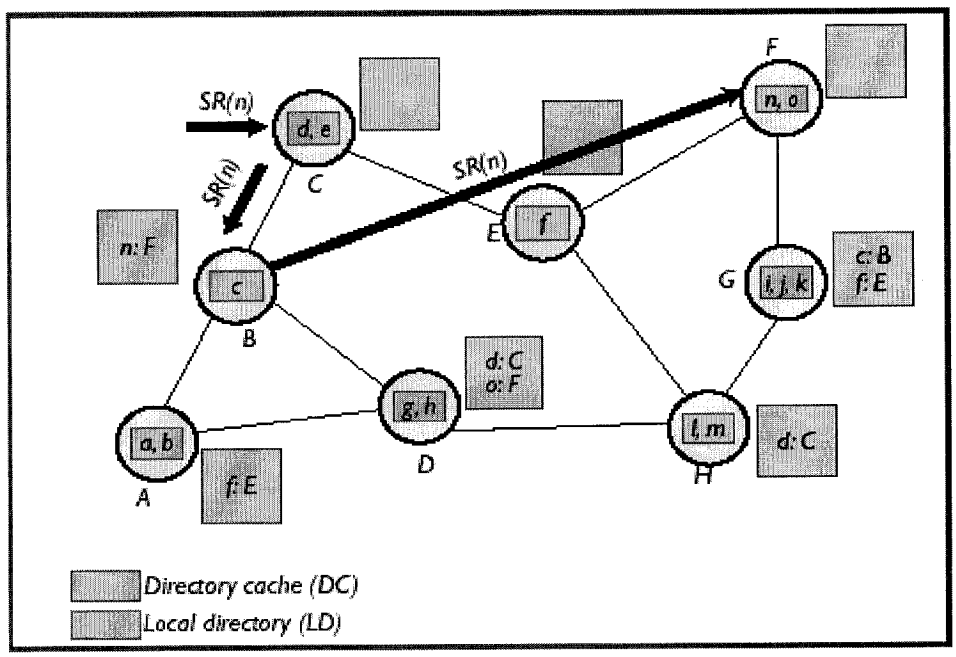


Fig 5.2: SearchRequest Example [43]

We should keep in mind that when performing a probabilistic forward, it is often the case that the node will forward the SR to more than one of its neighbours. This will create several search paths which may yield a number of matching results for the source node. Of course this also means portions of some paths may be traversed repeatedly,

although there is an equal chance of traversing a wider variety of paths. Note that with the restriction of TTL, all search paths would end eventually.

ResourceFound

When a resource is found locally in a node, the search process ends and the reply propagation starts. The node that offers the resource will initiate a ResourceFound message which is sent back along the path that the SearchRequest has traversed. The ResourceFound message also contains four pieces of data.

- *Source* - Once again this is the source node that had originally requested the resource.
- *Resource* - This is the description of the found resource.
- *Reverse Path* - This is the list of nodes along the original search path. This list is used to forward the ResourceFound message back to the source node.
- *Resource Node* - Finally this is the location of the node that offers the resource.

When a node receives a ResourceFound message, it first checks if itself is the source node that initiated the search for the found resource. If yes, then it will directly contact the resource node with a request to access the resource. If not, then the node looks at the next node in the Reverse Path and removes it from the Reverse Path. It then forwards the ResourceFound message to that node.

In both cases, the node will store the information regarding the found resource if it is not already known. The resource information is therefore pushed to nodes who do not necessarily need it. We can see that although pulling is used on the surface, it initiates push operations and thus creating a nice view of the two modes cooperating together. Fig 5.3 demonstrates an example of a ResourceFound propagation. It is a continuation of the SearchRequest example shown in Fig 5.2.

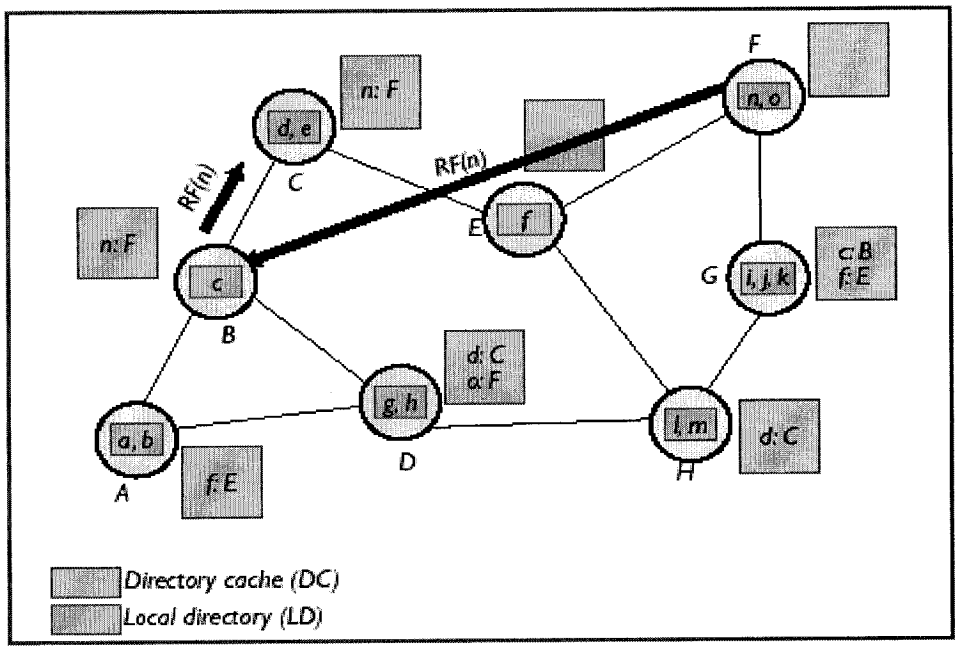


Fig 5.3: ResourceFound Example [43]

Node F is the resource node that contains the sought-after resource *n*. To prepare for its reply, node F first looks at the Reverse Path included in the SR message it has received. The last node in the path is node B. Node F then removes node B from the Reverse Path and puts the updated path along with the other aforementioned pieces of information into a new ResourceFound (RF) message. This RF message is then sent to node B.

Upon receiving the RF message, node B checks to see if itself is the source node. This is done by looking at the source that is included in the RF message. Node B is not the source and thus it again removes the last node in the Reverse Path which in this case is node C. The RF is then sent to node C. Node B does not need to update its DC since it already has the current information that node F contains resource *n*. However, if any node on the path does not have this information already then its DC will be updated.

The RF finally arrives at node C who is the source node. It stores the information regarding resource *n* in its DC and will proceed to contact node F directly to access *n*. This concludes the P2P search process.

5.2.3 P2P Search in Service Discovery

Menascé had tested the search algorithm with different values for the SearchRequest forward probability. He found that in a network with 120 nodes, the algorithm reaches an optimum result when the probability is at 0.6 or 60%. At this probability value, the chance of finding a resource is 90% while only 10.5% of the total number of nodes are contacted during the search. But even as the forward probability value reaches 1, only around 50% of the nodes are contacted. This is due to the use of DCs to direct searches toward the correct node. The search algorithm therefore proves to be an extremely efficient algorithm.

The important thing of these results is that they show with the use of this algorithm, it *is* possible to find a resource by only visiting a small portion of the nodes. While searching may take longer at the beginning, it will become increasingly efficient as more searches are carried out. This is because nodes update and acquire more and more

resource information in their DCs and are therefore more capable of locating the correct resource nodes.

Our distributed Service Discovery design will certainly benefit from this kind of efficiency. Also, by using *pull* as the initiation method of discovery, the devices in our network will not become suddenly flooded with service information they do not necessarily need. Even though services will still be pushed to the devices during discovery, it is done with an orderly fashion which is more manageable in small devices. Also note that a device's knowledge of foreign services would now be useful when performing discovery even if the device itself does not need these foreign services. There are of course a number of shortcomings of this search algorithm that especially concerns our discovery design. These will be examined as we discuss how this algorithm is modified and enhanced for our distributed discovery model.

5.3 Distributed Agent Architecture

Before delving into the detailed operations of the distributed design, we first look at the structure for our model. Recall that in the centralized model, there is one SDA for the conference room and it is supported by the Description Agent and the Search Agent. Since it is a centralized design, the SDA is the discovery control entity for the entire room and the Description Agent maintains the list of all the services in the room. Now that the model is assuming a distributed design, it means each device is responsible for its own service discovery. The unit of discovery is now the device instead of the room. Therefore we can duplicate the centralized agent structure into each device. Fig 5.4 shows an overview of this new distributed agent structure.

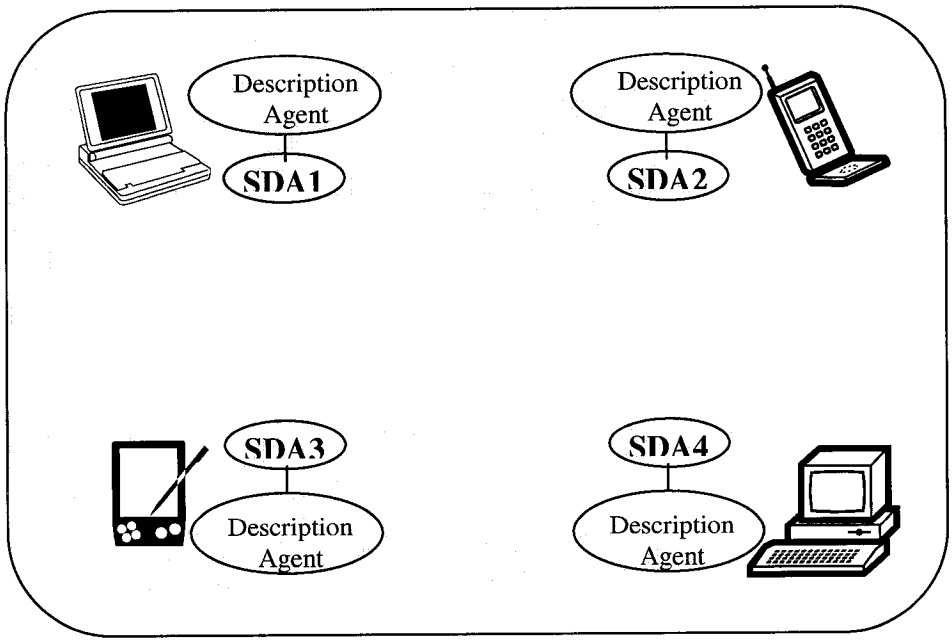


Fig 5.4: Distributed Discovery Architecture Overview

Each device in the room will have the same set of agents to perform discovery for it. The SDA and Description Agent are now present on every device. The SDA is the main agent in control of the discovery actions for the device. It is responsible for initiating searches when requested by the device user. It also receives and processes the different types of discovery messages from other devices (similar to SearchRequest and ResourceFound) among other things. Fig 5.5 gives a clearer view of the agent structure within a device.

The user interface (UI) allows the user to view and access available services both local and foreign (those that have been discovered). It also provides the option to search for services. The Description Agent maintains not one, but two service directories in accordance to the P2P search algorithm. These are once again the local directory (LD) and the directory cache (DC).

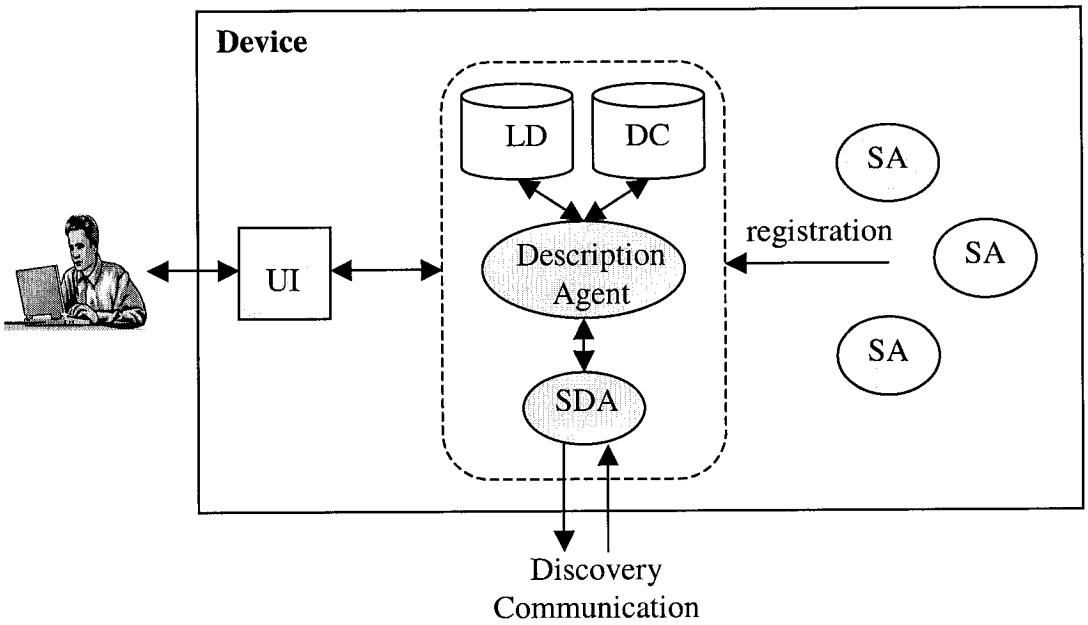


Fig 5.5: Agent Structure Within Device

From Fig 5.5, we can also see that each service offered by the device is represented by its own service agent (SA). This is the same as in the centralized model. The service directories would contain links to the SAs rather than the actual services. The SAs act as interfaces to their services. All discovery communications between the nodes are carried out by the SDAs. With the agent structure set up, we can now discuss the details of the discovery operations and how the P2P search algorithm is adapted for our model.

5.4 Discovery Operations

In this Section we introduce the detailed discovery procedures and rationale of the distributed design. In doing so we will examine the problem areas of the P2P search

algorithm and how they are improved for our model. Certain essential aspects are also added to complete the design to our purpose.

5.4.1 Presence Awareness

In the P2P search algorithm, it is assumed that a node is already aware of its neighbours and knows their addresses. This information is static and does not change (unless manually) since the target network for the algorithm is a fixed network. Therefore nodes do not have to concern with finding out where other nodes are located. This is a luxury that our discovery work does not have.

Because it is an ad-hoc environment, the discovery model must provide constant device presence awareness in addition to actual discovery. Otherwise devices (i.e. nodes) would not have the correct information to be able to forward search and reply messages. Since the model is distributed, the easiest way to achieve awareness is through distributed announcements. Devices should make themselves known to the other devices in the room.

In our distributed model, a device's SDA will send a presence notification to the room when the device enters the network. This presence notification contains the device identifier (type, name, etc.), its address and a lease time. Thus we adopt the leasing concept and require the devices to continue resend presence notifications after a pre-specified period of time.

When a device's SDA receives a presence message from another device, it logs this information for this specific device, and starts a counter with the expiration time set to the lease time specified in the presence notification plus a short period of cushion time.

If another presence notification from the same device is received before the counter expires, then the counter is reset and starts over from the beginning. If however, the counter expires and still no presence notification is received from the same device, then this device is assumed to have left the network. The SDA proceeds to forward the device expiration information to the Description Agent. The Description Agent will check the DC and remove all service links that belong to the expired device, therefore maintaining the correctness of the service information in an as timely fashion as possible. Fig 5.6 summarizes the algorithm for the presence awareness process of our discovery model. The algorithm illustrates the actions of one SDA upon occurrence of various events.

```
PN = Presence Notification; EN = Exit Notification

Event = Parent device launched (i.e. now active in the network)
    Send PN(device ID, address, lease time) to room;
    Set lease timer for parent device

Event = Lease timer of parent device expires
    Send PN(device ID, address, lease time) to room;
    Reset lease timer for parent device

Event = Parent device shutting down
    Send EN(device ID) to room
```

```
Event = Received PN from Device d
    If d exists in device list
        Reset lease timer for d
    Else
        Add d in device list;
        Set lease timer for d

Event = Lease timer of Device d expires
    Remove d from device list;
    Notify Description Agent to remove all services offered by
    d from the Directory Cache

Event = Received EN from Device d
    Remove d from device list;
    Notify Description Agent to remove all services offered by
    d from the Directory Cache
```

Fig 5.6: Presence Awareness Algorithm

The leasing process is to eliminate incorrect service information when a device unexpectedly becomes unavailable. However, devices are assumed to properly carry out shutdown or exit procedures before leaving the network. This will cause the device SDA to make an exit announcement to the network. Upon reception of such a message, an SDA will inform the Description Agent to update the DC as described above.

Recall that in the P2P search algorithm, DCs can only be updated by adding new resource entries received in ResourceFound messages. There is no mechanism to remove service entries when they are no longer available. Therefore the presence awareness process takes care of this essential feature as required in our ad-hoc environment.

Note that only presence information is announced in the awareness process. Service information is not piggybacked onto the presence notifications. As stated before, we do not wish to flood devices with all the network services. Some of the most common mobile devices (e.g. laptops) can contain much more services than others. Smaller devices may be incapable of storing a large amount of information, many of which they will not need. We therefore decide to adapt the P2P search algorithm as the basis for our discovery mechanism where service pushing is reduced and done in a more orderly manner.

5.4.2 Search Operations

The presence awareness mechanism allows devices to have a constant current view of other devices in the network. It also serves as a major feature that helps to adjust the P2P search algorithm for our dynamic environment. Although the main search operations are still based on this algorithm, but with presence awareness running in the

background, the operations automatically become adjusted to function properly in the ad-hoc network. We now discuss the search operations to examine these adjustments as well as other enhancements.

5.4.2.1 Search

When a service is required on a device, the SDA of that device initiates a search by creating a request message. We will still call this message SearchRequest (SR) as in the P2P search algorithm. The message has the same set of information, namely source device, service requested, reverse path, and TTL. The SR will then be propagated through the network in search of the service. The actions a device takes upon receiving an SR are generally the same as those specified in the P2P search algorithm except for a few modifications.

When a device's SDA receives an SR, it uses the Description Agent to check the LD and DC as described earlier for the P2P search algorithm. If the service is offered locally, the address of the matching SA will be sent back in the reply message (discussed in the next Section). If the service is in turn found in the DC, then the SR will be forwarded to the appropriate device. All of this is the same as in the P2P search algorithm. However the action becomes a little different when no matching service is found in either directory and the SR must be forwarded probabilistically.

In the original algorithm, such a forward is based solely on probability. An SR can potentially be forwarded to a node that has already been visited on the current path. This wastes precious time and bandwidth, plus it increases the possibility of the message going in useless loops before the process finally terminates when the end of TTL is

reached. We therefore modify the probabilistic forward by adding another condition before an SR can be forwarded. The potential destination node must not be already in the reverse path. If this condition is satisfied, then the SDA can apply the probability on the node to see if the SR will be forwarded to it.

SR messages are probabilistically forwarded to just a node's neighbours in the original algorithm. We on the other hand, have a fully ad-hoc environment where users can move around with their mobile devices. This added to the fact that it is a small network within a room makes the notion of neighbours less significant and more difficult to track. Therefore, we look at the entire room network as a single neighbourhood and every device is a candidate for receiving a probabilistic forward from any other device. In this case, if a device needs to perform probabilistic forward but finds that there is no qualified device to forward to, then it usually means the SR has traversed the entire network without success and the search will terminate.

Since the presence awareness process is constantly running in the background, the devices in our model always have a current view of the other devices in the network. This automatically makes the search process adapt to our dynamic environment and increases the possibility of finding a service. As an SR message is traversing through the network, the network topology can change at any time. Therefore, it is not feasible to have a freeze view of the network throughout the entire search propagation such as the situation the original algorithm assumes. Our model ensures that searches will be carried out according to the dynamic view of the network.

After a device's Description Agent searches unsuccessfully in the LD for the requested service specified in the SR message received, the next step would be to check

the DC. At this point, there is the danger of finding a matching service in the DC whose parent device does not exist anymore. However, with the presence awareness running, the chances of this happening is greatly reduced. The absence of a device is either announced or eventually detected, and this information is immediately made known to all devices who can then update their DCs.

Detections and updates happen dynamically as changes occur. Therefore as an SR message travels from device to device, it is always processed according to the latest view of the devices in the network. This latest view does not only concern the DCs of devices, but it also impacts probabilistic forward. A device's SDA may have a current view that is different then the one looked at by a previous device in the SR message's path. This current view may show new devices that just entered the network and it may be missing previous devices that are not available anymore. Thus this probabilistic forward will include the new devices as forward candidates and reduce the probability of forwarding to non-existent devices. Therefore, we can see that as a search goes along its path, it always adjusts to the latest network view and takes full advantage of dynamically recruiting new devices in the search. Fig 5.7 illustrates an SR propagation scenario.

In the scenario SDA1 initiates the search. A matching service cannot be found in its DC, therefore it forwards an SR probabilistically to SDA2. SDA1 is aware of SDA3 and 4 but in this case these two devices does not pass the probability. Thus the SR is only sent to SDA2. SDA2 does not have the required service in either directories. Therefore it also performs probabilistic forward. It is aware of the other three devices but excludes SDA1 as a forward candidate since SDA1 is the originator of the SR message. In the scenario the SR is probabilistically forwarded to SDA3.

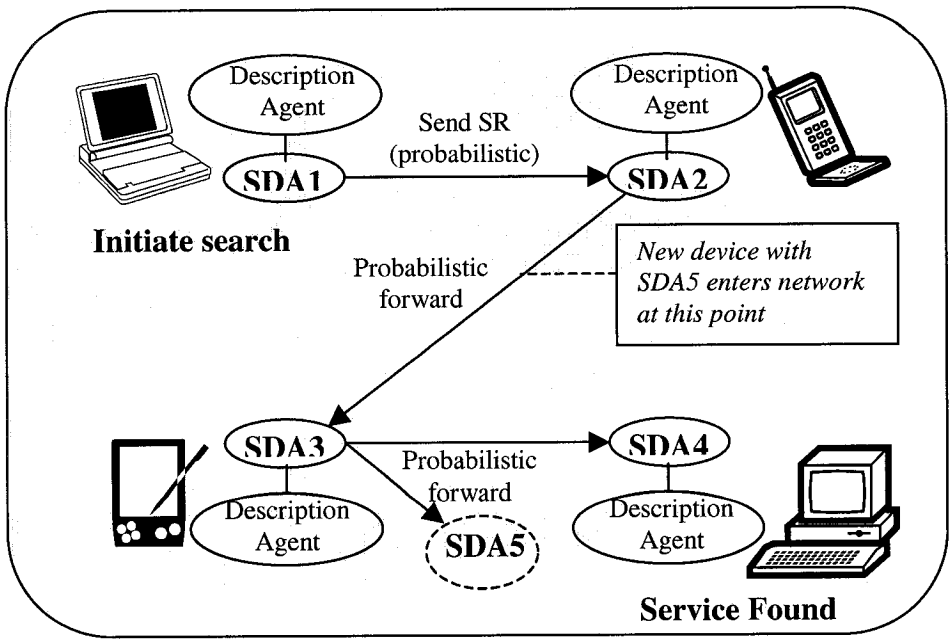


Fig 5.7: SearchRequest (SR) Propagation Scenario

At this point, a new device (with SDA5) enters the network. SDA5 notifies the other devices of its presence as described in the presence awareness section. When SDA3 is ready to perform probabilistic forward (i.e. it does not find the required service either), it will have SDA5 in its current network view. The SR is then probabilistically forwarded to both SDA5 and SDA4 in this scenario, thus making use of the new device that was not available earlier in the search. Note that the search path splits in two here. The one that goes to SDA4 finds the service locally in SDA4 while the other path to SDA5 will continue on as normal.

5.4.2.2 Search Reply

When a service is found, a reply message will be sent back. This reply message also contains the same pieces of information as specified for the ResourceFound message

in the P2P algorithm. These include the source device, the device that contains the service, a service description that identifies the required service for this search, and finally the Reverse Path the reply will be sent to.

In the original P2P search algorithm, when a resource is found, the reply is sent back along the same path the SR had traversed. The purpose of this is to update those nodes on the path of the resource information. This approach may be fine in a fixed network, but it is not suited for our mobile ad-hoc environment.

First of all, search delay can be reduced if the reply is sent directly from the device offering the service to the source device. Receiving a reply that comes from traversing the reverse path creates more waiting time for the source device. Therefore we modify the reply procedure to have the reply sent directly to the source device.

Of course the devices in the Reverse Path cannot be ignored, otherwise we lose the all-important *push* aspect of the algorithm which is the key to achieving progressive search efficiency. Thus in addition to informing the source directly, the reply will also be sent to those nodes in the Reverse Path. However, as stated earlier, having the reply traversing the Reverse Path is not suitable for an ad-hoc environment. We must keep in mind that devices can enter and exit the network at any time. Therefore there is no guarantee that a device in the path would still be present when the reply message is sent. If the reply is sent by traversing the Reverse Path, then when the message comes to a non-existent device, the reply propagation will be broken. This is also another reason for sending a reply directly to the source first.

Our solution is to have the reply sent directly from the device that offers the service to each node in the Reverse Path. Before doing this, the SDA of this device will

check the current view of the network so that it will only send the reply to those devices that are still present. This will eliminate the problem of broken Reverse Paths and it also delivers the service information to the devices fasters. This increased reply speed can mean the difference between failure and success for other searches going on in the network. Fig 5.8 shows the reply propagation scenario that is a continuation of the search propagation illustrated in Fig 5.7.

In the scenario, the device with SDA4 is the one that contains the service locally. It immediately sends a reply to SDA1 which is the source device that requested the service. SDA4 then proceeds to send replies to the devices in the Reverse Path. After consulting the current network view, it finds out that the device with SDA2 is no longer available. Thus only SDA3 is sent the reply. SDA3, upon receiving the reply, informs its Description Agent to update the DC with the new service information. Note that SDA5 was not involved in this particular search path (see Fig 5.7). Its address is not in the Reverse Path and therefore the reply message does not affect it. It will receive replies from the search paths it participated in.

Our search reply process is more robust in that the reply messages are not dependent on any devices in the Reverse Path. This complies with the nature of our ad-hoc environment. Also, even if the source device becomes unavailable before it receives any reply, the search is not wasted. Other devices in the Reverse Path will still receive replies and be able to update their DCs for use in other searches.

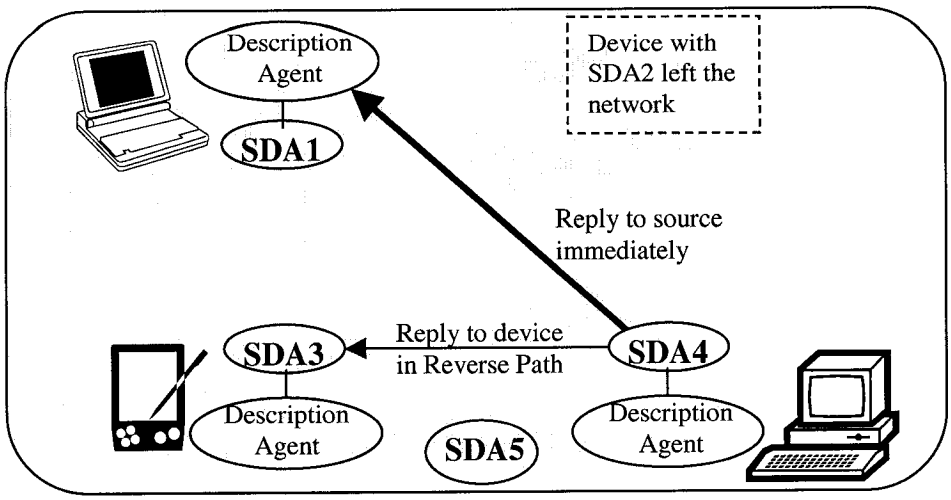


Fig 5.8: Search Reply Propagation Scenario

Because of the use of probabilistic forward, a search can be split into a number of search paths along the way. This means the source device, or any device that performed probabilistic forward during the search, can receive more than one replies. The same type of service is offered by several devices. In this case all the service information received are stored in the DC. It increases the possibility of finding a service.

We have discussed search reply in the situation when a service is found. To cover for the chance that a matching service cannot be found, the SDA shall wait for a pre-specified period of time after initiating discovery. After this period expires with no reply received, the SDA will notify the user. The failure of finding a service would suggest that most likely no matching service is available. Such a failure can occur under several circumstances. One is that the TTL value in the SR message becomes zero and the service is still not found. The other situation is when a probabilistic forward cannot proceed due to the lack of qualified forward candidates (i.e. all devices have been contacted without success). Fig 5.9 outlines the algorithm that summarizes the discovery

actions of our distributed model. The algorithm shows the actions of one SDA upon occurrence of various events. This combined with the presence awareness algorithm (Fig 5.6) completes the discovery operations.

```
SR = Search Request; SF = Service Found reply

Event = Received service request from user (i.e. via GUI)
    For each SDA in the current network
        Apply random probability prob;
        If prob is within specified Forward Probability P
            Send SR(source ID, service type, reverse path, TTL)

Event = Received SR from Device d
    Notify Description Agent to check for service requested;
    If service found in Local Directory
        Send SF(matching SA ID) to SDA of source device;
        For each SDA in reverse path
            Send SF(SA ID, service type)
    Else if service found in Directory Cache
        Add own SDA ID to reverse path;
        Decrement TTL by 1;
        Forward SR to SDA of the device with the matching service
    Else if TTL > 0
        Add own SDA ID to reverse path;
        Decrement TTL by 1;
        For each SDA in the network
            If the SDA is not in reverse path
                Apply random probability prob;
                If prob is within Forward Probability P
                    Forward SR

Event = Received SF from Device d
    If itself is the source of the search request
        If no previous reply was received for this request
            Contact the matching SA to access the service;
        Notify Description Agent to add service in Directory Cache
    Else
        Notify Description Agent to add service in Directory Cache

Event = Search for service type st time out
    Notify user of parent device that service of type st not found
```

Fig 5.9: Discovery Algorithm

5.5 Service Access

Now that discovery, the core part of our Service Discovery model, is done, the next step is to actually use the service. Recall that a source device can receive multiple service replies. While the SDA will inform the Description Agent to store all these services in the DC, only the first one received will be used for simplicity. It may happen that this service's parent device has left the network and the access will thus fail. In this case the SDA will look to the other replies received to attempt access again. Thus we see one of the benefits of having multiple search replies.

Services come in different types and varieties. Each one requires a unique access method. We assume users do not know how to access a service beforehand. Thus a universal way is needed for service access.

Each service on a device is represented by an SA. As stated earlier, SAs act as interfaces to the services. Thus we decide that is exactly how users are going to access remote services: through UIs offered by SAs. Since services are different, SAs are also different. Each is implemented to suit its particular service. Some services may already have an attached UI, others may not. In any case, the SA will provide a UI for users to invocate the service. The UI must be able to communicate with the actual service remotely to make access possible.

We have seen in our centralized model the advantages of using agent mobility. Here we use it again for service access. One of the other agent characteristics is that agents can encapsulate and carry information with them. This means they can also carry executable code. Here mobile agents are used to carry UIs from service providers to users' devices. Fig 5.10 shows the service access process.

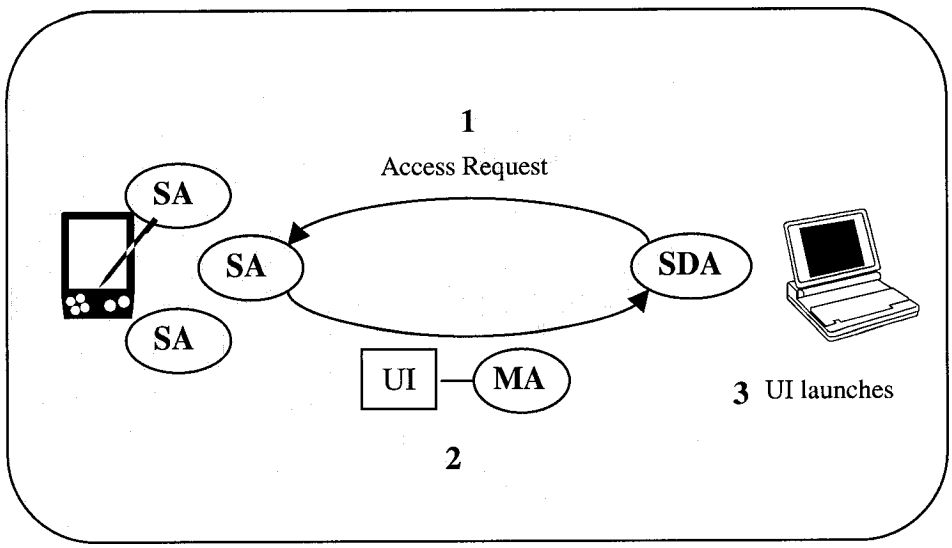


Fig 5.10: Service Access

When a service is found, the address of the SA is made known to the user device's SDA. The SDA will send a request directly to the SA asking to access the service. This request will include the identity and location of the user device. After receiving the access request, the SA dispatches a mobile agent (MA) to carry the service UI to the user device. Finally the UI will be launched when it arrives at the destination. The user can now access the service through the UI.

Each service has its own set of access requirements. Its SA must know how to properly deliver the service when it is requested. Some services may not be possible to access remotely. In this case the service itself can be copied and transported to the user. We will present an example of this in the next Chapter where we discuss the implementation details.

5.6 Summary

Our target environment has changed from a hybrid network to a fully ad-hoc network. This prompts us to develop a distributed Service Discovery model. In order to build a discovery model that creates less work for the users as well as being efficient, a compromise between *push* and *pull* must be reached. We use the P2P search algorithm proposed by Menascé to achieve this. The algorithm uses a probabilistic search approach to reduce the amount of communications needed for searches. User initiated searches are required, but is supplemented by organized pushing of discovered service information to nodes that had participated in the search. This allows searches to become progressively quicker and less necessary, therefore reducing the work for users. Experiments indicate that only a small fraction of the nodes needs be contacted in order to find a resource.

This algorithm is designed for fixed network topologies. We adapt it to our ad-hoc environment by making necessary modifications and enhancements. Among these a presence awareness layer is added to ensure all devices are dynamically aware of other devices in the room. Certain aspects of the discovery operations are also enhanced to accommodate the dynamic nature of our environment. Service access, which is not addressed in the original algorithm, is then achieved through agent mobility and code encapsulation. The resulting distributed discovery model is one that enables mobile devices to be aware of and adjust to network changes. It also achieves the goal of reducing user work while increasing discovery efficiency.

Chapter 6

Implementation and Results

In this Chapter we discuss the implementation details of both the centralized discovery model and the distributed model. We will first introduce the main aspects of the FIPA-OS agent platform that are essential for our implementation. The overall ad-hoc communications project for which our discovery model is a part of will also be briefly introduced. We look at the project's main components and how they cooperate as a whole. We then discuss the implementation of our discovery models. Each agent involved in the architectures will be looked at. Although service agents (SAs) are implemented differently for each service, we will present one SA we built as an example of how such an agent functions. Detailed discovery interactions between the different agents will be examined. Certain scenarios used to test the implemented models are then outlined and the results are reviewed.

6.1 Agent Platform and Environment

Since agents can be social and mobile entities, a standard is needed to specify the detailed structures of agents, agent environments, and communication between agents. This would allow for the universal interoperability of agents from different platforms as long as they are all implemented according to the same standard. FIPA [31], as described in Chapter 2, is such a standard and is already widely adopted. Here we briefly introduce the agent platform structure that FIPA specifies as it is essential to understand for our implementation.

Within a FIPA agent platform, every component is an agent. To adapt to an agent platform, an application should have a wrapper agent to represent it and interact with other agents. Our search agents are in fact wrapper agents since they are created to represent services that are not agents. Within every FIPA platform, there is a set of special agents that perform management and communication duties. The first of these is the Agent Management System (AMS). As the name indicates, AMS is the agent that is in control of the agent platform. All agents in the platform must register with the AMS.

The next special agent is the Directory Facilitator (DF). The main function of DF is to provide the means for finding an agent. Therefore, if an agent wishes to make itself available to other agents, it would register with the DF in addition to registering with the AMS. The DF is not to be confused with our Description Agent. The Description Agent maintains registries of Service Agents only, while the DF makes no distinction between different types of agents and serves to offer any agent that may be accessed by others. The DF proves to be a very important and convenient agent that aids in our implementation.

Finally there is the Message Transfer System (MTS). MTS is responsible for all inter-agent and inter-platform communications. If an agent wishes to communicate with another agent either in the same or on a different agent platform, it must use the MTS on its own platform to send the message. Each device participating in a FIPA agent communication system must have its own FIPA platform installed and running. This is the case for our implementation.

Communications between agents also follow a specified format. All agent messages must be constructed based on the same standard structure called Agent Communication Language (ACL) so that any agent can understand a message sent to it by any other agent. ACL describes a set of message parameters such as type and content that can be defined to convey any type of message. For details on ACL please see [44].

As stated earlier, there are several FIPA compliant platform implementations that allow developers to create different agent environments. For our experiment we have chosen to use FIPA Open Source (FIPA-OS). It is the platform used for our ad-hoc communication project, and thus is also used for the Service Discovery model. FIPA-OS is a Java based toolkit that is most compliant with JDK 1.2. The operating system that hosts our project implementation is Microsoft Windows 2000 and the programming language used is Java.

6.2 Ad-hoc Communications Project

The Service Discovery model is but one component of an ad-hoc communications project we have developed. In this project, different functional components collaborate

together to offer an ad-hoc conference room environment. Fig 6.1 shows the architecture of the communications system and how each component fits in.

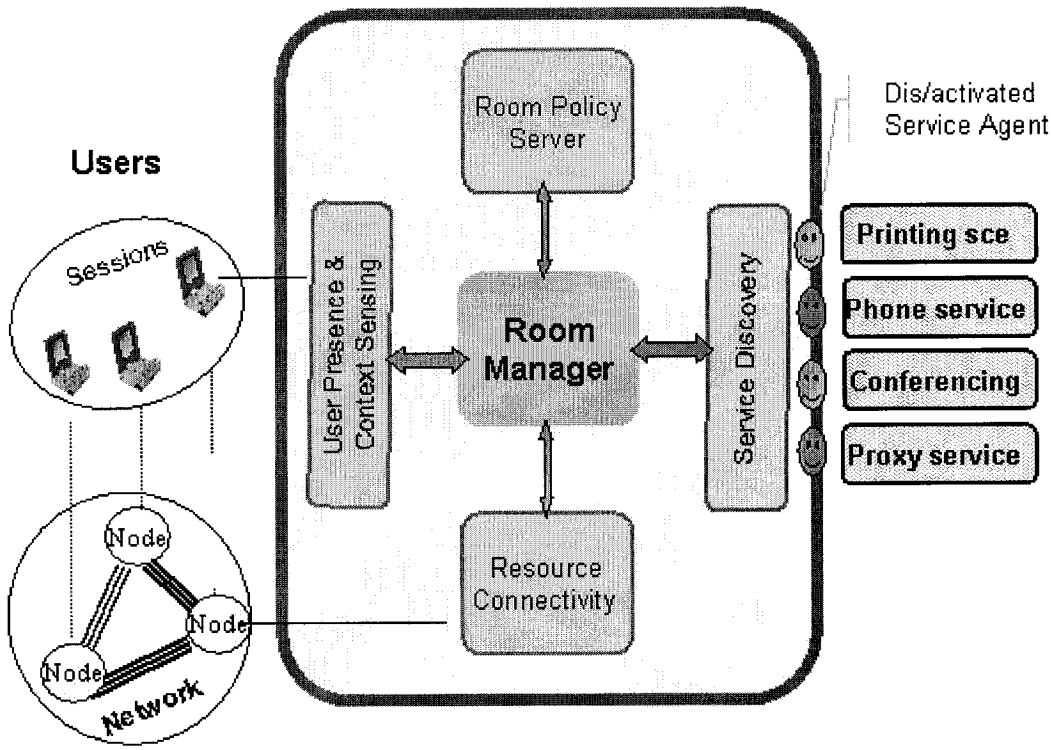


Fig 6.1: Ad-hoc Communications System Overview

The purpose of this communications system is to provide easy collaboration between different room entities, be it users or services. The system is agent-based and is built upon the FIPA-OS agent platform. The architecture has a central controller called Room Manager that manages and facilitates interactions between the different components. One of the lower level components is the Presence and Context Awareness layer. This component can be considered as the first line of operation since it is responsible for detecting the presence and absence of users and services. It is also obliged to prepare these context information ready for use by other components.

The Resource Connectivity module is responsible for managing communication sessions between users and between users and services. The Room Policy Server maintains the set of policies regarding users and services. These policies are mainly used to determine service access permissions and restrictions. An example of a policy may state that a service can only be accessed by users with specific credentials.

Finally Service Discovery takes on the role of connecting users with services. As we have learned, it is responsible for the automatic discovery and access of services. We now present its implementation details and how it is integrated with the ad-hoc communications system.

6.3 Centralized Discovery Model

Recall that in the centralized Service Discovery model, there is one SDA to manage discovery operations in the room. Services are represented by SAs and must register and deregister with the SDA. Users are represented by PAs and are responsible for listening to service announcements and making search requests. As service information changes, announcements are made by the SDA to those PAs authorized to access the affected services. Therefore the users always have the most current list of services in the network they are allowed to use, reducing their work to a minimum. These are the operations and the structure for the basic architecture of the centralized model and it is what has been implemented thus far.

6.3.1 SAs and Room Manager

The basic architecture is composed of three types of components: SAs, PAs, and the SDA. As we are implementing this model, we must keep in mind that it is to be integrated with the rest of the modules in the ad-hoc communications system. There are certain aspects of other modules and system requirements that have an affect on the Service Discovery implementation.

When a user walks into the conference room, she along with her mobile device are detected (via sensors) by the Context Awareness layer of our ad-hoc system. The Context Awareness module is also designed to launch SAs for services when they become available. Therefore initiating SAs is in this case not a concern of our discovery module implementation. An SA performs one of two tasks during discovery:

- Registration with the SDA immediately after being launched. A list of service attributes is sent along with the registration message.
- Deregistration with the SDA when the service is shut down.

These two actions drive the SDA to maintain a dynamic list of active services in the conference room.

Another system component that affects Service Discovery is the Room Manager. The Room Manager, as seen in Fig 6.1, manages and is in touch with the other modules in the room. It has access to the Policy Server and thus as part of its design, it is responsible for distributing services to the appropriate user PAs according to policies. Therefore, from the SDA's perspective, Room Manager is considered a super PA with the

highest credentials. All service information are forwarded to the Room Manager since it has the privilege to oversee all the services in the room. Note that the Room Manager is an agent. Since SAs and PAs are now implemented as part of the Context Awareness module and the Room Manager, this leaves the SDA as the only entity we are concerned with for our centralized discovery module implementation.

6.3.2 SDA Implementation

We now look at the details of the SDA implementation. The structure of the SDA, agent communications, test scenario and other aspects are discussed.

6.3.2.1 SDA Breakdown

As an agent in the FIPA-OS platform, the SDA must register with the AMS in order to be a valid agent. We also require the SDA to register with the DF. Although this is not necessary, the SDA is however an agent that serves the entire room network. Therefore it should be readily visible to the other agents. DF is a very convenient tool for locating agents. Since it is a fundamental agent on all FIPA-compliant platforms, it is always present and ready to be used. Its function is similar to that of address location protocols such as DHCP (used in SLP and other discovery protocols as mentioned in Chapter 2). And yet the DF is a part of the agent platform, therefore eliminating the need for us to use a third-party entity to locate the SDA.

By registering with the DF, other agents such as SAs and the Room Manager can easily find the SDA. Although in the current implementation of the ad-hoc system, the

address of the SDA is statically configured for the SAs and the Room Manager, but the SDA is ready to accommodate future scalability upgrades to these agents.

From the basic centralized architecture, we distinguish the implementation of the SDA discovery actions into the following:

- Receives registration messages from SAs, updates service registry, updates affected PAs
- Receives deregistration messages from SAs, updates service registry, updates affected PAs

The centralized model employs a push operation mode. The actions of the SDA are therefore driven by the advertisements of the SAs. Whenever a registration or deregistration message is received from an SA, it triggers the SDA to update its service registry and inform the authorized PAs of this service information.

We can see from the above operation breakdown that the SDA actions are composed of two categories. First the SDA must perform standard agent operations (i.e. initialization, registering with DF, shutting down, etc.). Then it has it own set of unique tasks which in this case refers to the discovery operations. This breakdown forms the basis for our implementation of the SDA. Figure 6.2 shows the class make-up of the SDA.

There is a main SDA class called *SDAgent*. It is the main representation of the SDA and takes care of the agent operations such as initialization and shut down. It is initiated immediately after the agent platform starts running (i.e. AMS, DF, and MTS all

initiated). *SDAgent* registers itself with the AMS and DF. It then initiates a data structure (i.e. a hashtable called *SA_list*) for the storage of available services. After initialization, *SDAgent* prepares for discovery operations by launching a task (class *IdleTask*) that listens for and processes incoming messages from the SAs.

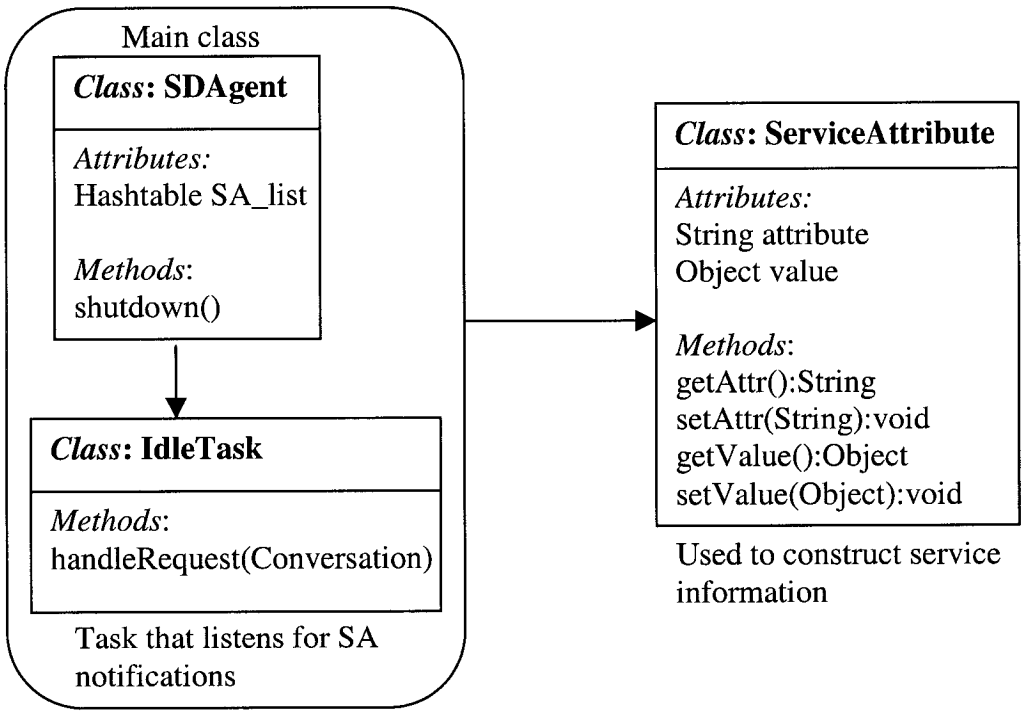


Fig 6.2: SDA Class Make-up

We therefore have the two classes to implement the two types of SDA operations. Note that there is also a third class called *ServcieAttribute*. This class allows for the creation of objects that represent individual service attributes. These service attribute objects are stored along with their corresponding services in *SDAgent*'s service registry (*SA_list*).

6.3.2.2 Discovery Operations in IdleTask

We now take a closer look at the discovery operations implemented in class *IdleTask*. If an incoming ACL message from an SA is a service deregister message, the SA information is removed from *SA_list*. A new ACL message is then constructed and sent to the Room Manager. The message contains the information of the exit of this SA.

If the ACL message is a service register message (see code snippet in Fig 6.3), then the service information is extracted from the message and processed. This information contains all the attribute values of the service. It is processed and added to the service registry *SA_list*. The service information is then put into a new ACL message that is sent to the Room Manager. This message also contains the SA identifier and its new registration status. With *IdleTask* taking care of these two discovery operations, the Room Manager will always receive service update information when changes occur.

```
//if this is a registration message from an SA
} else if (msg.getOntology().equalsIgnoreCase(SA_STATUS_REGISTER)) {
    System.out.println("SDA: Received register message from service agent "+key);

    //obtain the service information from the message
    Hashtable ht = (Hashtable)msg.getContentObject();
    Enumeration keys = ht.keys();

    //process the service information into the appropriate format
    ServiceAttribute sa;
    Vector saVec = new Vector();
    while (keys.hasMoreElements()) {
        String attr = (String)keys.nextElement();
        Object value = ht.get(attr);
        sa = new ServiceAttribute(attr, value);
        saVec.add(sa);
    }

    //store the information in service registry
    SA_list.put(key, saVec);

    //indicate in the ACL message to the Room Manager that
    //this is the registration of the SA,
    //all service information (attributes) are included as well
    ht.put("Action", "Register");
    ht.put("AgentID", msg.getSenderAID());

    //send the message to Room Manager
    acl.setContentObject(ht);
    forward(acl);
    DIAGNOSTICS.println("SDA: Sent register update to "+_RM_id.getName(), DIAGNOSTICS.
}
```

Fig 6.3: Code snippet for processing service registration

6.3.2.3 ACL Structure

We have seen that ACL is used exclusively for agent communication. It allows the capability for object encapsulation and therefore enables a wide variety of information to be carried between agents. The construction of an ACL message is shown in the code snippets. Each ACL message contains the same group of fields such as sender ID, receiver ID, message content, etc. As an example, the service registration ACL message that our SDA sends to the Room Manager sets the following fields:

- Performative = REQUEST (the nature of the message)
- SenderAID = agent ID of SDA
- ReceiverAID = agent ID of Room Manager
- ContentObject = hashtable containing service attributes, agent ID of the SA, and action (i.e. service registration)

The ContentObject field stores a Java object that contains all the information a sender agent needs to send to the receiver agent. Since it is of type Object, any type of Java Object is valid. This allows for great versatility in agent communication.

6.3.2.4 Test Scenario

The implementation of the centralized model was successful in that the Room Manager promptly receives all service notifications from the SDA whenever an SA

becomes active or inactive. One test scenario used is to have a service enter and then leave the network. Fig 6.4 shows the outputs of the running SDA in this scenario.

```
main ! SDAgent ! 5:     Agent has been assigned AgentID (agent-identifier :name
sda@sdaDis :addresses (sequence fipaos-rmi://137.122.109.170:3000/sda ) )

main ! SDAgent ! 5:     fipaos.platform.ams.AMSRegistrationException: Registrati
on with (agent-identifier :name ams@sdaDis :addresses (sequence fipaos-rmi://137
.122.109.170:3000/ams ) ) failed :already-registered

SDA: Received register message from service agent Service00011528@csa

main ! N/A ! 4: SDA: Sent register update to h-mmarlRma@h-mmarl

SDA: Received deregister message from service agent Service00011528@csa

main ! N/A ! 4: SDA: Sent deregister update to h-mmarlRma@h-mmarl
```

Fig 6.4: SDA Output Example

We can see that the SDA is first initialized and assigned an agent identifier (sda@sdaDis where sdaDis is the name of the agent platform the SDA belongs to. After the initialization process, the SDA waits for messages from SAs in the network. In the example, it first receives a service registration message from an SA with ID Service00011528@csa. The SDA updates this service information in its registry and then we see that it sends the registration update to the Room Manager (h-mmarl@h-mmarl).

The next message the SDA receives is a deregistration message from the same SA. The SDA removes this service from the registry and we see that it then sends the deregistration update to the Room Manager. Fig 6.5 illustrates the agent interactions in this scenario with a sequence chart.

We have therefore started off with a basic centralized model implementation that operates smoothly as part of the agent-based ad-hoc communications system. It paves the way for further successful implementations of our centralized discovery design.

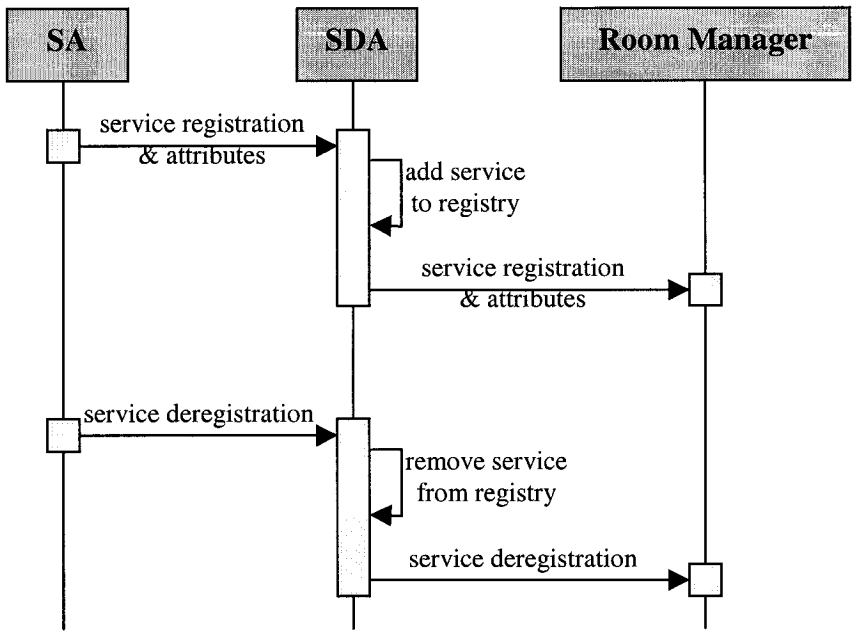


Fig 6.5: Sequence Chart for Test Scenario

6.4 Distributed Discovery Model

The distributed discovery model assumes a fully ad-hoc environment within the conference room. There are no existing devices in the room prior to the entrance of any user. Recall that in the distributed model, each device carries its own SDA and Description Agent. Two service registries are maintained by the Description Agent: a local directory (LD) for locally offered services, and a directory cache (DC) for foreign services found. The model employs the pull operation mode but also uses push when services are discovered, creating a balance that allows for both discovery efficiency and better usage of limited device space. An existing P2P search algorithm is used as a basis for the discovery operations and is built upon to suit the dynamic environment of our network. In this Section we will discuss the details of the distributed model implementation as well as the implementation of a service agent (SA).

6.4.1 Device Class Structure

The distributed design has specified that within a device, there should be an SDA, a Description Agent, and an UI for the user. Therefore we have two agents and an interface to implement. Fig 6.6 outlines the class structure for these three entities.

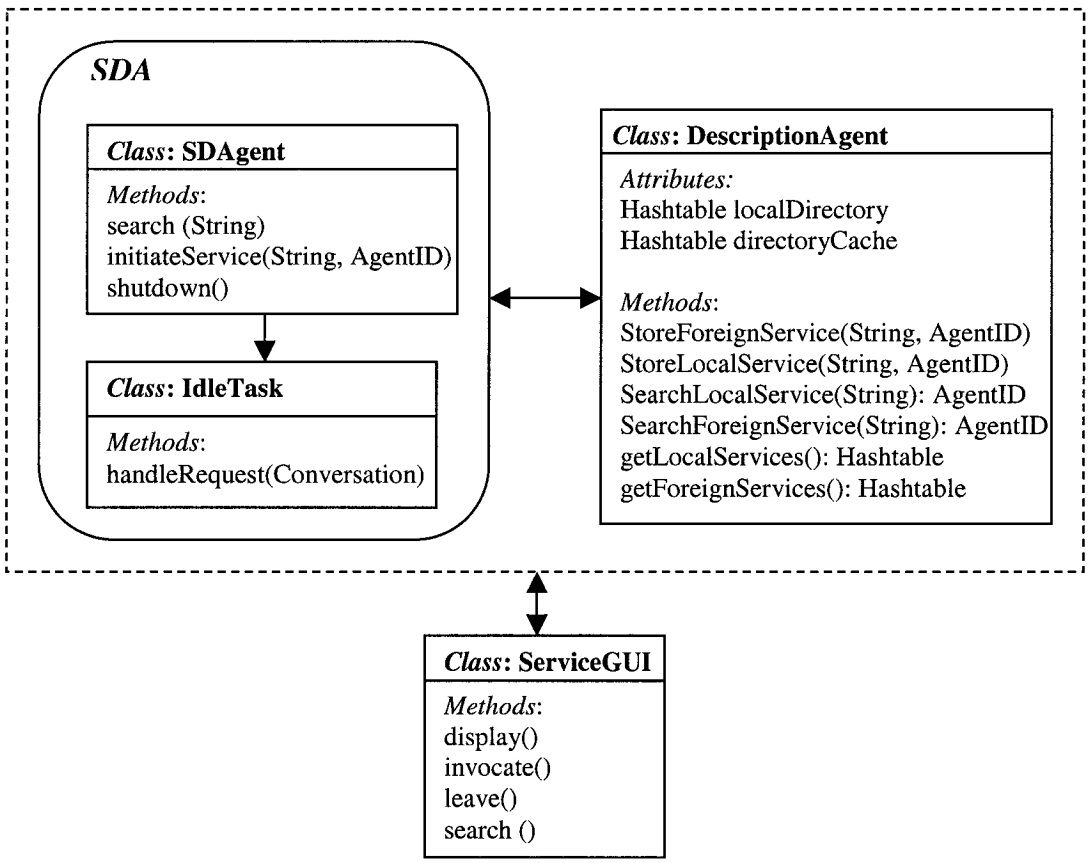


Fig 6.6: Device Class Structure

The SDA again is constructed of two classes. *SDAgent* is the main SDA class that takes care of agent initiation and registrations with AMS and DF. A device becomes visible discovery participation when its SDA initiates and starts presence awareness. The SDA also provides the public methods *search* and *initiateService*. These two methods correspond to the two types of actions the user can perform from the GUI. Therefore

these methods are called from the GUI (*ServiceGUI*). Once again *SDAgent* initiates *IdleTask* to listen for and process incoming discovery messages from other SDAs. *IdleTask* performs the core discovery functions which will be discussed in the next Section.

The Description Agent (*DescriptionAgent*) maintains two hashtables that act as the two service directories LD and DC. It also provides a string of access methods for the SDA to perform service queries and retrievals. Finally *ServiceGUI* is the graphical interface presented to the user of the device. It displays dynamically the local and foreign services found. The user is able to initiate a service from the list and also perform searches by providing the type of the service.

The GUI has a simple design that clearly guides the user to the possible tasks that can be done. A drop-down menu will display the currently available services. This list is updated dynamically whenever a service is found or removed. The user has the option of selecting a service from the menu and click on *Invocate* to use the service. This is not required however if the user had originally initiated a search. When a service is found, it will be added to the list in the GUI and the interface for the service will be initiated automatically for the user. This brings us to the second option on the GUI which is the search function. The user is required to enter a service type for discovery. Service GUI is initiated by the SDA when after startup. For our implementation, the GUI is an essential tool that must be present permanently for the user to access services and participate in discovery. Therefore if the GUI is shut down, then this will signal the end of the device presence in the network. The SDA will in turn stop operating and notify the room.

6.4.2 Discovery Operations in IdleTask

In this Section we look at the implementation of the main discovery functions of our distributed discovery model. These function reside in SDA's *IdleTask* which is always listening for discovery messages and processing them.

6.4.2.1 Presence Awareness Implementation

In the distributed model, presence awareness is the underlying layer that ensures all devices are always aware of the other current network devices. A device's SDA will announce its presence to the network periodically. If a device fails to resend the notification after the specified time then it is assumed to be unavailable. All the other devices will update their DCs to remove those services belonging to the now inactive device, therefore keeping a current and correct view of the network services.

To translate this design into actual implementation, we begin the awareness process in *IdleTask* (initiated by SDA after startup). When the task starts, it immediately sends a presence notification to all the other SDAs in the network. This is done by first obtaining a list of all current SDAs from the DF. The DF only belongs to the agent platform running on the current device, but it is capable of performing a federated search with DFs on other devices. When the list of SDAs is obtained, it is stored locally to represent the current view of the network devices. The list can now be used to send presence notifications (using ACL) to all the SDAs. The content of the message would indicate it is a presence notification. The sequence chart in Fig 6.7 illustrates the notification process.

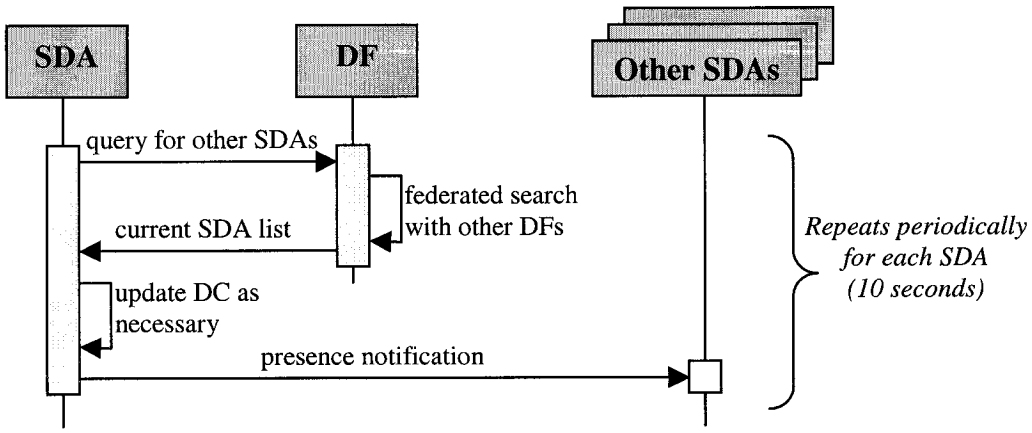


Fig 6.7: Sequence Chart for Presence Notification

The same process is repeated for each periodic notification. Therefore the locally stored SDA list is refreshed constantly. This allows *IdleTask* to perform presence notification and updating the current network view at the same time. If the newly refreshed SDA list is missing SDAs from the previous list, then these missing SDAs are no longer in the network. IdleTask will remove the affected services from the DC using the Description Agent.

When *IdleTask* receives a presence notification, it adds the SDA to its SDA list if the SDA is not already there. This may seem unnecessary since a refreshed SDA list will be obtained from the DF periodically as stated above. However if a device enters the network before the next DF query, then we want this new device be made known immediately. For simplicity, we set the delay between periodic presence notifications to be 10 seconds for all devices. If a device leaves the network, a proper shutdown of the SDA will take place and a shutdown message is sent to all other SDAs. Upon reception of a shutdown notification, *IdleTask* will remove this SDA from the current SDA list and also update the DC if necessary. We now have the presence awareness layer in place.

6.4.2.2 Service Search Implementation

An SDA's IdleTask can receive four types of messages from other SDAs:

- Presence Notification (for presence awareness)
- SDA Shutdown (for presence awareness)
- Search Request (for actual service discovery)
- Search Reply (for actual service discovery)

The first two (Presence Notification and Shutdown) we have already discussed in the previous Section. They serve presence awareness purposes for our distributed model. This leaves Search Request and Search Reply which are the messages used for performing actual discovery of services. We look at the processing of these two messages in this and the next Sections.

The distributed model specifies that when an SDA receives a Search Request (SR), it will first check its LD to see if the service is offered locally. The Description Agent is first consulted to find the service in the LD based on the type specified in the SR message. If this is successful, the ID of the matching SA (this includes the location of the SA) is put into a new ACL reply message. This message is sent directly to the source SDA that requested the service. A second reply message is then sent back along the search path. Fig 6.8 displays the section of code that concerns with sending the reply messages.

```
    forward(acl);
    DIAGNOSTICS.println("SDA: Sent search reply to original sender "+originalSender.getN

    //then send reply back along the forward path if needed
    if (!senderID.getName().equals(originalSender.getName())) {
        ACL acl2 = getNewConversation(FIPACONSTANTS.FIPA_REQUEST);
        acl2.setPerformative( FIPACONSTANTS.REQUEST );
        acl2.setSenderAID(_owner.getAID());
        acl2.setOntology(SERVICE_FOUND);

        acl2.setReceiverAID(senderID);
        ((Vector)h.get("path")).removeElementAt(pathLength-1);
        acl2.setContentObject(h);
        forward(acl2);
        DIAGNOSTICS.println("SDA: Sent search reply back along path to "+senderID.getNam
    }
```

Fig 6.8: Code Snippet for Sending Reply Messages

If the service is not found in the LD, the SR will need to be forwarded. If TTL is still greater than zero, the DC is checked for any matching foreign service. If a match is found, then the SR message will be forwarded to the SDA of that device. The TTL will be decreased for this message and the current SDA is added to the search path. Fig 6.9 shows a portion of the code that implements this.

```
//then check if we know another SDA has it
} else if (ttl>0) {
    result = dirAgent.SearchForeignService(type);
    if ((result!=null)&&(!result.getName().equals(originalSender.getName()))) {
        System.out.println("Service found in foreign list");
        //if found, forward search to that SDA(s)
        h.remove("ttl");
        h.put("ttl", new Integer(ttl--));

        ((Vector)h.get("path")).addElement(result);

        ACL acl = getNewConversation(FIPACONSTANTS.FIPA_REQUEST);
        acl.setPerformative( FIPACONSTANTS.REQUEST );
        acl.setSenderAID(_owner.getAID());
        acl.setReceiverAID(result);
        acl.setOntology(FORWARD_SEARCH);
        acl.setContentObject(h);
        forward(acl);
        DIAGNOSTICS.println("SDA: Forwarded search request to "+result.getName(), DIAGNO
```

Fig 6.9: Code Snippet for Forwarding SR Message

If the service is not found in the DC, then probabilistic forward is performed. The probability is hard coded in the code. A random number generator is used to generate a number for each SDA in the SDA list that is maintained by presence awareness (see earlier discussion on presence awareness). This number is compared with the probability to determine whether the SR message will be forwarded to this SDA. If yes and the SDA does not already exist in the search path, then the SR is forwarded to it. Again the TTL is decremented and the current SDA is added to the search path in the SR message. This concludes the processing of an incoming SR message. Note that an SR is originally generated by calling the *search* method provided in *SDAgent*. Recall this method is called by the GUI when the user click on *Search*.

6.4.2.3 Search Reply Implementation

When a service is found locally by an SDA, a reply message is sent to the source SDA as well as back along the search path. In our implementation, these two reply messages take on the same format so that IdleTask only have one type of reply message to receive and process.

After receiving a reply message, *IdleTask* first checks if the current device is the source device. If yes then it checks to see if it has already received a reply for this search. Only when it is the first search reply will *IdleTask* immediately call the *initiateService* method in the *SDAgent* class to access the service. If the current device is not the source device then it is just a participant in the original search path. The reply message will be forwarded to the next SDA in the reverse path. Note that in any case, the

service information is saved in the DC and the GUI will be informed to update the service list in the drop down menu.

6.4.3 MP3 Service Agent

Now that the main discovery operations are implemented, the next step is to create SAs for services so that service access can be realized. For the centralized model, the creation of SAs is the responsibility of another module in the ad-hoc communications system. For the distributed model implementation, we are assuming a fully ad-hoc environment and thus has not yet integrated the implementation with the centralized design of the communications system.

As stated earlier, not all SAs are the same since each SA must cater to the requirements of the type of service it represents. Some services can be used remotely and some cannot. In this Section we look at an example SA that is implemented for an MP3 service. This service provides a player interface as well as the underlying functions to play MP3 and other types of music files. A Java based MP3 player application called jlGui [45] is used as the actual service.

An MP3 player cannot be accessed remotely. Therefore it must be transported to the user's device. Agent mobility comes into play here. We use a designated mobility agent to carry the player to the user. The problem here is that the FIPA-OS platform does not support agent mobility. Therefore another method should be used to simulate mobility.

FIPA-OS uses HTTP as the underlying protocol for inter-platform agent communications. Since the HTTP server is already present and running, it can be used

for mobility purposes. The executable Java player file of jlGui is placed under the HTTP folder. The mobility agent's task is therefore to obtain this file through HTTP and execute it on the user device.

We know that the SA of this MP3 service will provide the player to the user. Fig 6.10 shows the sequence chart of the MP3 service access process. When a user's SDA issues an access request to this SA, the message is actually a move request. It contains the ID of the mobility agent on the user's device. When the SA receives this request, it will reply back to the mobility agent on the user's device with the location of the executable player file. The mobility agent can then take this file and execute the player on the user's device, thus completing the service access process.

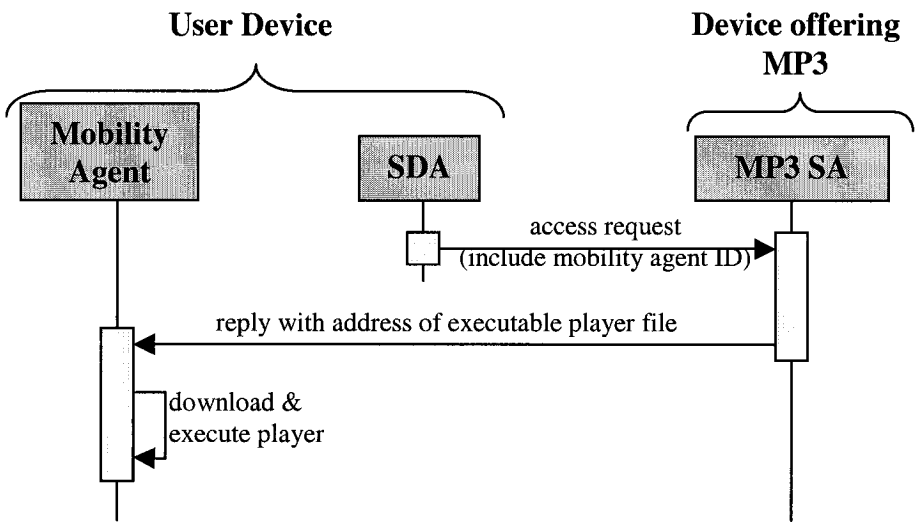


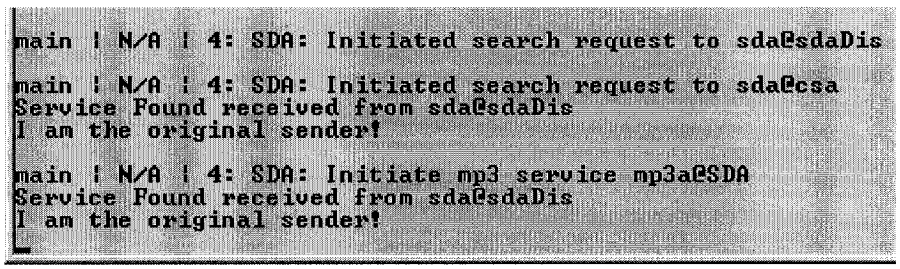
Fig 6.10: Sequence Chart for MP3 Service Access

6.4.4 Test Scenario

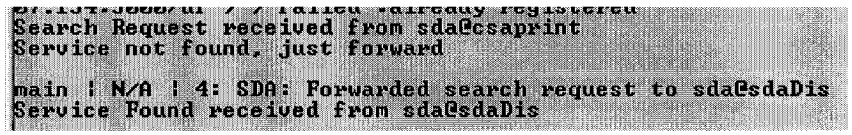
Due to limited resources in our research laboratory, we perform initial testing on three desktop computers that simulate an ad-hoc environment. The startup and shutdown of the SDA indicates the entrance and exit of the device. One discovery scenario

involves having one device initiate a search for an MP3 service. This service is located on one of the other two devices.

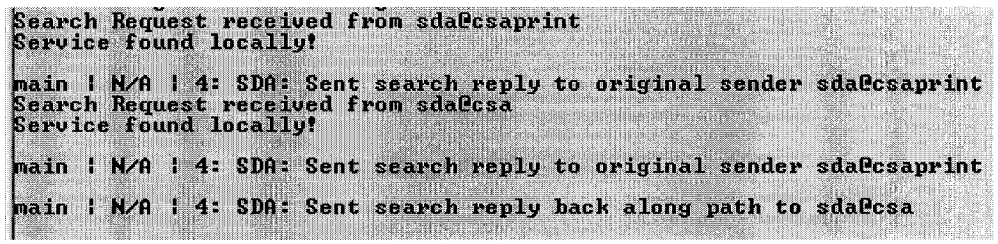
The agent platforms on the three devices are named *csa*, *csaprint*, and *sdaDis*. The search for MP3 is initiated on *csaprint* when the term 'mp3' is entered into the search field of the GUI and *Search* is clicked. The MP3 service resides on *sdaDis*. None of the other two devices have this service stored in their directory cache. Fig 6.11 shows the output of each of the three SDAs.



a) Output for *sda@csaprint*



b) Output for *sda@csa*



c) Output for *sda@sdaDis*

Fig 6.11: SDA Outputs for Test Scenario

Since we only have three devices, the forward probability is set to 100%. When the search is performed, all three devices are present in the network. We see that sda@csaprint sends out the search request to both sda@csa and sda@sdaDis. This is

because the forward probability is 100%. Now there are two search paths traveling concurrently. Fig 6.12 illustrates the progression of these two search paths. The output of sda@csa indicates that it has received the request and the service cannot be found in either of its service registries. It then forwards the request to sda@sdaDis. Notice that it does not forward back to sda@csaprint since it is the source SDA.

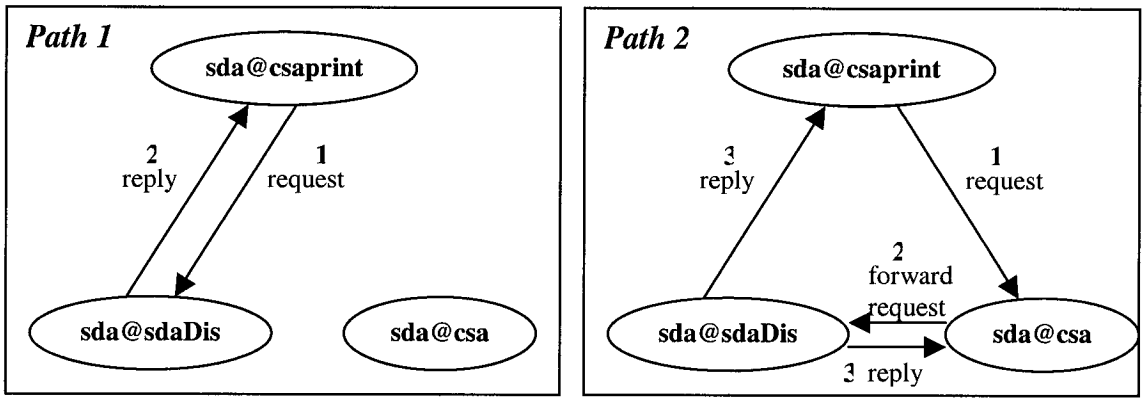


Fig 6.12: Search Paths for Test Scenario

The output of sda@sdaDis shows that it receives the search request twice. The first request is sent directly from the source sda@csaprint. The second one is forwarded from sda@csa. In both cases sda@sdaDis will reply directly back to the source. In the second case, it will also send the reply back to sda@csa so that the reply message can travel back along the search path. We see that the source sda@csaprint receives two identical replies, but the MP3 service is only initiated when the first reply is received. The search and access operations are successfully executed. Through testing these types of scenarios, we can conclude that the main discovery functions are implemented successfully. Further evaluation on a larger number of devices is still needed to do better and more thorough testing on a dynamic environment.

6.5 Summary

Implementations of the discovery models are done using Java under the Windows 2000 environment. FIPA-OS is the agent platform used for the agent-based models. The centralized model implementation is integrated with the ad-hoc communications system. The basic architecture of the centralized model is realized where there is one SDA to control all discovery actions in the network. Integration with the ad-hoc communications systems simplifies our test scenario into having only the Room Manager as a PA. Discovery operations run successfully for the centralized model implementation. The distributed model implementation involves creating the same SDA, Description Agent, and GUI for each device. An example SA shown is the MP3 SA where the MP3 service itself must be physically carried to the user. Mobility is simulated by using a HTTP server. The implementation is tested with a small network of three devices. The main discovery operations executes successfully. This paves the way for more future evaluations involving a greater number of devices.

Chapter 7

Conclusion

Service Discovery is a highly useful and convenient feature for any network. But in an ad-hoc environment, the presence of Service Discovery has become even more essential. The discovery solutions proposed in this Thesis explore different approaches of improving Service Discovery in ad-hoc environments with the use of Agent Technology. In this concluding Chapter, we summarized the major contributions of the Thesis and outline some possible future research directions to take.

7.1 Summary

Service Discovery mechanisms enable networks and their users to automatically and spontaneously discover services without doing much work. This is an especially useful tool for ad-hoc environments where the dynamic nature of devices makes services much more difficult to manage.

Some current ad-hoc discovery-related projects and protocols such as Konark and UPnP provide very thorough discovery features that cover all areas of concern. However they usually rely on many other Internet protocols for communication and other purposes, increasing the complexity of the overall discovery model. Our proposed models take on a completely agent-based approach where the agent platform handles all underlying communication details. This makes the designs simpler while still very capable of handling most discovery concerns, especially for our distributed discovery model.

From studying discovery protocols such as SLP and UPnP, we can organize the general discovery techniques into four categories: distributed pull, centralized pull, distributed push, and centralized push. Each has its own merits and drawbacks. Some are suited for one type of network environment. Others are suited for a different type of environment. These are kept in mind as we design our discovery models.

Our first contribution is the design and implementation of the centralized discovery model for the conference room environment. The goal was to reduce user work as much as possible. We assume a hybrid network environment and therefore use a centralized push approach. Services are dynamically discovered and maintained by the central agent SDA who automatically pushes all services and update information to all users based on user authorization. This model ensures that users will have the most

current service list at all times and therefore their work is reduced to a bare minimum. Service searches are only necessary when a service is not in the room.

The extension of our centralized model into an inter-domain discovery design forms the second contribution of this Thesis. The proposal here is to designate each domain's (i.e. room) SDA to perform any inter-domain discovery communication. Agent cloning and mobility are used here to enable remote access of services in other rooms. This type of discovery brings great convenience for different networks that are related or work together.

Our final contribution is the design and implementation of the distributed discovery model. We assume a fully ad-hoc environment with a dynamic P2P topology. Reducing user work is still important as well as reducing the strain on individual devices since now all devices are mobile and most have only limited space and power. These concerns translate into a balance between push and pull. This balance is offered in Menascé's P2P search algorithm where discovery becomes progressively more efficient and unnecessary for the users. We use this algorithm as a basis and enhance it for our ad-hoc environment. Therefore our distributed model achieves discovery with a unique combination of distributed pull and push while reducing the risk of flooding devices with too much service information.

7.2 Future Research Directions

We have seen from our implementations that more thorough testing and evaluations are still needed. This requires a better testing environment with more

resources to create realistic situations. It will allow us to better judge our designs and their efficiency.

From the design perspective, an area that can be refined is inter-domain discovery, especially for the distributed discovery model. Inter-domain discovery is more difficult with a distributed design. A room does not have a control entity, therefore it is trickier to find a uniform way of communicating with other rooms. However with the use of agents and agent platforms, it may be possible to use the idea of federated DFs to achieve inter-domain discovery. We have already used agent cloning and mobility to our great benefit in the designs. Other potentially beneficial agent characteristics such as learning can also be explored in the future.

Looking at our distributed design, there are a few places that can be improved. In the current design, it is possible for an SDA to receive more than one *identical* search replies if this SDA has participated in several search paths of the same search. While this does not harm the search result, it is creating redundancy that wastes bandwidth. A mechanism can be put in place to prevent an SDA from processing more than one search request of the same search.

Another thing to note is that in the original P2P algorithm, a probabilistic forward is only applied to the node's neighbours. Since in our situation, the devices are mobile and this makes the notion of neighbours difficult to track. However it is beneficial to only forward to neighbours, again to improve bandwidth efficiency. We may try dynamic sensing and triangulation techniques to calculate neighbours. As we proceed with these possible future research areas, our discovery designs will become more discovery efficient and bandwidth efficient for ad-hoc environments.

References

[1] D. Chakraborty and H. Chen, "Service Discovery in the Future for Mobile Commerce", *ACM Crossroads*, vol. 7, issue 2, December 2000.

[2] R.E. McGrath, "Discovery and Its Discontents: Discovery Protocols for Ubiquitous Computing", *Presented at Center for Excellence in Space Data and Information Science*, NASA Goddard Space Flight Center, April 2000, available at http://www.ncsa.uiuc.edu/People/mcgrath/Discovery/dp.pdf

[3] C. Bettstetter and C. Renner, "A Comparison of Service Discovery Protocols and Implementation of the Service Location Protocol", September 2000, available at http://wwwtgs.cs.utwente.nl/Docs/eunice/summerschool/papers/paper5-1.pdf

[4] C. Tschudin, "Lecture Notes for Data Networks II", Uppsala University, Sweden, 2000, available at http://user.it.uu.se/~tschudin/lect/20002001/dn2/slides/adhoc-4up.pdf

[5] S. Motegi, K. Yoshihara and H. Horiuchi, "Service Discovery for Wireless Ad Hoc Networks", *The 5th International Symposium on Wireless Personal Multimedia Communications*, vol. 1, October 2002, pp. 232-236.

[6] A. Shaikh, R. Tewari and M. Agrawal, "On the Effectiveness of DNS-based Server Selection", *Proceedings of Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies*, vol. 3, April 2001, pp. 1801-1810.

[7] C. Severance, "Could LDAP Be the Next Killer DAP?", *IEEE Computer*, vol. 30, issue 8, August 1997, pp. 88-89.

[8] J. Veizades, E. Guttman and C. Perkins, "Service Location Protocol", *RFC 2165*, IETF, June 1997.

[9] R. Droms, "Dynamic Host Configuration Protocol", *RFC 2131*, IETF, March 1997.

[10] E. Guttman, "Service Location Protocol: Automatic Discovery of IP Network Services", *IEEE Internet Computing*, vol. 3, issue 4, August 1999, pp. 71-80.

[11] J. Caldwell, "Service Location Protocol: A Java Prototype", Columbia University, April 1998, available at http://www.cs.columbia.edu/~hgs/teaching/ais/1998/projects/slp/slp_project.pdf

[12] A. Foster and K.J. MacGregor, "The Use of a Java Implementation of SLP in a Tele-teaching Application", *Proceedings of 7th IEEE Workshop on Future Trends of Distributed Computing Systems*, December 1999, pp. 273 - 278.

[13] K. Arnold, B. O'Sullivan, R.W. Scheiffler, J. Waldo and A. Wollrath, "The Jini Specification", MA, USA, Addison-Wesley, 1999.

[14] K. Arnold, "The Jini Architecture: Dynamic Services in a Flexible Network", *Proceedings of the 36th ACM/IEEE Conference on Design Automation*, June 1999, pp. 157-162.

[15] J. Allard, V. Chinta, S. Gundala and G.G. Richard III, "Jini Meets UPnP: An Architecture for Jini/UPnP Interoperability", *Proceedings of the Symposium on Applications and the Internet*, January 2003, pp. 268-275.

[16] Universal Plug and Play Specification, available at http://www.upnp.org

[17] Goland *et al.*, "Simple Service Discovery Protocol", *IETF Draft*, 2000, available at http://www.upnp.org/download/draft_cai_ssdp_v1_03.txt

[18] "Extensible Markup Language", W3C, available at http://www.w3.org/XML

[19] "Simple Object Access Protocol v. 1.2", *W3C Technical Report*, 2001, available at http://www.w3.org/TR/soap12

[20] Cohen, Aggarwal and Golan, "General Event Notification Architecture", IETF *Draft*, 2000, available at http://www.upnp.org/draft-cohen-gena-client-01.txt

[21] B.A. Miller, T. Nixon, C. Tai and M.D. Wood, "Home Networking with Universal Plug and Play", *IEEE Communications Magazine*, vol. 39, issue 12, December 2001, pp. 104-109.

[22] Salutation Specification, The Salutation Consortium, available at http://salutation.org

[23] Bluetooth Specification Part E: Service Discovery Protocol, available at http://www.bluetooth.org/spec

[24] The Ninja Project, available at http://ninja.cs.berkeley.edu

[25] M. Nidd, "Service Discovery in DEAPspace", *IEEE Personal Communications*, vol. 8, issue 4, August 2001, pp. 39-45.

[26] R. Hermann, D. Husemann, M. Moser, M. Nidd, C. Rohner and A. Schade, "DEAPspace - Transient Ad-Hoc Networking of Pervasive Devices", *First Annual Workshop on Mobile and Ad Hoc Networking and Computing*, August, 2000, pp. 133-134.

[27] S. Franklin and A. Graesser, "Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents", *Intelligent Agent III*, Lecture Notes in Artificial Intelligence 1193, Springer Verlag, 1997.

[28] P. Maes, "Artificial Life Meets Entertainment: Life like Autonomous Agents", *Communications of the ACM*, vol. 38, issue 11, November 1995, pp. 108-114.

[29] D.B. Lange and D.T. Chang, "IBM Aglets Workbench - Programming Mobile Agents in Java", *White Paper*, IBM Corp. September 1996.

[30] OMG, Common Facilities RFP3, Request for Proposal OMG TC Document 95-11-3, November 1995, available at http://www.omg.org/

[31] "FIPA Abstract Architecture Specification", Foundation for Intelligent Physical Agents, February 2002, available at http://www.fipa.org/

[32] M.K. Perdikeas, F.G. Chatzipapadopoulos and I.S. Venieris, "An Evaluation Study of Mobile Agent Technology: Standardization, Implementation and Evolution", *IEEE International Conference on Multimedia Computing and Systems*, vol. 2, June 1999, pp. 287-291.

[33] ightweight Extensible Agent Platform, available at http://leap.crm-paris.com/

[34] Java Agent DEvelopment Framework, available at http://jade.cselt.it/

[35] J. Macker and S. Corson, "Mobile Ad hoc Networking (MANET): Routing Protocol Performance Issues and Evaluation Considerations", *RFC2501*, January 1999, available at http://www.ietf.org/rfc/rfc2501.txt

[36] C.E. Perkins and P. Bhagwat, "Highly Dynamic Destination-Sequenced DistanceVector Routing (DSDV) for Mobile Computers", *ACM SIGCOMM: Computer Communications Review*, vol. 24, no. 4, October 1994, pp. 234-244.

[37] S. Murthy and J.J. Garcia-Luna-Aceves, "An Efficient Routing Protocol for Wireless Networks", *ACM Mobile Networks and Applications Journal, Special Issue on Routing in Mobile Communication Networks*, October 1996.

[38] D.B. Johnson and D.A. Maltz, "Dynamic Source Routing in Ad Hoc Wireless Networks", *in Mobile Computing*, edited by T. Imielinski and H. Korth, Chapter 5, Kluwer Academic Publishers, 1996, pp. 153-181.

[39] C.E. Perkins and E.M. Royer, "Ad-hoc On-Demand Distance Vector Routing", *Second IEEE Workshop on Mobile Computing Systems and Applications*, February 1999, pp. 90-100.

[40] S. Helal, N. Desai and V. Verma, "Konark - A Service Discovery and Delivery Protocol for Ad-hoc Networks", University of Florida, USA, available at http://www.harris.cise.ufl.edu/projects/publications/konark_paper.pdf

[41] M. Storey, G. Blair and A. Friday, "MARE: Resource Discovery and Configuration in Ad Hoc Networks", *Mobile Networks and Applications*, vol. 7, issue 5, October 2002, pp. 377-387.

[42] C. Dabrowski and K. Mills, "Understanding Self-healing in Service-Discovery Systems", *Proceedings of the First Workshop on Self-healing Systems*, November 2002, pp. 15-20.

[43] D.A. Menascé, "Scalable P2P Search", *IEEE Internet Computing*, vol. 7, issue 2, March/April 2003, pp. 83-87.

[44] "FIPA ACL Message Structure Specification", Foundation for Intelligent Physical Agents, October 2001, available at http://www.fipa.org/

[45] jlGui - Java Music Player, available at http://www.javazoom.net/jlgui/jlgui.html