Applicability Of Mobility, a Case Study

Frederik Nyssens
fnyssens@vub.ac.be

September 6, 2001

Abstract

Intelligente autonomiteit en mobiliteit kunnen beschouwd worden als de twee belangrijkste punten in het onderzoek naar multi-agent systemen. Tot op heden werd het gebied van de mobiliteit nog niet goed onderzocht en gebruikt, laat staan dat het gecombineerd werd met autonomiteit. Door het ontwikkelen van deze gebieden, kunnen er potentiële nieuwe, nuttige applicaties ontsaan en verbeterde efficiëntie van bestaande applicaties.

Het doel van deze thesis is aan te tonen dat de combinatie van autonomiteit en mobiliteit bij agents, de veiligheid van de applicaties verbeterd en roaming van gebruikers en applicaties mogelijk maakt. Verder kan mobiliteit en autonomie gebruikt worden om de betrouwbaardheid te verbeteren en automatische load balancing te verkrijgen.

Deze thesis toont deze vier punten aan door middel van een voorbeeld dat dit verduidelijkt.

Abstract

Intelligent autonomy and mobility are the two main points of research in the area of multi-agent systems. Currently, mobility is neither well explored nor utilized, and has not yet been combined with autonomy. By developing these aspects of the field, there is potential for new, useful applications and improved efficiency of existing applications.

The aim of the thesis is to show that the combination of autonomy and mobility in agents enhances the security of applications, enables roaming for applications and users, enhances the reliability of applications and enables automatic load balancing

This dissertation demonstrates these four points by means of cases that illustrates them.

Acknowledgments

I am grateful to my parents that gave me the opportunity to get this far, even if it took more time.

I am very thankful to my friends that let me complain about the workload and supported me during the creation of this dissertation. I specially want to mention Mark Desmaele, who found the time to read this dissertation at the last minute.

And last but not least, Prof. Dr. D'Hondt and his research assistants at the Programming Technology Lab from the Brussels Free University. Especially, Werner Van Belle who encouraged me to finish.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Agents and multi-agent systems are an emerging field in computer science where the emphasis lays on the intelligence and mobility of agents. Up until now mobility is never been used except in data-mining, where agent crawl over the web to filter whatever it encounters. Mobility of agents can be used for other applications than just data-mining.

An agent is a process that is capable of performing autonomous action in an environment. If autonomy is applied to migrating schemes agents can be used to:

- Enhance security of applications and data;
- Add automatic roaming for users and applications;
- Make applications more reliable;
- Add automatic load balancing to applications.

The security of applications can be enhanced by using several agents that migrate. It is quite easy to create one stub-agent, that is used by other agents or users to acquire the data from. This stub-agent creates several agents with the actual data, whenever the stub-agent is contacted for information, this stub-agent will contact its own created agents. These agents migrate themselves as an ongoing process. This way users cannot access nor copy the data agents for other than the intended use, trough the stub-agent. This scheme is applicable to applications as well as protected data.

Nomadic users often use their portable device just for storage. Once in a while a nomadic users connect to the corporate network to exchange data and update their applications. Many of the applications they use require an exchange of information of the places they visit. The use of agents can make this process of exchanging information easier:

- Agents can browse trough each network the nomadic user visits. Once the agents are on the network, there is no need anymore for a connection of the user. The agents will migrate back to the portable device, whenever it comes available and continuous its quest for information without the interference of a user.

- Agents can be used to automatically update and upgrade applications and databases, without human interference;
- Agents can be used to notify the arrival of a nomadic user to the interested parties. The nomadic user can also be notified on the availability of other users where the nomadic user is interested in.

In short this can be summarized as follows:

- Roaming of applications;
- Remote management;
- Roaming of users.

Distributed systems have often to difficult schemes to be reliable. With the use of agents these difficult schemes can be avoided. The reliability of an application is determined by two concepts, persistency and fault-tolerance. An application is considered reliable when it is available at all times or when faults that occur have no impact on the application. A 100% reliable system is not possible with agent systems. But, when no faults occur, an agent based application can be made persistent. It is easy to inform an agent that an agent platform is shutting down. When the agent is notified the agent will migrate to another system that keeps running. This way the agent will remain available.

A fault that can occur in multi-agent systems is the unavailability of certain agents. There are two possible causes that can make an agent, or several agents, unavailable. The least troublesome fault is when the agent is still running but when parts of the network are unavailable trough network faults. The worst problem occurs when an agent actually dies, and looses all its information. The solution to both problems is the same, distribute all information. When an agent dies or is unavailable a new agent can be created an all the information it requires can come from all the surviving, available, agents. When the network-faults are solved there are two agents that do the same task, in this case the agents can be combined to one agent. In this case no information is lost when the agent dies, eg due to a system crash. The amount of information or data that is lost depends on the number of agents that are dead.

Load balancing distributed applications is a combination of hardware and software mechanisms to determine which server will execute which client request. This means that for load balancing each server owns a replica of the objects or databases that are available. Applications written in mobile multi-agent systems consist of several agents that are performing a task. These agents can migrate to wherever they want in that system, agents can be made to migrate when the load on a system is to heavy. A load balancing scheme like presented here is not optimal, but nonetheless better then no load balancing.

This dissertation will demonstrate how to use mobility by the means of one application that uses agent migration to enhances security to application, enables roaming for users and applications, enhances reliability for applications and make applications more load balanced. The application is a simple agenda application. A user keeps his agenda on his portable device. Whenever the user connects to a network that has this agenda application running the agenda of the user is copied and available for the application. The aim of the agenda application is to find other persons that are interested in a meeting. After the fiat of the user, that has reconnected again, an appointment will be made, again using the agenda application that tries to find an optimal time for both parties.

This dissertation starts with an introduction to agents, and agent systems. Because the case studies have a focus on mobile multi-agent systems this chapter has a focus on distributed multi-agent systems, and how agents can work together.

The second chapter takes a look at what is possible with mobile code, and compares the most important possibilities. This includes the client/server model as reference model. The last part of this chapter is an overview of the requirements in mobile multi-agent systems.

Because security is of major importance, I included a chapter on the possible threats for agents and the computing environment. There is also an overview about the possibilities to make agents and mobile multi-agent systems more secure.

The next chapter should give a clear explanation of the problems with application security, roaming, reliability and load balancing. For each of these four points there is also a very small section that explains some of the solutions to these problems.

The design of the agenda application is being explained in chapter 6. The explanation starts with the internal design of the agents to show what methods and data are important for the security, roaming, reliability and load balancing. The last part explains what kind of communication between the agents is important.

The dissertation ends with the mandatory conclusions and future work.

Chapter 2

Agents and Multi-Agent Systems

Distributed computing and distributed artificial intelligence are the two fields in computer science that use agents. Although both fields differ enormously the requirements to be an agent are the same. In this chapter I will explain what agents are how agents are related to their environment and how agents can work together to reach one common goal. I conclude this chapter with the origin mobile multi-agent systems and what requirements mobile agents impose on the systems.

2.1 Agents

The term agent is not yet fully defined [25][37][41]. Each domain that uses the term agent also uses another definition for the agent they use. Nevertheless there is some consensus on what an agent should meet as criteria. The criteria that best describe an agent are found in [41] :

> An *agent* is a computer system that is situated in some *environment*, and that is capable of *autonomous action* in this environment in order to meet its design objectives

According to [37] an agent can be categorized in two domains. First, the agents that are used in distributed artificial intelligence (DAI), these types of agents are often used to simulate the behavior of large populations. The second type of agents are used in distributed computing (DC), instead of having the traditional client server architecture we have data servers and applications that are going to the data. Although DC and DAI differer enormously, they both want autonomous agents in some environment.

Autonomous means "an agent can make a decision without any interference from another system or user". This makes it clear why I do not use the term intelligent agents in the definition. An agent that controls the air conditioner is as much an agent as an agent that has to learn how to cultivate food on some land. This also means that the environment could be anything, ranging from simple to complex environments. The environment for the air conditioner agent is the room temperature and the air conditioner, the environment for the cultivating agent is the land on which it has to work, the climate, the number of seeds the agents have, sources of water ...

2.1.1 Intelligent Agents

It is still not clear when an agent is considered to be an intelligent agent. An intelligent agent possesses three more criteria [41] [42]:

- *reactivity* : an intelligent agent is able to perceive the environment and responds to the changes in that environment;
- *pro-activeness* : an intelligent agent takes the initiative in order to satisfy the design goals;
- *social ability* an intelligent agent is capable of interacting with other agents and possible humans, in order to satisfy the design goal.

Building a pure reactive agent is not hard, neither is a pro-active agent. The combination of those two criteria is hard to accomplish in an agent.

2.2 Multi-Agent Systems

An agent always resides in an environment. In this environment it is possible to have multiple agents trying to work towards their design goal. In this case we are talking about a *multi-agent System*. When the agents start to communicate with each other, then it is called a *society of agents* [41]. Although it is not always true, most multi-agent systems or societies of agents are working distributed.

Although distributed multi-agent systems can be written in such a way that the agents are all running on one machine, there is much to say about distributed solutions [41]. If local calculation can be avoided through communication it is better to use communication. Computers nowadays are close to the physical limits in processing powers and memory, communication is still far from the limits [33]. Sometimes it is also necessary to work distributed because the data is distributed. Some companies wants to keep their data local for competitive reasons, or there might be to much data to keep it in one place.

The environment in a multi-agent system has three characteristics [41]:

- the environment provides an infrastructure for communication (see section 3.3.1 on page 19);
- the environment is typically open and has no centralized design;
- the environment contains agents that are all autonomous but they may be self interested or cooperative.

The two last characteristics have no real impact on what a multi-agent environment is. The criteria for agents outlines that an agent can take autonomous actions upon its environment (see section 2.1 on page 10). This never excluded helping other agents, nor does it exclude that the environment includes all other agents on the system.

Communication is important in multi-agent systems. Agents should be able to request an action or data from another agent, even when the agent is a simple not intelligent agent. Because agents are autonomous in their actions it should be not surprising that a requested action or data could be denied. The communication between agents should be transparent [37]. This enables the programmers not to worry about the agent communication but on the functionality of the agent.

As described in chapter 4 on page 26 an open system with agents is not the synonym of an insecure system. Many authors are claiming that an open system is not secure [13]. Other authors claim that mobile code can be used to implement security systems rather than requiring security [35].

2.3 Agent Societies

Like every normal, decent sized object-oriented program it is normal to have multiple objects. These objects are responsible for certain actions, a certain behavior [2].

Conceptual an agent is the same as an object in object oriented programming [34]. Message can be send to request an action. however there is a huge difference between objects and agents, objects are expected to have a fixed behavior, agent being autonomous in their actions (see section 2.1 on page 10) do not have a fixed behavior.

In the worst-case scenario agents could be working independent and just requiring information or actions from other agents. The behavior of the agents is then what is best suited to reach their design goal. In these multi-agent systems following messages are possible[41]:

- propose an action;
- accept an action;
- reject an action;
- retract an action;
- disagree with a proposed action;
- counter pose an action.

In the cultivation agent example of section 2.1 where there are two farmer agents A and B. A can ask agent B to build an irrigation channel from the water source

of agent B. This is proposing an action.
Agent B can :

- accept the action : building the irrigation channel;
- reject the action : not building the irrigation channel;
- retract an action : Agent B is willing to build the irrigation channel at first and later retract the action by not sending water trough the channel;
- disagree with the proposed action : agent B might be willing to give the water supply but agent A has to build the irrigation channel;
- counter pose this action by proposing the irrigation channel and monthly payments if the irrigation channel is build.

There is no reason why an agent could not react the way objects would react. Autonomy does not mean working only for its own interest. Agents could be working together to reach one common goal. The autonomy here could mean the location where the agents are running or what kind of data agents collect. In this case working with agents is similar to working in object oriented style [19].

Similar to working in object oriented style means just asking for actions. This does not mean that data passing is the same as in object oriented style, eg agents work in their own data space so passing by reference should not be available.

2.4 Mobile Multi-Agent Systems

Mobility in computer science existed since the introduction of laptops. Nowadays there is another form of mobility, *mobile computing* where mobile code moves between devices [4]. Mobile code is actually not enough, there are situations where a mobile programs or agents are preferred [32] (see section 3.1 on page 15 for the distinction between mobile code and agents).

Mobile agent systems used to be compared with client server architectures. While client server architectures allow you to retrieve data and do computations with the data on your local machine, mobile agents work the other way around. A mobile agent goes to the data server, does the computation with the data on the spot and comes back to the owner or goes to other servers to retrieve other data. This is useful in mobile devices because they have a relatively low bandwidth, limited processing power and memory and are not always connected to a network [13].

Moving a mobile agent should be done with *strong mobility*. This requires that the entire state of the running agent, including the execution stack has to be saved. This ensures that when the agent is transported to its new location can continue computing where it stopped. This however does not implicate the necessity of strong mobility to have mobile agents [13].

Strong mobility is one of the many applications of *checkpointing* [27]. Because agents are autonomous, transparent checkpointing at user-level should be used. With non-transparent checkpointing the programmer incorporates the checkpoints to the program. Checkpoints in transparent checkpointing are performed by the program, thus more useful for strong mobility (see also section 3.2.3 on page 19).

Mobile calculi are equal to common process calculi, like λ-calculus [4] and others [8] [38]. Programming languages for mobile computation are therefore possible. Special care has to be taken in case of security (see section 4 on page 26). In the ideal world mobile agents could go to any device connected to the network of devices, but problems arise when agents are not allowed on certain devices. Even when an agent is allowed on a device, there is no way the device can know if the agent has a virus like behavior [13].

Chapter 3

Mobile Multi-Agent Oriented Programming

Mobile multi-agents are a part of the mobile code paradigm. The mobile code paradigm is a solution to the increasing problems that client/server architectures have. This part of the dissertation first gives an overview of the mobile code paradigm and compares the possibilities of the remote code paradigm with remote procedure calls, which is the way client/serves architectures are mostly implemented.

After the explanation of mobile code paradigm and the client/server architecture, the section also explains why weak mobile code is not suitable for mobile multi-agent systems, and why strong mobile technology is better suited.

When agents are used to make an application, a society of agents, the last requirement is communication between agents. I explain what is a good communication language and how agents can communicate with each other.

3.1 The Mobile Code Paradigm

Networks are becoming bigger and better and the computation infrastructure is coming to the physical limits [33]. Besides the traditional client/server architecture (see section 3.2.1 on page 17) mobile code fills the gap to consolidate the explosion of network-centric applications[11].

In the mobile code framework the computational infrastructure is composed of a, possible world-wide, distributed environment with several *computational environments* that support the computations of *executing units*. The executing units may change their computational environment and even the code they execute dynamical[11].

The mobile code paradigm is developed to reduce the deficiencies of the client/server architecture. In the client/server architecture, applications cannot move across different machines, they must run on the machines they reside on.

Other requirements are high network-bandwidths and continuous user-computer interaction[5].
By allowing to move code over the network, applications can interact on the same computer as the resource instead of communicating over the network. Thus reducing network-load. Mobile agents can work on behalf of the users autonomously. This autonomy allows the user to delegate tasks to the agent, which does not require continuous computer interaction[5][24].

Table 3.1 on 16 gives a brief summary of the possibilities used in mobile code. The client/server architecture is included as a reference model. The know-how represents the way of solving the problem, i.e. code or compiled code. The state is the state of the computation, the resources are the input/outputs of the computation, i.e. databases, other agents ... The arrows indicate what moves to where [5][11].

Paradigm		Local Side	Remote Side	Computation takes place at
Client/Server			know-how	remote side
			state	
			resources	
mobile code	Remote Evaluation	know-how →		Remote Side
			state	
			resources	
	Code on Demand	←	know-how	Local Side
		state		
		resources		
	Mobile Agent	know-how →		Remote Side
		state →		
			resources	

Table 3.1: Classification of Mobile Code Paradigms [5][11]

Where in *remote procedure calls* procedures always reside on the remote side in *remote evaluation* on the other hand, the local side sends a procedure to the remote side where it is evaluated and the results are send back to the local site[40][11]. If the results contain references to other distributed resources new remote evaluations can be started [11].

Code on demand is probably the most used technique of the mobile code paradigm. Although highly controversial because of the security threats, Java Applets, Active-X control and Java-Script are embedded in some HTML-pages. Once the HTML-page is loaded into a Java or Java-Script enabled browser this code is executed. This is exactly what is shown in table 3.1 on 16. The code or know-how is transported to the local side, all computations and resources that are being used are on the local site [11][20][40].

Remote evaluation does the opposite of code on demand. In this paradigm a client sends a procedure to a server to execute on its behalf. Once the computation is finished the results came back to the local site [40].

With *mobile agents* the state and the know-how can be moved together. Here each site has some resources that the agent can use. If there are resources on other sites the agent, know-how and state, is migrated to the site where the resource is available. When the computation on that sites is finished, the agent will migrate to another site that has other resources [11][40].

3.2 Mobile Code Technology

For some authors ([10][13][32]) everything that is mobile is an agent. Doing this only contributes to the various misunderstandings towards agent-oriented programming. It is better to classify a mobile code technology using table 3.1 on page 16 because it is more clear to see what is really possible with the technology.

It is possible to make following distinctions in what is possible with todays technology[11]:

- *Remote Procedure Calls* does not provide any mobility. It is only possible to call a procedure.
- *Weakly mobile* technologies provide a mechanism to send code together with some initial data to a remote execution unit where it will be processed.
- *Strong mobile* technologies enable execution units to move their code, data and *state* to other computational units.

3.2.1 Remote Procedure Calls

Remote Procedure Call (RPC) is often confused with client/server architecture[13]. RPC is just a part of the client/server architectures [33]. RPC provides a convenient way to allow a high-level interpretation of low-level system functionality to be used at the application level [7].

RPC or its newer implementation Remote Method Invocation (RMI), is used to access and modify remote state, it is a synchronous blocking invocation of a statically typed interface. CORBA and JINI have been implemented on top of RPC, which alleviates the applications from the RPC semantics [29].

RPC allows an applications to make simple function calls to a well defined interface. What happens is that a stub is called on the client side, this stub does not perform any computations but puts the arguments in a Marshalled representation. This representation is send to the server where the remote procedure is called and processed. The results of the process is returned in the same manner [7].

RPC is overly restrictive, inflexible and inefficient for applications that require some sort of mobility : [29]

- RPC has a tightly coupled bi-directional semantics i.e. a discrete request followed by a discrete reply in the opposite direction. The calling threads are suspended even when no return value is required.
- pure RPC semantics do not allow for the return value to be delivered to a different host or process
- RPC's bi-directional semantics mean that the properties (eg Quality of Service) of messages in either direction cannot be changed independently.
- RPC is not suited for group communication and thus not a good idea to use for collaborative sensor rich environments.
- All remote interface must be known at compile time. This means loosely coupled systems, with different components designed and implemented separately at different points of times requires a lot of work.

3.2.2 Weak Mobile Technology

Weak mobile code is code that is send to a computational unit to be processed there. Once the code reaches its destination it is not possible anymore to move this code. It is clear to see that this is the best technology for remote evaluation and code on demand (see section 3.1 on page 15)

Weak mobile code is best written in higher-order languages such as Scheme. These languages provide support for data abstractions via first-class procedures and support for sophisticated control-abstractions. Because of their generality, procedures and continuations can be used to expresses operations for distributed programming. Using these languages as building blocks for high-level distributed programming avoids the need to define an entirely new language or to introduce ad-hoc primitives to an existing one [14].

Weak mobile code has several advantages over RPC.

- Weak mobile code can be non-blocking and non-bi-directional.
- Weak mobile code and the results it has processed are usually smaller than the amount of data that an average RPC transports over the network.
- It is easier to engineer loosely coupled systems.

On the other hand weak mobile code still has disadvantages:

- Implementing loosely coupled systems is not optimal;
- Group communication is not optimal.

3.2.3 Strong Mobile Technology

An *agent* is a persistent autonomous software component and it is mainly intended to provide a particular service (see section 2.1 on page 10) . Viewed as processes, agents run concurrently on one or more machines and have their own data and computational space. A *mobile agent* is an agent that is capable to migrate between locations [36].

The working of an agent, or any other arbitrary program, is determined by [36][27]:

- A code section which describes the behavior of the agent;
- A data section which contains the environment of the agent;
- One or more structures that describe the computational state of the agent;
- Connections to resources that the agent uses.

Ensuring migration is a relatively easy task. All it takes is some sort o checkpointing. Checkpointing is a mechanism to store the program's data necessary to restart the program (see figure 3.1 on page 20).

The *looping model* [36] transfers all the parts of an agent except the computational state. The only way to migrate an agent is to ensure that the computational state is empty. This makes it impossible for an agent to be completely autonomous in migration, which is to restrictive to make a true mobile multi-agent system.

It is easy to see that the looping model is a form of non-transparent checkpointing. Although non-transparent checkpointing goes a step further. The programmer is responsible for storing and recovering the execution state of the program by the means of primitives build in the programming language [27]. Although these models make it possible to migrate, they are weak mobile code technologies because it still requires a lot of effort to migrate them.

Strong migration supports the agent to migrate at all times, regardless of the computational state. As said in section 2.4 on page 13 some sort transparent checkpointing is best used to make this technology possible.

3.3 Communication Between Agents

3.3.1 Communication Languages

The simple travel booking example shows the need for agent communications. A customer agent comes to a retail travel service. The customer agent wishes to know a particular type of holiday within a price range and for a set of dates. The only problem remains communicating these desires towards the retail agent [25]. In this situation there are two agents that have a need to communicate with each other, the customer agent and the retail agent.

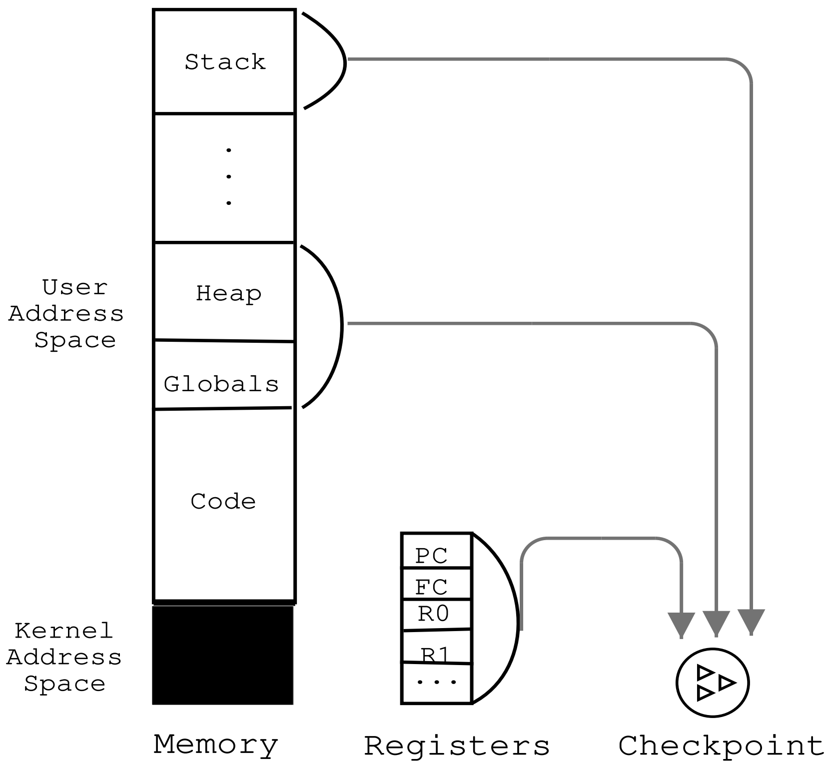

Figure 3.1: Uniprocessor Transparent Checkpointing [27]

In the ideal world both agents would be written in the same programming language. If this would be the case agent communication is not a problem. Provided that the programming language has some form of agent communication.

In the real world it is unlikely that the agents are written in the same programming language. If there is no standardized communication mechanism a customer agent would be bound to only the retail agents that support the communication the customer agent understands.

The requirements for an agent communication language are divided into eight categories [21]:

1. The *form* should be declarative, syntactically simple and readable by humans.
2. The *content* should have a well-defined set of communication primitives that captures most of our intuitions about what constitutes a communication act. The usability of the agent communication language in various systems is not determined by the application (database, object-oriented systems ...).
3. The *Semantics* should be grounded in theory, it should be unambiguous and it should exhibit canonical form.
4. The *implementation* should be efficient, both in terms of speed and bandwidth utilization. The interface should also hide implementation details like the network layers and it should be independent from programming languages.
5. All *connection types* (point-to-point, multi-cast, broadcast) should be available. The languages should provide a rich enough set of communication primitives such that higher order protocols (SMPT, HTTP, FTP, ...) can be built. Moreover these higher order protocols should be independent of the transport mechanism.
6. The communication language is required to work in a distributed, heterogeneous and highly dynamic environment. Thus the communication language should be able to cope with heterogeneity and dynamism. It must support interoperability with other protocols and other languages.
7. The communication language must support *reliable and secure* communications between agents. The should be a way to guarantee authentication of agents.
8. The language should support reasonable mechanisms for identifying and signaling errors and warnings.

Knowledge Query and Manipulation Language (KQML) is a language that is designed to support interaction among intelligent software agents. It was developed by the ARPA Support Knowledge Sharing Effort and independently developed by several research groups. Communication takes place at several levels. The content of a message is only a part of the communication. Locating and engaging agents is part of the process. [21]

Foundation for Intelligent Physical Agents (FIPA) an non-profit organization for producing specifications for open agent interfaces, has created an agent communication language ACRCOL. According to FIPA, KQML has a number of shortcomings. The biggest drawback is that KQML is informally defined so that human interpretation is necessary [25].

A good agent communication language has many needs, some of which are in competition. It is not surprisingly that a communication language addresses many, although not all, of these needs. The inevitable compromise between the theoretical requirements and efficient-easy to use implementations may not result in a communication language that suits everyone [21].

3.3.2 Communication Channels

Good mobile multi-agent systems should make the communication transparent, which means that the underlying network should offer as much as possible while hiding the underlying network. This means also shielding the agent-programmer for worrying about the location of the agent and whether a send message is serialized correctly [37].

Making an mobile multi-agent system like this releaves the programmer from huge burdens that are present in remote procedure calls and code on demand (see section 3.2.1 on page 17) [36]:

- Stubs do not have to be compiled like with remote procedure calls;
- Locations are found automatically and a programmer does not have to worry about where an agent is;
- Sending messages is like sending messages to objects, all serializations are don automatically.

There are two possible ways to send messages to agents. The first one is message based, and works in the same fashion as messages in Object Oriented systems. The other possibility is sending messages for information, and not starting any other process in the other agent.

Both methods are equally expressive and can be implemented in an synchronous or asynchronous way(see section 3.3.3 on page 22).

3.3.3 Communication Forms

Communication is a fundamental concept in computing. It allows two pieces of software to interact and to exchange information. Communication in agent systems is essential because most of the functionality in multi-agent systems is communication [22].

Communication between software has always been between two processes. When the two processes reside on different computers communications also involves latency, reliability and uncertainty. With the use of remote procedure calls this has always been in a synchronous way[29]. Communication between agents is asynchronous and must allow communication between several processes [28].

Synchronous Communication

Synchronous communication can be described as a form of communication agree to synchronize and remain synchronized during the duration of the communication. The pattern of the communication follows a particular order so that each party involved knows when to send data and when to receive data (see figure 3.2 on page 23).

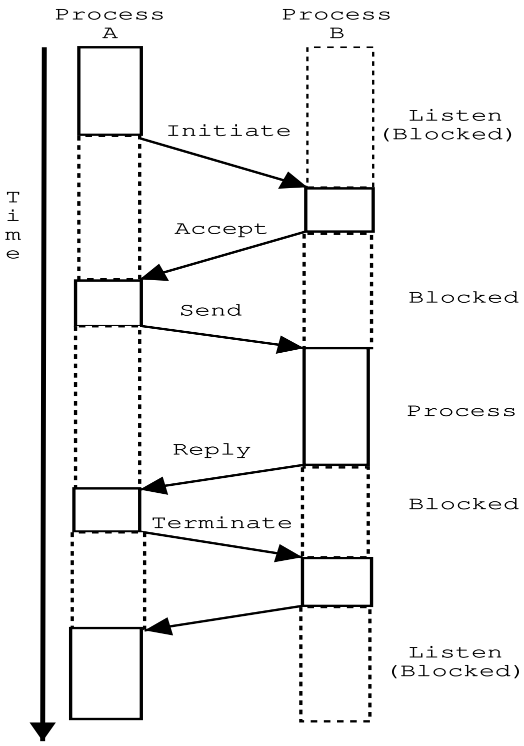

Figure 3.2: Synchronous Communication[22]

Because there is a typical start and end to the communication with communication in between, synchronous communication is typically a connection-oriented communication. Synchronous communication is blocking because when an involved party is waiting for data trough a communication channel it waits until it receives that data. This way of communicating is highly suited for situations in which shared data or state information is being transferred and when it is important that all parties involved are focused on one task [22].

For agents synchronous communication is not suitable:

- Agents are autonomous, thus agents should also be autonomous when to accept messages and when to reply, if there is a reply needed.
- Both parties have to remain on-line from start to end of the communication. This means that agents cannot migrate freely, thus strong mobility is lost.
- The history and state of a communication is spread over all parties involved in a conversation.
- Synchronous is more oriented to a client-server interaction and less to a peer-to-peer interaction. Peer-to-peer interaction allows processes to the take the role of client or server as their needs dictate.

Asynchronous Communication

With asynchronous communication the processes that communicate do not synchronize. As there is no explicit start and no explicit end to the communication, messages can be send at any time [22] (see figure 3.3 on page 24).

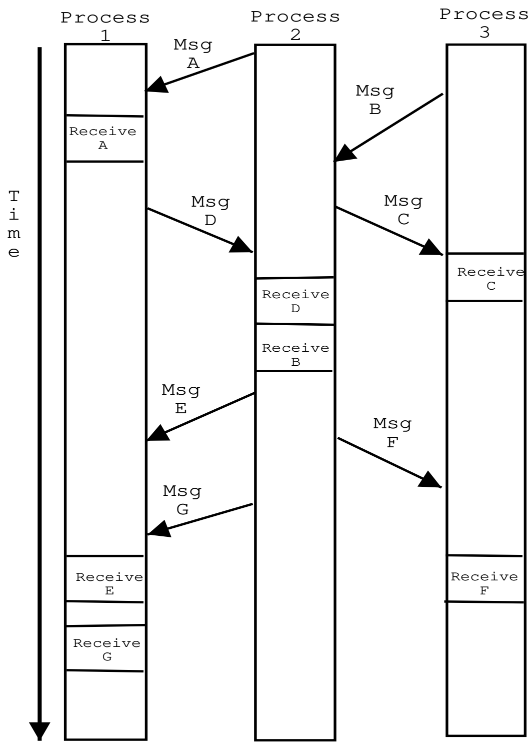

Figure 3.3: Asynchronous Communication[22]

Asynchronous communication is typically connection-less because there is no permanent communication between the two processes. It is non-blocking unless the process block themselves deliberately[22]. Processes that use asynchronous communication require buffering for the received messages, because the processes should be able to receive messages at all time. These buffers have the advantage that the process can browse trough the buffer to select the message it will handle first.

Asynchronous communication models are suitable for communications that can be spread over a larger time-frame or when it is important that a communication process can choose between the received messages.

For agents asynchronous communication is better suited than synchronous communication:

- Asynchronous communication can be used in situations where agents are not always connected to a network like in intermittently connected devices such as laptops and portable digital assistants.
- Latency, delays and network-failures are not a problem in asynchronous communication.
- Agents are not blocked and can choose between messages to process.
- Asynchronous communication does not mean that agents cannot synchronize when needed.
- An agent can act as a client and a server at the same time for different agents, without the need of multi-treading.

Chapter 4

Security in Mobile Multi-Agent Systems

Mobile multi-agent systems are typically implemented using strong mobile technology with asynchronous communication. These implementation decisions are made because an mobile agent has to be mobile at all times and is an autonomous piece of software that can make decisions itself. Implementing a mobile multi-agent system in this way does not guarantee that the agents are secure.

A mobile multi-agent agent system is considered to be secure when information stored inside the agent is only available to the agent and nothing else than the agent can change the information. Information should be seen in a broad context, using strong mobility with agents means that code, data, the execution stack and connection to other resources are vulnerable.

Before I tackle some solutions to protect agents from threats, I first explain the possible problems that may occur using agents and more specific mobile multi-agent systems.

4.1 Security Problems

One of the main reasons why agents are not so widespread is security. Not only does the agent execution environment requires security mechanisms, users and system managers must have the confidence to let agents migrate freely on their system [38].

Up until the introduction of mobile agents, software was stationary. This means that the main security problem was protecting the host from malicious applications [35]. In mobile computing the host does not contain data thus the threat for malicious agents towards host is reduced. There are however many other security issues[13][32][35][38].

Although there are no solutions to every security problem, multi-agent systems are attractive enough to start using them. Theoretical advances in cryptography and the design of security aware agent interactions are encouraging to make the security issue handleable. If this should not be the case, we would be bound to the physical security as it is today [35].

4.1.1 Exogenous Threats

These attacks occur outside the mobile agent environment. Usually when the agent is stored on disk or during the transport of the agent is transported over the network.

Typical solutions are encryption, to protect the agent from being read, and secure sockets, to protect the network connection [35][9].

4.1.2 Endogenous Threats

Endogenous threats are typical for multi-agent systems and are more likely to occur. Endogenous threats are threats that an agent form towards another agent or the environment where it is working in. The environment itself can attack an agent too.

Agents always communicates with agents it does not completely trusts, also called horizontal hostility. While on the other hand the agents do not completely trust the environment, this mistrust is mutual, also called vertical hostility [38].

Horizontal Hostility

As described in 3.2.3 on page 19, an agent is just a piece of code with private data in. This means that following attacks are possible [38][35]:

- *Unauthorized disclosure*: secret data stored within the agent is leaked to an unauthorized agent or environment.
  - Through spying or eavesdropping on communications between agents.
  - Through actually looking internally at agents.
- *unauthorized modifications*: Code or data in the agent is changed, thus damaging the agent.

Vertical Hostility

Vertical hostility is hostility between the agents and the environment. Attacks towards agents are the same in horizontal hostility. On the other hand agents can be a serious threat towards the environment [38].

The host is the entity that executes the mobile agent, sometimes also called the execution environment [35]. Occasionally the host is nothing but bare computer hardware but usually it is an interpreter that executes the agents high level instructions. (see section 3.2.3 on page 19). It is not hard to see that following attacks are possible towards the host: [38]

- *Unauthorized Disclosure* : Secret data stored in the kernel, or the enclosing environment of the kernel, is leaked to an unauthorized agent.
- *Unauthorized Modifications* : The code or data of the kernel, or the enclosed environment, is modified, paralyzing the whole platform.
- *Denial of Service* : One or more agents reduce the availability of some shared system resource. CPU, memory and the network throughput are classic examples of this attack.
- *Introducing Viruses and Trojan Horses* : the Kernel is tricked into executing malicious code. This code can cause disclosure, modification and denial of service attacks.

4.2 Protecting the Agent for Transmission Attacks

Eavesdropping on communications can be avoided using the Needham-Schroeder public key protocol with lowe's fix [35][30][38]. Another technique is *Hiding in the Masses* which does not require encryption. By "chaffing and winnowing" the data stream, it is possible to make the attacker clueless about which bits belong to what stream. Only the intended receiver is able to pick up the right bits [35].

Extending the Trusted Computing Base [35][16] is not exactly a solution that is wanted but it works. It basically means to make every element in the agent system trustworthy using a symmetric cryptosystem. It is easy to see that this remains a closed system. An asymmetric system is also a possibility known as "Encryption a la carte". Here the security is shifted from the execution level to the mobile agent level. Each system in the environment has a public key used by the agent to encrypt itself during transfer. When the agent arrived at the destination the host can decipher the agent [35].

4.3 Protecting the Agent Against Agent Tampering

Protecting an agent against malicious agents or hosts is a lot more difficult. With the classical example of the airline reservation-agent it is easy to see that unauthorized modifications are as bad as unauthorized disclosure [35]. Several solutions are proposed the one better than the other.

Detecting Agent Tampering [35][16] can only be used with unauthorized modifications. The so called "execution integrity" is checked. This can be done through several techniques like:

- Dummy items and functions that are checked if they are correctly processed;
- Range verification;
- Timing;
- Context information;
- Cryptographic techniques.

It is of no importance whether the agent is checked during the execution or on a host imposing some security policy. After the detection of tampering with the agent several possibilities are available. This may range from simply destroying the agent to have a reputation system, where judges or an arbiter program, decides if the hostility happened.

Clueless Agents [35] works in the same manner as most computer viruses do i.e., code sequences that trigger only when certain environmental conditions are met. These conditions could be time and date or in multi-agent systems when an companion agent resides on the same host. When this condition is met, the agent goes from decrypted state to encrypted state. Note that this does not guarantee integrity of code execution after the agent is transformed.

A smartcard [35] is a cryptographic co-processor attached to each local host. If this smartcard is manufactured by a trusted independent entity, the co-processer can be used by the agent to computed security sensitive code.

Scaling and Signing of Results [35]. To hide the data, a mobile agent can encrypt the acquired data with the current host public key. This data is then unusable for the rest of the agents journey. After the encryption, the local host still has to "sign" the new state of the agent. If this is not done another host still can change the encrypted information because everybody knows the public key.

Executable Encrypted Functions [35][5] protects the internal logic of the agent. The approach consists of encrypting polynomials in a way that they are still executable. Many computation can be effectively concealed this way, although the host carries them out. Just like in agents that do not use executable encrypted functions, it accepts input in clear form and computes an encrypted result without providing a clue on which function is computed. While this is not applicable to arbitrary applications it is a possible venue to completely "cloak" an agent and it does not need the inclusion of a secure computing base.

Time Limited Blackbox Security [16][5] takes an arbitrary agent and uses some conversion mechanism to make a blackbox agent. An agent is a blackbox agent if at any time:

1. Code and data of the agent cannot be read;
2. Code and data of the agent cannot be modified.

This is about the same as *executable encrypted functions* that are currently not available for each agent. If the definition above is weakened to

- An agent is a limited time blackbox in a certain time interval of
  1. Code and data of the agent cannot be read;
  2. Code and data of the agent cannot be modified.
- Attacks after this interval are possible but have no effects

There is a way to achieve this definition by *obfocussing* or applying a *mess-up algorithm* (see table 4.1 on page 30) to the agent. The "semantics in the large", i.e. the overall semantics of the elements to the overall semantics of the application, are changed. The central idea now is not to allow the attacker to build such a model of the agent in the specified time interval.

Original Code
if(a(b) < c) {
b = s(d(e) + f);
}
Converted Code
z = 0;
DO
if(z = 0) then t1 = a(b); z = 1; continue;
if(z = 1) then t2 = t1 < c; z = 2; continue;
if(t2) then t3 = d(e); z = 3; continue;
if(z = 3) then t4 = t3 + f; z = 4; continue;
if(z = 4) then b = t4; z = 5; continue;
if(z = 5) then break;
LOOP

Table 4.1: A Mess-Up

4.4 Protecting the Host fsrom Malicious Agents

Virus Detection [13] [6] is not an option to decide if an agent would behave like a virus. Virus detection is possible in and only in Turing-complete languages, but it is easy to write non-Turing-complete languages where it is not decidable.

Proof Carrying Code [35] requires a proof that shows that every piece of mobile code behaves according to some security policy. This approach has been demonstrated to work for native machine code.

Single Virtual Machine Architectures [35] places all agents and (parts of the) kernel in the same address space. Because memory access is proved to be bound agents cannot tamper with other memory, all memory interaction is only possible through the official method calls only.

Authentication [35] will most likely introduce digital signatures to an agent. The author has to sign the code, the sender signs the agent and adds other parameters like the start time, send time .... The receiver host can read the signature and convince himself that the agents is not tampered with. Hash-codes can be encrypted and checked with public-keys. After the authentication a host can add security parameters to the agent based on the author and sender of the agent.

Denial-of-service attacks can be avoided by *allocating resources.* If resources become unavailable the host has to penalize agents that take up to much resources. Another way to avoid denial-of-service attacks is market-based. Each Agent has to pay for the resource it uses. The problem then becomes one of steering and securing the flow of money.

Chapter 5

Advantages of Mobility in Mobile Multi-Agent Systems

Adding *security to applications* is not so difficult to handle. Various methods exist to protect an application from using the application unlawful. On the other hand *data security* is becoming a major problem. Data, a paper, music, customer addresses, whatever you can come up with, has to be some sort of public so the data can be used defined by the owner. Agents can be used so that nothing else than the *intended use* is available.

Nomadic user use a portable device like a laptop or Personal Digital Assistant, with this device they often go from site to site, gathering data. Nomadic users require two things, the first is having applications on the networks that they frequently visit so that data exchange can be done easier and faster, or *application roaming*. Application roaming also includes *remote management* so that applications are automatically updated and upgraded. The other requirement is that data that the user carries with him has to be made public, if the user wants that of course. Other users want to be notified when some information becomes available. All this falls under the term *user roaming*.

when "normal" users work on a system they want to have a reliable system. This means if a server goes down for maintenance, the user still wants to work with his program. Making a system persistent is not hard with mobile agents. Just move the agents from the system that goes down. A fault tolerant system is harder to make, 100% fault tolerance is not possible with agent systems. But agents can be used to recover automatically from faults. The faults I address are the faults that happen when agents, or in a client/server architecture servers, are not available. When this is due to a network fault the system can be still continue to run without loosing data. When agents are not available because the agent actually died, the dead-agent can be recreated using the knowledge distributed over all the other agents.

Load balancing applications is a difficult task, usually it is done using several servers that contain the same data only the traffic is redirected to a server that is less used. Because agents are mobile, replicas are not needed anymore. When the resources on a system are to busy an agent can just move to another system that is less busy.

Now that it is clear what agents are, how they are implemented, communicate and how they can be made secure, I will elaborate on application and data security security, roaming for users and applications, reliability of applications and automatic load-balancing. As last point in this chapter I will explain the application.

5.1 Security for Applications and Data

Several schemes exists to make applications secure. Most of theses schemes require some sort of identification before installing. Once the applications is installed no further checks are being made to check if the user is indeed the only one that the program is using.

Applications build using agents can be made quite different than applications build using other mechanisms. Mobile multi-agent applications are distributed, this means that the applications can be engineered in such a way that there should be only one installation of the application. Making an application this way requires identification of the user.

All users are represented by their own agent. These agents are then used to perform the actions the user wants from them, eg collecting data, making appointments. This agent can also be used for identification, and can be used to check if the user is granted access to the application.

In fact all agents in an agent application, are owned by somebody or something. This way it is easy to add security and extra flexibility to applications. Extra flexibility is explained in section 5.2.3 on page 35.

Adding security to applications or data using agents can be done with the use of a stub-agent. This stub-agent is an intermediate between the user of the application and the actually application or data.

I will use the paper example that is used in the case to clarify the idea (see section 6.2.1 on page 48). Here the content of a paper (technical or otherwise) is divided over several agents. One agent for chapter one, another agent for chapter two and so on. To combine all these agents a stub-agent is needed. This stub-agent will get the contents from the desired agent.

Because the stub-agents is the only agent that knows where to get the data from it is easy to control the data streams. Eg a user is only permitted to read the paper and not permitted to print it.

The same idea can be applied to applications as well. The idea is hat a complete application is written, using agents or otherwise, and only the stub-agents are allowed to contact the application.

5.2 Roaming

Users of computer systems are becoming more and more mobile, so called nomadic users. To enable real practical use of this mobility, roaming is a necessity. Without roaming a laptop or a Personal Digital Assistant (PDA) is nothing more than a small portable desktop computer. It might be suitable for some users but most users want more than this.

I use the term *user roaming* if the user receives events when connected to the network. *Application roaming* goes a little further, it lets agents migrate form a portable device to a more permanent network connection. I make this distinction between user roaming and application roaming because application roaming requires remote management.

5.2.1 User Roaming

User roaming is basically just nothing more than when a user connects to the network it automatically receives all the events that are stored for the user in that network.

This can be compared to a mail program. In a distributed environment this means that the user polls the mail server to see if there is mail. With roaming this means that the mail server notifies the user. making polling, a process that mostly wastes resource, redundant.

Ubiquitous systems require user roaming[15][29]. An ubiquitous system is a sensor/actuator rich environment that provides both mobility-transparent and context aware (adaptive) access to system resources [29]. A domotica system is an ubiquitous system, sensors trace where somebody is and the actuators change the room to the condition that the user likes. This can go from changing the light settings to change data streams for music and television. Another, more useful, example is finding the closest printer so the user can print. If the printer is controlled using an agent, the security mechanism described in section 5.1 on 33 can be used.

Except the user roaming "A user walks from one office into another and logs on, they expect to see their desktop an settings, not the machines"[17]. Roaming is not fully implemented using the more traditional programming paradigms.

5.2.2 Application Roaming

The requirement to let applications roam over various networks almost became a necessity for data-mining. Databases are becoming more and more distributed thus the data-mining applications must go to the servers. Acquiring data and computing the results on one host is not applicable because of the network-load it would create.

Not only data-mining application can benefit from application roaming. Most applications that a nomadic user requires are data gathering tools. Eg a sales-person. The example of a nomadic user in the application is a visitor of a computer science department. Gathering data here consist of acquiring papers that are of interest for the visitor. On the other hand the visitor could have papers himself that he wants to distribute in the department.

There are plenty of examples of roaming in mobile multi-agent systems ([15][29][23]). All these examples still do not show what is fully possible, like moving graphical user interfaces, with application roaming in mobile multi-agent systems.

5.2.3 Remote Management

Application roaming requires *remote management*. When applications roam over various networks network managers will never know what is running on the systems. A nomadic user could have introduced a new application on the servers without their knowledge. On the other hand a nomadic user that has placed an application on a server wants to update or upgrade the application.

To solve this problem automatic remote management is used. This automatic remote management system uses agents to update older agents. Upgrade might not be a good word to describe the way they work. They just kill the agents that are to old and replace them with new ones.

If the application security mechanism is used from section 5.1 on 33 this is quiet easy. Just replace the old agents that contains the data. The only agent that has to be notify of the agents change is the stub-agent.

In this case study the paper application is used to show how remote management works.

5.3 Reliability

When applications roam over the network it is a must that these applications are reliable. An application that does not work anymore when some part is not available anymore is of not much use for a nomadic users that checks on the application only once in a while. Even non-nomadic users of applications do not want to be bothered with servers that are not available. They want an application that works at all times.

A system is reliable when it keeps functioning at all times and when no data is lost if something goes wrong. In a multi-agent system a fault is loosing an agent, due to a network fault or when the agent actually died.

The reliability scheme proposed here is not a 100% fail save scheme. It all depends on how many agents are actually dead or unavailable.

5.3.1 Persistency

I consider an application persistent if it keeps working when there is something happening on parts of the system.

In mobile multi-agent systems it is not hard to keep an application running when a part of system is shutting down correctly. All it takes is to move the agents before the system shuts down completely.

To ensure persistency it is needed that the system knows which agents are working on the system. Only when the system is sure no more agents reside it can shut down.

It is more difficult when agents are not available anymore. This is discussed in the following section.

5.3.2 Fault Tolerance

Besides the faults happening in every application, eg divide by zero, there is one other problem. Agents that are not available anymore. Because agents use asynchronous communication (see section 3.3.3 on 24) it is not possible to be sure that the agent is dead. With timed synchronization it is only possible to assume that an agent is not responding anymore.

When an agent is not available anymore it might be possible to recreate the agent. To recreate an agent the data inside the agent should be distributed. when there is a request to recreate the agent all data that is distributed should go to that newly created agent.

For an agent that is dead the new agent could take over its functionality. For an agent that is not dead, eg a network fault, the presumed dead agent was doing calculations and did not answer its messages, the old and the new agent have to merge. So that there is only one agent left.

5.4 Load Balancing

Scheduling n jobs on m machines (with $n >> m$) is a widely studied problem in computer science [18]. Jobs come in one by one and have to be scheduled without further knowledge of jobs that might come in the future. The goal of this scheduling is minimizing the time needed to complete all the jobs. [18].

Most scheduling algorithms schedule a job before the job has to start. Because of the mobility of agents in mobile multi-agent systems this can be changed to a more dynamic algorithm. Computing the optimal location for an agent usually takes some time too, especially when there are lots of locations where an agent can go to.

Load balancing is more than just scheduling a job so that is completed on time. In mobile multi-agent systems load balancing is the optimal use of resources. Agents do not usually have a deadline, just a design goal. It is undetermined how they reach their design goal (see section 2.1 on 10). This means that time constraints are not an issue, the use of resources, eg CPU time, memory load and network usage, is the main problem.

Applications that run on low memory or low processor devices, such as PDAs, could use roaming to enhance the performance when roaming is used. When the PDA is in a network connection, other more powerful devices can be used to continue the computations.

Some load balancing issues in mobile multi-agent systems can not be performed by the system because of privacy issues. Two agents that communicate with each other frequently should be close to each other to reduce latency, but acquiring that information should be impossible (see chapter 4 on page 26).

The only thing that is left for agent systems is to let the agent decide when to move and where to move to. Agents themselves can decide autonomously to move because the latency is to high or the resources needed are not available. Agents can be asked to move because resources, like memory and CPU, become overloaded.

Scheduling problems in distributed systems are typically done low-level. When servers become to busy other servers with the same data are available and the requested connection is redirected to that server. When the servers have a lesser load the several servers will synchronize [26]

Scheduling algorithms in parallel computing are static or dynamic. Static algorithms must know the load in advance and then compute what the best scheduling is. Dynamic scheduling do not know the load in advance, and try to schedule a job so that its time constraints are met [18].

5.5 The Advantages in One Application

The application I use for the case study is some sort of application that can be used in a department of a university or at a convention. The intention is to have an application that shows how security, roaming, reliability and load balancing is implemented.

The case is some sort of agenda, that scans the network for persons with the same interests. When an person or rather agent is found with the same interest the owners are contacted – user roaming – and when the are both positive to have a meeting the agents will start to negotiate on the time to have the meeting.

Because the persons meeting usually want to discuss papers there is an "automatic version system" available. Here the owner of the paper gives the rights to the other persons agent to read the paper – application roaming and security –, when the paper is updated the readers of the paper automatically get the update as well – remote management –. Reliability and load balancing are omnipresent.

Chapter 6

Application Design

Agents that are mobile and use asynchronous communication can be used to add the advantages of security, roaming, reliability and load balancing. In this part I show how the applications can be designed to add the advantages. In the first place , I will show the internal structure of the agents and why these fields have to be present in the agent. The second step in the design is the the communication and migration of the agents.

6.1 Agent Design

I will explain the internal structure of the agents using the class representation of the Unified Modeling Language (UML) class diagram. This will only explain the methods and data necessary to implement security, roaming, reliability or fault tolerance. Extra functionality required for actual application is omitted to make the explanation as clear as possible.

The UML class representation is chosen because agents also have data and methods. All data in this representation is private, meaning that it cannot be accessed. All methods are public, meaning that each other agent can start that function using message based communication (see section 3.3.2 on page 22. All other functionality, inheritance, templates, etc, in UML is not used because it is not applicable to all mobile multi-agent systems.

6.1.1 Data Agent

The Main Data Agent

Each user of the application has a data agent running on his portable device. This data agent will never leave that device for this application, but this agents gets cloned. This means that all required data, the appointments and the interests in this case, is put in a new agent that will roam over the network. This is done for two reasons. First the original agent might contain data that is not needed for the agenda application, like addresses and tasks. If this is the case it probably means that the agent has to be available for other applications that might be available on the portable device . Second, and more important, leaving the main data agent on the portable device helps for the reliability.

Main Data Agent
Appointments Interests Cloned Agent Reference
Start Roaming(System Agent) Stop Roaming(System Agent) Clone Agent(System Agent) Merge()

Figure 6.1: The Main Data Agent

Application security is not an issue in the main data agent for the agenda application. This does not mean that it cannot be implemented. The only thing that is required is that it has to be possible that the required data for the cloned agent agent can be copied. If, for some reason, the application security is used then this is done for other applications than the agenda application.

The Start Roaming(System Agent) method uses the Cloned Agent Reference to see if the roaming already started. If it did not yet start the main agent gets cloned. This cloned agent will then roam over the network with the other agents. If the roaming did already start, the main agent has a reference to the cloned agent so it can start to merge. This merge is just a synchronization between the appointments and interests.

If the main agent already contains a reference but it cannot communicate with this reference, the main agent assumes that agent is agent is dead and just restarts the roaming.

For garbage collection reasons the Stop Roaming(System Agent) should be implemented also. This just informs the *application server agent* that it can kill all applications started for the cloned agent. The cloned agent will get killed also.

The Cloned Agent

The cloned agent is the agent that roams over the network for the *main data agent*. This agent tries to find appropriate candidates to make an appointment with. When it finds a candidate it is stored so the owner of the main data agent can make a judgment if it really wants an appointment. When both candidates agree the agents try to find a suitable time for this appointment.

For the application security agents cannot know the real reference of the agents in Register(Agent Number) and Negotiate Appointment(Agent Number). If the agents would get the real reference both agents could start to communicate themselves. This is the same for Interests(Locator Agent), where the locater agent serves as a stub-agent between the two agents.

Cloned Data Agent
Appointments Interests Interested Agents Agent Number System Agent
Interests(Locator Agent) Register(Agent Number) Negotiate Appointment(Agent Number) Merge() Merge(Agent Reference) move()

Figure 6.2: The Cloned Data Agent

Merge() is intended for the Main Data Agent. This methods just synchronizes the appointments and the interests between the original main data agent and the cloned agent when the main data agent becomes available again.

When the main data agent cannot find the Cloned Data Agent when it starts roaming it automatically makes a new Cloned Agent for reliability. When there is a network fault it might be possible that there are two agents for the same main data agent, which should be impossible. Thus those two agent should merge with the Merge(Agent Reference). In this case the agents can know the reference of each other because they contain, almost the same data that belongs to the same owner.

Agent Number is also used for reliability. The *application server Agent* uses this number for application security. All cloned data agents use this number to a reference they cannot know for application security. If this number would not be stored inside the real agent the application server would not be able to know or recreate the numbers.

6.1.2 Application Server

The *application server agent* is the core agent that controls the roaming and application security. To enable finding interested parties some agent has to be used that knows each data agent by reference, the applications started for that agent and the unique agent number.

References to all the application agents are kept for one reasons, garbage collecting after the roaming stops. When the roaming stops, invoked with Stop Roaming(Data Agent Reference). The applications that the cloned agent has used have to be killed, otherwise they would continue doing their task for nothing.

Next Agent(Agent Reference) can only be used by the locator agent, hence the *Agent Reference*.

Application Server
Reference to Data agents, the Applications for that agent, and the unique Agent Number
Start Roaming (Data Agent Reference) Stop Roaming(Data Agent Reference) New Data Agent(Old Data Agent Reference) Next Agent(Agent Reference) Application Agent Dead (Agent Reference) Data Agent Dead (Agent Reference) Move() Merge(Agent Reference)

Figure 6.3: The Application Sever Agent

Application Agent Dead(Agent Reference) and Data Agent Dead(Agent Reference) are for reliability. It just notifies that application server that these agents are dead. In case of *application agents* new application agents can be made. In case of data agents, the references will not be used anymore until a new cloned agent is presented with the method New Data Agent(Old Cloned Agent Reference).

Merge(Agent Reference) is used in case there was a network fault. A network fault separates agent system, it should be clear that with the reliability mechanisms there is automatically a second application server agent. When the network fault is fixed one application agent should be killed, but first the two agents must synchronize their data, if this is not done various agents will work for nothing and other agent will refer to the wrong agents.

Move() is for persistency and load balancing. It just tells the application agent to move to another system of the system is shutting down or when the application server agents uses to much resources on the system.

6.1.3 People Finder Application

The people finder application just compares interests between two agents and decides if they should be considered interested in an appointment. For each cloned data agent in the system there should be one and only one people finder agent.

People Finder Agent
Move() Die()

Figure 6.4: The People Finder Agent

The people finder agent is a stub-agent between two cloned agents. It protects both agents from direct communication with each other. This is important because all cloned agents could have the same interface and agent tempering could be happening with direct communication.

When this agent dies the application server agent will automatically make a new people finder agent for the cloned agent. If it would happen that two people finder applications are available for one agent then one people finder application will have to finish the communication that is began and can then `Die()`.

Again, the `move()` method is necessary for load balancing and persistence.

6.1.4 Appointment Maker

The appointment maker is the agent that finds a suitable time for an appointment. Again this is a stub-agent that is used to avoid direct communication between two cloned data agents.

Appointment Agent
move() die()

Figure 6.5: The Appointment Agent

As for the people finder agent the appointment agent will be automatically recreated when it would happen that the appointment agent is not available anymore. If there are two appointment agents for one cloned data agent then one of the two agents can `die()` after it finished with making an appointment.

The `move()` is used for persistency and load balancing.

6.1.5 Automatic Version System

The automatic version system (AVS) is a system that can used for distributing data. In this case the data is a paper but this could also be a music file or other data than has to be consulted or manipulated. The AVS automatically changes the old version of the data to the new version when it becomes available.

AVS Agent
Open Files
Request File(File Name) Update File(File Name) Register File(Agent Reference, File Name) Clone File(File Name) Merge(Agent Reference) Move()

Figure 6.6: The Automatic Version System Agent

Some agent that requires a file asks the AVS to open that file for the agent. This file is is divided into several agents and one stub-agent is created as an interface for all the agents that form the file. This stub-agent is used as an interface, defining the use of that file. This way the file is protected for undesired actions.

The data source can use the remote management function of the AVS. This happens when Update File(File Name) is called. The AVS will track down the agents used to represent the files in Open Files and change the content of those agents.

The AVS is used to recreate agents that became missing. When the stub-agent is unable to communicate with an part of the file a new agent will be created with the contents of that dead agent. When the agent suddenly comes online it can just be deleted. When the AVS is unavailable the data stored in the stub-agent and file agents can be used to reassemble a new AVS agent – Register File(Agent Reference, File Name). Merge(Agent Reference) is used when there are two AVS agents. It just works in the same way as the Application server agent, merge the knowledge of the open files, kill agents that are redundant and inform the stub-agents of the changes.

Close File(File Name) and Move() are for load balancing. The move method moves the AVS agent to another location if it uses to much resources. When the file is closed, all agents that represent the file can be killed.

6.1.6 User Interface

The user interface is an example of a roaming application. The user interface does nothing but contact the cloned agent and user main agent to display information stored in that agent. When the user disconnects from the system the user interface can still be used to make changes in the main data agent. It can also be used to contact the cloned agent when disconnected, but because it uses synchronized communication the message will only get to the cloned agent when the portable device connects to the network again.

6.1.7 System Agent

The system agent does not have to be an agent. It is used to denote the system, CPU, memory and network connection.

System Agent
Residing Agents
Register Agent(Agent Reference) Unregister Agent(Agent Reference) Negotiate AVS() Negotiate Application Server()

Figure 6.7: The System Agent

The system agent needs to know which agents are. When to many resources are used the system can decide on which agent that has to go. When the systems wants to go down it has to inform the agents to move. When all agents are gone the system can go down safe.

The system is responsible for the two main agents, the application server agent and the automatic version agent. When those agents are dead only the system is able to restart them using the Negotiate AVS() or Negotiate Application Server(). These methods will contact all surrounding systems and negotiate on who is restarting the dead agents. This can be a full fledged negotiation algorithm. It can also be that the system that finds out the agent is dead, makes a new agents and informs all other systems. When there are several agents made they can simply merge to one agent.

6.2 Agent Communication

The agent communication is modeled using message sequence charts. Basically this only shows communication and migration as well as actions based if the agent receives the message. The charts only show in which order messages are send, received and what action is taken. It does not show the time when the actions are taken because of the asynchronous communication (see section 3.3.3 on page 24).

Appendix A on page 67 contains an explanation of the message sequence charts (msc) as they are used here.

6.2.1 Application Security

The application security is implemented in several places of the application. The main idea behind application security is that there are several data sources that want to share data for computing reasons but this data should not be public for other agents. This is achieved using a stub-agent. This stub-agent acquires the data from both agents and computes the result. This result is then passed to the agents.

Application Security in the Application Server Agent

The application server agent is an agent that creates the stub-agents. To do this each cloned agent has to register to the application server agent. The application server agent will start the stub-agents, in this case the people finder agent.

The application server agent is used to manage the stub-agents that ensure the security. The application server also gives unique numbers to the cloned agents, just in case an agent needs to keep a reference to an agent where direct communication is not allowed.

A disadvantage of this approach is that only the application server agent is allowed to make stub-agents, this means that when some user has a custom made people finder algorithm, the user cannot use this algorithm. If it would be allowed then there is no check that the algorithm does not pass on data.

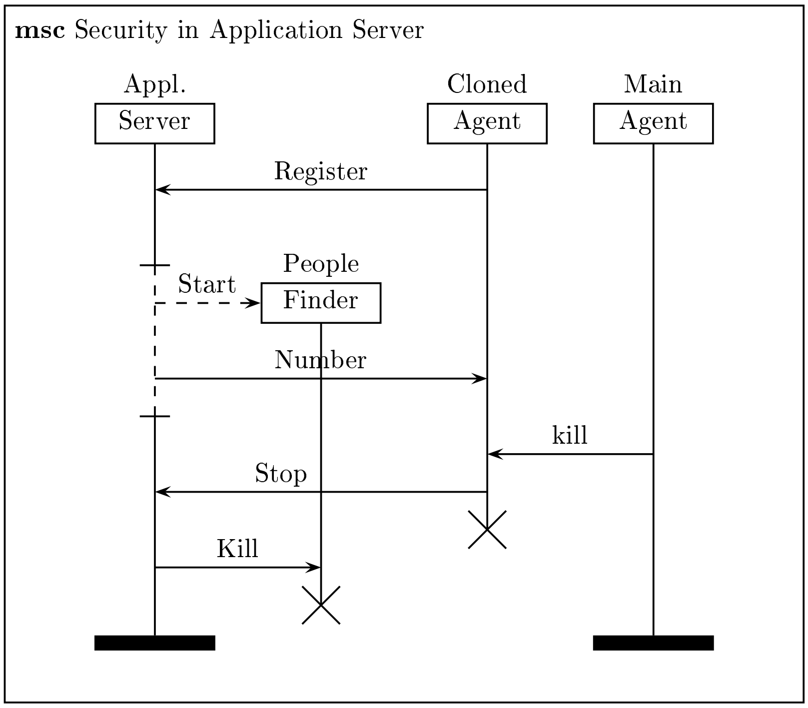

Figure 6.8: Security with the Application Server Agent

In this chart (figure 6.8 on page 46) the roaming is started for a main agent (see section 6.2.2 on page 51), the cloned agent registers to the application server agent so the people finder agent can be started. The people finder agent is the agent that ensures the security.

The application server agent is used to protect agents (see figure 6.10 on page 49). The people finder application finds cloned agents for possible appointments. If the people finder application agent proposes another agent for an appointment some sort of reference to the proposed agent has to be kept. This reference cannot be the real reference.

To solve this problem the application server agent sends a unique number to the cloned agent. Other agents can use that number to request a stub-agent for communication between the two agents.

Application Security with the People Finder Agent

The people finder agent is an example of a more permanent stub-agent. In this case a people finder agent is created for each cloned agent (see section 6.2.2

on page 51).

The people finder agent is the agents that compares interests from two agents, if both agents are considered to be interested in each other they are notified.

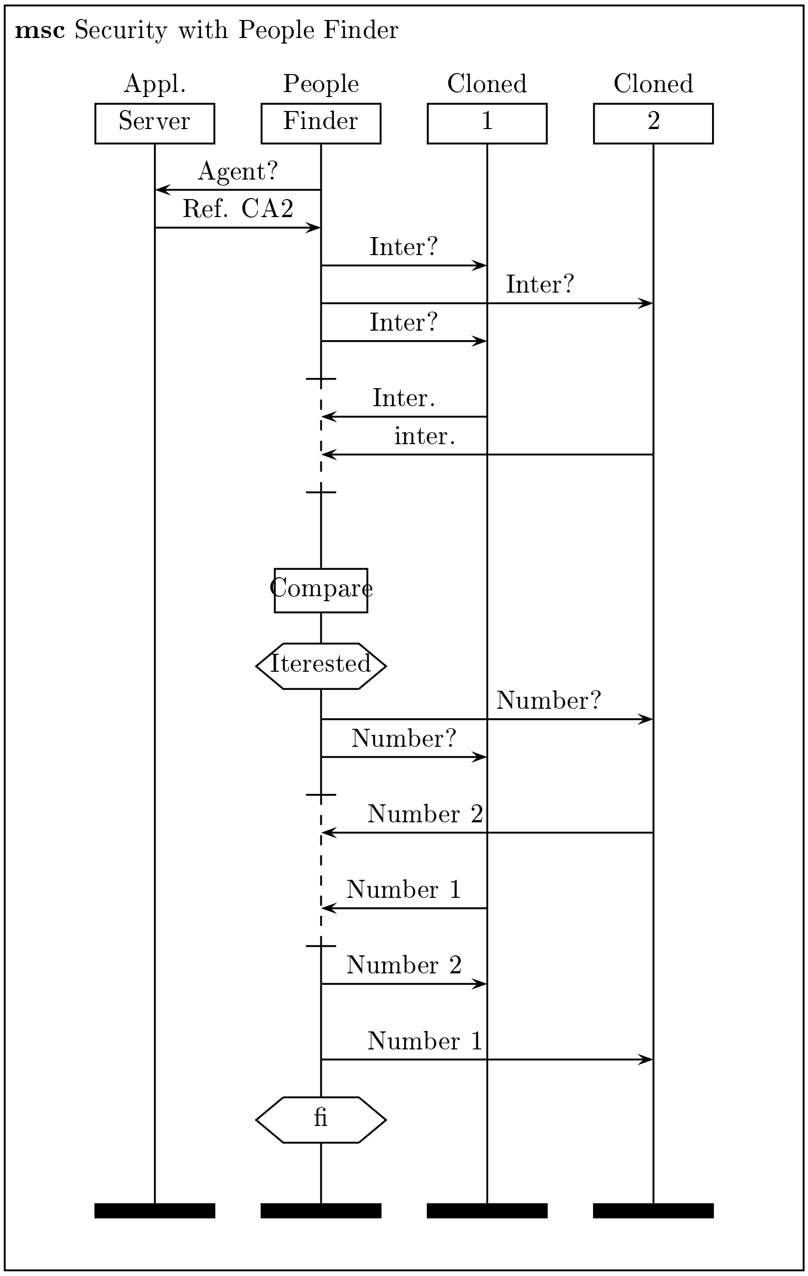

Figure 6.9: Application Security with Stub-Agents

This chart (figure 6.9 on page 47) shows how a stub-agent works for tow agents that cannot communicate directly with each other for security reasons.

At first the people finder agent asks the application server agent for an agent to communicate with. When the people finder agent receives the reference to the agent it can start to communicate with that agent, here cloned agent 2.

Because agents are dynamic the people finder agent requests the interests from both agents. When the people finder agent has received both interests it compares them. Based on the comparison between the two interests the people finder agent decides if both agents are compatible. If this is the case the reference number is asked on both agent. When the number is received it is send to the opposite agent. After this exchange a new agent is asked from the application server agent by the people finder agent, and it all starts from the beginning.

If there is no interest from both parties, the people finder agent requests a new agent reference from the application server agent and it all starts again.

It is clear from this chart that both cloned agents have no direct communication, and can never have direct communication.

Application Security in the Appointment Agent

The appointment agent does the reverse of the people finder application. If a user wants an appointment with another user of the system, both cloned agents do not know who the agent is.

The way the cloned agent communicates with the application server agent shows how the unique number of the agent is used. See figure 6.10 on page 49 for the chart.

When the first cloned agent wants to make an appointment with the second agent, it gives the unique number to the appointment agent. The appointment agents receives the second agent reference to the appointment agent and the desired actions can be taken.

The way it is shown here, the appointment agent requests and receives both agendas. It computes a suitable time for the appointment it is sends it to the cloned agents. After which the appointment agent can stop working. In the actual implementation this is done the same, but the agents can disagree with the proposed time. If the disagree with the time a new time will be searched.

Application Security with the Automatic Version System

The Automatic versions system was initially intended to show program roaming. But it could also be used to show how the application security can be used to protect data. This is the same as the client/server architecture, but with the advantages of mobility. I show to do this in mobile multi-agent systems.

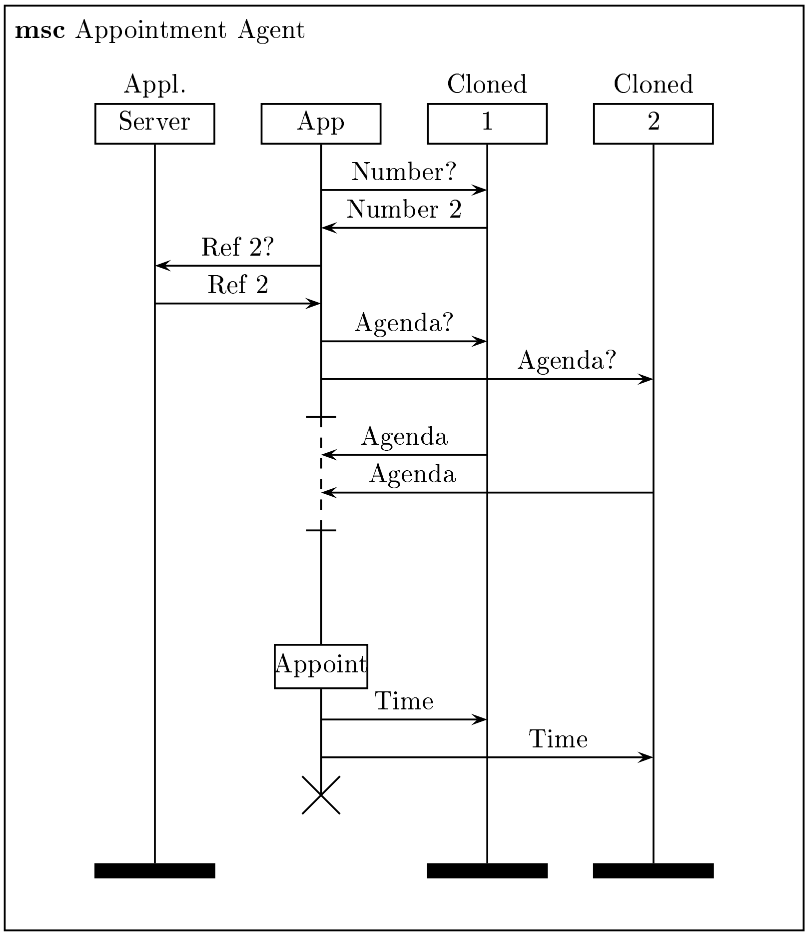

Figure 6.10: Application Security in the Appointment Agent

In the automatic version system there are several users that want to share data, but their data should be protected from malicious use.

See figure 6.11 on page 50 for the chart.

The agent that request data does not know where the data can be found, but the automatic version system does know, so it is requested there. The request is forwarded to the owner of that data. This owner makes the agents used for the data sharing. The reference to the stub-agent is send to the AVS which in turn forwards it to the requester.

Users that are not allowed to access the data are just being ignored. Like you can see on the chart.

Other methods are possible too:

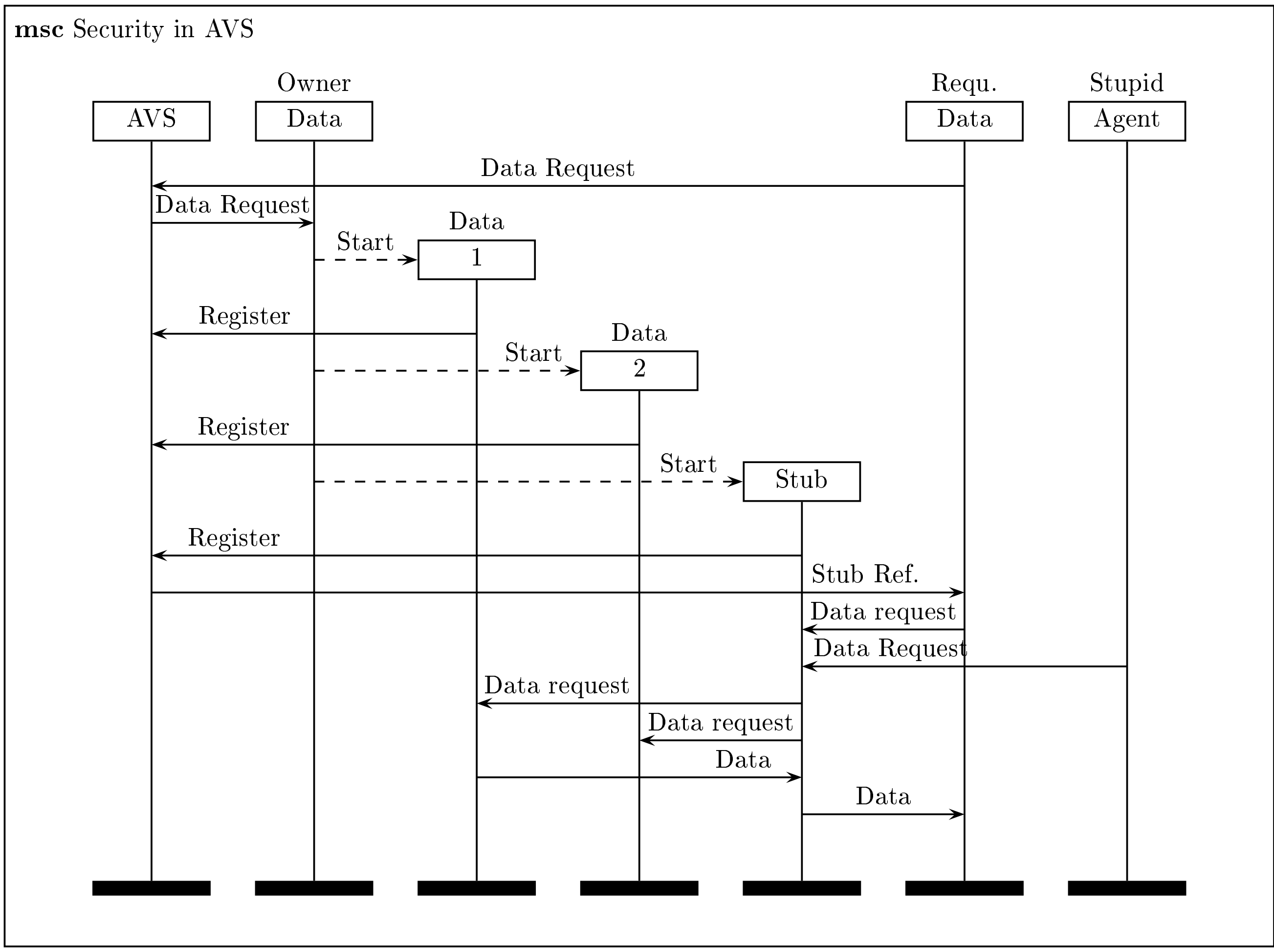

Figure 6.11: Security in the Automatic Version System

- If the requester knows where it can find the data it can request the data direct from the owner of that data
- The AVS does not necessarily need to know where the data it, it might as well just know who shares data, the request for that can then be multi-casted.
- The owner of the data does not necessarily need to make a new stub, existing stubs can be told to share data with other requesters.
- The data does not necessarily needs to be recreated, other stubs can be made for different requests.

The method used depends on the requirements of the system.

Considerations

The first scheme, creating a stub-agent for the roaming agents, in this example is rather silly. But it is useful for certain situation. Eg the organization of travel agencies has an agent system with this application security. Customers could use that agent system to find the best price for a certain destination. With the use of the stub-agent promotions from some travel agencies are not communicated to other travel agencies.

What is not shown in the first scheme is that messages from agents where the stub-agents is not communicating with are ignored, just like in the second scheme.

The second scheme is useful for everything that shares data. The data agent should implement all functionality and the stub-agent controls what is passed to the data agents. This is the same as a client/server architecture that uses a database, but the data agents can use the advantages of mobility, something a client/server is not capable to do.

For security and other reasons like persistency and roaming, the mobility of the data agents can be used. If more security is wanted the stub agent can be placed on another location. If this is done the user cannot take the agents with him on the portable device. Or the data agent is on the device or the stub agent is on the device. For persitency the database can be split, the stub agent will know where to locate the desired data, and placed on several locations, if some agents are not available anymore the data might be found in other agents.

6.2.2 Roaming

User Roaming

Letting users roam the network is not that hard in mobile multi-agent systems, once the connection is made the user just has to send a message to some agent that has knowledge of applications, like the application server, the user can use.

This requires a network connection at all times, and when the user is disconnected it cannot share its data or locate data.

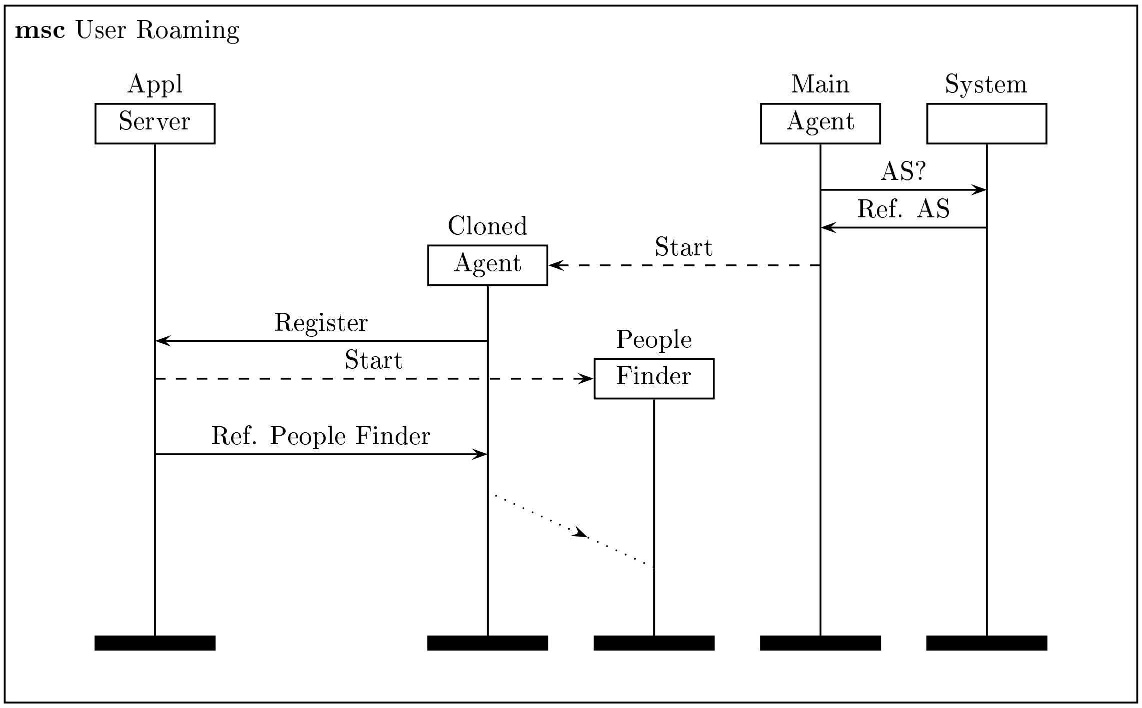

Figure 6.12: Roaming Start

Figure 6.12 on page 52 shows how the roaming starts. First the main agent request the reference of the application server from the system where it did connect. After that the main agent makes a clone agent, this agent contains some internal data from the main agent necessary for the application. This cloned agent registers itself to the application server, this is needed to let the application know that there is a new agent.

For the security (see section 6.2.1 on 46) a stub-agent is needed. This stub-agent is also useful for roaming. When a stub-agent is used the cloned agent is only a data agent. This means that the cloned agent just needs functions to retrieve, change and store data. In case of roaming this is an advantage because, different systems can implement different applications and the user can roam over all those systems.

When the cloned agent is just a data agent, the cloned agent needs an agent that implements functionality. This is in the chart done by the application server agent. The application server agent create the people finder agent. It sends the reference of the people finder agent to the cloned agent so they can start to communicate.

Up until now the cloned agent is still on the portable device. This means when the user disconnects, the cloned agent is not available to roam. Moving the agent to the same system as the people finder applications is safe to do because the

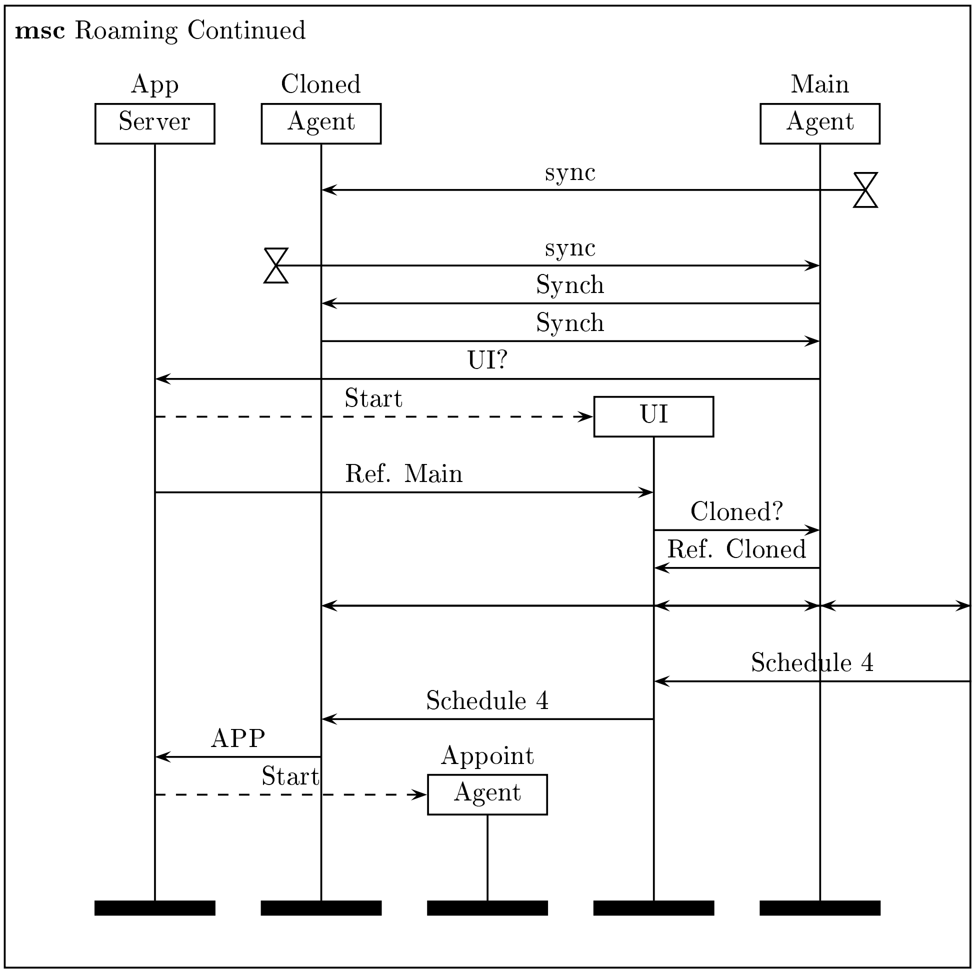

Figure 6.13: Roaming Continued

people finder application will remain on a location that is permanent on the network.

Once in a while the user will come back to check on the progress of his cloned agent (see figure 6.13 on 53). The first message send is an synchronize message to the cloned agent, this is for reliability (See 6.2.3 on page 57). After the synchronization is over both the cloned agent and main agent will synchronize their data content, this is done because the cloned agent could have gather data, it is also possible that the main agent has acquired other data.

In this case the application provides a user interface for the main agent, the user interface is actually an example of remote management (see 6.2.2 on page 54).

The UI will communicate with the user, Main agent and cloned agent, In this chart (figure 6.13 on 53) the user decided that an appointment with the owner of the cloned agent 4 is appropriate. The user interface will forward this message to the cloned agent.

The cloned agent will ask the application server for an appointment agent, how this continues can be seen in figure 6.10 on page 49.

Because the cloned agent is still looking for appointments the user has to connect to the system again when it wants to quit the roaming (figure 6.14 on page 55. This is about the same is continuing the roaming, first the contents of the cloned agent and the main agent will synchronize, because the cloned agent can still have found data where the user is interested in.

After the synchronization the main agent will command the cloned agent to kill itself. The main agent lets the application server know it stops with roaming. For load balancing reasons all agents that are started for the cloned agent have to be killed. If they would not get killed those agents will just keep running on the system using resources.

The agents do not get killed by the application server. The application server just informs the agents that they can terminate, before terminating the agent can perform some cleanup, depending on the application.

Application Roaming

Figure 6.11 on page 50 is besides an example of security also an example of application roaming. The stub-agent of the paper could migrate to the portable device where it could be used by the user of the portable device. When only the stub-agent is used the user will not get any reaction until it connects to the network. If security permits, the data agents could also migrate to the portable device.

It only starts to get interesting when the data agents are modified, thus requiring remote management.
in this case the data agents created by the data owner keep the changes. When the user connects to the network the data agents can update the data by the owner. This is shown in figure 6.15 on page 56.

Remote Management

Figure 6.15 on page 56 shows how ater the roaming has started by the user, the original data agent is availble, after that both agents update their contents.

Not only data agents need remote management, application agents could be changed during the time the user has disconnected from the network. This is shown by the UI agent in figure 6.16 on page 56

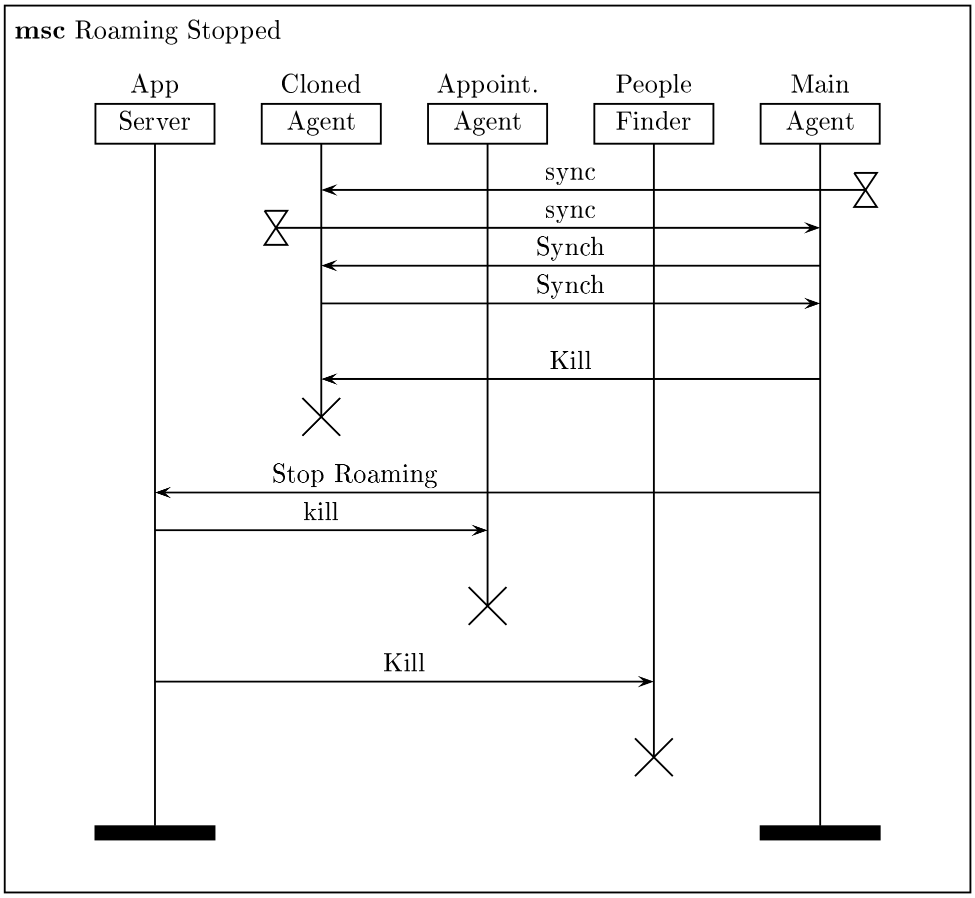

Figure 6.14: Roaming Stopped

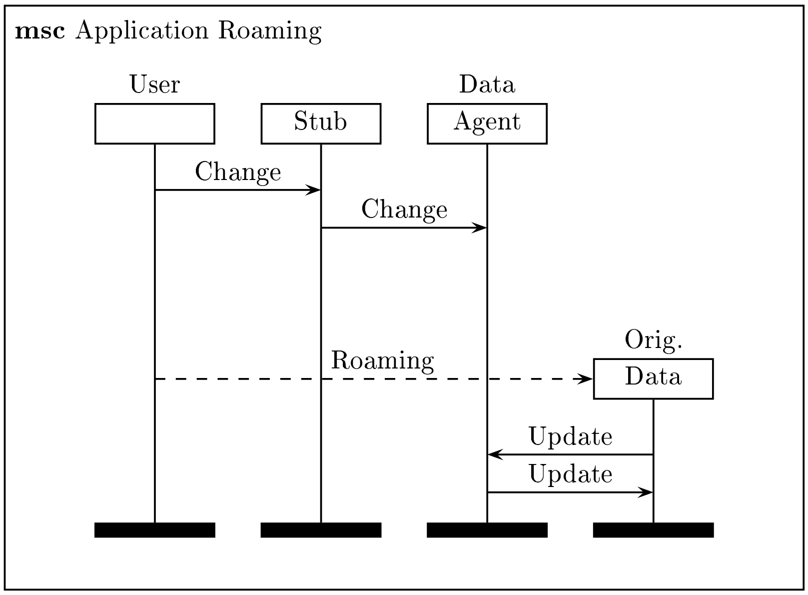

Figure 6.15: Application Roaming

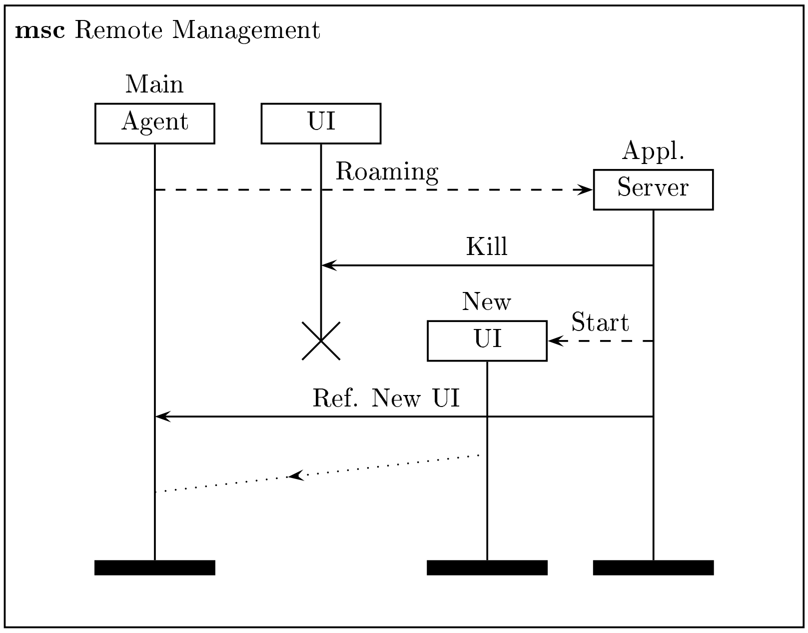

Figure 6.16: Remote Management

When the application server agent is available after roaming, the application agent server checks if the user interface for the main agent is still up to date. If there is an upgrade available the old user interface is notified that is has to terminate, in this case the user interface will notify the user of the upgrade. the main agent gets the reference for the new user interface so it can be used.

6.2.3 Reliability

Persistency

When systems shut down the agents that are on the system should use the mobility to migrate to another system. After they are all migrated the system can shut down (see figure 6.17 on page 57).

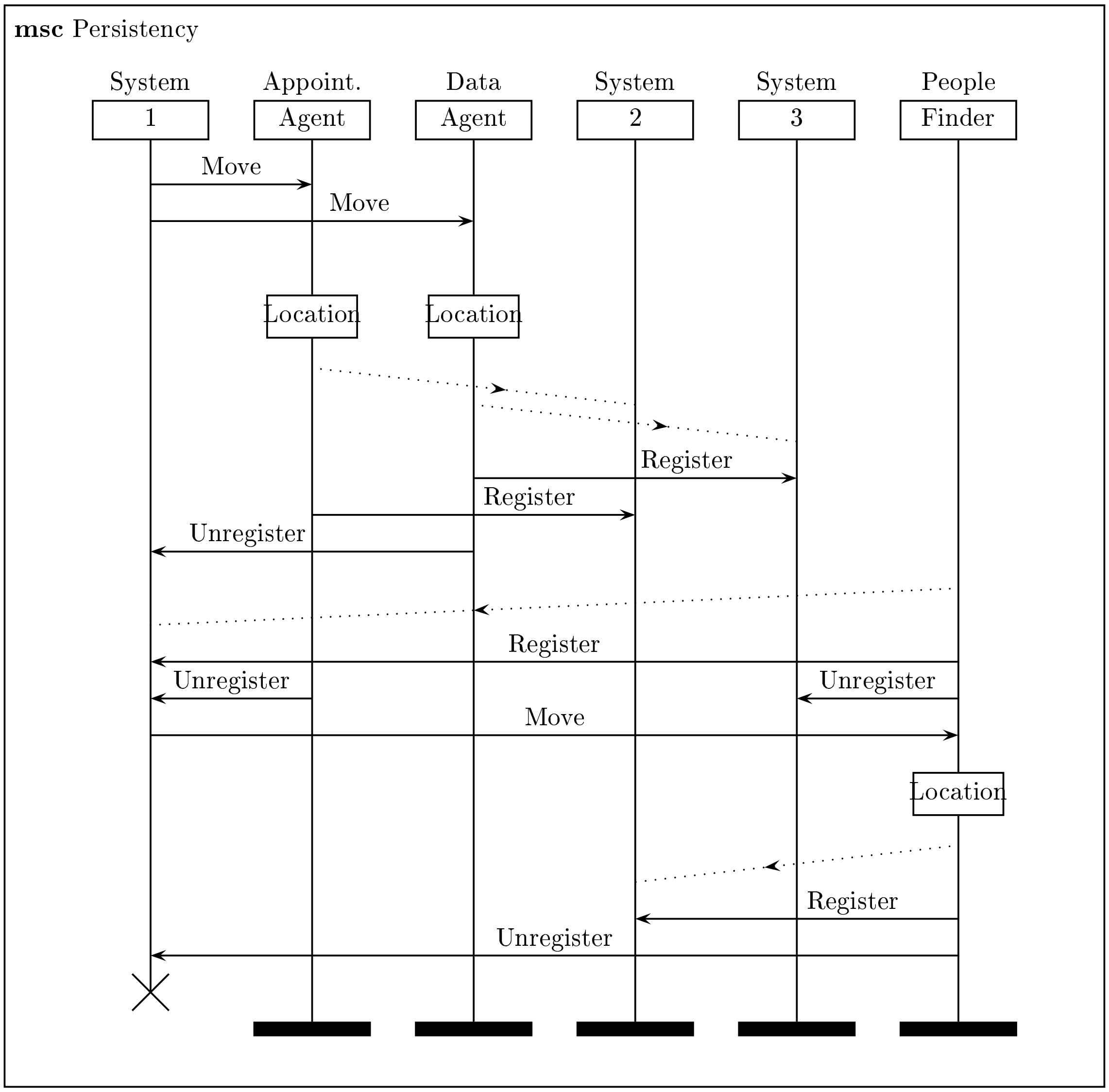

Figure 6.17: Persistency

All agents residing on the system that wants to shut-down are ordered to leave. Because they have to move it is an ideal time to consider load balancing. Once the agents have moved, they tell the system that they have arrived and notify the other system that they have left.

When the system is still waiting for messages it is still possible that agent migrate to the system, when an agent migrates it first registers at the system. When the system is shutting down it notices that it has received an agent and there it orders the agent to move. Only when all messages are received the system can shut down safely.

Fault tolerance

To ensure persistency the system needs to be robust to faults. At this level the fault that can occur is that an agent is not available. When an agent is not available I consider the agent dead. When a network error is the reason an agent is not available it is possible that the agent returns, when this happens both new agents need to synchronize the contents and one of the agents has to terminate.

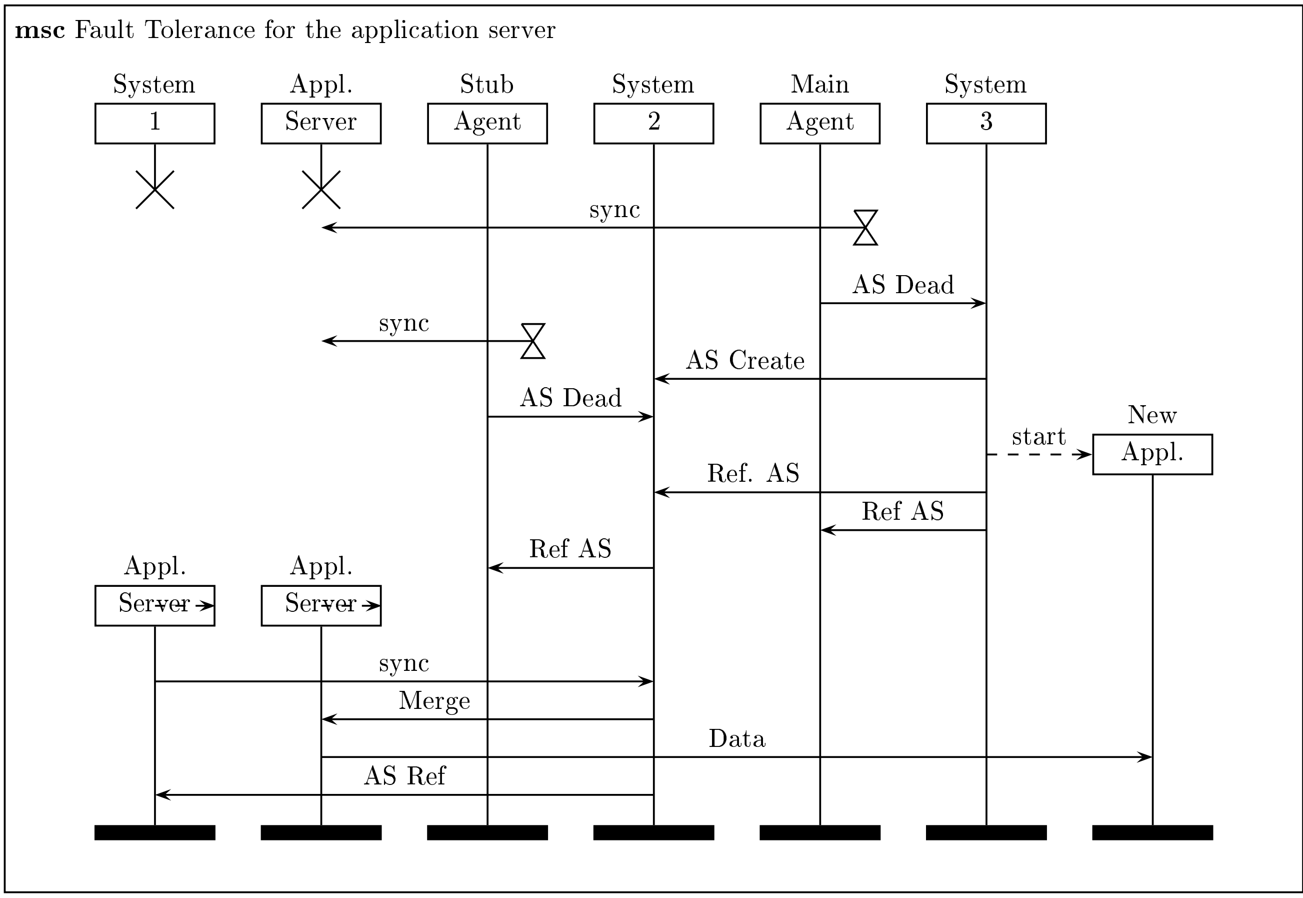

Figure 6.18: Fault Tolerance for the Application Server

The fault tolerance measures are best explained with the application server agent (see figure 6.18 on page 58). All agents need to communicate with other agents, some of those other agent rely heavily on this communication. These agents are the agents that most likely discover the unavailability of the killed agent.

For the application server agent the agents that rely on communication are the main agent and the stub agents. At some, arbitrary, points those agents synchronize with the application server agent, when this synchronization times out the agent that requested the synchronization considers the agent dead.

When an agent thinks the application server agent is dead it notifies the system where it resides on. The system will first notify the other systems that it will create a new application server agent. If one of the other systems receives messages that the application agent is dead, the system can simply ignore those messages, but it has to remember which agents notified the disappearance. If another system is creating an application server agent some selection mechanism can be used to decide who is making the new application server agent.

When the new application server agent is created, the system that created the application server agent sends the new reference to all systems. It also notifies the agents that reported the new reference to the agents that reported the disappearance or requested the reference of the application server agent. All other systems do the same.

When the old application server suddenly comes online, the contents of one of the agents need to by synchronized. This is a delicate task (see figure 6.19 on 60) because when a network fault occurs a great deal of agents are considered dead. All these agents will be recreated so that the application can still continue.

When the surviving agent merges the contents, it has to check which agents are are redundant. Those redundant agents have to be terminated. Together with this termination, all other agents that uses the agent have to be notified of the change in reference.

6.2.4 Load Balancing

The load balancing scheme used here (see figure 6.20 on page 60) is very simple. When the resources on the system are overloaded it just informs the agent that uses to much resources to move.

The decision where to move is not shown, this does not relate much to what is said here, but it is worth noting that sever load balancing algorithms could be made like load balancing algorithms

- based on communication;
- based on memory usage;
- based on CPU usage;
- based on a combination of the above;
- based the load of all systems.

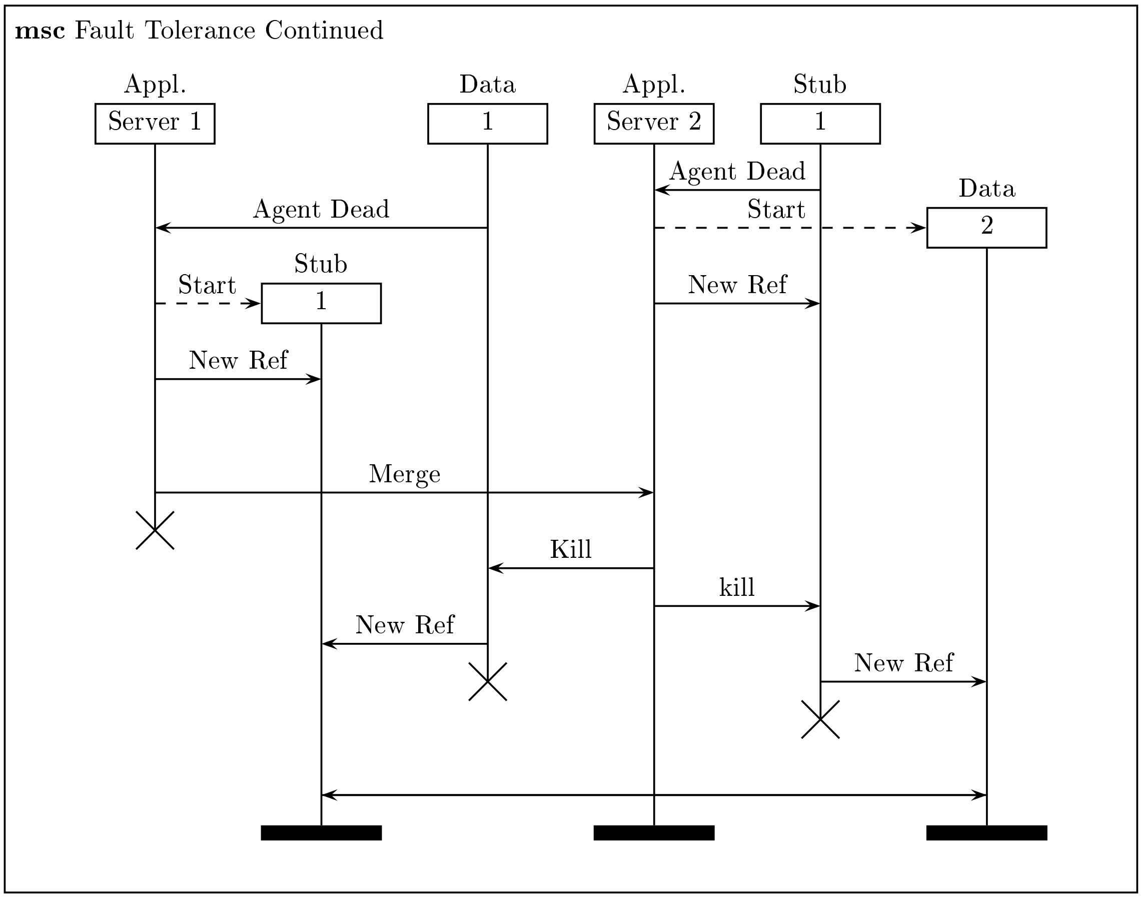

Figure 6.19: Fault Tolerance for the Application Server Continued

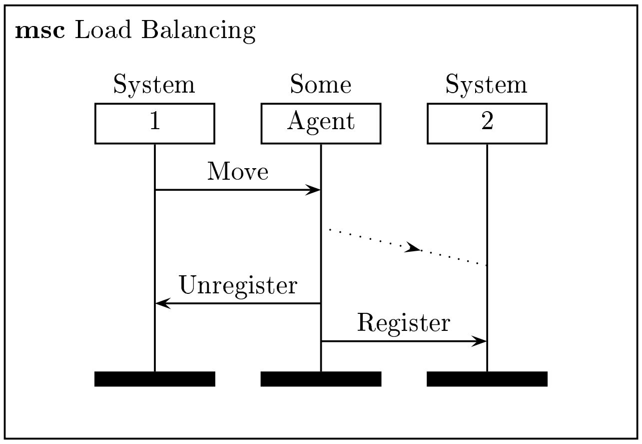

Figure 6.20: Load Ballancing

Chapter 7

Conclusions

7.1 Application Security

For application security I assume that agents are secure, this means that everything in the agent is not available for other agents unless this is available when asked for, using the appropriate communication channels.

In this scheme I make a distinction between data gent, the agents that hold the data, and application stub-agents, the agents that do the computations. The data agents are protected from each other using the stub-agent. This stub agent acquires all desired data from both agents and computes the result.

To be secure when both agents want to exchange data, the stub agent has to be manufactured by a third party that both agent owners trust. If the stub-agent is made by one of the data agent owners the stub-agent could leak data to the data agent. Examples of this are the people finder application and the appointment maker application.

When the stub agent is used in only one direction, meaning one agents that serves as data server and another agent that only acquires data, the stub-agent can be made by the data server owner. In this case that data server agents can be implemented to serve as a complete application. An example of this is the way the papers are distributed.

7.2 Roaming

7.2.1 User Roaming

User roaming is implemented using a clone of the actual data stored on the portable device. This clone is used to acquire data for the user, when the original agent is available for communication both agents synchronize their content.

The advantages of user roaming are shown with the people finder application and appointment maker application, where the cloned agents are used to exchange data.

Some of the aspects, like automatic receiving messages, of user roaming are not shown in this dissertation because the are inherently connected to the asynchronous communication.

7.2.2 Application Roaming

Application roaming uses the mobility of the agents. Agents that from an application are send into the system where it can start to work.

In this case the application roaming is showed using the paper example, where a paper is divided over several agents. Agents that require that paper are free to use the stub agent.

Remote Management

When data is changed by the owner of that data, new data agents are made, references to the old data are changed to the new data and the agents that contain the old data can be killed.

This is a transparent way to change data without interference for the application.

7.3 Reliability

7.3.1 Persistency

Persistency in ensured using the application server agent and the automatic version system agent. These agents contain references to the agents that are being used by one agent. In case of a failure these agents know what agents are missing and they can recreate those agents.

Persistency is also maintained while parts of the systems are going down. In this case it is only necessary to move the agents away from the system that is going down.

7.3.2 Fault Tolerance

When agents are recreated there is a problem to know with whom those agents should communicate. To solve this references are distributed so they can be recovered when an agent is dead.

In this case there are two agents that store all agent references used for their application. The application server agent and the automatic version system agent. When data agents are recreated, those data agents inform the application server or automatic version system that they are now to be used as reference. All agents that use the old reference will be notified of the change.

When the application server agent or the automatic version system agent dies it can be recreated from all the other agents. These agents send there references to the new agent.

It should be clear by now that this scheme is not 100% fault tolerant. When When some data agents or other application agents die together with the application server agent it is not possible anymore to recreate all lost agents.

7.4 Load Balancing

Mobility is very useful for load balancing. Agents can be migrated when the system is to busy. Although the load balancing scheme used here is very primitive, it keeps the systems from being chocked with to much work.

There is however a small problem with roaming and load balancing. When users stop roaming all agents that have been created for the user have to be killed, this is a pretty straightforward task when references are kept in combination with the agent.

7.5 Final Conclusions

Before I started the dissertation I found that moving agents where just for fun, it was nice to see them hop from one system to another system to continue working there, but that was it. It might be useful in distribute artificial intelligent, but for distributed applications the usability was not shown.

I think that this dissertation shows that it is possible to use mobility for other reasons. The dissertation does not claim to show all possibilities with mobile multi-agents. It only shows that mobile multi-agent systems should be considered as a possibility to solve a problem.

Chapter 8

Future Work

8.1 Security

The case as shown here only takes one version of the applications. This means that every agent uses the same algorithm for the specific problem. It is quiet easy to modify the application server agent so that there can be a request for another algorithm when available.

A disadvantage of application security is that custom made applications are not available. One way to overcome this is to submit the application to the application sever owner. The application server owner can then asses the code of that agent for its security.

An automated version of this system might work. There can be made an agent that analyzes the code. This could be a stub-agent so the actual code is not made public to the application server agent.

Either way there remains a security problem, using the stub-agent does not guaranty that the agent is executing other actions then what it promises it will to execute. On the other hand the stub-agent is a problem too, when both parties do not thrust each other a third party has to manufacture the agent.

Because the stub agent can compute and compare data between the two agents, the stub-agent can also be used to check if one agent is not trying to corrupt another agent. This is not shown in this dissertation.

8.2 Roaming

One form of roaming I did not implement, nor explained is system roaming. When systems are back online they want to share their resources. I did not explain this because the systems need to learn where other systems are, and the agents need to learn that the system is available.

8.2.1 User Roaming

Problems arise when the user starts a roaming session and never comes back, or comes back in long time intervals. This means that several applications and the cloned agent are working on the system for nothing. To avoid this a timeout can be made, killing all the agents that are there to long.

I am afraid that this will remain to be a user problem because starting the roaming requires an action from the user.

8.2.2 Application Roaming

In this dissertation is another use of application roaming not shown. I did not include the ability to migrate the paper to the portable device, so the user of the portable device could use the paper when it was not connected to the network .

Doing this does not impose any security risks, however there cold be problems with the remote management. If the user of the portable device is allowed to make changes to the paper it is not clear how the remote management scheme should react when there are two new versions of that paper.

Remote Management

In this dissertation only one form of remote management is shown. When the owner of the data decides to make the new data available, remote management makes sure that the users of the old data automatically acquire the new data.

Data can change inside the agents that roam. There are situations where the owner of the data only uses a copy of that data. In this case the owner will never be notified if the data changes on that system.

It is not hard to add this feature to this case. When the data has changed in the system the owner of that data is then responsible to add the data to the original version.

8.3 Reliability

8.3.1 Persistency

With data that comes from roaming agents ensuring persistency is done when those agents are roaming again. This can cause some frustrations by users of the data from the roaming users.

In some cases, it might be required to make a backup of various agents so that those agents can be made available again when required.

8.3.2 Fault tolerance

Fault tolerance remains a problem when only a part of the computations and communications are finished. In this case, the computations start from the beginning. This might form a problem when the surviving agent is waiting for a synchronization with the dead agent.

Solving this problem has nothing to do with agents but with the ability of the agent systems.

8.3.3 Load Balancing

The load balancing scheme used in this scheme is rather stupid. It only sends agents away when resources become to busy. In some cases this could lead to systems that are almost not used and other systems that are almost completely used, in this case the load balancing as presented in this dissertation will not work.

When the systems would communicate internally about the state of their resources, the systems could distribute the agents more evenly over all systems.

Other improvements are also possible. Eg when there is one agent using all CPU power and another agent is using all the communication resources. These agents could be together on one system without any problem. In this case both agents will be ordered to move.

Appendix A

Message Sequence Charts

Message sequence charts (MSC) is a visual language for the description of the interaction between different components of a system [1]. MSC can be used to show how the different agent communicate. This communication is always asynchronous. For mobile multi-agents this means that only reactions to send messages can be shown. Some of the visual components of the MSC have to be abused to show the communication between agents.

A.1 Agents

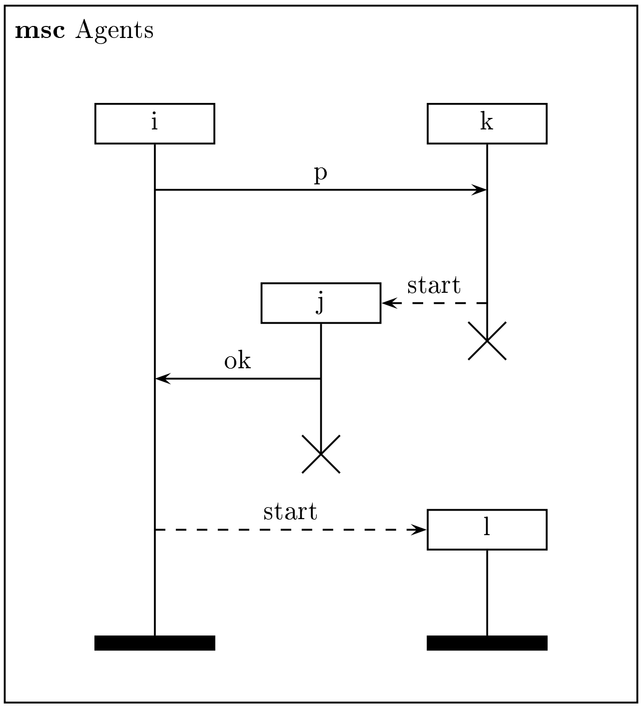

Figure A.1: Agent Creation and Termination

In this msc diagram there are four different agents, i, j, k and l. Agents i and k are agent that have an undefined start point, this means that they could have been doing other communications before this sequence.

Agent k starts an agent j, after which agent k dies. In the mean time agent j is doing computations or other undefined things, as last action it send a message k to agent i (see section A.2) and then dies. After agent i receives this message it starts agent l. The creation of agent l was the last action in this sequence. Agent i and l die at some undefined time in this sequence of actions.

A.2 Communication

The communication shown here is just content of communication. As said in section 3.3.2 on 22 there is a difference between message based communication and information communication. This is not shown in the mcs.

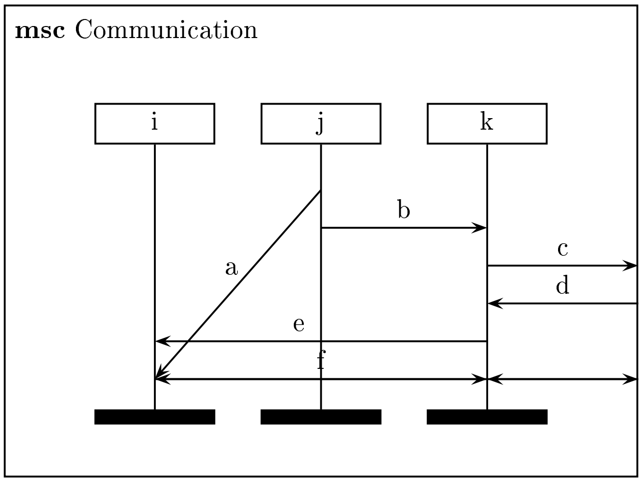

Figure A.2: Possible Messages

In this chart there are three different agents i, j and k that are communicating. Message a is a message that is first send from agent j to agent i. This message will be used by agent i only after message e is processed.

Message a must be send before agent i can send message b to agent k. Messages d and c are output and input to a user.

Message f is used to denote an undefined amount of communication between the agents and the user. This is only used in combination with the user interface, where lots of information is send and received.

For the timed synchronization communication I used the $\bowtie$ symbol. if an a response is possible I immediately return the request. If the request will time out, I do not reply.

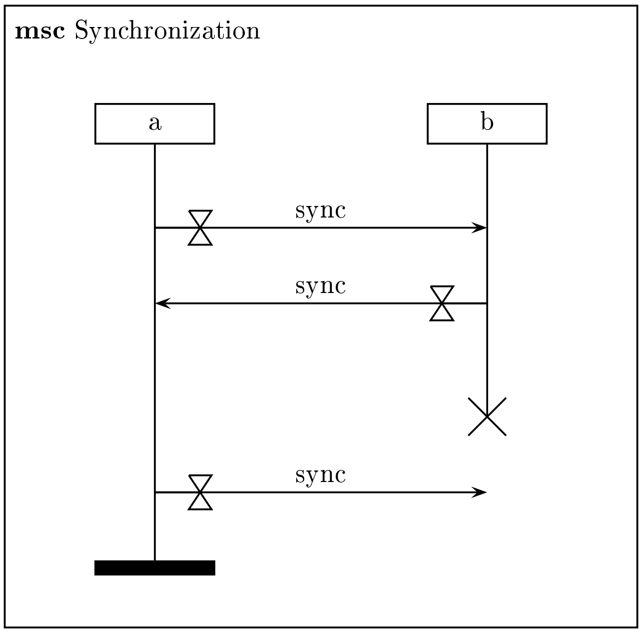

Figure A.3: Timed Synchronization

A.3 Order

In some cases the order in which the messages a received and send is not important to show this the solid line in the agent is changed to a dashed line.

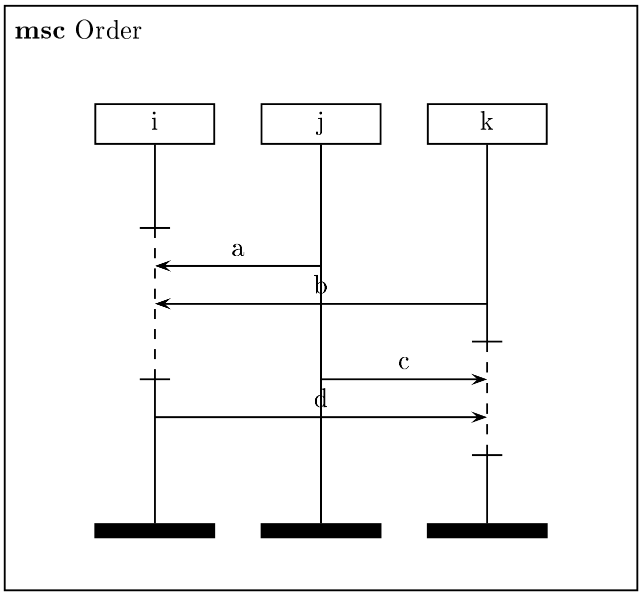

Figure A.4: Undefined Message Order

A.4 Migration

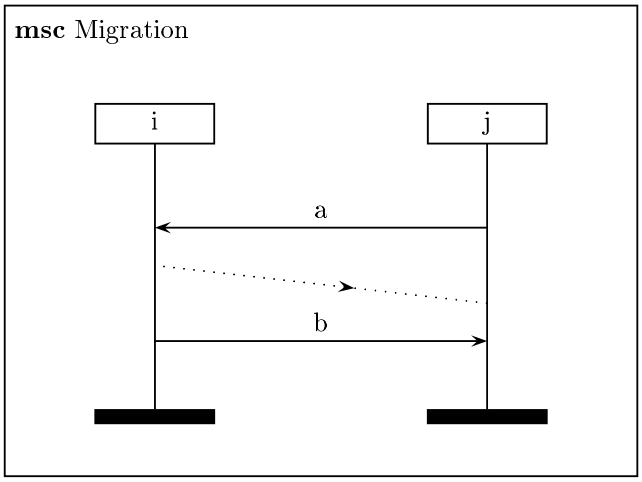

Figure A.5: Agent Migration

This chart shows that agent i migrates to the same place as where agent c is, after it receives a message a. After the migration agent i sends a message to agent j agent i sends a message b.

A.5 Actions

Actions can be invoked when a certain type of message arrives, it is pretty useless for what I use it but it stresses that some action, a computation in this case, is being performed by the agent.

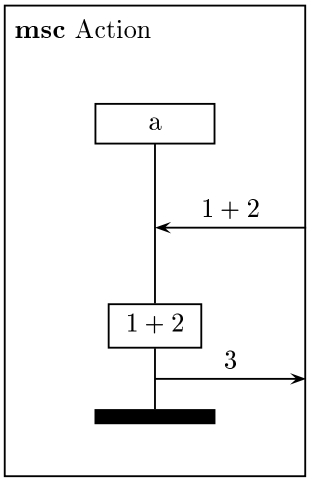

Figure A.6: Actions in Agents

In this case the user asks agent a to compute $1 + 2$. Agent a computes the answer and the result is send back to the user.

A.6 Conditions

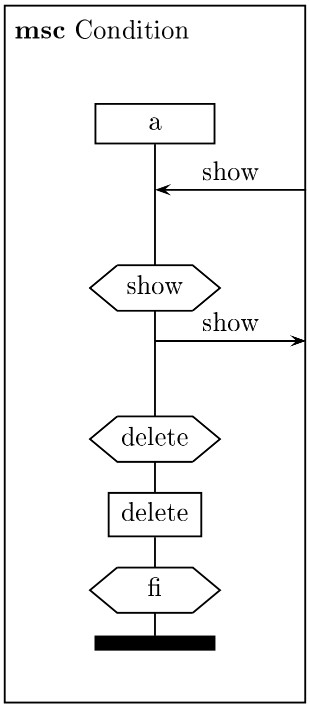

Figure A.7: condition

In some cases there is a message expected, but the content is not know. in this case a condition is used to express this. Agent a receives a message, *shown*, but it could contain *delete* to express the different reactions to the different messages a condition is used.

In case *show* is send to the agent the agent send a *show* message back. In case the message was *delete* the delete action is performed. The normal continuation of the agent is denoted by the condition fi.

Bibliography

[1] V. Boss et.al., *A LaTeX Macro Package for Message Sequence Charts*, Eindhoven University of Technology, Department of Mathematics and Computer Science, The Netherlands, 1999.

[2] T. Budd, *An Introduction to Object-Oriented Programming*, second edition, 1997, Addison-Wesley Longman, Inc.

[3] L. Cardelli, *A Language With Distributed Scope*, Digital Equipment Corporation, Research Center, USA, 1995.

[4] L. Cardelli et. al., *Mobile Ambients*, Digital Equipment Corporation, Research Center, USA, 1997.

[5] A. H. W. Chan et. al., *The Mobile Code Paradigm And Its Security Issues*, Department Of Computer Science and Engineering, The Chinese University of Hong Kong, .

[6] F. Cohen, *Computer Viruses: Theory and Experiment*, Computers and Security, 6, 1987, pp 22-35.

[7] F. L. Drake, *Remote Procedure Calls*, Department of Computer Science, Virginia Tech, USA, 1995.

[8] C. Fournet et. al., *A Calculus of Mobile Agents*, INRIA Rocquencourt, Le Chesnay, 1996.

[9] A. O. Freier, *The SSL Protocol Version 3.0*, Netscape Communications, 1996.

[10] M. R. Genesereth et. al., *Software Agents*, Stanford University, Computer Science Department, Logic Group.

[11] C. Ghezzi et. al., *Mobile Code Paradigms and Technologies: A Case Study*, Dip. Elettronica e Informazione, Politecnico di Milan, 1997.

[12] M. Grivas, *Generic Model for Load Balancing using Agents*, Department of Computer Science, University of Exeter, United Kingdom, 1999.

[13] C. Harrison et. al., *Mobile Agents : Are They A Good Idea?*, IBM Research Division, 1995.

[14] h. Cejtin et.al., *Higher-Order Distributed Objects*, ACM transactions on Programming Languages and Systems, 17(5), pp. 704-739, 1995.

[15] T. D. Hodes et.al., *Composable Ad-hoc Mobile Services for Universal Interaction*, Computer Science Division, University of California at Berkeley, USA, 1997.

[16] F. Hohl, *Time Limited Blackbox Security: Protecting Mobile Agents From Malicious hosts*, Institute of Parallel and Distributed High-Performance Systems (IPVR), University of Stuttgart, Germany, 1998.

[17] D. House, White paper : *How To Enable roaming Users in Windows 2000*, IBM Technical Report - TR 29.3294, 1999.

[18] D. r. Karger et.al., *A better Algorithm for an Ancient Scheduling Problem*, Department of computer Science, Stanford University, USA, 1996.

[19] O. Koskimies et.al., *Partitioning Applications with Agents*, Second International Workshop on Mobile Agents for Telecommunication Applications (MATA2000), pages 79-93.

[20] D. Malkhi et.al., *Secure Execution of Java Applets using a Remote Playground*, IEEE Symposium on Security and Privacy, May 1998, Oakland, California, pp 40-51, 1998.

[21] J. Mayfield et.al., *Evaluation of KQML as an Agent Communication Language*, Computer Science and Electrical Engineering Department, University of Maryland Baltimore County, USA, .

[22] F. g. McCabe et. al., *Asynchronous Messaging*, Fujitsu Laboratories of America Inc., USA, 1998.

[23] A. L. Murphy, *Lime: A Middleware for Physical and Logical Mobility*, University of Rochester, USA, .

[24] D. Milojicic, *Mobile Agent Applications*, IEEE Concurrency, Trend Wars column, July-September 1999, pp. 80–90.

[25] P. D. O'Brian et. al., *FIPA, Towards A Standard In Software Agents*, British Telecom Technology Journal, Vol. 6, No. 3, July 1998, pp. 51–59.

[26] O. Othman et.al., *Strategies for CORBA Middleware-Based Load Balancing*, IEEE Distributed Systems Online, Vol. 2, No. 3, March 2001.

[27] J. S. Plank, *An Overview Of Checkpointing In Uniprocessor And Distributed Systems, Focusing On Implementation And Performance*, Department Of Computer Science, University of Tennessee, Technical Report UT-CS-97-372, 1997.

[28] J. Pockele, *BORG Tutorial*, Programming Technology Lab, Computer Science department, VUB, 2000, http://borg.rave.org/.

[29] U. Saif et. al., *Communication Primitives for Ubiquitous Systems or RPC Considered Harmful*, Computer Laboratory, University of Cambridge, United Kingdom, .

[30] S. Schneider, *Verifying Authentication Protocols in CSP*, IEEE Transactions on Software Engineering, Vol. 24, No. 9, September 1998

[31] N.P. Sudmann, *TACOMA*, University of Tomso, Norway, 1996.

[32] N. Suri et. al., *Am Overview of the NOMADS Mobile Agent System*, , Institute for Human & Machine Cognition, University of West Florida.

[33] A. Tanenbaum, *Computer Networks*, International Edition, Third Edition, 1996, Prentice-Hall International Inc.

[34] A. R. Tripathi et.al., *Mobile Agent Programming in Ajanta*, Department of Computer Science, University of Minnesota, Minneapolis, USA, .

[35] C. F. Tshudin, *Mobile Agent Security*, Intelligent Information Agents of M. Klush (Ed) - Agent based information discovery and management on the Internet, 1999, Springer, pp. 431-445.

[36] W. Van Belle et. al., *Agent Mobility and Reification of Computational State*, Programming Technology Lab, Computer Science Department, Vrije Universiteit Brussel, 2000.

[37] W. Van Belle et. al., *Location Transparent Routing in Mobile Agent Systems* Merging Name lookups With Routing, Programming Technology Lab, Computer Science department, Vrije Universiteit Brussel, .

[38] J. Vitek et. al., *Mobile Computations and Hostile Hosts*, Proceedings of JFLA'99, 1999, Avoriaz.

[39] j. Vitek et.al, *Towards A Calculus Of Secure Mobile Computations*, Centre Universitaire d'Informatique, Université de Genéve, Suisse, .

[40] D. Volpano, *Provably-secure Programming Languages for Remote Evaluation*, Computer Science Department, Naval postgraduate School, Monterey, CA, USA,.

[41] G. Weis, *Multiagent Systems*, A Modern Approach to Distributed Artificial Intelligence, 1999, Massachusetts Institute Of Technology.

[42] M. Weiss, *A gentle introduction to agents and their applications*, http://www.magma.ca/ mrw/agents/