USING POLICY TO CONTROL DATA SYNCHONISATION IN MIDDLEWARE FOR AN AD-HOC MOBILE NETWORK

A THESIS SUBMITTED TO

THE UNIVERSITY OF KENT

IN THE SUBJECT OF COMPUTER SCIENCE

FOR THE DEGREE

OF DOCTOR OF PHILOSOPHY

By

Vorapol Jittamas

May 2007

Abstract

Many devices are now using wireless communication, either by using base stations or by forming ad-hoc networks, but this style of mobile platform increases the probability of disconnection because of loss of network access. Replication of shared data is one way to increase data availability in a mobile environment, but leads to the problem of inconsistent copies of data after periods of disconnection, and so requires some means of data synchronisation. Therefore, there is a need for an approach that can provide synchronisation support for various types of wireless application.

This thesis investigates how policy can be used to resolve problems of data conflict in a way that can be tailored to meet the needs of different types of application in different situations. A middleware has been created to investigate the problem. This middleware supports the sharing of data in a wireless environment using a tuple-space paradigm. It provides data caching which will enable individual devices sharing data to work with a single virtual space. This method does not guarantee that data in a local device will always be up-to-date but it does provide some level of information for applications that need the data for decision making while disconnected from other devices.

A set of policies is maintained within the middleware, and these policies are used to express a wide range of synchronisation options to restore the consistency of the data after periods of disconnected operation. This thesis tests several policies in a number of scenarios based on different wireless applications. These involve a range of context information to support the policies in decision making. The thesis also includes an investigation into how policy can be defined independently for each device and the implementing of a synchronisation process that can support the process. Such environments require a more complex synchronisation process that can detect and resolve any policy conflict. The performance of the system has been measured and analysed in term of costs and benefits offered to applications. It shows that it is possible for such a system to have performance sufficient for applications that are not especially time sensitive in environments where the periods of connection are long enough for the system to progress its synchronisation.

Acknowledgements

My first and foremost thank goes to my supervisor, Professor Peter Linington, who gave me an opportunity to write this thesis. He gave me a lot of valuable suggestions and comments during the time I built the system and wrote this thesis and arranged financial support for me during my Phd. studies. Special thanks to my supervisor panel - Ian Utting and Gerald Tripp - for a lot of discussions of the system.

Many thank to Su-Wei Tan, my colleague, who gave a lot of suggestions especially in choosing and implementing the system's Event Distribution Layer using a multicast protocol. Thanks to a lot of friends in the University of Kent Computing Laboratory who provided a great friendly environment for the three years I've been studying here. These includes Ben, Rodolfo, Gift and many others.

A lot of thanks to my parents who always gave me a lot of support and encouragement to finish this thesis. Lastly, thanks to a lot of my friends here in Canterbury who always gave me respite from the stress involved in doing this thesis.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Setting the Scene

Improvements in wireless networks and small device processing power have increased mobile computing usage recently. New wireless network and telecommunication standards allow devices to communicate using higher speeds with longer range and more reliability. Devices such as mobile phones, PDAs, or laptops are getting smaller with higher processing power. A number of them now support several types of connection. This situation makes extensive use of mobile computing more attractive.

This new technology gives more flexibility. Mobile computing allows the creation of applications that are hard or impossible to implement while relying on wired connectivity. Examples are advanced forms of mobile computing such as ad-hoc sensor networks where a number of small sensor nodes periodically collect information and communicating with each other using wireless, personal area networks where a user's small personal devices can communicate with each other, and ubiquitous computing where small computing devices are embedded within appliances.

1.1.1 Connectivity Problems

There are a number of obstacles to the widespread use of mobile computing. One problem is that writing a mobile application is generally more complex than writing a static application. For example, compared to general applications that are self-contained, network access is an essential features that characterised a mobile application. This leads to a range of difficult problems such as how to make a connection secure, for example.

Compared to applications operating on an office network, a mobile application has to deal with situations where a network connection does not exist. Moving a device from an office network to a home network causes a disconnection and a re-connection. The application has to be able to operate both in the situations where there is a network connection and in those where there is no network connection.

This could happen when a user uses the device while he is moving.

Furthermore, a mobile application that is designed to operate on a wireless ad-hoc network (see section 2.3.1) has to take into account the unreliable nature of the connection. In this case, there is no control over when a network access ceases to exist. This problem is different from moving the device from one network to another network. In the previous case, a user expects the disconnection before it happens but an ad-hoc network connection could be disconnected at any time.

Moreover, an ad-hoc network introduces a transient disconnection problem. A disconnection from the network does not mean that the network connection is not going to exist for a long period of time. A transient disconnection can occur in the network as a result of movement of the other devices forming the network. This means there is a higher chance that communication in an ad-hoc network will be interrupted than for other communication types.

1.1.2 Data Access Problems

Accessing resources or data over both wired and wireless networks requires a way to find resources. Generally, a resource discovery service is required. Such a service receives registration information including a service description and a service location from other services that wish to announce their existence. An application queries the discovery service to obtain this information which then allows the application to access the resources directly.

Wireless ad-hoc networks add more complexity to resource discovery. In a structured network, the discovery service can be operated on a permanent server. General services such as file stores or print servers are usually located on static devices which register information on the discovery server and then provide services. However, in an ad-hoc network, there is no server that can guarantee to stay connected to the network all the time due to the changing nature of the network. Therefore, service discovery using a single server is not possible.

Apart from dealing with the physical connection itself, a mobile application has to deal with disconnection in terms of resources that are not going to be available while a device is disconnected from the network. During disconnection, data that is not stored in the device cannot be accessed; any services provided by servers on a network cannot be used. In this case, one solution is to store some or all of the information that is going to be needed locally.

Storing or caching information locally can be done in many ways. A number of research projects have looked at how to provide the best data replication technique or caching algorithm, and some of these projects are mentioned in section 2.4.1. However, there is no way to ensure that there is not going to be a cache miss where information is not available offline, unless a user pre-defines all the possible information that is going to be used before a disconnection.

1.1.3 Data Synchronisation Problems

Data caching is normally used to reduce problems arising from being unable to access data during a network disconnection situation. However, there are problems in using a cache. One of the problems is how to synchronise the caches belonging to each user when they are reconnected. During disconnection, if a user is allowed to modify any information in his/her device, there is a chance that more than one user is going to modify the same information in different ways. Data conflict is then going to happen when the two devices are re-connected.

One way to prevent the problem is not to allow a user to modify information that is not pre-defined as modifiable. The simplest way is to give one user an owner token for a particular piece of data so that only he or she has the right to modify the data during a disconnection period. However, this severely limits the system flexibility, since other users cannot modify the data even if they have a good reason to do so.

Another way to resolve the data conflict problem is to do data synchronisation when devices reconnect. In this way, every device is allowed to modify any data in their caches and there is no need for the owner token and its limitation. When devices reconnect, caches on them are analysed for any data conflict, which is generally called a conflict detection process. Then, any conflict in the data will be reconciled, which is normally called a conflict resolution process.

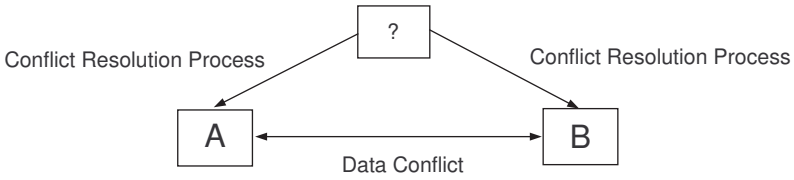


Figure 1.1: Data Conflict and Conflict Resolution 1

These two processes can potentially resolve the differences between caches but introduce their own problems. First, as in figure 1.1, what entity will tell the device resolution processes which data to preserve? Next, if a conflict does happen between the two devices, what is going to be done with data that is rejected during the resolution process?

In the simplest case, it is easy to create a basic conflict resolution engine that contains a fixed algorithm for the resolution process. However, this limits the number of situations where the system can be used. A detection and resolution process that is designed to be used in one application may not be able to be used in another application. Therefore, the engine needs to be flexible enough to understand requirements set by different applications and to tailor its resolution process to meet these requirements. Ideally, the algorithm for deciding the process should be able to receive external input that can control how the process should be done in a particular situation.

Next, similar to the problem of the discovery service discussed above, the resolution engine cannot be located on a specific server, as shown in figure 1.1, because, in an ad-hoc network, devices may require the resolution process from the engine when a route to it does not exist. Therefore, the resolution engine needs to be installed on each device that may require the process, as illustrated in figure 1.2.

Figure 1.2: Data Conflict and Conflict Resolution 2

Installing the engine on each device and allowing the resolution process algorithm to be affected by external inputs leads to another problem. In a situation where different engines try to resolve the same data conflict in different ways, who is to choose the process to be executed? Moreover, higher level resolution processes to resolve the conflict caused by having two resolution engines may also be different between the two services which can then cause a higher level conflict.

1.2 Purpose and Aim of the Project

1.2.1 Purposes and Aims

This project tries to create a simple environment for developing mobile applications that can be used in an ad-hoc network by providing data communication and synchronisation processes based on a tuple-space paradigm (see 2.2). This is to reduce the complexity in writing the applications, because the project provides a single virtual space view to an application.

To some degree, this masks out complexity from accessing remote information and from any data synchronisation when writing a general mobile application. However, another aim of this project is to expose its conflict detection and resolution processes to the user so that he or she can alter the process to suit a particular application.

Communication complexity is reduced by creating a middleware layer that is responsible for data communication and synchronisation. The middleware should be able to provide a number of tuple-space like interfaces to an application so that it can support an application that is built for the tuple-space paradigm with as small a number of changes as possible.

The middleware needs to be able to operate on a mobile ad-hoc network where each device is connected at a peer level and there is no central server responsible for providing any specific service. Connections between devices in such a network are unstable and can be disrupted at any time.

In order to provide a unique virtual space point of view, the middleware is responsible for caching information from each connected device and storing it in a local space. In this way, a user can access and modify the stored information at any time without immediately being interrupted by any disconnection.

To maintain data consistency, the middleware is responsible for detecting and resolving conflicts between different copies of the same tuple. During connection between any two devices, the middleware conflict detection and resolution processes are triggered, and these will analyse tuple information between the two devices and resolve any conflict that exists.

Next, the conflict detection and resolution processes have to be flexible enough to support different types of application operating on the middleware. Its synchronisation engines should allow its tuple synchronisation process to be based on control information provided by any user.

Moreover, the middleware needs to keep track of information both internal and external to a device which can be used to support its tuple synchronisation process. For example, tuple historical information such as the number of times the tuple has been accessed or the time when a tuple was created should be collected by the middleware.

Also, since different applications may require different types of information to support their synchronisation decisions and the middleware may not know in advance what the information is, it should provide an interface that allows the information to be collected and stored in a local space. The information can then be used in helping the synchronisation engine in making its synchronisation decisions.

Next, since a synchronisation engine is attached to each device, there is always a chance that the control files written by different users will be different from each other. The middleware has to provide a way to resolve conflicts between these control files.

Lastly, the middleware needs to provide a way to propagate synchronisation information to other interested devices in an area. Information such as the policy that is used during the synchronisation and the information that is used in making the synchronisation decision should also be propagated. This is to make tuple information on the devices in the area converge and allow the middleware to be able to maintain the unique virtual space view for applications.

1.2.2 Contributions

This project demonstrates that it is possible to use a policy to control a synchronisation process in a tuple space environment by building a middleware prototype that supports tuple space based interactions in a wireless environment. The system is built so that it can support different types of application without a need to modify the system mechanism.

Next, the project investigates a number of aspects required for a policy to make the right decision during a synchronisation process. For example, it investigates the case where different policies are applied in different devices, which will cause a policy conflict during a synchronisation process, and proposes a simple process to resolve the conflict.

This project also looks into how context information can help to make the synchronisation process better, and the cost of using the information, including a situation where improper usage of the context information leads to an incorrect synchronisation decision. Lastly, the project shows the cost of providing the synchronisation service for a tuple space environment. Even though the system helps in extending a virtual space view to a disconnected sub-network, there are costs for providing the service.

1.2.3 Topics not Covered

This project does not claim to resolve every problem discussed above and the middleware built in this project has not been tested to show it operates flawlessly for every application. It will be used primarily to investigate the control of data synchronisation processes using a policy (control file) in a shared tuple space paradigm that can operate over a mobile ad-hoc network.

Due to the time limit on the project, the middleware could not be implemented to include all the features normally present in commercial software. The middleware will contain the functions that are the most important for supporting the tuple communication and synchronisation processes.

The efficiency of the middleware may also not be as good as a commercial product might require (its performance can be seen in chapter 7). It is intended for use in a small cooperative application environment such as sharing a calendar or making room bookings and does not have some of the features available in larger systems. The middleware does not provide any security control such as per tuple access permission control. However, some measures of security are provided by using the Java security policy file [Sun02], which allows a user to define access controls for any entity that is to access a space.

The middleware still has to rely on a single set of higher level policies that can resolve conflict between low level policies. This means a user still does not have total control over how a synchronisation process on his/her device should be done. It will take more resource in order to find a way to provide this total control to a user.

Finally, the middleware needs a device with enough processing power to operate. It has been tested on a laptop computer which has enough processing power to run the Sun's Jini services and JavaSpace. Devices that can support the two services should be able to run the middleware. It is not designed to be used in a low processing power devices such as in a sensor network, or a mobile phone with limited resource.

1.3 Why Policy?

There are several ways for a user to control the middleware's synchronisation behaviour. On one hand, the middleware could provide a number of different synchronisation algorithms, one of which could be selected by the user. These could be provided so that a user was able to select his or her synchronisation algorithm from an interface such as a drop down menu, set of tick boxes, or radio buttons.

On the other hand, a user could be allowed to write his or her own control files using a natural language. In this case, the middleware needs to contain a very complex compiler that can translate from the user's language to a language that the middleware understands.

The first method is too inflexible. A user's choices are limited to a number of synchronisation algorithms pre-defined by the middleware and these algorithms are unlikely to be able to cover every type of application needed by different users. A user cannot introduce a new synchronisation and conflict resolution algorithm that is different from those that are already provided.

Even though the second method gives a user freedom to express an algorithm without any pre-defined limit, it is too flexible and too complicated. It is very hard to create a compiler that can translate from a natural language to a machine understandable language. Even if such a compiler can be created, it is still very hard to control the expressive power available to the user.

These two methods presented above are the two extreme ends of the flexibility scale. In between the two, there are policy languages. These languages allow a user to create a policy file tailored to suit a particular application which provides more flexibility than the pre-defined algorithm approach. Meanwhile, they have a simple enough syntax that a compiler can be created without too much work. Some of the languages even have an existing compiler that can compile from a policy file to a programming language file making it easier to integrate them with the middleware.

A policy language allows a user to control the middleware behaviour at a high level. Comparing this to changing the behaviour by using a programming language, a user does not need to understand every detailed mechanism relating to how the middleware works in order to change it. Moreover, controlling the middleware using a policy separates the user from any change that could happen in the middleware mechanism. As long as it still provides the same set of interfaces to the policy engine, there is no need for a user to edit his or her policy when the middleware is modified.

More importantly, a policy allows the middleware behaviour to be changed on the fly. Changing a system by editing its code requires the system or part of it to be shut down in order to apply the changes. Changing it via altering the system policy does not interrupt the system. A new set of policies can be compiled separately. The system needs only to load the new policies and replace the old set. This project benefits greatly from this feature which allows the middleware synchronisation behaviour to be changed depending on the current situation. For example, the behaviour can be changed depending on the different types of tuple participating in the synchronisation process.

1.4 Thesis Outline

This thesis consists of nine chapters. Chapter 2 provides background knowledge relating to this project. It gives information about the tuple space paradigm which is the basis for the middleware, including

examples of application areas that benefit from the paradigm. It also contains information about wireless networks; especially some basic knowledge concerning mobile ad-hoc networks which offer the environment in which the middleware is intended to be used.

Then, it explains the basic concept of data synchronisation taken from works in the database area and shows a number of previous attemps to provide a data synchronisation service in both a structured and ad-hoc wireless network. It also gives some information about middleware and policy containing examples of existing tuple space paradigm middleware, and a brief introduction to a number of policy models, standards, and languages. There is also a comparison between policy languages in this section.

Chapter 3 gives an idea of types of applications the middleware aims to support. The chapter also shows the process of building and testing the middleware. It gives an explanation of the test framework that is used to test the middleware synchronisation process and a brief introduction to the performance testing process of the middleware.

Chapter 4 explains the JavaSpace For Mobile environment (JSFM) middleware architecture proposed in this dissertation. It gives an overview of the JSFM by describing a number of components used in the middleware. Then, it explains each middleware component in detail starting with the components that are responsible for the middleware supporting processes (e.g. starting up, IP address monitoring and changing). Next, it describes the components that make up the middleware policy engine, and are responsible for making decisions during each synchronisation process. The last section describes the Synchronisation Event Propagation layer which is the layer that is responsible for creating and maintaining the synchronisation event distribution tree providing messaging services between devices in the same area.

Chapter 5 contains information about the policies that are used in the middleware. It starts by describing the main policy components, how are they related to each other, and how they are used in the middleware. Then, it discusses the implementation of the components in the middleware. Next, the chapter describes how the decision making process and the policy enforcement process are done. Lastly, the chapter contains information concerning how the middleware allows user-defined information to be gathered using the context gathering agent interface.

Chapter 6 discusses how the synchronisation process in the middleware is done. The chapter is divided into two main sections. The first section illustrates how a synchronisation process is done where every device uses the same set of policies and the second section discusses policy conflicts occurring during the synchronisation process when different devices use different local policies. Next, the chapter introduces a number of methods that can be used to resolve these conflicts. The section also discusses how to make data on different devices converge using direct and indirect methods.

Chapter 7 shows the result of testing the middleware. It compares access times for three general tuple space operations between an application accessing a space directly and accessing it via the middleware. Next, it shows results from the tree building test process. The chapter also contains a synchronisation time test determining the amount of time the middleware takes for synchronisation between two related

tuples.

The test also discusses a situation where policies on two devices are in conflict with each other and makes a comparison between the two cases. Lastly, the section gives an estimate of the time needed for a tuple propagation process where a new tuple introduced into an area is propagated over the area's event distribution tree based on the results from the previous tests.

Chapter 8 contains discussions regarding to the middleware and the policy used by it. The discussion ranges from the middleware design and implementation process,with discussions of its supported applications, to a policy usage in the middleware including costs and benefits of more complex synchronisation processes that can support multiple policy sets in the same environment. Chapter 9 contains the project conclusion and possible improvement that might be made in future work.

Early results from this work were published in [JL06].

Chapter 2

Background Knowledge

2.1 Introduction

This project employs several technologies to be able to support wireless tuple synchronisation, from the physical layer such as the IEEE 802.11 wireless LAN to the technologies in the higher layer such as the tuple space paradigm and policy language.

This chapter gives some background to the technologies that are used in the project. The first section gives an explanation of the tuple space paradigm which is the data structure and data storage model used in this project. The section explains a tuple space as used in Linda and some systems that have since adapted the tuple space idea.

The next section gives a brief explanation of mobile communication and wireless networks. These technologies allow communication between devices without using any physical wiring between them. Next, the data synchronisation section gives an idea of the current technology of data synchronisation. Several synchronisation methods for both wired and wireless communication are presented.

The middleware section shows examples of existing middleware using both wired and wireless communication. Different middlewares aim to support different features. The examples raised in the section place stress on the middlewares that address synchronisation and communication problems.

The last section contains some background information about policy languages. There are several minor languages that have been created with a specific application in mind. This section give examples of some of the work that uses the specific policy languages but places more emphasis on more popular languages, especially the Ponder language which is the language used in this project.

2.2 Tuple Spaces

2.2.1 Linda

The tuple space paradigm was first used in a concurrent programming language called Linda [Gel85]. Communication between programs using a space is done by a sender writing a tuple into the space which will be withdrawn by the receiver. The tuple is an ordered list of typed parameters including the actual information to be communicated. The parameters in the tuple are used in a matching process that allows an application to access the tuple.

This type of communication allows time and space de-coupling in that there is no need for the sender to send its data at a specific time when the receiver is available to receive the data. The sender has no need to stay in the same area where the receiver is after the data has been sent. The information will stay in the space until some receiver removes it.

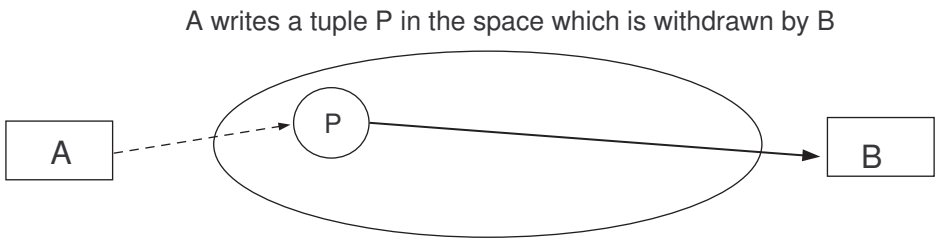


Figure 2.1: Tuple Space

Figure 2.1 shows the basic relation between an application, a tuple, and a space. There are three commands used in the language to interact with a tuple - in(), out(), and read(). Each tuple has tags attached which are used to identify it when it is in the tuple space. The out() and read() commands withdraw a tuple from the space by matching their tag queries with any tuple in the space with similar tags.

An example of how the tuple space of the Linda language can be used is shown in [CGL86]. The article discusses how a replicated-worker program can benefit from using the Linda language. A worker program is executed by several replicated processes. The processes work together using tuples in the space as their communication medium. The tuple space allows the system to be scaled by allowing more workers to be added easily.

More experiments in using the Linda language to solve problems can be seen in [CG88]. Linda was used, for example, in a system for finding resemblances between DNA sequences by distributing tasks, each returning results from searchers to a master process via the tuple space.

Implementations of Linda in parallel systems have been discussed in [BCGL86] and [CG86]. The usage of Linda and its tuple space paradigm has extended into many areas as can be seen from the number of middlewares that are discussed in 2.5

New ideas are being used to improve the tuple space such as the multi-agent system in [MT03] that uses a swarm bio-computing inspired technology to increase the efficiency of the Linda tuple space paradigm in a large scale multiple space system

2.2.2 JavaSpace

JavaSpace[Sun98] was created by Sun Microsystems. It is implemented using a number of Jini [Sun99] services and is distributed with them. JavaSpace adopts Linda's tuple space paradigm in an object-oriented world, and is implemented using the Java language. Instead of using the tuple space to implement a concurrent system as was the original aim in Linda, JavaSpace aims to be a way to provide data communication over a network.

To provide the JavaSpace service to an application several components are used; one or more servers run Jini's "outrigger"[Sun03] as a space service. Clients can interact with them by querying the Jini's "reggie"[Sun03] which is a registry service, obtaining references to the spaces. The client can then use a reference to access the space directly. Figure 2.2 shows an interaction between the components.

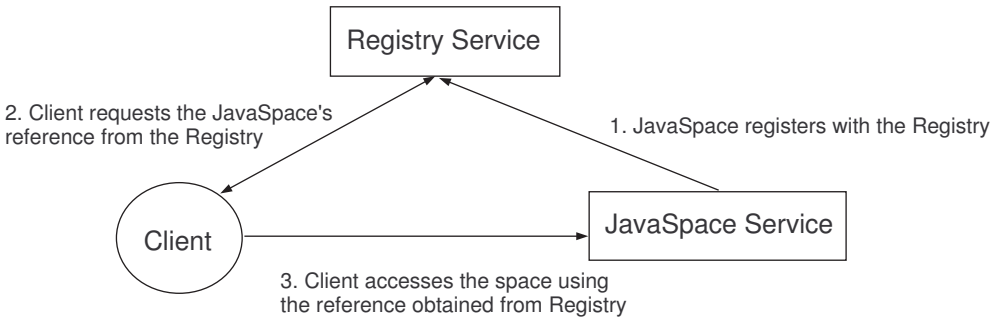


Figure 2.2: Client-Registry-JavaSpace Setup

The interface is provided by JavaSpace attempts to match the Linda language space interaction commands (in(), out(), read()). The JavaSpace calls these take(), write(), and read() instead. Moreover, it provides two more commands - takeIfExists() and readIfExists(). The basic commands block a querying process until a tuple can be acquired or a time out is reached, while the two extra commands return null if the tuple does not exist.

JavaSpace also provides an interface for an application to register an interest in a particular type of tuple. A notification is then returned to the registered listener when the requested tuple is written into the space.

JavaSpace uses a class implementing the "net.jini.core.Entry" marker interface to represent a tuple. There are two simple restrictions for this class. First, the tuple class needs to have a no argument constructor to allow a serialisation process by which the tuple is transferred into and out of a space. Next, all the fields in the tuple class should be declared as public fields and the values in the fields should each be a reference to an object. This is to allow proper lookup/query processing for the tuple in a space.

A tuple's life time can be specified by using a lease which is a part of the Jini package. Examples of more background information about how to use a JavaSpace can be found in [BW02], [FAH99], [Hal01], [FH99].

JavaSpace is the system selected for use in this project. However, there are a number of other systems that provide a tuple space service such as the IBM's TSpace [WMLF98] or JSpace [Led98]. There is not much information regarding to JSpace apart from the paper. It is built on RMI instead of Jini and supports the same basic commands used in JavaSpace. Like JavaSpace, TSpace is also implemented using Java and provides functions similar to JavaSpace. The same functions permitted in JavaSpace can be done in TSpace, but by using different commands [IBM]. The main different between the two implementations is that TSpace does not rely on the Jini services. Therefore, it is easier to use TSpace in a system that does not currently use Jini. On the other hand, a system already implementing services using Jini can add JavaSpace as just another service provided from the same service discovery process.

2.2.2.1 Applications using the JavaSpace

A number of projects use JavaSpace to provide coordination between their subsystems. For example, [HJ01] reports using JavaSpace in cluster management. A node writes a tuple into a space when it decides to join or depart from the cluster. A work scheduler receives a notification from the space when a new node joins or an existing node leaves the cluster. Information from it is written as a tuple and its lease time shows how long it intends to stay in the cluster, allowing the scheduler to assigns jobs properly.

[Bla01] uses a JavaSpace to allow communication between its rule-driven coordination agents. The agents contain specific rules for the components they represent and use JavaSpace as an event server to form part of a workflow automation system. [TNO02] changed the inter-processor communication system of their simulation system from using RMI to JavaSpace and report a better system performance from the change.

RDBSpace [AKR02] proposes to improve performance of a JavaSpace by utilising a relational-database system as the space back-end. The system also provides a simple backup and data replication mechanism from its mySQL database system which is not provided in the basic JavaSpace. GigaSpace [Gig06] is a commercial implementation of JavaSpace supporting more functions such as APIs for other languages, replication, and load balancing.

In term of the JavaSpace performance, [NZ01] uses JavaSpace in several applications that rely on

different algorithms to measure the efficiency of JavaSpace in various situations. The results in this article show that JavaSpace still has the communication latency problem that generally exists in Java and so may be more suitable for an application that is not communication intensive.

2.2.2.2 Reason for using the tuple space paradigm and JavaSpace

This project is based on the tuple space paradigm as a communication model because of its potential to support mobile applications. It provides both space and time decoupling which is essential in a wireless environment where devices are allowed to move and connections between them are unstable.

Moreover, the tuple space paradigm provides simplicity of tuple synchronisation at a basic level which comes from a limit on the operations available for interacting with a space when compared to those of file structures or databases.

Next, the tuple as an object provided by a JavaSpace can be a powerful storage mechanism. Instead of just storing data, because it stores objects, these can also contain methods. Therefore, a tuple can contain processes that can be used for interacting with data contained in the tuple. In this case, a tuple acts like an agent that can be passed to gather and process data on a remote host. However, this is out of the scope of this project.

2.3 Mobile and Wireless Network Background

A wireless network enables devices to communicate without relying on a physical connection between them. There are several existing technologies for wireless networks using a radio frequency carrier as their medium, such as HomeRF [NSL00], IEEE 802.11 [Kap02a], HiperLan/2 [VKP03] and Bluetooth [Bis01]. Each has different characteristics in several respects such as bandwidth and coverage area and they are suitable in different situations.

For example, Bluetooth uses less power than IEEE 802.11 and operates on a shorter range, which may be more suitable for a network between personal devices. IEEE 802.11, apart from IEEE802.11e, does not support QoS while HiperLan2 and HomeRF2 do [VZV03].

This project is based on IEEE 802.11 since it is widely available. Therefore, this section only contains information regarding this technology and its ad-hoc mode. However, the system built in this project should be able to operate on any network, including a wired network, as long as the Java Virtual Machine is usable, since the project uses only Jini services and sockets for its data communication.

IEEE 802.11 is a family of standards that provides an Ethernet-like facility as a wireless system over radio frequency channels. Each device communicates using a network infrastructure which may include a base station. A number of the standards are currently in use in different wireless devices such as IEEE 802.11b, 802.11a, and 802.11g. Each of the standards supports a different maximum theoretical bandwidths but the actual bandwidth available depends on several factors such as the distance between

the device and its base station.

More background knowledge on wireless networks can be obtained from the literature. [VZV03] reviews the various types of network. The paper contains information about the attributes of the networks and a comparison between them. [DVLLX02] gives background knowledge regarding mobile telecommunication standards (GSM, UMTS) and an overview of IEEE 802.11. [Kap02b] and [Kap02a] give an overview of the IEEE 802.11a and 802.11b standards.

2.3.1 Mobile Ad-Hoc networks

Ad-hoc networking is one of the communication modes that exists in the IEEE 802.11 standards. The widely used structured wireless network system requires each device to connect to a particular base station in order to connect to the network. An ad-hoc network increases the freedom with which the network can be set up by allowing devices to communicate directly without using any infrastructure. Figure 2.3, shows the different between the two network structures.

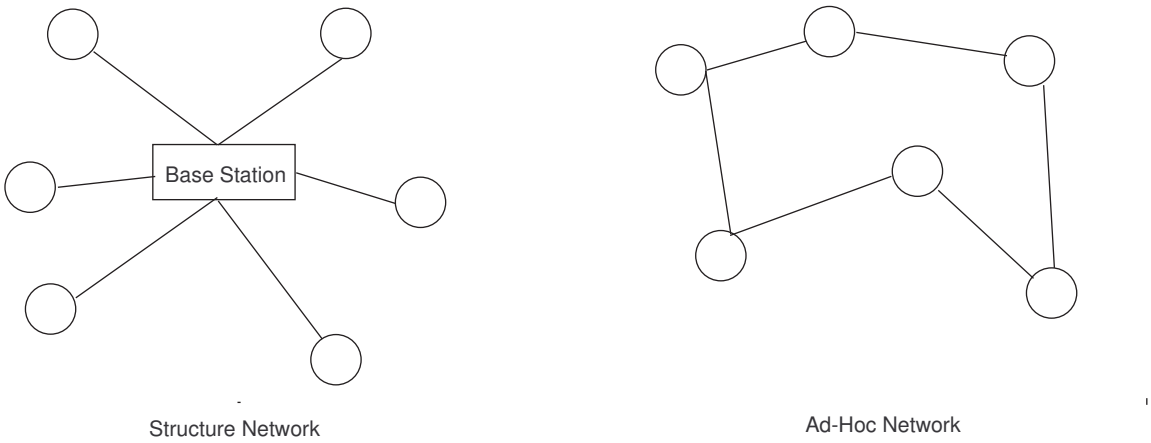


Figure 2.3: Structured and Ad-hoc Wireless Network Topology

There are a number of advantages of ad-hoc networks over structured networks. Without any need for fixed infrastructure, it is faster and probably easier to set up an ad-hoc network. For example, it is costly to establish a structured wireless network on a highway while an ad-hoc network can be created between cars travelling along it without any need for base stations.

However, ad-hoc networks also have several disadvantages. For example, the more freedom each device has, the more likely the network is to be unstable. Since the network relies on each node to act as a communication medium, there is more chance that a network will be broken into sub-networks which may cause interruptions in communication. More background knowledge about ad-hoc networks can be found in [RR02] [CCL03] [CMC99] [FJL00], for example.

There are several projects that are based on ad-hoc networks. Wireless sensor networks [CES04] is one of the application areas that benefits greatly from ad-hoc data communication. These systems employ a number of small inexpensive devices to monitor events. The ad-hoc network allows the system to be deployed in an environment where a structured network is not established.

Personal Area Networks and Personal Networks [NHdG02] [NHdG03] are other application areas that benefits from the ad-hoc communication mode. For example, ad-hoc communication allows devices attached to a user's body such as their PDA or mobile phone, to interact without using any base station. The ad-hoc mode allows the system to be more flexible by freeing it from any fixed network infrastructure.

2.4 Data Synchronisation and Conflict Resolution Process

One of the most important concepts in distributed systems is data replication. It increases data availability in that it allows access to the data even if a device cannot access the original data source. Data replication also increases the system fault tolerance by increasing data availability. Replicating data across several sites allows one or more sites to be shut down while a user can still access data from the other sites. Another benefit from data replication is performance improvement. Access time will be improved if a device copies data into its local storage rather than accessing the data across a network.

Nevertheless, data replication has its problems. One of the most important problems comes from changes that are made to data; these may make the data inconsistent. Without any control of the changes, data on different sites will diverge. A replication control technique or data synchronisation process is required to control the changes and reconcile any differences. Figure 2.4, shows an overview of different types of the technique.

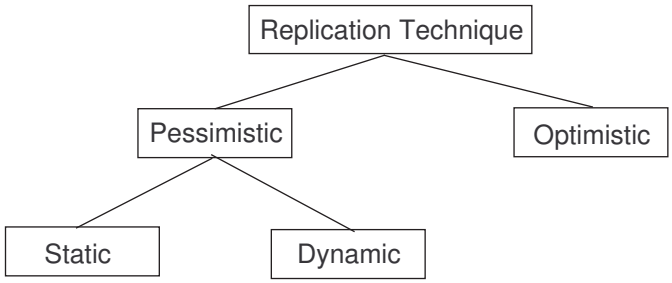


Figure 2.4: Replication Control Technique

There is much research in replication control and data synchronisation in the database area. One of the earlier techniques in replication control is pessimistic replication control [AD76] [TN89] [JM90] in which any operation that may create data inconsistency has to be done on the "primary" replica or

replicas which are then also responsible for propagating the update. As a result, the serialisation of such updates will prevent any inconsistency between replicas.

Selecting the primary groups is vital in a pessimistic scheme in that it affects the overall data availability of the system. If the groups cannot be established when there is a network failure, no one can update data in the system. There are two ways for selecting the primary group - static and dynamic selection.

Static selection is a process where the structure of partitions caused by network failures determines the primary group. Simple voting is an example of this method. The number of replicas in a group determines if the partition can be established as a primary group. [Gif79] is an example of using voting where the number of votes in a partition has to be more than a pre-defined threshold in order for the partition to be a primary group.

Dynamic selection not only uses a partition structure but also uses more information from each replica to help in determining the primary group. [JM87] [JM90] uses the number of up-to-date copies of a file contained in a partition to determine the primary group. This method gives a replica that is not up-to-date no vote. A partition that has more than half of the up-to-date files is a primary group. [MW82] uses a token to determine the primary group. Each file is given a token when it is written. A partition that has a replica that contains the token for a file that is going to be updated is a primary group.

Another replication technique is the optimistic replication scheme [Dav84] [Her90] [SS05]. This scheme allows data from any replica to be updated and uses a background data synchronisation process to reconcile any conflict resulting from several updates. How data synchronisation and conflict reconciliation are done varies from one system to another. Some of the systems using this replication control scheme are reviewed in 2.4.1.

The two replication schemes each have their advantages and disadvantages. One major advantage of the optimistic scheme is that it allows each replica to be updated during network partition. This is required in an application that is operating in a mobile environment in which each replica has a high chance of being separated. In an ad-hoc network, an individual device may often become separated, as when a user with a PDA is travelling back home. In this case, a network with twenty devices has a high chance of being divided into twenty partitions in which a simple voting algorithm cannot determine a primary group.

However, compared to a pessimistic scheme using voting, the system with optimistic replication control is more complex. The system has to rely on synchronisation and conflict reconciliation processes. The complexity of the two processes varies from a simple synchronisation for a new data entry to more complex conflict reconciliation in which information for making a decision comes from several sources. The system also requires a transaction rollback process to undo changes resulting from conflicting data.

This project follows the optimistic replica control scheme. The most important reason is that the project is built so that it can be used in a small mobile ad-hoc environment. In this environment, the

chance of a network partition happening is high, especially in the case where each replica is isolated as explained above. Therefore, a simple voting scheme is impossible. Moreover, an application in this environment requires each device to be able to update and modify its data while it is not connected to other devices, as users need to access and modify their data when they are at home. This condition rules out the pessimistic replication control which would allow only data on one partition to be updated.

2.4.1 System with Data Synchronisation

2.4.1.1 AdhocFS

AdhocFS [BI03a] is a file system for use in a mobile ad-hoc environment. This system uses data replication to ensure data availability within a group of peer devices and file servers. Files sharing in AdhocFS is done by extending the hierarchy of a local file system to identify available files and copying those files to be accessed into local storage if the files do not exist there.

File consistency in an ad-hoc group is maintained by an exclusive writer scheme. A write operation is locked for one writer at a particular time while a read operation can be used by any device. This process is done by giving each device in an ad-hoc group a state with respect to a particular file. For example, a device that can update the file has to be in a read/write state allowing the device to both read and write while the other devices must be in a read state.

A concurrency control list(CCL) is attached to each file for conflict detection and resolution processes caused by file updating while a device is disconnected from its peers. The CCL uses a timestamp for each file, indicating when it is taken from its reference copy, together with information about devices that have the current copy and any updates to that copy.

To enhance its performance, [BI03b] uses a device's profile in order to find a suitable device on which to store a replica in an ad-hoc group. The profiles are based on the devices' energy reserves, the duration for which each device has been in the group and the availability of local storage. For example, when the profile of a device storing the reference copy of the data becomes weak (e.g. low battery), the data will be propagated to another device with a stronger profile to preserve the data in the group.

2.4.1.2 Bayou

Bayou [DPS+94] is a client-server based system built to support data sharing between mobile devices. Data in Bayou servers is fully replicated and clients can access and modify data on any server which is then responsible for propagation of the update. A server in Bayou does not have to be a traditional fixed database server system. A mobile device with sufficient power can act as a lightweight server providing data for the other devices.

For data replication, Bayou uses a write-any/read-any replication scheme in which any device can read and write data. There is no locking used in the system which will provide more data availability for

each user. A client can access any available replicated server. Inconsistency from using different servers is managed by using session guarantees [TDP+94] such as making a read reflect any update done by the client or enforcing serialisation of the writing order.

The guarantee is provided on a per-session basis because an application loses a certain amount of data availability from using the guarantee. The loss of data availability comes from the limited number of servers that can provide the guarantee. Since the process is based on lazy data propagation, servers that can participate in a session guarantee for a particular client's session must be sufficiently up-to-date. Therefore, some servers may not be able to participate in the process. If any session guarantee cannot be provided by the system, the application requesting the session will be informed.

Conflict detection in Bayou is done by using a dependency set which is an application-specific query. A conflict is detected when results from the query are not equal to the expected results. The dependency set allows Bayou to do conflict detection which is tailored to each application. The check is done at a server before a write operation from a client is processed. The write operation is not performed if the check failed until the conflict resolution process has been performed.

The conflict resolution is done by attaching a "merge procedure" [TTP+95], which is a fragment of mobile code generated by the client updating the data and which will be run at the server in which conflict happens. This code can invoke the server database read or write command and can be customised so that each merge procedure satisfies the need of a specific application.

More background information about Bayou and its example applications can be found in [EMP+97], [TPST98].

2.4.1.3 Coda

For a system with files synchronisation, the Coda [SKS87] file caching system can provide information to a user while the user's device is disconnected. Coda is a distributed file system based on a client-server architecture. The trusted Unix servers contains files which will be cached at clients when they are requested using the client cache manager called Venus [KS92].

A client requests data from only one server. The server can be selected randomly or by using information from the server such as its performance or the distance between the client and the server. Before transferring requested data to the client, the server checks with the other servers to ensure that its information is up-to-date. Coda uses a read-once, write-all approach [SKK+90] for its replication mechanism where modified data are propagated. The propagation process is done using clients to reduce the workload on the servers.

While connecting to the servers, Venus carries out a caching process using information such as the file reference history and a customisable file list from the user. It changes to emulation mode when a client is disconnected from the servers. Normal file operations can still be done to the previously cached files in this mode. However, there is a chance that a cache miss can happen when an application accesses

information that has not previously been stored. In this case, an error code is returned to the application.

Venus contains a logging system for keeping track of file updates during disconnection. The information is used when the client reconnects to the servers. The re-integrating process contains conflict detection and resolution processes to reconcile conflicts occurring during the disconnection period. Conflict detection relies on information attached to an object when it is modified. Conflict resolution is based on using an Application-Specific-Resolver(ASR) [KS93] which is a rule file using system commands to resolve a conflict. A user intervention is required if no ASR file is found for the object.

More background information about Coda can be found in [Bra98] [LLS99] [Sat96]. [Sat90] contains background information about Coda and the Andrew File System(AFS) from which Coda inherits many aspects. Some information about Coda performance is described in [SKK+90] [NS94].

Together with Odyssey [NS99], Coda has also been used to create another project called Aura [Aur] [Sat01] [SG02], which tries to provide a ubiquitous service that reduces user interaction. It proposes that the most important resource in a system is now "user attention" as it tries to provide a distraction-free service to a user. One of the most important functions of Aura is to try to adapt to its environment such as where a user is going, to make it suitable for a user's application.

There are some example applications provided in [GSSS02]. First of these is People Helpdesk, which is a location and information finding service. It can be used to find the location of someone and their information such as their meeting schedule. The system supports a speech-based command interaction, which is easier to use on a mobile device, and a legacy user interface for a stationary user. Second, Ideal Link is a distributed shared whiteboard for collaborative work between mobile users. This application is also integrated with the People Helpdesk to determine which users can participate in a meeting.

2.4.1.4 Concurrent Version Control System

There are a number of projects that implement data synchronisation and conflict reconciliation. Concurrent Version Control System (CVS) [Fog99] allows developers to access and modify source files and employs synchronisation and reconciliation processes for merging source files from different developers.

CVS employs a copy-modify-merge methodology for its synchronisation and conflict resolution [Ber90]. A developer copies a source file, modifies it, and merges it back with its original. There is no locking or serialisation of the source file. The CVS can detect conflict between different versions of the same source file but the developers are responsible for conflict resolution.

2.4.1.5 Ficus

Ficus [PPGH90] [PGPH90] [GHM+90]is another distributed file system providing a data replication service. It is implemented as a value-added layer to be inserted between a system kernel and its file system. In this way, the project provides higher flexibility in that it can still be used when a new and

better file system is created. The stackable layers in Ficus allow other value-added layers such as an encryption layer to be inserted into the system.

The implementation of each layer in Ficus can be separated from the others. For example, the Ficus replication layer does not have to be in the same location as the actual file system. A transport layer can be inserted between any other two layers to provide interface matching transparently between the layers.

The current Ficus implementation is done using the Unix File System [PPGH90]. File replication is done by linking between a Ficus logical file layer which presents the file to an application and a number of replicated physical files which could be stored on remote locations. In other words, replicated files from different physical sites are presented to an application via the Ficus logical file layer as a single file.

Each logical file contains a set of inodes, one for every physical file replica and additional information to identify the physical files. The inode is information attached to each file in a Unix file system containing data such as a page pointer to the file and its ownership.

File updating is done by the logical replication layer which notifies each physical replica of any new update. Then, the replicas cache the new update and wait for an updating daemon to update the file. The daemon is used so that the update frequency can be controlled to suit the current situation.

A Version Vector is attached to each file to help determine conflict. Conflict reconciliation in Ficus is done by its reconciliation protocol and a list of predefined and user-defined resolver files [RHR+94]. When Ficus detects a conflict between two replicas, it does a system-wide search for a resolver that matches the conflicting files' type (using file name, and content). If the conflict cannot be resolved, Ficus calls a generic resolver to notify the user via electronic mail.

2.4.1.6 Differences Between Systems

The systems reviewed in the section above differ in a number of ways.

- One of the differences is in the architecture of the systems. Coda uses a client-server based architecture where a server is only responsible for storing information while an application is run on a client which caches files from the server. AdhocFS extends the notions to support more flexible networks in an ad-hoc environment by allowing caching between devices in a ad-hoc group together with a client-server approach similar to Coda.

  Bayou blurs the line between client and server by allowing a device to act as both client and server. A device can act as a server by fully replicating files from the other servers while serving as a proxy for other devices with lower performance.

- For data replication, all the systems use an optimistic replication scheme which avoids data locking to provide higher data availability. Instead of simply replicating data, Ficus attempts to add more value into the process. Its stackable approach provides a place where extra processes such as encryption-decryption can be inserted. AdhocFS stresses replication efficiency by using predefined

rules based on device parameters such as battery power to direct the replication process. This is to prevent devices with a weaker profile, which have a higher chance of disconnecting from the system, from storing a huge amount of data.

A client device in the Coda system determines which data is to be stored locally by using Venus. While the client is connected to a server, Venus in hoarding mode selectively caches data that a user may then use during disconnection. Data is selected using various parameters such as the user's usage information and preferences.

- In conflict detection and resolution, AdhocFS, Coda, and Ficus all rely on attaching information relating to any operation done to different replicas in order to detect conflict. Bayou, on the other hand, provides conflict detection based on a set of queries. A Bayou server checks a set of application specific queries before an update can be applied in order to detect any conflict.

  Conflict resolution in the systems is based on each user setting up a set of rules controlling the process. Each system uses the process with a different scope. Coda ASR and Ficus resolver files are attaches to a file or directory object which enforces the rules every time a conflict is detected. Conflict detection and resolution in Bayou are done on a per-session basis, which allows users to relinquish a certain amount of data consistency to achieve higher data availability.

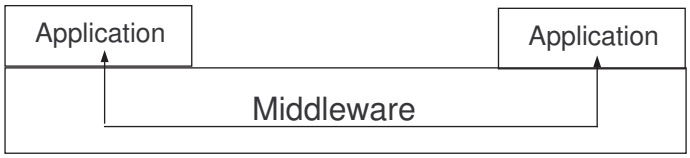
Apart from the systems reviewed in this section, there are a number of other projects that use data synchronisation and conflict resolution processes. Examples of the projects are the Andrew File System (AFS) [HKM+88] from which Coda inherits a number of its properties, and Locus [PWC+81] which is one of the earliest of distributed operating systems supporting some measure of synchronisation and conflict resolution. To get more information about the other systems, see [BNK89] which provides a survey of several distributed file systems or distributed operating systems including a number of systems that have not been discussed here.

2.5 Middleware Background

Middleware is a popular solution to link different programs, generally with a structure like that shown in figure 2.5. The general aim is to solve problems regarding to system heterogeneity and information distribution, and to provide an application writer with a unified programming model [BCRP98] [Ber96].

An example of a middleware that fits the above description is the Common Object Request Broker Architecture {CORBA} by the Object Management Group (OMG) [Gro06] [SFM+96] [Sie99]. The middleware allows applications from different vendors implemented using different programming languages to work together over a network.

In term of service distribution, CORBA helps to make application writing simpler by making the location of a service transparent as a result of using its naming or trading service. An application does



A middleware provides communication for applications

Figure 2.5: Basic Middleware layer

not have to know the location of the service to use it. The middleware is responsible for locating the service and then transferring information between an application and the service.

A service in CORBA is represented as an object. Interfaces offering the service are specified by the OMG IDL which is independent of any programming language but maps to the current popular languages such as C, C++, or Java. The IDL enables service encapsulation in that a client application only needs to know the service interfaces but not the mechanism that provides the service.

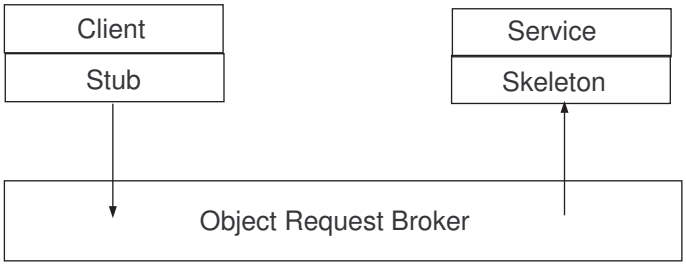


Figure 2.6: Basic CORBA Architecture

To provide these features, CORBA employs a number of components between a client application and an application providing service as shown in figure 2.6. IDL files are compiled into client stubs and service skeletons which act as proxies. A request from a client is passed through the client stub to the service skeleton via the Object Request Broker (ORB). The ORB helps the client find the service by using an object reference provided by the client; it may also provide additional services such as load balancing and fault tolerance. The client can get the service reference via a number of well-known services such as the Naming Service or the Trading Service.

Apart from CORBA, a number of middlewares currently exist. Java RMI [Sun06b] and DCOM [Mic06] are two more object-oriented middlewares in a similar style to CORBA in allowing an application to access remote objects as if they were local objects.

IBM MQseries [GS96] and Java Message Queue [Sun06a] are middlewares providing asynchronous message passing between senders and receivers. The sending process uncouples the client from the server, which allows different types of system to interact as long as they agree on the message format.

Next, transactional middleware provides a transaction service to components over a distributed system using a two-phase commit protocol. This type of middleware provides more system reliability. Tuxedo [BEA06] is one of the examples of middleware in this category.

Currently, as a result of growing usage in wireless networks, more new middlewares have been proposed to try to provide support for communication in a mobile environment in both the structured mode and the ad-hoc mode. These middlewares support mobile applications that run on lightweight devices such as notebooks, PDAs, or mobile phones, which typically have low processing power, battery capacity, and communication bandwidth.

The difference between these mobile middlewares and the conventional middleware is that a middleware designed for a wired environment usually makes a number of assumptions. First, a system node is generally powerful. Second, information transfer between terminals is done by a high bandwidth and stable communication sub-system. None of these can be assumed in a mobile middleware which has low power devices and unstable network connection, particularly because a terminal can itself physically move away causing disconnection.

Apart from the limitations above, the mobile middleware has important features such as context-awareness [SAW94] which is a property of software that can react or change its behaviour depending on its external and internal environment. Context-awareness includes, but is not limited to, visibility of location information, devices information, and user information. It is generally associated with ubiquitous systems [Wei91] [Wei99] but a number of middlewares also provide these features.

A technique called "reflection" can be used to provide system level context-awareness for mobile middleware. The technique was first used in programming languages to develop a language that is more extensible [KRB91]. The technique is used in middlewares such as CARISMA [CEM03], FlexiNet [HHB99], and Gaia [RC00] to allow the middleware to tune itself and change its behaviour according to stored context information.

There are a number of existing mobile middleware solutions. However, since this project stresses the tuple space paradigm, the next section contains a review of some of the mobile middlewares that are based on a tuple space. [GK03] provides more information for general mobile middleware. [Emm00] gives more background knowledge and information regarding several traditional middlewares.

2.5.1 Examples of Tuple Space Related Middleware

2.5.2 Limbo

Limbo [DWFB97] [BDFW97] is an example of a middleware based on the Linda tuple space concept and providing the tuple space operations it defines. Instead of using only Linda's single tuple space, Limbo allows an application to create a number of tuple spaces. Each space can be customised to suit the application needs, such as user authentication and space persistence.

The tuple space lifecycle can be controlled by a client. A new space can be created by the client sending a request to a common space in a server, stating any characteristics that the new space is to have. In the same way, a space can be destroyed by a request sent from a client but this request is put into the space to be destroyed instead of the common space. Limbo allows a tuple to have a type. The tuple type can be used during the tuple matching process in Limbo. The process will find tuples with either the same type, or a sub-type, of the specified tuple.

Another extra function that Limbo provides is Quality of Service (QoS). QoS attributes in Limbo are associated with tuples and space operations. For example, the QoS attributes on operations that request tuples allow a space to reorder the requests to provide a better service and utilise network connections efficiently.

Limbo also provides several system agents responsible for various tasks. For example, Limbo has a Bridging agent that is responsible for tuple propagation between spaces. The agent can be programmed to do the propagation based on factors such as the tuples' QoS attributes. It can also act as a gateway to translate tuples during the propagation.

Limbo has a QoS monitoring agent that constantly keeps track of several QoS attributes in a space. Each agent is associated with one space keeping track of information such as bandwidth available for communication from the space or the remaining power of a host. More agents monitoring other attributes can also be added. The information from the agents is made available as a tuple in the space allowing other hosts to make use of it.

Each tuple space in a host is implemented using a daemon process. The processes on different hosts collaborate to replicate tuple information between them. The replication allows an application to access a tuple even in a situation where there is no network connection. Tuple consistency and replication control are maintained by the Distributed Tuple Space Control Protocol [DFWB98] which is based on IP multicasting. The protocol enforces rules such as allowing only the device that has ownership of a tuple to delete it.

The Limbo name was later changed to L2imbo to prevent any confusion with another product from Lucent Technologies [Wad99].

2.5.2.1 LIME

Linda In a Mobile Environment (LIME) [PMR99] [PMR00] [MPR06]is one of a number of middlewares that apply the tuple-space paradigm from Linda to a mobile environment. In LIME, each mobile agent owns a tuple-space, which will be transiently merged (called "engagement") with the others when there is a connection between hosts. An agent can move between hosts and has control over its space in term of sharing permissions but an application that does not require a moving mobile agent can use the agent as if it was a fixed agent.

A host-level tuple space is created by merging the tuple spaces from all the mobile agents in a single

device. Control of data to be shared is done by allowing only those tuple spaces that have the same name on different mobile agents to be merged. When a mobile host is engaged with others, a federated space composed of the spaces from the mobile agents of different hosts is created. This will generate the sense of a local shared space for an application.

Access to data in LIME is done in the same way regardless of connectivity using the primitive commands which are provided in Linda. LIME also uses special commands which are an extension of the three basic space interaction commands provided in Linda to move a tuple between different spaces on different mobile agents or mobile hosts. The same is done for operations that read or take a tuple which allows an application to specify the scope of the commands. This will also create a location-aware image for an application, in that it allows fine-grained control over tuples.

Moreover, LIME introduces reactive programming by using a special command "reactsTo(s,p)". A code fragment is executed by a mobile agent whenever a matching tuple is added into its space. The semantics of the reaction command in LIME is based on the Mobile UNITY reactive statement described in [MR98]. The command can be attached to a specific tuple space which allows a finer grain of control.

The number of times the reaction command is going to be activated can be controlled using "mode" parameters. For example, the command can be set to activate only once or once per tuple; a command can also be specified with location parameters similar to the other normal commands.

There are a number of extensions and applications using LIME. [HR02] builds an application upon LIME to act as a jini-like service provisioning system in an ad-hoc environment. A tuple represents a service advertisement which allows an application to search for a specific service available in the area. TinyLIME [CGG+05] and coreLIME [CdOVV01] [CVV04] are two more extensions for LIME. TinyLIME was developed for wireless sensor networks where multiple mobile base stations each collect data from nearby sensor nodes and share it with other base stations. CoreLIME extends LIME to support security between untrusted agents using access control. It also claims to remove some of the inefficiency in LIME by removing some of LIME's features such as the once per tuple reaction mode which requires LIME to store an unlimited number of tuple IDs.

2.5.2.2 TOTA

Tuple On The Air (TOTA) [MZL02] [MZL03b] [MZL03a] uses a tuple space paradigm for supporting a mobile environment as LIME does. However, the way TOTA uses a space is different from the shared space model in LIME. Instead of a tuple being owned by a space, a tuple in TOTA will be propagated through a network containing a number of spaces.

A tuple is injected into the network with a specific pattern that controls its propagation as well as its content. Therefore, instead of a tuple containing only data, a tuple in TOTA will be defined using both data and a propagation pattern. The pattern can provide information such as how many hops the tuple should be propagated or how propagation should be affected by other tuples.

TOTA does not maintains tuple replica consistency between nodes. The information in a tuple may be changed when it passes each node. In brief, TOTA floods the network with information from different sources using the tuple as a carrier. A client, which has its own local space, queries this space to match any tuple that it receives. There is no notion of a remote query in TOTA. A client can query only its local tuple space but it can propagate a tuple which can then be interpreted as a query at the application level.

A good example scenario for TOTA is the museum example in [MZL02]. Each exhibit is a source for the TOTA middleware, and sends a tuple containing its information and location to the network. A traveler comes into a museum with a TOTA client device and receives information about an art object and its location. Since each tuple is propagated in the network with a propagation pattern, this information can be used to generate a set of directions to get the client to the art object.

2.5.2.3 TuCSoN

TuCSoN [OZ98] [OZ99] is another middleware using the tuple space paradigm. In TuCSon, a tuple space (called a tuple centre) attached to a host can be accessed by mobile agents using Linda-liked commands. Similar to LIME, TuCSoN employs environment parameters attached to each command to specify a tuple centre that a mobile agent should interact with, which can be a remote tuple centre or a local one.

Moreover, TuCSoN extends the simple tuple space model used in Linda in that its tuple centre is programmable using the ReSpecT language [DNO98]. TuCSoN's tuple centre can be designed to react to an operation from a mobile agent. The reaction command is defined by an association between a communication event and a stateful transition of the tuple centre in "reaction(Op, Body)" form. Op represents an action that will cause a transformation in Body. A reaction can access and modify tuple information in a tuple centre and can have access to information about reaction related entities such as the mobile agent that caused the reaction.

TuCSoN is used in the HiMAT project [COZ99] which aims to develop a framework for mobile agent applications with support for access control and security. The framework supports the movement of mobile agents between nodes, mobile agent authentication/authorisation, and interaction between agents.

TuCSoN is used as a mobile agent coordination system, primarily as an organisational abstraction in [OR03]. A tuple centre represents an organisational node defining roles and relationships. A resource in the centre can be accessed by agents that join the organisation. The access is controlled by statements in theReSpecT coordination language which allows an agent to be controled in term of rules, constraints, or permissions defined for the agent role.

More information about agent coordination in TuCSoN can be found in [RVO04] [ORVR04].

2.5.3 Differences Between the Middlewares

Even though the middlewares reviewed above are all based on the tuple space paradigm, their aims are different and this is reflected in how the middlewares are implemented.

- The first difference is in the entity that a space is attached to. Apart from LIME, all the other middlewares associate a space with a fixed host in the network. LIME, on the other hand, associates a space with a mobile agent and allows the two to be able to move together from one mobile host to any other. A host-level space in LIME is created by transiently merging local agent-level spaces.

- Next, Limbo explicitly makes replicas of tuples on different spaces via its system agents. It maintains tuple consistency by using the Distributed Tuple Space Control Protocol which limits the chance that a tuple conflict can happen by using tuple ownership. The other middlewares do not explicitly replicate tuples but allow them to be moved using special parameters, as in LIME and TuCSoN, while TOTA does tuple replication but does not enforce tuple consistency between replicas.

- LIME and TuCSoN have an explicit reaction command that triggers operations when a tuple is written into a space. However, there are some differences between the two middlewares' reaction commands. LIME based its command on a matching tuple in a space. Its reaction operation will be activated when an agent finds a tuple that matches a specified template. On the other hand, TuCSoN associates its reaction operation with an interaction between an agent and a space. A specific reaction operation will be triggered when a corresponding interaction occurs.

- Apart from the differences shown above, the middlewares each have their own specific features. Limbo stresses the control of a tuple space's life cycle by a client. It also aims to provide a tuple space that suits the client by allowing the space's parameters to be changed. LIME aims to provide a transiently shared space where a mobile host can see information that resides on nearby mobile hosts.

- TOTA does not provide a shared virtual space view but aims to provide a system in which tuple information can be changed depending on its propagation pattern, which can be used to reflect the physical location of each mobile host. Instead of using the tuple space paradigm as a means for data communication, TuCSoN extends it to form an agent coordination system.

2.6 Policy Background

There are various definitions for policy languages provided by different projects. [LBN99] defines a policy language as a language that describes strategies for a plan of action to achieve a goal while in

[Wie94] policies are derivable from higher level goals to define behaviour within distributed heterogeneous systems and networks.

The difference between the various definitions of policies is to be expected since policies are used in several different areas. Policies designed for managing system behaviour will be defined in terms of a language that is responsible for controlling the system while policies that are designed for security control will be defined based on its concept such as permission and access control. In general, a policy language is a language that is designed for organising situations where a number of decisions need to be managed in a consistent way [Lin06a]

2.6.1 Policy Model Overview

2.6.1.1 Policy Core Information Model

There are a number of policy standards and models from governmental and non-government organisations including groups formed by different companies [And06]. One of the most well-known standards for a policy model is the IETF RFC3060-Policy Core Information Model (PCIM) [MESW01]. A policy in PCIM is a set of rules and consisting of a set of conditions and actions. The actions which can change the state of an object will be activated when the conditions are evaluated to true. A hierarchy of policies can be created from nested policy groups where each group is formed by a number of policy rules. The components such as "PolicyAction" and "PolicyCondition" are represented as main policy classes in the model shown in fig 2.7.

Policy groups in PCIM can be classified into seven different categories, such as Motivational Policies, Security Policies, and Service Policies. This grouping allows easier usage of policies in term of querying or finding a policy since policies that fall in different groups are different in term of when or how they can be used, for example.

There is an extension to this standard in RFC3460-Policy Core Information Model (PCIM) extensions [Moo03]. The new standard makes several changes. For example, PCIMe defines two type of policy variables - explicit variables which will be evaluated outside the model and implicit variables that are defined in the model. An explicit policy variable has an exact formulation in the model while an implicit policy variable does not. The model also defines the way to bind the policy variables to values.

2.6.1.2 Policy Based Admission Control

Another example of a model for policy is given in RFC 2753-A Framework for Policy-based Admission Control [YPG00], which specifies policy control over admission control decisions. The model describes two important elements for decision making using policies - a policy decision point (PDP) and a policy enforcement point (PEP). The PDP may reside in a policy server responsible for making decisions while the PEP is in a node that actually does the controlling. The PEP request a decision from the PDP

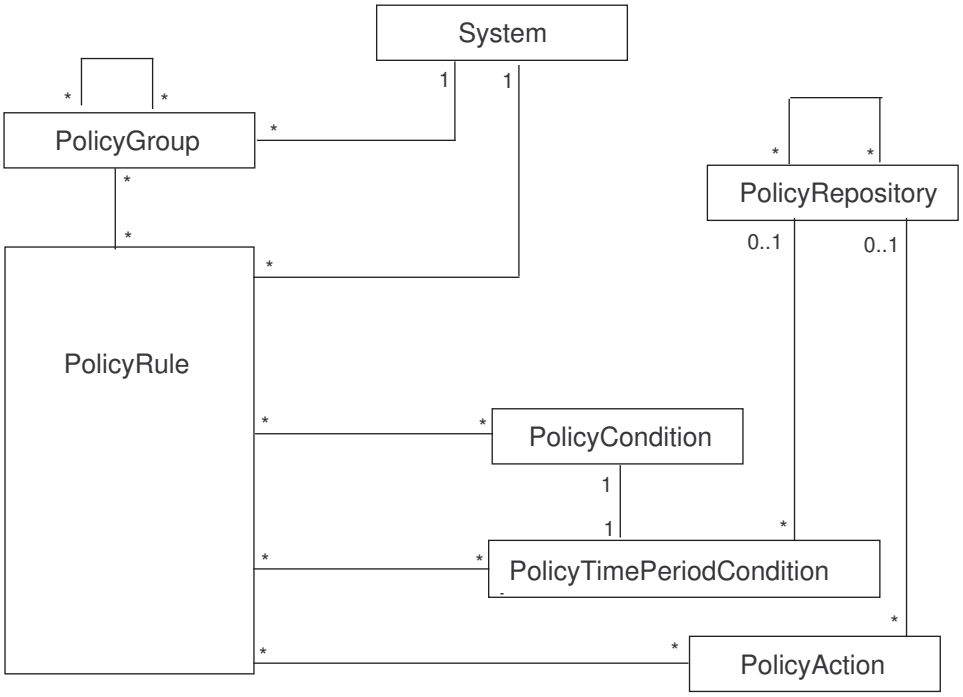


Figure 2.7: Overview of PCIM main policy classes and their relations

regarding any request to access resources. A policy decision is made in the PDP and its result is sent to the PEP for enforcement. The document also defines basic interactions between the two elements.

The protocol for interaction between the PDP and the PEP is defined in RFC 2748-Common Open Policy Service (COPS) protocol, as shown in figure 2.8. It employs TCP for reliable message passing between the two elements. In normal circumstances, the PEP is responsible for establishing a connection to the PDP, initiating a stateful communication in the form of requests/decisions. Definition of the service discovery mechanism is outside the scope of the protocol.

Figure 2.8: COPS model

2.6.1.3 Other IETF Standards

In large scale organisations, a policy for managing a large number of network devices is defined at a high level so that a single policy can control a number for devices. This automated control process, called Policy Based Management, uses a group of Management Information Bases (MIB) [MR91] defining properties of objects to be managed. RFC 4011-Policy Based Management MIB [WSH05] defines the MIB objects that are used to model policy control using roles, capabilities, and time.

A role defines the administrative characteristics of an object, and is used to identify a set of policies to control the managed object. One role can be used to define a group of managed objects and one object can be in several roles. The RFC offers a number of examples of roles, such as a role related to organisational rule (a class of user) or a financial role (paid customer).

A capability indicates what a device is capable of. A management station queries a list of capabilities to help the station in choosing suitable policies for the device. Moreover, the station can register for any new capability which will cause the system to notify the station when a new capability exists. Including time in a policy allows the management station to specify when policies should be active. The MIB provides a schedule table that allows a policy to be inactive at some defined time, to become active during a specified period, and then to become inactive again.

The RFC 2622 [AVG+99] shows another area where policy can be used. Its defines a Routing Policy Specification Language (RPSL) which allows a network manager to specify high level routing policies that are used to generate a router configuration. The RPSL is an object-oriented language that is device-independent. The RFC defines a number of objects used in the RPSL. For example, a maintainer class (mntner) defines the authorisation that is needed for an entity to add, delete, and modify objects. An autonomous system (aut-num) class represents the routing policy for an autonomous system. A route class specifies a route between autonomous systems.

```
aut-num:    AS1
import:     from AS2
            accept {128.9.0.1, 128.8.0.1}
```

Figure 2.9: An example of RPSL

Figure 2.9 shows an example of a routing policy. In this policy, devices that belong to AS1 accept route steps 128.9.0.1 and 128.8.0.1 from AS2. A router configuration can be generated from an RPSL policy together with a router identity such as the router ID and an autonomous system number for the router.

Apart from standards from the IETF, there are other organisations defining models regarding policies.

For example, ITU-T X.812 Access Control Framework [Uni95] defines a model that separates a decision making point from an enforcement point in a similar way to the PDP and the PEP concepts.

2.6.1.4 Web Service Related Model

The Web Service Policy Framework (WS-Policy) [BBC+06a] defines a policy model that can be used in web services. The policy is used to ensure a correct interaction between two web-service end-points. It can be used, for example, when a service provider allows a requester to check the service policies before deciding whether or not to use the service.

A policy in WS-Policy is written using XML to allow interoperability between systems. A single policy is composed of a number of policy alternatives which represent choices in the policy. A requester of a web service can look up the different policy alternatives and select one of them. Each policy alternative contains a number of policy assertions, each of which is about one requirement or capability of the policy.

WS-Policy does not define any particular expression of policy assertions, or how a policy can be attached to a web service. A policy assertion is domain specific and examples of it can be seen in a number of standards. For example, the Web Services Security Policy Language (WS-SecurityPolicy) [DLGHB+05] defines assertions concerning security constraints or requirements. Web Services Reliable Messaging Policy Assertion (WS-RM) [BBB+05] defines policy assertions concerning reliable communication between web services such as a re-transmission rate or an acknowledgement interval.

The Web Services Policy Attachment (WS-PolicyAttachment) [BBC+06b] specification defines how to associate a policy with an object. This can be done by specifying a policy as part of an object's definition or having a special tag separately specifying policies that are attached to resources.

2.6.2 Policy Language and Usage Overview

Apart from the various policy models described in the previous section, there are a number of widely used policy languages which will be reviewed in this section.

2.6.2.1 Ponder

The Ponder language [DDLS00] [DDLS01a] [Dam02] is a declarative object-oriented language that originally focused on access control and management policies. Policies apply to actions by subjects on targets, and are triggered by events, conditional on constraints expressed in OCL expressions. Ponder also supports management structure by defining roles, groups, and relations between them, to reflect the organisational structure being used.

Policies in the language can be roughly divided into four different types - authorisation, delegation, refrain, and obligation. An authorisation policy is an access control policy specifying actions that a

subject, which represents a user, can or cannot do to a target,which represents a service or a resource that the user wants to access. A subject or a target in the Ponder language is defined using a domain scope expression (DSE) representing a group of entities grouped together for management purposes. The DSE supports simple set operators (such as + , -) allowing more flexibility in defining the subject and target.

A delegation policy is used to allow a subject to be able to grant access rights temporarily to a certain target. The subject must also possess the right that is to be granted before it can do so. A grantor exercises a fined-grain control over the right to be granted. For example, it does not have to grant the right as a whole. It can specify part of a target that a grantee can access or grant a right for a grantee to do only some of the actions the grantor has a right to do. Moreover, the right can be limited by specifying a period or duration for which the right is to remain valid.

A refrain policy prevents a subject from doing a set of actions to a particular target. This policy is similar to a negative authorisation policy in that it prevents an action from happening. The difference between the two policies is that the negative authorisation is enforced at a target while the refrain policy is enforced at the subject. It is used in the case where a target cannot be trusted to enforce an access control.

Lastly, an obligation policy defines an action that a subject has to do when a certain event happens. When a specified event occurs, which could be a simple timer event or an event that is monitored by an external process, the defined subject must act as specified in the policy action part within the specified constraint. An exception clause is also specified in the policy in case the specified action cannot be done. The obligation policy is going to be used in this dissertation to control the synchronisation process that happens between nodes. An example is shown in figure 2.10.

```
inst oblig pol1
{
    on                      NormalSync_SingleSide()
    subject  <space>        s = /space;
    target   <tuple>        t = /tuple;
    do                      s.retrieve();
    when                    /tuple -> exists(t1, t2 | t1.localtuple = true and
                                        t2.remotetuple = true and
                                        t1.writtenBefore(t2));
}
```

Figure 2.10: An example of an Obligation policy

Most of these policy types can be controlled using constraints specified in the OCL language [Obj97], which is a formal language to express side effect free constraints. The constraint expression is evaluated giving a boolean value which can then be used to decide if the policy is applicable in the current situation or not.

Ponder also provides ways to group policies together using groups, roles, relations, and management structures which make it easier for a manager to organise policies. For example, a role structure allows policies with the same subject type to be grouped together, defining, for example, policies for a manager or policies for a technician. A relation specifies policies that are used when a defined subject role interacts with a defined target role such as policies that are to be enforced when a manager interacts with a technician.

During deployment, a policy is compiled into a policy object which contains information and states of the policy. The Ponder deployment model relies on a number of services. The Policy Service stores policy classes and creates policy objects. The Domain Service supports a subject and target evaluation process. The Event Service composes events and forwards them to registered agents to trigger an obligation policy. Detailed information of the deployment model can be found in [DDLS01b].

Ponder provides a toolkit which allows a policy writer to compile a policy file and create a policy object [Pol06]. The compiler is roughly divided into three parts: a syntax analyser, a semantic analyser, and a code generator. The toolkit has a JavaCodeGenerator that can create a policy Java object [Dam01] from an intermediate code generated by the analysers. The compiler framework also supports other code generators that are compatible with the model provided by the framework.

Apart from the compiler, [DDL$^+$02] lists a number of additional components provided in the toolkit. For example, the Domain Browser obtains data from the Domain Service providing a user with interfaces to group and manage objects into domains. The Management Console Tool allows a user to manage the policies' life cycle such as loading and unloading policy objects. The Conflict Analyser can detect static conflicts [LS99] between policies such as the conflict between a negative and a positive authorisation policy. A significantly different product called "Ponder2" which is a re-designed of Ponder product is available at [Pon07]

2.6.2.2 Rei

Similar to Ponder, Rei [Kag02] [KFJ03a] is also an authorisation policy language using the concept of condition-action. It stresses decentralised control and aims to make the language simple. Rei is based on the logic language called "Prolog" [Swe06]. The Rei language defines three policy constructs - policy objects describing policies which represent rights, prohibitions, obligations, and dispensations, speech-acts defining policy management interactions such as delegation and revocation, and meta-policies defining constructs to resolve policy conflicts,

A policy object in Rei can be roughly divided into two parts. The first part defines a policy body composed from a condition-action (*PolicyObject(Action, Condition)*) which represents the four policy types listed in the previous paragraph. The action part encompasses several elements describing the action such as an action name, a target of the action, a pre-condition that has to be met before the action can occur, and any effect of the action.

The second part links the policy body defined in the first part with a subject (*has(Subject, PolicyObject)*). This allows a policy body to be reusable by linking it with different subjects. The subject can be an individual entity specified using a URI or a number of entities in the same group or role.

There are four speech-acts in Rei representing the abilities granted to a subject to do certain actions on a policy. A delegation speech-act allows a subject to give a right to a target which creates a new right at the target. The subject can also cancel the given right. A request speech-act can be used to ask for an action or a right from a target. Asking for an action creates an obligation on the target and asking for a right causes the target to delegate the right to the subject, if the receiver decides to accept it. A revocation speech-act is used to remove a right for performing an action and a cancel speech-act is used to cancel a request speech-act.

Rei provides meta-policies that can be used to resolve conflicts during run time by using priorities and precedence which follow the conflict model described in [LS99]. The priorities are constructs that can be used to set priorities between policies or groups of policies. They allows one policy to override another policy during conflicts. The priorities can also be used to set an evaluation order between policies. In Rei, precedence is used to define priorities between different policy modalities (negative/positive). For example, if a negative modality takes precedence over a positive modality, prohibition will have higher priority than right and dispensation will have higher priority than obligation.

An example of a Rei implementation in a mobile network can be found in [PKKJ04]. A mobile device hosts a policy enforcer, a context manager, and a policy manager. It connects to a policy engine in a server via an access point. The server contains policies to be used in its domain. The context manager monitors the device status and any communication from the server. It acquires a new set of policies from the server to replace default policies in use when the device boots up. The default policies only allow minimal capabilities sufficient for the device to connect to the server to ask for working policies.

A Rei policy that resides on a mobile device is valid for a certain period. This is to contain the policy effect in a domain. Each policy validation depends on a time kept track of by the context manager. While the device stays in the area where it can connect to the policy server, the context manager receives a keep-alive message (heartbeat) from the server and renews timers for each policy. Once the device leaves the area, the timer will not be reset and policies will soon become invalid.

More information about Rei and related works can be found in [KFJ03b] [KF06] and in several papers contained in the project website [UMB06].

2.6.2.3 A Policy Description Language (PDL)

The PDL [LBN99] [VLK00] [BLK00] is another policy language based on the event-condition-action paradigm where an action is taken if an event occurs and a condition is evaluated to true. The PDL aims to be a simple but expressive language which is influenced be previous works such as a state language used for planning in [GB98] and an action description language in [GL93].

Figure 2.11: A structure of the Policy Description Language

A policy in the PDL is divided into three parts - event, action, and condition - as shown in figure 2.11. An event defines a situation that will trigger the policy. It can be a simple single event or a group of events. The PDL provides constructs that allow a group of events to be ordered in a number of ways. For example, a group of events composed of events that happens after each other (e1 & e2 & e3) or a group where any event occurring (e1 $\mid e2 \mid e3$) can trigger the policy.

The condition part in the PDL is a set of predicates with operands which are primitive attributes from an event in the event part, a constant, or a result of an operation applied to the event. The PDL also defines an aggregator that can be used to aggregate primitive event values. For example, a count aggregator can be used to count the number of times a specific event type occurs over the course of some working period. Lastly, the action part defines actions that will be done if the condition predicates are evaluated to true.

[VLK00] gives an implementation example of a system that uses the PDL. The language is used in the management layer of a SARAS softswitch, which is an IP network software switch. The switch is a pure java based software element that can be operated on a general platform. It can switch traffic from different sources using different protocols.

A Policy Enabling Point (PEP) is set up at each network component that will be controlled by the PDL. It is composed of an event filter mapping real world events into policy events, an action evaluator enforcing a policy, and an SNMP sub-agent reporting state of the component to the Aggregator.

The Aggregator aggregates information from each PEP and provides an overall viewpoint to a policy writer. It is also responsible for rerouting policy actions to a PEP when a policy is triggered. The Policy Engine runs each policy as a process inside it and the Directory Coordinator keeps persistence information which is used for establishing communication and recovery.

A number of policies defined in the PDL are deployed in the switch management layer. For example, a policy can be used to send an SNMP messages to a management station monitoring failures that occur in the system. A burst of events can be collected and sent as one single event. At the same time, another policy counts the number of events that occurs in each burst and reports the number to the management node. Moreover, policy can be used to force each switch component to reload during the nighttime or can be used to specify that a component is going to be the back up for some other component.

2.6.2.4 Language Comparison

The previous sections give an overview of three policy languages. Some features of the languages are similar, such as their abstract structures that are based on an event-condition-action model. However, there are a number of differences between them which will be discussed in this section.

- First, even though the three languages are based on the event-condition-action structure, the limits on what can be expressed using the languages are different. Ponder and Rei can be used to define authorisation, obligation, and refrain policies while the PDL is limited to expressing an obligation policy.

- Second, how a policy can be linked to a subject is different in the three languages. Rei explicitly distinguishes between a policy object and a subject that the policy object is related to. The two parts are linked using a "has{}" structure. This allows the two parts to be separated and linked back easily. The Ponder language defines a subject as a part of a policy instance but allows a user to separate between the two parts using "type". The more complex separation method in Ponder has higher flexibility since it allows any policy element to be passed as a formal parameter. The PDL, on the other hand, does not explicitly define a subject as a part of its policy structure.

- Third, Both Ponder and Rei have a structure for policy delegation using a delegation policy structure in Ponder and the speech act in Rei. The two support several similar features, such as a condition for delegation, a target of delegation, and validity period. However, Rei's speech act goes further than just policy delegation. The speech act allows more powerful control over a target such as revocation of a right, or a request speech act which allows the requesting of a right or action.

- Fourth, Ponder supports policy groups and relations, which allows policies with similar parameters to be grouped together which makes them easier to manage. Rei and the PDL do not have any construct to define grouping.

- Fifth, concerning policy conflict, Rei explicitly defined a construct that can be used to resolve a conflict happening between policies. For example, Rei has the "overrides()" structure that can be used to define priority between policies. Moreover, Rei provides constructs that can be used to define precedence between policy modalities. These rules can be associated with a specific action or subject.

  Ponder and the PDL do not explicitly define any structure that is used to resolve a policy conflict. However, the language structure of Ponder can be used by itself to build a higher level policy that can be used to resolve a policy conflict. This concept is used for conflict resolution in this project.

- Lastly, the structures of how a policy can be written using the three languages are different. Ponder separates each policy part on different lines using standard keywords which makes the language

easier to read than Rei which does not have any explicit segmentation into parts. For a reader or writer that is not used to Prolog or any other logic programming language, it will take longer to understand the same policy when written using Rei than it will in Ponder.

2.6.3 Meta-Policy

Apart from a policy, there is also a concept of meta-policy that has been explored by a number of researchers. Generally, it is used in order to specify how a policy should be defined[BM02b]. A meta-policy can be used to define the goal of policies such as specifying a set of privileges that a group of users must be assigned or must not be given by policies.

Moreover, a meta-policy can be used to allow inter-domain communication with confidence. A set of meta-policies is agreed between two domains which is used to establish communication and access policies that will be used between them. Communication policies are tested against meta-policies before an actual communication can happen to verify that the two sides follow the rules (meta-policies) that are set beforehand.

One example of work that applies the meta-policy concept is OASIS [BM02a], which is a role based access control architecture that allows interoperation between domains that are managed independently. The access control rules in OASIS are defined using policies and meta-policies are used to define access control rules between domains. OASIS presents an example of meta-policies that can be used in the National Health Service (NHS) system such as a high level regulations that allow a doctor to access patient information that resides in a different domain.

By specifying meta-policies and enforcing compliance among policies in user devices, a policy conflict during a synchronisation process can be prevented. However, there is an issue of how much freedom should be given to a user. If a user has to comply to a set of meta-policies, they may not have enough flexibility to define their own policies. This project decided to allow a conflict to occur and then to use user defined higher level policies to solve the policy conflict after it has happened.

2.7 Conclusion

This chapter has given a review of background knowledge and projects that are related to this work. First, the tuple space paradigm which is the communication paradigm employed in this project, was explained. This concept has been used in several areas since it was first used in the Linda parallel language. It provides a space and time decoupling between a sender and a receiver which makes it suitable for a mobile environment. Then, JavaSpace which is a part of the Jini services was discussed together with other pieces of work which provide tuple space services.

Next, wireless communication was briefly discussed. Different standards for wireless communication were introduced and mobile ad-hoc communication was briefly explained. Then, data synchronisation

which is an important part of this work was discussed, starting with several replication techniques taken from work in the database area. A number of examples of work in synchronisation and conflict resolution were also discussed in this section.

A brief explanation of the idea of middleware was also given in this chapter. The section used CORBA as an example to explain how a middleware can provide several services that make creating an application easier. Then, a number of middlewares providing a tuple space service were introduced together with a comparison between them.

Lastly, the concept of a policy language was introduced. The section showed a number of policy model standards from different standardisation bodies and research groups. Different standards are used to model policies that will be used in different types of system. Afterwards, examples of policy languages were briefly explained. These languages are not specific to any particular area of work and can be applied in a number of different systems such as access control, or system management. A brief discussion of the differences between them was also given in the section.

Chapter 3

Application Examples and Methodology

3.1 Introduction

In order to investigate the control of a tuple synchronisation using policy, a middleware that supports a controllable synchronisation process is needed. Building one will allow an examination of how the process can be done. The middleware must provide interfaces that allow different applications to control and benefit from its synchronisation process.

Moreover, since this project is interested in providing the service in a mobile ad-hoc network, the system built here must provide communication support so that applications do not have to worry about remote communication. This implies that the middleware cannot rely on several things that are generally available in centralised systems, such as persistent servers for services, reliable network connections, and constant data availability.

This chapter explains the purpose of building the middleware and outlines the methodology used in doing so. It gives examples of a number of application areas that can benefit from this kind of middleware. It lists a number of requirements for the middleware and the policies that are responsible for making decisions during the synchronisation process. It then explains how the system was tested to ensure that it can support the requirements.

3.2 Application Scenarios

The basic idea in building the middleware is that different applications operating on either a wired or wireless network may need different types of data synchronisation process and use different types of information in doing their synchronisation. This section provides examples of a number of applications

that illustrate the idea.

These applications are mostly applications that benefit from an ad-hoc communication environment but some are based on legacy networks. Some of them do not yet exist but still provide good examples of requirements for future ad-hoc network application. The examples provide one variant of each application but the same application could be implemented differently with different types of policy.

At the end of each example application, there is a table giving examples of policies that can be used with it. There is also a list of the context information required for use with these policies.

Most of the applications shown in this section require specific methods and information to carry out their synchronisation processes. There is no regular pattern for the types of information required which confirms the need for middleware flexibility and the ability to change the synchronisation behaviour to better suit different applications.

3.2.1 Meeting Management

The meeting management application allows users in a group to synchronise their meeting schedules. Each user can create a meeting by writing information such as the time of the meeting, the room number, and the number of persons in the meeting into their data storage. The information will be synchronised whenever users move into range of the group's ad-hoc network. The goal is to maintain a unique virtual space view that allows each user to see the meeting schedule of the other users. Even though the information on each device may not be up-to-date, this is a best effort situation where having some of the data may be better than not having it at all.

Example Policies	- AcquireNonExistingTuple - ReplaceTupleConflictOnHigherClass - ReplaceTupleConflictOnEarlierTime
Context Information	Class of User Information Tuple Creation Time Information

Table 3.1: Meeting Management Policy Components and Required Context Information

Each user can modify, add, or delete his or her meeting information at any time. This situation introduces a conflict between users when they create a meeting using the same room at the same time and a policy that can be used to resolve a tuple conflict is required in this situation.

An example of conflict resolution in this case might be to use a class of user to decide the priority between the users. For example, a user higher-up in an organisation (manager, CEO) is given a higher class and higher priority in choosing and creating a meeting. This can be interpreted using a bronze, silver, and gold class provided to each user device.

However, there is always a chance of a conflict occurring between users with the same class in which just using the class of user information will not be able to resolve the conflict. Extra information such as the time of day when the conflicting meeting tuples were created must be used as a second level conflict

resolution policy.

3.2.2 Parking Network [BL02]

A parking network is a good example of an application that is operated over a wireless ad-hoc network. Cars and parking meters in the network are equipped to use the middleware and act as nodes in the application. Each parking meter writes a tuple to a space which will represent an individual parking space for which it is responsible; this will be propagated through each car and parking meter in the network.

In this way, the unique virtual space view in this system provides information about available parking spaces and their specific information. When a car decides to park in a specific parking space, it edits the tuple representing the parking space, establishing a state such that it is reserved for the car.

Conflict in this scenario can occur where more than one car tries to reserve the same parking space while they are not connected to each other. Several policies can be used to resolve the conflict depending on the intention of the application designer. The simplest policy could be a policy that gives the higher priority to the car that reserved that parking space earlier.

However, with the help of some context information, usage of more complex policy rules allows reservation of the space to be more flexible. For example, a set of parking spaces situated in a university may allow a lecturer to have higher priority than a student, while this policy is not used for parking spaces in the town centre.

Similarly, a rule concerning time of day can be used to make the system more flexible, for instance, by applying the above lecturer-student rule only between 6am to 6pm, while everyone has equal priority outside of this period.

Example Policies	- ReplaceTupleConflictOnHigherClass IfWithinTimeConstraint - ReplaceTupleConflictOnEarlierTime IfOutofTimeConstraint
Context Information	Class of User Information Time of Day Information Tuple Creation Time Information

Table 3.2: Parking Network Policy Components and Required Context Information

3.2.3 Taxi Dispatching Service

The Taxi Dispatching System is aimed to allow each taxi driver to be able to communicate their intention of picking up a customer. The information made available by the system allows a taxi driver to select a customer who has requested a pick up and at the same time notify the other taxi drivers that he is going to pick up the customer. This example will be used to introduce more complex rules than in the earlier

applications. However, this example is not necessarily a good way to implement the service in this kind of environment; a centralised server might be more reliable. However, it is intended as an example to show in simple terms how situations in an ad-hoc network can make a policy more complex and how an incorrect usage of policy can cause an application error.

The basic equipment in each taxi is able to communicate with the other taxis in a wireless ad-hoc mode. A taxi in the system has a device supporting the JSFM middleware and a taxi dispatching application which will show on a screen the list of users requesting a taxi. The list may include the requester's origin and destination. The driver has an interface to interact with the application, such as a touch screen or pointing device.

To call a taxi, a customer uses a client device that can inject information into the virtual space of the system to request a taxi. This might be specialised equipment situated at each taxi stand or it could be a web service that the customer can interact with over the Internet.

The request from each customer is propagated via a synchronisation process from one taxi to the others in the form of a tuple containing customer information. When a new customer information tuple reaches a taxi, the taxi dispatching system in the car reads the tuple and displays it on the driver's screen.

To accept a customer request, a driver picks a request and marks the tuple as accepted (by using a tuple editing process). The system will write information about the taxi that has accepted the customer tuple, such as the driver identification, taxi number, taxi location, and time the tuple was edited. The modified information is propagated to other taxis via the wireless interface. In the same way, if a customer has a device that can access the information, the customer can be notified of the taxi that is going to pick him up.

After the driver accepts the request, there is a chance that another driver who has also received the customer request tuple may decide to accept the request. In this case, modified information from both taxis will cause a conflict during the synchronisation process.

The simplest policy to use in this application is to allow the driver that committed to the request tuple earlier to pick the customer up. The use of this policy is ineffective, for example, in the case where a remote taxi commits to the tuple earlier than a local taxi, as shown in figure 3.1.

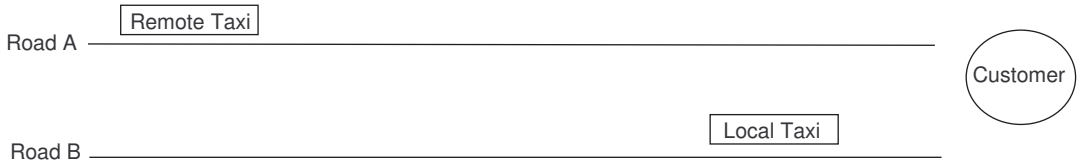


Figure 3.1: A Taxi Dispatching Situation

If there is enough taxi traffic between the two taxis to allow some communication between them, the

local taxi will get a tuple that is modified by the remote taxi which will generate a conflict. The simple policy will give a higher priority to the remote taxi since it committed to the request earlier. At the time the conflict is resolved, the remote taxi may be several miles from the customer while the local taxi may be less than a mile from the customer.

On the other hand, each taxi can also use a policy that gives a higher priority to a taxi that is nearer to the customer (assuming that each taxi is equipped with a GPS (Global Positioning System) so that positioning information will be written when the system on the taxi modifies the customer request tuple) In this way, a customer will be picked up by a taxi that is the closest to him. However, there is still a chance that the policy may be ineffective. Consider a situation where a local taxi is moving away from the customer and there is no easy way to turn back, while a remote taxi is moving in the right direction. In this case, the remote taxi may be able to arrive before the local taxi does.

Another example is a situation where a local taxi and a remote taxi are on different roads. If the local taxi is nearer to a customer than the remote taxi, but the road that the local taxi is on has a traffic jam, the remote taxi may be able to get to the customer faster than the local taxi.

A solution to a situation where a policy is composed from several pieces of information may be to use a Utility Function [KW04] in which a decision is made by comparing values obtained from a function of several factors that affect the situation. For example, the taxi dispatching situation may require a utility function combining the creation time by the taxi driver, the current speed of the taxi, and the distance from the taxi to the customer. The function created using the information could help in making a better decision. However, the problem with the function, as in any system using a Utility Function, is how to create the best function from the available information.

The examples above show that there are many situations where the data synchronisation process, if not done properly, makes the system less effective. However, there are cases where improper data synchronisation not only leads to an ineffective system but also causes an error in the system. An example is the situation where data communication is not reliable enough. This can occur where there is too low a traffic level to communicate modified tuple information. For example, consider a case where there is only traffic from a remote taxi to a local taxi, and where the two of them use a policy that gives a higher priority to a taxi that is nearer to a customer. When both taxis accept a request from the customer, modified information from the remote taxi can reach the local taxi but information from the local taxi cannot reach the remote taxi.

Therefore, in the extreme case, the local taxi gets the remote taxi information but rejects it because it is itself being to the customer. However, since the remote taxi does not get information from the local taxi it will not know that the local taxi is still going to pick up the customer. By the time the remote taxi reaches the location where the customer was, the local taxi may already have picked the customer up. The remote taxi will not be able to find the customer, and there is no information whether someone has already picked the customer up or not.

Example Policies	- ReplaceTupleConflictOnNearerLocation - ReplaceTupleConflictOnNearerLocationAndLowerTraffic
Context Information	Device Location Information Geographic Information Traffic Information Tuple Creation Time Information

Table 3.3: Taxi Dispatching Service Policy Components and Required Context Information

3.3 Methodology

3.3.1 Middleware Creation and its Purpose

Apart from the applications shown in the previous section, a number of other applications and synchronisation scenarios have been investigated in order to get an idea of what is required from a flexible synchronisation process. Some of the examples are taken from the ad-hoc network applications listed in [CCL03]. Table 3.4 summarises different policies and context information from the applications shown above and shows that each application needs different types of synchronisation and uses various types of context information.

Application	Example Policies	Example Context Information
Meeting Management	- AcquireNonExistedTuple	Class of User Information
Parking Network	- ReplaceTupleConflictOnHigherClass IfWithinTimeConstraint	Time of Day Information
Taxi Dispatching Service	- ReplaceTupleConflictOnNearerLocation	Geographic Information

Table 3.4: Different Policies for Different Applications

The diversity of synchronisation processes in different applications leads to a need to create a system that can support the modification of its synchronisation operation and allow different kinds of information to be used in making synchronisation decisions.

The JSFM middleware has been built to investigate the support for tuple space based applications operating in an ad-hoc network environment by allowing the synchronisation process to be altered to suit different application requirements. The middleware design is based on widely available components that provide a basic tuple space service. The Jini services and JavaSpace as discussed in section 2.2.2.2 are used for the task. Then, the other components of the middleware have been built around them to provide wireless communication support not available in the basic JavaSpace. The detailed discussion of the JSFM components is in chapter 4.

Next, the components that are responsible for making a decision during a synchronisation process (the policy engine) were built into the JSFM middleware. The Ponder policy language was adopted, providing users with a way to control the middleware synchronisation process. A number of components in the PonderToolkit such as its policy compiler and Java object generator are used to transform a policy

file into a policy object that can be used in the middleware. More information about the policy and the middleware synchronisation process can be found in chapters 5 and 6.

After the middleware was built, a number of tests were carried out to ensure that the middleware operated properly. The main processes can be divided into two parts - a synchronisation engine testing process and a performance testing process.

3.3.2 Synchronisation Testing

The synchronisation tests are intended to make sure that the middleware policy engine can be controlled by users' policies and that it makes the right decision according to their policies. To do this, a test framework and a set of testing scenarios were created.

3.3.2.1 Test Framework

The test framework was built in order to reduce the complexity of testing the middleware synchronisation engine. Since the tests were aimed at showing only the correctness of the engine, there was no need to involve the other middleware components, since they would increase the process complexity. In this way, the policy engine can be assessed using a wide range of policies in different situations quite quickly, without having to manipulate a real mobile environment to generate disconnections.

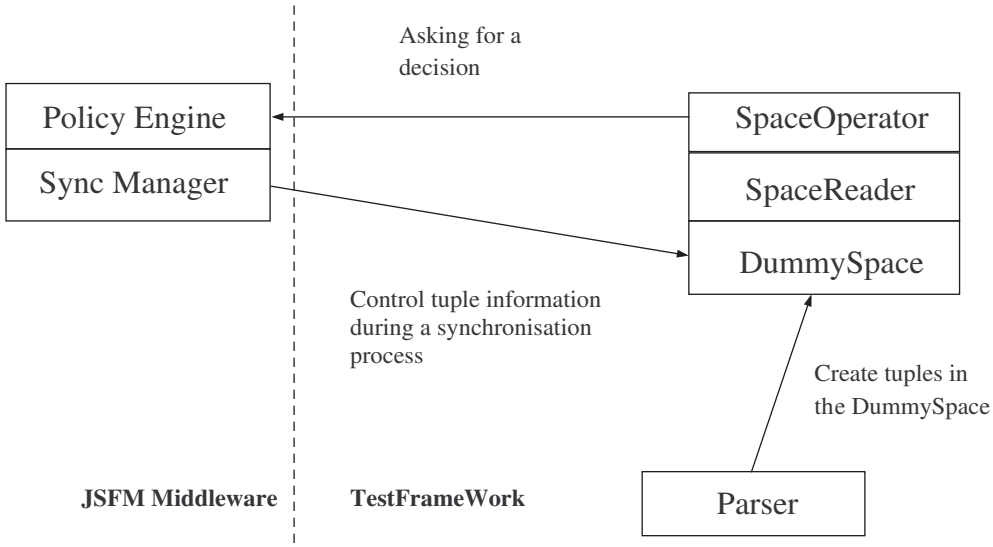


Figure 3.2: A Test Framework Architecture

Figure 3.2 shows the architectural view of the framework. Only the middleware policy engine controlling the synchronisation process and the synchronisation manager, responsible for interacting with JavaSpace, were being tested.

The framework for carrying out the test consists of a parser which obtains a scenario file from the user and creates the test environment by loading existing tuples and context information to establish the assumed state into a DummySpace. The DummySpace is a simplified object that is built to represent the JavaSpace. It does not support many of the services defined by JavaSpace, such as Transaction and Event Notification. However, it does provide basic space interactions such as read, write, and take which is enough to test the Policy Engine. Tuple information is simply stored in a linkedlist and accessed by JavaSpace-like interfaces provided by the DummySpace.

The SpaceOperator and SpaceReader are responsible for carrying out the test. They obtain information from the DummySpace and contact the Policy Engine in the middleware for a synchronisation decision. When the engine determines a suitable action from a policy, it sends the action to the DummySpace via the Sync Manager which translates the action from its policy language form to the form that can be understood by the space.

3.3.2.2 Test Scenario

A library of test scenarios has been created to be used with the test framework, each of which defines a starting state and an assumed sequence of user actions, network status changes and subsequent synchronisation events. Each scenario also states one or more self-consistent states of the shared data that would be a suitable outcome of the scenario.

The scenarios cover a wide range of situations, from the simple cases of synchronising after an extra tuple has been added to only one space, to more complex scenarios where multiple spaces have copies of an application tuple containing different information as a result of independent updates. Some of the scenarios used for testing the policy are obtained from the example applications such as the Taxi Dispatching Service explained earlier.

The policy engine and a selected set of policies can then be assessed by applying them to each scenario in turn, determining the state changes expected form applying the policies to each of the synchronisation events defined. The resulting state is compared to the set of reasonable outcomes in the scenario to determine if the policy engine worked properly and the policy can be considered successful.

Although the scenarios reduce the complexity of the application by omitting the example cases where there are errors in the application, they still represent a wide set of applications, where each has different goals and require different types of synchronisation policy.

An example of the representation of a simple scenario is given in figure 3.3. In this example, V denotes a virtual space created when devices A, B and C connect together, V.consistOf () defines by enumeration the member spaces that constitute the virtual space, and X.contains() defines the tuples in the space X.

This scenario illustrates a potential conflict between space A's point of view and remaining spaces' shared point of view since both sides edited the tuple P during the period while A was disconnected

```
Scenario
    Initial
          V.consistsOf( {A, B, C} );
          V.contains( {P} );
    Scenario
          SC =    A.disconnected();
                  ( V.del(P); V.add(Pv) ||| A.del(P); A.add(Pa);)
                  synchronisation; exit;
-----------------------------------------------------------------
Model Answer
          V.consistOf( {A, B, C} ); /* majority view */
          V.contains( {Pv} );

          /* V.consistOf( {A, B, C} ); A's point of view rejected */
          /* V.contains( {Pa} ); */
```

Figure 3.3: A sample synchronisation scenario

from the virtual space (by deleting and adding a new tuple P during the disconnection). Without further information, the most acceptable result would be to satisfy the greatest number as shown in the earlier part of the model answer section, although participant A would not be happy with this.

Instead of testing a single scenario and a single policy set at one time, the test process uses a number of files, as shown in figure 3.4, to enable a number of scenarios to be tested with a particular set of policies.

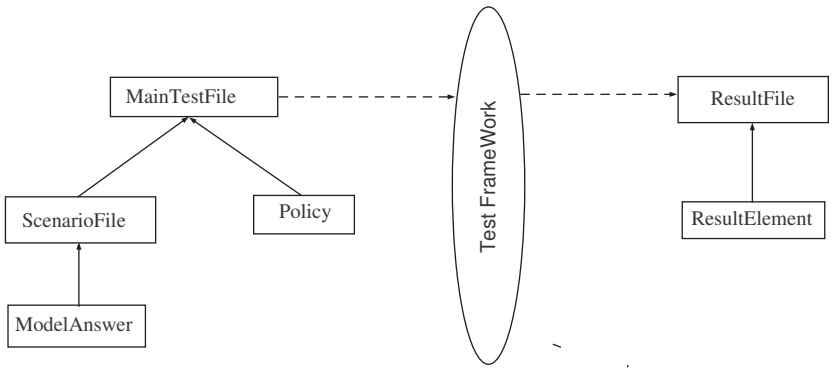


Figure 3.4: Test File Architecture

The MainTestFile makes reference to a number of the ScenarioFiles which each contain a single test scenario and a policy file to be used with the particular scenario. The scenario file contains information such as the initial state of the test environment, the testing process, and a reference to the ModelAnswer file for the test. The ModelAnswer file is a pre-defined answer for the test that will be used to check the test result.

After the test is done, The framework creates a main ResultFile that contains a brief overview of the results of the test such as the number of scenarios that passed and failed the test. Moreover, the file contains references to the ResultElement files storing detailed information for each test such as the trace of commands that were applied to the space, information about tuples in the space, and information about tuples that did not match the model answer for the test.

This framework makes the testing of the policy engine a lot easier. A large number of scenarios can be used without having to manually set up a real test environment. This reduces both the time and the cost of testing the system since it does not require a number of workstations to be used in each test, which would be required if the tests were done in a real environment. A number of scenarios have been tested with the Policy Engine to ensure that the engine works properly. The result in this test are summarised in the discussion about the effect of increasing context information to the tuple synchronisation in section 8.3.4

3.3.3 Performance Testing

Another kind of test is needed to determine the performance of the middleware. The aim of these tests is to estimate how much time the middleware takes to perform each kind of operation. For a developer, this information can be used to decide if the middleware is suitable for a particular application.

A number of middleware operations need to be tested, as shown in figure 3.5.

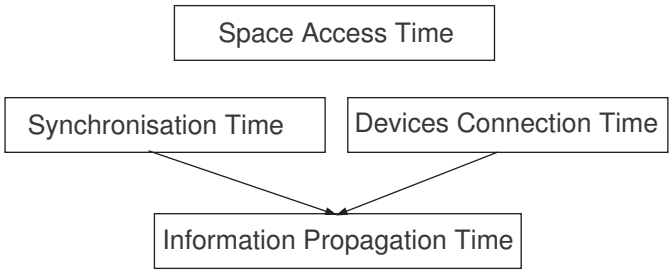


Figure 3.5: Performance Testing for the Middleware

The space access time test estimates the time the middleware takes for a single space interaction such as reading, or deleting a tuple. The device connection time test is to obtain the average time to establish a connection between devices using the middleware. The synchronisation test aims to obtains the time the middleware needs for synchronising one tuple in different situations.

The information propagation test is done to approximate the time information takes to converge. This is the time for a single tuple introduced into an area to be propagated to every connected device in that area. However, time and resource limitations prevent the actual testing of large scale configurations. Therefore, an approximation is calculated. The approximated propagation time can be obtained from

the measured information about individual operations such as the devices connection time and the single tuple synchronisation time.

More information about the performance tests and their results can be seen in chapter 7.

3.4 Conclusion

This chapter has discussed the aim in building the JSFM middleware and policy language. It gave a number of example applications that require different types of synchronisation processes controlled by different policies using different context information.

From analysis of policies that are used in different applications, there are a number of basic policies that can be used in many different applications such as a policy that makes a decision depending on only a tuple's creation time. These policies do not usually need context information that is specific to the applications and so are simple to implement. However, to make the synchronisation process efficient, a more application-specific solution will eventually need to be used.

For example, a simple application such as the meeting management system may not require a complicated policy for its synchronisation process. On the other hand, a more complex application such as the taxi dispatching service requires a more complex policy to make it efficient.

The chapter has also illustrated the methodology that is used in this project. It explained the process by which the middleware was built. It also concentrates on explaining how the system has been tested which is one of the most important processes. The test process is divided into the synchronisation testing process and the performance testing process.

The chapter gave the idea of the test framework which was used to test the middleware synchronisation engine without having to manipulate a real synchronisation environment. A number of test scenarios obtained from a number of real applications and scenarios were used in order to test that the engine built for the middleware can control the synchronisation processes correctly.

Lastly, the chapter gave a brief introduction to the performance testing process used to obtain information concerning the middleware's actual efficiency in various situations. The tests cover the information access time, the devices connection time, the synchronisation time, and the information propagation time.

Chapter 4

Middleware Architecture

4.1 Introduction

The JSFM is a mobile middleware built at the University of Kent as a system that can be used to examine the behaviour of a set of policies in synchronisation processes in an ad-hoc wireless environment. The middleware is built from several components working together. It is responsible for providing the appearance of a virtual space to an application, and so it has to provide several services such as device finding, tuple transfer over the wireless network, and the synchronisation service itself.

This chapter explains in detail the architecture of the middleware and describes each of its components. The first section of this chapter gives an overview of the middleware. This section explains the basic ideas that the middleware is built upon. The next sections gives detailed information about each major component, saying what their responsibilities are and how they work together to create a system that can provide a virtual space image to the application when using an intermittently connected wireless ad-hoc network.

4.2 Middleware Architecture Overview

To explain the middleware architecture, this first section gives an overview of the middleware. It starts with the set of existing components that the middleware relies on. Then it gives a short explanation for each of the middleware's main components. Lastly, it explains the different types of tuple used in the system.

4.2.1 External Components

The JSFM is built to satisfy the various requirements defined in chapter 3. Since the JSFM is built to provide a virtual space service, the first step is to identify a set of basic pre-built systems that are

going to be its basic infrastructure. The first component needed is an implementation of the space service. There are several pre-built space services available, such as Sun's JavaSpace [Sun98] and IBM's TSpace [WMLF98]. The JavaSpace is part of Sun's Jini technology which provides several network communication services. A list of the Jini services that are used by the JSFM is given here:

1. Lookup service (Reggie) - the Lookup service is a central service that allows other services to register a reference to themselves and to acquire a reference to other services.

2. JavaSpace service (Outrigger) - the JavaSpace service provides tuple space storage to the middleware. It also provides the JavaSpaceAdmin interface that allows the middleware to access its administrative interfaces. The main administrator interface used by the middleware returns the AdminIterator that can iterate over a set of tuple in the space.

Another service that is needed by the JSFM is a policy parsing and compiling system. The JSFM uses the Ponder language [DDLS01a] to express policies to control the system. The language has a supporting toolkit that can be used to compile and build a Java object for each policy. This toolkit is applied in the JSFM to parse policy files and create objects. In the JSFM, this is accessed by the Sync_Object, which acts as an enforcement point for policies, as will be explained later.

4.2.2 Architecture Overview

The JSFM can be divided into three different parts. The first part is responsible for initiating the JSFM and providing the set of interfaces used by an application to interact with the middleware. The main component in this section is called Port_to_Space (PTS). This component provides central control of the middleware and starts each of the middleware components during the middleware initialisation process. After this process is finished, the component provides a set of space interfaces to an application by allowing the application to call a set of methods providing space operations. Additionally, the JSFM is responsible for controlling the tuple factory components. These components receive an application level tuple from the Port_to_Space object and encapsulate it together with middleware level information.

The second part in the JSFM is the Policy Engine (PE) which is the part that is responsible for the synchronisation process. The components in this part can be further divided into two groups - the components that are responsible for processing policies, making decisions and controlling the synchronisation process, and the components that do the actual low level synchronisation. For the components that are responsible for low level tuple synchronisation, the main components in this part are the Sync_Manager and the Sync_Object. The Sync_Object is responsible for transferring tuples between a local space and a remote space. Each Sync_Object is only responsible for the tuples created by a single writer.

The Sync_Manager has control over all the Sync_Objects. Its main task is to make all the Sync_Objects work together properly. The Sync_Manager also provides a set of interfaces to the Policy_Engine to let

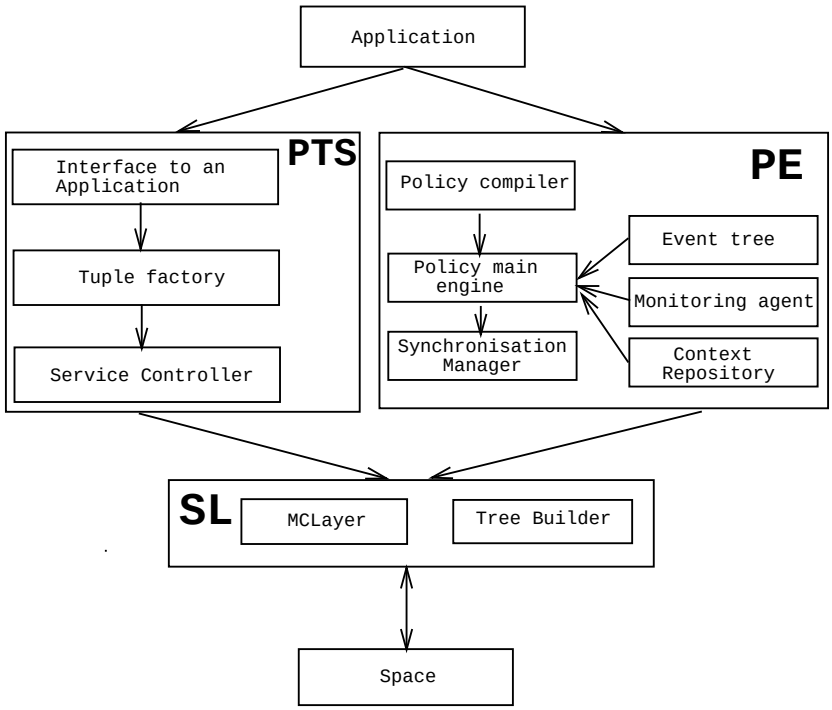


Figure 4.1: The JSFM Architecture

the engine send a set of primitive commands to control the synchronisation behaviour. These commands will be processed by the Sync_Manager and sent to the Sync_Object.

The third part of the JSFM is the Synchronisation Event Distribution Layer (SL) which is responsible for building and maintaining a synchronisation event distribution tree. The MCLayer is the main component in this part, and it coordinates with the MCLayer components from other nodes to form the tree. After the MCLayer is initialised by the Port_to_Space, it repeatedly broadcasts to locate an available event distribution tree so that the node can participate in the tree.

4.2.3 Middleware Level Tuples

The middleware relies on a set of tuples to store its information. The tuples are not only used as persistence information storage for the middleware but also for communicating information between the devices which are connecting together for synchronisation. This section explains the function for each middleware level tuple used in the JSFM.

1. SeqMessage tuple - the SeqMessage tuple is an abstract tuple used as a superclass for the InMessage, CommandMessage, and TombStone tuples. The tuple contains two fields that are used in the middleware to query for a specific tuple; these are the tuple sequence number and the tuple writer. Tuples written by the same writer will have different sequence number. A writer name is derived

from the name taken from the user during the middleware start up process concatenated with the device's IP address to make the name unique.

2. InMessage tuple - the InMessage tuple encapsulates an individual application level tuple. The information in the InMessage tuple is used by the middleware in order to find a specific tuple without knowledge about the tuple's application context. With some modification to the query process in JavaSpace, the tuple encapsulation process allows the middleware to be able to search for a specific tuple using either the middleware information or the original information in the application level tuple.

3. CommandMessage tuple - the CommandMessage tuple is used to communicate a command between spaces during the synchronisation process. At the current stage, the deletion command is the only command that is sent in this way. In order to notify other spaces of tuple deletion, a CommandMessage tuple carrying a deletion command is written into the space after the tuple is deleted. The CommandMessage tuple contains the deleted tuple's sequence number and its writer name, so that a space that receives the command can locate the tuple to be deleted.

4. TombStone tuple - the TombStone tuple is used to represent a tuple that is deleted. Once an InMessage tuple has been taken from a space and a CommandMessage tuple issued, a TombStone tuple containing the same sequence number and message writer as the deleted tuple is written to the space. The TombStone is important in the synchronisation process because it helps distinguish a tuple that has been deleted from a tuple that has not yet been written. It can also be used to store a deleted tuple's information in case the deletion process needs to be rolled back.

5. Connect Service List(CSL) tuple - the CSL tuple is used to store remote writer information. In the middleware, tuples are divided into sets based on their writer name, which is a field in the SeqMessage. Each tuple set has a separate sequence number. This implies that a tuple with sequence number equal to "1" may exist in several sets with different writer names. They represent tuples from different devices. For each space there is one CSL tuple keeping track of tuples from different writers in the form of a pairing between the maximum tuple sequence number that currently exists in the space and the space's name. After every process that involves writing a tuple to a space, either locally or remotely, information stored in the CSL tuple needs to be updated. When a new tuple is added into a space, the maximum sequence number for the writer that writes the tuple is increased. The information is used during the synchronisation process when a space checks its CSL tuple against a remote space's CSL tuple so that it can selectively acquire only those tuples that are not in its copy of the space.

6. Context tuple - the Context tuple stores context information which is used by the PolicyManager in order to make decisions during a synchronisation process. Every Context tuple must inherit

from Context_t class which is an abstract context tuple that defines essential common information. Adding a new type of context information can be done by creating a new sub-class of Context tuple. More than one context tuple can be associated with one InMessage tuple. Each context tuple contains different types of context information. Examples are:

(a) The Access_Context tuple that stores access information such as the last access time and the number of times the tuple has been accessed.

(b) The Space_Context tuple that stores context information relating to the space as a whole, such as a class of service for the space.

(c) The TombStone_Context that stores context information about a deleted tuple such as a time the tuple was deleted. The information in this tuple also helps the Policy Engine in determining whether two tuples are derived from the same equivalent tuple.

Figure 4.2 shows the life cycle of an application level tuple.

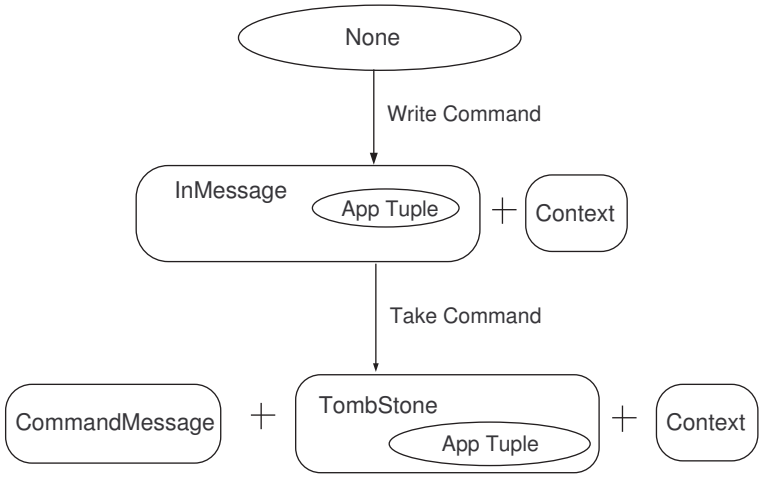


Figure 4.2: The tuple life cycle in a space

First, an application level tuple is sent from an application to the middleware. The middleware encapsulates the tuple in an InMessage tuple and writes the tuple to its local space together with a Context tuple relating to the InMessage tuple. The tuple stays in the InMessage form until either the local space or any remote space deletes the tuple in response to a take command from an application. An application can issue a "read" command in which the application tuple will be decapsulated and sent back to the application. When the tuple is taken, a CommandMessage tuple and a TombStone tuple are written to the local space together with a Context tuple relating to the deletion. More information about the context information gathering and garbage collection can be seen in 5.6

4.3 Middleware Architecture

In 4.2.2, an overview of the middleware architecture was presented. This section follows up by explaining each component in detail. The section is divided into three main sub-sections following the structure in 4.2.2.

4.3.1 The JSFM Central Components

4.3.1.1 Port_to_Space Mapper

The Port_to_Space is the central component of the middleware. It communicates directly with an application via a set of tuple space-like interfaces and with a group of JSFM components listed here. When it is initialised, it starts by creating a UnicastRegister component in order to search for a JavaSpace service operating on the local device. Next, it creates a number of message factories responsible for creating the middleware level tuples presented in 4.2.3. It then starts up the MCLayer component controlling a synchronisation event distribution tree, the Monitoring_Management component responsible for gathering and storing context information that needs constant monitoring, and the Policy_Manager controlling the policy and synchronisation-related processes.

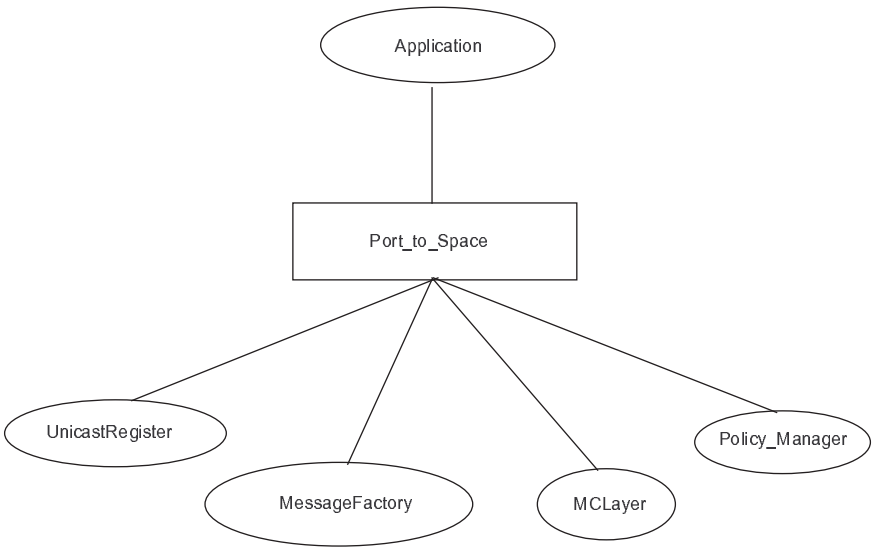


Figure 4.3: The Port_to_Space and its related components

After the Port_to_Space initialisation process is finished, the component still has two main tasks - providing tuple space-like interfaces to applications and notifying the Lookup service and the JavaSpace service to change their attached address when it detects a network disconnection and reconnection.

The operations for interacting with a tuple space are “write”, “read”, and “take”. There are also

"readIfExists" and "takeIfExists" operations which are similar to "read" and "write" except that they do not wait (unless a matched tuple is in a transaction and it is necessary to wait until the transaction state is settled) while the first three commands wait until the matched tuple exists or a specified timeout expires.

The "read" command is the simplest. The Port_to_Space encapsulates a tuple representing a query from an application into an InMessage tuple with every middleware level field set to null so that the middleware level fields are not used in the query, and sends this tuple to the local space using a JavaSpace "read" command. If the command returns a tuple, the tuple will be removed from its encapsulation and sent to the application. Furthermore, an Access_Context tuple for the read tuple will be written to the local space to update the number of times the tuple has been read.

The "write" command is more complex. First of all, a tuple received from an application is sent to the InMessageFactory which will be discussed in 4.3.1.2. The factory returns an InMessage tuple that contains middleware level information and encapsulates the application level tuple. Next, the Port_to_Space takes a CSL tuple from the local space and edits its information to reflect the new tuple that is going to be written into the space. The CSL is then written back to the local space following by the InMessage tuple. Lastly, the MCLayer write notification process is activated to propagate a notification to other nodes in the event distribution tree.

The "take" command is similar to a "read" command followed by a "write" command to write a CommandMessage tuple. The Port_to_Space first tries to read from the local space using a query from the application. The process is similar to that in the "read" command except that there is no context information written to the local space at this point. Next, it uses the acquired tuple to create a CommandMessage tuple by sending the tuple to the CommandFactory which will be discussed in 4.3.1.3. The new command tuple obtained from the CommandFactory is written to the local space and the CSL tuple is updated as in a "write" process. Then, the target tuple is replaced by a TombStone tuple representing the tuple in a deleted state.

The current prototype does not have a tombstone and command tuple garbage collection process. The two tuple types are used to tell other devices about a deleted tuple. As a result, the longer the tuples remain, the more chance that the command will be propagated. A good indicator for the tuples to be deleted may be when a space has already propagated the tuples to every space it registered in its CSL tuple. This is because it signifies the moment when the command should have at least reached every device the space has met before the deletion. When the condition is met, a timer that starts the garbage collection process to delete the two tuples should be triggered. This will allow some more time for the tuples to be propagated before they are deleted.

Another space-related command provided by the JSFM is the "readAll" command. This command does not exist in the JavaSpace interface as provided in Jini 2_0_001. The command acquires every tuple in a space that matches a query tuple and returns the tuples, stored in a vector, to the caller. The command is implemented by obtaining an AdminIterator object which is an iterator object from the

JavaSpaceAdmin interface that can iterate over a set of tuples that matches the query tuple.

The "readAll" command passes a null tuple to the JavaSpaceAdmin so that the interface returns the AdminIterator that can iterate over every tuples in the local space. While the iterator iterates through each tuple, any application level tuple encapsulated in an InMessage tuple is put into a vector, which will be returned to an application.

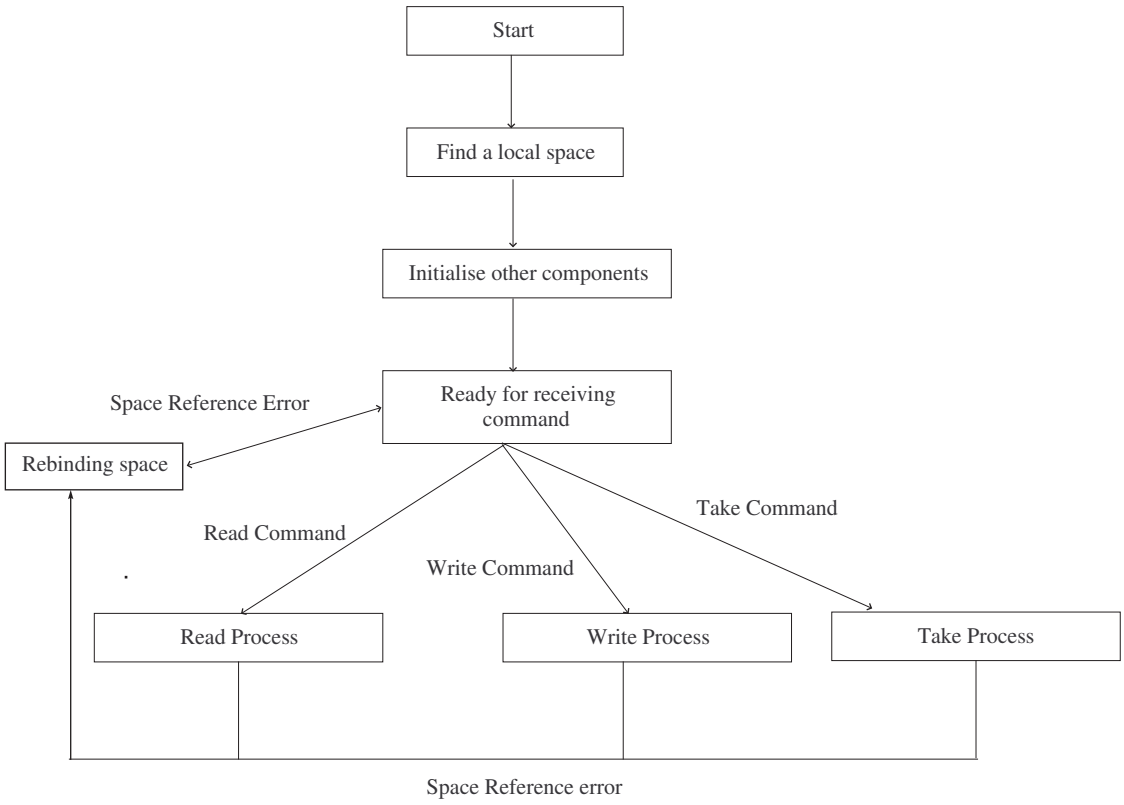


Figure 4.4: The Port_to_Space states

Apart from the space-related command explained above, the Port_to_Space component is also responsible for notifying the Jini Lookup service and JavaSpace service when there is a network disconnection. The notification is done by the Port_to_Space creating a socket for each component so that a notification can be sent. The Port_to_Space periodically queries each component for their status until their address changing process has been completed. Then, it activates the UnicastRegister again to obtain a new reference to the JavaSpace.

This process is required because the Lookup service and the JavaSpace service are attached to the IP address of the device when the middleware is initialised and the address is used in the reference used by the middleware to access the services. When a device enters a new network or is in an area that has no network connection, its IP address will be changed to either a new one or a loopback address which renders the old reference to the services unusable. Therefore, the services need to be attached to a new

address and the references need to be refreshed.

Another way to resolve the problem is to use Mobile IP [Per96], which allows a device to keep its address while it is moving. However, the technology is still not widely used and it requires an infrastructure to provide a routing which is not suitable for an ad-hoc environment in which the middleware will be used.

4.3.1.2 InMessage Factory

The InMessage Factory is responsible for encapsulating an application level tuple within an InMessage tuple containing middleware level information. The factory needs to keep track of information such as the next sequence number for the tuple to be written into a local space since this needs to be stored in the InMessage tuple when it is created. The counter used for the sequence number is shared with the Command Factory since the InMessage and the CommandMessage tuples of the same writer use the same sequence number set.

The factory can also receive an application level tuple from the Port_to_Space to create the query tuple used in a "read" or "take" command. The application level tuple is encapsulated within an InMessage tuple with every middleware level field set to null.

4.3.1.3 Command Factory

The Command Factory is responsible for constructing a CommandMessage tuple from information obtained from the Port_to_Space. In order to create a CommandMessage tuple, the Port_to_Space passes information about a tuple that is going to be deleted such as the tuple's sequence number and the tuple's writer name to the factory. The factory then uses the information to create a new CommandMessage tuple. Note that the factory obtains a sequence number for the next tuple to be written from the same counter that issues the numbers for the InMessage Factory.

4.3.1.4 UnicastRegister

The UnicastRegister is responsible for obtaining references to the various Jini services introduced in 4.2.1. After the UnicastRegister is created by the Port_to_Space, it tries to acquire a reference to the local Lookup service (reggie) using a unicast query to the local device address. Once the reference to the Lookup service is obtained, the UnicastRegister can query the reggie to obtains a reference to a JavaSpace service; the middleware needs an access to the JavaSpace interface and the JavaSpaceAdmin interface. The UnicastRegister will be activated again if the references to the services change.

4.3.2 Policy and Synchronisation Related Components

4.3.2.1 Policy Engine and Matching Engine

The Policy Engine is the main component of the JSFM concerned with synchronisation. It is responsible for processing a synchronisation unit containing information about the current synchronisation environment and making a decision about the event. The synchronisation unit will be discussed in 6.2.1. When the Policy Engine is initialised, it loads policy objects pre-compiled by the Ponder Toolkit. Whenever it receives a synchronisation unit from the Sync_Manager, it performs a decision making process by sending the synchronisation unit and a list of available policies to the Matching Engine responsible for selecting the right policy for the synchronisation unit. The Matching Engine checks if there is any policy that matches the synchronisation unit by comparing information in a policy's constraint with the information from the synchronisation unit. Once a matching policy is found, it will be returned to the Policy Engine. More information about how the Matching Engine finds the policy and how a policy object is implemented can be found in 5.5.1 and 5.4.2.

Each policy used in the system has an action part containing a command the system will follow if the policy matches the situation described in a synchronisation unit. The Policy Engine extracts the action part from the policy returned from the Matching Engine. Then, the action is translated into a set of primitive action commands. The commands are simple actions that the Sync_Manager and Sync_Object can understand. The commands are sent to the Sync_Manager for further processing. The Policy Engine can then process other synchronisation units. Examples of the primitive commands are shown in table 4.1. Currently, the Policy Engine does not support editing or adding a new action or primitive command dynamically. The engine source code needs to be edited to enable the middleware to support the new commands.

Command	Meaning
fetch_local_tuple	Obtains a tuple from a local space using specified sequence number
fetch_tuple	Obtains the next tuple from a remote space
local_write_tuple	Writes a tuple into a local space
local_delete	Deletes a tuple from a local space

Table 4.1: Primitive Command Examples

4.3.2.2 Sync_Manager

The Sync_Manager has two main tasks. The first task is to search the Event Tree for a synchronisation event type. The synchronisation event will be used as part of the information in a synchronisation unit. More information about the synchronisation event and the Event Tree can be found in 5.3.1. This process is done before the synchronisation unit is sent to the Policy Engine for the decision making process.

The Sync_Manager is also responsible for managing a number of Sync_Objects. The Sync_Objects are created and monitored by the Sync_Manager. Once the Sync_Manager receives a set of primitive

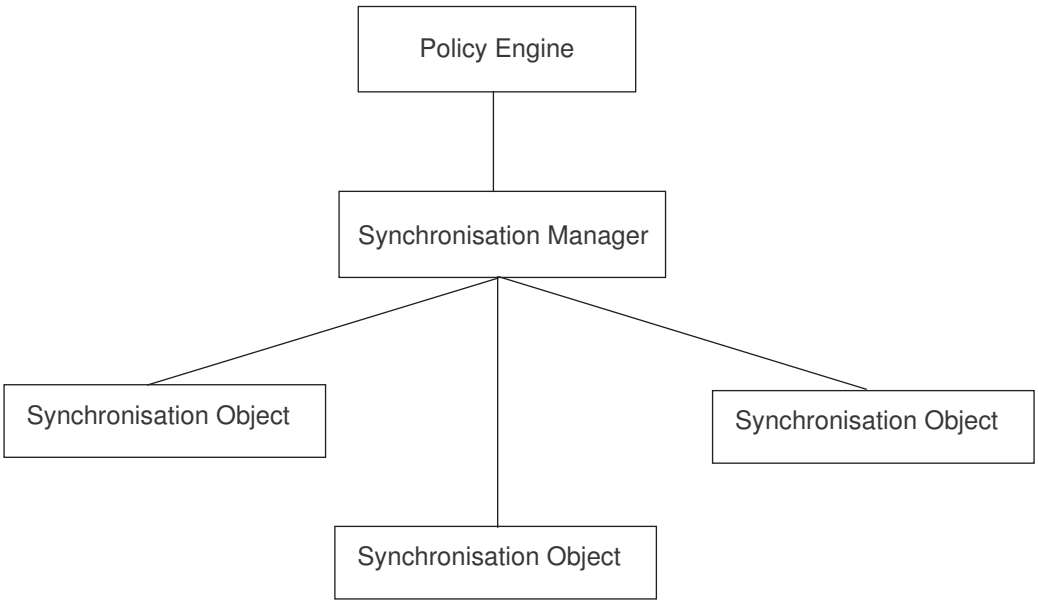


Figure 4.5: Synchronisation Related Components

action commands from the Policy Engine, it sends the commands to a Sync_Object that is responsible for the set of tuples that are targets of the commands. A new Sync_Object will be created if there is no Sync_Object currently responsible for the space.

4.3.2.3 Sync_Object

The Sync_Object can be seen as an enforcement point for the Policy Engine. The Sync_Object provides a set of interfaces to the Sync_Manager. Each of the interfaces provided by the Sync_Object corresponds to an action command used in the Policy Engine and the Sync_Manager.

The Sync_Object interacts with the local space and the remote space being synchronised using a set of JavaSpace commands. For every remote writer in the CSL tuple, there is one Sync_Object responsible for that writer. It is created when a local space interacts with a remote space and finds out about a tuple written by that writer for the first time. With each synchronisation, more than one Sync_Object might be created because a remote space being synchronised may contain tuples that it received from other remote spaces. In this situation, the remote space contains tuples written by more than one writer and requires more than one Sync_Object to take part in the synchronisation.

The Sync_Object keeps track of sequence information relating to tuples belonging to one writer. The information will be used when it receives a command from the Sync_Manager through provided interfaces. Dividing responsibility between several Sync_Objects makes the object implementation less complex and allows more of the decision making process to be moved to a higher level component such as the Sync_Manager.

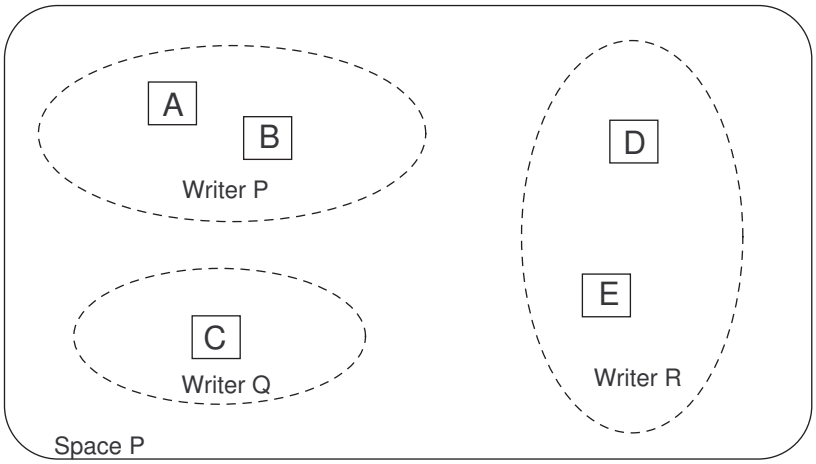


Figure 4.6: Tuples from many writers in one space

4.3.3 Synchronisation Event Distribution Layer

4.3.3.1 MCLayer

The tree building and tree maintaining process is implemented using the AMRoute protocol [XTML02] which is a protocol for creating and maintaining a multicast tree in a mobile ad-hoc network. The protocol divides nodes into two main groups - core nodes and normal nodes. Basically, a core node will always try to merge with other multicast trees in the area while a normal node will not. While connecting to other nodes, the core node periodically sends a keep-alive message to other normal nodes to notify them that the core node is still alive.

Any normal node that does not receive a keep-alive message for a set period of time will transform into a new core node and try to form a new tree. An extra node is a normal node with an extra interface that is of a different type from the interface connecting to the multicast tree. This is to allow devices with different types of interface to be able to connect to the multicast tree where there is some nodes that can act as a bridge.

When more than one tree with separate core nodes is in the same area, the core nodes will try to connect to others, and this will form connections between the trees. Any connection allows keep-alive messages from the core nodes to reach each other, which will start a core-reconciliation process. Core nodes other than the one that wins the process will be transformed into normal nodes and the multicast trees will be merged into one multicast tree. The core-reconciliation process is currently done using the IP address to determine which node is going to be the core node. However, this could be changed to use other device properties such as the type of device, battery power, or the average time the device stays in one place to make the multicast tree formed more stable.

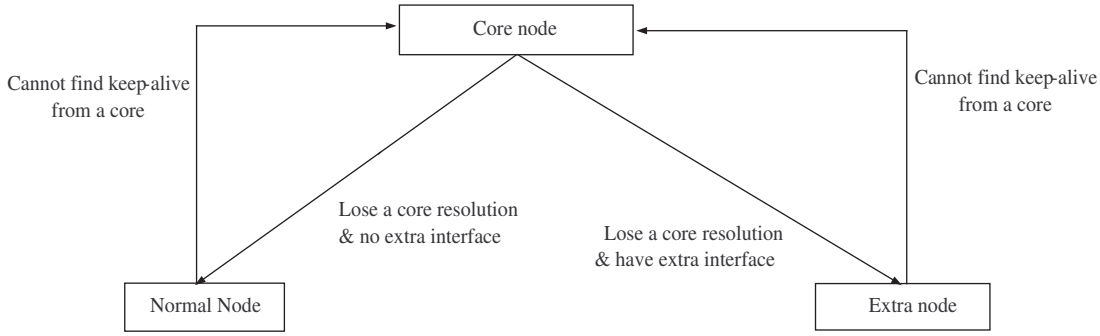


Figure 4.7: Node type in the Multicast Tree

The MCLayer is the main component responsible in building and maintaining the middleware synchronisation event distribution tree, using the process explained above. Most of the algorithms for building and maintaining the multicast tree are also implemented in this component.

4.3.3.2 Keep-Alive Related Components

The KeepAlive class defines a keep-alive message to be sent between nodes. The keep-alive message contains two kinds of information - core node and intermediate parent node information. The information about the core node of the tree contains the address and the name of the core node. The intermediate parent part is changed every time the message is passed from one node to another node. The immediate parent node information contains information such as the node's IP address and its level in the tree. The information is used when a node selects and forms a link with an existing tree.

A core node in an event distribution tree sends a keep-alive message to neighbouring nodes in the tree. Each node receiving the message from its parent stores a copy in its MCLayer. Then, it passes the message on to its children.

The KeepAliveSchedule class is responsible for ensuring a node is still connected to an event distribution tree and a core node exists somewhere in that tree. The task is done by the object periodically reading and deleting a keep-alive message sent from the core node via the tree. After each read, the class deletes the copy of the message stored in the node. During a period when no keep-alive message is received, the KeepAliveSchedule class will not be able to read the message. This situation causes the object to suspect that either the local node is disconnected from the tree or there is no core node in the tree.

In this situation, the KeepAliveSchedule object triggers the CoreSchedule timer responsible in changing a local node's status to a core node. A time is randomly chosen and when the time is reached, the CoreSchedule changes the local node's status to a core node. If the process causes more than one core node to exist in one tree, the core-reconciliation process in the MCLayer will detect this and resolve the

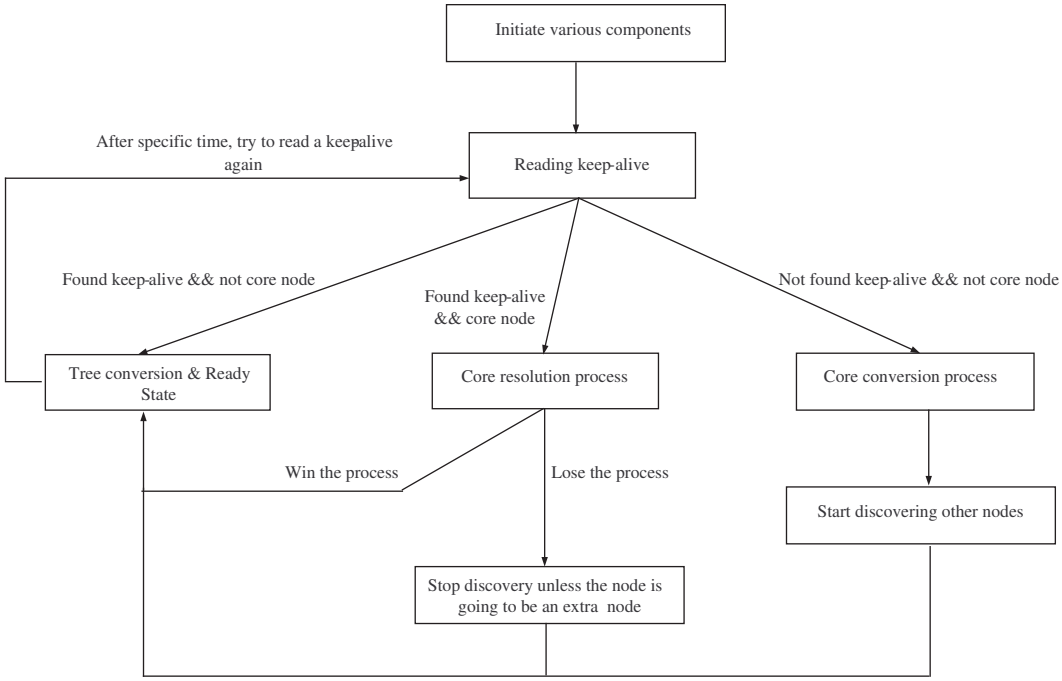


Figure 4.8: Synchronisation Event Distribution Layer Process

core conflict.

4.3.3.3 Synchronisation Notification Sending

Since one of the reasons for using an event distribution tree is to control when and where to send a notification, another important aspect of implementing the event distribution layer is to allow messages to be sent using the tree layer. When an application writes a tuple into its local space through the middleware, the Port_to_Space notifies the MCLayer about the updated information. The MCLayer then tries to obtain a CSL tuple from the local space, which will contain the most up-to-date information about tuples in that space. Next, the MCLayer creates a Notification object, which contains the CSL tuple and the IP address of the device. The object will be sent through every tree link from the node to other nodes in the tree.

Each node receiving the Notification object checks that it carries up-to-date information, by comparing the CSL tuple it obtained with its local CSL tuple. If the CSL obtained contains more up-to-date data, the node uses the IP address obtained to connect to the remote space using the discovery service on that machine. Once it obtains a reference to the remote space, it can pass the information to the Policy Engine to start a synchronisation process. Once a node has received a notification message, it has to finish the

synchronisation before it can process another notification message. This is to protect data consistency during the process.

After the process is completed, the node propagates the Notification object together with its CSL tuple and its IP address via every other tree layer link except the one it received the notification from.

4.4 Policy Engine and Other Components Relation

Since the engine is a part of the middleware and needs to communication with the other components, the section describes the relation between the engine and components such as the features in the Port_to_Space mapper and the MCLayer that the engine relies on. This information will be useful if other components are to be modified or the engine is moved to be used on other systems.

The engine relies on the Port_to_Space mapper only on the address changed notification function. This is used when the engine cannot directly access the JavaSpace service because the device network address has been changed. However, the engine heavily relies on information that is generated from several message factories which are part of the mapper. For example, the information that is attached in middleware-level tuples such as the sequence number and the message writer are used to control a synchronisation process. Moreover, the engine heavily relies on the CSL tuple in order to fetch the correct tuple for the synchronisation process.

The engine can directly access the JavaSpace service without relying on the mapper. Therefore, as long as the mapper still provides proper synchronisation information within the middleware-level tuples and the address changed notification function, its other features can be changed without affecting the Policy Engine.

The Policy Engine relies more on the components in the MCLayer than it does with the Port_to_Space mapper. An activation of a synchronisation process in the engine relies on a notification message it receives from the MCLayer components of another node. The Policy Engine needs information sent with the notification message such as an address of the remote node sending the message and its CSL tuple in order to do a synchronisation. Also, when the engine finishes a synchronisation, it has to notify other nodes in the tree. The engine relies on the MCLayer to carry out the notification sending process.

Therefore, the tree building and maintaining function in the MCLayer can be changed without affecting the Policy Engine. However, its notification sending process and the information that is sent in the process has to remain the same to not affect the engine.

4.5 Conclusion

This chapter described the detailed architecture of the JSFM. It listed the external components the middleware operates on. The chapter introduced all the types of middleware level tuple containing information the middleware needs for synchronisation. Then, it gave an explanation of each of the main components of the middleware from components responsible for linking between the middleware and an application to components responsible for building and maintaining a synchronisation event distribution tree.

This chapter only explained the parts of the middleware but did not discuss any other aspect of it. More information about policy and detailed implementation of policy-related components can be found in chapter 5.

More information about a synchronisation process provided by the middleware and a discussion relating to the process is in chapter 6. An information obtained from the testing done on the middleware is in chapter 7 and further discussion of the middleware can be found in chapter 8.

Chapter 5

Policy in the Middleware

5.1 Introduction

The middleware is divided into three main parts - the middleware interface, the middleware networking layer, and the middleware decision engine. The first two parts were described in detail in the chapter 4. This chapter discusses the middleware decision engine, especially the components that are involved in the decision making process when using a policy.

The middleware adopts the Ponder language to control its behaviour during its synchronisation process. However, only part of this language is used since this project is only concerned with controlling the synchronisation mechanism; only obligation policies are used. Other Ponder policy types such as access control, delegation, and refrain policies are not used.

This chapter discusses policies and their implementation in the JSFM. The chapter starts with a discussion of application level identity, which is used for identifying a tuple identity clash. Then, it gives a description of the policy structure in the middleware. The next section describes an implementation of the two parts of a policy. The last section discusses how context information can be used and how context information-related components are implemented.

5.2 Application Level Tuple Identity

In the original work on tuple spaces, a space was an associative store, and there was no independent concept of tuple identity. If a tuple is removed with a take operation, then updated and a written back to the store, the result is a distinct tuple. To be able to base synchronisation behaviour on the history of a tuple through a series of changes, a stronger concept of its identity is needed.

Tracing the thread of evolution can be based on the content itself, but this may progressively be replaced, changing the identity. Indeed, changes may result in the convergence of the history of two

tuples into one, and this is a particularly difficult case for synchronisation to handle. It needs to be possible for an application to identify its view of a particular tuple over a longer period.

The identity is constructed from a group of fields in the application-level tuple, which together have a unique set of values in each tuple. For example, a tuple representing an appointment object in a calendar might use the date and time slots as its identity, since these values, when combined together, represent a slot in a calendar and each combination is unique.

The concept of tuple identity is thus extended from the query processing performed when searching for a tuple. Each tuple needs to have an identity to allow an application to retrieve it, based on matching selected fields. The equivalence class established by this matching process can be reused to define the necessary concept of identity, so that the middleware can track each tuple, even though it may have been deleted and written back into the space.

The middleware can then use this concept to identify clashes that occur when two tuples have a similar identity in different spaces. More detail about the conflict detection process is given in 8.3.2. Since the middleware cannot understand what fields are used for defining an application level identity for each type of tuple, it has to provide an interface so that the identity can be defined.

Using the identity in this way makes the system simple but is not a perfect solution. This method of conflict detection cannot detect a conflict in an application which has a tuple identity that is not a precise value. The calendar is a good example for this situation. If one user creates a meeting between 9.00 to 10.00 and another user creates a meeting between 9.30 to 10.30, the system will not detect the two tuples as being in conflict. Also, there is an issue of manually changing the tuple identity before the synchronisation process, which will be discussed in 8.3.2.

The application level identity is defined in the same file that is used to define the event type definition, which will be discussed in the next section. An entry is added to the file for each type of application tuple. Figure 5.1 shows an example defining the identity for the calendar tuple discussed earlier.

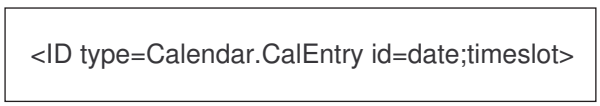


Figure 5.1: An Entry for an Application Level Identity

Once there is this concept of identity, the middleware can maintain a trace for each tuple, giving a history of all the actions performed on that tuple. A policy can then be used to decide the behaviour of the middleware during the synchronisation process on the basis of this information.

5.3 Policy Components

In the JSFM synchronisation process, the Policy Engine relies on segmenting the policy into two parts when making a decision - an event and a policy constraint. In this way, the two parts tend to be complex and should be separated to make the policy easier to read.

```
inst oblig pol1
{
  on                  sync_time >= 0 and
                      local_tuple <> null and remote_tuple <> null
                      and local_tuple.type == remote_tuple.type
                      and local_tuple.ident == remote_tuple.ident
    subject  <space>   s = /space;
    target   <tuple>   t = /tuple;
    do                 s.retrieve();
  when                /tuple -> exists(t1, t2 | t1.localtuple = true and
                                  t2.remotetuple = true and
                                  t1.writtenBefore(t2));
}
```

Figure 5.2: A simple policy with no division

Figure 5.2 shows a simple policy to resolve a conflict between two tuples. Even in this policy, the event part and the constraint part are both quite complex. Therefore, the policy in the JSFM is divided into two parts - a synchronisation event definition and a policy body - which will make the policy easier to read. This section discusses how the synchronisation event part can be separated from the policy body, how an event tree can be built, and what remains in the policy body.

5.3.1 Synchronisation Event and Event Tree

In a synchronisation process, different synchronisation requirements can be seen as different event types. For example, a synchronisation where one space has an extra tuple can be seen as one type of event and a synchronisation where two spaces each have a conflicting tuple can be seen as another event type. These event types require different actions to achieve synchronisation.

Even though they are different in detail, most of the events depend on the same set of conditions related to the basic synchronisation information which can be detected by the Sync_Object. The Sync_Object uses the basic synchronisation information in order to find the right synchronisation event name from the Event Tree. Table 5.1 shows an example of basic synchronisation conditions used in the middleware.

Defining a synchronisation event in the Event Tree can be done by combining the synchronisation conditions with logical “and” operations. For example, using the three example conditions in table 5.1 during a synchronisation between two devices, the first condition is true when no local tuple exists with

Name	Condition	Description
local_tuple	local_tuple == null	a local tuple participates in the synchronisation
remote_tuple	remote_tuple == AnyType	a remote tuple participates in the synchronisation
FirstSync	FirstSync == false	the first time there is a synchronisation between the two spaces

Table 5.1: Example primitive event in the middleware

the same tuple identity as a remote tuple. The second condition is true when there is an extra tuple in the remote device. The tuple can be of any type. The last condition is true if the synchronisation between the two devices is not the first synchronisation to take place between them. An example of an event composed from the three conditions is a synchronisation event triggered when there is an extra tuple on the remote device (of any tuple type) and the two devices have synchronised at least once.

The Event Tree is defined before the system is initialised. Each node in the tree contains a definition of a synchronisation event. Currently, the tree is defined using XML. A definition of a node is divided into two main parts - a node definition in terms of synchronisation condition and the definition of its children. This structure is used recursively to define nodes in deeper levels of the tree.

An event in the tree is always a super-type of events in its child nodes. This restriction is made so that searching for an event can return the nearest super-type event when there is no specific match for a particular event. In this way, if an event cannot be matched, the policy for the closest possible event to it is used.

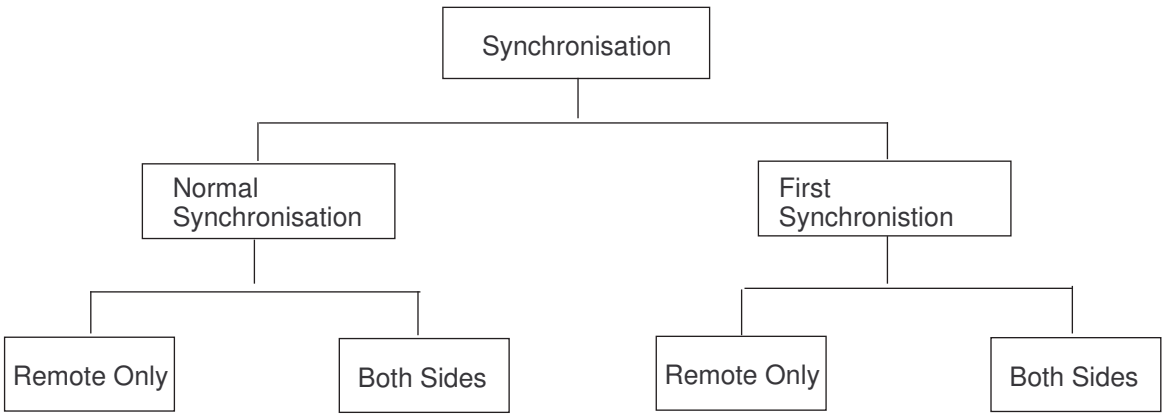


Figure 5.3: An example of an Event Tree

Figure 5.3 shows an example of a basic Event Tree. During any synchronisation, after the Sync_Object finishes gathering basic synchronisation information, it compares the information with the conditions defining nodes in the local Event Tree until it finds an event type that is compatible with the information. A breadth-first search for the closest match will return an event name, which is then used to search for the right policy.

The root node represents a super-type for every synchronisation event. Basically, every synchronisation type that does not match any other child node will match the root node. In figure 5.3, the node labelled "Synchronisation" represents a general synchronisation. It has two child nodes. The First_Synchronisation node represents an event where synchronisation happens between two devices for the first time, while The Normal_Synchronisation node is a synchronisation event where two devices have met before.

5.3.2 Policy Body

The policy body encompasses the other policy elements - subject, target, action, and constraint. Each component apart from action allows a policy writer to use a feature call written using the Object Constraint Language(OCL) to help the Policy Engine decide whether the policy should be used to control the middleware behaviour during a particular synchronisation process. However, this project only supports a limit set of policies that relate to tuple synchronisation using the JSFM.

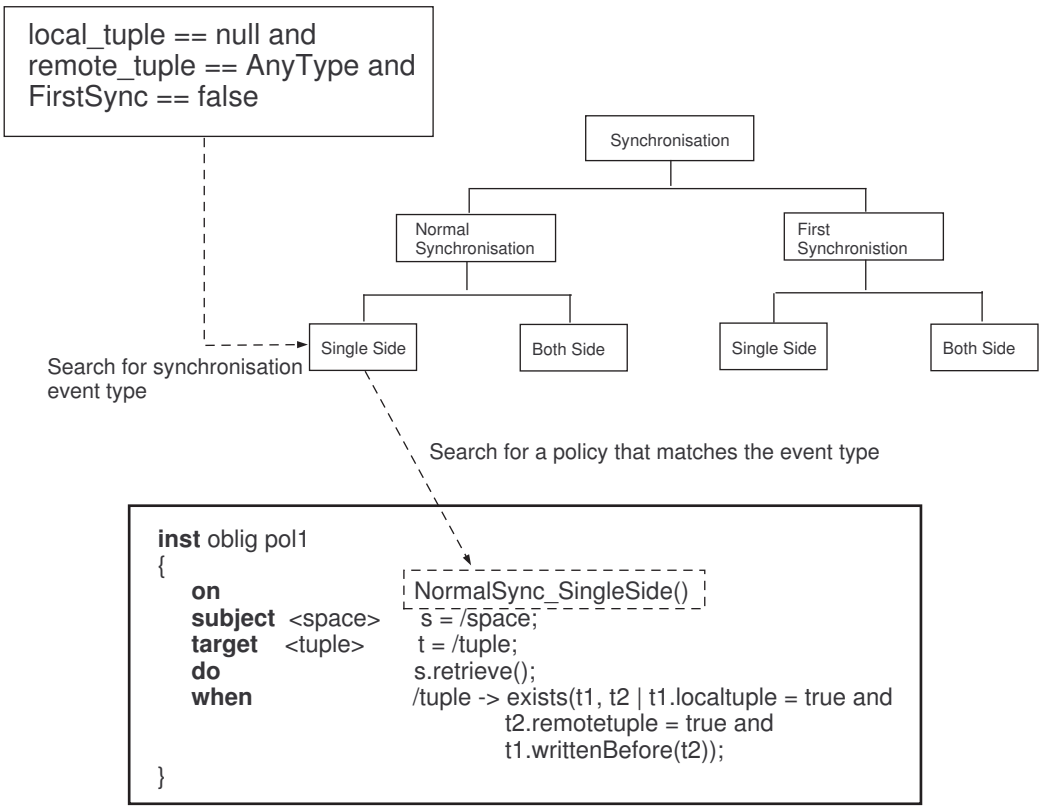


Figure 5.4: Relation between a policy body and an event tree

The subject component always represents a device participating in the synchronisation. It can be

used to filter a subset of the devices to be controlled by a policy using context information such as the type, identity or class of device.

The target component in a synchronisation policy is always the tuple that is being synchronised. Like the subject part, the target component can be used to filter whether a tuple is to be controlled by the policy using tuple-related context information. The last, and probably the most important part is the constraint. Both space-related, tuple-related and environment-related context information can be used in a constraint to control a policy usage.

To start the synchronisation process, the Sync_Object searches the Event Tree on the local device and obtains an event name; a synchronisation unit is then created from the information available. The Policy Engine then receives the synchronisation unit from the Sync_Manager and starts the decision making process.

During the process, the Policy Engine compares the information in the synchronisation unit with each existing policy stored in a local policy repository. Policies with an event part that matches the event in the synchronisation unit are interpreted. Each component in the policy body in each loaded policy is evaluated using information from the synchronisation unit, until the Policy Engine finds a policy that it is satisfied. Finally, the action part is extracted from the policy selected by the Policy Engine, which will then be used to control the middleware via a command that is passed through the Sync_Manager and Sync_Object. More information on how policy matching is done can be found in 5.5.1.

5.4 Policy Components Implementation

The implementation of the policy components is divided into two parts - the Event Tree implementation and the policy object implementation. 5.4.1 describes the structure of the Event Tree, how it is implemented, and how it is searched to get hold of an event name. In 5.4.2, the policy object implementation explains how a policy is transformed into a policy object, how the objects are kept, and a way they are used during a decision making process.

5.4.1 Event Tree Implementation

The Event Tree is a simple tree structure built from a collection of objects of the TreeNode class. The TreeNode class contains information used for event name searching, such as the conditions associated with an event and the event name. It also contains references to the other TreeNodes that are its children.

The Event Tree class contains a reference to the root node of the tree. It contains a method that allows other objects to search the tree providing they already have information used in the conditions which is normally gathered by the Sync_Object during the initial stages of a synchronisation process.

The Event Tree has to be defined before the middleware initialisation in an XML file which will be loaded when the Sync_Manager is created. Therefore, it is currently not possible to edit the tree or any

event information in it when the middleware is operating. A discussion about the customisation of event definition in different devices can be found in 8.3.5.1

5.4.2 Policy Body Implementation

Policies to control the Policy Engine are written using the Ponder language. They are compiled by the Ponder compiler provided with the Ponder Toolkit to create Java class files. One class file represents each policy. A policy object is created when the policy class files are loaded into the JSFM Policy Engine.

A Java policy object created by the Ponder Toolkit is divided into five different parts, one for each of the five policy components (event, subject, target, action, constraint). The subject and target parts are each represented using a domain scope expression stored in terms of its parse tree. A constraint is also represented by a tree-like structure. Apart from simple expressions, the subject, target, and constraint can also contain feature call expressions. More information about the policy object can be found in [Dam01]. More information about the evaluation process in the JSFM can be found in 5.5.1 belows.

5.5 Decision Making Process

The JSFM synchronisation decision making process can be divided into two parts - a policy matching process and the action processing. The policy matching process takes each policy object stored in the Policy Engine and uses the Matching Engine to check whether the policy matches the current synchronisation situation.

The action processing is started when the Matching Engine returns a matched policy. The action part of the policy is extracted and translated into a set of primitive action commands by the Policy Engine. These commands will be sent to the Synchronisation Manager for further processing.

5.5.1 Matching Engine

The Matching Engine is the main component responsible for the policy matching process. It receives the current synchronisation requirements and a list of policies, finds the first policy that matches the situation and return this policy to the Policy Engine.

To match a policy, there is a need to have a link between conditions in the policy and an object, such as a space or a tuple, so that the component responsible for doing the task can get hold of information about a real world object while not being able to access or modify the real object itself. A proxy object is needed to satisfy this condition. It has to provide an interface that allows the Matching Engine to access context information but at the same time prevents the engine from accessing other information that is not intended to be used.

The JSFM has an “Ident” object to represent an object identifier (space, and tuple) in a policy during the policy matching process. The Matching Engine creates and stores an Ident object for each identifier.

The Ident contains two important parts - a Pol_Obj object and real world objects the Ident represents. The first component is a Pol_Obj object acting as an interface or proxy to a particular type of real world object. For example, a space can be represented by a space Pol_Obj object while a tuple is represented by a tuple Pol_Obj.

The Pol_Obj allows the Matching Engine to access libraries for each real object by presenting a set of interfaces representing the library. For example, the Matching Engine can obtains the creation time for a particular tuple in a synchronisation process by querying the Pol_Obj in the Ident object that represents the tuple. The Pol_Obj can then accessing a context tuple attached to the queried tuple and return the result to the Matching Engine. In this way, the Matching Engine is separated from a real world object and provided only with a number of libraries for obtaining information from the object. Table 5.2 shows an example of libraries associated with each Pol_Obj type.

To match a policy, the Matching Engine divides each policy into different parts, based on the policy components. The first matching process matches the event part. The Matching Engine only supports simple event matching. There is no support for temporal event matching, such as a detecting when events occur in parallel or are one after another.

During subject or target matching, the engine creates an Ident object for any object identifier found in the policy. The Ident object can be used during constraint matching so that the constraint can query information from the subject and target.

Another important part of the policy matching is a quantification. The Matching Engine currently supports two quantification types - there exist and for all. The first type looks for at least one match while the second type only matches when every instance is true. The Matching Engine iterates through each instance until it can satisfy the quantification condition or it finds that the condition cannot be satisfied.

The Matching Engine currently supports combination of terms using logic operations “and”, “or”.

Object Type	**Example Libraries**
Sync_Tuple	isRemote() isLocal() after(anothertuple) compareAccessNum(anothertuple)
Sync_Space	isRemote() localname()
Timer	before(time) between(time1, time2)

Table 5.2: Example Pol_Obj Libraries

5.5.2 Action Processing

A policy matched by the Matching Engine is sent back to the Policy Engine, which extracts the action part from the policy. The action command written in a policy is a high level command such as retrieve, or

replace, so that it is easier for a user to understand than the primitive commands used in the middleware level. The Policy Engine has to translate the command into a set of primitive action commands so that the Sync_Object can process the action.

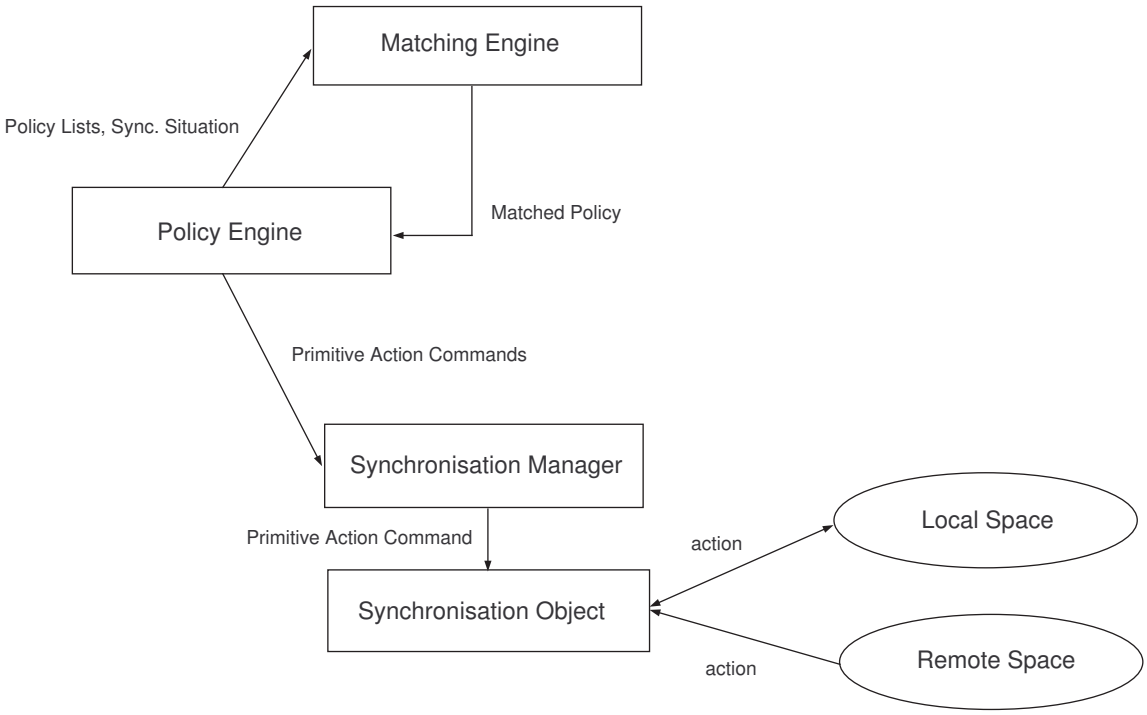


Figure 5.5: Components Participating in Action Processing

For each primitive action command sent from the Policy Engine, the Sync_Manager chooses a Sync_Object that is responsible for the tuple writer or creates a new Sync_Object if none exists. The Sync_Object processes the action; it can access both the local space and the remote space that participate in the current synchronisation. However, it can only alter information in the local space. This is to force the design decision that information transfer between devices uses a "pull" methodology.

5.5.2.1 Push or Pull Style Action Commands Discussion

During the process of synchronising two devices, moving a tuple from one device to device can be done in two ways - a tuple owner can push a tuple into another device's space or that device can request the tuple from its owner.

There are not many differences between the two ways; however, they should not both be used at the same time, because, if each individual user is allowed to choose between the two ways, a synchronisation process could occur between users that have made different decisions. This situation could cause a data inconsistency problem after the synchronisation process. During a tuple synchronisation where

one device uses the push style and another device uses the pull style, the result will be either a tuple duplication because one side pulls the tuple while it is being pushed by the other side, or no resolution process because the side that needs to get the tuple cannot pull the tuple and the other side cannot push it.

Because of this, the actions allowed in the middleware operate in a pull style based on retrieving tuples from the remote space; the middleware does not allow one space to directly affect another by using an action command in a push style.

Note that, even though system security is out of the scope of this project, using the pull style actions makes it easier to add security features into the system. This is because it would be easier for a user using push style actions to create a malicious policy that distributes tuples that are not wanted by other users. This could be done, for example, by using policies that give the local tuple the highest priority without looking at any context information.

With pull style actions, a tuple will only be distributed if the receiver side allows it to be propagated. This action style also makes the user feel that he or she has a control over his or her device and it is easier to introduce security if there is a need for it.

5.6 Context Information Gathering Process

The Policy Engine contains a number of types of context information that can be used in a policy. However, different applications may need different types of context information for making decisions during a tuple identity clash. The basic context information mechanisms for collecting the tuple, space, or time information that are provided with the middleware may not be suitable for all applications.

Therefore, the middleware needs to provide a way for a user to extend the set of context information used in the synchronisation process, and this information has to be collected while the middleware is operating. The information collected needs to be stored in persistent storage and to be available during the synchronisation process.

The JSFM contains a set of agents responsible for gathering information and an interface is provided for the agents to interact with the local space regarding the context information they are responsible for. To do this, the JSFM provides an abstract agent that a user can extend to create his/her custom agent. The abstract agent provides a method to start, stop, and store the agent. The local space stores information about which agents are activated. This information is persistent and will be used for re-activating the agents when the middleware is restarted. A user only needs to extend the abstract interface with a method for gathering information and storing it in the local space via an interface provided by the middleware.

For example, to build an agent that gathers battery power information, a user creates the battery agent class extending the abstract agent class. The only functions the user needs to define are to access the API

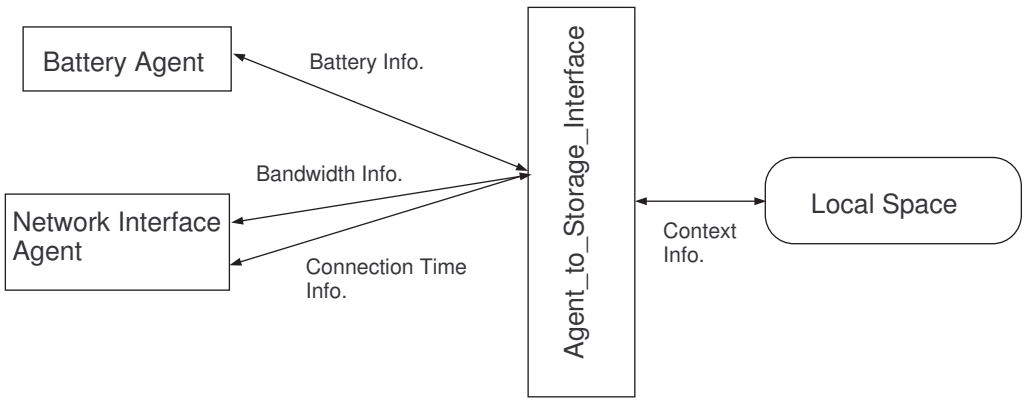


Figure 5.6: Context Gathering Components

from the operating system to obtain the information about the battery and to control the time between information gathering events.

The Agent_to_Storage interface connects each agent with the local space. Context information is identified by using an agent name and an information name. The interface allows an agent to store its gathered context information and retrieve the stored information if historical information is needed for a calculation. The storage interface is implemented so that no agent can affect user and middleware information in the local space, since each agent interaction is limited to the information it is responsible for.

There is currently no pre-defined garbage collection process. It is left to the user to define when and how the context information can be deleted, which can be done using the take command provided by the storage interface. The taken decision will affect how long the information can be used in a synchronisation process by the Policy Engine. The same mechanism used for setting an interval for the agent to gather information can also be used to make the agent do the garbage collection process.

5.7 Conclusion

This chapter explained how a policy is implemented and used in the middleware. The policy is divided into two main parts - the event definition tree and the policy body. Different events in the event tree are distinguished by a set of basic conditions. Each condition is based on the current tuple synchronisation process information.

The event and the policy body are used in a policy matching process which is performed by the Policy Engine and the Matching Engine. The process iterates through each part of the policy and checks whether the policy condition matches the current synchronisation situation. The action part of the matched policy is extracted by the Policy Engine and translated into a set of primitive action commands that can be processed by the Sync_Object.

Apart from basic context information provided by the middleware, a user can also create an agent that gathers extra context information from the environment. The middleware provides a basic framework for the agent by having each agent extend an abstract agent class containing the basic methods for storing data or starting, stopping, or restoring the agent.

Chapter 6

Synchronisation Process Discussion

6.1 Introduction

Synchronisation is the main reason why the JSFM was built. The JSFM offers a data synchronisation service between devices in order to provide an application with a unique virtual space view even if it is operating in a wireless environment, in which constant update of data between devices is not possible. There is a need for a data synchronisation process which includes data conflict detection and data conflict resolution sub processes to establish data convergence when devices are connected together.

This chapter gives an explanation of how synchronisation is done in the JSFM. The chapter is divided into two main sections. The first section discusses a synchronisation process where the same set of policies is used on every device. The second section discusses the situation where each device is allowed to have its own policy set. Each section explains the corresponding requirements for the synchronisation process, shows how the process can be done, and discusses the problems with this style of synchronisation.

6.2 Uniform Policy Synchronisation

The simplest situation for data synchronisation in the JSFM is an environment where synchronisation is done using a uniform policy set. The synchronisation process assumes that:

- Every device participating in every synchronisation in the same environment contains exactly the same set of policies.
- A device cannot add, edit, or delete its policies unless there is a system-wide policy update affecting every device in the environment.
- Every device collects and processes the same kind of context information and understands it in the same way.

The assumptions above allow the synchronisation process to be simple. There is no need for the Policy Engine to reconcile different policies or context information. However, this kind of environment limits an application to only one set of policies and prohibits any modification to the policies by an individual device. The process sacrifices synchronisation flexibility in order to reduce the problems that happen in a multiple policy environment, which will be discussed in 6.3.

6.2.1 Tuple Synchronisation

A synchronisation process happens after a device receives a synchronisation notification message from another device via the MCLayer. The message sending process is initiated either by the MCLayer after it finishes building a tree connection to another device or by the Port_to_Space after it finishes its interaction with a local space. The notification message contains information that allows a remote device to be able to connect back to the local device using services provided by Jini. Moreover, the notification messages also contain a CSL tuple which is used by the remote device to determine if it requires any update from the local device.

After a device received a CSL tuple in a notification message, it compares the tuple with its local CSL tuple to find if there are any more up-to-date tuples in the remote device. If so, it obtains each new tuple from the remote space via a command provided by the JavaSpace. For each tuple obtained, the local device builds a synchronisation unit containing:

- Event Type - The event type is obtained by comparing the current synchronisation situation to the Event Tree as explained in 5.4.1 It will be used to notify the Policy Engine of the type of policy that is to be used for this synchronisation process.
- Participating Tuples - Both tuples that are participating in the synchronisation will be sent to the Policy Engine with the synchronisation unit. This is to allow the Policy Engine to make decisions that require information from the tuples such as the tuple types or information in the tuple.
- Space References - The references to the two spaces participating in the synchronisation are sent with a synchronisation unit. These references will be used by the Sync_Manager for manipulating tuples in the two spaces.

The synchronisation unit is sent to the Policy Engine on the local device. The Policy Engine containing policy objects loaded when it was initialised and uses these to make a decision using information from the synchronisation unit and context information from the local device and the remote device. The context information is not put into the synchronisation unit like other information because of its size and some of the information may not be useful in every synchronisation. The decision making process was discussed in 5.5

A decision from the Policy Engine is interpreted in term of a set of primitive commands which we sent to the Sync_Manager. The primitive commands are a set of commands that the Sync_Manager understands. The Sync_Manager provides an interface that receives the primitive commands and their parameters. Examples of the commands are shown in table 4.1. Every command has parameters specifying a specific space and tuple that will be the target of the command. The Sync_Manager passes the command to the Sync_Object that is responsible for the remote space. After a set of commands for a tuple is undertaken by the Sync_Object and the remote tuple is synchronised, the Sync_Object has to update the local CSL tuple so that it will reflect the current local tuple status.

After each tuple is synchronised by the Sync_Object, the Sync_Manager activates a separate Context Synchronisation process to copy the context information relating to the synchronised tuple from the remote space. This process of information propagation will be discussed in 6.2.2.

After all modified tuples are synchronised, the local space creates and sends a notification message to notify another node on the synchronisation event distribution tree and starts a synchronisation process with it. This notification propagation will allow information about an updated tuple from one node in an event distribution tree eventually to reach all other devices on the tree.

Figure 6.1 depicts basic communication between devices during a synchronisation and notification propagation process. After Device B receives a notification message and A's CSL tuple, it sends a request to A for the new tuples it knows about from checking the CSL tuple. If the new tuple is accepted by B, it will request context information to accompany the tuple. After B finishes its synchronisation process, it composes and sends a new synchronisation notification message to another device on the event distribution tree.

6.2.2 Context Transfer

The context transfer process is important for synchronising information between devices. Since each policy relies on context information to be able to decide the action to take on a tuple, context information for the tuple needs to be present wherever the tuple exists.

Context transfer can be divided into two sub processes. The first process is to transfer context information along with a tuple during the synchronisation process. Context information is kept within a context tuple, as discussed in 5.6, and this will be transferred with any tuple being propagated. The context tuple life time is equal to the life time of the tuple it is attached to.

Apart from transferring the context tuple during a synchronisation process, context information on a remote device needs to be updated whenever the information on a local device is updated. The local device propagates update information through the event distribution tree, whenever new context information is written to the local space. A remote device that receives the information updates the relevant context tuples.

To make the context information updating process as fast as possible, each context tuple is tagged

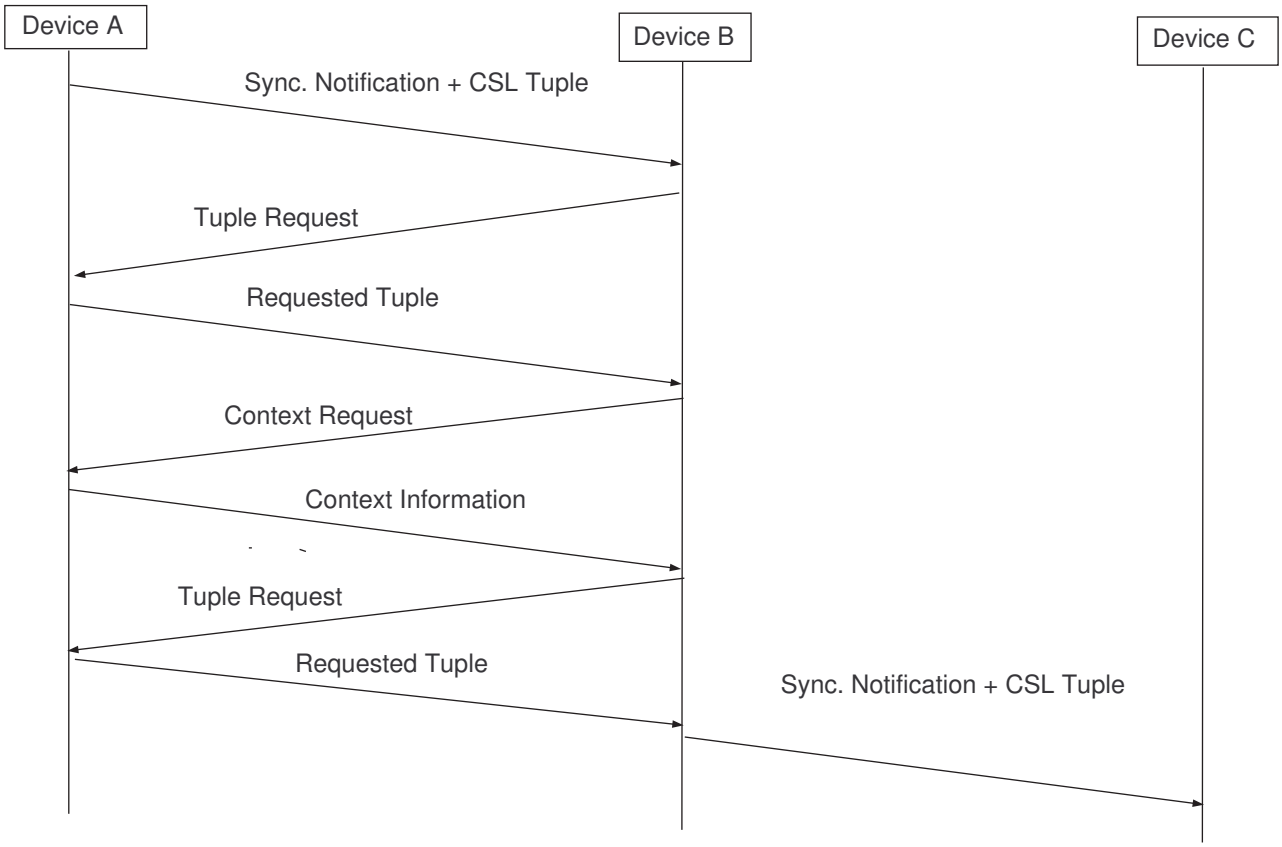


Figure 6.1: Tuple Synchronisation Process

with its version number. Basically, updating of the context information is done by simply writing the new context tuple into the space. An advantage of updating context information in this way is that it reduces the time taken for the process, which may reduce the chance of interrupting any other process that may be more important. However, the approach has a disadvantage, in that context information that is not useful anymore still takes up storage space in each space. A context information garbage collection process can be done using the context agents described in 5.6

6.2.3 Conflict Marker

One of the aims of the middleware is to provide a unique virtual space view to an application. Any data conflict that remains after a synchronisation process is going to violate the aim. The conflict marker is introduced to ensure that no data will be left unsynchronised.

A situation needing special measure occurs when a tuple being synchronised is not newly added, but results from a bouncing effect that has occurred during the tuple propagation (the detail explanation of the effect is in 6.3.3.1). For example, this situation is often caused if a device “A” that originally writes a tuple has lower priority than a device “B” at the other side of the event distribution tree. The

problem occurs if, while the tuple is being propagated to the other side, the device "A" updates its context information so as to increase the tuple priority to be higher than the device "B".

In this situation, the tuple will be turned back from B together with the conflicting tuple written by B which has higher priority. The tuple from B will be propagated through the tree back to A. At some point in the tree, there will be a synchronisation between a space containing the tuple from B and a space containing the tuple from A with its updated context information, which makes the tuple win the process. At this point, the space that supposes to replace the tuple from B with the tuple from A will not be able to see the tuple because the tuple is not a new tuple according to the information in the two devices' CSL tuples.

There are a number of ways to resolve the problem. The current implementation chooses the conflict marker because it is simple and cheap to implement; it indicates to another space that the conflict tuple has been found in a conflict. This is done by adding an extra field indicating a tuple's conflict status to the middleware level tuple (InMessage). Each tuple is created with the status set to no-conflict. During a synchronisation process, if there is an identity clash synchronisation event and the result from a policy engine is to do nothing, the conflict marker on a remote tuple will be set to indicate a possible conflict. At the end of the synchronisation process, each device participating in the synchronisation searches its space to find any tuples that are marked as possibly causing a conflict and starts a post-synchronisation process if any are found.

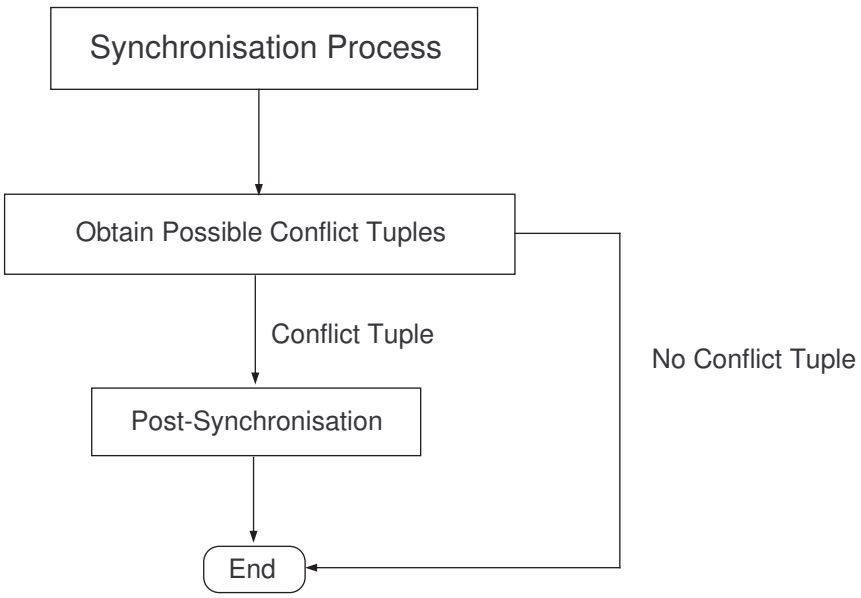


Figure 6.2: Synchronisation Process With Conflict Marker

Finding a tuple that has possibly caused a conflict can be done easily by using a space command

searching for a tuple with its conflict marker set to possible conflict status. The post-synchronisation process can be seen as redoing the normal synchronisation process with only the tuple obtained above. The post-synchronisation process checks and ensures that there is no tuple left unsynchronised.

Since this is a special situation, more attention is required to make sure that the data in every device is consistent, and a sequence number and writer name for the tuple with the marker is sent together with the notification event instead of just using a simple notification message to ensure that the other devices check that there is no data inconsistency concerning the tuple.

An alternative approach would be simply to use the detailed event notification every time context information is changed. This can also solve the problem and is easier to implement. The notification is sent every time context information attached to a particular tuple is updated, which will re-trigger the synchronisation process for the tuples that are associated with the context information. However, this process costs more than using the notification with the conflict marker because, in this case, every update, even one that does not require any attention, will trigger a synchronisation process throughout the event distribution tree.

6.3 Multiple Policy Synchronisation

In 6.2, the synchronisation process assumes that every device in an environment uses the same set of policies. However, in reality, a user or a programmer can create, edit, or delete a policy from his or her device either while it is connected or disconnected from other devices. In this environment, a user has more control over the conditions under which he or she wants to receive information from others, because they can design their policies to suit their needs.

However, allowing a user to have more flexibility in controlling their policies increases synchronisation complexity. More problems occur in an environment where several devices in an event distribution tree each try to synchronise their information using different sets of policies. This section discusses the situation by explaining what the extra problems are, what the extra requirements for the middleware to support such a situation are, and how the synchronisation can be done.

6.3.1 Multiple Policies Synchronisation Problem

Allowing each device the freedom to change its policy makes the synchronisation more complex. There are several extra requirements the middleware needs to support concerning the synchronisation process and the policies used by the process.

1. Policy Conflict - allowing users to edit their policies creates a situation where a synchronisation process occurs between devices with different sets of policy. The difference between policies implies a high chance that two devices will take different actions for the same event which will lead to them not being able to make data converge. Since the goal of the JSFM is to create an

image of a unique virtual space for an application, the conflict between policies has to be resolved before any synchronisation process can happen.

2. Failure of data convergence in the synchronisation event distribution tree - even though synchronisation between pairs of nodes with different policies can be resolved, differences between policies on different nodes can also cause data convergence problems afterwards, in the event distribution level. A basic example is where there are three devices connecting together and the middle device has policies and context information that allows a tuple from either of the other two devices to overwrite its local tuple, as shown in figure 6.3. The three devices all have tuples with the same tuple identity, but containing different information. The Synchronisation process used in 6.2 will not be able to make the information in the tuple converge. The information on the middle device will oscillate between the information it received from the leftmost and the rightmost device. More details of the problem are explained in 6.3.3.

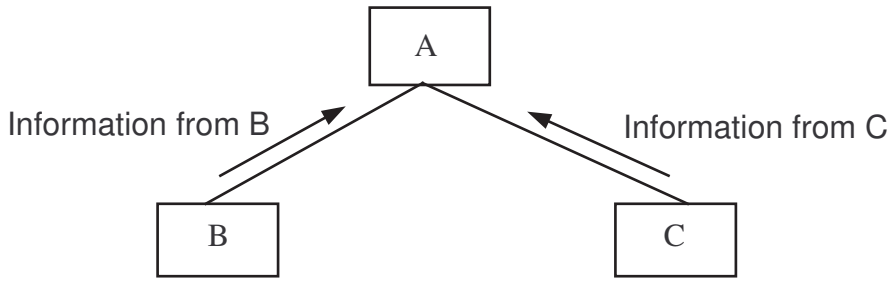


Figure 6.3: Bouncing Effect

3. Tuple Synchronisation Control Information Transfer - in a uniform policy environment, the tuple propagation process is always controlled by the same policy. However, since a tuple is always controlled by a set of policy on its host device, when the tuple is transferred between devices in the multiple policy environment, control of the tuple is changed from one device's policy to another device's policy. In this situation, no device can reliably create a tuple where content is meant to have higher priority than the others because when the tuple is transferred to a device that has a weaker policy and context information it will be overwritten by a tuple from another device that has a stronger policy than the intermediate device, even though it is not stronger than the original writer device. To solve the problem, there is a need to find a way to transfer information regarding the priority of a tuple with the tuple. This information will then be used when the tuple is on a remote device. It represents the correct synchronisation control information the updated tuple should have.

6.3.2 Policy Conflict

Policy conflict during a synchronisation process can happen when different devices are using different policies. As a result, a synchronisation process will not result in spaces that have the same set of tuple. For example, the policies shown in figure 6.4 are always in conflict with each other.

```
inst oblig pol1
{
    on                  IdentConflict();
    subject  <space>     s = /space;
    target    <tuple>    t = /tuple;
    do                  s.retrieve();
    when                /tuple -> exists(t1, t2 | t1.localtuple = true and
                                    t2.remotetuple = true and
                                    t1.writtenBefore(t2));
}

inst oblig pol2
{
    on                  IdentConflict();
    subject  <space>     s = /space;
    target    <tuple>    t = /tuple;
    do                  s.retrieve();
    when                /tuple - > exists(t1, t2 | t1.localtuple = true and
                                    t2.remotetuple = true and
                                    t1.writtenAfter(t2));
}
```

Figure 6.4: Policies in Conflict

The first policy gives a precedence to the tuple that was written most recently while the second policy favours the tuple that was written earlier. If two devices try to synchronise while one of them uses the first policy and the other one uses the second policy, the information from the two devices will both never converge. Both of them will either do nothing or they will try to replace their tuples at the same time. In order for the problem to be resolved, it should be divided into two smaller problems - policy conflict detection and conflict resolution.

6.3.2.1 Policy Conflict Detection

When solving the problem caused by conflict between policies, the first step is to understand how the conflict can be detected. Since the JSFM does not involve an access control policy, policy conflict for the JSFM can only happens during a synchronisation process and most of the policies that cause conflict are policies concerning identity clash. A conflict detection process used in the JSFM will operate in the synchronisation layer and be activated during each synchronisation process.

There are several ways for a conflict to be detected; each has its advantages and disadvantages. They

are shown in figure 6.5.

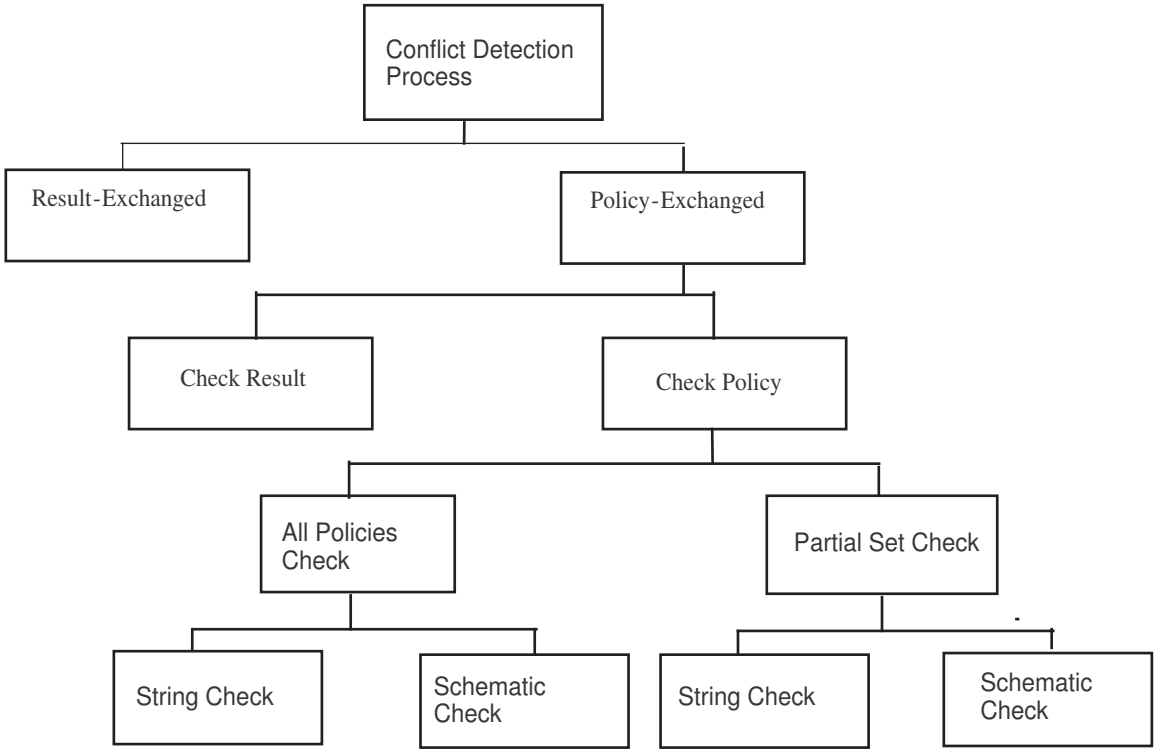


Figure 6.5: Conflict Detection Methods

In order for devices to check if there is a conflict between their policies, information has to be exchanged between them prior to the synchronisation process. Two sets of information can be used in this case - decision results from applying the policies or the set of policies themselves. In the first case, each device carries out a synchronisation process and they exchange their results. In the second case, policies are exchanged so that each device can use the information for detecting if there is a policy conflict.

When performing result-exchange conflict detection, most of the synchronisation process is similar to the synchronisation process in 6.2. A device checks the CSL tuple from the other device to determine if there is any tuple it needs to acquire. Once a new tuple is acquired, the device creates a synchronisation unit containing information needed for the policy engines on both sides to make a decision. This is similar to the synchronisation unit sent from the Synchronisation Manager to the Policy Engine during the synchronisation process in 6.2. A copy of the synchronisation unit is then sent to a policy engine on the two devices. On the remote device, the synchronisation unit is used the same way a locally created synchronisation unit will be used. The only difference is that the references to context information in the synchronisation unit will point to the information where the unit is created instead. This is to provide the

remote policy engine with the same environment the local policy engine has, which will allow an easy conflict detection between the results from the two engines.

Policy engines from both devices then process the synchronisation units and produce independent decisions. The decision is sent back from the remote device to the device generating the synchronisation unit. The two decisions are compared to detect any synchronisation conflict between the policies on the two nodes. The decisions can be compared since a decision created by the Policy Engine is composed of an action that is going to be taken and a tuple object that is going to be the target for that action. The simplest process is just to compare the actions and the tuple sequences. If they are not exactly matched, there is an indication that policies governing the synchronisation event on the two devices are in conflict.

The policy-exchange based conflict detection can be divided into sub cases depending on how the policy is used in conflict detection. Once a tree link is created between two devices, one of the devices requests the set of policies used in the synchronisation process from the other device. Once the policies are obtained, a conflict detection and a data synchronisation process can begin.

The simplest approach is then to detect a conflict between results obtained by processing a synchronisation unit using the two different sets of policies. After the device obtains policies from the remote device, it does a synchronisation process using both its policy and the remote policies. The results obtained are compared to detect any conflict.

Another way to detect a conflict between policies is to compare the policies themselves. There are two different types of comparison process - policy string comparison and policy semantic comparison. The first method is a basic string comparison process between the policies. Policies are compared with each other word by word. The latter method compares the policies' semantics. The two methods can be divided further into two more sub methods as shown in figure 6.5.

There are different ways for choosing between policies to be compared. The most basic way is to compare every policy that is defined in the engine. In this case, policies that would not be used in a particular synchronisation may be compared. On the other hand, another method is to compare only policies that are going to be used in the immediate synchronisation event.

The conflict detection methods shown here have their advantages and disadvantages.

- Comparing results is less strict than comparing policies. Different policies may return the same result even if the intention of the policies is different. For example, a policy that gives precedence to a tuple that is written afterwards may not be in conflict with a policy giving precedence to a tuple from a space that has a higher class in certain situations. However, they would be in conflict with each other if the comparison process was policy-based and either policy strings or policy intentions were compared.

- In policy-exchange based conflict detection, comparison between policy strings is simpler and easier to implement. Semantic comparison requires the comparing object to understand the policy's

meaning which is a lot harder to implement. However, comparing between strings directly can be seen as a more error prone or more strict process. Either way, the process will report a lot more results it considers as conflicts than a semantic comparison process since it will detect differences that should not be counted as a policy conflict such as a naming difference or differences in ordering of logical operands.

- Comparing between the entire sets of policies ensures that there is no chance a conflict will ever happen between the two devices, now or in the future. Comparing only the policies that are used in the immediate synchronisation event is faster.

Figure 6.6 shows the level of strictness for each type of conflict detection. There is not much difference between result-exchange and policy-exchange result based comparison. The size of a policy object transferred may be a little larger. In a case where disconnections occur quite often, it may be better to use a policy exchange since a disconnections will not terminate a synchronisation event which is being processed on one device since there is no need to wait for a result from the other device.

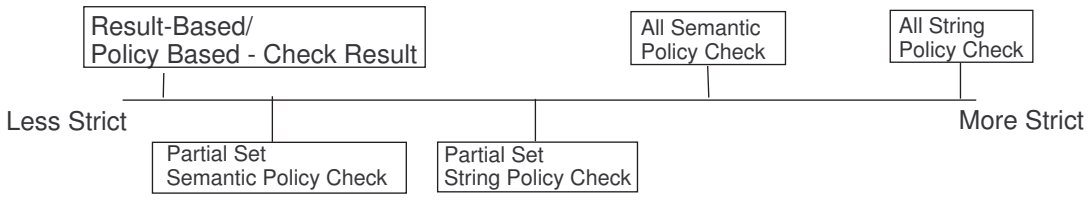


Figure 6.6: Conflict Detection Strictness Level

6.3.2.2 Conflict Resolution Process

After a conflict has been detected by a conflict detection process, the policy engine processing the current synchronisation has to find a way to resolve it. The Policy Engine in the JSFM chooses to allow a user to employ a higher level policy that can make a decision about which set of policies is going to be used during the synchronisation process. In order for the process to operate properly, the JSFM makes a number of assumptions based on an environment it is to be used in.

- Since the JSFM is intended for use by groups where users try to cooperate with each other by sharing their information, the JSFM assumes that at a higher level there is a set of policies that everyone will agree upon.The policies are employed on every device to resolve a conflict occurring from different lower level user-created policies. This assumption reflects an enterprise-wide working environment where each employee has control over his or her own device but still needs to follow higher level policies set up to prevent conflict in the group.

- Since this project does not stress security in using policy, the JSFM assumes that there is no user who intends to break the system by manipulating his/her device's policy and context information. For example, a system might give the highest precedence to a tuple from a specific server; if a device masquerades as the server by manipulating the device's ID in its context information, it can propagate a possibly malicious tuple with the high priority of the server.

With the assumptions above, the JSFM uses a higher level policy to resolve any lower level policy conflict. When a conflict is detected by the conflict detection process, a device doing synchronisation initiates a conflict resolving process which can be seen as supplying the Policy Engine with a set of higher level policies to obtain a result which is not an action on a tuple but a selection of which set of policies to use to determine the action.

If each user could define a higher level policy for his or her device, there would always be a chance that the different higher level policies from different devices would still be in conflict, which means different devices would choose different sets of policy to use for the synchronisation process. However, under the assumption that the same enterprise-level policies exist in every device, policy conflict will certainly be resolved at the level where all devices agree upon one set of enterprise policies.

At a level where the higher level policies are not in conflict, their results will tell the policy engine which lower level policy is to be used. This process is recursively repeated until it reaches the lowest level policies and the policy engine obtains an action on a tuple.

The structure of the higher level policies is similar to those of the first level policy. The only difference is in the action the higher level policy will take. Conflict detection in the higher level policy is also similar to the process for the first level policy. Thus, the processes that are used in the first level policy can also be applied at the higher level.

6.3.3 Data Convergence

The data convergence problem is another issue arising from using multiple sets of policies. Since each device has a different set of policies, they can take different decisions concerning the same tuple. The situation leads to different spaces having different sets of tuples and so data convergence does not happen. An important issue regarding the data convergence problem is the bouncing effect.

6.3.3.1 Bouncing Effect

The bouncing effect happens when a number of devices with weaker policies and context information are situated between devices with stronger policies, as shown in figure 6.3. The information hold by the devices in the figure is shown in table 6.1 below.

Consider a situation where the three devices (B, A, and C) are trying to synchronise; they have a tuple P which has the same identity on the three spaces but contains different information. This tuple

	B	A	C
Life Time	Oldest	Younger	Youngest
Device's Class	Silver	Bronze	Gold
1st Level Policy	Oldest First	Oldest First	Youngest First
2nd Level Policy	Highest Class First	Highest Class First	Highest Class First

Table 6.1: Bouncing Effect Scenario Information

will cause an identity clash during the synchronisation process, which will be resolved by the process explained in 6.3.2. However, even though an individual conflict between intermediate devices can be resolved, a bouncing effect still occurs, since devices that are not directly connected cannot resolve their differences. The most basic scenario is one where no extra information (policy and context information) is transferred with the tuple is shown below.

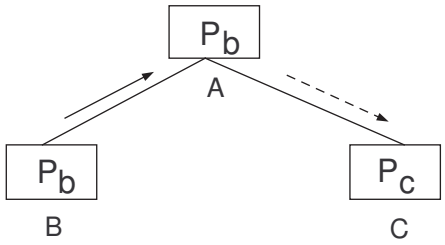


Figure 6.7: Bouncing Effect Scenario 1

The solid line shows the direction of tuple propagation in a synchronisation process and the dash lines shows the direction in which a synchronisation notification message is sent. In figure 6.7, a synchronisation process happens between B and A. Both B and A use a first level policy that gives higher precedence to a tuple that was written earlier. Therefore, the tuple P in A is replaced by the tuple P from B. At the end of this synchronisation, a notification message is sent from A to C.

After C receives the notification sent from A, it initiates a synchronisation process with A concerning the tuple P which A has received from B. Since no policy and context information is transferred over from B with the tuple, A uses its local information in the synchronisation process with C. Assuming a policy-exchange result based conflict detection is used, a policy conflict is detected because results from using the policies from A and C in the synchronisation are different. The policy from A gives precedence to its space since its policy gives precedence to an older tuple and context information in A indicates that it has an older tuple. However, the result from the policy in C contradicts the result from A because the C's policy gives precedence to a tuple that is younger and C has a younger tuple.

After the conflict is detected the second level policy is used. The two devices have the same second level policy. The policy allows the lower level policy of the space that has a higher class to be used, which is the policy from C. Therefore, the synchronisation result is to replace the tuple P in A which it has received from B with the tuple P from C. As a result of this synchronisation process, a synchronisation notification is sent from A to B as shown in figure 6.8

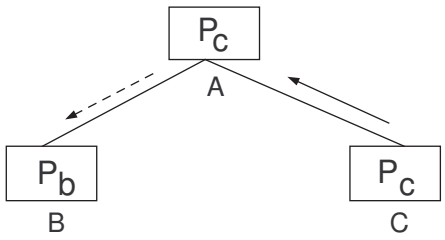


Figure 6.8: Bouncing Effect Scenario 2

The notification message sent from A starts a synchronisation process between B and A. The result in this synchronisation will be similar to the result shown in figure 6.7. Thus, it seems that a synchronisation process bounces between B and C and the tuple in A is always changing as a result of the synchronisation.

6.3.3.2 Bouncing Effect Solution Example

The problem occurs because information that represents the tuple priority is not moved with the tuple from the tuple origin. When the tuple is moved to another device because of a synchronisation process, this happens as a result of information from the local device which may have a different level of priority. To solve the problem, there is a need to find a way to identify and move the synchronisation control information with the tuple.

Synchronisation control information is information that represents a tuple's priority during the synchronisation process. Each synchronisation where there is an identity clash to be resolved can be seen as a comparison between the tuple's synchronisation control information. A tuple that has a higher priority in its control information is going to overwrite a tuple with lower priority. The information that is commonly used as a tuple's synchronisation control information is context information and policies from a space that writes the tuple. Only context information is not enough, since a different policy on a different space can interpret or use the context information differently.

In the scenario shown above, if context and policy information can be transferred with a tuple, the bouncing effect will not happen. After the first synchronisation between B and A, information in each device concerning the tuple P will be as shown in table 6.2. The synchronisation control information from B is transferred to A together with the tuple. In figure 6.8, even if the synchronisation control information from B is transferred to A during the first synchronisation, the synchronisation between A and C still has the same result. There will be a conflict in the first level policy and the second level policy will give a precedence to the policy from C because of its space class (Gold) is higher than the context information transferred from B (Silver). Therefore, the tuple P from C will overwrite the tuple P in A and the synchronisation control information of the tuple P in C will be copied to A as shown in table 6.3.

The difference is in the next synchronisation. Since context information and policies are transferred

	B	A	C
Life Time	Oldest	Oldest	Youngest
Device's Class	Silver	Silver	Gold
1st Level Policy	Oldest First	Oldest First	Youngest First
2nd Level Policy	Highest Class First	Highest Class First	Highest Class First

Table 6.2: Scenario Information for the tuple P after B-A synchronisation

	B	A	C
Life Time	Oldest	Youngest	Youngest
Device's Class	Silver	Gold	Gold
1st Level Policy	Oldest First	Youngest First	Youngest First
2nd Level Policy	Highest Class First	Highest Class First	Highest Class First

Table 6.3: Scenario Information for the tuple P after A-C synchronisation

with each tuple, the information from C is transferred with the tuple P. During the synchronisation between A and B regarding the tuple P from C, there is a conflict between the first level policies but the second level policy will give precedence to a policy for P in A which is transferred from C. At the end of the synchronisation process, every device has the tuple P originated from C and no bouncing effect happens.

More complex situations have been tested by adding more devices into the set up in two different ways as shown in figure 6.9. The moving of policy and context information still prevents any bouncing effect. Transferring only the context information, as in 6.2.2, is not enough to prevent the effect.

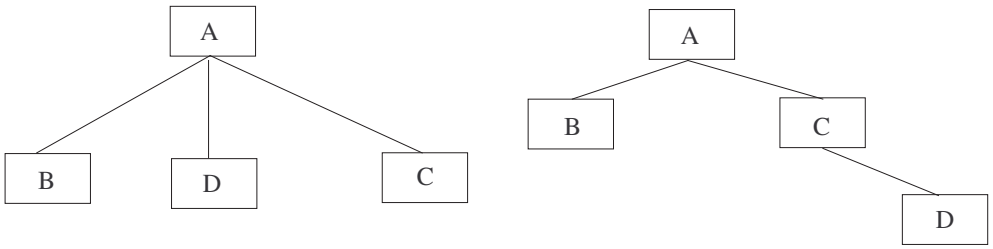


Figure 6.9: More Complex Bouncing Effect Scenario

After the synchronisation control information is identified, the next issue is how the information can be transferred or if there is any other way to prevent the bouncing effect without transferring the information. The issue will be discussed in 6.3.4

6.3.4 Information Transfer

One of the solutions to the bouncing effect that caused the data convergence problem in 6.3.3.1 is to find a way to move with a tuple its synchronisation control information. This section discusses further the issues concerning the transfer of the information to solve the data convergence problem and any other

methods that can be used instead of transferring the information.

The first sub-section lists how synchronisation control information can be transferred and discusses possible problems. The second sub-section illustrates possible methods that can be used in place of the direct synchronisation control information transfer discussed in the first section. Neither of the methods from the two sections are perfect solutions. Each has its advantages and disadvantages over other methods.

6.3.4.1 Direct Synchronisation Control Information Transfer

The basic synchronisation control information transfer process is based on simply attaching to each tuple its policies and context information before it is transferred to another device during a synchronisation and propagating the updated information when it is changed. The process is an extension of context synchronisation in the uniform policy environment discussed in 6.2.2. The difference is in the extra policy information that has to be transferred with the context information. Since the policy and the middleware policy enforcement is only concerned with the synchronisation process, the attached information will not affect any access control for the tuple. Consequently, it will not affect any control the remote space has over its tuples.

During the synchronisation process, after obtaining a tuple for a synchronisation event, the Policy Engine checks whether the tuple was written by the local device or any remote device. If the writer of the tuple is the local device, it uses the policies stored in the local device in the synchronisation process. If the tuple is written by any remote device, the policies and other information used in the process are obtained from the synchronisation control information attached to the tuple.

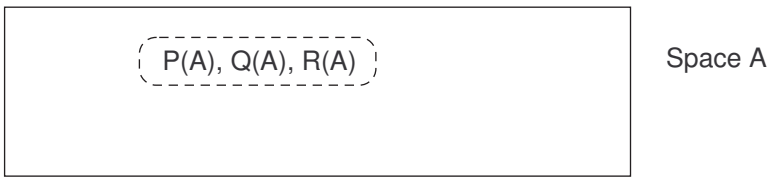


Figure 6.10: Synchronisation Control Information Transfer 1

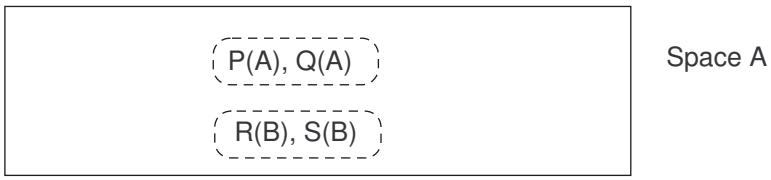


Figure 6.11: Synchronisation Control Information Transfer 2

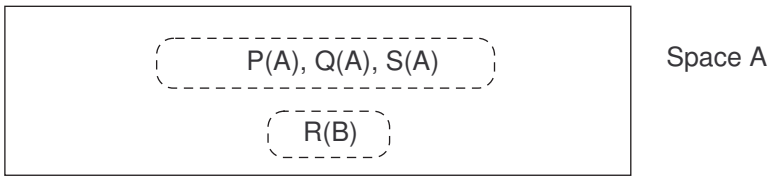


Figure 6.12: Synchronisation Control Information Transfer 3

Figures 6.10, 6.11, and 6.12 illustrate synchronisation control information that is attached to a tuple. "P(A)" refers to a tuple with an identity P controlled by policies and context information from space A. In figure 6.10, space A contains tuples that have been written by it. They are controlled by synchronisation control information from space A. Assume that the tuples R and S are updated in space B which has higher priority, after the synchronisation process with space B, the tuple R and S are propagated to space A with their synchronisation control information as shown in figure 6.11. After the synchronisation, if A tries to synchronise with other spaces, the tuple P and Q will be controlled by the control information from A but R and S will be controlled by the information A has received from B.

Even though A does not have control over the two tuples it receives from B, the situation is not permanent. Since the two tuples are in space A, A can delete or edit the tuples. If A changes information in the tuple S, the middleware will delete and create a new tuple S with information from A. Any synchronisation control for the tuple S will be provided by A instead of from B, as shown in figure 6.12

Even though transferring and dividing the control domain of a space can help prevent a bouncing effect, there are a number of disadvantages that need to be explained.

- Transferring the policy information and dividing the control domain of one space may generate unexpected behaviour from a user's point of view. This is because the link between synchronisation control information and a tuple is not visible at an application level and a user will not be able to differentiate between a tuple that is tied to local control information and a tuple that is tied to remote control information. Since synchronisation is sometimes controlled by synchronisation control information that is tied to a remote tuple, the remote tuple's behaviour during the synchronisation is going to be unexpected by the user.

- Allowing a remote device to gain indirect control of elements in local device is not good in terms of security. A malicious node can propagate dangerous information freely by setting up a policy with the highest priority. However, since a synchronisation process between devices is done using pull-mode, it is the receiving device that chooses whether to receive a tuple or not. It is harder for a malicious device to forcefully propagate a tuple to other devices.

6.3.4.2 Indirect Synchronisation Control Information Transfer

The direct synchronisation control information transfer introduced in 6.3.4.1 can prevent the bouncing effect but still has a number of disadvantages. This section is going to introduce other possible solutions that can prevent the effect without needing a direct transfer of the controlled information. Nevertheless, there is no perfect solution to the problem. Each solution may overcome the problems of direct information transfer but it has its own problems that will be discussed in this section.

- The first method is to give the responsibility for detecting the bouncing effect to the synchronisation event distribution level. This is the opposite of the method used in 6.3.4.1 , in that this method does not require any synchronisation control information transfer. The event distribution level is used because it already sends a notification message to each device. Adding more information into the message is simple and can be used to detect the loop that occurs when there is a bouncing effect.

  Information relating to the sender of the a notification message such as its name and its address is attached to the message. When the message reaches a remote device, it initiates a new synchronisation process. After the synchronisation process is finished the remote node creates a new notification message to be sent to another remote node in the event distribution tree. It attaches its address information together with the address information it has received from the previous notification message to the new message.

  The information attached to a notification message can be used to check if the bouncing effect is occurring. Before the synchronisation process initiated from a notification message begins, the device that receives the message checks the path that the message has taken during its propagation through the event distribution tree. When a device detects that there is a loop in the path information, it contacts the devices at the two ends of the loop to resolves the conflict between them. The two end devices are used instead of any intermediary in the loop because the conflict is actually happened because the two of them, any node between them is just a path for propagating the tuple and should not affect the outcome of the conflict.

  In figure 6.13, assuming that there is a conflict between policies in B and D that causes the bouncing effect, a synchronisation process starts between B and A and propagates along the event distribution tree. Each notification message along the propagation path adds a new address to the information in the message. In figure 6.14, a synchronisation initiated from B is bounced back from D. When C receives a notification message back from D, it can detects that there is a loop and the two ends of the loop are B and D. Since the message contains each device's address information, C can contact B and D to resolve the conflict.

  Even though this method does not require any control information transfer, it still has its disadvantages. The main disadvantage of this method is that the conflict resolution process in this method

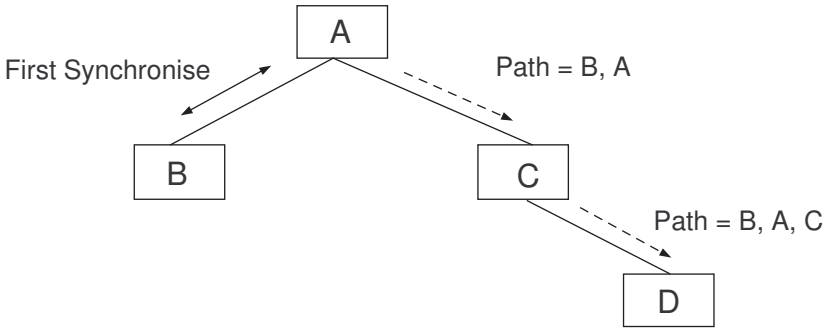


Figure 6.13: Using Notification Message to Solve Bouncing Effect 1

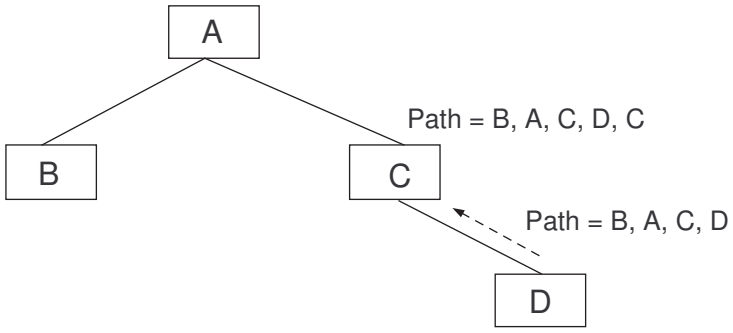


Figure 6.14: Using Notification Message to Solve Bouncing Effect 2

requires the two end of the loop to be present when the loop is detected. If one of them has already disconnected, the conflict cannot be resolved correctly. On the other hand, since the direct synchronisation control information process transfers information required for conflict resolution from one device to another, there is no needs for the device that originates the conflict to remain in the event distribution tree, which is probably more suitable for a wireless ad-hoc environment.

- Another method for preventing the bouncing effect while not directly transferring control information is to use a synchronisation counter. The idea of this method is to allow less synchronisation precision but to prevent issues occurring from direct information transfer as discussed in 6.3.4.1. The synchronisation counter can be implemented in the middleware level tuple that encapsulates an application level tuple. Each time a tuple wins an identity clash, the counter is increased by one. Each time a synchronisation path is going to be reversed (detected by checking the immediate preceding path of a synchronisation notification message), the counter in the two tuples being in conflict are compared. The tuple that has higher synchronisation count wins the conflict.

  The synchronisation counter prevents any bouncing effect but does not need synchronisation control information to be transferred or connections to devices at the two ends of the bouncing loop. However, since it simplifies the information it uses to prevent the effect, it cannot guarantee that every synchronisation happens as it should have, using synchronisation control information. There

will be cases where a tuple from a device with lower priority wins over a tuple from a device that has higher priority, if the lower priority tuple is propagated through the event distribution tree using a path with a higher number of low priority devices while the tuple from a device with higher priority is propagated through a path with a lower number of devices.

Nevertheless, the counter itself is a good representation of the tuple priority because the counter can be increased only if the tuple wins a conflict, which shows that the tuple is from a device that has higher priority. A tuple with a high count will be a tuple that comes from a path that has a device with high priority.

6.4 Conclusion

This chapter contains a discussion of the synchronisation process in the JSFM. The JSFM supports both uniform policy and multiple policies synchronisation processes. The uniform policy synchronisation process is simpler, synchronising tuples from different devices but using the same set of policies. In this case, the middleware can assume that the two devices agree on every synchronisation result and there is no need for the middleware to detect any conflict from the policies.

However, allowing users to define their own policies for their devices gives more flexibility to an application operating on the JSFM. Users using the same application may have different ideas about how their information should be synchronised. A good example is an application in an organisation structured into sub-divisions where each of them defines different synchronisation policies.

The JSFM suports multiple policy synchronisation by dividing the process into two main parts - policy conflict detection and policy conflict resolution. Even with the two processes, the JSFM still needs to rely on an assumption that at a some level there exists a uniform policy for any two devices. The synchronisation process fails if the two devices cannot agree on a single action for a particular tuple.

Furthermore, using the multiple policy synchronisation process costs more than the simple synchronisation process. The process needs the remote device to be involved in conflict detection and requires time for the higher level policy to resolve policy conflict in the lower level. These extra costs need to be considered when a decision on whether the JSFM is suitable for a particular application is made.

The cost of the two synchronisation processes is shown in chapter 7.

Chapter 7

Performance

7.1 Introduction

The previous chapters have explained the architecture of the middleware and its policy engine. The next step is to show the performance of the system by putting the system through a number of tests. This chapter contains tests for several parts of the system.

Apart from the test results, each section in the chapter also contains a discussion concerning the test, including an explanation of the nature of the results. The information in this chapter helps in determining if an application can function properly with the middleware.

The chapter is divided into five sections. The first section contains tests relating to the access time for each space interaction. The second section determines the average tree building time between two devices. The third section contains tests of the synchronisation process between devices with a similar set of policies, while the fourth section contains the same kind of test but between devices with different sets of policies. The last section makes an estimation of a tuple propagation time in a number of situations. Calculated estimates and simulation are used because it is not practical to do these tests in a real environment.

7.2 JSFM and JavaSpace Access Test

This section contains basic information from access time measurements of the JSFM and of a normal JavaSpace. The access time is the amount of time for a space to act upon one access command from a user. The test shows results for the three most commonly used space access commands - read, write, and take.

The reason for the test is to show that even though using the JSFM increases the access time for each space operation, it is still possible to use the JSFM in simple non real-time data sharing applications. The

first sub-section explains how the test was done and the second sub-section shows the results of the test and gives an explanation of the results.

7.2.1 Access Test Set Up

All of the tests are done on a simple 2.5 GHz desktop machine with 512 MB of memory. The space service used is the Jini JavaSpace service version 2.0.001. The JavaSpace used is the persistence version which stores its tuples on the machine's harddisk instead of using the machine's memory. This is to allow tuple data to persist after the space is terminated.

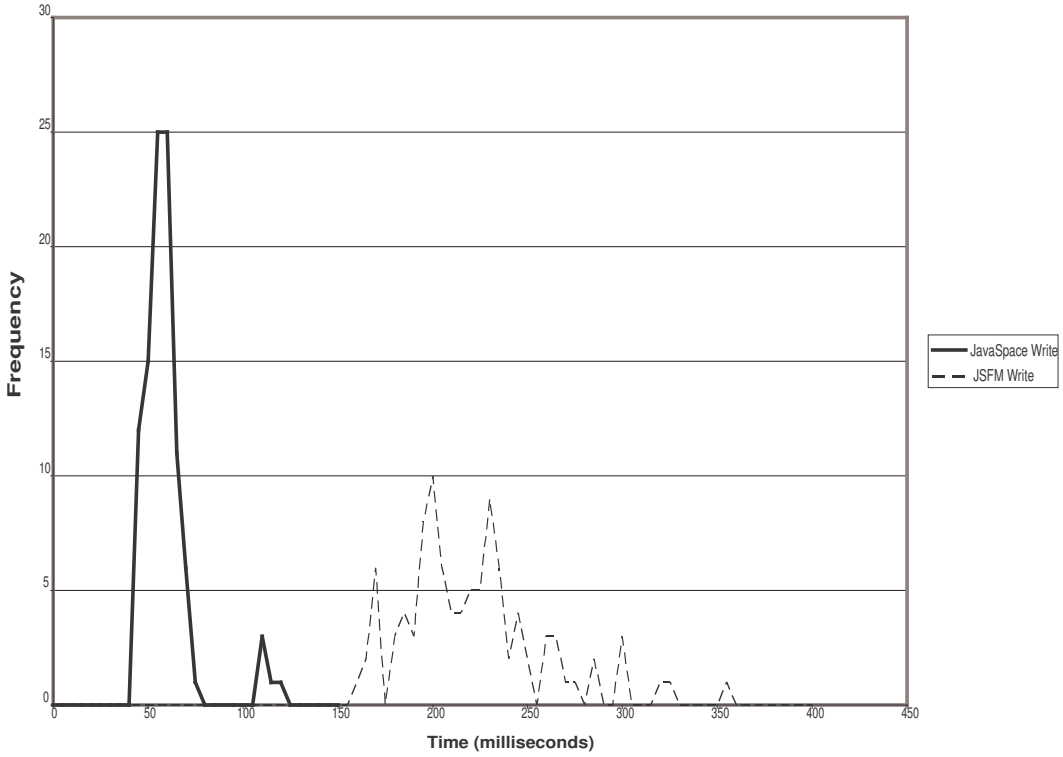


Figure 7.1: Write Test Result

The JavaSpace access test was done by building a simple program that repeatedly uses a specific space command on a local space with a wait interval between each attempt. The test program first writes one hundred tuples into the local space. Then, it reads a random tuple one hundred times. Lastly, it takes each tuple out of the space.

Each access is timed using a clock package provided by Prof. Peter Linington [Lin06b]. This package uses the processor clock to obtain a high resolution time in microseconds. The test system calls the clock interface to obtain the time just before each space access and just after the access. The times are kept in an array and are saved after the test is finished.

The same kind of test was done using the JSFM. Another simple program was created to access a space through the JSFM. This program is similar to the program used for testing the JavaSpace. It uses the same test process with the same type of tuple and the same clock package. The only difference is that the second test program accesses the JSFM instead of using the JavaSpace directly.

7.2.2 Access Test Result

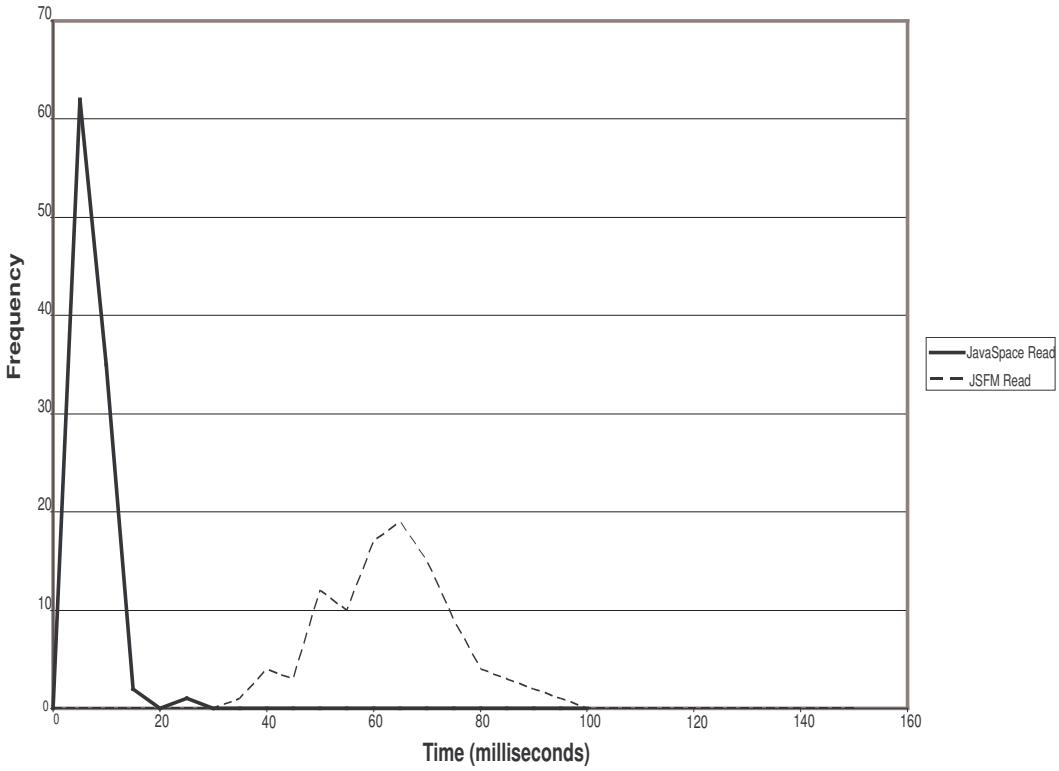


Figure 7.2: Read Test Result

Figure 7.1, 7.2, and 7.3 show the results of the testing process. The JSFM takes more time in order to process the three space commands because:

- The JSFM has a larger tuple size to accommodate extra information which it needs to do the synchronisation process. The JSFM encapsulates an application tuple into an InMessage tuple containing the required information. Moreover, during each access, the JSFM has to write or edit basic context information regarding the tuple that is accessed. This increases the amount of information being transferred during each command.

- During a single JSFM space command, there are a number of interactions between the JSFM and the local space. For example, a take command from the JSFM is done by taking the actual target

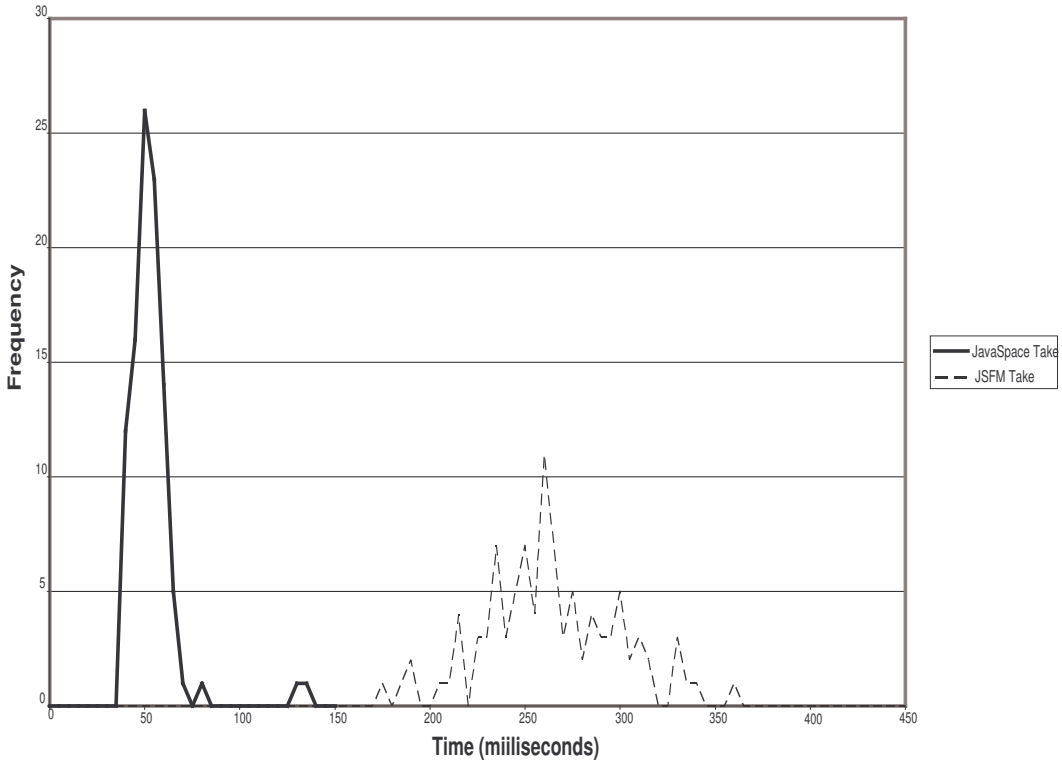


Figure 7.3: Take Test Result

tuple, writing a command tuple, and writing a tombstone tuple. The two latter tuples are written to allow the JSFM to do any synchronisation process properly.

JSFM Access Command	Number of Write	Number of Read	Number of Take
Write	3	0	1
Read	1	1	0
Take	3	0	2

Table 7.1: Number of JavaSpace Commands Used In a JSFM Access Command

Table 7.1 shows the number of JavaSpace commands used in each JSFM access command. The extra JavaSpace commands are used to alter information needed for the JSFM synchronisation process. For example, in the JSFM "write" command, three JavaSpace "write" commands are used to write a tuple, write its context information, and write an updated CSL tuple. The JavaSpace "take" command is used to remove the old CSL tuple. The JSFM "read" command needs one JavaSpace "write" command to write an extra context information showing that the tuple has been accessed and the JSFM "take" command uses three JavaSpace "write" commands to write a command tuple, a tombstone tuple, and an updated CSL tuple. Two JavaSpace "take" commands are used in the JSFM "take" command to take the targeted tuple and the local CSL tuple. Table 7.2 shows an average access time using the JSFM compared to an

Access Command	JavaSpace Access Time(ms)	STDEV	JSFM Access Time(ms)	STDEV	Calculated Access Time(ms)
Read	5.0	2.4	60.6	11.8	61.9
Write	56.9	14.2	218.6	37.1	222.1
Take	51.4	13.6	267.6	58.9	273.5

Table 7.2: Average JSFM Access Time

average JavaSpace access time and a predicted result taken from using the JavaSpace access time and the number of accesses given in table 7.1.

The average CPU usage can be used to show that the process depends largely on disk access time and does not saturate the CPU time. By removing the wait period between each access test, an average CPU usage can be determined. Table 7.3 shows an average CPU usage during an access test without any wait period. This means accesses are done consecutively one after the other. This test, therefore, shows the maximum amount of CPU usage by the JSFM access commands.

JSFM Access Command	CPU Usage (Percentage)
Write	45-50
Read	65-75
Take	35-45

Table 7.3: CPU Usage for each JSFM access command

Even though it takes more time to interact with a space through the JSFM than accessing the space directly, the time taken by each command is small enough for a simple application to operate properly. The difference between the access times are in the millisecond range. Depending on the characteristics of an application, this difference may not affect a non time-critical application with limited data access significantly.

7.3 Tree Building Test

The tree building process builds a tree link between devices when they connect to each other. The process is initiated as soon as a new device moves into the area where there is a group of connected devices.

The reason for doing this test is to determine the minimum amount of time a device has to stay in the same area before a synchronisation process can be started. This is because a network connection with another device has to be established before a device can synchronise. Then, the Jini service registry situated on each device, detecting references to the other Jini services, notifies the JSFM to start building the tree link. After the process is finished, the synchronisation process can begin.

7.3.1 Tree Building Test Setup

The test process measures the time between a core node receiving a notification from its registry service that a new device is in the area and the link being established so that the event distribution layer signal the policy engine to start a synchronisation process. More detail of the tree building process is in 4.3.3.

To make the test simpler, one device is selected to be responsible as a core node and the other device acts as the normal node. The JSFM event distribution layer on the normal node is edited so that it will never change into a core node; the measurement is done on the core node. The test uses the same clock package as in the previous test. The machine used as the core node is the same machine described in 7.2.1. The normal node machine is a 2GHz laptop with 1 GB of memory connecting to the core machine using the IEEE 802.11a wireless network via a base station.

7.3.2 Tree Building Result

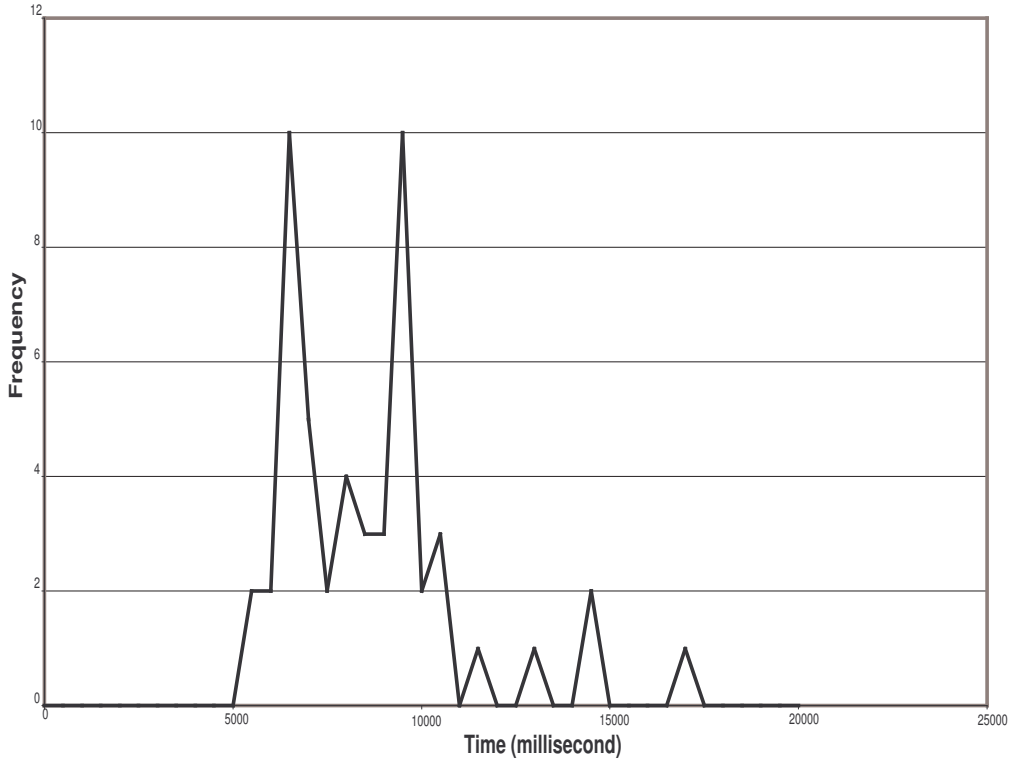


Figure 7.4: Tree Building Test Result

Figure 7.4 shows the result of the tree building test. The tree building process depends heavily on the time between the keep-alive messages periodically sent from the core node. This makes the result fluctuate. For example, to establish a proper tree link after a network connection is established, the normal node has to wait for the next keep-alive message propagated from the core node so that it can

determine where in the event distribution tree it should create a tree link.

The average time for a tree building process is eight seconds (8.38 seconds). However, this depends on the interval between keep-alive transmissions. By varying the amount of time between keep-alive transmissions from the core node, the amount of time needed to build the tree link can be changed.

The test was done for four different sending periods - 2500, 5000, 7500, and 10000 milliseconds. The average result of each period is shown in table 7.4 and plotted in figure 7.5.

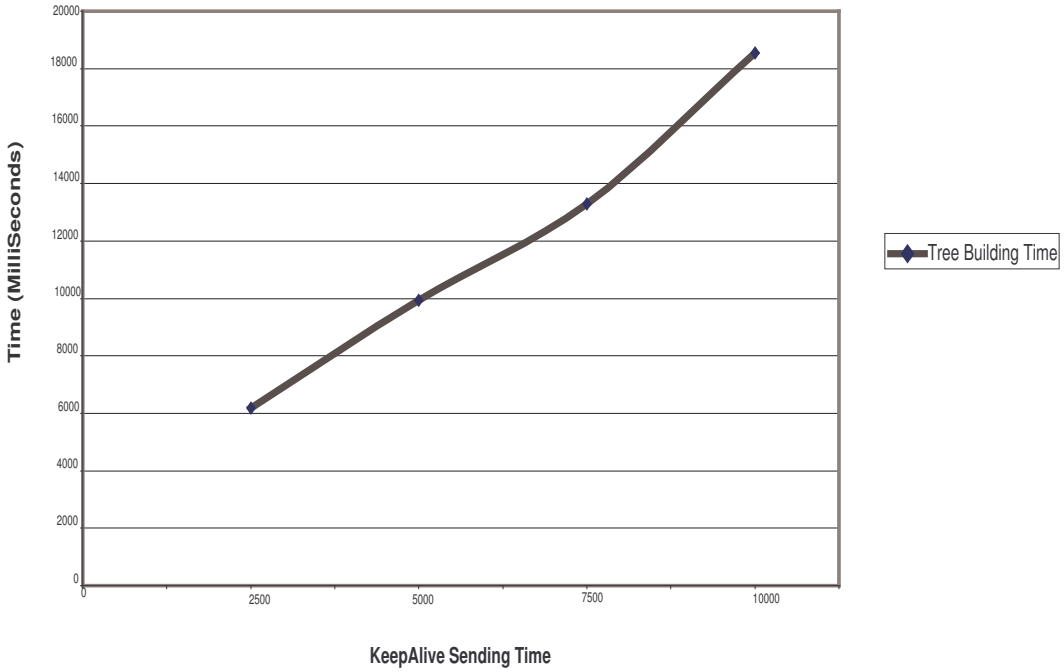


Figure 7.5: Keep-Alive and Tree Building Time

Reducing the time between keep-alive transmissions can reduce the time needed for the tree building process. However, reducing the time also has its disadvantages. For example, reducing the time means more keep-alive messages are sent during any particular period of time. This situation can lead to flooding the network with these messages which defeats the propose of creating the distribution layer.

Moreover, more keep-alive sending means there are more interruptions to the JSFM, which can cause other processes to slow down.

The tree building time can be reduced to six seconds using two and a half seconds between each keep-alive transmission. Lower tree building times can be achieved by reducing the keep-alive period further but the effect seems to become less. In figure 7.5, the line intercepts the Y axis at around 3 seconds. Therefore, the least amount of time the middleware needs to establish a tree link is about three seconds.

Keep-Alive Sending (MilliSeconds)	Tree Building Time (Seconds)
10000	17.7
7500	12.5
5000	9.0
2500	6.1

Table 7.4: Keep-Alive Sending and Tree Building Time

7.4 Simple Synchronisation Time

This section contains the result of a simple synchronisation time test. The reason for this test is to estimate the minimum time needed for a synchronisation process to complete for a single tuple. With the information from this section and the information from 7.3, an estimate of the time the whole synchronisation process takes can be calculated.

7.4.1 Simple Synchronisation Test Setup

To get the lowest synchronisation time for a tuple, the simplest synchronisation process is required. Therefore, this test uses a synchronisation where an extra tuple exists on one device but does not exist on the other device. The policy used in this test is to transfer an extra tuple from one space to the other space with no constraint.

The test is done by using a simple program that keeps writing a new tuple into the space. However, after each write, the program sleeps for three seconds to make sure that the other device finishes its synchronisation. This is to prevent tuple updates from interfering with one another during the synchronisation process.

The time measurement starts from when the policy engine receives a request from a synchronisation object for a decision about a tuple. The measurement ends after an action from the policy engine is processed by the synchronisation object and the basic context information for the tuple has been transferred.

7.4.2 Simple Synchronisation Test Result

Figure 7.6 shows the time the Policy Engine takes for a simple synchronisation process. The average time is 250.7 milliseconds or around 0.25 seconds. However, this is the least amount of time for a synchronisation process for a single tuple. As a comparison, figure 7.6 also shows the result for a more complex synchronisation process where there is identity clash, which take approximately 0.4 seconds (391.7 milliseconds).

This second result is obtained by simply writing a tuple with the same identity to a space. During the synchronisation, the policy engine finds out that there is a conflict and uses the appropriate policy to solve the problem. In this case, the policy constraint uses context information about the time when the tuples were written and gives precedence to a tuple that was written later. The conflict situation takes

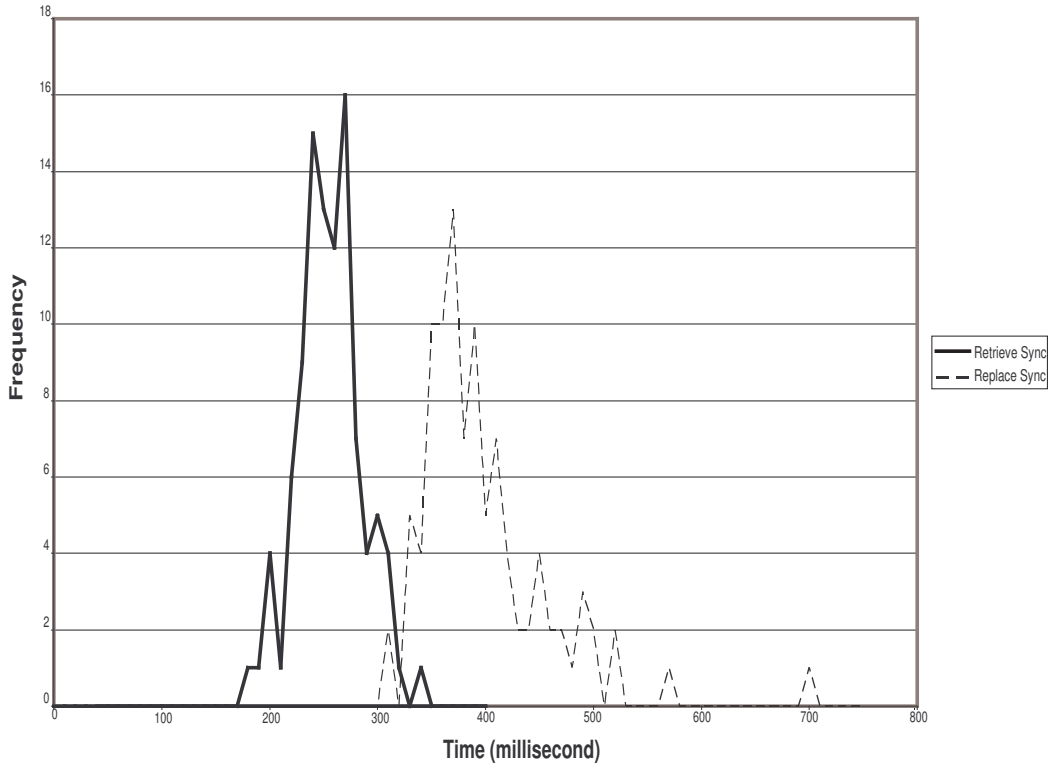


Figure 7.6: Simple Synchronisation Time

more time because the policy engine has to access the context information used in the constraint which involves more data access from both the local and the remote space.

From the information obtained from these tests, an estimate of the minimum time that a device has to stay connected can be obtained. For example, on average, if a device normally has a back log of 50 extra tuples each time it builds a connection, it will take approximately 20 seconds(8 + 50 x 0.25) to finish its synchronisation process. This information can be used to determine if the JSFM is suitable for a particular application or not.

For example, without any performance tuning such as keep-alive time adjustment, the configuration would probably not be suitable for an ad-hoc mobile network between passing cars but it should be feasible for the meeting sharing application where users are walking about in an office building.

7.5 Multiple Policies Synchronisation Test

The multiple policies synchronisation test is to show that it is possible for the middleware to support a situation where different devices have different sets of policies. Furthermore, even though supporting the feature increases the amount of time the process takes, it is still possible to use it in some applications.

7.5.1 Multiple Policies Synchronisation Test Setup

The setup of this test requires the synchronisation process of the middleware to be changed. The changes are in three parts - policy conflict detection, policy conflict resolving, and context and policy transfer. The conflict detection process is done using a result comparison. The Policy Engine gathers information regarding to a synchronisation for a tuple and sends it to an engine on another device participating in the process. A policy conflict can be detected from the results obtained from the local and remote engines.

Conflict resolving is done by using a decision from a higher level policy. Results that lead to a policy conflict are stored in stacks and will be chosen by the higher level policies. For example, if the higher level policies choose to give precedence to local policies, the result from the local stack will be used. The process is done recursively until it reaches the first level policy. However, in this test, the setup is done so that the conflict from the first level policies can always be resolved by using the second level policy.

The last change is in a context transfer. Instead of only transferring context information regarding the tuple, the policies from the tuple origin are also attached to the tuple. The policy files are kept in a middleware level tuple with a list indicating policy names for each policy level. This information is used for later synchronisation between more distant nodes in the tree. More information regarding the multiple policies synchronisation process is in 6.3;

The actual synchronisation process and timing are done as in the single level synchronisation conflict test discussed in the test result section for simple conflict synchronisation. The timing starts when a device receives a notification for a new tuple from another device and ends when all the context and policies are completely transferred.

7.5.2 Multiple Policies Synchronisation Test Result

Synchronisation Type	Average Synchronisation Time (MilliSeconds)	STDEV
Simple Sync	250.7	29.3
Conflict Sync with No Remote Check	391.7	58.5
Conflict Sync with Remote Check	887.9	110.9
Two Level Policy Conflict	1193.5	154.5

Table 7.5: Average Synchronisation Time Comparison

From the result shown in figure 7.7 and Table 7.5, merely adding a result check from a remote Policy Engine increases considerably the amount of time the synchronisation process takes. This is because with the remote checking process, the Policy Engine spends more time preparing for the synchronisation and then it has to wait for the result from the remote device for comparison.

However, adding a policy conflict does not increase the synchronisation time as much as adding the remote checking process. Each synchronisation conflict adds another synchronisation to the process in order to solve the lower level policy conflict. The result shows that the fixed cost for the process is larger

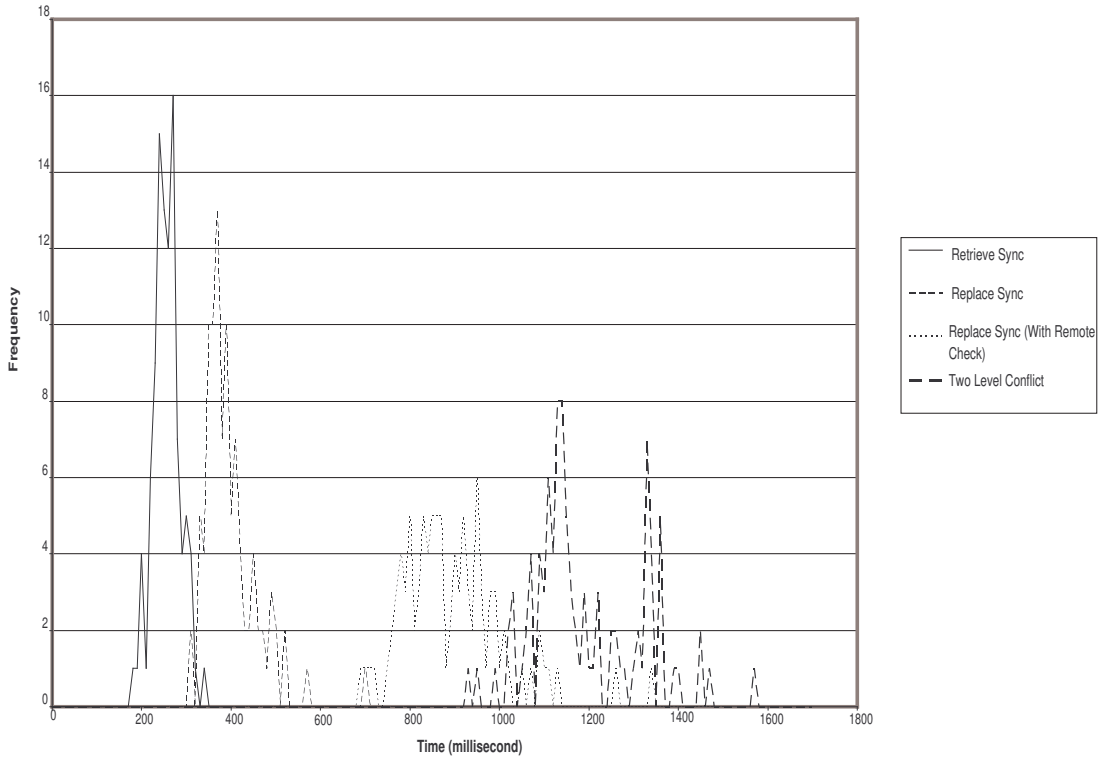


Figure 7.7: Two Levels Policy Conflict

than the cost for adding another synchronisation and it is possible to increase the level of the policy conflict further without too large an increase in the synchronisation time.

The reason why adding another synchronisation is not as costly as the first one may come from the fact that there are several objects that the Policy Engine can reuse from the first synchronisation. For example, some context information required for the second synchronisation can be reused.

It is also possible to reduce the time for the second synchronisation further by not sending information that has already been transferred during the first synchronisation. However, if this is done, the receiving device needs to store remote information in its Policy Engine, which prevents the Policy Engine from being stateless. Storing remote information makes the engine more complex and creates a possibility that information from different devices might mistakenly be used during the remote decision making process.

For comparison, the simple synchronisation in 7.4.2 needs approximately 20 seconds to synchronise fifty extra tuples. The engine needs 28 (8 + 50 x 0.4) seconds to synchronise conflicting tuples without enabling a remote decision making process. It takes 53 (8 + 50 x 0.9) seconds to synchronise if the remote checking process is enabled. In the case of a policy conflict, the engine takes 68 (8 + 50 x 1.2) seconds to complete the two levels policy conflict synchronisation process.

Therefore, it is possible for the JSFM to support multiple policies synchronisation even though the feature has its cost. On average, the cost of the process will double the middleware's synchronisation

time. It can still be used in a scenario where users stay in the same area for a certain amount of time.

7.6 Estimated Tuple Propagation Time Calculation

Another element that will affect performance of the middleware is the time it takes for a new tuple to be propagated from one side of an event distribution tree to the other side. There are several factors that can affect the tuple propagation time. For example, different types of synchronisation processes require different amounts of time. If the new tuple is an extra tuple with no identity clash, the process will take less time that a synchronisation process with conflicting tuples or a process where there is a policy conflict.

Moreover, properties of the distribution tree such as the maximum number of child nodes that a node can support alter the propagation time. Ideally, the more child nodes a single node can support the smaller the number of links the tuple has to be propagated over for the same number of nodes.

The tests in this section could not be set up easily. Therefore, a calculated approximation will be used to show the performance of the system. For this approximation, we assume there are one hundred nodes already in an environment and a new node enters the area, and that the distribution tree configuration for the devices is altered for each test case. The duration for each activity such as synchronisation times, are obtained from the tests in the previous sections. The value obtained from these approximations can give an idea of how long the update of a tuple takes to reach all the user in the environment, in typical and extreme cases.

7.6.1 General Propagation Scenario

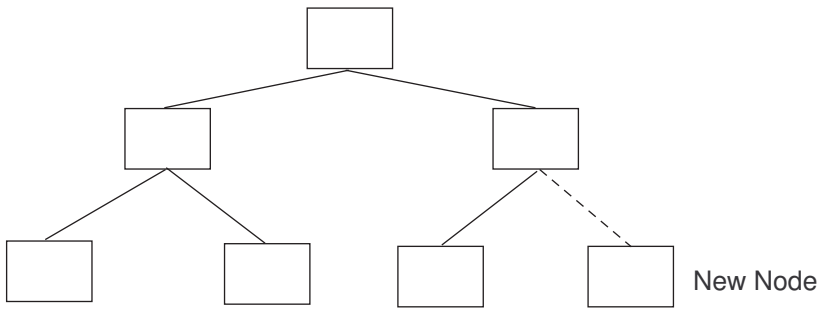


Figure 7.8: General Multicast Tree Setup

This section discusses the synchronisation time in a general situation assuming that a well-formed distribution tree exists and the new node connects as a new leaf node to the tree. To find the time, two main calculations are required. The first is to find how many synchronisation processes need to happen to carry a tuple from a new node to every other node. However, simply finding the number of branches

in a tree is not enough because after the new information reaches a node with more than one child node, a number of synchronisation processes are overlapped with each other.

The simplest way is probably to start by finding the number of levels in the tree. Every node in the tree will connect to the lowest level (as near to the root node as possible); this is managed by the middleware event distribution Layer. The level of the tree can be calculated by doing a simple calculation.

$$Number\ of\ nodes \leq \sum_{i=0}^{L-1} c^i \tag{7.1}$$

"c" is the maximum child size for a node and L-1 (Level - 1) is the smallest number that makes the right hand side of the equation higher than the number of nodes. The number of synchronisation steps excluding synchronisations that are overlapped can be estimated using the equation 7.2. Note that this number is not precise because it also depends on the direction in which the node chooses to send its first notification message.

$$Number\ of\ Sync = (Level * 2) \tag{7.2}$$

The next step is to estimate the time for each synchronisation process based on different types of synchronisation.

$$Average\ Sync\ Time = \sum_{i=1}^{n} synctime_i * prob_i \tag{7.3}$$

The average time is calculated from a synchronisation time for each synchronisation type multiplied by the probability that the type of synchronisation is going to happen. The total synchronisation time can be estimated by using the number of synchronisations and the average time.

$$Total\ Time = Number\ of\ Sync\ *\ Average\ Sync\ Time \tag{7.4}$$

For example, in the case above, there are one hundred node in the environment. Assume that each node's maximum child size is four, the tree will have four level (1 + 4 + 16 + 64 + 256). Assume that there are two types of synchronisations as shown in table 7.6

Synchronisation Type	**Synchronisation Time (MilliSeconds)**	**Probability**
Conflict Sync with Remote Check	887.9	0.8
Two Level Policy Conflict	1193.5	0.2

Table 7.6: Synchronisation Probability

Replacing each "synctime" and "prob" in equation 7.3 with the information in table 7.6, the average synchronisation time is 949.02 milliseconds. From equation 7.2, there are eight synchronisation processes happen before the tuple is propagated over the tree. Therefore, The total from equation 7.4 is

7592.16 milliseconds (8 seconds).

With the event distribution tree building period, it takes around twenty seconds for a device to start building a tree link and to distribute a new tuple to all the other nodes.

However, this is only an estimate of the synchronisation time. The real synchronisation time is probably different from the time shown here. For example, each synchronisations in different branches may not be completely overlapped with each other because of the difference between the times they receive notification messages. The synchronisation time may be reduced or increased if there is a conflict that rejects the change, bouncing the tuple back.

Nevertheless, the result from the formula should be a good representation of the synchronisation time which can be used in designing an application. Note that the node does not need to stay in the area until the propagation process is finished because of the direct synchronisation control information transfer discussed in 6.3.4.1

7.6.2 Worst Case Approximation Time

This second approximation gives an idea of the maximum propagation time to cover the one hundred node in the area in the worst case scenario. The purpose of this estimate is to show the lower bound of the system performance. Assume that the policy conflict level limit is two, and a new tuple causes an identity clash and a first level policy conflict in every node. To calculate the worst propagation time, the maximum child size for each node is set to one.

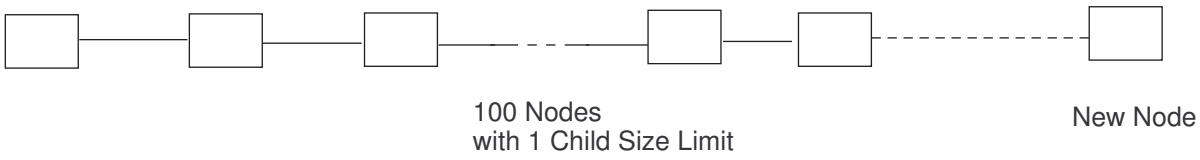


Figure 7.9: Worst Case Setup

In this case, propagation between pairs of nodes takes the longest synchronisation time. Moreover, the tuple is propagated one hundred times to reach the farthest node since there is no overlapping between each synchronisation because of the event distribution tree child size limit. The result will show the worst possible time for propagating a tuple between one hundred nodes.

The result in 7.3.2 showed that a tree building process takes approximately eight seconds. This process only happens once to connect the new device to the existing distribution tree. From the synchronisation time test result in 7.5.2, it takes around 1.2 second to finish a synchronisation process where there is a policy conflict. Therefore, an approximate time for the tuple to be propagated to every nodes

in the area is 8 + (1.2 x 100) = 128 seconds.

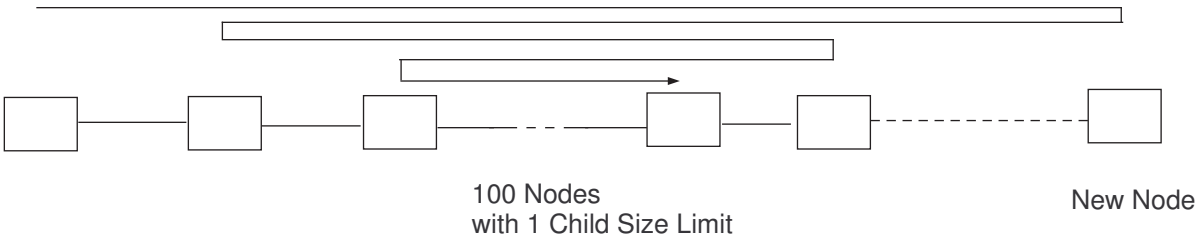


Figure 7.10: Bounce Worst Case Setup

Finally, if the policies in the nodes were set up so that they bounce a tuple back and forth as shown in figure 7.10, the time that is taken for the system to settle will be the worst tuple synchronisation time which is

$$8 + (1.2 * 100) + (1.2 * 99) + (1.2 * 98) + ...(1.2 * 1) = 5948\ seconds \tag{7.5}$$

This is the worst case scenario which rarely happens unless it is intentionally set up. However, the test shows that the performance distribution is long tailed and programmers that use the JSFM for their projects have to think about the possibility that these cases can occur.

The two scenarios give an idea of the amount of time the middleware takes to stabilise the new tuple in the event distribution tree and how long a node needs to stay connected to the tree. The estimation does not cover all the cases that are possible. For example, in the best case, there can be no nodes that can overrule the new tuple from the new node and thus no disconnection that prevents propagation of the tuple. The node then only needs to stay connected to the tree for the time needed for the first synchronisation step. This is because once the tuple is passed to a node in the tree, that node takes over the responsibility to pass the tuple to the other nodes in the tree. The presence of the original node is not required.

It is also harder to stabilise the tree where a node repeatedly disconnects and reconnects at different places in the tree. In this case, there is a chance that the overlap between synchronisation processes will not be as efficient as has been assumed earlier.

7.6.3 Tuple Propagation in an Unstructured Environment

The previous section estimated the system performance in an environment where a device is added into an existing tree. All the devices in the environment are connected together to form a single tree. In this case, the time for a tuple to reach every node depends heavily on the time to propagate a tuple between tree nodes.

However, there will be situations where devices cannot join the event tree because they do not stay in the same area for long enough. In this situation, devices' and the environment's properties such as speed or the area covered also affects the synchronisation time.

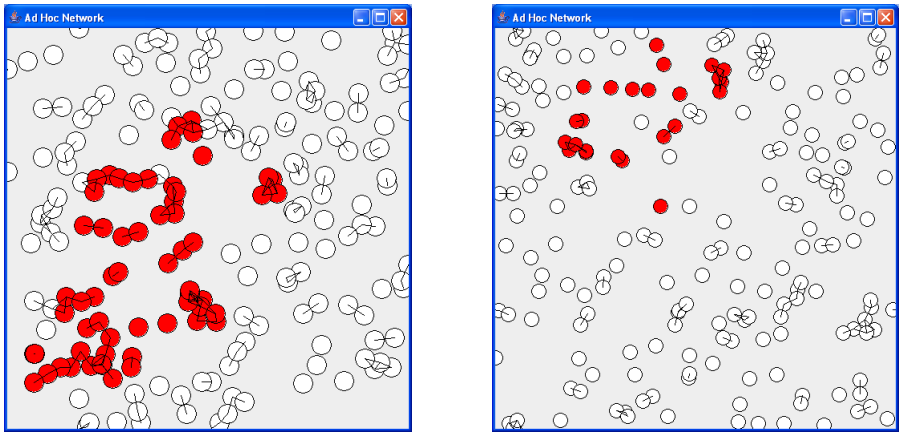


Figure 7.11: MobiSim Screen Capture

This section shows results from a set of tests that illustrate how the middleware will behave in a situation where devices are not completely connected but form many small trees. In this environment, the characteristics of devices in an area and the way they move affect the performance of the tuple propagation processes. The tests are done using a modified version of MobiSim - an ad-hoc mobile network simulator built by Prof. Peter Linington [Lin07]. The test scenario contains a number of devices in a toroidal area formed by linking each side of a square area to the opposite side. In each scenario, a number of nodes representing mobile devices move at random in this area. One node is selected as a device which creates an extra tuple that will be propagated to the other nodes. During each test, various information such as the nodes' average speed, the average connection and disconnection time, and the time needed to propagate a tuple to other nodes are all collected.

Figure 7.11 shows screens captured from the simulator. It shows the ability to modify parameters of devices and environment in the simulator. The circles show network coverage and each device in figure 7.11A has a wider range wireless coverage than the devices in figure 7.11B which leads to bigger trees and so allows more synchronisation time and leads to higher chance of successful synchronisation.

Figure 7.12 shows the time for synchronisation to reach a different numbers of nodes. The graph shows results from four tests on different sized areas and different number of nodes, specifically 100, 200, 300, and 400 nodes. Each node has a network diameter equal to fifty metres and moves at an average of approximately 0.1 metre per second which is to represent an environment where devices are moving at low speed. To be able to compare the results, each test area is made so that node density in

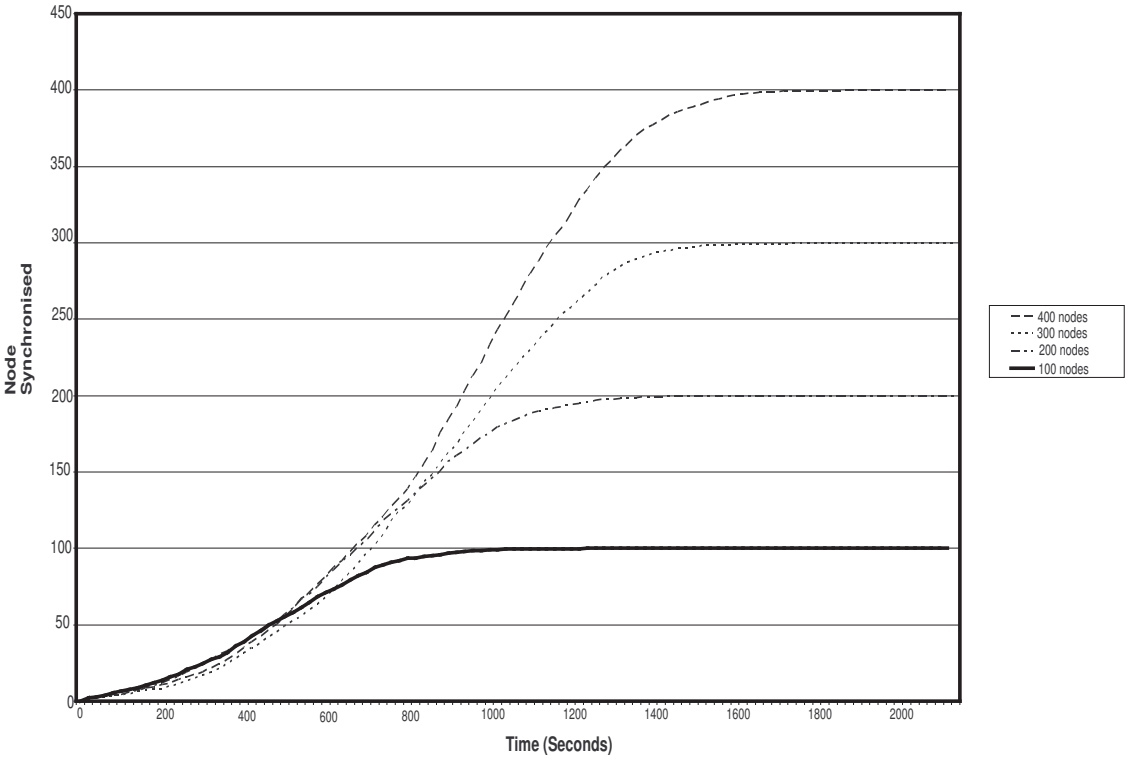


Figure 7.12: Synchronisation time for different numbers of nodes

the four tests are equal. The test area size is shown in table 7.7.

Node Number	Area (Square Metre)
100	1000 X 1000
200	1441 X 1441
300	1772 X 1772
400	2000 X 2000

Table 7.7: Node size and test area

The test shows that there is a certain period during which the tuple synchronisation characteristics are not affected by the size of nodes in the area, as long as node density and other node characteristics are equal. This is shown in the early section of the graph where the lines representing tests with different node numbers and area sizes are nearly the same. This represents outward growth from the origin of the change without effects from the closed nature of the space. Therefore, as long as a result is measured before saturation can affect the test result, it can be used to compare a tuple synchronisation characteristic of a mobile ad-hoc device in a way that is independent of node number and area size.

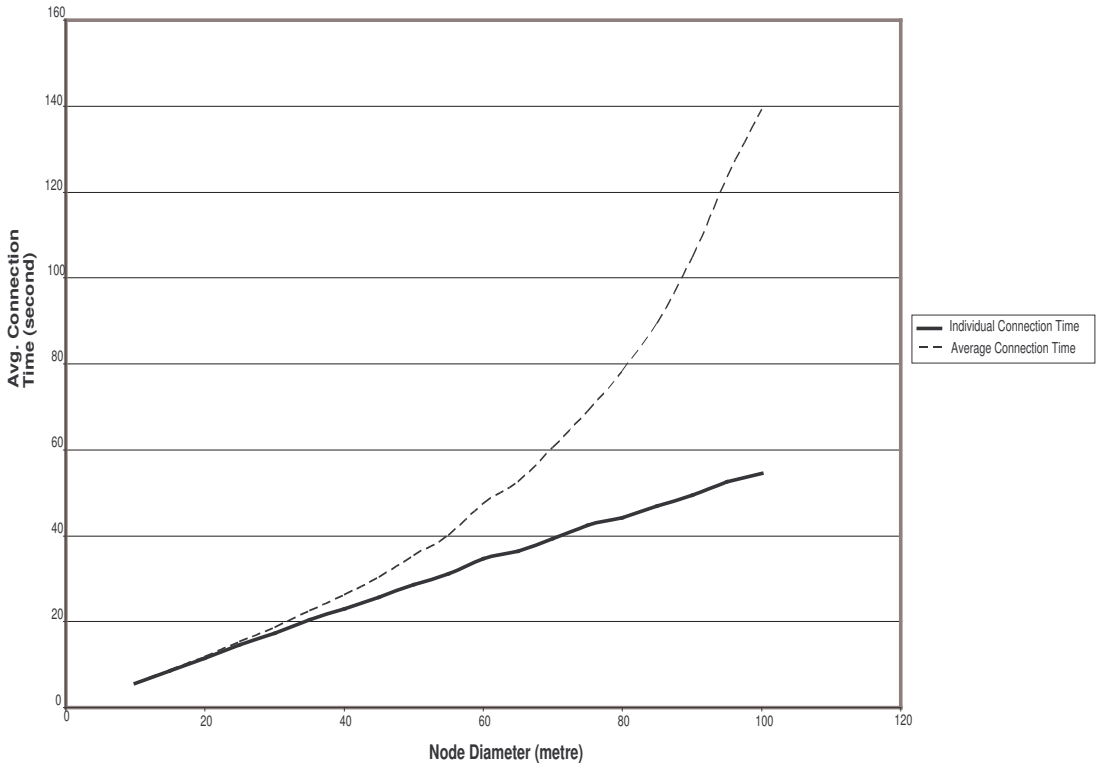


Figure 7.13: The average connection time for different network ranges

7.6.3.1 Node Connection and Disconnection

This section shows the connection and disconnection characteristic of nodes in an ad-hoc network environment. The first graph shows the average connection time for a node as a function of network range. The average connection time could be compared to the amount of time the middleware takes to synchronise a tuple in order to determine the success rate of the synchronisation process. Two types of average connection time are measured - the first is by dividing the total connection time of a node by the number of times the node is completely disconnected and the second is by measuring the average connection time during which a node has a connection as peer to some other node. As shown in figure 7.13, The average connection time measured by the first method is higher than the second method but both of them are higher with devices which have larger network range.

The average connection time in the second method is generally lower than the first method because it only measures a connection time between a pair of nodes while the first method measures the time on a cluster basis. In a situation where a node is connected to two other nodes, and the connection to one of the two nodes is broken, the first method still considers the node connected since it can still connect to one of the two nodes whereas the second method measures the two connections separately. The difference is larger with larger node sizes because there is a higher chance that nodes are connected in a cluster compared to tests where node sizes are small.

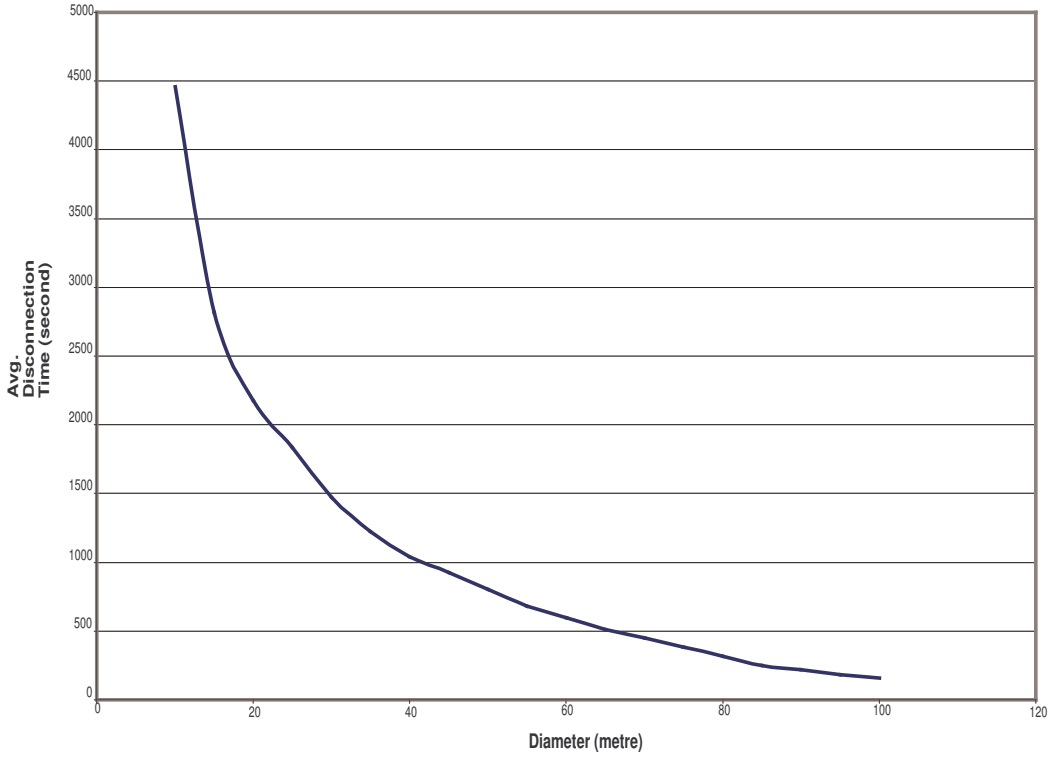


Figure 7.14: The average disconnection time for different network ranges

In a real scenario, the average connection time that should be used to predict the success rate of the synchronisation process falls between the two measured types. The cluster method can be too optimistic in the situation where a cluster is partitioned while a synchronisation is in progress between the two resultant parts. This will cause the synchronisation process to fail. The pair method is pessimistic because it only measures the time a node connects directly to the node containing the tuple, but aborts the synchronisation even though an indirect path still exists. In this case, the synchronisation process is not affected because the information can still be transferred via the link established through the other nodes. The simulation checks the path for each pair of nodes at every step.

Figure 7.14 shows the average disconnection time for nodes with different network ranges. Increasing the network coverage area significantly increases the chance that nodes will stay connected longer and improve a success rate of a synchronisation.

7.6.3.2 Synchronisation Time and Node Speed

Apart from a network coverage area, the node's speed is also one of the factors that affect the tuple synchronisation. Figure 7.15 shows the time to propagate a tuple to one hundred nodes in an environment where there are two hundreds nodes. The average speed of nodes in the tests is varied to demonstrate its effect on the total propagation time. Each line in the graph gives the result for a different time needed to

finish a single synchronisation step.

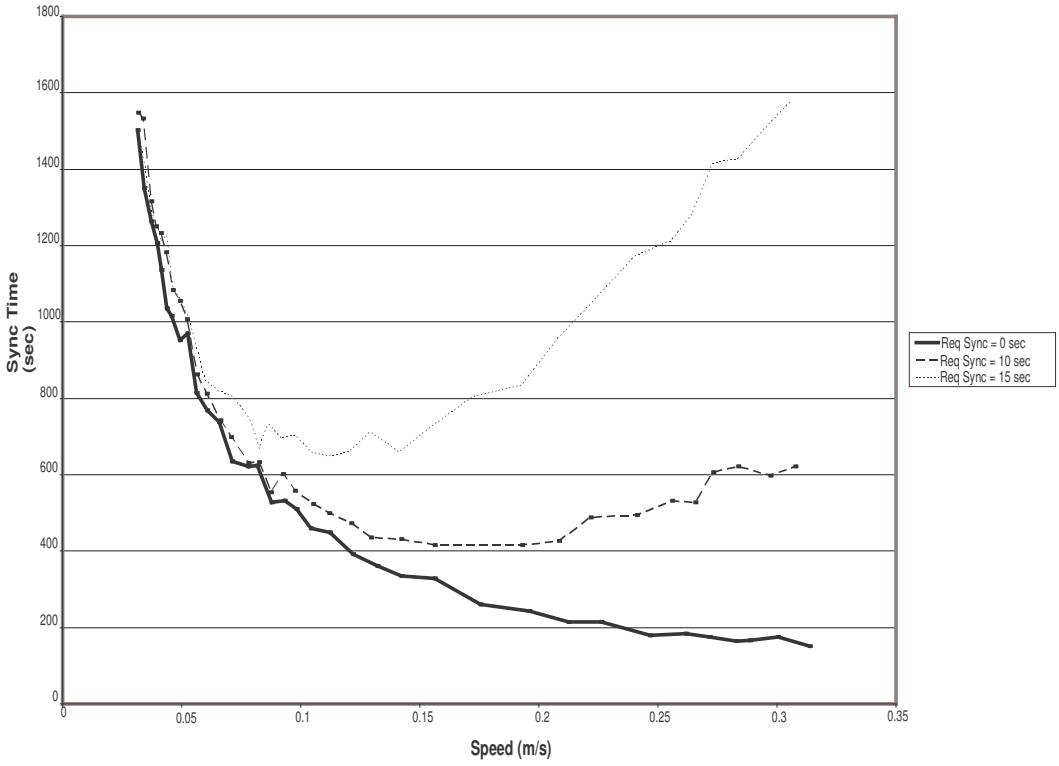


Figure 7.15: The effect of speed on the synchronisation time

As shown in figure 7.15, in the case where the synchronisation is instantaneous, the faster each node moves, the lower the time needed to propagate a tuple to one hundred nodes. This is because there is more chance for each node to connect with the others, which leads to more chance of synchronisation. However, this is not always the case in the other two lines. During the part with lower speed, the three graphs are similar even though the times required for synchronisation are diverging. This is because the average connection time between nodes is still greater than the time required for the synchronisation process. In the later part of the graph, the average connection time is lower when node speed is higher, increasing the chance that the synchronisation process will fail, resulting in higher total required synchronisation time. This situation causes the lines to separate because the tests with higher required synchronisation time needs time to redo any failed synchronisation process.

The result from this section shows that node speed significantly affects the time used to propagate a tuple in an ad-hoc environment. To be able to use the JSFM efficiently, the middleware synchronisation time, especially the period between keep-alive sending, has to be adjusted to suit the users' pattern of movement. The aim is to reduce the chance that the synchronisation process will fail because there is not enough time to finish the process during the connection window.

7.6.3.3 Synchronisation of Tuples from Multiple Nodes

The previous sections showed results based on a situation where only one tuple is propagated to the other nodes. This section, however, discusses the synchronisation characteristic when a number of tuples are written by several different nodes. In this situation, a queue of tuples awaiting synchronisation is an extra factor that will affect the time to propagate a tuple when more than one tuple is to be synchronised during each connection between two nodes. A tuple queue will be generated when the rate at which tuples are written is totally higher than the rate at which a synchronisation process is successful and the tuple can be passed on.

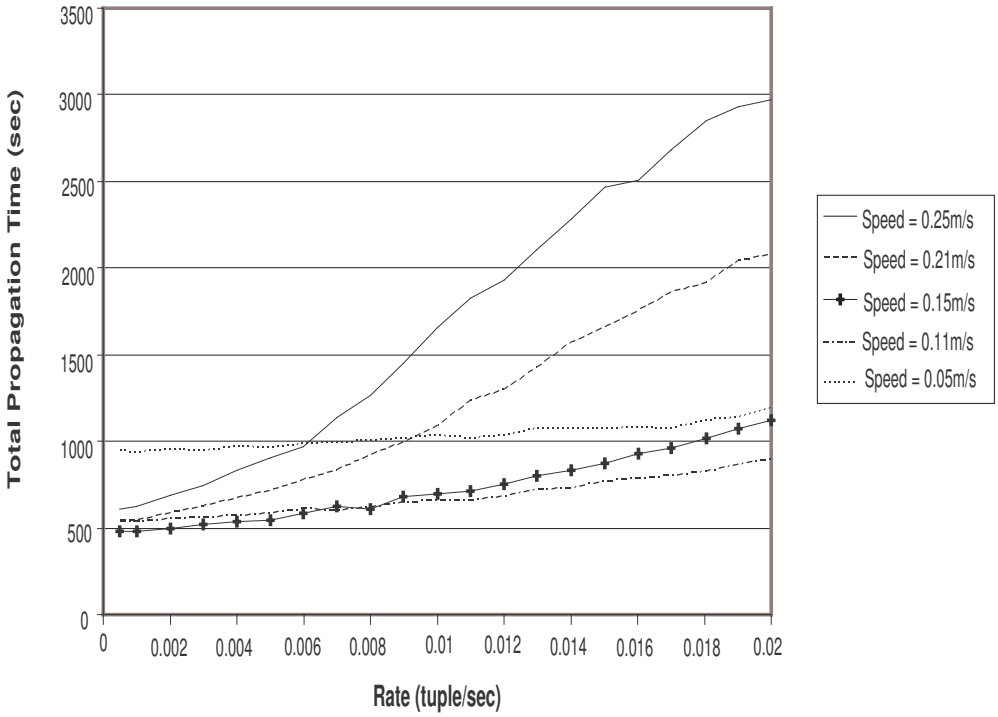


Figure 7.16: Multiple Tuples Synchronisation

During a particular synchronisation between a pair of nodes, the more tuples there are waiting for synchronisation, the more time the nodes need to stay connected. If there is not enough time to finish the synchronisation, some tuples will not be propagated. Figure 7.16 shows the effect of varying the duration between generating of new tuples on the time needed to propagate a tuple to one hundred nodes in an environment with two hundreds nodes. Contrary to the earlier test where only one tuple is propagated in the environment, tuples are generated from different nodes which will generate queues during the synchronisation. The size of the queue generated in this test depends on the rate at which tuples are written and the rate at which the tuples are synchronised.

With a smaller period between new tuple generations, in the right hand part of the graphs, the total

time required for propagating a tuple rises steeply. This results from the effect of tuples queueing to be synchronised. The graphs for different speeds have gradients that reflect the effect on propagating time. Generally, when devices are moving at the higher average speed, they have less chance to finish their synchronisations due to the lower connection time. This leads to the longer queues and a higher total tuple propagation time. However, there is an exception which can be seen from the graph for the lowest speed. The total tuple propagation time at that speed is generally higher than the results for higher speed. This is because at the lower speed, tuple queues are minimal and do not affect the propagation time as much as they do in the tests where nodes move at a higher speed, and the graph approaches a straight horizontal line.

The left hand part of the graphs shows the total synchronisation time in a situation where tuple generation rate is low and queues are minimal. It shows the behaviour of the system where the queues are short, in which the total propagation time will be affected more by the node speed than the queue. This is similar to the time to propagate one tuple as shown in figure 7.15. In figure 7.16, building a tree link takes eight seconds and a tuple synchronisation takes three seconds which results in roughly eleven seconds for a single tuple synchronisation. Therefore, the intercept for each line in the graph roughly reflects the tuple propagation time in the line representing ten second required tuple synchronisation time in figure 7.15. In this situation, the best performance can be gained by having nodes move at the optimal average speed.

The result of this test shows that the tuple queue during the synchronisation process affects the time needed to propagate a tuple to devices in a region and the effect is more relevant when devices are moving at higher speeds. Because of this, the performance of the system depends more on the number of tuples that are being changed in the region more than on the number of nodes in the region. The performance of the system should be acceptable even in a situation where there is a large number of nodes, as long as the number of nodes that generate the tuples or the rate at which the tuples are generated are small enough for the average synchronisation queue length to be small. Additionally, to maintain the system performance, the number of nodes and the rate have to be reduced when devices are moving at higher speed.

7.7 Conclusion

This chapter shows the results of several tests to measure the performance of various parts of the middleware. The access time test gives an idea of how long an application or user has to wait for the data. Different access commands take different amounts of time. In general, the JSFM takes more time to process each of its access commands than a normal JavaSpace does. This is because there is more than one JavaSpace Access command for a single JSFM Access Command in order to store the information which allows the JSFM to support tuple synchronisation.

The distribution tree building time gives the application designer an idea of how long a device has to stay in an area for it to be able to start a synchronisation process. Together with the average synchronisation time, this lets the designer determine if the JSFM can support an application. With the default keep-alive time, a situation where the device does not stay in the same place for more than eight seconds is not usable since the tree link will not be finished before the node leaves the area again.

The estimated tuple propagation time tests use the results of the initial tests to estimate the amount of time needed for a tuple to be distributed over the whole distribution tree. This estimate gives an idea of how adjusting the characteristic of the distribution tree significantly affects the overall synchronisation time. Lastly, tuple synchronisation in an unstructured environment was simulated. The section shows the characteristics of the synchronisation process in an environment where a number of small trees are created. In this situation, other factors such as device's speed and device's network coverage can affect the time needed to propagate a tuple.

Chapter 8

Middleware and Policy Discussion

8.1 Introduction

The previous chapters introduced all the concepts and implementations that are used in the middleware and its policy engine including the results of the middleware performance testing processes. From these chapters, there are a number of issues that should be discussed in more detail. These discussions cover points in the middleware building process - design, implementation, and testing - that required important decisions to be made.

Each of the discussions gives an idea of the available choices in the processes, their advantages, and their disadvantages. This chapter is divided into two main sections. The first section contains discussions that are related to the middleware implementation and design process. The second section contains discussion of issues concerning the policy and context information that is used in the middleware.

8.2 Middleware Design and Implementation Discussion

8.2.1 Application Requirement for a Virtual Space

There are a number of design decisions taken during the system building process, many of which affect the types of application that can benefit from the system. This section discusses the unique virtual space concept employed in this middleware, instead of looking at data in each space individually as is done in a number of other middlewares, and how a data synchronisation process affects applications that can profit from the system.

First of all, the concept provides an easy to use environment for writing general tuple space applications. It provides an environment that is similar to the environment that an application writer uses when writing a tuple-space based application on a static network because the entire data communication and most of the tuple synchronisation process is managed by the middleware. Therefore, an application

that is currently used in a centralised static tuple space environment could be ported to the middleware provided environment without many changes.

Together with information caching and tuple synchronisation, the unique virtual space view can be extended over disconnected sub-networks. It is a simple idea for increasing data availability without much involvement from a user. In this way, an application that requires remote information can be used in the environment. However, the version of the cached information on a disconnected device is probably not up-to-date. Thus, it is not suitable for an application that needs to make a decision based only on the most up-to-date information.

Moreover, the virtual space concept means all information is shared among devices. Therefore, it is suitable for an application in which the data access pattern is nondeterministic. However, the cost of storing every piece of data means that it is not suitable for the application that store a huge amount of data compared to the size of data storage on the device.

To provide the virtual space view, several underlying mechanisms have to be provided by the middleware, which increases the system complexity and reduces its performance. For example, the access time for each space interaction via the middleware is roughly three times the space access time without the middleware. Therefore, the system may not be suitable for a time-critical application.

Another main issue that affects the types of application is the time the middleware takes to synchronise tuples, which affects the chance that the information from two devices will converge during a limited connection period. From section 7.6, the middleware needs some time to complete its synchronisation process, if there are a number of tuples to be synchronised.

However, synchronisation in the JSFM is divided into a number of synchronisation processes each handling a single tuple. A disconnection will affect only the synchronisation process that is being done during the interruption. The JSFM has to redo only the tuple synchronisation that was affected by the disconnection. The processes that were finished before the disconnection are not affected.

The average connection time for an application that can use the middleware should at least be longer than the average synchronisation time of the tuples the application uses. This will allow information from two devices to converge as long as the rate at which new tuples are added to a space is lower than the rate at which the devices can connect and finish their synchronisations.

8.2.2 Advantages and Disadvantages of Using Policy Control

Using a set of policies to control a synchronisation process makes the process more flexible. There are two basic ways to do this. First, the system could allow a user to select from a pre-defined list of available policies. Second, the system could provide an interface that receives policies written by the user. This section discusses the costs and benefits from using the two ways.

For the first method, the middleware could provide a number of pre-defined policies that can be used to control its synchronisation process. The selectable policies could be taken from the policies that are

common and can be used in most applications, such as the policy that assigns tuple priority depending on the time the tuple was written.

In the second method, the middleware allows a user to create his or her policies and specifies the policies to be used. There is still a need, however, to limit what the user can express in term of the language he or she can use, the context information that is available for making a synchronisation decision, or the actions that are possible.

There are costs and benefits from using the two methods. For the first method, one of the obvious benefits is that the system has a lower learning curve than the second method especially for users who are not familiar with a policy language. Since this method allows the user to select the middleware synchronisation mechanism from a list of available options, it is possible to present the choices in a way that is easy to understand even to a user who has no programming knowledge.

Next, it is easier to create such a system because the method does not require a number of complex components such as a policy compiler and part of the policy engine. Since there is no need to support the policies that are created outside of the system, there is no need for the system to have a policy compiler. Its policy engine can be less complex because it only has to deal with existing controlling mechanisms.

Moreover, the process makes it easier to resolve a policy conflict problem. Since the programmer knows every possible policy that could be used in the system, he or she can define a mechanism that can be used to solve the policy conflict problem for each type of conflict or each pair of control mechanisms.

Lastly, the system will be more efficient and its performance will be better. For example, policy conflict detection will take less time since the possible conflicts and the mechanisms to resolve them could be pre-defined.

On the other hand, the first method limits the user's expressive power compared to the second method. The system does not provide any way for a user to introduce a new control mechanism that may suit a new application better than the existing ones. Even though a set of policies that can be used in most applications may be offered, there is no way that they can cover every possible control mechanism. A good example can be seen in 3.2.3 where data conflict in the taxi application could be resolved using a number of different policies.

It is possible to increase the expressive power of the first method by increasing the number of choices available to a user. However, this will increase the complexity and make it harder for a user to find and choose the policy to be used, which defeats the intention of using predefined policies in the first place.

8.2.3 Synchronisation Event Propagation Layer and JavaSpace Communication Service

After finishing a synchronisation process, the middleware needs to notify the other devices in the area of the fact that a new tuple has been created. JavaSpace provides a number of communication services that could be used for the notification process. However, this system does not employ these services.

It creates its own synchronisation event propagation layer to do the task instead. This section explains why the layer is required and why the middleware does not simply use the communication and event notification services provided by the basic JavaSpace interface.

Without the propagation layer, the middleware could rely on the communication services provided by JavaSpace. JavaSpace has a notification interface that allows a listener object to be registered which will notify an application when a matched tuple is written into the space. Using this interface, each device can register with the other devices in the area to receive a notification when a new tuple is written.

The most obvious advantage of using the interface is that the middleware does not have to spend time building and maintaining the propagation tree. This greatly reduces the delay when a device enters an area and establishes a tree link with the device in the area. As seen in the testing result in 7.3.2, the tree building process takes a long time compared to the synchronisation process.

Next, the middleware does not have to constantly send and receive keep-alive messages used for maintaining the tree. This frees up some network bandwidth and processing time. The size of this effect depends on how often a keep-alive message is sent which can be adjusted in the middleware.

However, the system still chooses to have its propagation layer because it reduces the problems the middleware could have if it simply used the interfaces provided by JavaSpace for its synchronisation event propagation process. The main disadvantage of using the JavaSpace interface is that it is harder to control and limit event propagation.

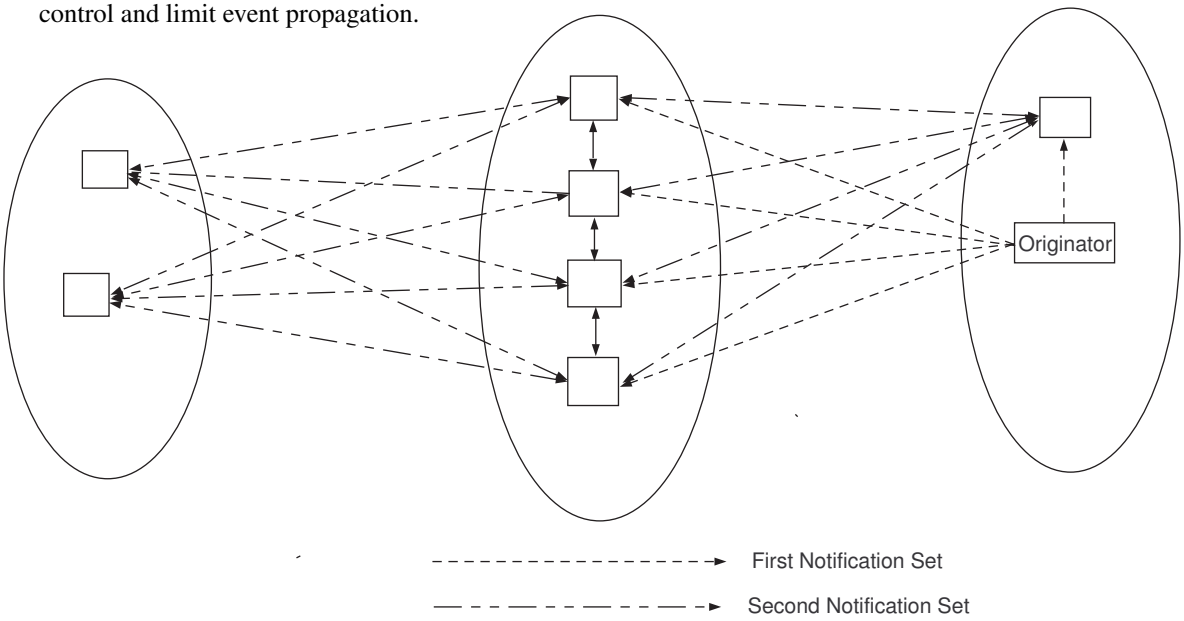


Figure 8.1: A Multicast Storm Situation

An example of the control problem can be seen in a situation where devices in one group cannot connect directly to devices in another group, but can connect to them using devices in a third group as a bridge; this situation is shown in figure 8.1. The devices that act as a bridge are responsible for delivering

an update message from one side to the other. However, since every device has to register with all other devices, the notification messages from the bridge devices will also be sent back to all the devices on the side that originates the message except the message origin. Moreover, the devices on the receiving side of the bridge will receive redundant notification messages from every bridge devices they connect and register with. The situation results in a number of wasted notification messages that have to be filtered by the middleware.

Using the event propagation layer does not suffer from this problem because it creates a tree linking between devices. The destination of the synchronisation propagation message can be controlled by the propagation layer. Figure 8.2 shows a layout of the event tree that could be built in the same situation. The number of notification messages is roughly equal to the number of tree links, which is less than the number of messages sent in figure 8.1.

Even though the tree maintenance process uses network bandwidth for its keep-alive messages, the bandwidth used in this way is small and constant. On the other hand, the broadcast storm in the first situation could be more disruptive. The middleware still uses JavaSpace provided interfaces for the actual tuple communication between devices because of their simplicity.

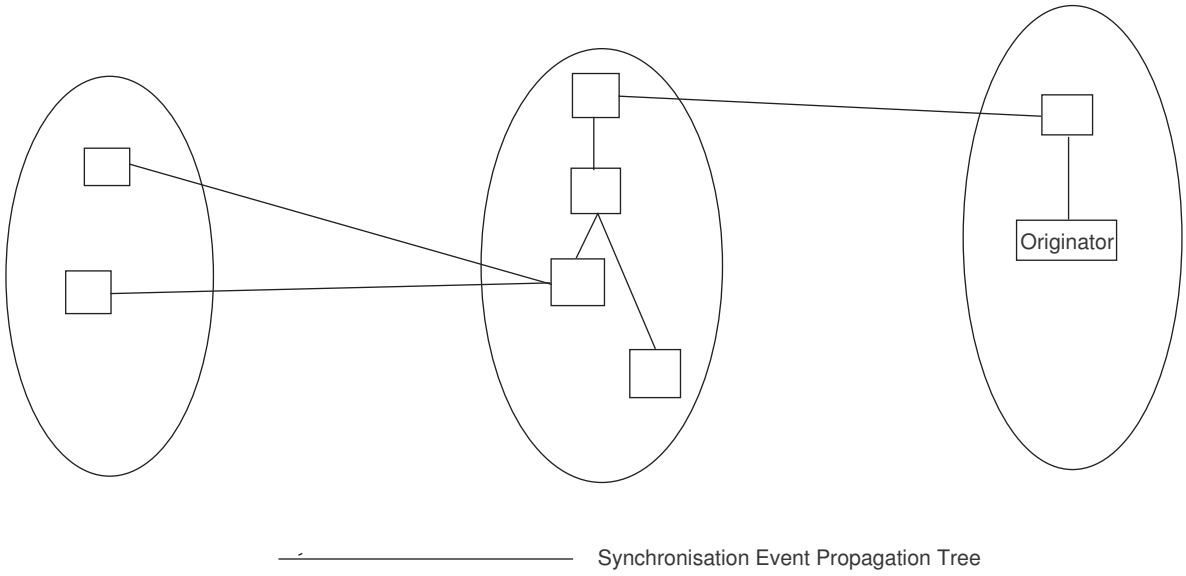


Figure 8.2: An Event Notification Sent via Event Tree

8.3 Policy and Context Information Discussion

8.3.1 Event Types and Policy Body

The policy used in this middleware is divided into two parts - the event definition and the policy body. The first part contains a definition of the type of synchronisation events such as a synchronisation where the extra tuple exists in one space, or an event where there is a tuple identity clash. The policy body contains the other elements of the policy such as the subject definition and the constraints. This section discusses the benefits and some disadvantages of dividing the definition of policies in the middleware into these two parts.

Dividing the policy this way makes it easier to read. A synchronisation event in the middleware is usually composed of information regarding the tuple in the two spaces being synchronised and some miscellaneous information. To write the whole information in the policy body makes the policy look more complicated than it actually is. The dividing process allows us to use just an event name to represent the event which makes the whole policy shorter and easier to read.

However, the benefit gained from the situation also depends on how well an event is named. The benefit from the process will be lost if the event is not named properly. The name must convey the right meaning from the event definition so that a policy writer does not have to look into its definition or risk using the event incorrectly.

Next, the division of the policy allows the two parts to be reusable. Some time can be saved in defining a number of policies, since an event needs to be defined only once to be used in several policies. In the same way, this makes managing and modifying policies easier. If the definition of an event that is used in a large number of policies needs to be changed, only one event definition has to be modified, which will affect the entire set of policies that employ the event.

The entire definition of every event can also be changed by changing the file that is used as the event definition. For example, in a situation where the system needs to be changed back and forth and there are differences in the event definitions between the two versions, the user only needs to point the middleware at the new event definition file each time the system is changed, assuming that the event names in the two versions are similar.

Nevertheless, there are some disadvantages from the division, particularly if the system uses a small number of policies and each is applied to a different event. The policy definition process will take a little longer than defining the whole policy in one place. Next, as discussed earlier, the event name becomes very important. There is a chance that a policy will be defined incorrectly because of the misunderstanding of the event names, especially in the case where more than one person is responsible for defining policies for a system.

8.3.2 Conflict Detection Process and Tuple Identity Discussion

Conflict detection and resolution are the main processes that are performed by the middleware policy engine. The middleware conflict detection relies on detecting tuples in different spaces that have the same tuple identity. The conflict can then be resolved using a resolution process controlled by selected policies. This section discusses issues concerning the middleware's conflict detection and resolution processes.

8.3.2.1 Conflict Detection Scope

To make the conflict detection process simple, the policy engine decides that there is a tuple identity clash if, on the two spaces being synchronised, there are tuples with the same application level identity but with different writer and sequence number fields, which is the middleware level information representing the last device that modified the tuple.

There are two reasons why the process can detect the identity clash. First, from an application point of view, a tuple is identified by its application level identity. An application accesses a tuple by using its identity for the tuple space pattern matching process. If any two devices have tuples with similar identities, they are either the same tuple or they are conflicting tuples.

Second, the difference in the middleware level information shows that there is a high chance that the information carried in the two tuples is different, since the tuples are edited or created by two different devices. This second fact shows that the two tuples, at least from the middleware point of view, are different and their information was last modified by different devices.

In brief, the two notions show that the two tuples will be seen as the same tuple from the application point of view but have been modified from different devices. This situation leads to a high chance of a conflict between the two tuples.

However, the process does not take into account situations where the two tuples may still have similar information which does not cause a tuple identity clash even though their middleware tags (the sequence number and the writer) are different. This situation can occur if the two devices editing the tuples happen to write the same information into the tuples and there will be no information conflict between them, at least from the application point of view.

On the other hand, this situation will not occur if the process checks each field in the two tuples to identify whether they are similar. In this way, an unnecessary synchronisation process between the two similar tuples can be avoided.

However, the reason why the conflict detection process is implemented in this way is to reduce its cost. Checking every application field in the tuples that have identity matches takes a lot of time. Moreover, this process needs to be done every time there is a tuple identity clash. It is better to allow an unnecessary synchronisation process if the two tuples are actually similar, which should not happen frequently because the tuples are edited from two different sources, than to increase the time of every

synchronisation process to check for a case that rarely happens.

Moreover, even in the case where the two tuples are similar from the application point of view and there is no need for a synchronisation, there is still a problem of different context information that is attached to the two tuples. The two tuples modified by different sources will always have different context information such as the time they are modified, or the space that modified them, for instance.

The information may not be required for the current synchronisation process but it may affect the result of the later synchronisation when there is an actual conflict. The conflict detection currently used in the middleware does not have this problem because it always detects tuples as conflicting if they are modified by different sources.

8.3.2.2 Tuple Identity Issue in Conflict Detection

As introduced in 5.2, there are a couple of problems that cannot be addressed in this type of detection process. First, the process cannot detect some types of conflict. Since the process relies on precisely matching identity fields, it cannot support a detection process between information identified by a range of data that has a conflict in some of its parts.

Time is the first example of this. Consider a room booking application where users share their room booking information using a room number and a booking period as the identity. There will be no problem if users are only allowed to book a room at a specific period such as only at the start of each hour and the booking period lasts for a predefined duration, such as shown in figure 8.3.

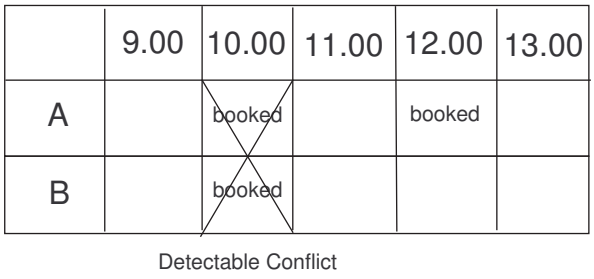


Figure 8.3: Room Booking with Precise Booking Period

However, a problem occurs if there are no such rules in the room booking application because one booking may cover a part of another booking. For example, if one user books a room between 9.30 to 12.00 and another books the same room between 10.00 to 11.00 as shown in figure 8.4, the current conflict detection, which relies on matching the tuple identity, cannot detect the conflict.

A similar situation can occur in a number of applications that rely on more complex rules for detecting conflict. The middleware does not provide an interface to resolve this kind of conflict because it is more complex and will require a lot more user intervention than the current process. On the other hand, since

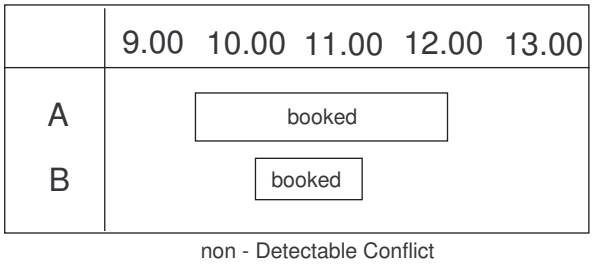


Figure 8.4: Room Booking with Non Precise Booking Period

the detection process follows the tuple matching process, an application that requires more complex rules for detecting conflict will find that the tuple matching process provided by the tuple space paradigm cannot easily be used. For example, in the room booking application, if a user can book a room at any particular time, the process of tuple matching in order to find the tuple will not be usable, since the process also relies on the exact match of the query fields. To allow this kind of application to be usable in the tuple space environment, the application needs to read the entire set of tuples (for example, using "readAll()" command) and then use an application dependent search algorithm to find the queried tuples after they are read from the tuple space.

To allows the conflict detection process, the middleware needs to provide an interface that allows a user to tell the middleware how a conflict should be detected. This could be done using an extended policy language that allows the user to specify the semantics of the process. This semantics would have to be associated with each type of tuple just like the tuple identity.

The second issue concerning tuple identity in conflict detection is what happens if the identity of a tuple is changed. Since the tuple identity is simply a set of fields in the tuple, there is no rule that prevents an application from changing values in the fields. The conflict detection process implemented in the middleware cannot support this kind of conflict.

An example of the situation can also be shown with the same room booking example. Instead of just booking a room, there is a situation where a room number has to be changed. This could happens from a change in organisational structure that changes how a room is labelled. In this case, the identity of a tuple is changed while it is still meant to be the same tuple. However, the conflict detection process will understand the tuple with the new identity as a new tuple and no conflict resolution process will happen.

Again, the middleware needs user intervention in order to solve the problem. The situation can be resolved with the modifiable conflict detection policy discussed earlier. In this way, a user could modify the middleware conflict detection for the room tuple to detect the situation as a conflict.

However, there are a number of issues that arise from using the process in this case. First, since the new conflict detection process is introduced while the system is in operation, there is a need for a way to

propagate the new detection process to the other devices to prevent a conflict between conflict detection processes.

Second, there is a question of how this could affect the policies that are being used in the synchronisation process. For example, should the changing of a tuple identity have a greater precedence over other kinds of change and how could this be described in the policy. It would be unsatisfactory if the change made to a room number is done by a low level officer with lower priority and so is always overwritten by other users.

8.3.3 Context Information Usage in Synchronisation

To resolve a conflict between tuples, the middleware policy engine relies on context information attached to the tuples to make its synchronisation decisions. However, since the process relies heavily on the information, misuse of context information can create problems in the system. This section discusses effects that can happen as a result of using the information incorrectly.

The usage of context information that in itself is not unique will eventually leads to the situation where the middleware cannot resolve a conflict between tuples. Examples of these context information are class of spaces and device type. It is better to ensure that the combination of types of context information makes the information in an individual device unique. For example, two policies may be used in a synchronisation process. The first policy may determine the process result using the class of device or user information. This information is normally not unique. Therefore, a fail safe policy is used in the case where the two devices being synchronised have the same class

It is possible to write such policy using a condition in the policy constraint. The second policy is to be used only if the two devices belong to the same class, whereas the first policy is only used if they are from different classes.

Figure 8.5 shows an example of the situation described above. A class of tuple is the main information used for making a decision in “pol1”. “pol2” is a fail safe policy using the time when a tuple is written to determine a synchronisation result. The combination of tuple class and written time information should solve most of the cases.

Without using any context information at all, some simple policies can cause problems. For example, a policy that always gives precedence to a remote device or to a local device will not make the synchronisation process possible, especially in an environment where the policies on every device have to be similar. The process will result in the two spaces either exchanging their conflicting tuples or doing nothing.

```
inst oblig pol1
{
    on                  Sync_BothSide
    subject  <space>    s = /space;
    target   <tuple>     t = /tuple;
    do                  s.replace();
    when                /tuple -> exists(t1, t2 | t1.localtuple = true and
                                t2.remotetuple = true and
                                t1.class < t2.class);
}

inst oblig pol2
{
    on                  Sync_BothSide
    subject  <space>    s = /space;
    target   <tuple>     t = /tuple;
    do                  s.replace();
    when                /tuple -> exists(t1, t2 | t1.localtuple = true and
                                t2.remotetuple = true and
                                t1.equalclass(t2.class) and
                                t1.writtenBefore(t2));
}
```

Figure 8.5: An Example of Proper Policy Usage

8.3.4 Effect of Increasing Context information in Synchronisation

The decisions made during a synchronisation process by the middleware policy engine depends on the policies that control it. In order to make the right decision, each policy compares the context information attached to a tuple to determine which tuple will be preserved and which tuple will be deleted. This section discusses the effect of adding more context information for use during a synchronisation process.

New context information could be added to the middleware. The middleware provides interfaces that help gathering information and keep it in a space. This extra information helps the middleware to support more types of application or allows a user to be more flexible in designing his or her synchronisation process to suit a particular application.

For example, the taxi service application does not require any more context information than simply the time a tuple is created to make a basic decision. However, information about taxi location, speed, and the average traffic flow in the area would help to make policies used in the taxi application more adaptable and less prone to make the wrong decision.

However, increasing context information increases the amount of information that needs to be transferred during a synchronisation process. The context information for a particular tuple needs to be attached to the tuple in the form of a linked context tuple. Every time a tuple is propagated from one

space to another, the context tuple must be propagated as well. This is to support the synchronisation process when the tuple is in a space where it did not originate.

Moreover, more context information needs more storage. The context information is normally stored in the form of a set of context tuples associated with the data tuple. Adding new context information means adding another context tuple to be associated with each data tuple. Even though each context tuple uses only a small amount of storage space, a large number of them may add up to a large amount of space needed to store the context information.

Adding more context information may also cause a problem of consistency during the synchronisation process. Currently, this project assumes that every device has the same set of context information available. Without this assumption, using extra context information may causes the policy engine to make a wrong decision or cause an error if the policies being used are not written to cover the situation where a participant may not have the required context information.

8.3.5 Policies Customisation Discussion

The aim of customisation of policies in individual devices is to allow a user to have control over his or her own device. However, even though the idea increases the system flexibility, it also has a number of costs which are discussed in this section.

The section discusses the advantages and disadvantages of implementing a synchronisation process using multiple policies in the middleware. It is divided into two parts. The first parts discusses the costs and benefits of allowing event types to be customised. The second part discusses the advantages and disadvantages of using multiple sets of policies in different devices.

8.3.5.1 Event Type Customisation Discussion

The policies used in the middleware are divided into two parts. Ideally, the system should allow both the two parts to be customised by each individual user. Event type customisation would allow a user to create his or her own event definitions.

However, this project does not cover this point because it would add a lot more complexity to the system. Since the policy to be used for tuple synchronisation is initially determined by the event type, the process assumes that the meaning of the event is similar in every device.

With this assumption, the two devices being synchronised will select their policies using the same synchronisation type. If a device is allowed to modify its event definition without notifying the other devices, the decisions made by their policy engines may diverge.

This is because for multiple policies, the synchronisation process relies on one device sending synchronisation information to another device, and using its decision to compare with the result of the local decision to detect conflicts between the policies in the two devices. The event type is one of the pieces of information that is sent during this process.

There is also a problem if a device creates a new event definition without the other devices knowing about it. In this case, the synchronisation process may not work properly because the space that receives the event type does not have any information about the event and there is thus no policy associated with it.

If event customisation is to be allowed, an event definition synchronisation process has to be created. This process has to check the definition for each event used in a synchronisation process and make sure that their definitions are synchronised before the tuple synchronisation process is started.

The event synchronisation process may, as a result, also require another set of policies to resolve any event definition conflict. This may in turn lead to another set of policy conflicts, if the policies used to resolve the event conflict are themselves different between devices.

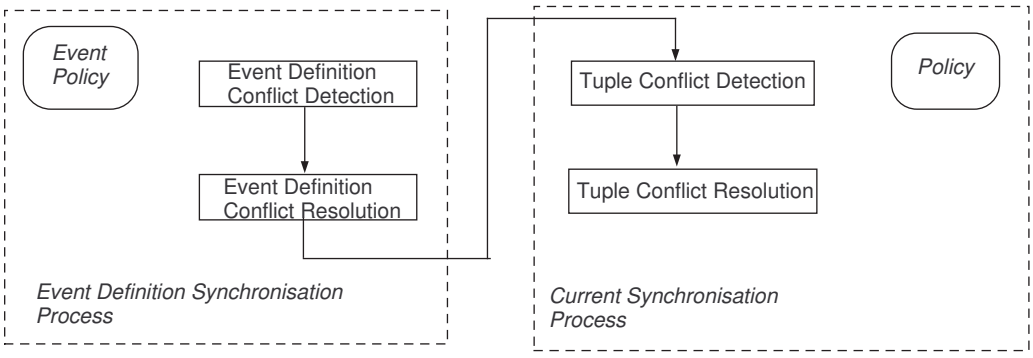


Figure 8.6: A Synchronisation Process with Event Customisation Enabled

As shown in figure 8.6, to enable event customisation in each individual space, the event definition synchronisation process must be done before the current synchronisation process can operate. The cost of the new process may be as high as the current process because it again requires both conflict detection and resolution. This situation will double the cost of the middleware synchronisation process, which may outweigh its benefits.

8.3.5.2 Effects of Multiple Policies

Even though the middleware does not support the individual customisation of event definitions, it still supports a synchronisation process where each device has a different set of policies. With this process, a user can create, modify, or delete a policy on his or her device without a need to notify any other users. The synchronisation process is responsible for detecting any conflicts that happen between policies, and for finding a resolution of them. More details about the process are given in section 6.3.

Allowing a user to define policies for his or her device gives a greater sense of control to the users

than setting the same set of policies for every device even if the policies defined are changeable. This also increases the system flexibility and allows interactions between users with different policies.

For example, users from different divisions in the same company may access the same application but use different policies for data synchronisation. The multiple policy support allows the users in the two divisions to create a virtual space even though the policies on their devices are different. Without this support, one of the two divisions needs to change its policies to the set that the other division currently uses before they can interact.

However, the multiple policies mechanism greatly increases the complexity of the middleware policy engine. A number of processes have to be added to the engine such as policy conflict detection and resolution. These added processes have a significant impact on the performance of the middleware.

The results from the synchronisation testing presented in 7.5.2 show that the process needs more than double the time needed for the simple synchronisation process. This situation reduces the number of applications that the middleware can support, especially disadvantaging applications where the devices are moving at higher speed. Next, the multiple synchronisation process makes it more complex to define policies for an application, since a higher level policy for resolving conflicts between the lower level policies also needed to be defined. There is also a need, at some levels, to create a policy that can always resolve a policy conflict.

Moreover, the more levels of policies exist in the system, the more chance there is that the system will take a longer time to finish the synchronisation of a tuple. The system administrator needs to decide how many levels of policies can be modified by users, weighting the users' freedom against the cost of the synchronisation process.

8.3.6 Synchronisation Policy Discussion

Even though the Ponder language is used to control the synchronisation process in this project, it is not a tailor-made language for the process. The language is designed so that it can be used in different areas which makes it impossible to satisfy every requirement from each area. This section discusses example features that could be added in order to make a policy language that is more suitable for the synchronisation process.

First of all, it is not easy to define what could and could not be expressed in a policy using the language. For example, one action may be useable in a particular application but might not be understood by others. An unwanted effect may occur if a policy contains an action that is not supported by an application. It would be better to have a feature that allows a system administrator to limit available actions that can be used in each application. The same could be done for the available subject, target, and constraint sections. More discussion concerning this issue can be seen in the future work (9.2.4)

Next, tuple identity clash is one of the most important and complex issue in tuple synchronisation. As discussed in 8.3.2.2, there are different type of conflict for different applications and another set of

policies that is used to define how conflict is detected in each application is needed. Conflict detection semantics should be written in a separate file which allows them to be reusable. The event type specification file needs to contain the conflict detection process to be used for each type of tuple. This could be specified in the same way the tuple identity is currently specified.

Another feature that might be useful in a synchronisation policy is to allow a writer to specify an in-device policy priority. With this feature, the policy engine would always try to enforce the policy with the highest priority first, so long as the current situation matches the policy constraint. If it does not, a policy with lower priority will be used. The feature is not crucial but it makes specifying policy easier. For example, it is possible to specify a set of policies that will check a tuples' class and then a tuple's last written time if there is a policy conflict. However, it is harder to write these policies without knowledge of which policy will be checked first. The addition of a priority section into each policy will make policy ordering possible.

8.4 Conclusion

This chapter has discussed several issues in the design of middleware and policy systems. The first part discussed application areas that could benefit from the system. The middleware tries to emulate the unique virtual space environment that is generally used in a wire based tuple space environment. This will allow a number of applications to be ported to the middleware without a lot of modifications.

The middleware synchronisation process determines if it is suitable for an application, especially considering the time the middleware takes to finish the process. Therefore, the synchronisation process is designed so that it can be broken up into several small steps so that a disconnection will interrupt just the step that is being done.

The second section discussed options on how to use a policy to control the synchronisation process. Two methods were raised as examples. Allowing a user to select a policy to be used from a list is easier and more efficient. However, even though the option makes a synchronisation process simple, it severely limits the flexibility of the process. This project chooses to provide an interface that receives policy files from a user instead. The flexibility gained from it outweighs the cost of providing the service.

The next section discussed the benefit and cost of implementing the event distribution layer to replace the notification service provided by JavaSpace. The new layer allows the middleware to control the distribution of synchronisation events so as to reduce the broadcast storm problem that occurs if a simple notification service is used. In most situations, this benefit outweighs its cost even though events will be propagated from one device to another device which makes the propagation process slower.

Next, the chapter discussed the benefits and costs of dividing the policy into two parts - the event type and the policy body. This makes the policy easier to read and easier to modify. The next section discussed issues concerning tuple identity in a synchronisation process. It directly affects the system's

power in conflict detection because the process relies on values in the identity fields to determine conflict between tuples. The conflict detection process was designed to be simple so that the synchronisation process will not cost too much. The section discussed options that will make the process more powerful and enables it to detect more types of identity clash.

The next section discussed how context information can be used to help the policy engine make synchronisation decisions and the effects of increasing the amount of context information used in the system. The more context information the middleware has, the more chance that its synchronisation process can be tailored to match application requirements. A small amount of context information is enough to allow it to do the basic synchronisation process in most cases but with more context synchronisation, it can make decisions that are better suited to the situation.

The last section discussed issues concerning the synchronisation process in an environment where there are several sets of policies in different devices. A more complex synchronisation process is required to support such an environment. The section discussed the reason why the system supports only policy body modification but not event type modification. The section also discussed the effects in term of costs and benefits of providing a multiple policies synchronisation process which increases the system's flexibility but also raises the cost of synchronisation.

Chapter 9

Conclusion and Future Work

The aims of this project have been to investigate how to provide an information synchronisation process for a tuple space based application that is flexible enough to support a number of applications without a user having to modify a system heavily to support each new application.

A policy language was chosen as the way to provide flexibility for the synchronisation process, since it allows a user to modify the middleware behaviour without having to modify the system. A set of policy files written by the user is stored in the system. When the synchronisation situation meets the condition defined in one of the policies, that policy will be used to make a decision in the synchronisation process.

To make the right decision, a policy needs some information regarding the circumstance in which the current synchronisation process takes place. Information relating to the tuples being synchronised, such as the time they were written or their class, could be used. Likewise, information regarding the surrounding environment can also be used. This context information, together with a policy, allows the system to support a wider range of applications and allows its synchronisation process to be tailored to suit a particular application.

Next, the project went further by trying to support an environment where different users are allowed to apply different sets of policies to their systems. The environment gives users more freedom to choose policies for their systems but creates a chance that there will be a conflict between policy decisions during the synchronisation process.

The project chose to provide a multi-level synchronisation process that uses higher level policy sets to resolve conflicts between lower level policies. Each user can define his or her high level policies but this project still relies on the assumption that there is a single default higher level policy set that is agreed upon between users.

To investigate the idea, the JavaSpace For Mobile Environment middleware has been created. It is a tuple space based middleware that supports use of policy for its tuple synchronisation process. The middleware provides a unique virtual space that extends the tuple space view to an ad-hoc sub network

using information caching. The middleware synchronisation sub-system is controlled by a set of policies that can make synchronisation decisions based on context information.

9.1 Conclusion

This project shows that it is possible to build a system that can support a wide range of synchronisation options where the process is controlled by a set of policies written by users. Since different applications have different ways to resolve conflicts in their information, it is hardly possible to provide a single synchronisation algorithm that can support them all. Therefore, a system that is built to support such a situation needs a way to adapt its synchronisation engine to different situations and new applications.

Using a policy is one solution to the problem. The policy language should at least allow a user to define what action is to be applied to each synchronisation event and under what conditions. There are other ways to achieve the same effect. A group of selectable choices and a controlling process using a programming or natural language can do the same task. Each has its advantages and disadvantages. Policy Languages seem to be the best way in terms of expressive power, the user's learning curve, and the implementation complexity.

The Ponder Language was the language chosen to be used in the middleware. It allows a user to write policies by defining an action that will be undertaken when the condition in the policy is met, meeting the requirements described above. However, since the language is designed to solve a range of problems in various types of system, its expressive power is more than required for the middleware synchronisation system. For this reason, one of the most important future work items is to prevent a user from defining a policy that uses functions that are not supported by the middleware.

The middleware prototype built in this project has been used in a number of different synchronisation situations by applying different sets of policies. With the middleware, it is easier to extend a tuple space based application that is currently used in a static network so that it can be operated on a wireless network. The system provides a communication and synchronisation sub-system, which reduces application complexity.

One factor that affects the power of policies used by a synchronisation process is how much context information is available during the process. Only a small amount of context information allows a policy to support several types of synchronisation behaviour. However, increasing the context information allows the policy to express decisions that may be more appropriate to the situation. Because of this, it is better to create a system that allows a user to add or remove context information to support new policies for new applications.

An environment where different users apply different sets of policies to their devices makes the synchronisation process more complex. The synchronisation engine then requires extra functionality that gives it the ability to detect any conflict between decisions made in different devices and allows it to

reconcile such conflicts.

There are several ways that a policy engine can detect policy conflicts. This project chooses to detect conflict by comparing decisions made by the engines on different devices. This makes the system less complex than detecting conflict at a policy level by comparing policy files, even though it is less powerful. A policy conflict reconciliation process is provided by using a set of higher level policies to resolve conflicts that occur between lower level policies. This method is easy to implement but still requires a certain level of cooperation between users to set up a high level default policy for use in cases where a conflict cannot be resolved.

Furthermore, policies and context information associated with a tuple (synchronisation control information) have to be sent along with the tuple, when it is transferred to another space. This is especially important in an environment where different devices use different policies because it allows a tuple to maintain the priority level given by its origin. However, the process of transferring the information can be expensive. A less expensive but slightly less precise method has also been proposed in this thesis.

Even though the middleware can be beneficial for an application writer who wants to create a tuple space based application in a mobile environment, there are costs in providing the synchronisation service. First of all, there is still a learning curve for a user to learn how to write a proper policy. Next, the application efficiency will be somewhat reduced compared to a basic tuple space service because of the middleware overhead of providing its synchronisation sub-system information.

Moreover, the synchronisation cost increases when more flexibility in policy definition is given to the user. For instance, the multiple policy environment allows each user to define policies on his or her local device but the synchronisation process needed to synchronise tuples in the environment is more complex and takes a longer time to finish than the simple synchronisation process.

The same trend will continue as more flexibility is added to the system. Customisable event definitions will allow a user to define a specialised event for his or her device but will increase the system synchronisation process complexity to support the differences between devices' events during the synchronisation process. Therefore, this synchronisation cost has to be taken into account before new features that make the system more flexible and convenient are added.

9.2 Future Work

9.2.1 Increasing the System Performance

Since the prototype is built as a proof of concept, its performance may not be as good as a general commercial product. The system efficiency is mainly based on three sets of operations - a space access operation, an event tree building process, and a synchronisation process. Increasing the efficiency of the three operations will help in improving the system performance which will allow it to be used in a wider set of applications.

The space access time depends largely on the number of accesses needed for one command and the storage access time. The only improvement that could be done in this area is to reduce the number of times the middleware needs to access the JavaSpace. However, it is rather hard to reduce the number because a number of interaction is required to save the support information used during the synchronisation process.

A better tree building and maintaining protocol could be used in the synchronisation event distribution layer, and this could reduce the time it takes to create a tree link. The improvement in the process would allow the middleware to be used the applications which have a short connection window, such as applications operating in a car. Since the prototype can be divided into a number of layers, it is possible to replace the entire event distribution layer with a new layer implementing a new protocol without many changes to the other layers.

The synchronisation process that supports multiple policies has to rely on decisions from the remote space. There is not much that can be done to improve the algorithm as long as a remote decision is still required, since the network information transfer and the time the local device has to wait for the remote decision is longer than the time it takes for the local system to make a local decision.

Another improvement that could be made to the middleware is to create a new tuple space service to replace JavaSpace and Jini. First of all, they are not designed to be used in a mobile environment and using them this way is inefficient. Next, they are designed to be operated on a well provisioned static system, but the same level of available resources may not be available in a mobile environment.

9.2.2 Make the System Easier to Use

Currently the system requires some modification in order to allow new types of context information to be declared to its policy engine. It is currently not easy as to add new information without recompiling the system.

The system modification should allow the definition of new context information without the need to edit the middleware source code. A user should only need to create a new Pol_Obj (see 5.5.1) which is an interface to the new context information. This will contain information such as the library calls that allows it to be queried from a policy file. The synchronisation engine should be able to read the new object and then modify its behaviour to support the new information without any intervention from a user.

A similar case also applies to policy actions. A user currently needs to modify the middleware source code to define new actions to be used in the policy action part. Even though the middleware provides a set of primitive actions that are supported by the system, the process of putting together the primitive actions to form a new action command to be used in the policy still needs a modification to the middleware source code. Additionally, different applications will require different types of action and some are not available. For example, an application may require the JSFM to notify it when the tuples

that it wrote or the subset of the tuples has been accessed or removed. This kind of action is not hard to add but still requires editing of the middleware source code.

Apart from the system modifications to support dynamic editing given as examples above, another way to make the system easier to use is to provide a tool that can help users in interacting with the system. Currently, the middleware does not provide any tool that helps a user in creating new policies and notifying the system about them. A graphic user interface tool that helps in creating a policy, checking if the policy will be compatible with the system (see 9.2.4), compiling it to a Java object class, and letting the user declare the level of the policy, would make the process of interacting with the system easier.

Lastly, a debugging tool that allows a user or a software developer to see how their applications behave in the system would help in the process of creating or modifying applications to be used with the middleware. An example is the tool that reads and shows information of every tuple in a particular space. This would allow an application writer to inspect results for each space interaction including the result of a synchronisation process. This kind of tool has actually been used heavily during the middleware debugging process.

9.2.3 Other Ways of Conflict Resolution

Currently, the system allows a conflict to be signalled if there are differences between the policies used in the synchronisation process. The policy conflict resolution process will then be responsible for reconciling the conflict using a set of user defined higher level policies. This method is based on the assumption that there is always a single set of higher level policies that are used to control every space in a particular working environment. These policies are used to resolve conflicts that cannot be resolved by user defined lower level policies.

Apart from this method, there is another way to control policy conflict between devices. A set of default policies or policy can be used to suggest how a lower level policy is defined, which could be another style of conflict resolution to be explored. Instead of letting a user write every policy to be used in a local space, an application designer could attach a set of default policies to an application. These policies would be suggested policies to be used in the application but could be overwritten by a user.

This suggested policies method represents another choice between the single policy set and multiple policy sets approach, giving more flexibility than a single policy set environment and more chance of avoiding policy conflict than the multiple policy sets environment. Processing a synchronisation that does not have a policy conflict costs a lot less than a synchronisation with a conflict.

A suggested default policy such as defining that every device requires a policy that selects a tuple from a device that is of higher class can be attached to a data sharing application. Some of the application users may not change the suggested policy and data synchronisations between them will not cause a conflict. Moreover, this suggested policy can be used by a user who modifies it to prepare for a conflict to happen between his or her modified policy and users of the suggested policy. For example, a user who

overwrites the suggested policy should prepare a higher level policy that could resolve the conflict.

9.2.4 Policy and System Compatibility

Since the Ponder language can be used to express policies that cannot be supported by the middleware, one of the most important future work items is how to limit the expressive power so the user will not define a policy element that may cause an adverse effect to the system. Currently, there is no way of checking if a policy is compatible with the system. The current policy compilation process relies on the Ponder policy compiler and its Java code generator to transform from a policy file to a policy object file.

In a simple example, a policy action that has not yet been incorporated into the middleware will cause an incorrect synchronisation process. Since an action is simply a name in the Ponder language compiler's view, it is perfectly correct to define it in a policy. However, if the action has not been defined in the middleware, a synchronisation process controlled by policies using the action will not do anything because the middleware cannot understand the action. The same will happen if a user tries to write other policy types such as a refrain policy or an authorisation policy. The synchronisation system is only written to support synchronisation policies expressed as obligations.

It is possible to modify the compiler and the code generator to add this checking process. However, the process has to take into account the fact that different devices may have different sets of policy elements. Therefore, the checking process needs to contain some levels of interaction with the middleware so that it can check with the system at compile time whether each element in a policy is supported by the local system.

In this way, a user will be notified before his or her policy is used in a synchronisation process. However, this limits the utility of a policy to the local system. A user writing a policy for one device may find it does not work correctly on another device that supports difference sets of policy elements. With this compile time checking, a policy that is manually transferred by a user to another device may not be checked there before it is loaded into the middleware.

Another method is to let the system check every policy object just before the policy is loaded by the policy engine. The process can then notify a user about any policy that is not supported by the system when it is installed. However, this checking process may require the middleware to be in operation in order to do run-time checking and may need the synchronisation engine to stop processing other tasks during the checking process.

This problem is closely related to the problem in 9.2.6. A problem still occurs even if a local device can completely support local policies when the policies are transferred with a tuple to remote devices. A way to transfer extra definitions is required in order to process the synchronisation correctly on a remote device.

9.2.5 Exploring Other Kinds of Synchronisation Problems

Currently, the project concentrates on synchronisation processes where each tuple is seen as a separate piece of information. For example, a tuple that represents a room in a room booking application does not have any direct relationship with the tuple representing another room. Using information in this way makes the synchronisation process simpler because an effect of one synchronisation process is contained within that process. Any action done on a tuple will not have a direct effect on another tuple.

However, in real life situations, a number of applications have pieces of information that are associated with other information. For example, a tuple that contains a result calculated from information in another tuple will be affected if a synchronisation process makes the information in that tuple change. If there is a disconnection before the result tuple is synchronised there will be data inconsistency in the system since the result tuple will not reflect information from its related tuples.

With this kind of association between tuples, a synchronisation process has to resolve or at least notify a user of the inconsistency between the information in the two tuples. Another way is for the system to make sure that any associated tuples are always synchronised in the same connection period.

A further development of the middleware synchronisation process is needed in order to support this kind of application. This requires more user involvement in preparing information for the system such as notifying the middleware how a tuple is related to other tuples and what the middleware should do if it needs to perform an action to one of the tuples.

A new tuple inconsistency checking and resolution process has to be carried out after a synchronisation process. The simplest way could be to put synchronisation processes of associated tuples into a transaction. In this case, if there is a disruption before the whole synchronisation transaction can be finished, the entire synchronisation process will be rolled back. This process prevent any inconsistencies between associated data from occurring in the first place.

However, implementing the system this way reduces the chance that a synchronisation process can be done in a connection window, especially in an environment where the connection duration is short. To make the process efficient, this may also involve a change in the middleware synchronisation operation, which currently relies on the sequence number of a tuple. It will take a long time to finish a transaction if one associated tuple's sequence number is a lot lower than another tuple since the middleware processes the tuples in a sequential order which heightens the chance that it will be terminated because of a disconnection.

9.2.6 Supporting Differences in Context Information

The modification of the system to allow easier addition of new context information will entail more issues. Allowing adding of context information gives more flexibility for a user to create a new synchronisation policy but it makes the synchronisation process more complex because of the presence or

absence of context information on synchronising devices.

If an individual user can add new context information to the system, there will be a situation where synchronisation occurs between systems with different policies and their policies rely on different types of context information. Since a synchronisation process only requires comparison between decisions made from two synchronising devices, there is no need for a device to understand policies and context information written to be used in another device. Therefore, the problem of extra context information definition will not happen between originators of a particular policy.

However, if a tuple has a higher priority and is propagated to another device, policies and context information associated with the tuple have to be attached to it and be propagated together with the tuple in order to maintain the tuple priority (see 6.3.4.1). In this case, a device that obtains the tuple and accompanied information may not understand the policies that use the extra context information. Since the policy engine in the device cannot interpret the new context information, the synchronisation process of the tuple will not be done correctly.

Apart from using the extra information in synchronisation, a device receiving the extra context information from a remote device will have to forward the information to another remote device together with the tuple with which the information is associated. To do this, the device has to register that the tuple has the extra context information that is not normally used locally and must remember to forward the information.

Even though there are a number of features that can be added into the middleware, the current system shows that policy can be used to control a tuple synchronisation process and the middleware can be used to provide the process with a reasonable performance.

Bibliography

[AD76] Peter A. Alsberg and John D. Day. A principle for resilient sharing of distributed resources. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, pages 562–570, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[AKR02] Geoffrey C. Arnold, Gregory M. Kapfhammer, and Robert S. Roos. Implementation and analysis of a JavaSpace supported by a relational database. In *Proceedings of the 8th International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, June 2002.

[And06] A. Anderson. Policies in the alphabet soup. in 7th IEEE International Workshop on Policies for Distributed Systems and Networks. Keynote, June 2006.

[Aur] Project Aura: Distraction-free ubiquitous computing. `http://www.cs.cmu.edu/~aura/publications.html`.

[AVG+99] C. Alaettinoglu, C. Villamizar, E. Gerich, D. Kessens, D. Meyer, T. Bates, D. Karrenberg, and M. Terpstra. Routing policy specification language (RPSL). RFC 2622, , United States, 1999.

[BBB+05] Stefan Batres, Ruslan Bilorusets, Don Box, Luis Felipe Cabrera, Derek Collison, Donald Ferguson, Christopher Ferris, Tom Freund, Mary Ann Hondo, John Ibbotson, Lei Jin, Chris Kaler, David Langworthy, Amelia Lewis, Rodney Limprecht, Steve Lucco, Don Mullen, Anthony Nadalin, Mark Nottingham, David Orchard, Shivajee Samdarshi, John Shewchuk, and Tony Storey. Web services reliable messaging policy assertion (WS-RM Policy), February 2005.

[BBC+06a] Siddharth. Bajaj, Don Box, Dave Chappell, Francisco Curbera, Glen Daniels, Phillip Hallam-Baker, Maryann Hondo, Chris Kaler, Dave Langworthy, Anthony Nadalin, Nataraj Nagaratnam, Hemma Prafullchandra, Claus von Riegen, Daniel Roth, Jeffrey Schlimmer, Chris Sharp, John Shewchuk, Asir Vedamuthu, Umit Yalcinalp, and David

Orchard. Web services policy 1.2 - framework (WS-Policy). http://www.w3.org/Submission/WS-Policy/, 2006.

[BBC+06b] Siddharth Bajaj, Don Box, Dave Chappell, Francisco Curbera, Glen Daniels, Phillip Hallam-Baker, Maryann Hondo, Chris Kaler, Hiroshi Maruyama, Anthony Nadalin, David Orchard, Hemma Prafullchandra, Chris von Riegen, Daniel Roth, Jeffrey Schlimmer, Chris Sharp, John Shewchuk, Asir Vedamuthu, and Umit Yalcinalp. Web services policy 1.2 - attachment (WS-PolicyAttachment), March 2006.

[BCGL86] Robert Bjornson, Nicholas Carriero, David Gelernter, and Jerry Leichter. Linda in adolescence. In *EW 2: Proceedings of the 2nd workshop on Making distributed systems work*, pages 1–4, New York, NY, USA, 1986. ACM Press.

[BCRP98] Gordon S. Blair, G. Coulson, P. Robin, and M. Papathomas. An architecture for next generation middleware. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, London, 1998. Springer-Verlag.

[BDFW97] G. S. Blair, N. Davies, A. Friday, and S. P. Wade. "Quality of service support in a mobile environment: An approach based on tuple spaces". In *Proceedings 5th International Workshop on Quality of Service (IWQOS'97)*, pages 37–48, Columbia University, New York, USA, May 1997. IFIP.

[BEA06] BEA Systems. BEA Tuxedo. http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/tux/, 2006.

[Ber90] B. Berliner. CVS II: Parallelizing software development. In *Proceedings of the USENIX Winter 1990 Technical Conference*, pages 341–352, Berkeley, CA, 1990. USENIX Association.

[Ber96] Philip A. Bernstein. Middleware: a model for distributed system services. *Commun. ACM*, 39(2):86–98, 1996.

[BI03a] M. Boulkenafed and V. Issarny. AdHocFS: Sharing files in WLANS. In *Proceeding of the 2nd IEEE International Symposium on Network Computing and Applications*, Cambridge, MA, USA, April 2003.

[BI03b] M. Boulkenafed and V. Issarny. A middleware service for mobile ad hoc data sharing, enhancing data availability. In *Proceeding of the 4th ACM/IFIP/USENIX International Middleware Conference*, Rio de Janeiro, Brazil, June 2003.

[Bis01] Chatschik Bisdikian. An overview of the Bluetooth wireless technology. *IEEE Communications Magazine*, 39:86–94, December 2001.

[BL02] P. Basu and T.D.C. Little. Networked parking spaces: architecture and applications. In *IEEE 56th Vehicular Technology Conference, VTC 2002-Fall.*, volume 2, pages 1153–1157, 2002.

[Bla01] M. Brian Blake. Rule-driven coordination agents: A self-configurable agent architecture for distributed control. In *ISADS '01: Proceedings of the Fifth International Symposium on Autonomous Decentralized Systems*, page 271, Washington, DC, USA, 2001. IEEE Computer Society.

[BLK00] Randeep Bhatia, Jorge Lobo, and Madhur Kohli. Policy evaluation for network management. In *INFOCOM (3)*, pages 1107–1116, 2000.

[BM02a] Jean Bacon and Ken Moody. Toward open, secure, widely distributed services. *Commun. ACM*, 45(6):59–64, 2002.

[BM02b] A. Belokosztolszki and K. Moody. Meta-policies for distributed role-based access control systems. In *POLICY '02: Proceedings of the 3rd International Workshop on Policies for Distributed Systems and Networks (POLICY'02)*, page 106, Washington, DC, USA, 2002. IEEE Computer Society.

[BNK89] Uwe M. Borghoff and Kristof Nast-Kolb. Distributed systems: A comprehensive survey. Technical Report TUM-I8909, Postfach 20 24 20, D-8000 München 2, Germany, 1989.

[Bra98] Peter J. Braam. The Coda distributed file system. *Linux J.*, 1998(50es):6, 1998.

[BW02] Philip Bishop and Nigel Warren. *JavaSpace in Practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[CCL03] Imrich Chlamtac, Marco Conti, and Jennifer J. Liu. Mobile ad hoc networking: imperatives and challenges. *Ad Hoc Networks*, 1(1):13–64, July 2003.

[CdOVV01] Bogdan Carbunar, Marco Tulio de Oliveira Valente, and Jan Vitek. Lime revisited: Reverse engineering an agent communication model. *Electr. Notes Theor. Comput. Sci.*, 54, 2001.

[CEM03] Licia Capra, Wolfgang Emmerich, and Cecilia Mascolo. CARISMA: Context-aware reflective middleware system for mobile applications. *IEEE Transactions on Software Engineering*, 29(10):929–945, 2003.

[CES04] D. Culler, D. Estrin, and M. Srivastava. Guest editors' introduction: Overview of sensor networks. *Computer*, 37(8):41–49, 2004.

[CG86] Nicholas Carriero and David Gelernter. The S/Net's Linda kernel. *ACM Trans. Comput. Syst.*, 4(2):110–129, 1986.

[CG88] Nicholas Carriero and David Gelernter. Applications experience with Linda. In *PPEALS '88: Proceedings of the ACM/SIGPLAN conference on Parallel programming: experience with applications, languages and systems*, pages 173–187, New York, NY, USA, 1988. ACM Press.

[CGG+05] C. Curino, M. Giani, M. Giorgetta, A. Giusti, A. L. Murphy, and G. P. Picco. TinyLIME: bridging mobile and sensor networks through middleware. In *Pervasive Computing and Communications, 2005. PerCom 2005. Third IEEE International Conference on*, pages 61–72, 2005.

[CGL86] Nicholas Carriero, David Gelernter, and Jerrold Leichter. Distributed data structures in Linda. In *POPL '86: Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 236–242, New York, NY, USA, 1986. ACM Press.

[CMC99] M. Scott Corson, Joseph P. Macker, and Gregory H. Cirincione. Internet-based mobile ad hoc networking. *IEEE Internet Computing*, 3(4):63–70, 1999.

[COZ99] Marco Cremonini, Andrea Omicini, and Franco Zambonelli. The HiMAT model for mobile agent applications. In *PODC '99: Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, page 272, New York, NY, USA, 1999. ACM Press.

[CVV04] Bogdan Carbunar, Marco Tulio Valente, and Jan Vitek. Coordination and mobility in CoreLime. *Mathematical. Structures in Comp. Sci.*, 14(3):397–419, 2004.

[Dam01] N. Damianou. Ponder: Runtime object model policies as Java object. `http://www-dse.doc.ic.ac.uk/Research/policies/ponder/PoliciesAsJavaObjectsV1.0.pdf`, January 2001.

[Dam02] N. Damianou. A policy framework for management of distributed systems, Phd. thesis, 2002.

[Dav84] Susan B. Davidson. Optimism and consistency in partitioned distributed database systems. *ACM Trans. Database Syst.*, 9(3):456–481, 1984.

[DDL+02] N. Damianou, N. Dulay, E. Lupu, M. Sloman, and T. Tonouchi. Tools for domain-based policy management of distributed systems. In *Proc. NOMS2002: IEEE/IFP Network Operations and Management Symposium, Florence, Italy, 15-19 Apr. 2002.*, 2002.

[DDLS00] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. Ponder: A language for specifying security and management policies for distributed systems. Technical Report DoC 2000/1, October 2000.

[DDLS01a] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder Policy Specification Language. In *POLICY '01: Proceedings of the International Workshop on Policies for Distributed Systems and Networks*, pages 18–38, London, UK, 2001. Springer-Verlag.

[DDLS01b] N. Dulay, N. Damianou, E. Lupu, and M. Sloman. A policy language for the management of distributed agents. In *Proc. IEEE/IFIP International Symposium on Integrated Network Management (IM2001), Seattle*, pages 84–100. IEEE Press, May 2001.

[DFWB98] N. Davies, A. Friday, Stephen P. Wade, and G. S. Blair. L2imbo: a distributed systems platform for mobile computing. *Mob. Netw. Appl.*, 3(2):143–156, 1998.

[DLGHB+05] Giovanni Della-Libera, Martin Gudgin, Phillip Hallam-Baker, Maryann Hondo, Hans Granqvist, Chris Kaler, Hiroshi Maruyama, Michael McIntosh, Anthony Nadalin, Nataraj Nagaratnam, Rob Philpott, Hemma Prafullchandra, John Shewchuk, Doug Walter, and Riaz Zolfonoon. Web services security policy language (WS-SecurityPolicy), July 2005.

[DNO98] Enrico Denti, Antonio Natali, and Andrea Omicini. On the expressive power of a language for programming coordination media. In *SAC '98: Proceedings of the 1998 ACM symposium on Applied Computing*, pages 169–177, New York, NY, USA, 1998. ACM Press.

[DPS+94] A. J. Demers, K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and B. Welch. The Bayou architecture: Support for data sharing among mobile users. In *Proceedings IEEE Workshop on Mobile Computing Systems & Applications*, pages 2–7, Santa Cruz, California, 8-9 1994.

[DVLLX02] J. De Vriendt, P. Laine, C. Lerouge, and Xiaofeng Xu. Mobile network evolution: a revolution on the move. *IEEE Communications Magazine*, 40(4):104–111, April 2002.

[DWFB97] N. Davies, S. P. Wade, A. Friday, and G. S. Blair. Limbo: a tuple space based platform for adaptive mobile applications. In *ICODP/ICDP '97: Proceedings of the IFIP/IEEE international conference on Open distributed processing and distributed platforms*, pages 291–302, London, UK, UK, 1997. Chapman & Hall, Ltd.

[Emm00] Wolfgang Emmerich. Software engineering and middleware: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 117–129, New York, NY, USA, 2000. ACM Press.

[EMP+97] W. Keith Edwards, Elizabeth D. Mynatt, Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, and Marvin M. Theimer. Designing and implementing asynchronous collaborative

applications with Bayou. In *UIST '97: Proceedings of the 10th annual ACM symposium on User interface software and technology*, pages 119–128, New York, NY, USA, 1997. ACM Press.

[FAH99] Eric Freeman, Ken Arnold, and Susanne Hupfer. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley Longman Ltd., Essex, UK, UK, 1999.

[FH99] Eric Freeman and Susanne Hupfer. Make room for JavaSpaces, part 1: Ease the development of distributed apps with JavaSpaces. `http://www.javaworld.com/javaworld/jw-11-1999/jw-11-jiniology.html`, 1999.

[FJL00] Magnus Frodigh, Per Johansson, and Peter Larsson. Wireless ad hoc networking: The art of networking without a network. *Ericsson Review*, 4, 2000.

[Fog99] Karl Franz Fogel. *Open Source Development with CVS*. Coriolis Group Books, Scottsdale, AZ, USA, 1999.

[GB98] H. Geffner and B. Bonet. High-level planning and control with incomplete information using POMDP's. In *Proceeding of AIPS'98 Work. on Integrating Planning, Scheduling and Execution in Dynamic and Uncertain Environments*, 1998.

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

[GHM+90] Richard G. Guy, John S. Heidemann, Wai Mak, Thomas W. Page, Jr., Gerald J. Popek, and Dieter Rothmeir. Implementation of the Ficus Replicated File System. In *Proceedings of the Summer 1990 USENIX Conference*, pages 63–71, June 1990.

[Gif79] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles (SOSP)*, pages 150–162, 1979.

[Gig06] GigaSpaces Technologies Ltd. GigaSpaces platform. `http://www.gigaspaces.com/`, 2006.

[GK03] Abdulbaset Gaddah and Thomas Kunz. A survey of middleware paradigms for mobile computing. Technical Report SCE-03-16, Carleton University Systems and Computing Engineering, July 2003.

[GL93] Michael Gelfond and Vladimir Lifschitz. Representing action and change by logic programs. *Journal of Logic Programming*, 17(2/3 & 4):301–321, 1993.

[Gro06] Object Management Group. CORBA basics. `http://www.omg.org/gettingstarted/corbafaq.htm`, 2006.

[GS96] Len Gilman and Richard Schreiber. *Distributed Computing with IBM MQSeries*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[GSSS02] David Garlan, Daniel Siewiorek, Asim Smailagic, and Peter Steenkiste. Project Aura: Toward distraction-free pervasive computing. *IEEE Pervasive computing*, 1(2):22–31, April–June 2002.

[Hal01] Steven L. Halter. *JavaSpaces Example by Example*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[Her90] M. Herlihy. Apologizing versus asking permission: optimistic concurrency control for abstract data types. *ACM Trans. Database Syst.*, 15(1):96–124, 1990.

[HHB99] A. J. Herbert, R. Hayton, and M. Bursell. Mobile Java objects. *BT Technology Journal*, 17(2):115–125, 1999.

[HJ01] K. A. Hawick and H. A. James. Dynamic cluster configuration and management using JavaSpaces. In *CLUSTER '01: Proceedings of the 3rd IEEE International Conference on Cluster Computing*, page 145, Washington, DC, USA, 2001. IEEE Computer Society.

[HKM+88] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6(1):51–81, 1988.

[HR02] Radu Handorean and Gruia-Catalin Roman. Service provision in ad hoc networks. In *COORDINATION '02: Proceedings of the 5th International Conference on Coordination Models and Languages*, pages 207–219, London, UK, 2002. Springer-Verlag.

[IBM] IBM Almaden Research. TSpace FAQS. `http://www.almaden.ibm.com/cs/TSpaces/faq.html`.

[JL06] Vorapol Jittamas and Peter F. Linington. Using a policy language to control tuple-space synchronization in a mobile environment. In *POLICY '06: Proceedings of the Seventh IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'06)*, pages 239–242, Washington, DC, USA, 2006. IEEE Computer Society.

[JM87] Sushil Jajodia and David Mutchler. Dynamic voting. In *SIGMOD '87: Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 227–238, New York, NY, USA, 1987. ACM Press.

[JM90] Sushil. Jajodia and David Mutchler. Dynamic voting algorithms for maintaining the consistency of a replicated database. *ACM Trans. Database Syst.*, 15(2):230–280, 1990.

[Kag02] Lalana Kagal. Rei: A policy language for the me-centric project. Technical Report HPL-2002-270, 2002.

[Kap02a] Steve Kapp. 802.11: Leaving the wire behind. *IEEE Internet Computing*, 6(1):82–85, 2002.

[Kap02b] Steve Kapp. 802.11a: More bandwidth without the wires. *IEEE Internet Computing*, 6(4):75–79, 2002.

[KF06] Lalana Kagal and Tim Finin. Modeling conversation policies using permissions and obligations. *Journal of Autonomous Agents and Multi-Agent Systems*, December 2006. (to appear 2006).

[KFJ03a] Lalana Kagal, Tim Finin, and Anupam Joshi. A policy language for a pervasive computing environment. In *POLICY '03: Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks*, page 63, Washington, DC, USA, 2003. IEEE Computer Society.

[KFJ03b] Lalana Kagal, Tim W. Finin, and Anupam Joshi. A policy based approach to security for the semantic web. In Dieter Fensel, Katia P. Sycara, and John Mylopoulos, editors, *International Semantic Web Conference*, volume 2870 of *Lecture Notes in Computer Science*, pages 402–418. Springer, 2003.

[KRB91] Gregor Kiczales, Jim d. Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, 1991.

[KS92] James J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Trans. Comput. Syst.*, 10(1):3–25, 1992.

[KS93] Puneet Kumar and M. Satyanarayanan. Supporting application-specific resolution in an optimistically replicated file system. In *Workshop on Workstation Operating Systems*, pages 66–70, 1993.

[KW04] Jeffrey O. Kephart and William E. Walsh. An artificial intelligence perspective on autonomic computing policies. In *POLICY '04: Proceedings of the Fifth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'04)*, page 3, Washington, DC, USA, 2004. IEEE Computer Society.

[LBN99] Jorge Lobo, Randeep Bhatia, and Shamim Naqvi. A policy description language. In *AAAI '99/IAAI '99: Proceedings of the sixteenth national conference on Artificial intelligence and the eleventh Innovative applications of artificial intelligence conference innovative applications of artificial intelligence*, pages 291–298, Menlo Park, CA, USA, 1999. American Association for Artificial Intelligence.

[Led98] Parcal Ledru. JSpace: implementation of a Linda system in Java. *SIGPLAN Not.*, 33(8):48–50, 1998.

[Lin06a] Peter F. Linington. Policy specification: Meeting changing requirements without breaking the system design contract. In Joao Paulo A. Almeida, Peter F. Linington, Akira Tanaka, and Bryan Wood, editors, *Workshop on ODP for Enterprise Computing (WODPEC06)*. IEEE Digital Library, October 2006.

[Lin06b] Peter F. Linington. Private communication, 2006.

[Lin07] Peter F. Linington. Private communication, 2007.

[LLS99] Yui-Wah Lee, Kwong-Sak Leung, and M. Satyanarayanan. Operation-based update propagation in a mobile file system. In *USENIX Annual Technical Conference, General Track*, pages 43–56. USENIX, 1999.

[LS99] Emil C. Lupu and Morris Sloman. Conflicts in policy-based distributed systems management. *IEEE Trans. Softw. Eng.*, 25(6):852–869, 1999.

[MESW01] B. Moore, E. Ellesson, J. Strassner, and A. Westerinen. Policy core information model – version 1 specification. RFC 3060, , United States, 2001.

[Mic06] Microsoft Corporation. COM: Component object model technologies. `http://www.microsoft.com/com/default.mspx`, 2006.

[Moo03] B. Moore. Policy core information model (PCIM) extensions. RFC 3460, , United States, 2003.

[MPR06] Amy L. Murphy, Gian Pietro Picco, and Gruia-Catalin Roman. LIME: A coordination model and middleware supporting mobility of hosts and agents. *ACM Trans. Softw. Eng. Methodol.*, 15(3):279–328, 2006.

[MR91] K. McCloghrie and M. T. Rose. Management information base for network management of TCP/IP-based internets: MIB-II. RFC 1213, , United States, 1991.

[MR98] Peter J. McCann and Gruia-Catalin Roman. Compositional programming abstractions for mobile computing. *IEEE Transactions on Software Engineering*, 24(2):97–110, 1998.

[MT03] Ronaldo Menezes and Robert Tolksdorf. A new approach to scalable Linda-systems based on swarms. In *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*, pages 375–379, New York, NY, USA, 2003. ACM Press.

[MW82] Toshimi Minoura and Gio Wiederhold. Resilient extended true-copy token scheme for a distributed database system. *IEEE Trans. Software Eng.*, 8(3):173–189, 1982.

[MZL02] Marco Mamei, Franco Zambonelli, and Letizia Leonardi. Programming context-aware pervasive computing applications with TOTA. Technical Report DISMI-2002-23, University of Modena and Reggio Emilia, August 2002.

[MZL03a] Marco Mamei, Franco Zambonelli, and Letizia Leonardi. Programming coordinated motion patterns with the TOTA middleware. In *Proceedings of 9th International Euro-Par Conference on Parallel Processing (Euro-Par 2003), Klagenfurt, Austria*, pages 1027–1037. Springer, August 2003.

[MZL03b] Marco Mamei, Franco Zambonelli, and Letizia Leonardi. Tuples on the air: A middleware for context-aware computing in dynamic networks. *ICDCSW '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*, page 342, 2003.

[NHdG02] I.G. Niemegeers and S.M. Heemstra de Groot. From personal area networks to personal networks: A user oriented approach. *Wireless Personal Communications*, 22(2):175–186, August 2002.

[NHdG03] I.G. Niemegeers and S.M. Heemstra de Groot. Research issues in ad-hoc distributed personal networking. *Wireless Personal Communications*, 26(2-3):149–167, September 2003.

[NS94] Brian D. Noble and M. Satyanarayanan. An empirical study of a highly available file system. In *SIGMETRICS '94: Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 138–149, New York, NY, USA, 1994. ACM Press.

[NS99] Brian D. Noble and M. Satyanarayanan. Experience with adaptive mobile applications in Odyssey. *Mobile Network and Application*, 4(4):245–254, 1999.

[NSL00] K.J. Negus, A.P. Stephens, and J. Lansford. HomeRF: wireless networking for the connected home. *IEEE Personal Communications*, 7(1):20–27, February 2000.

[NZ01] Michael S. Noble and Stoyanka Zlateva. Scientific computation with JavaSpaces. *Lecture Notes in Computer Science*, 2110:657–??, 2001.

[Obj97] Object Management Group (OMG). The object constraint language specification 1.1, September 1997.

[OR03] Andrea Omicini and Alessandro Ricci. Reasoning about organisation: Shaping the infrastructure. *AI*IA Notizie*, XVI(2):7–16, June 2003.

[ORVR04] Andrea Omicini, Alessandro Ricci, Mirko Viroli, and Giovanni Rimassa. Integrating objective & subjective coordination in multi-agent systems. In *SAC '04: Proceedings of*

the 2004 ACM symposium on Applied computing, pages 449–455, New York, NY, USA, 2004. ACM Press.

[OZ98] Andrea Omicini and Franco Zambonelli. TuCSoN: a coordination model for mobile information agents. In David G. Schwartz, Monica Divitini, and Terje Brasethvik, editors, *1st International Workshop on Innovative Internet Information Systems (IIIS'98)*, pages 177–187, Pisa, Italy, 8–9 June 1998. IDI – NTNU, Trondheim (Norway).

[OZ99] Andrea Omicini and Franco Zambonelli. Tuple centres for the coordination of internet agents. In *SAC '99: Proceedings of the 1999 ACM symposium on Applied computing*, pages 183–190, New York, NY, USA, 1999. ACM Press.

[Per96] C. Perkins. Ip mobility support. RFC 2002, , United States, 1996.

[PGPH90] Gerald J. Popek, Richard G. Guy, Thomas W. Page, Jr., and John S. Heidemann. Replication in Ficus distributed file systems. In *IEEE Computer Society Technical Committee on Operating Systems and Application Environments Newsletter*, volume 4, pages 24–29. IEEE Computer Society, 1990.

[PKKJ04] Anand Patwardhan, Vlad Korolev, Lalana Kagal, and Anupam Joshi. Enforcing policies in pervasive environments. *Mobile and Ubiquitous Systems: Networking and Services*, 00:299–308, 2004.

[PMR99] Gian Pietro Picco, Amy L. Murphy, and Gruia-Catalin Roman. LIME: Linda meets mobility. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 368–377, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[PMR00] Gian Pietro Picco, Amy L. Murphy, and Gruia-Catalin Roman. Developing mobile computing applications with LIME. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 766–769, New York, NY, USA, 2000. ACM Press.

[Pol06] Policy Research Group, Imperial College London. Ponder toolkit. `http://www-dse.doc.ic.ac.uk/Research/policies/pondercompiler.shtml`, 2006.

[Pon07] Ponder2Project. `http://ponder2.net/`, 2007.

[PPGH90] Thomas W. Page, Gerald J. Popek, Richard G. Guy, and John S. Heidemann. The Ficus distributed file system: Replication via stackable layers. Technical Report CSD-900009, Los Angeles, CA (USA), 1990.

[PWC+81] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel. LOCUS a network transparent, high reliability distributed system. In *SOSP '81: Proceedings of the eighth ACM symposium on Operating systems principles*, pages 169–177, New York, NY, USA, 1981. ACM Press.

[RC00] Manuel Roman and Roy H. Campbell. Gaia: enabling active spaces. In *EW 9: Proceedings of the 9th workshop on ACM SIGOPS European workshop*, pages 229–234, New York, NY, USA, 2000. ACM Press.

[RHR+94] Peter Reiher, John S. Heidemann, David Ratner, Gregory Skinner, and Gerald J. Popek. Resolving file conflicts in the Ficus file system. In *Summer USENIX Conference Proceedings*, pages 183–195, Boston, MA, June 1994. USENIX.

[RR02] R. Ramanathan and J. Redi. A brief overview of ad hoc networks: challenges and directions. *Communications Magazine, IEEE*, 40(5):20–22, 2002.

[RVO04] Alessandro Ricci, Mirko Viroli, and Andrea Omicini. Agent coordination context: From theory to practice. In Robert Trappl, editor, *Cybernetics and Systems 2004*, volume 2, pages 618–623, Vienna, Austria, 13–16 April 2004. Austrian Society for Cybernetic Studies. 17th European Meeting on Cybernetics and Systems Research (EMCSR 2004), 4th International Symposium "From Agent Theory to Theory Implementation (AT2AI-4)". Proceedings.

[Sat90] Mahadev Satyanarayanan. Scalable, secure, and highly available distributed file access. *Computer*, 23(5):9–18, 20–21, 1990.

[Sat96] M. Satyanarayanan. Mobile information access. *IEEE Personal Communications*, 3(1), 1996.

[Sat01] M. Satyanarayanan. Pervasive computing: vision and challenges. *Personal Communications, IEEE [see also IEEE Wireless Communications]*, 8(4):10–17, 2001.

[SAW94] Bill Schilit, Norman Adams, and Roy Want. Context-aware computing applications. In *IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, US, 1994.

[SFM+96] Jon Siegel, Dan Frantz, Hal Mirsky, Raghu Hudli, Peter de Jong, Alan Klein, Brent Wilkins, Alex Thomas, Wilf Coles, Sean Baker, and Maurice Balick. *COBRA fundamentals and programming*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[SG02] J. Pedro Sousa and David Garlan. Aura: an architectural framework for user mobility in ubiquitous computing environments. In *WICSA 3: Proceedings of the IFIP 17th World*

Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture, pages 29–43, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V.

[Sie99] Jon Siegel. *CORBA 3 Fundamentals and Programming with Cdrom*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[SKK+90] M. Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, April 1990.

[SKS87] M. Satyanarayanan, J. J. Kistler, and E. H. Siegel. Coda: A resilient distributed file system. In *IEEE Workshop on Workstation Operating Systems*, November 1987.

[SS05] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, 2005.

[Sun98] Sun Microsystems. JavaSpace specification. `http://java.sun.com/products/jini/specs`, 1998. Sun Microsystems. JavaSpace Specification, March 1998.

[Sun99] Sun Microsystems. Jini technology architectural overview. `http://www.sun.com/software/jini/whitepapers/architecture.pdf`, 1999.

[Sun02] Sun Microsystems. Default policy implementation and policy file syntax. `http://java.sun.com/j2se/1.4.2/docs/guide/security/PolicyFiles.html`, April 2002.

[Sun03] Sun Microsystems. Jini technology starter kit overview v.2.0_001. As a document for jini2.0_001 starter kit package, 2003.

[Sun06a] Sun Microsystems. Java message service (JMS). `http://java.sun.com/products/jms/`, 2006.

[Sun06b] Sun Microsystems. Java remote method invocation (Java RMI). `http://java.sun.com/products/jdk/rmi/`, 2006.

[Swe06] Swedish Institute of Computer Science. SICStus Prolog. http://www.sics.se/isl/sicstuswww/site/index.html, 2006.

[TDP+94] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent W. Welch. Session guarantees for weakly consistent replicated data. In *PDIS '94: Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, pages 140–149, Washington, DC, USA, 1994. IEEE Computer Society.

[TN89] T. Tang and N. Natarajan. A static pessimistic scheme for handling replicated databases. In *SIGMOD '89: Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pages 389–398, New York, NY, USA, 1989. ACM Press.

[TNO02] Y. M. Teo, Y. K. Ng, and B. S. S. Onggo. Conservative simulation using distributed-shared memory. In *PADS '02: Proceedings of the sixteenth workshop on Parallel and distributed simulation*, pages 1–7, Washington, DC, USA, 2002. IEEE Computer Society.

[TPST98] Douglas B. Terry, Karin Petersen, Mike Spreitzer, and Marvin Theimer. The case for non-transparent replication: Examples from Bayou. *IEEE Data Eng. Bull.*, 21(4):12–20, 1998.

[TTP+95] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP-15), Copper Mountain Resort, Colorado*, 1995.

[UMB06] UMBC ebiquity research group. Rei : A policy specification language. `http://ebiquity.umbc.edu/project/html/id/34/Rei-A-Policy-Specification-Language?pub=on#pub`, 2006.

[Uni95] International Telecommunication Union. ITU-T recommendation X.812, security frameworks for open systems: Access control framework. ISO/IEC 10181-3, Geneva, 1995.

[VKP03] E. P. Vasilakopoulou, G. E. Karastergios, and G. D. Papadopoulos. Design and implementation of the HiperLan/2 protocol. *SIGMOBILE Mob. Comput. Commun. Rev.*, 7(2):20–32, 2003.

[VLK00] A. Virmani, J. Lobo, and M. Kohli. Netmon: Network management for the SARAS softswitch. In J. Hong and R. Weihmayer, editors, *IEEE/IFIP Network Operations and Management Symposium, (NOMS2000) Hawaii*, May 2000.

[VZV03] K. Vaxevanakis, Th. Zahariadis, and N. Vogiatzis. A review on wireless home network technologies. *SIGMOBILE Mob. Comput. Commun. Rev.*, 7(2):59–68, 2003.

[Wad99] Stephen Paul Wade. *An Investigation into the use of the Tuple Space Paradigm in Mobile Computing Environments*. PhD thesis, Lancaster University, 1999.

[Wei91] Mark Weiser. The computer for the twenty-first century. *Scientific American*, 265(3):94–104, 1991.

[Wei99] Mark Weiser. Some computer science issues in ubiquitous computing. *SIGMOBILE Mob. Comput. Commun. Rev.*, 3(3):12, 1999.

[Wie94] René Wies. Policies in network and systems management - formal definition and architecture. *Journal of Network and System Management*, 2(1), 1994.

[WMLF98] P. Wyckoff, S.W. McLaughry, T.J. Lehman, and D.A. Ford. TSpace. *IBM System Journal*, 37(3):454, 1998.

[WSH05] S. Waldbusser, J. Saperia, and T. Hongal. Policy based management MIB. RFC 4011, , United States, 2005.

[XTML02] Jason Xie, Rajesh R. Talpade, Anthony McAuley, and Mingyan Liu. AMRoute: Ad hoc multicast routing protocol. *Mobile Networks and Applications*, 7(6):429–439, 2002.

[YPG00] R. Yavatkar, D. Pendarakis, and R. Guerin. A framework for policy-based admission control. RFC 2753, , United States, 2000.