AN INFRASTRUCTURE TO COMMUNICATE WITH WIRELESS DEVICES

SHARON KOUBI

An Infrastructure to Communicate with Wireless Devices

By

© Sharon Koubi

A thesis submitted to the School of Graduate Studies in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Memorial University

Submitted on March 2008

St. John's Newfoundland

Abstract

Contemporary and future network protocols allow wireless devices to send and receive information with reasonable reliability and at reasonable speed. Yet, for an application to take advantage of the full networking capabilities of modern devices, much overhead is needed. Although the physical networking capabilities are embedded in the wireless device, an accepted standardized software protocol for utilizing these capabilities is not fully in place yet. There is a need for an infrastructure and a protocol for data communication with wireless devices. Such an infrastructure could serve as a middleware tool for wireless application developers that will decrease the amount of overhead for wireless application development. This work proposes the function and structure for that infrastructure, the details of the protocol that can be used and discusses issues of selfishness and cooperation when such middleware is used cooperatively by uncoordinated parties.

Acknowledgments

I would like to thank the faculty and staff of the computer science department at Memorial University for their help and support through the years I have been a student here, in particular my supervisor Dr. Banzhaf and my Honours supervisor Dr. Mata-Montero. I would also like to acknowledge the great help I received from Dr. Shalaby from the Math Department. My Masters thesis would not be in the interesting area of wireless networks if not for the initiative and ideas of the people at Consilient, particularly Mr. Trevor Adey, Mr. Rod White and Mr. Dwayne Bennett. In addition to great ideas, this work was generously supported financially by Consilient. Finally, I would like to thank my wife, Inbal Bahar, for her support and for her help with editing this work.

1. Introduction and Roadmap

The Internet offers access to information sources worldwide. With the advance of wireless networking we expect to benefit from that access everywhere, not only when we arrive at familiar places such our homes or offices [11]. Contemporary wireless technology offers an increasing variety of wireless devices that allow Internet connectivity [22] and leads us to the vision of nomadic computing in which technology allows anyone to leave their office and still have seamless access to the same set of network services as they had at their office [12]. Enterprises are looking for mobile solutions that empower their employees to work more productively while on the road. In many areas there is a growing need for advanced applications that will decrease the gap between the level of productivity that can be achieved on a mobile device and on a desktop workstation [20].

However, while the availability of wireless networks and capable mobile devices are a necessary condition for mobile enabled applications, it is not a sufficient one. A significant trend is the requirement of ever-faster service development and deployment. An immediate conclusion is the requirement for various services and application frameworks and platforms; i.e., middleware that supports the rapid development of applications that will support mobile devices [24]. Typical middleware services include directory, trading and brokerage services for transactions, persistent repositories and most important different transparencies such as location and failure transparency [4].

This work lays a design and analysis for a middleware infrastructure that supports mobile devices. The presented infrastructure is named AIM: Advanced Infrastructure for Mobile devices and it is focused on allowing developers to create applications for mobile devices that will seamlessly combine with existing distributed enterprise applications.

This work is organized as follows. Chapter 1 is this short introduction. The 2nd and 3rd chapters review related work. Chapter 2 is reviews mobile oriented middleware. Chapter 3 reviews inducing cooperation. The algorithm that controls the infrastructure is designed to induce cooperation among the participants.

The chapters that follow describe the proposed infrastructure. Chapter 4 is an introduction to the AIM infrastructure. Chapter 5 has a more formal description of the AIM network components, and Chapter 6 includes a detailed description of the proposed infrastructure protocol. Chapter 7 is dedicated to the investigation of inducing cooperation among the participants of the infrastructure. The chapter presents the algorithm that is used to induce cooperation. Chapter 8 describes the implementation of the AIM infrastructure. Finally Chapter 9 summarizes the work and presents ideas for future related work.

2. The Need for an Infrastructure

This chapter reviews related work in the field of middleware for mobile applications. Applications for mobile devices present challenging problems to designers and developers. Devices face temporary and unannounced loss of network connectivity when they move, and connection sessions can be short and they need to discover other hosts in an ad-hoc manner. Handheld devices are likely to have limited resources compared to desktop workstations, such as low battery power, slow CPUs, little memory and a limited display. Changes in the working environment are likely to occur frequently, such as change of location or context conditions and variability of network bandwidth [16].

The development of distributed applications for mobile devices can be a complex process. The application designers should not have to deal explicitly with problems related to distribution, such as heterogeneity, scalability, and resource sharing. The role of middleware in this case would be to supply designers and developers with a higher level of abstraction, hiding the complexity introduced by distribution and the unique mobile environment. This chapter describes the characteristics for middleware that supports mobile devices and reviews existing solutions.

AIM is designed as a middleware application. In this chapter the concepts of middleware applications are reviewed. Several examples of other middleware applications for mobile devices are discussed.

2.1. Middleware Concepts

Building distributed applications, either mobile or stationary, on top of the network layer is extremely tedious and error-prone. Application developers would have to deal explicitly with all the non-functional requirements such as heterogeneity and fault-tolerance, and this complicates considerably the development and maintenance of an application. Middleware that takes care of these issues simplifies the process greatly. This chapter describes general concepts related to the design of middleware systems and more specific concepts that deal with middleware for systems that support mobile devices.

2.1.1. Middleware Systems

A distributed system consists of a collection of components, distributed over various computers (also called hosts) connected via a computer network. These components need to interact with each other, in order, for example, to exchange data or to access each other's services. Although this interaction may be built directly on top of network operating system primitives, this would be too complex for many application developers. Instead, middleware is positioned between distributed system components and network operating system components. The task of the middleware system is to facilitate component interactions. Figure 2.1 illustrates an example of a distributed system.

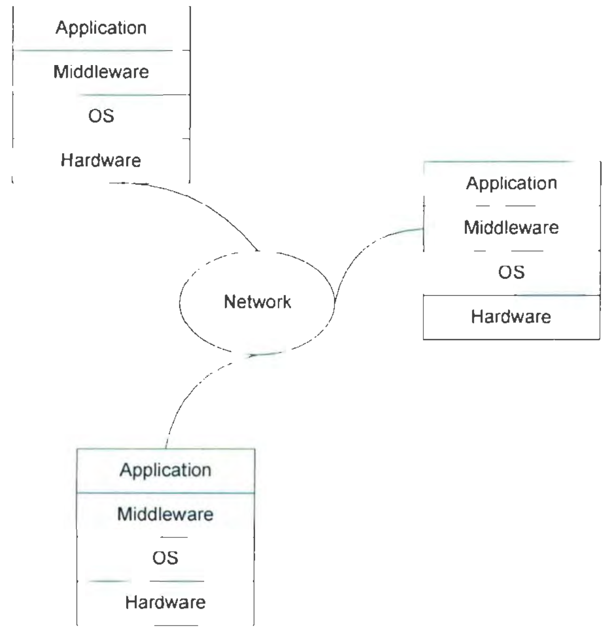

Figure 2.1 A distributed system (adapted from [16])

To support designers building distributed applications, middleware system positioned between the network operating system and the distributed application is put into place. middleware implements the Session and Presentation Layer of the ISO/OSI Reference Model as seen in figure 2.2. Its main goal is to enable communication between distributed components. To do so, it provides application developers with a higher level of abstraction built using the primitives of the network operating system. Middleware also offers solutions to resource sharing and fault tolerance requirements.

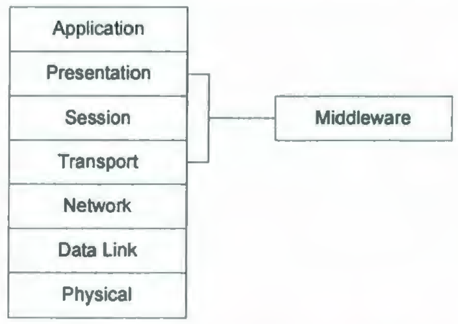

Figure 2.2 Middleware and the ISO/OSI model

2.1.2. Differences between Fixed and Mobile Distributed Systems

This definition of a distributed system applies to both fixed and mobile systems. The differences between the two systems are explained in terms of the great influence of the type of middleware system: the concept of device, of network connection, and of execution context. These concepts are described in Figure 2.3.

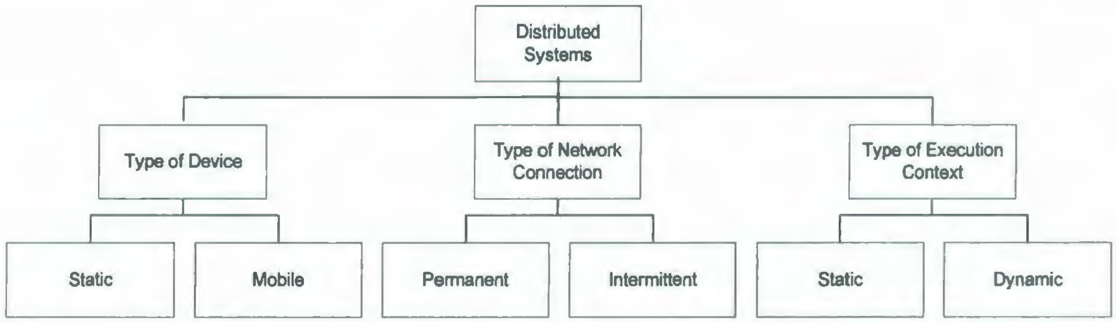

Figure 2.3 Characterization of concepts of middleware systems

Type of Device: as a first basic distinction, devices in a fixed distributed system are fixed, while they are mobile in a mobile distributed system. This is a key point: fixed devices vary from home PCs, to Unix workstations, to IBM mainframes; mobile devices vary from personal digital assistants, to mobile phones and cameras. While the former are generally powerful machines, with large amounts of memory and very fast processors, the latter have limited capabilities, like slower CPU speed, less memory, limited battery power and smaller screen resolution.

Type of Network Connection: fixed hosts are usually permanently connected to the network through continuous high-bandwidth links. Disconnections are either explicitly performed for administrative reasons or are caused by unpredictable failures. These failures are treated as exceptions to the normal behaviour of the system. Such assumptions do not hold for mobile devices that connect to the Internet via wireless links. The performance of wireless networks, as was demonstrated in Chapter 2, may vary depending on the protocols and technologies being used. Reasonable bandwidth may be achieved in some cases, for instance in the case of 3G networks. However, for some of the technologies, all different hosts in a cell share the bandwidth, and if they grow, the bandwidth drops. Moreover, if a device moves to an area with no coverage or with high interference, bandwidth may suddenly drop to zero and the connection may be lost. Also, while moving the network address might change, depending on the protocol. Unpredictable disconnections cannot be considered as an exception, but they rather become part of normal wireless communication. Some network protocols have a broader coverage in some areas but provide bandwidth that is smaller by orders of magnitude than the one provided by fixed network protocols. Also, cellular networks sometimes charge the users for the period of time they are connected; this pushes users to patterns of short time connections. Either because of failures or explicit disconnections, the network connection of mobile distributed systems is typically intermittent. With more networks available, mobile devices sometimes have the possibility to choose between several available networks. One type of network might be better for one task than another.

Type of Execution Context: by context, we mean everything that can influence the behaviour of an application. This includes resources internal to the device, such as amount of memory or screen size, and external resources, such as bandwidth, quality of the network connection, location or hosts (or services) in the proximity. In a fixed distributed environment, context is more or less static: bandwidth is high and continuous, location almost never changes, hosts can be added, deleted or moved, but the frequency at which this happens is by orders of magnitude lower than in mobile settings. Services may change as well, but the discovery of available services is easily performed by forcing service providers to register with a well-known location service. Context is extremely dynamic in mobile systems. Hosts may come and leave generally much more rapidly. Service lookup is more complex in the mobile scenario, especially in case the fixed infrastructure is completely missing. Broadcasting, transmitting information that will be received (conceptually) by every node on the network, is the usual way of implementing service advertisement; however, this has to be carefully engineered in order to save the limited resources (e.g., sending and receiving is power consuming), and to avoid flooding the network with messages. Location is no longer fixed: the size of wireless devices has shrunk so much that most of them can be carried in a pocket and moved around easily. Depending on location and mobility, bandwidth and quality of the network connection may vary greatly. For example, if a PDA is equipped with both a WiFi network card and a GPRS module, connection may drop from 10Mbs bandwidth, when close to an access point (e.g., in a conference room) to less than 48 Kpbs when we are outdoor in a GPRS cell (e.g., in a car on our way home).

2.1.3. Middleware for Fixed Distributed Systems

Middleware for fixed distributed systems can be mainly described as resource-consuming systems that hide most of the details of distribution from application designers. With the exception of message-oriented middleware, they mainly support synchronous communication between components as the basic interaction paradigm. We now discuss in more details the relationship between the physical structure of fixed distributed systems and the characteristics of associated middleware, in the context of the concepts mentioned in the previous chapter.

Fixed Devices → Heavy Computational Load: Wired distributed systems consist of resource-rich fixed devices. When building distributed applications on top of this infrastructure, it is worthwhile exploiting all the resources available (e.g., fast processors, large amounts of memory, etc.) in order to deliver better service to the application. The higher the robustness of the service, the heavier the middleware running underneath the application. This is due to the set of non-functional requirements that the middleware achieves, like fault tolerance, security or resource sharing.

Permanent Connection → Synchronous Communication: Fixed distributed systems are often permanently connected to the network through high bandwidth and stable links. This means that the sender of a request and its receiver (i.e., the component asking for a service and the component delivering that service) are usually connected at the same time. A permanent connection allows therefore a synchronous form of communication, as the situations when client and server are not connected at the same time are considered only exceptions due to failures of the system (e.g., disconnection due to network overload). Asynchronous communication mechanisms are however also provided by message oriented middleware and by the CORBA specification. Although asynchronous communication is used also in fixed networks, the bulk of middleware applications have been developed using synchronous communication.

Static Context → Transparency: The execution context of a fixed distributed system is generally static: the location of a device seldom changes, the topology of the system is preserved over time, bandwidth remains stable, etc. The abundance of resources allows the disregard of application specific behaviours in favor of a transparent and still efficient approach. For example, to achieve fault tolerance, the middleware can transparently decide on which hosts to create replicas of data and where to redirect requests to access that data in case a network failure inhibits direct access to the master copy, in a completely transparent manner. Hiding context information inside the middleware eases the burden of application programmers that do not have to deal with the achievement of non-functional requirements (e.g., fault tolerance) explicitly, concentrating, instead, on the real problems of the application they are building.

2.1.4. Middleware for Mobile Systems

Middleware systems for mobile devices differ in some aspects. However, they present a set of similar characteristics that influence the way middleware should behave.

Mobile Devices → Light Computational Load: Mobile applications run on resource-scarce devices, with less memory, slower CPU, and limited battery power. Due to these resources limitations, heavy-weight middleware systems optimized for powerful machines do not suit mobile scenarios. Therefore, a trade-off between computational load and nonfunctional requirements achieved by the middleware needs to be established. An example of this might be to relax the assumption of keeping replicas always synchronized, and allow the existence of diverging replicas that will eventually reconcile, in favor of a lighter-weight middleware.

Intermittent Connection → Asynchronous Communication: Mobile devices connect to the network opportunistically for short periods of time, mainly to access some data or to request a service. Even during these periods, the available bandwidth is lower than in fixed distributed systems, and it may also suddenly drop to zero in areas with no network coverage. It is often the case that the client asking for a service, and the server delivering that service, are not connected at the same time. In order to allow interaction between components that are not executing along the same time line, an asynchronous form of communication is necessary. For example, it might be possible for a client to ask for a service, disconnect from the network, and collect the result of the request at some point later when able to reconnect.

Dynamic Context → Awareness: Unlike fixed distributed systems, mobile systems run in an extremely dynamic context. Bandwidth may not be stable, services that are available now may not be there a second later, because, for example, while moving the hand-held device loses connection with the service provider. The high variability (along with the constrained resources) influences the way middleware makes decisions. The optimization of the application and middleware behaviour using application and context aware techniques becomes then more important, also given the limited resources.

2.2. Middleware for Mobile Distributed Systems

In this subsection we shall give some examples of middleware systems that are oriented toward servicing mobile devices. Each of the surveyed systems, Mobiware, UIC, Xmiddle and Jini is used as an example to demonstrate an important middleware system feature.

2.2.1. Asynchronous communication using JMS

Message-oriented middleware systems support communication between distributed components via message-passing: the sender sends a message to identified queues, which usually reside on a server. A receiver retrieves the message from the queue at a different time and may acknowledge the reply. Therefore, message-oriented middleware support asynchronous communication by achieving de-coupling of senders and receivers. In most cases, given the way they are implemented, these middleware systems usually require resource-rich devices, especially in terms of memory and disk space, where persistent queues of messages that have been received but not yet processed, are stored.

As discussed in [17], The Java Messaging Service (JMS) is a widely used interface than can be adapted to a mobile environment. However we shall discuss some of the adaptations needed for JMS in order to be truly adequate in a mobile setting. JMS is a collection of interfaces for asynchronous communication between distributed components. It provides a common way for Java programs to create, send and receive messages. JMS users are usually referred to as clients. The JMS specification further defines providers as the components in charge of implementing the messaging system and providing the administrative and control functionality (i.e., persistence and reliability) required by the system. Clients can send and receive messages, asynchronously, through the JMS provider, which is in charge of the delivery and, possibly, of the persistence of the messages.

Whilst the JMS specification has been extensively implemented and used in traditional distributed systems, adaptations for mobile environments have been proposed in the last several years. The challenges of porting JMS to mobile settings are considerable; however, in view of its widespread acceptance and use, there are considerable advantages in allowing the adaptation of existing applications to mobile environments and in allowing the interoperation of applications in the wired and wireless regions of a network.

If JMS is to be adapted to completely ad hoc environments, where no fixed infrastructure is available, and where nodes change location and status very dynamically, some issues must be taken into consideration. Firstly, discovery needs to use a resilient but distributed model: in this extremely dynamic environment, static solutions are unacceptable. A JMS administrator defines queues and topics on the provider. Clients can then learn about them using the Java Naming and Directory Interface (JNDI). However, due to the way JNDI is designed, a JNDI node (or more than one) needs to be in reach in order to obtain a binding of a name to an address (i.e., knowing where a specific queue/topic is). In mobile ad hoc environments, the discovery process cannot assume the existence of a fixed set of discovery servers that are always reachable, as this would not match the dynamicity of ad hoc networks. Secondly, a JMS Provider, as suggested by the JMS specification, also needs to be reachable by each node in the network, in order to communicate. This assumes a very centralized architecture, which again does not match the requirements of a mobile ad hoc setting, in which nodes may be moving and sparse: a more distributed and dynamic solution is needed. Persistence is, however, essential functionality in asynchronous communication environments as hosts are, by definition, connected at different times.

2.2.2. Mobiware - Using Traditional Middleware for Mobile Computing

In the following example traditional middleware is used for a mobile application. The focus is on provision of services from a back-bone network to a set of mobile devices. The main concerns in this example are connectivity and message exchange. In case of a less structured network or in case services must be provided by mobile devices, traditional middleware paradigms seems to be less suitable and a new set of strategies needs to be used. Therefore, communication of context information to the upper layers in order to monitor the condition of the environment and to adapt to application needs becomes vital to achieve reasonable quality of service.

Mobiware [2] is an example middleware that uses traditional middleware such as CORBA, IIOP and Java to allow service quality adaptation in a mobile setting. As shown in Figure 3.4, in Mobiware mobile devices are seen as terminal nodes of the network and the main operations and services are developed on a core programmable network of routers and switches. Mobile devices are connected to access points and can roam from an access point to another.

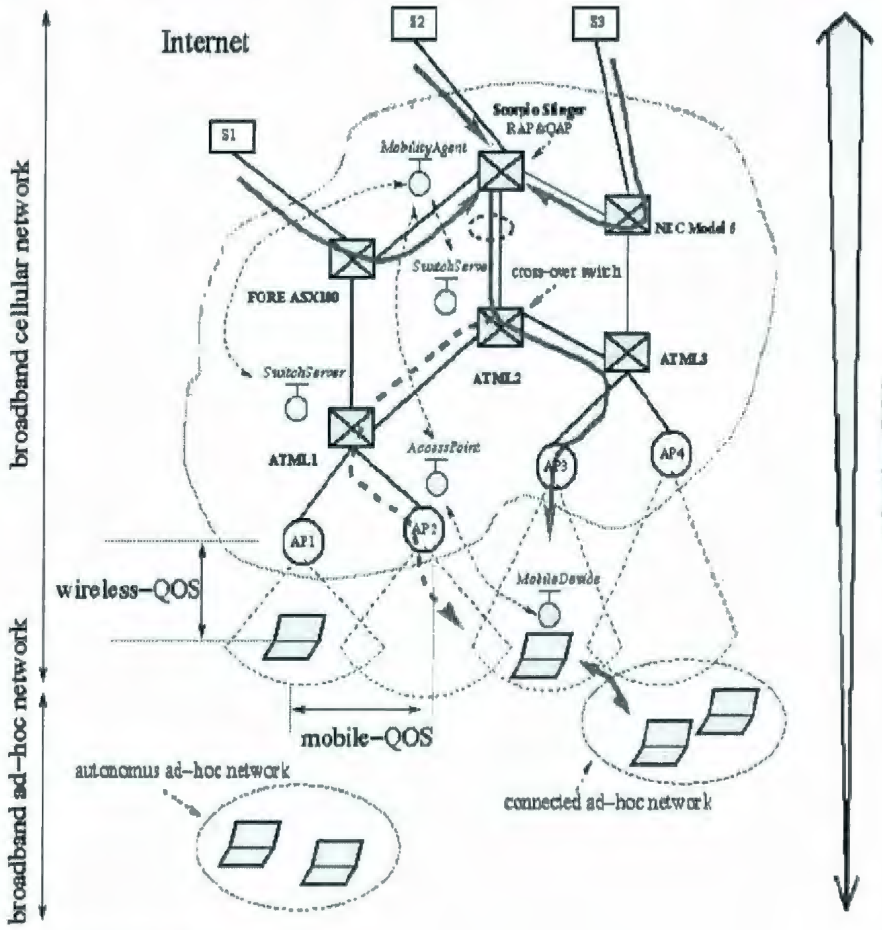

Figure 2.4 Mobiware architecture (from [2])

The main idea in Mobiware is that mobile devices will have to probe and adapt to the constantly changing resources over the wireless link. The experimental network used by Mobiware is composed of ATM switches, wireless access points, and broadband cellular connected mobile devices. The toolkit focuses on the delivery of multimedia application to devices with adaptation to the different quality of service and seamless mobility. Mobiware mostly assumes a service provision scenario where mobile devices are roaming but permanently connected, with fluctuating bandwidth. Even in the case of the ad-hoc broadband link, the device is supposed to receive the service provision from the core network through, first the cellular links and then some ad-hoc hops.

In more extreme scenarios, where links are all ad-hoc, these assumptions cannot be made and different middleware technologies need to be applied. One of the strength of Mobiware is the adaptation component to customize quality of service results. It is clear that middleware for mobile devices should not ignore context and that adaptation is a key point, given the limited resources and changing conditions.

2.2.3. UIC – Context Awareness Based Middleware

To enable applications to adapt to heterogeneity of hosts and networks as well as variations in the user's environment, systems must provide mobile applications the capability to be aware of the context in which they are being used [1]. Furthermore, context information can be used to optimize application behaviour counter balancing the scarce resource availability.

User's context includes but is not limited to:

Location: with varying accuracy depending on the positioning system used.

Relative: location, such as proximity to printers and databases.

Device characteristics: such as processing power and input devices.

Physical environment: such as noise level and bandwidth.

User's activity:, such as driving a car or sitting in a lecture theatre.

The Principle of Reflection has often been used to allow dynamic reconfiguration of middleware and has proven useful to offer context-awareness. The concept of reflection allows a program to access, reason about and alter its own interpretation. The role of reflection in distributed systems has to do with the introduction of more openness and flexibility into middleware platforms. In standard middleware, the complexity introduced through distribution is handled by means of abstraction. Implementations details are hidden from both users and application designers and encapsulated inside the middleware itself. Although having proved to be successful in building traditional distributed systems, this approach suffers from severe limitations when applied to the mobile setting. Hiding implementation details means that all the complexity is managed internally by the middleware layer. The middleware is in charge of making decisions on behalf of the application, without letting the application influence this choice. This may lead to computationally heavy middleware systems, characterized by large amounts of code and data they use in order to transparently deal with any kind of problems and find the solution that guarantees the best quality of service to the application. Heavyweight systems cannot however run efficiently on a mobile device as it cannot afford such a computational load. Moreover, in a mobile setting it is neither always possible, nor desirable, to hide all the implementation details from the user. The fundamental problem is that by hiding implementation details the middleware has to take decisions on behalf of the application. The application may, however, have vital information that could lead to more efficient or suitable decisions. Both these limitations can be overcome by reflection. A reflective system may bring modifications to itself by means of inspection and/or adaptation. Through inspection, the internal behaviour of a system is exposed, so that it becomes straightforward to insert additional behaviour to monitor the middleware implementation. Through adaptation, the internal behaviour of a system can be dynamically changed, by modification of existing features or by adding new ones. This means that a middleware core with only a minimal set of functionalities, can be installed on a mobile device, and then it is the application which is in charge of monitoring and adapting the behaviour of the middleware according to its own needs.

Universally Interoperable Core (UIC) [26] is a minimal middleware for mobile devices that is based on the concept of reflection. UIC is composed of a pluggable set of components that allow developers to specialize the middleware targeting at different devices and environments, thus solving heterogeneity issues. The configuration can also be automatically updated both at compile and run time. Personalities can be defined to have a client-side, server-side or both behaviours. Personalities can also define with which server type to interact (i.e., CORBA or Java RMI) as depicted in Figure 4.5: single personalities allow the interaction with only one type, while multiple personalities allow interaction with more than one type. In the case of multiple personalities, the middleware dynamically chooses the right interaction paradigm. The size of the core goes, for instance, from 16KB for a client-side CORBA personality running on a Palm OS device to 37KB for a client/server CORBA personality running on a Windows CE device.

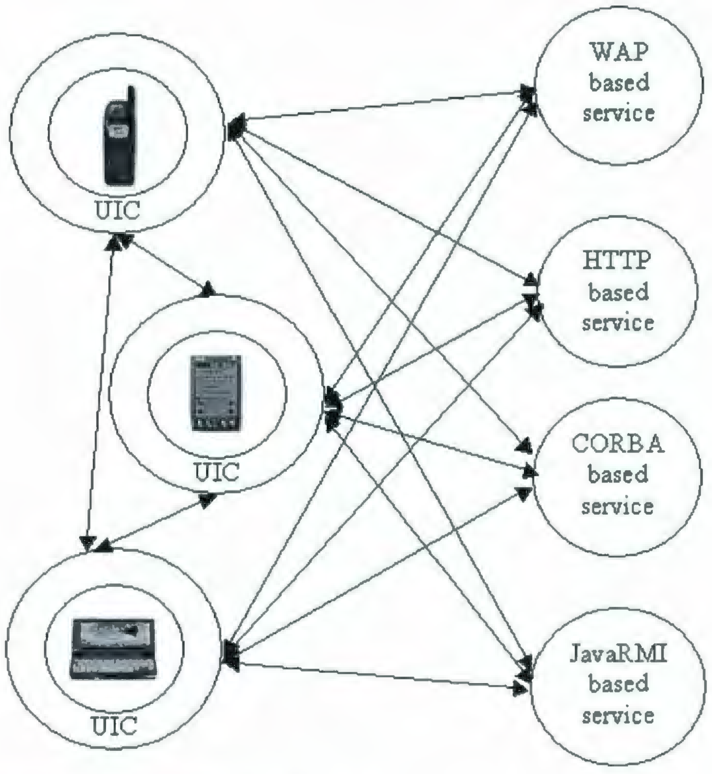

Figure 2.5 UIC Interaction (from [26])

2.2.4. Xmiddle – Data Sharing Oriented Middleware

One of the major issues targeted is the support for disconnected operations and data-sharing. Systems like Xmiddle [15] try to maximize availability of data, giving users access to replicas. Xmiddle allows mobile hosts to share data when they are connected, or replicate the data and perform operations on them off-line when they are disconnected. Reconciliation of data takes place once the hosts reconnect.

Xmiddle allows each device to store its data in a tree structure. Trees allow sophisticated manipulations due to the different node levels, hierarchy among the nodes, and the relationships among the different elements which could be defined.

When hosts get in touch with each other, they need to be able to interact. Xmiddle allows communication through sharing of trees. On each device, a set of possible access points for the private tree is defined; they essentially address branches of the tree that can be modified and read by peers. The size of these branches can vary from a single node to a complete tree. The unit of replication can be easily tuned to accommodate different needs. For example, replication of a full tree can be performed on a laptop, but only of a small branch on a PDA, as the memory capabilities of these devices differ.

In order to share data, a host needs to explicitly link to another host's tree. The concept of linking to a tree is similar to the mounting of network file systems in distributed operating systems to access and update information on a remote disk. As long as two hosts are connected, they can share and modify the information on each other's linked data trees. When disconnections occurs, both explicit (e.g., to save battery power or to perform changes in isolation from other hosts) and implicit (e.g., due to movement of a host into an out of reach area), the disconnected hosts retain replicas of the trees they were sharing while connected, and continue to be able to access and modify the data.

When the two hosts reconnect, the two different, possibly conflicting, replicas need to be reconciled. Xmiddle exploits uses tree differencing to detect differences between the replicas which hosts use to concurrently and off-line modify the shared data. However, it may happen that the reconciliation task cannot be completed by the Xmiddle layer alone, because, for example, different updates have been performed on the same node of the tree. In order to solve these conflicts, Xmiddle enables the mobile application engineer to associate application-specific conflict resolution policies to each node of the tree. Whenever a conflict is detected, the reconciliation process finds out which policy the application wants the middleware to apply, in order to successfully complete the merging procedure.

Xmiddle implements the tree data structure using XML and related technologies. In particular, application data are stored as XML documents, which can be semantically associated to trees. Related technologies, such as the Document Object Model (DOM), XPath and XLink [16], are then exploited to manipulate nodes, address branches, and manage references between different parts of an XML document. Reconciliation policies are specified as part of the XML Schema definition of the data structures that are handled by Xmiddle itself.

2.2.5. Jini – Service Discovery in Mobile Computing Middleware

In traditional middleware systems, service discovery is provided using fixed name services, which every host knows of its existence. The more dynamic the network becomes, the more difficult service and host discovery becomes. Already in distributed peer-to-peer network service discovery is more complex as hosts join and leave the network very frequently. In mobile systems service discovery can be quite simple: if we refer to nomadic systems where a fixed infrastructure containing all the information and the services is present. However, in terms of more ad-hoc or mixed systems, where services can be run on roaming hosts, discovery may become very complex and/or expensive.

Jini [3] is a distributed system middleware based on the idea of uniting groups of users and resources required by those users. Its main goal is to turn the network into a flexible, easily administered framework on which resources (both hardware devices and software programs) and services can be found, added and deleted by its users.

An important concept within the Jini architecture is the service. A service is an entity that can be used by a person, a program or another service. Members of a Jini system federate in order to share access to services. Services can be found and resolved using a lookup service that maps interfaces indicating the functionality provided by a service to sets of objects that implement that service. The lookup service acts as the central marketplace for offering and finding services by members of the federation. A service is added to a lookup service by a pair of protocols called discovery and join: the new service provider locates an appropriate lookup service by using the first protocol, and then it joins it, using the second one, as seen in Figure 2.6. A distributed security model is put in place in order to give access to resources only to authorized users.

Jini assumes the existence of a fixed infrastructure which provides mechanisms for devices, services and users to join and detach from a network in an easy, natural, often automatic, manner. It relies on the existence of a network of reasonable speed connecting

Jini technology-enabled devices.

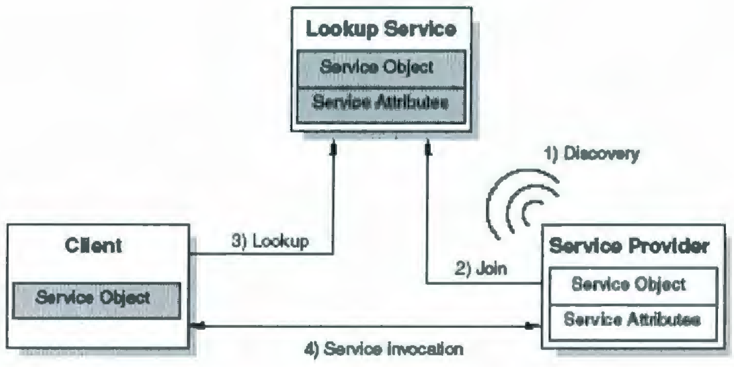

Figure 2.6 Discovery, join and lookup in Jini (from [3])

2.2.6. JCAF – Context Awareness

JCAF [21] is a ubiquitous programming environment, it is based on the high-level policy description language, a context-based access-control manager (CACM) for context-aware access control is described, and an adaptation engine which is integrated for context adaptation in dynamically changing environments.

The policy specification language consists of three parts: the entity relation definitions, access control rules, and adaptation rules. The language specifies the relations between the context entities to be used in the specification, the access control rules, and the adaptation rules. A context entity in a ubiquitous environment is either a physical or a logical space, a fixed object, or a moving object. An entity relation consists of two parts: context-relation and space-relation. A context-relation expresses a general relationship between entities, and a space relation expresses a space-containment relationship between a general entity and a space entity. An access control rule specifies that the given set of entities has the given right to the given object when the given condition is met. An adaptation rule specifies how to respond to events in a given context.

Access control rules are managed by CACM (Context-aware Access Control Manager). Before executing any method which is under access control, ubiquitous applications check the privilege of the calling entity to call the called entity. CACM manages a table which maps the related entities and a method name to a list of context conditions for the method call. Since there can be multiple sufficient conditions for a method call, each condition becomes an element of the list. CACM examines whether there exists any condition that is satisfied under the current dynamic context. If CACM finds one, it allows the requested access. Otherwise, CACM refuses the access by raising an exception.

Ubiquitous applications react to dynamically changing contexts. This is implemented by an adaptation engine. A user specifies the adaptation rules in a policy file, which describes how to respond when an event occurs in a given context. The adaptation engine is operated based on adaptation rules. For example, assume the setting of a hospital management system where there is an adaptation rule that specifies when a doctor and a patient are in the same consulting room and the doctor owns a PDA, then, in this situation, the information about the patient is displayed on the doctor's PDA automatically. Then, given any related event occurrence, such as the entrance of a patient or a doctor to a consulting room, the adaptation engine examines if all the context conditions are satisfied. If all the conditions of this rule are satisfied, the adaptation engine executes a method automatically, which displays information about the patient on the doctor's PDA.

2.3. Summary

The growing demand for mobile oriented software solutions have called for research and investigation of new middleware that will deal with those new computing challenges. In the last years we have seen active research in the field of middleware for wireless systems and a large number of new applications in that area. AIM, the system presented in this work, is designed to help solve some of the challenges facing mobile applications developers.

3. Inducing Cooperation

This chapter reviews related work regarding inducing cooperation. Inducing cooperation among participants is part of the AIM infrastructure. The way AIM induces cooperation is described in Chapter 7.

As described later in Chapter 4, AIM can operate in a public or in a private configuration. In the public configuration, Device Support Servers (DSS) are shared between different services. In this chapter we present the dilemma of a DSS whether to service requests from devices, and discuss relevant work for this topic. In the public configuration, a device can attempt to register to any DSS that is participating in that configuration. Unlike the private configuration, where the service provides enough DSS for all supported devices this is not always necessary in the public one. The public configuration allows devices to register to multiple services, therefore, services can share the resources they provide to support devices. The problem is how many device requests, and from whom, should DSS accept? When and why, if at all, should a DSS deny service from a device?

On the one hand, services would want to be serviced by a DSS as fast as possible; on the other hand, supplying DSS has a cost that services would prefer to minimize. Ideally, a service would not supply any DSS and have it devices supported by foreign DSS. Yet, if all services behave in this way no DSS will be supplied at all! It seems fair that every service should supply its fair share of DSS slots. The questions of what is the fair share of slots and how can it be calculated arise. In the public configuration, there is no central authority that can be assumed to assure that every service provides its fare share. Therefore, there is a need for a different strategy that will assure cooperation among the participating services. Our approach uses game theory to search for a strategy that could be deployed by each individual service and lead to cooperation among services that express rational behaviour.

Before modeling cooperation in AIM using we researched relevant work regarding similar cooperation problems and present the methods they suggest to induce cooperation.

We discuss a similar problem; this problem deals with cooperation in a network of participants where each node (or participant) wants to minimize the resources it uses while maximizing the level of service. The discussed problem is cooperation among mobile nodes in a mobile ad-hoc network [28].

3.1. Cooperation in Wireless Ad-Hoc Networks

3.1.1. Problem Description

A mobile ad-hoc network is a collection of mobile wireless nodes. It has no authority and is dynamic in nature. Ad-hoc networks have a wide array of military and commercial applications. Ad-hoc networks are ideal in situation where installing an infrastructure is not possible, the network is too transient or the infrastructure was destroyed. For example, nodes may be spread over an area that is too large for a single base station and a second base station is too expensive. Another example could be networks for wilderness expeditions and conferences that may be too transient if they exist only for a short period of time before dispersing or moving. Finally, if network infrastructure has been destroyed in a disaster, an ad-hoc network could be used to coordinate relief efforts.

Ad-hoc networks maximize total network throughput by using available nodes for routing and forwarding. Therefore, the more nodes that participate in packet routing, the greater the aggregate bandwidth, the shorter the possible routing paths, and the smaller the possibility of a network partition. However, a node may misbehave by agreeing to forward packets and then failing to do so, because it is over-loaded, selfish, malicious, or broken. An overloaded node lacks the resources to forward packets. A selfish node is unwilling to spend resources, particularly battery life to forward packets that are not of direct interest to it, even though it expects others to forward packets on its behalf. A malicious node launches a denial of service attack by dropping packets. A broken node might have a software fault that prevents it from forwarding packets.

Misbehaving nodes can be a significant problem. In addition to reducing the average throughput, nodes that are in proximity to misbehaving nodes might be affected severely, much more than the average. Different strategies where devised in order to overcome the problems caused by misbehaving nodes. The following chapters review some the approaches taken.

3.1.2. Classifying Node Behaviours

The first approach reviewed is classifying nodes by their behaviour and adjusting routing accordingly. The work [14] presents extensions to the Dynamic Source Algorithm (DSR) that attempt to detect and mitigate routing misbehaviour. In DSR every packet has a route path consisting of the addresses of nodes that have agreed to participate in routing the packet. It is an "on-demand" protocol because route paths are discovered when a source tries to sends packets to a destination for which the source has no path to. DSR contains two main functions: route discovery and route maintenance. Route discovery is done by sending a ROUTE REQUEST, as is illustrated by figure 3.1. Route maintenance handles link breaks. A link break occurs when two nodes on a path are no longer in transmission range. If this happens the source must try a different route or perform a route discovery.

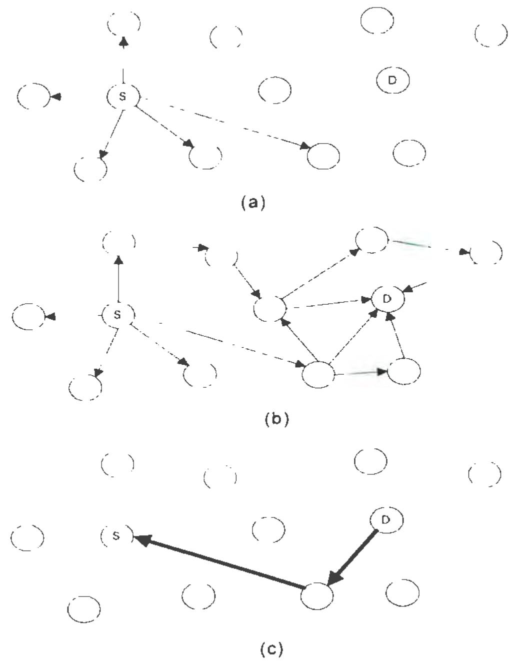

Figure 3.1 ROUTE REQUEST (a) node S sends a ROUTE REQUEST packet to find a path to node D. (b) The request is forwarded through the nodes of the network, each node adding its address to the packet. (c) D send back to S a ROUTE REPLY using the path in one of the ROUTE REQUEST packets it received.

The work further presents two methods to overcome node misbehaviour, Watchdog and Pathrater. Both methods assume that the wireless interfaces support promiscuous mode operation. This means that if node A is within the range of node B, then node A can overhear communications to and from B, even if those communications do not directly involve A.

The Watchdog method detects misbehaving nodes. By listening to the outgoing traffic from neighbouring nodes the Watchdog determines which of its neighbouring nodes is forwarding packets and which is misbehaving. There are a few problems with the Watchdog method. It might not detect misbehaviour due to collisions or limited transmission power or malicious nodes can collude in order to execute a more sophisticated attack. For example, node B might be receiving packets from node A to forward; it forwards them to C that drops them without B reporting to A that the packets are being dropped.

The Watchdog method comes to use when employed by the Pathrater. The Pathrater, run by each node in the network, combines knowledge of misbehaving nodes with link reliability data to pick the route most likely to be reliable. This differs from DSR, which chooses the shortest path. The Pathrater assigns rating to nodes according to the following algorithm. A node assigns itself the value of 1.0. A node previously unknown is assigned the neutral value of 0.5. The Pathrater increments the ratings of nodes on all actively used paths by 0.01 at periodic intervals. An actively used path is one which the node has sent a packet through. When a link break is detected and the node is unreachable its rating is decreased by 0.05. The maximum value a neutral node can attain is 0.8 and the minimum is 0. A misbehaving node is assigned a special high negative value, -100. When a Pathrater learns that a node in a path misbehaves and that no alternative paths is free of misbehaving nodes then it sends a ROUTE REQUEST. The extension that enables the additional request is called Send Route Request (SRR).

The Watchdog, Pathrater and SRR methods were tested using simulations. The simulations also included the simulation of misbehaving nodes. Different method combinations were tested: a network with no defenses, a network using Pathrater only, a network using Pathrater and Watchdog and a network using all three methods. The results showed a significant increase in network throughput for a network that employed the three methods.

Another work by Buchegger and Boudec [6] presents the CONFIDANT protocol. Similarly to the work by Marti et al. [14] CONFIDANT is an extension to DSR that aims at mitigating the effect of node misbehaviour by identifying misbehaving nodes. The CONFIDANT protocol defines the following components: The Monitor, the Reputation System, the Path Manager, and the Trust Manager. Theses components collect information on misbehaving nodes and distribute it to a list of "friendly" nodes. It is not specified how friendships are determined. Since promiscuous mode operation is assumed then nodes can also detect misbehaviour between their neighbours to other nodes. Misbehaviour is propagated through ALARM messages and a list of misbehaving nodes is maintained independently on each node.

In order to evaluate the protocol, several metrics where defined. One metric is the resulting total network throughput, or *goodput* by there definition. The goodput of a network of n nodes is the data forwarded to the correct destination for each node i:

$$G = \frac{\sum_{i=1}^{n} \text{Packets}_{\text{Recieved}}}{\sum_{i=1}^{n} \text{Packets}_{\text{Originated}}}$$

Another metric calculated is the overhead resulting from ALARM messages. The total overhead in a network of n nodes is defined as follows:

$$O = \frac{\sum_{i=1}^{n} \text{ALARM}_{\text{tx}}}{\sum_{i=1}^{n} \text{ROUTE}-\text{REQUEST}_{\text{tx}} + \text{ROUTE}-\text{REPLY}_{\text{tx}} + \text{ERROR}_{\text{tx}}}$$

In order to evaluate the CONFIDANT protocol simulations were carried. The simulated network contained a third of misbehaving nodes. The simulations resulted in a higher network goodput for a network that was fortified by CONFIDANT while the total overhead of ALARM messages never exceeded 3%.

3.1.3. Modeling the Network as a Market

In this subsection we review the approach of modeling a mobile ad-hoc network as a market. Services are exchanged and through a virtual economy based on a virtual currency. Nodes are forced to pay to have their packets forwarded, and are being paid when they forward some data for other nodes. Selfishness is avoided with a rewarding technique: a node is free to be selfish, but behaving in this way it will soon leave it without the ability to pay and it will not be able to send any packet. Unfortunately, this solution requires a tamper-proof hardware module, since it is not possible to avoid forging or stealing.

The work presented in [8, 9] attempts to present a solution for service availability in ad-hoc networks. Services are defined as all networking services (e.g. packet forwarding, mobility management, etc.) and should be provided by the other nodes that are participating in the network. The problem is that nodes do not benefit directly from providing services to other nodes and thus selfishness is profitable. Another problem that is presented is the overloading. Services can be unavailable because the network is overloaded and can no longer carry useful information. The network can become overloaded because of a malicious denial-of-service attack or simply because users want to send to much information. The goal of the work in [8, 9] is to stimulate co-operation and prevent overloading in such ad-hoc networks.

The approach that is taken to stimulate a co-operative behaviour and prevent congestion is to introduce the concept of money and service charges. The idea is that nodes that use a service should be charged and that nodes that provide a service should be paid. The work introduces a node currency that is called *nuggets*. Nodes need to "pay" nuggets for services and the only way to earn nuggets is by providing services to other nodes. The paper presents two approaches, the Packet Purse Model (PPM) and the Packet Trade Model (PTM).

In the Packet Purse Model the originator of the packet pays the packet forwarding service. The service charge is distributed among the forwarding nodes in the following way: when sending a packet, the originator loads it with a number of nuggets sufficient to reach the destination. Each forwarding node acquires one or several nuggets from the packet and thus increases the stock of its nuggets. The problem with this approach is that it might be difficult to estimate the number of nuggets that are required to reach a given destination. If the originator underestimates this number, then the packet will be discarded and the originator loses its investment in this packet. If the originator overestimates, then the packet will arrive but the originator loses the remaining nuggets. The PTM model overcomes this problem.

In the Packet Trade Model, the packet does not carry nuggets, nut it is traded for nuggets by intermediate nodes. Each intermediary "buys" it from the previous one for some nuggets, and "sells" it to the next one for more nuggets. In this way, each intermediary that provided a service by forwarding the packet increases its number of nuggets, and the total cost of forwarding the packet is covered by the destination of the packet. An advantage of this approach is that the originator does not have to know in advance the number of nuggets required to deliver a packet. Furthermore, letting the destination pay for the packet forwarding makes this approach applicable in multicast packets as well. A disadvantage is that this approach for charging does not directly deter users from flooding the network. However, allowing each node to decide if it buys a packet or not can provide a mechanism which may deter a user from generating too much traffic, by ensuring that eventually nobody will buy packets from users who try to overload the network.

As mentioned the main problem with the market approach is the requirement for a tamper resistant security module that will manage the nugget exchange. Such a module must be implemented in hardware and makes it unlikely that such a solution will be practical.

3.1.4. The Backbone Method

In [13], the routing backbone method is described to mitigate cooperation in selfish wireless networks. The algorithm is based on the following social dilemma: a group of rational individuals want a single person from the group to volunteer to offer some service. This service expends some of the volunteer's resources, but all the individuals, including the volunteer, benefit from the service if it is provided. In other words, this service is a public good. Each participant in the network needs some of the nodes to volunteer to provide the public good, but no one wants to be one of the volunteers.

The algorithm presented in [13] is based on the model Volunteer's Timing Dilemma (VTD). In this model, each player's strategy is no longer to "volunteer or not," but rather a time $T \geq 0$ that denotes "when to volunteer." If no one volunteers until time t, then the public good is not available until then. To capture the loss in utility from waiting, each player's utility decreases by a standard exponential discount factor. The authors elaborate on the VTD model and developed the Generalized Volunteer's Timing Dilemma (GVTD) model. The VTD model assume that all players can observe and benefit from any volunteer. In multihop networks, however, this assumption does not hold; each node needs a volunteer within its one-hop neighborhood, and therefore does not directly benefit from a volunteer two or more hops away. The input to GVTD is an arbitrary, undirected graph G. Note that the original VTD game is a special case where G is a complete graph.

The backbone construction protocol consists of two logical steps: leader selection and the connection of the leaders. In the first phase, nodes play the GVTD game. Based on the information about the cost distribution and its two-hop neighborhood, each node independently computes its optimal waiting time before volunteering. When there is no volunteer neighbor for a long (enough) time, it volunteers as a leader to speed up the backbone construction, and thus minimizes loss of its own messages. In the second phase, bridge nodes are chosen to connect the leaders and obtain a connected backbone (specifically, a connected dominating set).

3.1.5. A More Formal Approach

The work done by Srinivasan et al. [28] uses a more formal game theoretic approach to address the issue of user cooperation in ad-hoc networks. The work deals with solving the forwarding problem in ad-hoc wireless networks. They propose a distributed and scalable acceptance algorithm called Generous TIT-FOR-TAT (GTFT). The acceptance algorithm is used by the nodes of the network to decide whether to accept or reject a relay request. The work demonstrates that GTFT results in a Nash equilibrium and proves that the system converges to the rational and optimal operating point. A Nash equilibrium is a solution concept of a game involving two or more players, in which no player has anything to gain by changing only her own strategy unilaterally.

It is assumed that the nodes are rational, i.e., their actions are strictly determined by self interest, and that each node is associated with a minimum lifetime constraint. However, the assumption fails to recognize malicious intent by certain nodes as self interest. Given the lifetime constraints and the assumption of rational behaviour, it is determined what is the optimal throughput that each node should receive. This point is defined to be the rational Pareto optimal operating point. Therefore, resource allocation is optimized in such a way that no shifting of resources can be made without making at least one node worse off.

The paper gives a formal system model. The system consists of a population of N nodes distributed among K classes. The nodes are distributed to the different classes according to a power constraint. The power constraint determines how many packets can a node forward for how long. This helps define a Normalized Acceptance Rate (NAR) as the ratio of the number of successful relay requests generated by the node, to the number of relay requests made by the node. This quantity is an indication of the throughput experienced by the node.

Then, the work studies the optimal trade-off between the lifetime (based on the power constraint) and the NARs of the nodes. Given the power constraints, a feasible set of NARs is identified. This provides a set of Pareto optimal values. That is, values of NAR such that a node cannot improve its NAR without decreasing some other node's NAR. As mentioned, the nodes are assumed to be rational, that is that their actions are determined strictly by self interest and that self interest is strictly to increase the node's throughput. Using this assumption, a unique set of rational and Pareto optimal NARs a identified for each user.

Since users are self-interested and rational, there is no guarantee that they will follow a particular strategy unless they are convinced that they cannot do better by following some other strategy. In game theoretic terms, a set of strategies which constitute a Nash equilibrium needs to be identified. Ideally, a Nash Equilibrium would result in the rational and Pareto optimal operating point. This is achieved by proposing a distribute and scalable acceptance algorithm, called Generous TIT-FOR-TAT (GTFT). The paper proves that GTFT is a Nash Equilibrium which converges to the rational and Pareto optimal NARs. The paper concludes with simulations that show that the algorithm results in a Nash Equilibrium after a reasonable amount of time. The algorithm seems practical to implement in order to enhance a real life network. The weakness point of the algorithm is that it does not consider how to deal with malicious nodes whose self interest to not to increase their throughput but to decrease the throughput of the other nodes.

3.2. Summary

In this part we reviewed the problem of cooperation in a shared system. Since the cooperation among middleware participants was not investigated previously we used ad-hoc networks as a similar model. In the next chapter we shall apply these concepts on the AIM system.

4. Introducing AIM: Advanced Infrastructure for Mobile Devices

The main focus of this work is the design and implementation of an infrastructure that will help develop applications for a wide range of mobile device and help connect between these devices to a variety of services and applications. I chose the name AIM for this infrastructure, which stands for **A**dvanced **I**nfrastructure for **M**obile devices. Figure 4.1 shows a schema of the infrastructure. It is notable from the figure that AIM is situated between the mobile client application and a fixed service. This chapter will describe the features that this infrastructure offers to application developers, the principles upon which the infrastructure is based and the structure of the infrastructure.

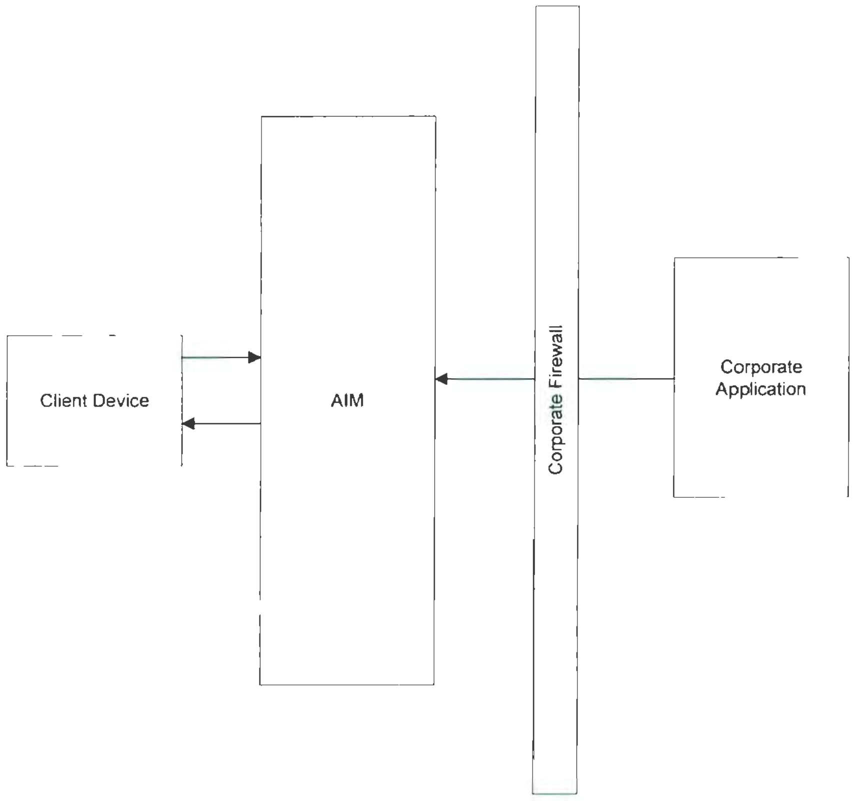

Figure 4.1 A schema of the AIM infrastructure. AIM will be a middleware layer that enables mobile applications to connect to corporate applications.

4.1. *Infrastructure Features*

AIM will provide several services that will make developing and adapting applications and services for mobile devices easier. The infrastructure makes unique mobile characteristics such as connection details, network identification and network problems transparent to the application developer, and allows the application to deal only with the application logic. In addition, AIM could serve as a connection point for mobile devices to various services and protocols. Infrastructure features can be divided into four categories: pushing data to mobile devices, connect mobile devices to corporate networks, handle intermittent connectivity and serve as a connection to adapters for protocols and applications. This chapter gives a brief overview of the features that are offered by AIM.

4.1.1. Pushing Data to Mobile Devices

Existing and new protocols allow data to be pushed to mobile devices that are connected to a network. The service level that is offered varies. In some cases large chunks of data can be pushed to an online device, while in other cases only notifications can be made [7, 25]. Also, the interface and other characteristics of these services can be very different. For example, pushing data to a mobile device through an SMS message is different mechanism and interface than using a listening socket in a java enabled device using MIDP 2.0. This adds much development effort to the extension of a push service to mobile devices; in particular, if the service is intended for a range of different devices [5]. A main feature of AIM is to make the pushing of data to mobile devices transparent to the application developer. The infrastructure supplies a standard API on both the client side and the server side in order to push data to the mobile device through the network or mechanism of choice.

4.1.2. Connect Mobile Devices to Corporate Networks

Security is a major concern for corporate network administrators [27]. Therefore, collaborative corporate applications are usually protected behind a firewall and reside only in the corporate internal network, except possibly for some limited interfaces. In an increasingly mobile environment corporate applications will be extended to mobile devices. However, the security configuration of such extensions is not necessarily trivial [10, 23]. AIM will provide controlled and safe tunneling for mobile applications to access information in the corporate intranet.

4.1.3. Handle Intermittent Connectivity

Connectivity in a mobile environment is likely to be interrupted in various situations. Problems can occur due to being out of coverage, low batteries, etc. [18]. AIM makes the handling of out-of-coverage situations easier by caching requests and responses and by taking care of potential data loss situations.

4.1.4. Filter Unwanted Information

Mobile devices, naturally, are more limited than desktop workstations. Less information can be displayed and processes, and in many cases the fees that incur are in proportion to the amount of data transmitted back and forth. AIM intends to allow a mobile user to filter the information she is receiving and thus still allows synchronization with services that are designed to serve desktops, but according to the rules that the mobile user is comfortable with. The protocol for this feature of AIM is not yet developed and it is not a part of the prototype.

4.1.5. AIM as a Connection Point to Protocols and Applications

Another feature of the AIM architecture is to serve as a connection point to various protocols and applications. For common application such as email protocols, there will be generic adapters that will allow quick and simple registration of corporate and private users. For proprietary services, AIM adapters could be created. These adapters should be customized per service. An adapter will be a software library that will mediate between the proprietary service and the AIM infrastructure. Through AIM adapters the infrastructure will supply an easy access point for mobile devices. In addition, the infrastructure provides a unique, randomly generated device id to identify a user on a device. The unique identification process of AIM will make provisioning services to new users easier. The protocol for this feature of AIM is not yet developed and it is not a part of the prototype.

4.2. Infrastructure concepts

There are several key issues that arise when designing an infrastructure. The AIM infrastructure is deigned to be used in private small settings, as a mobile service platform for solutions for large enterprises, and as a public platform that is shared by different enterprises or individual users. In the next few paragraphs the design concepts of the architecture are being described.

4.2.1. A Scalable Service

The platform must be able to handle a large number of service requests, which are coming from various wireless networks, concurrently. AIM is deigned as a distributed and scalable service without a central point that could serve as a bottle-neck. In order to transform from a small configuration that can handle several hundreds of users to a configuration that can handle hundreds of thousands or even millions of users, the number of participating servers needs to be increased, not requiring and complex configuration changes. More than that, there is no reasonable limit to the number of users that the infrastructure can handle in an efficient way. The scalability of AIM will be evident in later in Section 4.3 that describes the server and client structure of the infrastructure.

4.2.2. Security Policy

Since the infrastructure will act on behalf of the mobile user to access corporate resources, the infrastructure must obtain authorization based on user identity, channel security and corporate policy before accessing corporate databases, directories and email servers, etc.

4.2.3. Dependability and Decentralization

The infrastructure must be able to reconfigure itself dynamically when certain machines fail or become overloaded and continue to deliver services satisfying appropriate performance guarantees. The dependability of AIM is further discussed in chapter 5 dealing with the server structure and in chapter 6 in the protocol description.

4.2.4. Reduce Processing Time and Network Time for the Mobile Device

Mobile devices are usually limited in processing and network capabilities. Also excessive network usage can be very expensive. The infrastructure is designed to allow as much as possible processing on the infrastructure backend and to have network transmissions on the fastest and cheapest network available.

4.2.5. AIM as a Private or Shared Infrastructure

There are two modes in which AIM can be configured, private mode and shared mode. In the private mode the infrastructure is used by one organization and does not serve any external requests. The shared mode allows the infrastructure to be shared among many organizations, each contributing resources. There are advantages and disadvantages for both modes and each should be used according to the specific situation. In both cases however, corporate security is not compromised. The advantage of the private mode is a complete control over the available resources. The advantage of the shared configuration is that resources can be shared. This is advantageous in cases where mobile devices are using services and applications operated by several organizations.

4.3. The Structure of AIM

AIM is composed of a server and client components that monitor and control the traffic between the mobile device and an online service or application. The server components are used by online AIM enabled applications or AIM adapters to communicate with the mobile device. The client components are installed on the mobile device and allow AIM enabled clients to communicate with an online application. Traffic between an application and a mobile client could possibly, depend on the network situation, pass through an intermediate AIM server. Figure 4.2 shows a summary of client and server functions and properties.

AIM Client Components	AIM Server Components
AIM Client Application	AIM Service or AIM Adapter
Client Application Programming Interface (API)	Server API
	Device Support Servers (ADSS)
	Device Directory Service (ADDS)

Figure 4.2 A summary of client and server components in AIM

4.3.1. An Overview of the Structure

AIM is intended to serve as an intermediate middleware layer between online services and mobile devices. Therefore, the infrastructure is composed of AIM Services, AIM Client Applications, and intermediate components. AIM Services are applications that are created with AIM support or AIM adapters to existing applications that do not support the AIM infrastructure. An adapter for a corporate email service is an example for an AIM adapter for an existing application. AIM Client Applications are mobile clients for AIM Services. For example, an email client that connects to an AIM email Service. A typical AIM setting consists of a network of mobile devices running AIM clients and servers that run AIM Services. The intermediate components, AIM Device Support Servers and AIM

Device Directory Service run on separate servers that are connected to a fixed network. In a shared configuration, every AIM service must supply an ADSS and an ADDS. This will be further explained in section 4.4. Figure 4.2 describes the interaction between the different components of AIM. Figure 4.3 shows how the components interact within the AIM network.

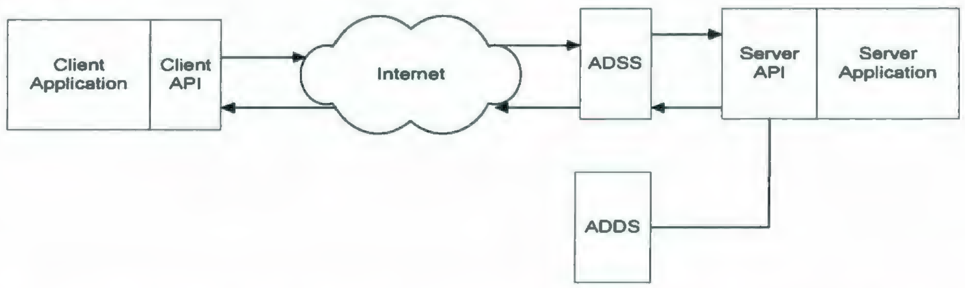

Figure 4.3 The interaction between the components of AIM. As shown, AIM is built as another software layer between the application and the network.

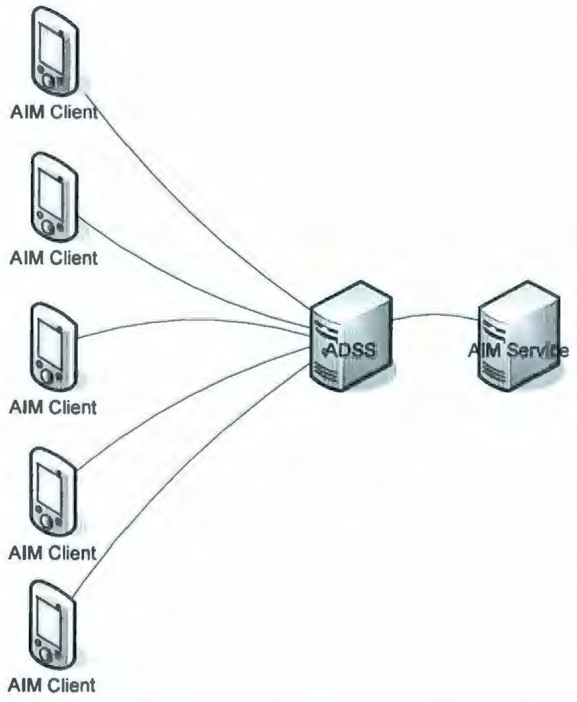

Figure 4.4 (a) The structure of the AIM network – private mode. Only one service supplier.

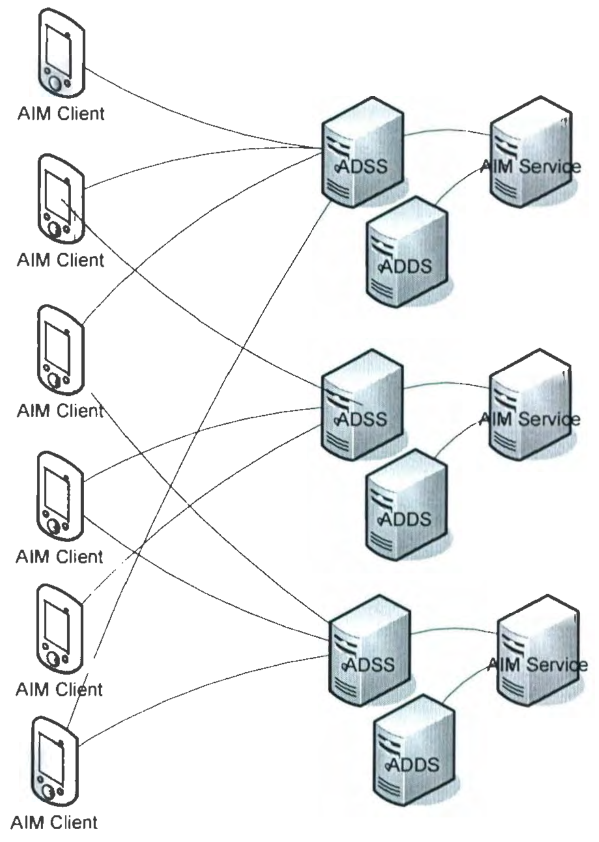

Figure 4.4 (b) The structure of the AIM network – shared mode. Devices can use different servers, private and public.

4.3.2. AIM Services

AIM Services are applications that serve mobile devices and support the AIM infrastructure. There are two types of possible AIM services. The first is an AIM Application that is an application that was written using the AIM Server API for an easy and scalable support for mobile devices. The second type of service is an AIM Adapter, an adapter that connects to an existing application on one side and uses the AIM Server API to extend that application to mobile clients. For most parts of this work, the AIM Applications and AIM Adapters are indistinguishable and will be referred to as AIM Services.

4.3.3. AIM Client Applications

An AIM Client Application is an application that is built for a mobile client and uses the AIM Client API in order to communicate with an AIM Service.

4.3.4. AIM Device Directory Service

The AIM Device Directory Service or ADDS allows AIM Services to locate what ADSS handles a certain client. The ADDS can run off the same machines that run the ADSS and are also arranged in a random way. Every mobile is identified by a unique system device ID. When an AIM Service makes a request to locate a client by its device ID, a peer-to-peer search is made through the ADDS network.

4.3.5. AIM Devices Support Servers

The heart of the AIM infrastructure lies within the AIM Device Support Servers or ADSS. The ADSS role is to serve a mobile device on different tasks related to communicating with the AIM Service and to help the AIM Service to push data to the AIM Client. There are several basic tasks associated with an ADSS: Discovery, Registration, Tunneling, Pushing, Storing and Filtering.

Before a client attempts to register to an ADSS it must first discover one. Discovery of an ADSS can be done in several ways. Preliminary ADSS addresses are configured on the client during the initialization of the AIM Client API or the AIM Client Application. The AIM Client sends an ADSS configuration request to the AIM Service. The response contains addresses of ADSS. These addresses should resolve to ADSS that are supported by the AIM Service. In case of a shared configuration a client can use ADSS that are not affiliated directly to this client. The ADSS in a shared configuration are connected in an ad-hoc manner. In case an ADSS rejects a request, it can still reply with the addresses of other potential servers. In some cases the client is connected to a small closed network and can only use an ADSS that is part of that network. In such a case, the client broadcasts a registration request within the network.

Every client must use one and only one ADSS, even clients that are running several AIM Client Applications. An ADSS can support many mobile clients, depending on its resources. In order to receive support from an ADSS, a client needs to register to an active ADSS. It is possible that a client will have to move to a different ADSS when moving to a different network. An ADSS can support a limited number of clients, or it has a certain capacity of client slots. Therefore, a registration request from a client could be rejected; also, an existing client could be denied further service according to the priorities programmed to the ADSS and the current available slots. In such cases the client will have to search for a different available ADSS. In general, the AIM Service is responsible to supply ADSS slots to its clients. The situation gets more complicated when a client is registered to several unaffiliated services; this situation is dealt in details in chapters 7 and 8. After a device is registered with an ADSS, the server will "represent" the device in front of the AIM Services. When the device moves to a different ADSS the current ADSS can aid in the registration process in order save network and processing time from the device.

The infrastructure tunneling refers to tunneling data into a secured network from a mobile device. A trusted ADSS can serve as a bridge between the secured corporate network and a mobile device. A mobile device that needs to send data to an AIM Service that lies inside a corporate network sends the information to the ADSS; the ADSS, in turn, verifies the device identity and tunnels the information to the AIM Service.

The ADSS also functions in pushing information to the mobile device. It stores and manages information regarding the current network status of the mobile device. When the AIM Service wants to push information to the mobile device it sends it to the ADSS that services that device, and the ADSS determines the best way to push the information to the device. The ADSS notifies the AIM Service if the information was pushed successfully.

The AIM Service could attempt to send data to the mobile device while the device is not connected, switching between network or any other situation that will obstruct the process. In such cases, when the send operation to the device fails, the ADSS stores the request and makes further attempts. Eventually, if after a certain period of time it still continues to fail, it consults the AIM Service on what it would like to do next.

A mobile user can filter the information that it wants to receive from AIM Services. In some cases a user will want to block a service from sending it messages to the mobile device. The ADSS stores filtering information in each device profile and forwards data based on that information.

4.3.6. AIM Components Identification

The components of the AIM infrastructure were introduced in this chapter. A unique identification is necessary in order to govern the interactions between the components. Therefore, each installation of an AIM Client API on a mobile device contains a unique n-bit device ID. Also, every type of AIM Server identifies itself by a unique n-bit server ID. Thus every AIM Service, ADSS and ADDS have an n-bit identifier.

The n-bit identifier is accompanied by an m-bit private key. The n-bit component ID is used as a public key that corresponds to that private key. The provisioning of a unique ID/private key pair is part of the installation process of an AIM component. It is important to ensure that this process will guarantee a unique ID and a secured private key.

4.4. Summary

This part introduced the AIM Infrastructure which is the main focus of this work. An overview of the features and of the structure of AIM was given. The rest of this work will deal with different aspects of the infrastructure. Refer back to the road map that is given in the introduction of this work in order to follow the next chapters.

5. The AIM Network

5.1. AIM Network Topology and Interactions

The AIM network is defined by all the AIM servers and mobile clients that are part of the infrastructure. There are certain rules that determine the interactions between the components of the network, and thus define the topology of the AIM network. The AIM network contains four types of entities: mobile devices (with an AIM client installed), AIM Services, ADSS and ADDS. Denote *D* as the set of mobile devices, *S* for AIM Services, *S1* for the ADSS set and *S2* for the ADDS set. In this chapter we describe the rules and interactions that are part of the AIM network that determine the topology and structure of the network. The formal notation that is described in this chapter will also be used in Chapter 6 in the description of the protocol.

5.1.1. A Device Registers to a Service

This relation is created by a user of a mobile device that registers a device to an AIM Service. This relation is denoted by the set $REG_{s\text{-}d}$. Therefore $(s,d) \in REG_{s-d}$ if $s \in S$ and $d \in D$ and device *d* is registered to service s. Also, $REG_{device}(d)$ denotes all the services that device *d* is registered to and $REG_{service}(s)$ represents all the devices that are registered to service *s*.

5.1.2. The Relation Between a Service and the ADSS and ADDS

In Chapter 4 the necessity of the ADSS and ADDS is explained. It is required that AIM services will supply the ADSS and ADDS servers for the mobile devices. In the private configuration there is only one organization that operates the infrastructure and this issue is trivial. The AIM infrastructure requires that each AIM service will supply one ADSS and one ADDS (one and only one). Note that in the implementation one ADSS/ADDS can correspond to more than one physical server by using a load-balancing scheme, e.g. Figure 4.4. Also the implementation allows that an ADSS/ADDS server can run several ADSS/ADDS instances with different IDs. However, there is a one-to-one correspondence between an AIM Service ID and an ADSS ID and similarly between an AIM Service ID and an ADDS ID.

For $s \in S$ and $a \in S1$ let $ASSOC_{adss}(s) = a$ denote that AIM Service s is associated with ADSS a. As explained A is a one-to-one correspondence and $ASSOC_{adss}^{-1}(a) = s$. Similarly, for $s \in S$ and $b \in S2$ let $ASSOC_{adds}(s) = b$ denote that AIM Service s is associated with ADDS a. In the same fashion, B is also a one-to-one correspondence.

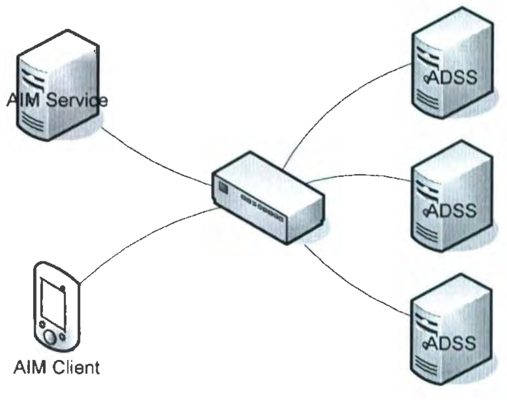

Figure 5.1 More than one physical server corresponds to a single ADSS address.

5.1.3. A Device "Knows" of an ADSS

This relation describes the available ADSS addresses that a device can use when attempting to register to an ADSS. Since ADSS are supplied by the AIM Services, the infrastructure is deigned to allow a device to attempt to register to ADSS that are associated with the AIM Services that the device is registered to. The device is supplied with an ADSS address when it registers to an AIM Service. The AIM Service is then responsible for sending updates to the device if the information changes. An ADSS will not accept a registration request from a device that is not registered to it associated service. For $d \in D$ let $KNOWS_{adss}(d)$ denote the set of ADSS addresses that d has their address. Therefore, for $a \in S1$, then $a \in KNOWS_{adss}(d)$ if and only if there exists $s \in S$, such that $s \in REG_d(d)$ and $ASSOC_{adss}(s) = a$.

5.1.4. A Device is Registered to an ADSS

As explained, an AIM Service uses an ADSS to communicate with a device, and a device needs to be registered to an ADSS. The relation REG_{adss} defines what devices are registered to what ADSS. $(a,d) \in REG_{adss}$ if $a \in S1$ and $d \in D$ and device d is currently registered to ADSS a. Also, $REG_{adss}(a)$ denotes all the devices that are registered to ADSS a. Note that for $d \in D$, $|\{a \mid (a,d) \in REG_{adss}\}| \leq 1$. Thus, a device should be registered to at most one ADSS.

5.1.5. An ADDS "Knows" of a Device

The ADDS are used in order to allow AIM Services to find the ADSS that a certain device is registered to. When searching for an ADSS address the AIM Services initiates a search in the ADDS network. Since the AIM network has no central focal point, there isn't any server that could serve as an authority that will contain all the registration information of all the registered devices. It is guaranteed that only the ADDS server that is associated with the ADSS that the device is registered to stores the needed information for that device. However, it is possible that other ADDS servers will cache this information. Therefore, for $d \in D$ and $b \in S2$, let $KNOWS_{device}(b)$ be the set of devices that ADDS b knows the guaranteed current ADSS registration address. Therefore, $KNOWS_{device}(b) = REG_{adss}(a)$ if the case when there exists $s \in S$ such that $ASSOC_{adss}(s) = a$ and $ASSOC_{adds}(s) = b$.

5.1.6. An ADDS "Knows" About Another ADDS

Since there is no centralized authority for a shared AIM infrastructure, then there is no authority that will construct and maintain the connections from which a peer-to-peer search of the ADDS network can be performed. Instead, these connections will be created by the interactions between the devices and the services. When a device registers to an AIM service, then by default it notifies it about the other services it is registered to. The AIM Service uses his information in order to establish connections between its associated ADDS and the ADDS associated with the other services it was notified of. Therefore, ideally all services that support a certain device will be interconnected. Define $KNOWS_{adds}$ as the set that contains the direct relations between the ADDS. For $b1, b2 \in S2$ then $(b1, b2) \in KNOWS_{adds}$ if and only if there exists a device $d \in D$ such that $ASSOC_{adds}^{-1}(b1) \in REG_{device}(d)$ and $ASSOC_{adds}^{-1}(b2) \in REG_{device}(d)$. Also, for $b \in S2$ let $KNOWS_{adds}(b)$ be the set of all other ADDS that b "knows". Therefore, $KNOWS_{adds}(b) = \bigcup_{d \in REG_{service}(ASSOC_{adds}^{-1}(b))} ASSOC_{adds}(REG_{device}(d))$.

5.2. Summary

This chapter gave a formal description of the AIM network topology. Laying out the objects that participate in the AIM network and the relations between them is important for the understanding the AIM protocol (Chapter 6) and for the discussion about cooperation in AIM (Chapter 7).

6. The AIM Protocol

The essence of AIM is defined in its protocol. The protocol defines the data structures that are exchanged between the different components and the behaviour of each component in each situation. However, it does not define what network mechanisms are used. The AIM protocol is composed of the different operations that are performed by interactions between AIM components. The details of these operations are described in the following subsections.

6.1. Protocol Overview

Before describing the operations of the AIM protocol, device and service identification need to be explained. Every device and each AIM Service is identified by a unique AIM ID. The AIM ID has a length of 128 bits and serves two purposes. The first as mentioned to uniquely identify an AIM Device or AIM Service. In addition, the AIM ID serves also as a public key. Therefore, each AIM Device and AIM Service has a public/private key pair that is used for identification and validation throughout the protocol.

As mentioned in Chapter 4, the infrastructure can operate in two modes, private and shared. In the private mode all infrastructure components are serving one organization. Therefore, all ADSS and ADDS are managed by that organization. In the shared configuration, each organization supplies its own AIM Services and the participating organizations share the ADSS/ADDS network. In such cases, it is intended that each organization supplies ADSS/ADDS according to the amount of users and traffic that its servers generate. There are some differences in the protocol when operating in a private or shared configuration.

The AIM protocol is implemented as part of the AIM Server and Client APIs and the AIM Servers: ADSS and ADDS. Therefore, when developing an application that uses AIM there is no need to implement the protocol details but only to use the AIM APIs.

XML is used for the formatting of the AIM protocol requests. Although it does cause some parsing overhead, it greatly increases the readability of the requests and simplifies the implementation and the development process.

6.2. Protocol Operations

This chapter describes the details of the protocol operations using a pseudo-code notation.

6.2.1. Searching for an ADSS

In order to push data to a device, an AIM Service needs to communicate with the ADSS that is taking care of that device. If the device has communicated before with that service, then it is possible that the AIM Server API has cached the address of the ADSS. Usually that is the case since a device will contact a service at least once before the service will push data to the device. However, even if there is a cached address it could be invalid if the device has changed to a different ADSS. This operation describes the steps that are being taken in order to find the current ADSS.

```
// This method describes the Server API function for
//   searching an ADSS for a unique device ID.
$d$ = Device ID to search
$s$ = The AIM Service that is searching
$C(s)$ = cached device ids and addresses
**SearchForADSS**($d$)
    // Check if an ADSS address is cached for the device
    If exists $(d', address) \in C(s)$ such that $d = d'$
        Return $address$
    // Get the associated ADSS
    $b = ASSOC_{adds}(s)$
    $address$ = **SendADDSRequest**($\{b\}$, $d$, 0)
```

```
        if address not null
                $C(s) = C(s) \cup \{address\}$
                Return address
        Else
                Return "failed"
    End

    // This method prepares and sends an XML request to
    // an ADDS
    B = ADDS that the request is sent to
    d = Device ID to search
    depth = Request depth, starts at 0 and increased every resend
    SendADDSRequest(B, d, depth)
        // Prepare XML and send to the ADDS
        requestXML = PrepareXML()
        // Send request to peers
        For each $b \in B$
                SendAsyncronousRequest(b, requestXML)
        // When the first valid response arrives, return
        // after parsing the XML response
        address = null
        while (address = null) and CountPendingRequests() <> 0
                responseXML = WaitForResponse()
                response = ParseResponse(responseXML)
                address = response.address
        Return address
    End
```

Figure 6.1(a) Algorithm used by the AIM Service API to look for a device ADSS

```
    // This method describes how the ADDS process search
```

```
// requests
$b$ = The ADDS that is performing the search
$C(b)$ = The collection of ADSS addresses stored on this server
$requestXML$ = The XML of the request
ProcessADDSRequest($requestXML$)
    // Parse the input XML
    $request$ = ParseRequest($requestXML$)
    // Read device ID
    $d = request.deviceid$
    // Check if an ADSS address is stored locally
    If exists $(d', address) \in C(s)$ such that $d = d'$
        Return $address$
    // Usually MAX_DEPTH is set on 1 since all service
    // that share a certain device are interconnected
    // It could be set to 2 to overcome broken connections
    $address$ = null
    If $depth$ < MAX_DEPTH
        $address$ = SendADDSRequest( $KNOWS_{adds}(b)$, $b$, $depth+1$)
    Return $address$
End
```

Figure 6.1(b) Algorithm for searching for an ADSS, used in the ADDS.

```
< addsrequest  type=”ADSS Search”>
    <id>Unique request ID</id>
    <info>
        <device>
            <id>Device ID< /id >
        </device>
    </info>
    <depth>Request depth</depth>
```

```
        <aimservice>
                <id>AIM Service ID that originated request<id>
                <address>Address of  AIM Service that originated request </address>
        </ aimservice>
    </ addsrequest >
```

Figure 6.1(c) XML format for the search ADSS request

```
    < addsrequest  type="ADSS Address">
        <id>Unique request ID of originating request</id>
        <info>
                <device>
                        <id>Device ID< /id >
                </device>
                <adss>
                        <id>ADSS ID</id>
                        <address>ADSS address</ address >
                </adss>
        </info>
        <adds>
                <id>The ID of the ADDS that replies</id>
                <address>The address of the ADDS that replies</ address >
        </adds>
  </ addsrequest >
```

Figure 6.1(d) XML format for the search ADSS response

6.2.2. A Device Registers to an ADSS

A device needs to be registered to an ADSS for optimal communication to the AIM Services it is registered to. Registration is a regular part of the AIM infrastructure activity. For example, a device might have to switch from the ADSS server it is registered to due to network limitations. After the device changes to a different network, ADSS can fail or cease to exist and a device might have to look for service elsewhere or due to inactivity device registration might be removed from its current ADSS. This chapter describes the procedures that are taken in order to register a device to an ADSS.

```
// This method describes the device API function for
//  registering an ADSS.
adss = The address of he ADSS that the device tries to register to
RegisterToADSS(adss)
    // Prepare XML for pre-registration request,
    // pre-registration XML contains the device id and
    // device type. Send the pre-registration to the ADSS.
    requestXML = PrepareXML()
    responseXML = SendRequest(adss, requestXML)
    // Parse the response. The response XML should contain
    // a status indicating if the ADSS is willing to
    // accept the request, a request ID that will be
    // used for validating the registration request. The
    // request id is encrypted by the ADSS using the
    // device ID a public key.
    response = ParseResponse(responseXML)
    if response.status <> "ok"
          Return "failed"
    // The request ID is decrypted using the device's
    // private key. The request id is encrypted to
    // that the requesting device is using its own ID
    requestid = Decrypt(response.encryptedid)
    // Prepare XML for the registration request,
    // registration XML contains the decrypted request
    // ID, the current ADSS, an indication whether to
    // copy registration data from old ADSS. It can also
```

```
        // contain updates to registration data that include
        // filtering information and adapter specific info.
        requestXML = PrepareXML()
        responseXML = SendRequest(adss, requestXML)
        response = ParseResponse(responseXML)
        // If request succeeded then return
        if response.status ="ok"
                Return "ok"
        Else if response.status ="failed to retrieve old registration info"
                // In this case registration succeed but it is
                // required to complete registration by sending
                // the full registration info since it was not
                // retrieved from the old server. Add the full
                // registration info to the XML.
                Return UpdateADSSRegistration(adss)
        // Otherwise registration failed
        Return "failed"
End
```

Figure 6.2(a) Client algorithm for registering to an ADSS

```
// This method describes the ADSS process for
// processing pre-registration requests
a = The ADSS
requestXML = The XML of the request
ProcessADSSPreRegistration(requestXML)
        request = ParseRequest(requestXML)
        // This should be a pre-registration request,
        // read device ID, deny a request that from a device
        // that is not registered to the associated service
        d = request.device.id
        $s = ASSOC_{adss}^{-1}(a)$
        if $d \notin REG_{service}(s)$
              DenyADSSRegistration()
        // Check that the device type is supported by this
        // server.
        If not CheckIfDeviceIsSupported(request.device.type)
              DenyADSSRegistration()
        // Use custom algorithm to approve request
        If not ApproveRequest(request)
              DenyADSSRegistration()
        // Request approved, generate an encrypted request id
        // using the device ID as a public key
        encryptedrequestid = GenerateEncryptedRequestID(d)
        // Prepare and send an XML back to the client
        responseXML = PrepareXML()
        SendResponse( requestXML)
    End

    // This method describes the ADSS process for
    // processing registration requests
    a = The ADSS
    requestXML = The XML of the request
    ProcessADSSRegistration(requestXML)
        request = ParseRequest(requestXML)
        // Read the decrypted request id and validate it
        If not ValidateRequest(request.id)
              DenyADSSRegistration()
        // Read the old ADSS address and notify it that the
        // device is registered with a new server
        d = request.device.id
        NotifyTermination(request.oldadss.address, d)
        // Check whether to use the old registration data
        failedToGetRegistrationInfo = false
        registrationXML = null
        If request.useoldregistration = true
                // Read old registration info from old ADSS
                registrationXML = GetRegistrationInfo(request.oldadss.address, d)
                if registrationXML is null
                        failedToGetRegistrationInfo = true
        // Update device registration information on ADSS
        UpdateDeviceRegistration(request, registrationXML)
        // Prepare and send an XML back to the client if
        // failedToGetRegistrationInfo = true then the return
        // status should be modified accordingly
        responseXML = PrepareXML()
        SendResponse( requestXML)
End
```

Figure 6.2(b) ADSS algorithm for processing client registration requests.

```
< adssrequest  type="Pre-Registration">
    <device>
            <id>Device ID< /id >
            <type>Device type</type>
    </device>
</adssrequest >
```

Figure 6.2(c) Pre-registration XML specification.

```
< adssrequest  type="Pre-Registration Reply">
    <encryptedid>Encrypted request ID</encryptedid>
    <status>Request status</status>
```

```
        <reason>If declined, then the decline reason</reason>
    </adssrequest >
```

Figure 6.2(d) Pre-registration XML reply.

```
    < adssrequest  type=" Registration">
        <id>Decrypted request id</id>
        <device>
            <id>Device ID< /id >
            <type>Device type</type>
        </device>
        <adss>
            <id>Old ADSS ID</id>
            <address>Old ADSS address</ address >
        </adss>
        <useoldregistration>Whether to use old registration info</useoldregistration>
        <adapterinfo>
            Adapter specific info
        </adapterinfo>
        <filterinfo>
            <service>
                <id>ID of service that filter info applies to</id>
                <rule>Filter rule</rule>
            </service>
            ...
        </filterinfo>
    </adssrequest >
```

Figure 6.2(e) Registration XML request

6.2.3. A Device Registering to an AIM Service

Much of the registration process of a device to an AIM Service is application specific. The AIM infrastructure provides wrapper procedures that take care of validating the identity of the device, informing the device with the appropriate ADSS information and updating the associated ADDS with the device information. This operation also contains the exchange of secret passwords between the service and the device.

6.2.4. Pushing data to a device

An AIM service can push data to a device by forwarding the request to the ADSS that the device is registered to. The ADSS decides whether to forward the request to the device based on the filter profile of the device. Using a secret password, the device can verify that the data that is received did in fact come from the declared service and the service can get validate that the client received the request. This transaction involves sending binary data. Therefore, the files will be formatted using multipart/related MIME type (RFC 2112).

6.2.5. Sending data to a Service

A device can send data directly to a service. However, in certain situations, a service that is behind a firewall cannot have ports open for receiving information. In order to overcome that without breaching corporate security policies AIM allows the device to send the data to an ADSS that is outside the corporate network. The AIM service can then poll the ADSS as often as possible to check if any new requests or notification arrived from a client device.

6.3. Summary

The AIM protocol provides the backbone that is needed in order for an AIM system implementation. The protocol described was used in a test implementation of AIM used for this work.

7. Modeling Cooperation

In this chapter we present the algorithms that are used by AIM in order to induce cooperation in the public configuration. As mentioned earlier, in the public configuration a device can be registered to several services. Therefore, it is not necessary for a service to supply ADSS slots to all its devices, but the services should share the burden of supplying sufficient ADSS slots. Similar to the works shown in [28, 29] we take a game theoretic approach in order to induce rational participants to cooperate. However, we also make the assumption that some participants might behave irrationally or maliciously and therefore extend the algorithms used in order to avoid the effects of such behaviours. In the following chapters we first discuss the attacks that should be prevented and we specify the requirements from the algorithms in order to be practical. Then we present the system model and the algorithm and investigate where that algorithm results in a Nash equilibrium. We continue with the simulation results of the algorithm. Finally, we show an extension to the algorithm to deal with irrational behaviour.

7.1. Problem Description

7.1.1. Attacks and Misbehaviours that Should be Prevented

The main purpose of the algorithm is to induce cooperation among rational participants who care for their own best interests. Yet, a rational participant might find it beneficial to take advantage of other participants' resources in order to increase its utility. As described in Chapter 4, a service must supply a device with the address of at least one ADSS that it is associated with. However, if a service knows that the device can get ADSS support from a different service, then it might supply it with a dysfunctional address, or that the service might supply less ADSS slots than is required in order to reduce its costs. On the other hand, a service that will supply an ADSS slot to any device that requests so, might end up supplying slots to all its registered devices while the other services that the devices are registered to do not contribute anything. Therefore, the algorithm should make sure that all services supply their fair share of ADSS slots. Our assumption is that the fair number of slots should be based on the number of devices that are registered and the average number of services that each device is registered to, since each additional registration generates more traffic. We define the fair amount of ADSS slots to be supplied by each service should be as follows:

$$\text{fair ADSS allocation} = \frac{\text{total number of devices registered to the service}}{\text{average number of services each device is registered to}}$$

Therefore the main goal of the algorithm is to induce rationally behaving services to supply at least the fair amount of ADSS slots and eliminate “free riders”.

However, it is not impossible that some participants will not behave in a rational manner or would have a malicious intent to reduce efficiency and cooperation in the framework. The algorithm should be able to detect such participants. If a misbehaving service is detected then all its traffic should not be processed by ADSS that are associated with the rationally behaving services.

7.1.2. Algorithm Requirements

There are certain requirements that need to be fulfilled in order to make the algorithm feasible. It should not be a burden, performance-wise on the ADSS. The number of devices and other services can be very large. Also, the algorithm will be used extensively to measure every request for service and its overhead must not exceed its benefit. Finally the algorithm should be able to scale regardless on how many devices or services are added to the system. Therefore, it is preferable that the algorithms time or memory complexity will not be dependant on the total number of devices served or on the number of participant services.

Another constraint is the type of data that the algorithm will be able to use. It would be an easier task if it were possible to get an accurate view on every parameter of the system. However, it is not possible for every service to collect every piece of information available in the system. Some of the useful information is too hard to collect or it is private information. There could be information that needs to be collected from other services or from client devices, in such cases there could be a problem with the reliability of the reported information. Therefore the algorithm should be designed such that all the necessary information for each service could be collected reliably.

The third requirement is probably the most challenging. It is possible that a participating service or a client device would have malicious intentions of hindering the execution of the algorithm. Such cases should be detected and excluded from any activity in the infrastructure. The requirement in this case is that the algorithm would be tamper proof to any such attempts. Such a requirement is hard to fulfill.

7.2. System Model

7.2.1. Basic Definitions

The system model will formalize the definitions of the system components and the relation among the different components. We consider the set D to be a finite population of devices and the set S a finite population of services. We assume that $|D| >> |S|$. Every device is associated with a set of services. As previously defined, let $REG_{device}(d)$ be defined for every device $d \in D$ as the set of services that d is registered to. Similarly, for every service s, the function $REG_{service}(s)$ is defined as the set of devices the service is associated with. Therefore, the number of devices that are registered to service s is $|REG_{service}(s)|$.

The "fair share" of devices that should be supported by a service s is determined by total number of devices registered to s, $|REG_{service}(s)|$, and the average number of services that each devices that is registered to s is registered to. This average is determined by the function $AVG(s)$ that is calculated as following:

$$AVG(s) = \frac{\sum_{d \in REG_{service}(s)} |REG_{device}(d)|}{|REG_{service}(s)|}$$

The "fair share" of devices that each service s should support is denoted by:

$$FAIR(s) = \frac{|REG_{service}(s)|}{AVG(s)}$$

The function $SUPPORTED(s)$ denotes the number of devices that service s is actually willing to accept.

The system operates in discrete time slots. In each slot any device might leave the ADSS that currently handles its traffic. The real life reasons that parallel to a device leaving a ADSS in the system model vary. It could be a representation of inactivity in a device, rejection of a device, a device that switched to a connection that does not allow it to use the same ADSS or other reasons. It could also be due to a device that is leaving the system. However, in the system model that is presented here the number of devices in the system remains constant. It is regarded as if the number of devices that leave the system is balanced by the number of devices that join the system. The probability for any device to leave the ADSS it is currently registered with is denoted by P_d. A device that is not registered to an ADSS at the beginning of a time slot will try and register to one. It will try and register until it is accepted. Only one registration attempt can be made in one time slot.

The algorithm that we suggest in this chapter determines whether a service should accept or reject an acceptance request from a device. The service uses acceptance rates information as parameters to base its decision on. The acceptance rates that are used are the service's own acceptance rate and the average acceptance rate for all requests made by all devices that are registered to the service. The former rate can be viewed as how "nice" has the service been to its devices and the latter can be viewed as how "nice" the other services have been to devices registered to this service. The acceptance rate of the service is calculated to be the number of requests the service accepted up to time slot k divided by the number of total requests the service received up to time slot k. Let $\mathrm{Accepted}_s(k)$ denote the number of requests that service s accepted up to time k and let

$\mathrm{Recieved}_s(k)$ denote the total number of request that server s received. Then the acceptance ratio of *s* at time slot k will be denoted as:

$$\psi_s(k) = \frac{Accepted_s(k)}{\mathrm{Re}cieved_s(k)}$$

The average acceptance rate for all devices that are registered to service *s* will be calculated by averaging the acceptance rate for all devices that are registered to *s*. For each device $d \in D$, $Granted_d(k)$ denotes the number of granted registration requests for this device up to time slot k. $\mathrm{Re}quested_d(k)$ denotes the total number of request that where made by *d*. Similarly to $\psi_s(k)$, then $\varphi_s(k)$ denotes the ratio of granted requests to total requests:

$$\varphi_s(k) = \frac{\sum\limits_{d \in REG_{service}(s)} Granted_d(k)}{\sum\limits_{d \in REG_{service}(s)} \mathrm{Re}quested_d(k)}.$$

7.2.2. Utility Function

The utility function measures the gain of each participant at a particular time slot *k*. If comparing to a market scenario, then the utility measures the profit of each merchant. There are two factors that are considered to contribute to the utility of the participants. The first factor is the number of resources that are used, the more resources that are used the lower the utility is. The second factor is the level of user service, the higher the service level the higher the utility. The amount of resources in this case is the number of devices that an ADSS is willing to accept. The level of user service is measured by the ratio of granted requests to total requests. The utility function can be adjusted by two constants uc_1 and uc_2. The constants adjust the value of each of the factors that contribute to the utility and unless otherwise stated both are equal to 1. Formally, for every $s \in S$ then $UTILITY_s(k)$ describes the utility of *s* at time slot *k*:

$$UTILITY_s(k) = uc_1\left(1 - \frac{SUPPORTED(s)}{FAIR(s)}\right) + uc_2(\varphi_s(k))$$

The value $SUPPORTED(s)$ can range between 0 and $FAIR(s)$, $\varphi_s(k)$ can range between 0 and 1.

7.3. Algorithm Description

7.3.1. A Simplified Scenario

In this chapter we put some constraints on the system model in order to better explain some theoretical concepts. In the simplified model there are $|D|$ devices and $|S|$ servers, however every device is registered to each and every one of the services. In addition, when randomly choosing a service to register to it is assumed that the choices are always perfectly distributes, thus if in one turn there are x requests the each server receives exactly $\frac{x}{|S|}$ requests. Also it is assumed that P_d is equal for all $d \in D$ and that if there are x registered devices at a given time slot then exactly $P_d x$ devices will have to switch that turn. Clearly in such a scenario $FAIR(s_2) = FAIR(s_1)$ for every $s_1, s_2 \in S$ and it will be referred to simply as FAIR.

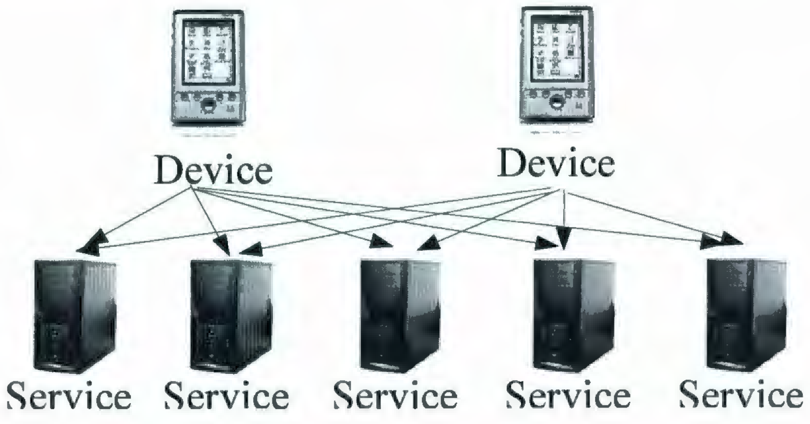

Figure 8.1 Simplified problem.

7.3.2. Rational and Pareto Optimal Operating Point

Pareto efficiency, or Pareto optimality, is a central concept in game theory with broad applications in economics, engineering and the social sciences. In a game, a change that can make at least one individual better off, without making any other individual worse off is called a Pareto improvement: an allocation of resources is Pareto efficient when no further Pareto improvements can be made. In the system that is described in this work, a Pareto efficient point would be a state of the system where it is not possible to increase the utility of any participant without reducing the utility of another.

Theorem 7.1: The system is Pareto efficient if every service is accepting exactly FAIR devices.

Proof for theorem 7.1: If every service is accepting FAIR devices then under the constraints mentioned in subsection 7.3.1 then at any given time slot k $\varphi_s(k)$ is maximal and equals to 1 for all $s \in S$. Also, $\frac{SUPPORTED(s)}{FAIR}$ equals 1 for all services and therefore $UTILITY_s(k) = 1$ for all services. Since it is not possible for any service s to increase its utility by increasing $\varphi_s(k)$ then it must increase its utility by decreasing its $SUPPORTED(s)$ value. By doing that, it will have to deny requests made by devices, thus reducing the utility of all other services. Therefore the system is Pareto efficient. QED.

7.3.3. The Distributed Tit for Tat Algorithm

This chapter describes the policy that ADSS use in order to determine whether to accept or reject a registration request from a device. The idea of the proposed algorithm is that it should encourage all rational participants to allocate resources in fair way, that is to have $SUPPORTED(s) = FAIR$ for every rationally motivated server s. The idea of the algorithm is that no server will accept more than a $FAIR$ number of devices. However, if other services are not cooperating then it will reduce its resource allocation. In order to determine whether to reduce its resource allocation it will test its own acceptance ratio against the acceptance ration of the other services. Figure 8.2 shows the algorithm description. $RESGISTERED(s)$ denoted the number of devices that are registered to s.

If $RESGISTERED(s) \geq SUPPORTED(d)$ *or* $\psi_s(k) > \varphi_s(k)$ *then Reject*

Else Accept

Figure 7.2 Algorithm to decide whether to accept or reject.

The following theorem states the condition in which the algorithm reaches a Nash equilibrium.

Theorem 7.2 If all participants apply the algorithm as described in figure 7.1 then the system will reach a Nash Equilibrium if $\frac{uc_1}{uc_2} \leq 1 - \frac{P_d|D| - P_d FAIR}{P_d|D| - P_d FAIR + FAIR}$.

Proof for theorem 7.1: The theorem states that the level of user service should be valued higher than the amount of resources used for the system to reach equilibrium and it gives and upper bound for this ratio (this is not a tight bound!). The proof needs to show that no service s can achieve a higher utility by reducing $SUPPORTED(s)$ to less than FAIR. Assume that for some service $s \in S$, $SUPPORTED(s) = FAIR - r$ for some $0 \leq r \leq FAIR$. Then the utility for that service will be:

$$UTILITY_s(k) \leq uc_1\left(1 - \frac{FAIR - r}{FAIR}\right) + uc_2(\varphi_s(k)) =$$

$$uc_1\left(1 - \frac{FAIR - r}{FAIR}\right) + uc_2\left(\frac{P_d FAIR(|S| - 1) + P_d(FAIR - r)}{P_d FAIR(|S| - 1) + P_d(FAIR - r) + r}\right) =$$

$$uc_1\left(1 - \frac{FAIR - r}{FAIR}\right) + uc_2\left(\frac{P_d|D| + P_d r}{P_d|D| + P_d r + r}\right)$$

it is given that $uc_1 \leq 1 - \frac{P_d|D| - P_d FAIR}{P_d|D| - P_d FAIR + FAIR}$ and that $uc_2 = 1$.

Therefore,

$$UTILITY_s(k) \leq \left(1 - \frac{P_d|D| - P_d FAIR}{P_d|D| - P_d FAIR + FAIR}\right)\left(1 - \frac{FAIR - r}{FAIR}\right) + \left(\frac{P_d|D| + P_d r}{P_d|D| + P_d r + r}\right)$$

For $0 \leq r \leq FAIR$, the maximum value $UTILITY_s(k)$ can reach is 1. It reaches this value when $r = 0$, i.e. cooperation or when $r = FAIR$, i.e. complete defection. However, it cannot exceed 1 therefore if all participants are cooperating, then one cannot improve his utility by not cooperating. Therefore if the utility function is as defined by the Theorem 7.2 then the system reaches a Nash equilibrium. QED.

To emphasize, since lower cooperation rates by one participant will reciprocate in lower cooperation rates from others then the high utility value is not likely to be maintained when one does not cooperate. Figure 7.2a-c illustrates the possible utility values for a defector. The x axis is the possible values of r and the y axis is the utility values.

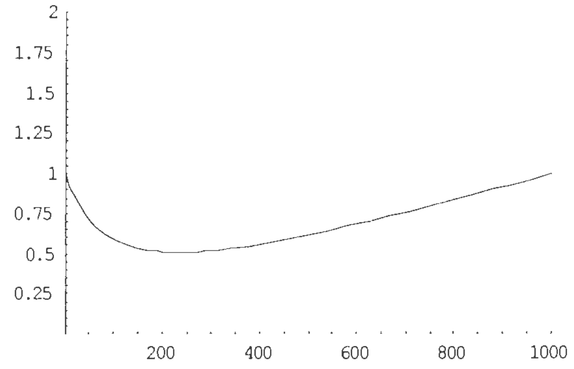

Figure 7.2a $uc_1 = 1 - \dfrac{P_d|D| - P_d FAIR}{P_d|D| - P_d FAIR + FAIR}$

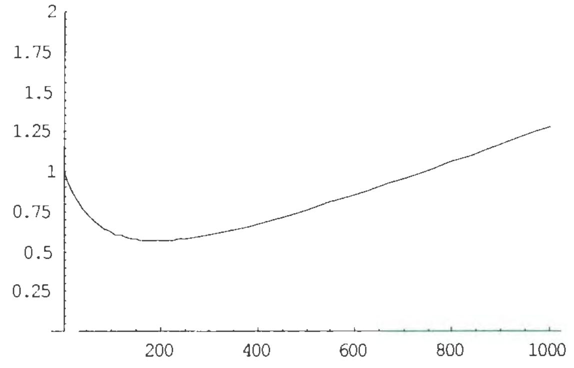

Figure 7.2b $uc_1 > 1 - \dfrac{P_d|D| - P_d FAIR}{P_d|D| - P_d FAIR + FAIR}$

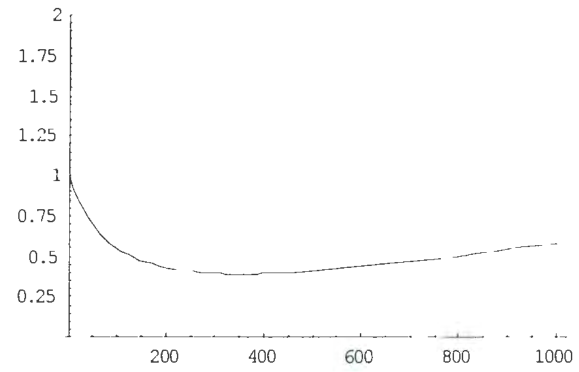

Figure 7.2c $uc_1 < 1 - \frac{P_d|D| - P_d FAIR}{P_d|D| - P_d FAIR + FAIR}$

7.3.4. Algorithm Simulations

Algorithm simulations were done both for the restricted simplified problem and with the restrictions removed. Details about the simulation implementation and environment are described in Chapter 8. In both cases simulation clearly showed that isolated services that did not contribute a FAIR amount of slots increased dramatically the amount of requests that were denied, thus rendering non-cooperation to be unprofitable. It is notable that in the unrestricted scenario the effects of non-cooperation were less dramatic and slower to propagate than in the simplified scenario.

The first simulation was of the simplified scenario with $|D|$ devices and $|S|$ servers where every device is registered to each and every one of the services. A device chose a service to register to by using a random function; therefore the device choices were nicely distributed. The simulated servers were selected to either be in a cooperative mode, thus accepting registration requests, or non cooperative. All cooperative servers employed the distributed tit for tat algorithm. The dependant variable was the parameter measured to evaluate the effectiveness of the algorithm. It was calculated as ratio of successful device registration requests over total device registration requests made. The independent variable was the percentage of servers in the system that are in cooperative mode. Figure 7.3 summarizes the results of this simulation.

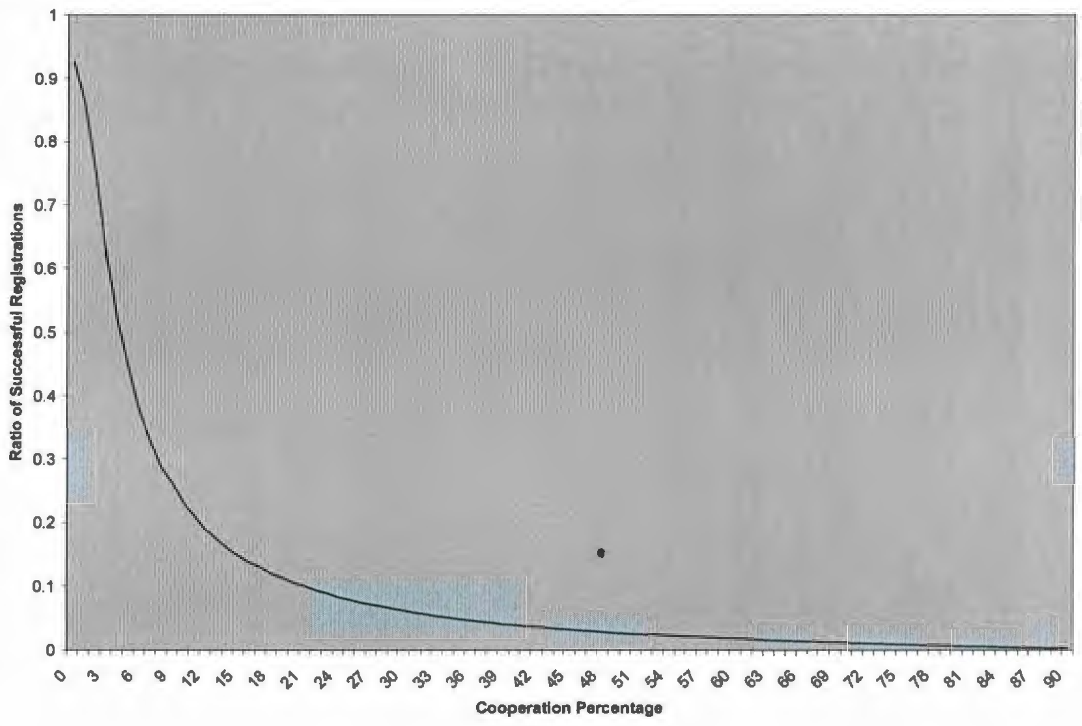

Figure 7.3 Simulation of the simplified problem. The simulation demonstrates that the ratio of successful quickly drops when there are servers that are not cooperating.

The first simulation was of a more realistic scenario. There were still $|D|$ devices and $|S|$ servers; however, each device registered to a random set of services. The simulation results where similar to the first one as shown in figure 7.4.

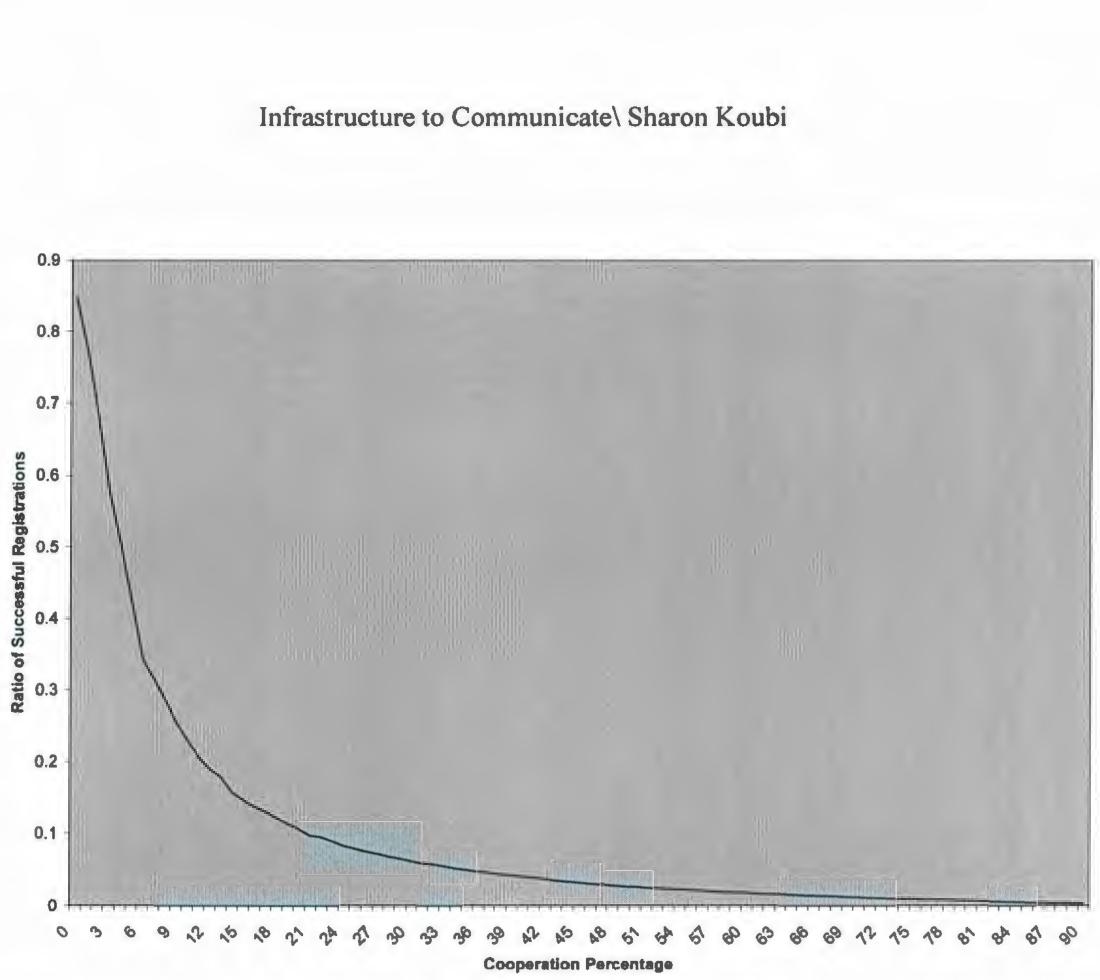

Figure 7.4 Simulation of the more realistic situation. The cooperation ratio was a little lower than in the simplified situation simulation, yet the results were still very similar to the simplified situation simulation.

7.4. *Handling Irrationality*

It is possible that a service that is participating in the network will deliberately not cooperate in order to decrease the level of cooperation in the network and hurt the system performance. Such cases are handled in AIM in a "semi-automatic" manner. Each server can collect information on the other services that are in its neighbourhood. Using reports from registered devices, service s_1 can estimate the acceptance rate of a neighbouring service, s_2. If it is estimated that $\psi_{s2}(k) << \psi_{s1}(k)$, then the system notifies the administrator of s_1. The administrator can then decide to ban server s_2. Therefore transmissions for service s_2 will not be processed and devices that are registered to s_1 will be notified not to attempt to use s_2 ADSS.

7.5. Summary

We have demonstrated that by using a simple and computationally cheap algorithm it is possible to protect the AIM system against ‘selfish’ parties. This algorithm was used in the implementation of AIM that was created for this work.

8. The Implementation of AIM

This chapter describes the software implementation and design of the AIM infrastructure. The implementation of the server was done in Java. Two client versions where made, a Java version for testing and simulations and a C++ version that was written for the Symbian 7.0 OS. A simulation system which automated the system simulation runs was also implemented.

The implemented infrastructure version, which is labeled 0.2, contains a basic server and client implementation. The server implementation is basic and does not contain the AIM server programs ADSS and ADDS described in previous chapter. The installation and configuration modules were not implemented and theses parts need to be done manually.

8.1. The AIM API

The AIM API is a set of interfaces and classes that can be used to develop applications based on the AIM infrastructure. The Server API was implemented in Java. The client API was implemented in Java for the Java client and in C++ for the Symbian client. The goal of the API desing was to keep them as minimal and simple to use as possible. Connection details are not visible through the API and are managed by the infrastructure using the configuration information.

8.1.1. Common API

This is a set of interfaces that is used by the server and client APIs.

AIMAddress
Represents an AIM server or client that a message can be sent to
GetId() – returns the client or server ID.

AIMMessage
A message to be sent from a client to a server or from a server to a client. Implementation

needs to be derived from CAIMMessage.
ToXml() – returns the XML message.

AIMRegistrationMessage
A registration message to be sent from a client to a server. Derived from AIMMessage.

AIMEventManager
A interface to receive callbacks by the client and server informing of system events
Regsitration() - A registration request event (usually used only on the server side).
Message() – A message, other than a registration.

8.1.2. Server API

The server API is the interface that a server application can use in order to use the AIM infrastructure. Through the server API it can communicate with client applications.

AIMServer
The interface to be used by an AIM service application.
Start() – Start the aim server.
Stop() – Stop the aim server.
GetClientManager() – Get the AIMClientManager instance.
SetEventManager() – Set the interface for callbacks.

AIMClientManager
The interface to be used by an AIM service application to manage clients.
Add() – Add a client, confirm registration.
Remove() – Remove a client, deny registration.
Message() – Send a message to a client.
GetClientInfo() – Get the client profile information.

8.1.3. Client API

The client API is the interface that a client application can use in order to use the AIM infrastructure. Through the client API it can communicate with an AIM service.

AIMClient
The interface to be used by an AIM client application.
Start() - Start the aim client.
Stop() - Stop the aim client.
GetServiceManager() – Get the AIMServiceManager insance.
SetEventManager() - Set the interface for callbacks.

AIMServiceManager
The interface to be used by an AIM client application to manage server registration and messaging.
Register() – Request to register to a service.
Message() – Send a message to a service.
GetServiceInfo() – Get the server profile.

8.2. Common Module

The common module contains the classes that are shared between the client and server. It contains the classes that represent the XML messages, the common API interfaces and the classes that implement these interfaces.

8.3. AIM Server

The AIM server module was implemented in Java. It contains the implementation for CAIMServer which implements AIMServer and provides access to the AIMServer. It also contains the classes that deal with implementing the "Distributed Tit-for-Tat" algorithm and the server configuration. In order to simplify simulation the server can run in two modes, regular mode where the server creates new threads and simulation mode in which the server runs only in the calling process thread and does not open a new thread.

8.4. AIM Client

The AIM client module contains the implementation for the CAIMClient which implements the AIMClient interface. It also contains the client configuration implementation and the classes that serve s interface to different networks types. The AIM client was implemented in Java and in C++ for Symbian 7.0.

The Java implementation was created for testing purposes and does not deal with switching to different types of networks. Similarly to the server the Java client can run only in the calling process thread and does not open new threads.

The Symbian client is deigned to be aware of whether a cellular network or a WiFi network is used.

8.5. AIM Simulation System

The simulation system was designed in order to simulate a large number of AIM servers and clients in collaboration. It contains a simple demo client Java application and a simple demo Java server application that uses the AIM APIs.

The demo server application is runs the AIM server and accepts new clients according to the "Distributed Tit-for-Tat' algorithm. The simulation system can override this in order to simulate a situation where there are servers that are not cooperating. The configuration information is supplied by the simulation system.

The demo client application is a Java application that uses the AIM client Java API. It can make registration requests to AIM services and send messages through the AIM services that it is registered to. The list of available services and other configuration information is supplied by the simulation system. The simulation system also determines the rules by which a client selects to how many and which servers to register to.

Simulations were run with up to 10000 clients and 100 services operating in parallel. The simulations ran on a PC with a AMD 64 Athlon and 2GB memory running Windows XP.

8.6. Summary

The AIM infrastructure version 0.2 is a preliminary implementation of the AIM infrastructure. A more complete implementation would include a full implementation of the ADDS and ADSS servers as well as automation of the configuration process and modules that would allow simple interface for common mobile applications.

9. Summary

As envisioned in [19] and [27], handheld devices are becoming more and more ubiquitous. The market for mobile software is rapidly growing and tools for facilitating this revolution are in need. I believe that some new concepts need to arise in order to efficiently develop software in an increasingly mobile universe and that there is a place for an infrastructure that would help developers to create mobile applications more easily and to smoothly integrate them with established corporate software.

9.1. Key Points of This Work

The focus of this work is AIM, an infrastructure that will provide services that will make developing and adapting applications for mobile devices easier and smoother. The approach that is taken in designing AIM is the middleware approach, based on general and mobile concepts of middleware, as similarly seen in [4]. The description of the AIM infrastructure includes the rules that determine the interactions between the components of the AIM network, and thus determine the topology of the AIM network and a detailed description if the AIM protocol. The last two chapters deal with inducing cooperation in the system.

9.2. The Benefits of a System like AIM

AIM could make unique mobile characteristics such as connection details, network identification and network problems transparent and in addition could serve as a connection point for mobile devices to various services and protocols. Whatever system is used, the key argument that is made in this work is that efficiently developing multiple, elaborate mobile applications need to be done on top of a mobile middleware layer that will take care of many of the technical details. Such middleware applications exist for desktop software; however, there is a need for a specialized platform for mobile implementations.

9.3. Proposals for Future Work

An obvious continuation of this work would be a full implementation of AIM and using it to adapt several existing applications for mobile devices. If such a system is implemented then the main obstacle for its commercial success would be to integrate it with some of the commercial mobile operating systems available. I envision that such a system can be useful for medium to small software companies that could use it as a tool on top of the operating system and standard development tools.

There are other theoretical aspects of AIM that can be explored as well. There are a couple of directions that would be particularly interesting in the context of an infrastructure such as AIM. One would be the handling of transactions in a mobile middleware system. In order for such a system to be reliable, transactions need to be an integral part of it. Another would be the issue of assuring privacy and anonymity. This area of research is especially relevant for the public configuration of the system in order to prevent from services the opportunity to match and possibly abuse private user information.

About the Author

The author, Sharon Koubi, has been a student, programmer and researcher in the field of computer science for more than ten years. He graduated from Memorial University in 2004 with B.Sc. (Honours) in computer science. In addition to his research with wireless networks he was also involved in researching combinatorial designs and proteomics. Currently, he is a third year medical student at Memorial University of Newfoundland and lives in St. John's with his wife and their two children.

Journal publications by the author

1. S. Koubi, M. Mata-Montero, N. Shalaby, Using Directed Hill-Climbing for the Construction of Difference Triangle Sets, IEEE Transactions on Information Theory 51(1): 335-339, 2005.
2. S. Koubi, N. Shalaby, The Combined Use of a Genetic Algorithm and the Hill-Climbing Algorithm to Find Difference Triangle Sets, accepted to the Journal of Combinatorial Mathematics and Combinatorial Computing in 2007.
3. H. Paradis, T. Islam, S. Tucker, L. Tao, S. Koubi, R. Gendron, Tubedown Associates with Cortactin and Controls Retinal Endothelial Cell Permeability to Albumin, accepted to the Journal of Cell Science in 2008.

References

1. F. Andrè and M.T. Segarra, *A Generic Approach to Satisfy Adaptability Needs in Mobile Environments*, 33rd Hawaii International Conference on System Sciences (HICSS'00), Maui, Hawaii, January 2000.

2. O. Angin, A. T. Campbell, M. E. Kounavis and R. Liao, *The Mobiware Toolkit: Programmable Support for Adaptive Mobile Networking*, IEEE Personal Communications Magazine, Special Issue on Adaptive Mobile Systems, Vol.5, pp. 32-43, August 1998.

3. K. Arnold, B. O'Sullivan, R. W. Scheifler, J. Waldo, and A. Wolrath, *The Jini Specification*, Addison-Wesley, 1999.

4. P. Bellavista, A. Corradi, R. Montanari and C. Stefanelli: *Dynamic Binding in Mobile Applications: A Middleware Approach*, IEEE Internet Computing, Vol. 7, pp. 34-42, 2003.

5. M. Bhide, P. Deolasse, A. Katker, A. Panchgupte, K. Ramamritham, and P. Shenoy, *Adaptive Push Pull: Disseminating Dynamic Web Data*, IEEE Trans. Computers, special issue on quality of service, pp. 265-274, 2002.

6. S. Buchegger, J.-Y. Le Boudec, *Performance Analysis of the CONFIDANT Protocol (Cooperation of Nodes: Fairness In Dynamic Ad-hoc Networks)*, MobiHoc, pp. 226-236, June 2002.

7. M. Buddhikot, G. Chandranmenon, S.-J. Han, Y.-W. Lee, S. Miller, and L. Salgarelli, *Integration of 802.11 and Third-Generation Wireless Data Networks*, Proceedings of INFOCOM, Vol. 1, pp. 503-512, March 2003.

8. L. Buttyan and J.P. Hubaux, *Enforcing Service Availability in Mobile Ad-Hoc WANs*, IEEE/ACM Workshop on Mobile Ad-hoc Networking and Computing, pp. 87-96, August 2000.

9. L. Buttyan and J.P. Hubaux, *Stimulating Cooperation in Self-Organizing Mobile Ad-hoc Networks*, ACM/Kluwer Mobile Networks and Applications, Vol. 8, pp. 579-592, 2003.

10. Y.-F. Chen, H. Huang, R. Jana, T. Jim, M. Hiltunen, R. Muthumanickam, S. John, S. Jora, and B. Wei, *iMobile EE - an enterprise mobile service platform*, ACM Journal on Wireless Networks, Vol. 9, pp. 283-297, 2003.

11. L. Kleinrock, *Breaking Loose*, Communications of the ACM, Vol 44, pp 41-45, 2001.

12. L. Kleinrock, Kleinrock on Nomadic Computing, Ubiquity, Volume 6, Issue 25, July, 2005

13. Lee et al., Backbone Construction in Selfish Wireless Networks, Proc. of the 2007 ACM SIGMETRICS, pp. 121-132, 2007.

14. S. Marti, T.J. Giuli, K. Lai, M. Baker, *Mitigating Routing Misbehavior in Mobile Ad-hoc Networks*, MobiCom, pp. 255-265, August 2000.

15. C. Mascolo, L. Capra and W. Emmerich, *An XML-based Middleware for Peer-to-Peer Computing*, Proceedings of the International Conference on Peer- to-Peer Computing (P2P2001), pp. 69-82, August 2001.

16. C. Mascolo, L. Capra, W. Emmerich, *Mobile Computing Middleware*, NETWORKING Tutorials, pp. 20-58, 2002.

17. M. Musolesi, C. Mascolo, S. Hailes, Adapting Asynchronous Messaging Middleware to Ad Hoc Networking, Proc. of the 2nd workshop on Middleware for pervasive and ad-hoc computing, pp. 121-126, 2004.

18. I.G. Niemegeers and S.M. Heemstra de Groot, *Research Issues in Ad-Hoc Distributed Personal Networking*, Special issue of Wireless Personal Communication, Vol. 26, pp. 149-167, 2003.

19. C. Nika.,et al., *The Challenge of Mobile IP in Wireless Networks*, 4th Wireless Personal Multimedia Communication (WPMC 2001) International Conference, 2001.

20. C. A. Nika, D. D. Vergados and M. Theologou, *Mobile IP: A Challenge in the Mobile World, Wireless IP and Building the Mobile Internet*, Artech House, pp. 232-242, 2003.

21. Oh et al., A Programming Environment for Ubiquitous Computing Environment, ACM SIGPLAN Notices, Vol. 42, Issue 4, pp. 14-22, 2007.

22. C. Perkins, *Mobile IP*, IEEE Comm., Vol. 35, pp. 84-99, 1997.

23. R. Prasad, and K. Larsen, *3G Networks and Standards, Wireless IP and Building the Mobile Internet*, Artech House, 2003.

24. K. Raatikainen, H. Christensen, T. Nakajima, *Application Requirements for Middleware for Mobile and Pervasive Systems*, ACM SIGMOBILE Mobile Computing and Communications Review, Vol. 6, pp. 16-24, 2002.

25. T. S. Rappaport, *Wireless Communications*, Prentice Hall, 2002.

26. M. Roman, *Ubicore: Universally Interoperable Core*, http://www.ubi-core.com. (last accessed July 2006).

27. J. Roth, C. Unger, *Using Handheld Devices in Synchronous Collaborative Scenarios*, Second International Symposion on Handheld and Ubiquitous Computing 2000 (HUC2K), pp. 187-199, September 2000.

28. V. Srinivasan, P. Nuggehalli, C.-F. Chiasserini, and R. R. Rao, *Cooperation in Wireless Ad-Hoc Networks*, in Proc. IEEE INFOCOM, pp. 808-817, 2003.

29. A. Urpi, M. Bonuccelli, and S. Giordano, *Modelling Cooperation in Mobile Ad-Hoc Networks: A Formal Description of Selfishness*, Proceedings of Modelling and Optimization in Mobile, Ad-hoc and Wireless Networks (WiOpt), pp. 56-67, March 2003.