

MAGUS
Mobile Agents Using Sound

Daniel Hägglund

24th April 2003

“There are two kinds of people, those who finish what they start and so on.”

Robert Byrne

Abstract

The mobile agents paradigm has emerged over the last decade. This paradigm is a distributed computing concept based on autonomous mobile objects. It has many advantages over the traditional server client-model, which is predominant today, and is believed to play a significant role in the future of distributed computing.

This thesis studies the feasibility of a mobile agents-based system, which uses sound in order to track and, to some extent, predict the movement of a sound source. A functional prototype of the proposed system has been developed in Java. This prototype clearly demonstrates the achievability of the system and indicates further studies that are needed before it may be of practical use.

Acknowledgements

The work on this thesis was performed from July to December 2002 at the Centre for Distributed Systems and Software Engineering at Monash University in Melbourne, Australia.

First and foremost, I would like to thank my supervisor Arkady Zaslavsky at Monash University for helping me with my work on this thesis and for making it possible for me to spend this time at Monash.

Second, I would like to thank Kåre Synnes, my examiner at Luleå University of Technology for his help with getting everything set up in Luleå and in Melbourne.

I would also like to thank all the administrative staff at both universities for all their invaluable help with the formalities and documents that had to be prepared in order for me to come here.

Finally, I would like to thank Stint and Sven Molin for the accommodation, all the PhD students at Monash for the Friday five o'clock staff meetings, my friend Victoria B. for the creative input and last but not least, my Swedish buddies for their company and especially Per Ekman for the many meaningful discussions on every subject ranging from superheroes to music to sweet and sour chicken.

Melbourne, 5 December 2002.

Daniel Hägglund

Contents

1	Introduction	3
1.1	Purpose	3
1.2	Scope and Demarcations	3
1.3	Outline	4
2	Background	5
2.1	Mobile Agents	5
2.1.1	Advantages of Mobile Agents	6
2.1.2	Standardisation	8
2.1.3	Security	8
2.2	Mobile Agents in Java	11
2.3	Sound in Java	12
2.4	Audio Analysis	12
2.4.1	Sound Matching	12
2.4.2	Speech Recognition	13
3	Method	15
3.1	System Description	15
3.1.1	Recording Sound	17
3.1.2	Identifying Sound	17
3.1.3	Discerning Nearest Listener	17
3.1.4	Activating Listeners	18
3.2	Development Model	19
4	Implementation	21
4.1	Initial Design	21
4.2	Problems	22

<i>CONTENTS</i>	1
4.3 Features	24
5 Summary	27
5.1 Evaluation	27
5.2 Future Work	28
5.3 Conclusion	28

Chapter 1

Introduction

1.1 Purpose

The purpose of this project is to consider a distributed system of agents that tracks a specific sound, moving through the network as it does so in order to always stay close to the source. The goals of the project are:

- To ascertain the feasibility of the proposed system.
- To design and implement a prototype of the system.
- To evaluate the prototype.

1.2 Scope and Demarcations

The project is intended as a proof of concept. Hence, possible uses of the proposed system are not explored in depth. For the same reason, security and privacy issues will not be addressed in the prototype itself but will be considered briefly in the summary. Furthermore, the latency of the networks the prototype will operate in will be assumed to be near zero; consequently, the prototype will not be optimised for low bandwidth.

Since this project is on the subject of software engineering, not signal processing, existing techniques for audio analysis will be utilised rather than developing new ones specifically for this project.

1.3 Outline

In chapter 2, topics related to the project are presented. In chapter 3, a description of the project is given. Chapter 4 documents the development of the prototype. Finally, in chapter 5, the project is summarised and future work is outlined.

Chapter 2

Background

This section gives an overview of mobile agents in general and in Java as well as a brief account of Java's sound capabilities, all related in some way to the Mobile Agents Using Sound (MAGUS) project.

2.1 Mobile Agents

The most popular concept for distributed computing today is the client/server paradigm. However, with rising demands on processing power and the need to conserve bandwidth on large, slow networks, the flaws of this approach have become obvious. In its place, several new approaches have appeared. Possibly the most interesting among them is the mobile agents paradigm. Mobile agents are a special case of mobile code, i.e. processes that can move from one host to another and resume execution at the new host without actually restarting. There is no exact and final definition of what a mobile agent is, but in an attempt to classify autonomous agents, a general definition of is provided in [5]:

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

From this definition, some properties all agents have in common can be obtained.

- Agents are autonomous, meaning that agents decide their own course of action, within the bounds of the program in the context of software agents.

- Agents are goal oriented; they have specific goals that they try to accomplish.
- Agents are reactive, sensing their environment and acting on it to achieve the previously mentioned goals.
- Agents are temporally continuous, meaning they execute continuously over time.

Needless to say, mobile agents have the additional property of mobility, enabling them to relocate themselves between different hosts on a network.

2.1.1 Advantages of Mobile Agents

Together, these properties give mobile agents some notable benefits over conventional distributed programming paradigms. It is much thanks to these benefits mobile agents have been studied so meticulously over the last few years. The most significant advantages of mobile agents are the following:

- Reduction of network traffic. When a distributed application needs to interact with a remote server several times the network load sometimes becomes very high. This is especially true when some sort of security scheme is used to ensure only authorised access, increasing the required number of interactions between server and client. Mobile agents solve this simply by moving the computations to the data instead of moving the data to the computations (fig 2.1). [14, 26, 15]
- Latency protection. Real-time systems may be sensitive to network latency and in critical systems delays of commands may cause severe and irreversible damage. Since a mobile agent would control such a system locally, latency because of slow or failing network connections is nonexistent. [14, 15]
- Asynchronous and autonomous execution. Mobile agents run autonomously; hence, they can be deployed and left to their own devices until they have completed their task. This can be of great value when network connections are precarious or expensive, as is the case with most mobile devices of today. A mobile device might want to deploy an agent and disconnect, eventually reconnecting to collect the agent and its result (fig 2.2). [14, 15, 6, 26]
- Dynamic adaptation. Being able to sense their environment, mobile agents can adapt dynamically to the present conditions, moving to a more favourable position or cloning themselves to distribute the workload over several hosts. [14, 15]

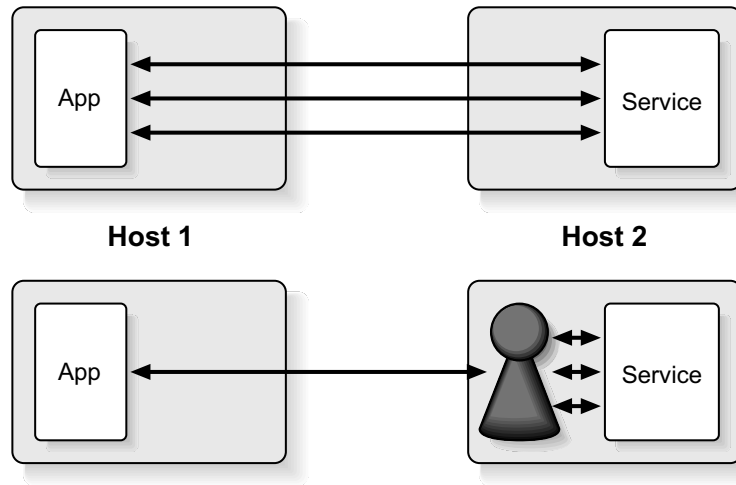


Figure 2.1: Network load when using remote procedure calling (top) in comparison to mobile agents (bottom).

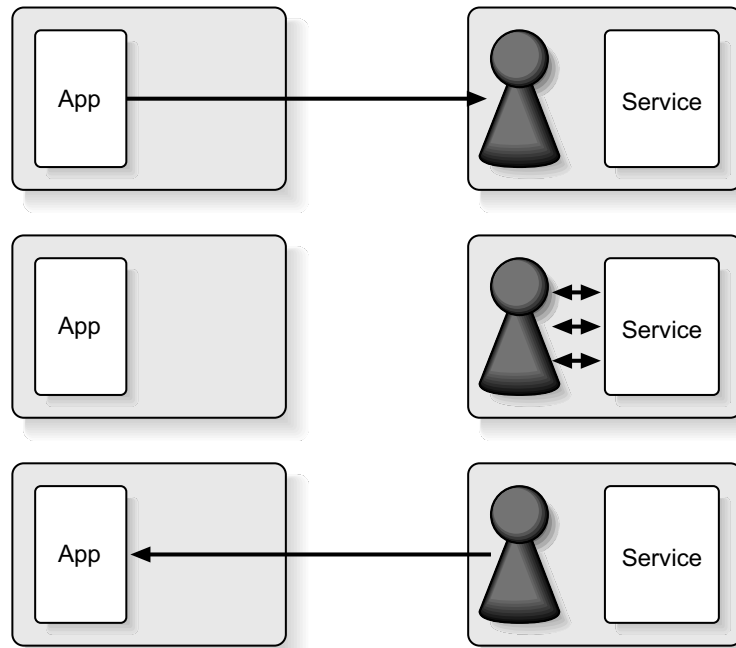


Figure 2.2: Illustration of a mobile agent's autonomous execution. First, the application sends the agent to a remote host. The application then disconnects from the network while the agent interacts with the service locally. Finally, the application reconnects and the agent returns with the results.

- **Heterogeneity.** Distributed systems often run in a heterogeneous environment, in respect to both hardware and software. Mobile agents are generally fully platform independent and hence optimal for such environments. [14, 15]
- **Fault tolerance.** Mobile agents make it easier to build more fault tolerant distributed systems. For instance, in case a host needs to be shut down, agents may move from that host or several clones of the same agent can run simultaneously on different hosts in case of a crash. [14, 15, 26]
- **Improved performance.** On slow networks, or when many small interactions are needed, mobile agents are typically associated with increased performance over client/server solutions. [6, 24] On fast networks, however, moving agents back and forth is generally more costly than using remote procedure calls [7, 22].

2.1.2 Standardisation

The need for standardisation of mobile agents' interfaces has become clear over the last few years; after all, mobile agents are of no use if they cannot readily communicate with each other and with the hosts they visit. For this purpose the Mobile Agents Systems Interoperability Facility (MASIF) [18] standard has been developed by a number of parties and presented by the Open Management Group. MASIF specifies an interface for mobile agents developed by different teams to enable them to effortlessly communicate with each other. MASIF is principally aimed at mobile agent platforms developed in Java. MASIF defines the mobile agent environment as, from top to bottom, regions, agent systems and places. A *region* (fig 2.3) is a group of agent systems, not necessarily on the same host. Agents may move and communicate freely within a region, and can move between regions. An *agent system* (fig 2.4) runs on a single host, although a host may have several agent systems running simultaneously. Each agent system contain one or more *places*. Each place may provide different services to agents. Agents are always in one single place at a time.

2.1.3 Security

Mobile agent security is an important part of mobile agents research. Since a computer virus is in essence a malicious mobile agent, the need for both internal and external security schemes is palpable. In contrast to the current situation, where hosts have to

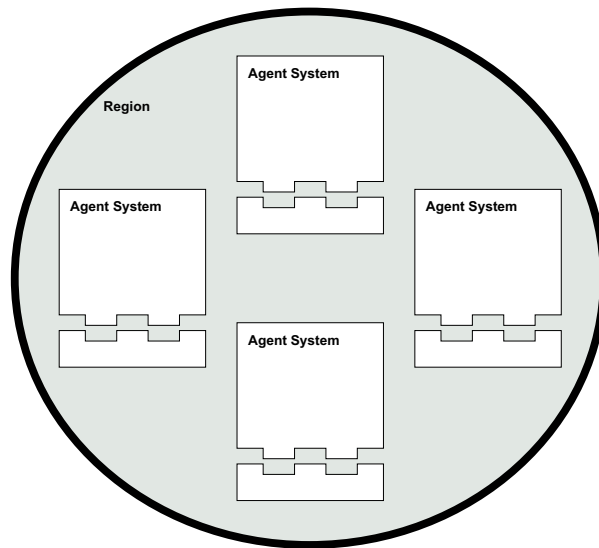


Figure 2.3: MASIF region concept.

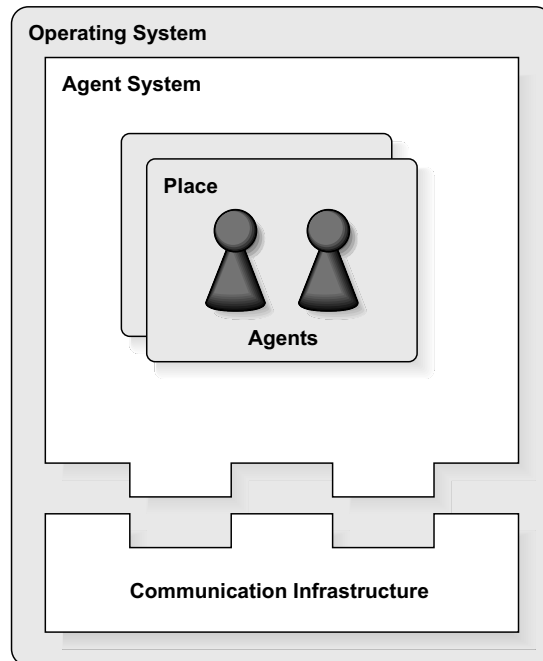


Figure 2.4: MASIF agent system with places, agents and communication infrastructure.

be protected against malicious interference from viruses, mobile agents introduce the additional needs to be protected from malign hosts and other agents. This introduces a number of issues that have to be addressed.

Some obvious threats and countermeasures are presented in [27]. On one hand, the agents have to be secure from attacks from an in all probability unsafe environment. First, a mobile agent's code may be modified before being instantiated and dispatched, thus changing its behaviour in a possible malevolent manner. Second, an agent may be intercepted during migration over insecure networks and its data, which may well be confidential, could be disclosed to harmful parties. Moreover, a mobile agent's code might be altered during migration. The destination of an agent may be changed so as to expose valuable information. A malicious host may trick an agent, not providing the expected services or changing it in the same way described above while inactive or during migration.

On the other hand, the agents are not the only ones in need of protection; the environment needs protecting from malicious agents as well. For instance, the hosts must be sure an agent is from where it says it is, and that it will do no harm to the host. Furthermore, hosts have to be alert so an agent does not perform any actions outside its jurisdiction. The following countermeasures are then presented:

- Authentication. Lets an agent verify it is at the right host, or vice versa.
- Authorisation. Makes sure an agent does not perform any undesirable actions by giving it a certain permission depending on its origin and purpose.
- Confidentiality. Encryption can help keep an agent safe during migration between trusted hosts or during storage.
- Integrity. Ensures the agent has not been tampered with.
- Logging. May detect and prevent agents being cheated by hosts.

Although these methods provide some security to mobile agents, there are a few unresolved issues and some that may, in fact, be unattainable [4]. For instance, keeping an agent's code or data secret may well be futile. Since the code has to be accessible by the hosts it visits, encryption serves little or no purpose unless the hosts can be trusted completely; something that is somewhat unrealistic considering the hosts may provide the same competing services. Furthermore, since the agent cannot carry any

private keys with it, collected data can neither be encrypted with public nor private key encryption if that particular data needs to be consulted further before the agent returns home.

2.2 Mobile Agents in Java

The increasing popularity of mobile agents has led to the development of several programming languages specifically designed with mobile agents in mind. Telescript [25] is perhaps the first and most well known example of such a language. Nevertheless, Java is currently the number one choice of mobile agents developers, despite its lack of mobile agents tools.

It is Java's characteristics that makes it especially favourable for developing mobile agents; it is inherently platform independent and a de facto standard in platform independent computing, it provides strong typing and it allows data to be transported smoothly over networks via its data serialisation mechanism [7, 26]. Although Java does not provide any innate mobile agent functionality, there are a number of tools and platforms extending the Java language for this particular purpose. IBM's Aglets [19] and IKV's Grasshopper [10] are among the most well known examples of such packages. Although far from all mobile agent platforms for Java conforms to MASIF, the features and the functionality they offer are very similar to those that do. The typical mobile agents toolkit extends the Java language with the following features:

- Creation of objects on remote hosts.
- Both synchronous and asynchronous server/client-type communication between agents on different hosts similar to Java's intrinsic remote method invocation.
- Weak migration between hosts on the network. Weak migration implies that the *data state* of the agent, i.e. its internal variables, is preserved. In contrast, strong migration keeps the agent's entire *execution state*, i.e. its program counter and frame stack, intact while migrating. Java-based mobile agents do not offer strong migration due to restrictions in the Java language. [11, 18]
- Basic security features are present in virtually all mobile agent platforms intended for widespread use.

2.3 Sound in Java

Java has incorporated sound functionality since release 1.3. Recording and playback as well as streaming of multiple audio formats are supported either through the Java Sound API [13] or the higher level Java Media Framework (JMF) [12].

Java Sound is a framework for recording, playing and processing sound within Java. It supports a wide range of formats, both 8 and 16 bits and sample rates from 8 to 48 kHz. Java Sound is relatively low level and is thus useful for sound processing applications. For instance, it is straightforward to record a sound and present it as a byte array, suitable for FFT or other forms of analysis.

JMF is also designed for use with time-based media content such as audio and video, and it too has methods for capturing and playback. However, the emphasis is on real-time streaming and presentation of media content rather than analysis.

2.4 Audio Analysis

There are two main categories of audio analysis that could be used in a system such as MAGUS; sound matching and speech recognition.

2.4.1 Sound Matching

Sound matching is a concept where a sound is compared to known sounds to determine how similar they are. Sound matching can use both physical properties of the sound, such as amplitude and frequency, and psycho-acoustical ones, such as onset and offset. Applications of sound matching range from automatic violence detection for films [20] to automatic detection of exciting parts in sports shows [17] to surveillance systems [3]. Sound matching could be used in MAGUS to detect sounds such as clapping hands or footsteps. Much research has been made on the subject and there are some techniques that could without doubt be useful in this project.

- Spevak and Polfreman's sound spotting approach described in [21], which uses mel-frequency cepstral coefficients for feature extraction, commonly employed in speech recognition. Subsequently, self-organizing maps, a particular type of neural networks, is used for classification and, finally, k-difference inexact matching, a string matching algorithm, is used for pattern matching.

- Pfeiffer et al's audio content analysis [20], primarily designed for automatic violence detection in films, classifies sounds using both physical properties, such as amplitude and frequency, and psycho-acoustical properties, for instance onset and offset.
- Comparisionics' commercially available sound-matching technology [2]. Since the system is commercial, its design is secret. However, its functionality is essentially equivalent to Spevak and Polfreman's method, creating signatures from audio content and comparing them to determine similarity.

2.4.2 Speech Recognition

Thanks to the commercial value of speech recognition, it has been a popular research subject for quite some time. Speech recognition can be divided into two types; dictation recognition and command recognition. Dictation recognition is designed to recognise a vast range of words and is used to quickly produce large quantities of text in a more natural way than typing. Dictation recognises many different words. As a result, it is not very accurate. The other type of speech recognition, command recognition, is used to parse commands and can be used in a speech user interface, replacing or enhancing a traditional user interface. Command recognition merely recognises certain pre-determined words, but with a much higher accuracy than dictation recognition.

Speech recognition is available in Java through the Java Speech API (JSAPI) [23]. JSAPI defines interfaces for both speech recognition and synthesis. As with all other Java technologies, it is very high level and aims at complete platform independence. There is no standard JSAPI speech engine. Instead, it relies on third-party software to implement the API. Currently, there are a few free open source speech synthesis engines written entirely in Java. Speech recognition engines, in contrast, are normally commercial, written in native code and hence platform dependent. Examples of available JSAPI implementations are Cloud Garden's JSAPI [1], which can run on top of several different speech engines, and IBM's "Speech for Java" [8], using IBM's own ViaVoice [9].

Chapter 3

Method

In this chapter the concept of the system is presented and different approaches that could be taken and areas in need of special attention are highlighted.

3.1 System Description

The concept of the entire system is a distributed system that, by means of sound only, tracks a sound source and follows it through the network as it moves. The source could be a person, a machine or anything else that is capable of emitting sound. The concept is visualised as a series of illustrations in figure 3.1. In image one, the system is set-up in an environment with nine hosts, A to I. The only host that is actively listening for sounds is H. A person enters the room at H, which H identifies. H then activates listeners on the hosts adjacent to it, i.e. E, G and I (in image two). The tracked person now moves towards G. When G detects a sound, it activates D. The person moves towards D, which activates A (image three), and finally to E, which will activate B and F (image four). Hence, there are essentially four properties of the system:

- Record sounds.
- Identify sought sound.
- Determine which listener is closest to the sound source.
- Activate listeners on hosts adjacent to the nearest listener.

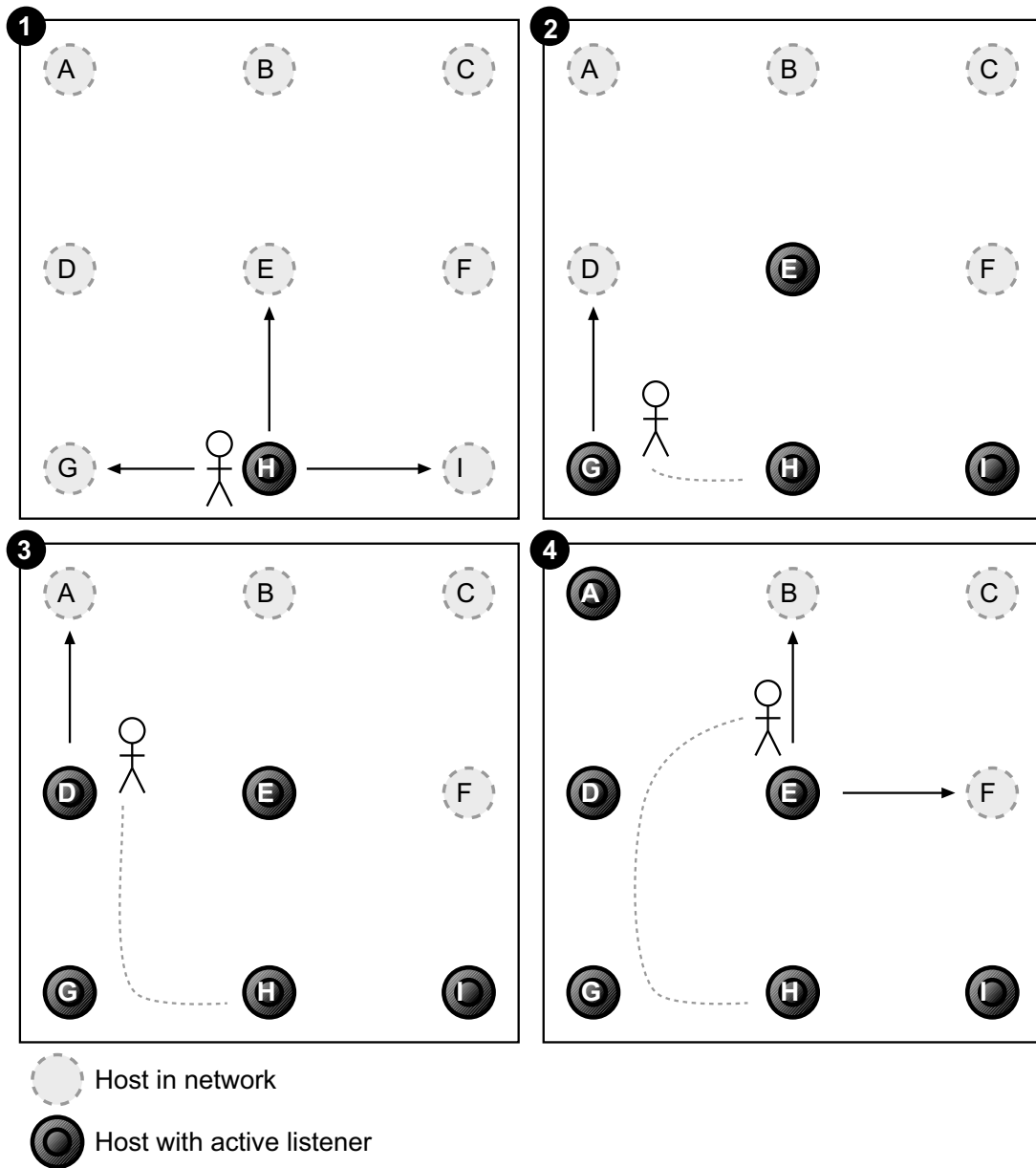


Figure 3.1: Illustration of the system concept.

3.1.1 Recording Sound

To record a sound, MAGUS needs to have access to a microphone at each host. Java provides methods to record sounds with available microphones, as mentioned in 2.3.

3.1.2 Identifying Sound

Possible ways to identify sounds are discussed in 2.4. The most flexible option would be to have sound matching capabilities, in order to utilise sounds such as footsteps or claps. A speech recognition engine could also be used for this purpose, but would partially limit the system. Whichever option is used, the sound can be identified either at the listeners themselves or at a central server. The former requires more powerful hosts in the entire network, all with sound matching or speech recognition capabilities while the server making the decision where to move does not have to be particularly powerful. In addition, this option would conserve bandwidth since no sound data would have to be transmitted over the network, but the listeners would almost certainly be larger, requiring more bandwidth when migrating over the network. The latter option, on the other hand, requires very little computing on the listeners. Virtually the only thing they would have to do is detect when a sound has been recorded. A volume threshold could facilitate this; all sound under a particular volume would simply be considered ambient noise, and all sound above it would be sent to the server for further processing. Hence, the listeners could probably be kept very small in size. The server, however, would probably need a fair amount of processing power to handle all sound data being sent to it from listeners. The network would also be burdened heavier. A combination of the two could also do the job, if the listeners make a very rough analysis of what they detect, for instance checking that the detected sound's frequency is in the right range, before sending it to the server for final processing the bandwidth usage could probably be reduced dramatically while still keeping the listeners fairly small in size and complexity.

3.1.3 Discerning Nearest Listener

Determining which of the listeners is closest to the sound source could be fairly difficult. First, information has to be extracted from the audio data that could be used for this purpose. Volume level is one option, assuming all microphones are set to the same recording level. The volume of a sound is, of course, affected by other factors

than the distance to the source. For instance, the acoustics of the room or which way a speaker is facing when listening for speech could have possibly huge impacts on the perceived volume of the sounds. Another method would be to register the time a sound is heard; obviously, the shorter a sound has to travel, the quicker it will get there implying the time of detection could be used as an indication of which listener was closest to a particular sound. However, as in the case of volume, acoustics would unquestionably affect this. In addition, comparing recording times requires the listeners' timers to be perfectly synchronised, which would be a very optimistic assumption to make. Once this information has been obtained, it has to be passed on to the server for additional analysis, unless, of course, the audio data is sent in whole to the server for it to extract the required information. Sending the information to the server could be achieved either by conventional message passing or by moving listener agents to the server. Considering that the agents would have to move back to their original locations, that the amount of data concerned is small and that the network the system will run on is expected to be relatively fast, moving agents back and forth is probably not justifiable [7, 22].

3.1.4 Activating Listeners

Activating listeners on other hosts is a simple task using a mobile agents platform, either create the agents remotely or create them locally and move them to the desired destination. However, to be able to activate listeners on adjacent hosts, the system has to know which hosts are adjacent physically. The most straightforward method by far is to have a static network structure and configure the system manually at initiation. Automatic configuration requires some mechanism by which the system can determine which hosts are physically close to each other. One possibility is to ping the entire network. However, pinging is normally not accurate enough. Besides, even if it were exact, there is no real guarantee that hosts close to each other on the network are also close physically. Another possible solution would be to have a location-aware protocol, for instance GRID [16], that would enable the system to automatically find hosts that are physically close to each other on the network. A major weakness of this method is the extremely high resolution positioning required. Furthermore, most such protocols are designed for use with GPS, which does not work well indoors. A third option would be to let the listeners emit sounds once in a while and using this sound to triangulate the position of a new host. However, this requires that each listener is in

hearing range of at least three other listeners, creating a need for a very high density of listeners. In addition, the complexity of the system increases.

3.2 Development Model

Because both the concept of this project and the technology utilised was unfamiliar to me, the development model found most suitable was an evolutionary prototyping model, in which prototypes were created first with a very basic functionality and then extended into increasingly complicated systems. The main advantage of this approach to the well-known waterfall model, for instance, is that, as I initially knew neither the full potential nor the limitations of the technologies used, the design could be revised and remade as the tools became more familiar.

Chapter 4

Implementation

This chapter documents the implementation stage of the project. It shows the initial design and how it was revised to what eventually became the final prototype of the system as well as the final prototype's features and restrictions.

4.1 Initial Design

The first design of the prototype was very rough. I envisioned a system that used some form of sound matching to recognise sounds such as handclaps and a mobile agents toolkit to move to the host closest to the sound source. The preferred solutions were:

- Recording with Java Sound. Java Sound can provide sounds as a byte array that is practical for further processing.
- Identification with external sound matching. The idea here was to send the recorded sounds as an array of bytes to a sound matching method, which would return the similarity of the sounds.
- Nearest listener determined by volume of recorded sound. The volume of a sound is very easy to extract, and setting the microphones to the same volume seems easier than synchronising the different hosts' clocks.
- Activation of listeners at new hosts with the Grasshopper mobile agents platform.

This initial design is based on two main classes, one for the server and one for the listeners.

The server class has five tasks. First, it should know the physical location of all hosts in the network. This should be set up manually at start. Second, it starts the first listener agent. Third, it should receive the volume of detected sounds from the listeners, starting a timer when the first volume is received so that more listeners may report successful detection. This can be done with Grasshopper's inbuilt communication mechanism using proxy objects for remote method invocation. Next, it should compare the volume of all received detections to determine which listener was closest to the sound source. Finally, it should activate the nearest listener and all listeners adjacent to that one, using the location information to decide which hosts to contact.

The listener class has the following features. Upon activation, it should acquire a microphone. It should then start recording and analysing all sounds it picks up. When the wanted sound is detected, it should send the volume of that sound to the server and deactivate until further notice.

Figure 4.1 is a sequence diagram demonstrating the initial design. In this scenario, there are three listeners: A, B and C. A is adjacent to B, but not C. B is adjacent to both A and C. At first, B is the nearest listener. Thus, the server activates all three listeners and then waits for any of them to respond. The listeners start execution and when a sound is eventually detected, they send the volume to the server. As the server receives the first report, it starts a timer, waiting for reports from other listeners. When the time is up, the server compares the volumes. In this case, listener A was closest, and the server activates it and B while C remains inactive.

4.2 Problems

During the implementation of the prototype a few problems became apparent. Most of the problems had to do with limitations in the technologies chosen for the project or the fact that I was not particularly accustomed to working with them.

The first problem with the initial design, and the one that influenced the entire project the most, was that there were no packages available in Java capable of sound matching. Since recognising sound is a crucial part of the system's functionality, a speech recognition engine was used instead. Naturally, the prototype could have been developed without speech recognition, using dummy classes to simulate sound matching. This alternative was opted against on the grounds that speech recognition is at least as advanced and demanding as sound matching. Hence, if the system worked

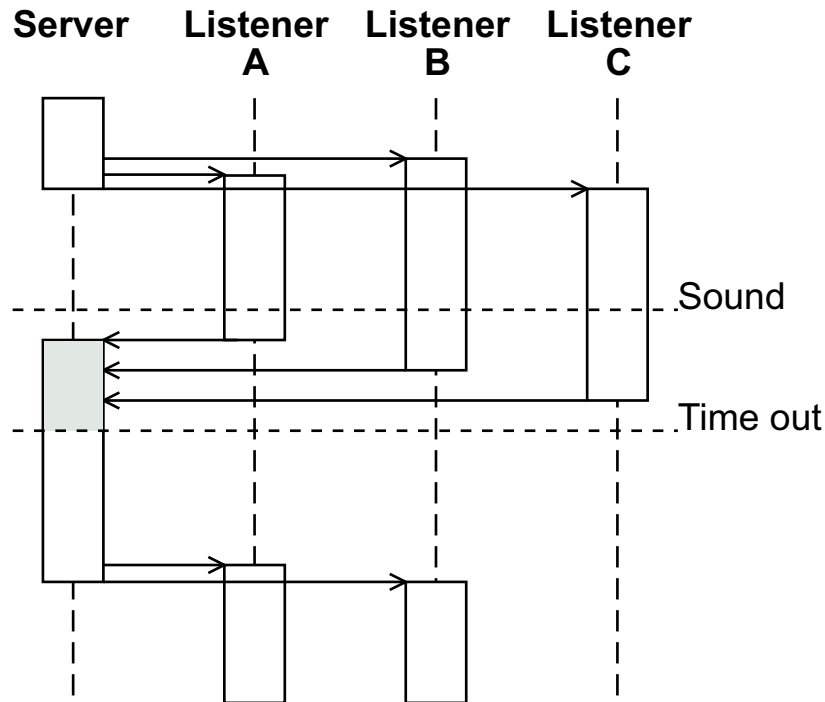


Figure 4.1: Sequence diagram illustrating the initial design concept.

with speech recognition, the feasibility of using sound matching in its place would to all intents and purposes be established.

The second problem was that each microphone could be used by one recogniser at a time only; the cause of this could be the ViaVoice speech recognition engine, limitations of the Java language or possibly the hardware or operating system used with the prototype. As a result, no more than one single MAGUS listener agent could operate on any host at any time. Only one system can run simultaneously as a result.

An additional problem was a minor limitation of Grasshopper. Grasshopper is able to use Java's classloader to move objects to a remote host that does not have the object's class stored locally, which is tremendously useful since the classes of mobile agents do not have to be distributed in advance to the hosts they will eventually visit. However, this mechanism fails when one tries not to move an instance of a class, but to actually create it directly on the remote host. Hence, the listener agents had to be created locally by the server and then move to their final destination. Although this is only a slight inconvenience, together with the impossibility of using one microphone with several listeners described above, it has some consequences. For instance, the listeners cannot

capture the focus of the microphone until it is clear that no other listener is active on the host. To solve this problem, the listeners must verify that they are at their intended location before trying to set up the speech recognition engine, introducing the need for additional safety checks that make the agents more complex and, ultimately, more prone to error.

Although it was not part of the initial plan, sending audio data received from ViaVoice to a central server for processing was not possible. Although audio data can be obtained from Java Speech, its class is not serialisable. Thus, it cannot be sent over the network. Sending audio to the server could be useful if, for example, some advanced processing is needed to determine which listener is closest or if the server would do the actual identification instead of the listeners.

4.3 Features

Because of the problems described earlier, the design had to be revised in several steps during the development process. As a consequence, the final prototype has slightly different features than what was originally intended. The core functionality, however, is the same. The final prototype was made in Java using Grasshopper and ViaVoice. The prototype is reasonably complete; it achieves the goals that were set for it, albeit in other ways than the first design specified. The solutions chosen for the final prototype were:

- Recording is handled by ViaVoice, the voice recognition engine used.
- Identification uses speech recognition performed at the listeners in a distributed manner using ViaVoice.
- Finding the nearest listener is done by comparing volume of sounds using methods provided by Java Speech. Information is sent to the server using conventional client/server methods. Because of imprecise volume information, this method provides a rather rough approximation. However, testing with the speech recognition showed that it would be more a question of which listener detected and identified the correct word than which sensed it at the maximum volume.
- Activation of new hosts using Grasshopper 2. Listeners are created locally by the server then moved to their final destination. Adjacencies are configured manually at start (fig 4.2).

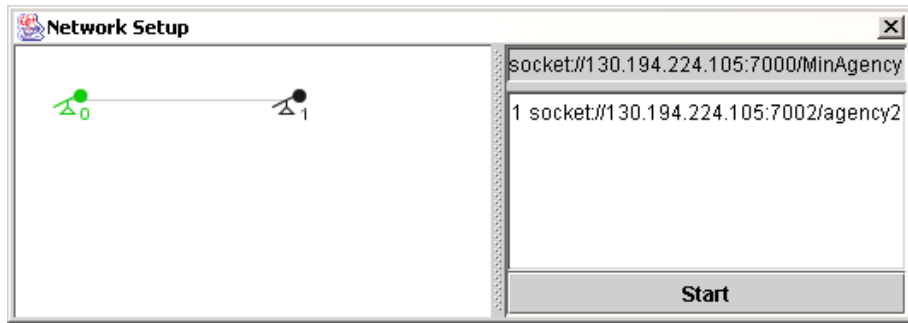


Figure 4.2: Network set-up window.

The system is started by creating a new agent of the MobileServer class in Grasshopper (fig 4.3). Once created, the server shows the network set-up window (fig 4.2), in

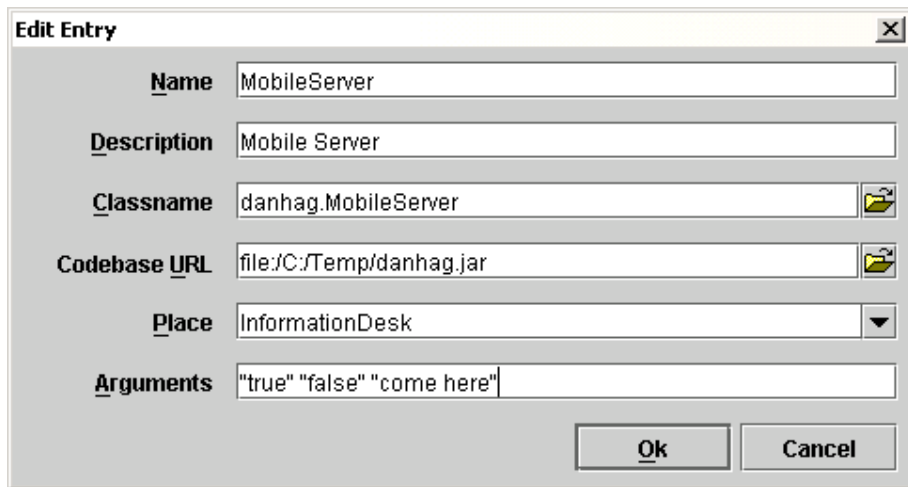


Figure 4.3: Setting up a mobile server in Grasshopper.

which the adjacencies of the hosts in the network are defined. All agent systems in the current region will be shown automatically on the display. Once the adjacencies are satisfactory configured, the server will create one listener at the same host it is currently on. This listener immediately starts listening for the pre-defined activation word. When it recognises this word, it sends the volume of the word to the server. This part of the prototype works as specified in the initial design; the server waits for a short while to allow time for other listeners to inform it of detected sounds. Next, it compares the volumes and activates the nearest listener and all listeners adjacent to it. If there is no listener on a host that is adjacent to the nearest one, a new listener will be

created locally by the server and move to the new host. At this stage, the server itself also moves to the nearest host and resumes execution at that host. This is done to make it clearer which listener actually was closest to the detected sound and has no practical purpose beyond that.

The finished prototype consists of approximately 20 classes and interfaces to a total of around 2,000 lines of code.

Chapter 5

Summary

In this chapter, the work done in the project will be summarised, the prototype will be evaluated, future work will be outlined and a conclusion will be drawn.

5.1 Evaluation

The prototype works and achieves most of the goals set for it. Although it has been tested in only a very small environment, in that environment it can identify specific words, and it can move between the hosts in the network when the speaker moves. As a proof of concept, it is fairly successful. However, for a system such as MAGUS to be not only useful in a practical sense but also justifiable, it must be enhanced in a number of ways. For instance, using mobile agents is somewhat difficult to justify when the structure of the network has to be known in advance. It would undoubtedly be easier to create a distributed system where the listeners are already positioned on all hosts at start than moving agents around the network.

There are two scenarios where mobile agents might be an improvement over a more conventional solution. First, if the hosts on the network can only run a limited number of programs simultaneously, a system where the mobile agents switch off or remove themselves may be beneficial. Since MAGUS would doubtlessly require reasonably powerful hosts to operate, however, this scenario is somewhat unrealistic. The second scenario concerns the use of location aware protocols and ad hoc networks; if the system could detect and incorporate new hosts dynamically, mobile agents would be a clear advantage over a static solution. Moreover, the fact that two systems cannot run simultaneously on the same network limits the system's usefulness.

As far as privacy goes, the security features of most mobile agents platforms, such as authentication and authorisation, suggest that MAGUS would have great difficulties tracking a person that would not want to be tracked. Of course, it might be made more like a computer virus. However, considering the resources such a virus would require in terms of local hardware and network bandwidth, it would probably be fairly effortless to detect and eliminate.

5.2 Future Work

Using sound matching is an obvious extension of this project. It would probably make sense to examine what implementation is the most efficient of the all the ones proposed in this report. For instance, would centralising the audio processing increase the performance, or would the system be hampered by the increased network traffic? The performance of the system compared to a more conventional distributed system needs to be studied to determine if there is any advantage at all to using mobile agents. Finally, developing the system with a location-aware protocol would be advantageous as it would enable the system to run on ad hoc networks and mobile devices and would eliminate the need for manual set up.

5.3 Conclusion

It has been shown that the proposed system is indeed possible to develop using existing techniques and tools, albeit with some minor limitations. Most of these restrictions are introduced because of the technology used in the prototype and it seems likely most of them could be worked around or eliminated with time or by using alternative approaches or technologies. Using a different mobile agents platform or exchanging the speech recognition engine for a sound matching system, for instance, could enhance the system.

Bibliography

- [1] Cloud Garden's Java Speech API, December 3 2002. URL: <http://www.cloudgarden.com/JSAPI/>.
- [2] Comparisonics' homepage, September 27 2002. URL: <http://www.comparisonics.com>.
- [3] M. Cowling and R. Sitte. Sound identification and direction detection in matlab for surveillance applications. In *Proceedings of Australasian MATLAB Users Conference*, November 2000.
- [4] William M. Farmer, Joshua D. Guttman, and Vipin Swarup. Security for mobile agents: Issues and requirements. In *National Information Systems Security Conference (NISSC'96)*, 1996.
- [5] S. Franklin and A. Graesser. Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents. In *Intelligent Agents III. Agent Theories, Architectures and Languages (ATAL'96)*, volume 1193, Berlin, Germany, 1996. Springer-Verlag.
- [6] R.H. Glitho, E. Olougouna, and S. Pierre. Mobile agents and their use for information retrieval: A brief overview and an elaborate case study. *IEEE Network*, 16(1):34–41, 2002.
- [7] D. Hagimont and L. Ismail. A Performance Evaluation of the Mobile Agent Paradigm. In *Proceedings of the 1999 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 306–313. ACM Press, 1999. ISBN 1-58113-238-7.
- [8] Speech for Java, December 3 2002. URL: <http://www.alphaworks.ibm.com/tech/speech>.
- [9] IBM ViaVoice, December 3 2002. URL: <http://www-3.ibm.com/software/speech/>.
- [10] IKV++ GmbH Informations- und Kommunikationssysteme. *Grasshopper Basics And Concepts*, March 2001.

- [11] IKV++ GmbH Informations- und Kommunikationssysteme. *Grasshopper Release 2.2 Programmer's Guide*, March 2001.
- [12] Java Media Framework API, December 5 2002. URL: <http://java.sun.com/products/java-media/jmf/>.
- [13] Java Sound API, December 5 2002. URL: <http://java.sun.com/products/java-media/sound/>.
- [14] Danny B. Lange. Mobile objects and mobile agents: The future of distributed computing? In *The European Conference on Object-Oriented Programming (ECOOP 98)*, pages 1–12. Springer-Verlag, July 1998.
- [15] Danny B. Lange and Mitsuru Oshima. Seven good reasons for mobile agents. *Communications of the ACM*, 42(3):88–89, 1999.
- [16] Wen-Hwa Liao, Jang-Ping Sheu, and Yu-Chee Tseng. GRID: A fully location-aware routing protocol for mobile ad hoc networks. *Telecommunication Systems*, 18(1-3):37–60, 2001.
- [17] Surya Nepal, Uma Srinivasan, and Graham Reynolds. Automatic detection of 'goal' segments in basketball videos. In *Proceedings of the ninth ACM international conference on Multimedia*, pages 261–269. ACM Press, 2001. ISBN 1-58113-394-4.
- [18] Open Management Group, Inc. *Mobile Agent System Interoperability Facilities Specification*, November 1997.
- [19] Mitsuru Oshima, Guenter Karjoth, and Kouichi Ono. *Aglets Specification 1.1 Draft*. IBM Corp., September 1998.
- [20] S. Pfeiffer, S. Fischer, and W. Effelsberg. Automatic audio content analysis. In *Proceedings of the fourth ACM international conference on Multimedia*, pages 21–30. ACM Press, 1996. ISBN 0-89791-871-1.
- [21] R. Polfreman and C. Spevak. Sound Spotting – a Frame-Based Approach. Technical report, University of Hertfordshire, 2001.
- [22] M. Scarpa, M. Villari, A. Zaia, and A. Puliafito. From client/server to mobile agents: an in-depth analysis of the related performance aspects. In *Seventh International Symposium on Computers and Communications*, pages 768–773. IEEE Press, July 2002.
- [23] Sun Microsystems, Inc. *Java Speech API Programmer's Guide*, October 1998.
- [24] Tommy Thorn. Programming languages for mobile code. *ACM Computing Surveys (CSUR)*, 29(3):213–239, 1997.

- [25] J. E. White. Telescript technology: The foundation for the electronic marketplace. White paper, General Magic, Inc.
- [26] David Wong, Noemi Paciorek, and Dana Moore. Java-based mobile agents. *Communications of the ACM*, 42(3):92–ff., 1999.
- [27] K. Yang, A. Galis, T. Mota, and A. Michalas. Mobile agent security facility for safe configuration of ip networks. In *Second International Workshop on Security of Mobile Multiagent Systems*, pages 72–78, July 2002.

Abbreviations

API	Application Program Interface
JSAPI	Java Speech API
MAGUS	Mobile Agents Using Sound
MASIF	Mobile Agent System Interoperability Facility
RMI	Remote Method Invocation
RPC	Remote Procedure Calls