Crisis Reporting System

Bachelor thesis of: Borislav Bonev

Date: 28.08.2008

University of Rousse

Delft University of Technology

Abstract

Since the terrorist attacks on 9/11 the issue about the collaboration of emergency services has become increasingly important. One of the conclusions is that better communication between the different services is needed. This motivated us to try a different modality for communication. The modality I have chosen is a graphical one, which is very useful when only a limited set of concepts need to be represented. This thesis describes how iconic communication can be applied to the field of Emergency Management.

I have designed and implemented an application, CRS, which is suited for emergency services to communicate with each other or regular citizens to report something that they may thing as an emergency situation, using a map and icons. Our main focus is on the graphical user interface, and the intelligence of the system. The system is designed as a client server application, where the client is focussed on the interface and the server concentrates on the intelligence. I use a Jess knowledge and rule base to provide a consistent world model at all times, while I represent the concepts in XML files. The interface and network is implemented in Java for Android mobile platform.

CRS gives the users the possibility to report about what they observe by placing icons on a dynamic map, grouped as a report. The reports will be send to the server, which fuses the multiple observations and constructs a new world model of it. Besides the world model, the server also sends information about the most likely scenario, and it will suggest icons that are expected in the world model but are not placed yet.

Keywords: icon, communication, map, emergency, crisis, interface, Android, Jess, world model.

Developed by: Borislav Iliev Bonev

Application name: Crisis Reporting System (CRS)

Submit date: 28.08.2008

Receiving University: Delft University of Technology

Project supervisor: Professor Leon Rothkrantz

Sending University: University of Rousse

Erasmus coordinator: Stoyanka Smrikarova

Preface

This thesis is the result of my graduation project at the Man Machine Interaction group of the faculty Electrical Engineering, Mathematics and Computer Science at the Delft University of Technology. This project is done as part of Erasmus program as an exchange student in Delft University of Technology.

Acknowledgements

First of all I would like to thank my supervisor Leon Rothkrantz for inspiring me to do my research. Our sessions always brought new ideas or different approaches. He was also a great help in ordering this report.

Last, but certainly not least, I would like to thank my fellow colleague for listening to my stories, trying to comprehend even the most technical problems I encountered.

Table of Contents

Table of figures

Chapter 1: Introduction

Since the terrorist attacks on September 11th 2001 the issue about the collaboration of emergency services has become increasingly important. One of the conclusions is that better communication between the different services is needed. Another important issue on that day was the total breakdown of the communication infrastructure, immediately after the attack.

Because our MMI department is doing research on, among other topics, multi modal interfaces and AI, it seemed promising to develop an intelligent system with a new modality for communication, using a wireless, ad-hoc network.

Before this project was started there had been some research about chatting with icons, in particular in crisis situations [Tat03]. The basic idea was to create a laguage, which is universal, easy to use and easy to learn. Human communication is based on exchange of ideas or concepts. An idea originating in the mind of the 'sender' is first converted to a string of words. The receiver processes the string of words and tries to understand the underlying ideas of the sender. Conversion, translation and interpretation introduce miscommunication. The challenge is to define a way of communication based on a direct exchange of ideas or concepts. That's why I start with a basic set of concepts, visualized by icons.

In a communication system for emergency services it is important to be able to talk about locations, hence I decided it could be useful to incorporate a dynamic map of the surroundings which can be zoom in and out. To communicate about geometrical positions of objects, I will use local maps of the world provided by Google Maps service. Now observers have to position events and icons on a map, so I have three kinds of communication:

1) Using strings of icons
2) Using strings of commands to locate icons on a global map
3) Locating icons on a local map of the world.

The proposed goal of this thesis is a system that is based on the second kind of communication. It can be used by the emergency services to keep each other up to date about what's going on in a particular area, e.g. a city, by placing icons on the map and sending them to each other. For now I will ignore the need of the wireless capabilities of the system [Kla05], and focus on the interface and intelligence.

1.1. Project Overview

What I would like to achieve is to get a structured World Model from a real life crisis situation. A World Model is composed of objects, characteristic features of the objects, and relations between the objects. Every observer has his own World Model. Police officers, firemen and laymen have different views of the world. An observer will look at the situation that is going on, from this he will form his own ideas of what is happening. When you get just a glimpse of a situation, and recognize certain aspects, the brain will automatically start to make assumptions about what is going on. The brain will construct its own model about the situation, the mental world model. Thoughts like these are based on what he observes, but also on his background knowledge.

I may also assume that observers are positioned differently in time and space. Not all observers will be able to see the same things because they report at another time, or from another place. What I want to do is mould these mental models into a computer system. To do this I need to make the mental model more concrete, so that it can be stored in a structured way. Next I want to

fuse the different reports into one shared World Model, see Figure 1.
The agent in the field observes what is going on in the Real World and forms his own Mental World Model of the situation. He then wants to report his thoughts with the report tool of the system. The tool will only be able to handle structured information; concepts represented by icons. Thus the reporter has to concretise his ideas in icons, that he can then place on the map and create a report of what he sees. Information for every icon is send to the server. The local model is updated and propositions and advices are sent back to the client. The agent will see these suggestions, forcing him to observe the situation again, to see if he missed anything.

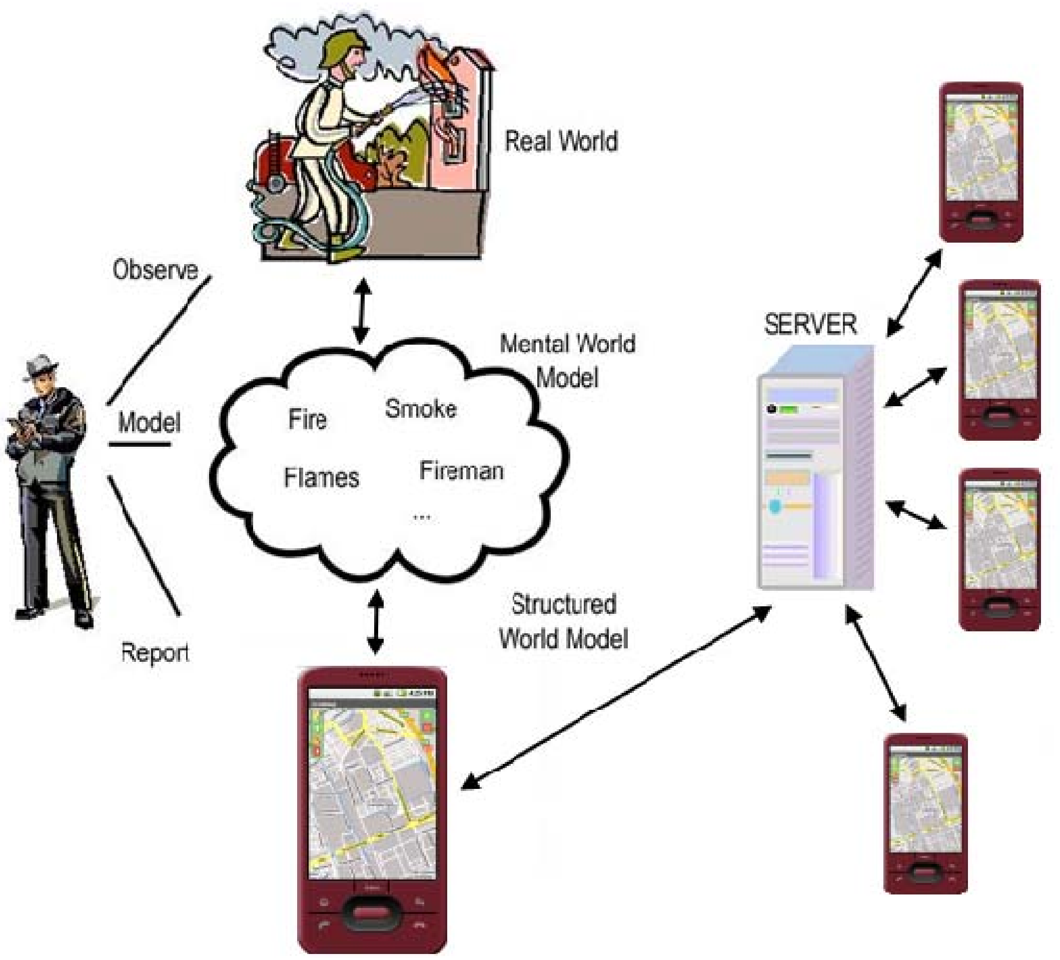


Figure 1 Project Overview

For an intelligent communication system like this I will have to look into several aspects:

- First of all I have to define a World Model, different sets of icons, corresponding to different crisis situations, and a priori information about characteristics and relations between the objects. What icons will I need? How are these icons related to each other, and what are the specific characteristics of each?
- Then there needs to be the interface of the report tool. The interface should provide a clear structure for the communication. What kind of information should be reported to the system, and what kind of information should be distributed back to the users? How will the information be represented to the user?
- A next aspect is the intelligence of the system, in particular in the fusion of the different reports. How does the system handle double or missing information? How should it deal with contradicting information? How does it keep its world model unambiguous and up to date? How does the system handle time and dynamic events?

- Another issue is the security of the system. Since it will be based on wireless communication, how can I prevent outsiders to intercept information? Should all information be send to all the users, and if not, how do I define different roles of users? How can I prevent the server from going down? And if it does go down, how can I prevent losing the information?

1.1.1. Define a World Model

To define a World Model I will need information about what concepts play a part in crisis situations and how the concepts are related to each other. It is important that the icons will be expressive enough to represent the needed information, and that the vocabulary of the system is extendible and editable. One of the first problems I encountered during the project is that the emergency services are very conservative with providing information. The information I have used is therefore for a big part originating from emergency scenarios as described in news articles. Although I tried to define the concepts that can occur in a crisis situation to our best knowledge, the reality of what the emergency services would actually use as concepts may be somewhat different. The information about icons should therefore be stored as flexible as possible, to be able to cope with different, more or other icons, or different scenarios.

1.1.2. Interface

As said in the introduction I will be communicating by the means of placing icons on a map. Since I are developing it for crisis situations I want the clients to run on handheld computers. This means the application has to fit a certain dimension, making it rather compact. It’s important for the interface that icons can be easily selected and placed on the map. The icons should therefore be grouped in a logical way, so one can find the needed icon quickly. It is also important that the icons can be placed with some accuracy, which can be obtained by developing zoom functions for the map. A last issue is that the interface should be able to deal with the flexible information about the icons. There is little use in making the information flexible, but the interface not.

1.1.3. Intelligence

I want a system that is intelligent, in the sense that it can deal with icons that are received multiple times, cope with icons that might be missing and icons that might be placed wrongly. If icons are received multiple times, the system should be able to handle this on its own, without human interference. In the other cases however it is very hard and dangerous to let the system delete, modify or add icons to its world model without human approval. Therefore the system should be able to ask for feedback to the users, if it detects that errors might have occurred. Again these errors are based on our own beliefs about crisis situations, and might be different from reality. Taking this in mind, I want the Artificial Intelligence of the system to be dynamic and easily adjustable. Another issue about the intelligence of the system is its ability to cope with time. Specific events may happen in a strict order.

1.1.4. Security

Since the application is going to run on handheld computers, there will be wireless communication. Wireless communication can quite easily be intercepted, allowing people to ‘listen’ to what is happening, and maybe worse, to actively send (wrong) information to the

system, making the system worthless. This can be solved by encrypting the information.
Another security issue is the failure of the server. I don't want a system with only a central server, because it could lose all information when it goes down. Therefore I want to have all the information distributed amongst the clients, simulating a central server or having redundant server/s.
It might also be necessary for the different users to receive different kind of information. Maybe ambulance personnel is not interesting in certain events that are important for the police or the fire department. This could be solved by giving the different users different roles, which determines what information will be send to them.

1.2. Problem Description

In the ideal case I would have a system that has its information safely distributed among different clients and servers. The system would have a clear interface that is easy enough to not make mistakes, but complex enough to handle difficult and unexpected situations. The system would at all times have a correct and up to date world model, that automatically adds missing information, alters wrong information and deletes excessive information. Furthermore I want a system that can easily be extended and altered, both on its vocabulary and its intelligence.

Taking in account that not everything can be handled in a single thesis work, the problem I are trying to solve in this particular thesis work is focussed on the interface and intelligence and secure communication, and is defined as follows:

> Design and implement multimodal system for mobile devices, capable of reporting crisis and emergency situations via wireless/mobile high speed network and using a map of the surroundings, which is expressive enough to handle complex and unexpected situations, yet intuitive enough to use without making (a lot of) errors. The system should be language independent, allowing the user to add icons, text and different drawings to the report. The system should be intelligent enough to assemble and maintain a correct and up to date world model. It should detect possible errors in the form of missing, double and wrongly placed icons. Furthermore the system should be dynamic in the sense that new concepts and rules can easily be added. It also should provide data availability against power down (in client and server side), data reliability and data security.

In chapter 3.1 I will elaborate on this problem description, splitting it up in workable components, which the final result will be tested against.

Chapter 2: Related Work

In this chapter I will describe some related work that was studied before starting the design of the system. I will first discuss some background information about icons, then I will look at an emergency system that has been designed by the Dutch government, and finally I will take a look at a system that uses icons to communicate.

2.1. About Icons

Because I will be using icons to communicate I will first present some information about icons. This section will tell something about the history of icons, followed by some information about the modern use of icons. See [Bea94], [Cha02], [Dor94], [Jon96], [Mea91], [Mea94], [NRC03], [Ric94], [Shn98].

2.1.1. The History of Icons

Icons are graphical symbols representing a concept or thing in reality. The term icon has been adapted from the Russian word ikon, which is a religious painting or statue. Icons have been around for a very long time, as early as the middle ages complex iconic systems have been used, for example to denote systems of astrological signs. It may even be argued that the ancient Egyptians were using icons as a language. They may not have called them icons, but they did communicate using graphics.

In the 1930s Otto Neurath developed Isotype, a system for communication which uses stylised graphics within a two-dimensional syntax. Neuraths work ranges from a very specific example of how a complex idea can be conveyed graphically, to a proposal for an international set of iconic images.

In the 1950s, Charles Bliss developed a set of atomic icons that represent basic objects in the world, and their features. These can be combined to form complex icons that map on to the set of words found in natural languages. Figure 2 shows how I can construct a symbol for telephone using: mouth-ear-language-electricity-telephone

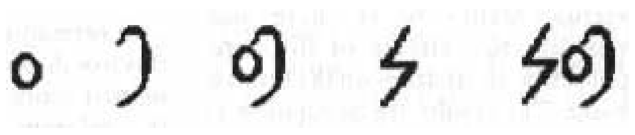


Figure 2 construction of the Bliss symbol for telephone

The work of Bliss has some resemblance with the work of linguist Anna Wierzbicka, who claims to be able to describe any concept with using only 61 different words. The combination of these atomic words lead to a new concept, just as the atomic pictures of Bliss lead to a new concept. Although Wierzbicka does not use icons, the possibility of mapping her atomic words to atomic icons seems interesting.

The iconic languages were not all as successful as their developers might have hoped for, but they do show that there are distinct advantages in a communication based on graphical icons.

Our ability to learn or to recall the meaning of a sign seems to be greatly enhanced to the point where I may not need to be told what the sign represents or to explicitly learn its meaning. There might also be some advantage in the efficiency of using icons over natural language in the sense that difficult concepts might be represented by only a small number of icons, as opposed to many more words. Furthermore our ability to recognize icons does not depend on the natural languages I know, suggesting that iconic systems may be a way to overcome linguistic differences. Note that this does not imply that icons are also culturally independent. [Colin Beardon]

2.1.2. Modern use of icons

Within the computing context the word icon is used to denote a small graphical representation of a program, resource, state, option or window. As such, icons form an important part of the Graphical User Interface (GUI).

An ideal icon language wouldn't need any explanation, the intuition of the user, based upon his life experience, should be enough to immediately understand it. Of course, this is not a very realistic goal. Just as any language, icon language is something that does need some training. Most people already have some training in recognizing icons however, because icons can be found anywhere. In many public places they are used extensively, for example to indicate the toilets, or to show where the emergence exits are. A lot of icons are used in traffic signs, they point out if you are allowed to overtake other cars, if a road is one way only, or is a dead end.

The challenge in designing icons is that they should be as easy as possible to learn, as easy as possible to remember, and as easy as possible to recognize. Therefore icons should be designed with the following criteria in mind:

- Graphically clear
- Semantically unambiguous
- Cultural independent
- Simple.

To make them graphically clear is straightforward, make the icons so that they resemble a concept in the real world, and keep them simple. Too much detail cannot be shown clearly in 32x32 pixels. Semantically unambiguous means that the icon only represents one concept, and that concept is only represented by that particular icon. Don't make two icons for the same concept, don't make one icon for 2 concepts. Cultural independence is harder to achieve, try to make the icons independent of any cultural background information. An easy example of doing this in a wrong way is using the road sign in Figure 3 in Great Britain, where this sign would mean that it is forbidden to be overtaken by other cars.

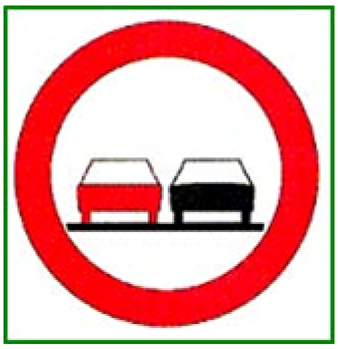


Figure 3 Cultural dependent icon

If icons are for some reason not clear to the user, they can be explained by written or spoken text. This has, however, a very big drawback. One of the most important advantages of icon use will be neglected in this way; icons can be used to communicate with people that don't speak each others natural language. If the explanation of an icon is given in a natural language, somebody who doesn't speak that language isn't able to use it.

A better way, perhaps, to explain icons that are not very clear is to animate them. Especially on computers this is very convenient to do and some icons are animated already. In most operating systems you see an animation when you copy files from one map to another, for example. Animated icons should only be used if normal 'static' icons will not suffice. In addition, most animation should only be used for the explanation of the icon; otherwise, you will become very distracted by all the moving images on your screen.

There are three styles of icons that are commonly used, see Figure 4:

1) Silhouette style; this one is very straight forward and clear, the drawback is that it is somewhat limited in the range of things it can represent.
2) Three-quarter top view; this style is very informative, but it requires some visual understanding.
3) Realistic style; this one is easy to recognize, but it is not very generalizing.

Although the use of these different styles makes it possible to select the best one for each icon, it is not recommended to use a mix of different styles, as it can be confusing.

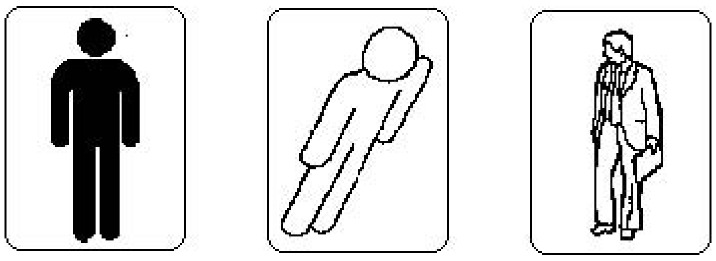


Figure 4 Three styles of icons

In systems that have a lot of different functions, it is not easy to design an icon for each function. To improve the recognition of the different icons, they may be divided in subsets. For example one could use a single group icon for representing surfaces, and have as subset different icons for representing circular and rectangular surfaces. The division of icons in subsets can also improve the overview and layout of certain applications.

The icons I am going to use in my system are all designed in a silhouette style. Since I use colours the icons may seem to fit in a realistic style as well. Because in a crisis situation I want as much information as possible, an icon on the map is often not descriptive enough. To add extra information I have tried a visual approach at first. I have tried 4 ways to visualize combinations of icons.

Transparent icons

By making the icons background transparent I can just place them on top of each other. For some icons this works, but a lot of icons cannot be recognized anymore, especially when more than 2 icons are used in the combination. For example a fire in a building is shown as follows in Figure 5. This is not a good visualisation, although there are only two icons used it becomes unclear what's shown in the first icon.

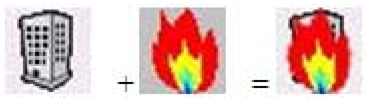


Figure 5 Transparent icons

Alternating icons

In this version the combinations will be visualized by alternating the icon every time unit. For example a building is shown for one second, then it gets alternated by one second of flames. This type of visualization is already a big improvement on the transparency, but still has some drawbacks. If there are a lot of combined icons on the map, the whole map is blinking, which is very distracting.

Another drawback I found after implementing and testing it is that some combinations are shown very poorly. If I would select a car, a crash, and another car, it would just show a car for two time units and a crash for one time unit. This is not nice because it looks like there is just one car in this case. Of course there can be worked around this problem, there could be certain rules that would not allow two identical icons in one combination. The example combination could then be rephrased like 2, car, crash.

If a very big combination is made, another drawback can be found. It takes the user several seconds to see the meaning of the icons because they are not shown simultaneously. They would have to wait for the whole message to come by, before they understand the meaning. I would rather have an instant overview with a single look at the map.

The final drawback about this implementation is that the application becomes pretty slow when a lot of combinations are made. The map keeps getting repainted every time unit, and this is of course really computer time intensive. Because the application is designed to eventually run on a handheld, this might prove to be a problem. It may be clear from the above that this is not the type of visualisation I am looking for.

Stacking icons

In this version the icons simply get stacked on top of each other. The advantage is that this is a very easy solution, because each icon will simply be placed shifted some pixels up and right of the original icon. Also the icons used in the combination can still be recognized and they are shown simultaneously. The only disadvantage is that the stacks take up more space than the other versions. This can be worked around by setting a maximum size of a combination. The stacking version is shown in Figure .

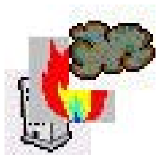


Figure 6 Stacking icons

Grouping icons

In this version the icons are group in events. So on top of the group is the icon that represents the type of event/scenario. When reporter selects the icon the group is expanded and all icons are visible. The advantage of this approach is that the map is not overwhelmed with icons when the map is zoomed in or out. Reporter easily can find desired location and select the event that is interested of. The main disadvantage of the approach is that the related icons are hidden until the user selects them. Below is shown the icon grouping.

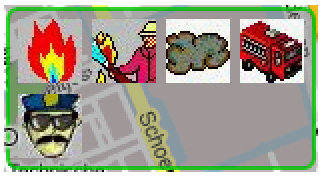


Figure 7 Grouping icons

A reason to abandon the idea of visual representation of additional information is that certain types of additional information are very hard to represent with an icon. If I want to say something about the status of a policeman, I use terms as busy, waiting, wounded. It is much easier to give this information in a natural language. That's why I implemented a way to add and view information by clicking on the icon and adjust some of its attributes.

2.2. C2000

In the Netherlands a new digital radio network for the communication of emergency services is being developed [C2000]. It is called C2000, and its goal is to maximally facilitate the communication between the fire-brigades, ambulance services, police-brigades and military police. The mobile communication between these emergency services should be supported and improved. The system should guarantee fast and secure communication, make communication between different emergency services possible, and help improve the safety of the emergency personnel.

The need for a reliable communication system for these services is high. Not only for the day to day activities, but also in case the different services need to cooperate with each other. The emergency services themselves are closely involved in the development if C2000. In 1996 the first steps were undertaken to develop it, and the system is (was) supposed to be in full operation at the end of 2003. In Figure 8 the design of the C2000 network is shown.

The numbered components are explained below:

1) Direct Mode Operation. DMO makes it possible for car phones and walkie-talkies to communicate with each other directly, without making use of the network.

2) Air Interface. Communication of a mobile station takes place using electromagnetic waves, with a transmitter mast, or with another mobile station via the Air Interface.
3) Inter System Interface. Multiple TETRA networks can be linked using the Inter System Interface. This is important for international communication.
4) Direct link with the central emergency room.
5) Gateways. The gateways make it possible to link the system to other external networks, such as the public telephone network, or the Nationale Noodnet (national emergency net).
6) Peripheral Equipment Interface. The PEI supports communication between laptops and mobile stations, such as car phones.

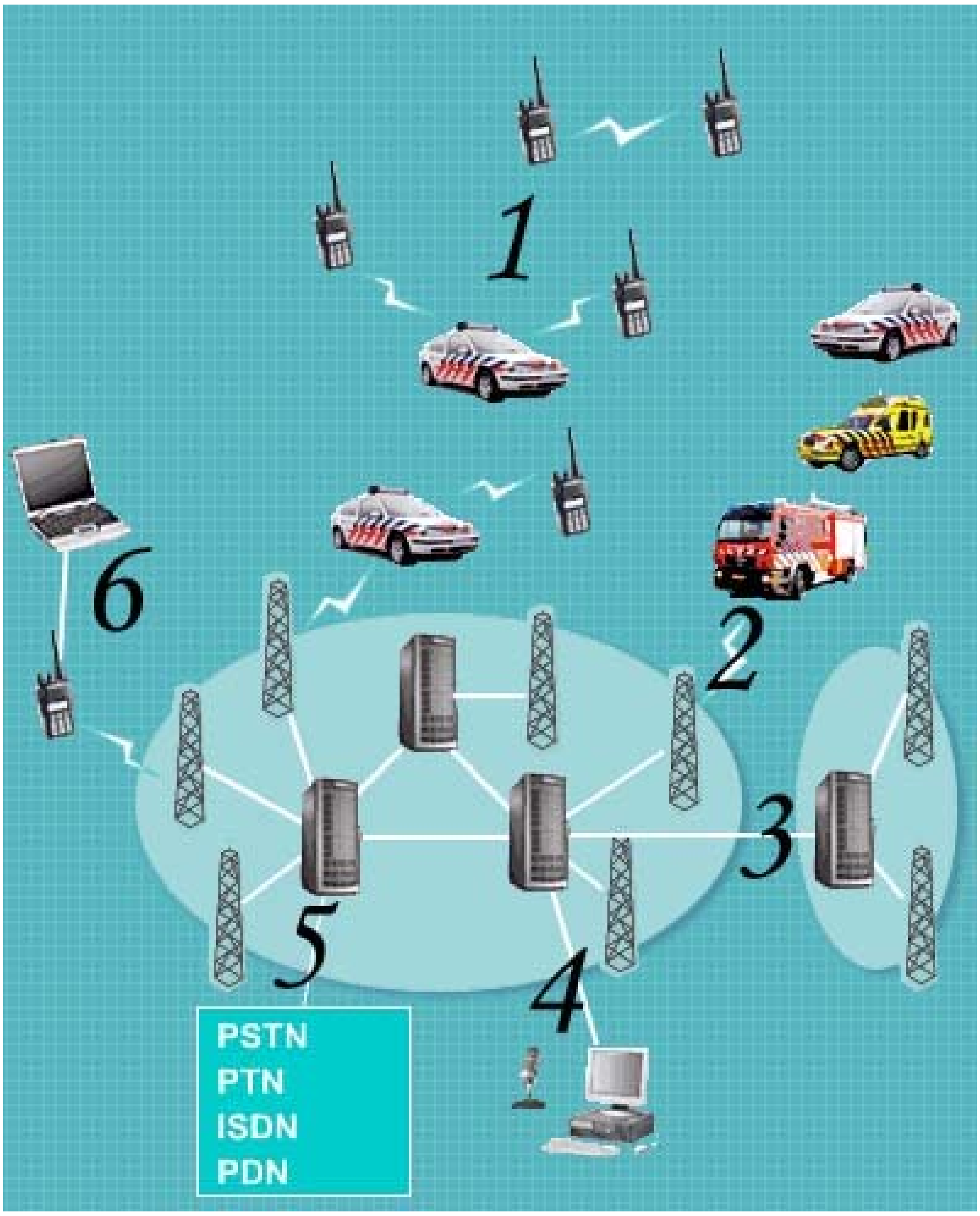


Figure 8 Overview of C2000

2.2.1. Advantages

C2000 makes the communication among emergency services fast, simple and reliable. This one national system will replace almost 100 local systems that are currently used by the different services. The digital network has big advantages over the old analogue systems:

- C2000 is suitable for multidisciplinary communication, whereas this was impossible with the old systems.
- C2000 is designed in a way that is easy to secure, making it virtually impossible to eavesdrop on it.
- C2000 has a national coverage, whereas the old systems only have regional coverage.
- C2000 has a much better sound quality for speech.
- C2000 is very suitable for data communication.
- All car phones and walkie-talkies are provided with emergency buttons.
- C2000 supports communication with foreign co-workers, improving provided services near the borders.

2.2.1.1. A joint radio network with national coverage

C2000 is a big improvement on the current situation. At this time all regional organisations of the four emergency services have their own networks in use, adding up to almost 100 networks, spread out all over the Netherlands. This makes communication among them very difficult. However, in the case of a big calamity, like the disaster with the fireworks deposit in Enschede, good communication can save lives. On top of this, the old networks are all analogue and outdated. Because C2000 is one national network, the communication will be fast, simple and reliable. It is believed that in the long term a lot of money can be saved on the acquisition of hard and software because it will be standardized. Money will also be saved by having a joint education of the personnel. Furthermore the network is believed to provide an excellent basis for future technologies, because of its state of the art technology.

2.2.1.2. A high level of security

All car phones and walkie-talkies get provided with an emergency button. When this button is pressed a connection is made directly with the central emergency room. The operator can then automatically listen along with what's going on. C2000 has an excellent quality of speech. The system is designed in a way that it can not fail because of excessive use, what happens with 'normal' communication such as the telephone when a lot of people use the network at the same time. Think about New Years Eve, when everybody is trying to call their families at the same time, the system then gets overloaded and fails to do its job. Furthermore the system is secured against eavesdroppers. While people with special scanners can now freely hear everything that's being said over the analogue systems, this can't happen in the new system because every conversation will be secured automatically.

2.2.1.3. An open European standard

C2000 is based on the European TETRA standard. TETRA stands for TErrestrial Trunked RAdio. Just like GSM is the standard for mobile telephony, TETRA will be the standard for emergency services. The standard is designed with the cooperation of the industries, ensuring that a customer is not dependant on only one supplier, but can purchase TETRA equipment at several suppliers. The TETRA standard is based on the latest digital technologies and is continued to be improved. This will ensure the systems based on the standard will be fit for the future. Most countries will be using, besides the TETRA standard, the same frequency. This will allow for international cooperation.

The infrastructure for the C2000 system has been delivered in July 2004 and at this moment some regions are working with it. The system is expected to be up and running in the whole of the Netherlands at the end of 2004.

2.3. Iconic Communication

A closely related project involving icons and emergency situations was done by Iulia Tatomir [Tat03]. The goal of this project was to create an application that allows its users to communicate with each other using an international 'language', icons.

First the graphical user interface was developed, which is designed to be as easy as possible to use. Because its is designed to be a simulation for a PDA application, big constraints on the applications dimension were imposed. The interface is shown in Figure 8

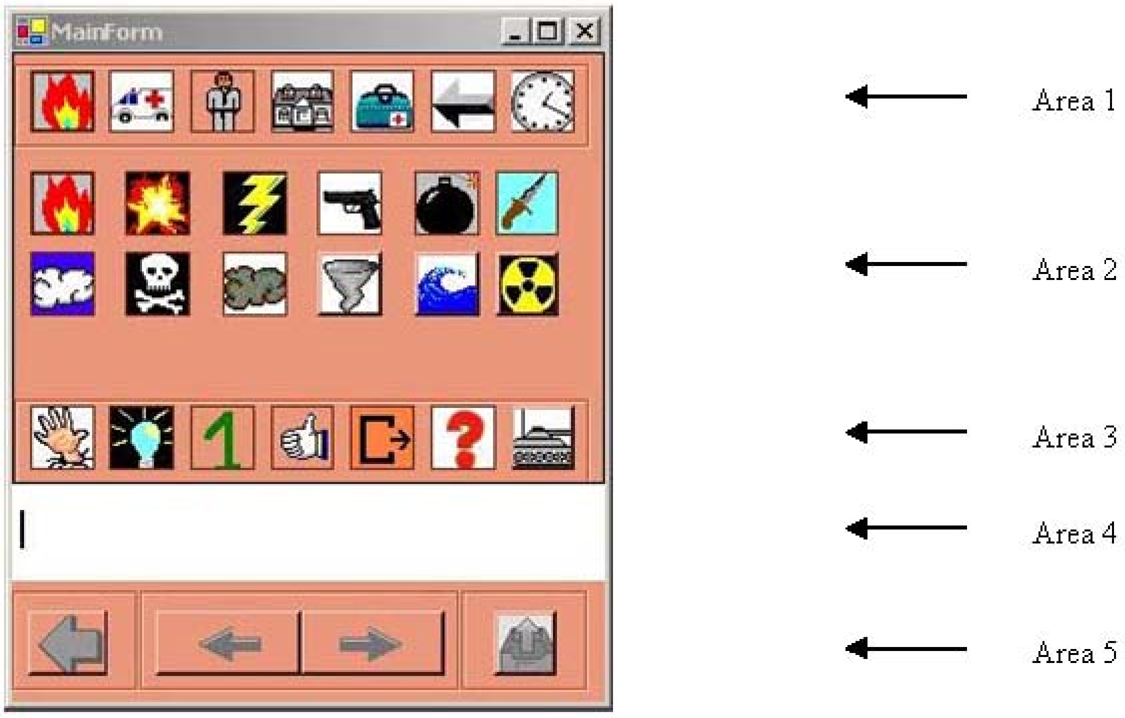


Figure 9 Interface for iconic communication

The application is divided in 5 areas

Area 1: This is where the main categories are represented by their defined icon. The categories are Crisis, Cars, People, House, First Aid, Directions and Time.

Area 2: This is where the icons of the chosen category are displayed. The first icon in this area is always the icon that represents the index icon for the category. When an icon in this area is being clicked on, it will appear at the cursor position in Area 4.

Area 3: This area is an extension of Area 1. The categories in this area are Human Actions, Information, Numbers, Yes/No, Special Signs, Intonation, Military.

Area 4: This is the area where the selected icons are placed to form the sentence that the user wants to send. There is room for 7 icons, which should be long enough for a sentence in icons.

Area 5: This last area is to edit the sentence. The middle left and right arrows are for navigating through the sentence, it will move the cursor through the sentence. The big left arrow is used for deleting an icon. The delete works as the backspace button on a keyboard, so it deletes the icon before the cursor position. The last button is used for sending the sentence.

The main part of the research in this project was done on defining the grammar of the icon sentence. Not all combinations of icons form correct sentences.

The grammar that was developed is a context free grammar, based on Chomsky's hierarchy. A grammar is basically a set of rules that defines how grammatically correct sentences can be formed. A grammar is formally defined as a quadruple G = {N, T, S, P} with:
N – a finite set of non terminal symbols
T – a finite set of terminal symbols
S – a special goal or start or distinguished symbol
P – a finite set of production rules

The union of the sets N and T form the vocabulary of the grammar and should not intersect.
The final grammar used in the application is defined as follows:

```
A = number B | O
S = negation A | number B | adjective C | noun R | verb
E | adverb H
O = verb E | P
P = verb | B
B = adjective C | C
C = noun R | noun
R = sign | D
D = negation F | F
F = verb | G
G = verb E | E
E = adverb | I
I = adverb H | H
H = number J | J
J = adjective K | K
K = noun L | noun
L = adverb
```

Figure 10 Grammar of the Iconic Communication system

When an icon is selected, the sentence gets parsed and the system decides which icons can follow. This is extended by the interface, by only making the icons that fit correctly in the sentence selectable.

The similarities of the work by Tatomir and this project are of course the use of icons and the context of a crisis environment. A lot of icons in her work will thus be seen again in this project. The categories of icons are different however, because on a map only nouns can be placed. There is no use for verbs, numbers, adjectives, and so on.
While the idea of defining full sentences with icons seems promising it is not very applicable in combination with a map. The only sentences I will need are of the form:

There is a <noun: icon> at position <number: x, number:y>

2.4. Icon based System for Managing Emergencies (ISME)

This project is most close to my own. It's developed by Paul Schooneman[Psc0] a few years ago. It was designed to suit for emergency services to communicate with each other, using a map and icons. My project is based on it with the main purpose to improve the graphical interface, communications, security and intelligence.

2.4.1. Overview

ISME is a prototype application which is designed to suit for emergency services to communicate with each other, using a map and icons, with main focus on graphical user interface and intelligence of the system. On Figure 11 is shown then graphical interface of ISME report tool which is developed in Java and Abstract Windowing Toolkit (AWT) API. The application is developed to fit on a handheld computer, and therefore has a resolution of 640x480 pixels. Most of this space is filled up with a map. Most of the space is reserved for the map, because the map should give a direct overview of what is going on.

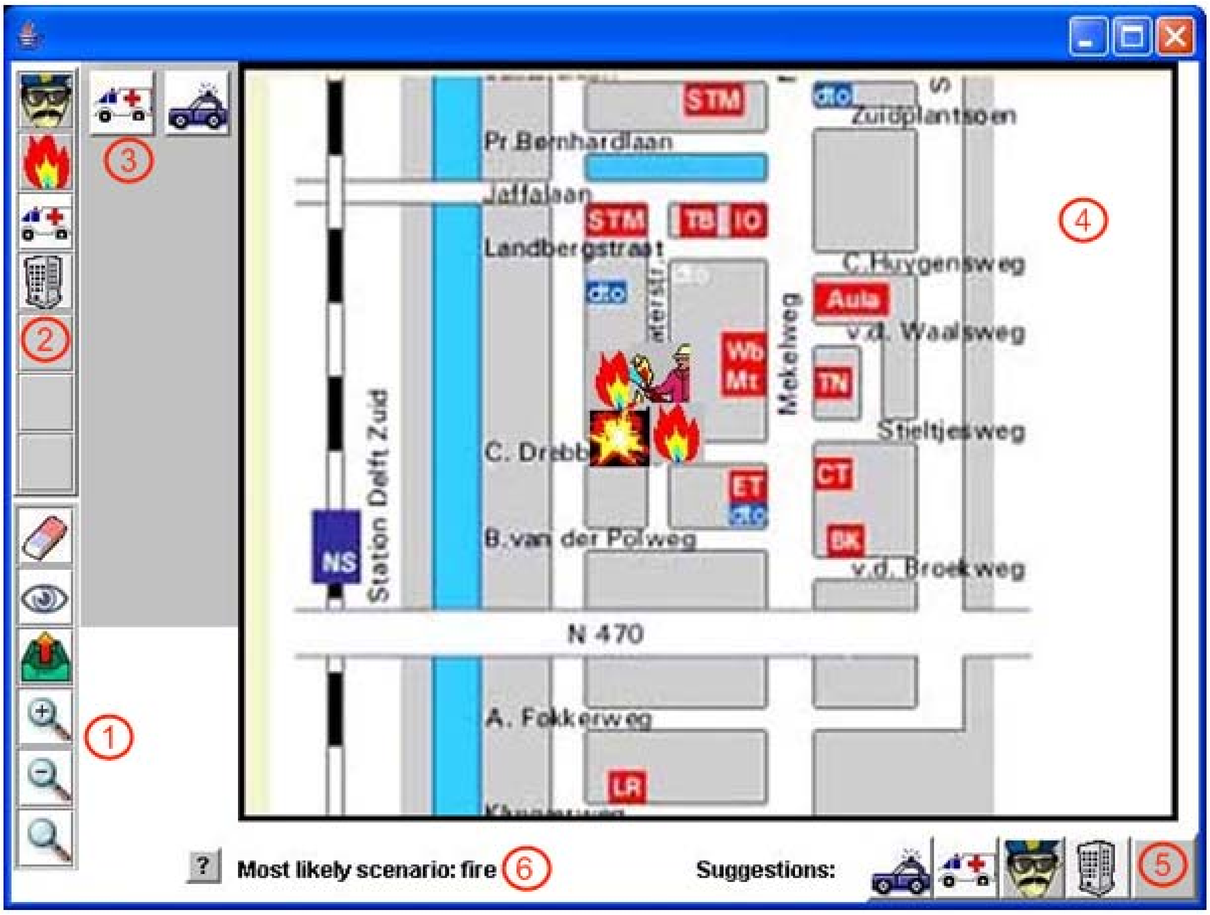


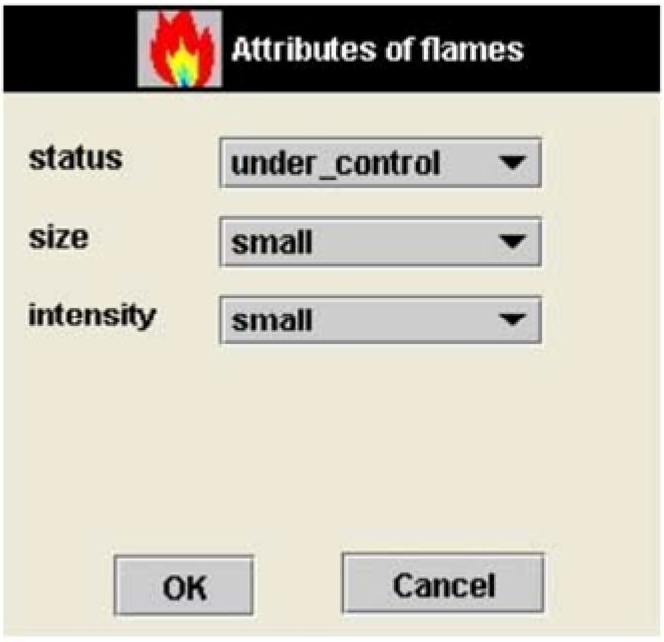
Figure 11 Icon based System for Managing Emergencies (ISME)

On the main window there are six distinguishing areas:

1) Control buttons. There are located the main control elements: a delete tool, an inspect option, the ability to send your information to the server, and some zooming possibilities.

The inspection tool is when user clicks on an icon from the map to see the additional attributes of that icon, see Figure 12.

2) Icon categories. There are seven boxes for different categories. Although there are only 4 categories in the figure, there is room for 3 extra categories. Categories and icons are described in a XML file then easily can be added more categories and icons in each of them.
3) In the grey area right of the categories are the icons that are current selected this category. This area will open up when a category is selected. There is room for 14 icons per category, so effectively 98 different icons can be used in the application. When an icon is selected to be placed a red border appears around the icon.
4) Main part of the screen is reserved for the static map. On the map are displayed all icons placed by users. The map can be zoomed moved but is limited to size, because in this application is used a static image as a map which is one of its main drawbacks.
5) In this area are show the icon propositions that server suggest as missing in users reports. There is space for maximum of 5 icons.
6) It's shown the current scenario proposed by the server based on reports of the users.


Attributes of flames

status: under_control

size: small

intensity: small

Figure 12 Attributes window.

2.4.2. Disadvantages

There system is well designed but there are some lapses:

- One of the main disadvantage of this implementation is using of static image as a map which is send by the server to the user when the last connects to it. This limits the flexibility of the application. Changing the map is not easy and if the map covers a vast area it can be difficult to transmit and load on a handheld device. It's difficult to navigates and understand what is going on if the events are to the edge of the map and some of the icons are on a different image (map). It's impossible for the application intelligence module to work with relative to image coordinates and calculate distances between users and events. A solution for this is using a dynamic worldwide map and Global Positioning System (GPS) to determine users and events positions and distances between each other. Such options offer the Google Maps service which provides a flexible map and access to global map coordinates.

- Other disadvantage is that icons that have to be places on the map are send by server as a Java objects serialized through socket connection stream along with the map. This generates enormous traffic. A solution to this problem is to keep all possible icons in the application installation package and during communication to send only icons name and needed attributes. There is one lapse in this approach and this is application extension in terms of adding new icons, is possible only via application updates.
- At last the application does not support adding a text or drawings by user to support his report.

In my project I tried to remove this disadvantages as can see in Chapter 4:Global Design.

In iconic application one of the preferred requirements about icons is they to have a similar or close design. Finding icons suitable for this application is difficult and creating this large number of icons is a tedious work. That’s why I’ll use the most icons from this application for the needs of my project.

Chapter 3: Tools

In this chapter I will describe the tools and technologies used in this project. I will first explain the mobile working platform, which will host the client application module, after that I will explain what is XML, and I give some information about the language used to implement the intelligence.

3.1. Android platform and Android emulator

I'm going to make a demonstration for a real system, which, except for the server, has to run in the field during a crisis situation. This means that the type of hardware I need should be usable in such situations. It is impossible to let the eventual system run on laptop computers, let alone normal PC's. The system has to run on handheld computers in mine case a prototype mobile device, and thus the demonstrator has to take into account the constraints which come with that.
In my project I will use Android mobile platform. This platform is still in development.
Android is a software stack for mobile devices that includes an operating system, middleware and key applications. The beta version of the Android SDK provides the tools and APIs necessary to develop applications on the Android platform using the Java programming language. The main features of the platform are:

- Application framework enabling reuse and replacement of components
- Dalvik virtual machine optimized for mobile devices
- Integrated browser based on the open source WebKit engine
- Optimized graphics powered by a custom 2D graphics library, 3D graphics based on the OpenGL ES 1.0 specification (hardware acceleration optional)
- SQLite for structured data storage
- Media support for common audio, video, and still image formats (MPEG4, H.264, MP3, AAC, AMR, JPG, PNG, GIF)
- GSM Telephony (hardware dependent)
- Bluetooth, EDGE, 3G, and WiFi (hardware dependent)
- Camera, GPS, compass, and accelerometer (hardware dependent)
- Rich development environment including a device emulator, tools for debugging, memory and performance profiling, and a plugin for the Eclipse IDE

Because the platform is in development process, there is no actual devices that runs on that operating system. For the purpose of the project for development I will use an emulator.

The Android emulator (Figure 13) is a QEMU-based application that provides a virtual ARM mobile device on which you can run Android applications. It provides a full Android system stack, down to the kernel level, and includes a set of preinstalled applications. It provides a skinnable mobile device UI, customizable key mappings, and a variety of commands and options for controlling the behaviours of the emulated environment.

The Android system image distributed in the SDK contains ARM machine code for the Android Linux kernel, the native libraries, the Dalvik VM, and the various Android package files (such as for the Android framework and preinstalled applications). The emulator's QEMU layers provide dynamic binary translation of the ARM machine code to the OS and processor architecture of the development machine.

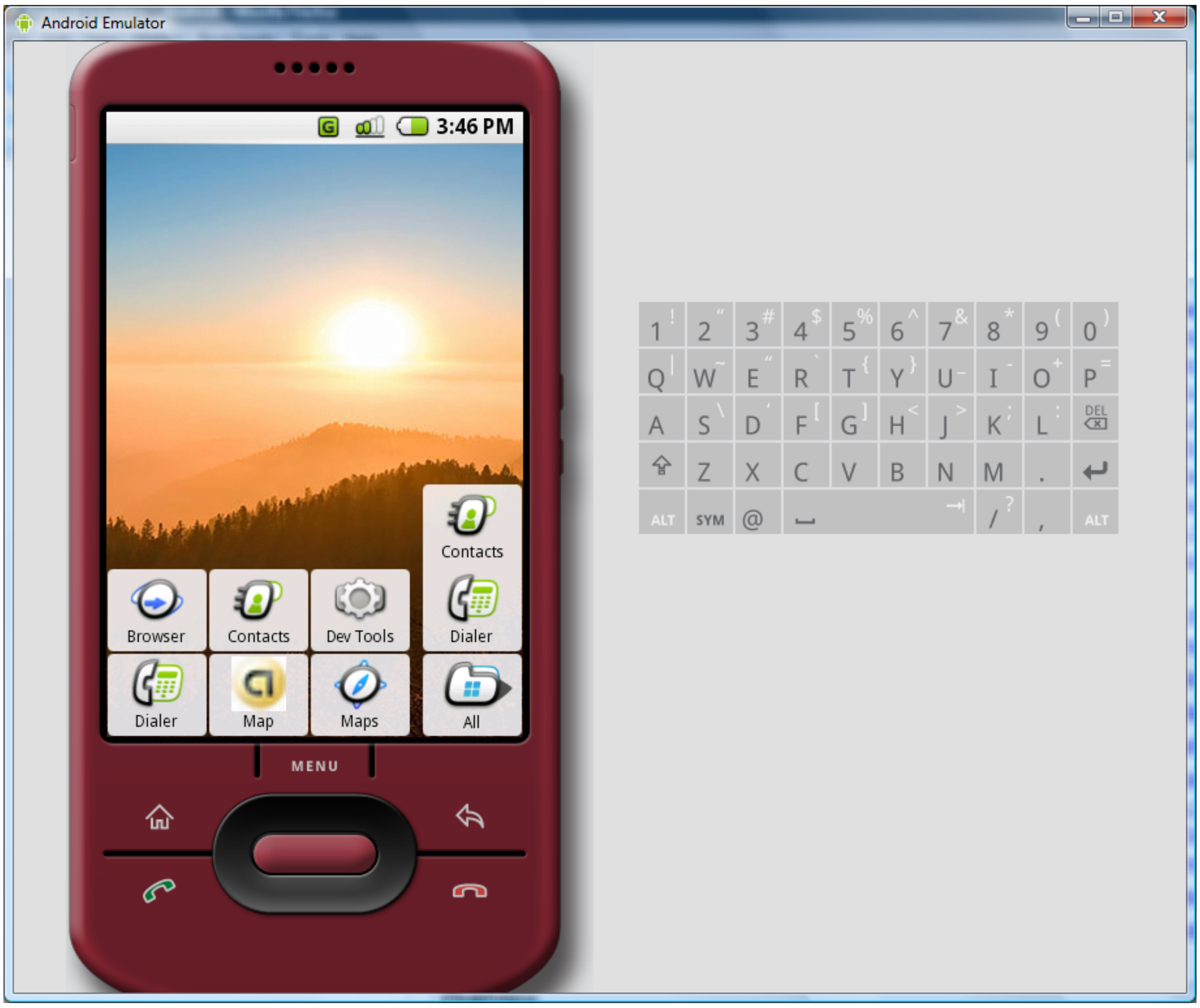


Figure 13 Android emulator

Adding custom capabilities to the underlying QEMU services, the Android emulator supports many hardware features likely to be found on mobile devices, including:

- An ARMv5 CPU and the corresponding memory-management unit (MMU)
- A 16-bit LCD display
- One or more keyboards (a Qwerty-based keyboard and associated Dpad/Phone buttons)
- A sound chip with output and input capabilities
- Flash memory partitions (emulated through disk image files on the development machine)
- A GSM modem, including a simulated SIM Card

3.2. Java, Android, Jess and XML

Following from the requirements and constraints I have made a choice on which programming language(s) to use.

The main programming language I will use is Java. There are some advantages of Java that made me decide to use it. The first, and most important, is that Java is platform independent, meaning that it should have little problems running on a handheld, or the server. The second important reason to use Java is that there is a lot of knowledge available about how to handle problems that may arise. A lot of people are familiar with the programming language. Many help forums are available as well.

The device platform that will use is Android. Advantage of using Android platform is that the developed applications have a direct support and access to Google Maps service and also to the Global Positioning System (GPS). Expected devices are powerful enough to process complex applications with map support as mine own. The main drawback is that first devices that natively support this platform are expected at the end of year 2008 and my application can be tested only on a PC or laptop.

The storage of the world model of the server and the intelligence of the server are both combined in one component, Jess [JESS]. Jess stands for Java Expert System Shell. In short Jess is an expert system that works with facts, and rules that are automatically triggered when the conditions are met. Jess is actually the Java version of CLIPS [CLIPS], with some added functionality to cooperate better with other Java classes. More about Jess and how it works is explained in section 3.4. The fact that Jess is written in Java made the choice of using it easy. It should give little problem to embed the Jess component in the rest of the application, written in Java.

The last part of the system is the storage of icons and the rules that apply to them. I have chosen XML files for this part. They are very light weight, and easy to edit. Yet they have constraints about how information is stored. The form of the XML files can be defined in special Data Type Declaration files, as will be explained in section 3.3. Another advantage is that XML files will be relatively easy to read by Java.

3.3. XML Files

XML stands for eXtensible Markup Language and is a language to define structured information. It is very lightweight, as opposed to a traditional database, and can be edited relatively easy. All XML files should be well-formed, meaning that they should obey to certain grammar rules of XML. This will lower the likelihood of making errors while editing or creating the file. Besides this protection, the structure of the document can be further constrained to be valid, meaning the file should obey a predefined structure. These structures can be defined in Document Type Declaration file. This file exactly defines what structures are allowed.

In DTD files is defined exactly what structures in the XML file are allowed. XML files use, just as HTML, tags that separate the structure from the data. A tag is an indication of what information will follow, at the end of the information will be a closing tag. Tags and information can be nested. An example to clear this up:

```
<book>
        <title>Artificial Intelligence: a modern
approach</title>
        <author>Russell</author>
        <author>Norvig</author>
```

Figure 14 Example XML fragment

Everything from the beginning to the end of a tag pair, is called an element. In the example in Figure 14 we have the following elements: book, title, author.

Elements can thus be nested. Note that elements may contain multiple of the same sub elements, while a book has only one title, it may have more than one author. This structure can exactly be defined in DTD files. The DTD file that defines the structure of the example is shown below:

```
<!ELEMENT title (#PCDATA)>
<!ELEMENT book (title, author*)>
<!ELEMENT author (#PCDATA)>
```

Figure 15 Example DTD

Every element has to be specified in the DTD, and it defines how each element is build up. From Figure 15 we can see that every book has one title, and zero or more authors. The title and the author elements are both specified to contain #PCDATA, this is Parsed Character Data, which means it can contain an arbitrary string of characters.

3.4. Jess

Jess is the expert shell in which I will program my expert system to add the systems intelligent behavior. First I will provide some information about how Jess works and in the next chapter I will discuss the design of my expert system.

3.4.1. About Jess

Jess is a rule-based expert system shell made in Java. This means that Jess's purpose is to continuously apply a set of if-then statements, the rules, to a set of data, the knowledge base. The user can define his own rules to make his particular expert system. Jess rules are of the form:

```
A
B
→
C
```

Figure 16 Structure of Jess rules

Which means that when A and B are statements that are both true, C will be made true as well. An example of a rule and its explanation:

```
(defrule library-rule
        (book (name ?X) (status late) (borrower ?Y))
        (borrower (name ?Y) (address ?Z))
        =>
        (send-late-notice ?X ?Y ?Z))
```

Translation:
Library rule:
If
A late book exists, with name X, borrowed by someone named Y
And
The address of borrower Y is known to be Z
Then
Send a late notice to Y at Z about book X

Figure 17 Example Jess rule

The book and borrower information would be found in the knowledge base. The knowledge base is a collection of facts about the world. In the knowledge base we will be using, the placed events and icons will be a big part of the knowledge base, and they will be the facts that are added and removed. Together the placed events and icons will form the world model. The attributes, or slots, that the facts are allowed to have are defined in statements called deftemplates. An example of a deftemplate for a book is shown in Figure 18. Actions like send-late-notice are user defined functions that can be either in the Jess language (deffunctions) or in Java (Userfunctions).

```
(deftemplate book
        (slot name)
        (slot author)
        (slot ISBN_number)
        (slot status)
        (slot borrower))
```

Figure 18 Example Deftemplate

A typical expert system has a fixed set of rules while the knowledge base changes continuously. However it's an empirical fact that, in most expert systems, much of the knowledge base is also fixed from one rule operation to the next. Although new facts are being added and old ones get removed all the time, the percentage of facts that change per time unit is rather small. For this reason the obvious implementation of an expert system is rather inefficient. The obvious implementation would require keeping a list of rules, and continuously cycle through that list to see if any rules left hand side (LHS) has been made true by checking each rule against the knowledge base. This is very inefficient, since most of the results of each cycle will be the same in the next cycle. Since the knowledge base is fairly stable, every cycle the same facts are checked against the same LHSs of the rules. The complexity this algorithm gives is of the order O(RF^P). Where R is the number of rules, F the number of facts, and P the average number of patterns per rule LHS. This increases dramatically if P increases. This is not a good solution for any expert system.

Jess instead uses a very efficient method called the Rete algorithm. This algorithm was the basis for a whole generation of expert system shells: OPS5, its descendant ART, and CLIPS. In the Rete algorithm, the inefficiency as described above is solved by remembering past test results across iterations in the rule loop, meaning only new facts are tested against any rule LHSs. Additionally new facts will only be tested against the rule LHSs to which they are most likely to be relevant. As a result the complexity drops to O(RFP), which is linear in the size of the knowledge base.

The Rete algorithm is implemented by building a network of nodes, each representing one or more tests on a rule LHS. Facts that are added are processed through this network of nodes. At the bottom of the network the nodes that represent individual rules. When a fact filters all the way down the network, it has passed all the tests of a particular rule, and this set becomes an activation. The RHS of the associated rule will be fired if the activation is not first invalidated by the removal of one or more facts that make the activation set incomplete.

There are two kind of nodes in the network: one-input and two-input nodes. One-input nodes perform tests on individual facts, while two-input nodes perform tests across facts and perform the grouping function.

If we have these two rules, they can be compiled into the network of Figure 19:

```
(defrule example-1                  (defrule example-2
        (x)                                 (x)
        (y)                                 (y)
        (z)                         =>)
=>)
```

The nodes marked x?, y?, and z? test if a fact contains the given data, while the nodes marked + remember all facts and fire whenever they have received data from both their left and right inputs. To run the network, Jess presents new facts to each node at the top of the network when they are added to the knowledge base. Each node takes input from above and sends its output downwards. A single input node generally receives a fact from above, applies a test to it, and if the test passes, sends the fact downwards to the next node. If the test fails, the one-input nodes simply don't do anything. The two-input nodes have to integrate facts from their left and right inputs. They must remember all facts presented to them and attempt to group facts arriving from their left input with facts arriving from their right input, to make up complete activation sets. A two-input node therefore has a left memory and a right memory.

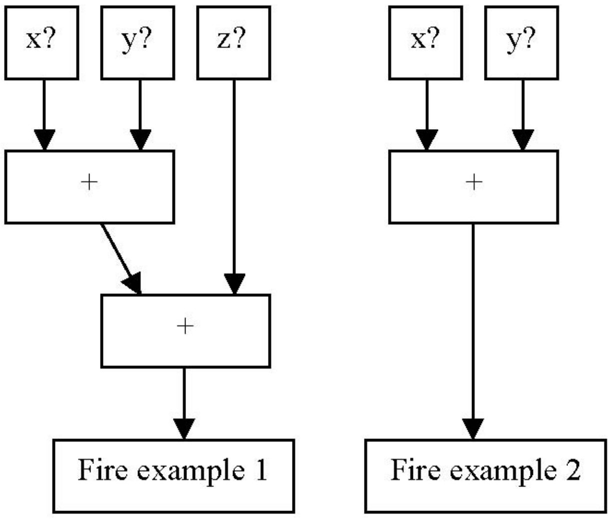


Figure 19 Network version 1

Its convenient to divide the network into two logical components. The single input nodes comprise the pattern network, and the two-input nodes form the join network. There are two simple optimisations that can make the Rete algorithm even better. The first is to share nodes in the pattern network. In the network in Figure 21 there are 5 single input nodes, while there are only 3 distinct ones. We can adjust the network to share the double nodes, as can be seen in Figure 20.

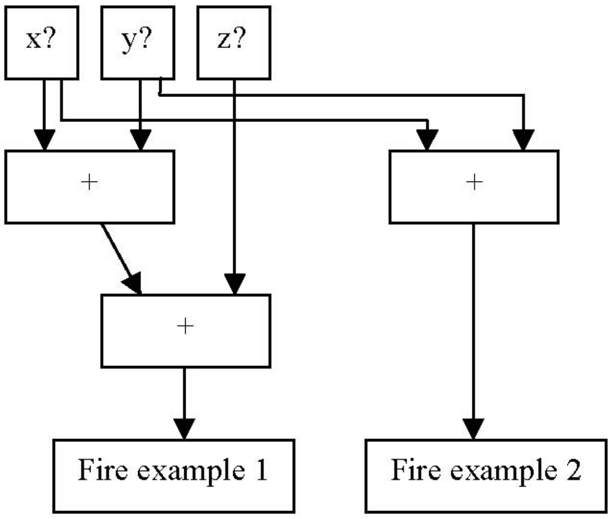


Figure 20 Network version 2

But this is obviously not the only redundancy in the network. We see that there is an identical two-input node in the join network, which is integrating x, y pairs. When we share that node as well we get to the situation in Figure 21.

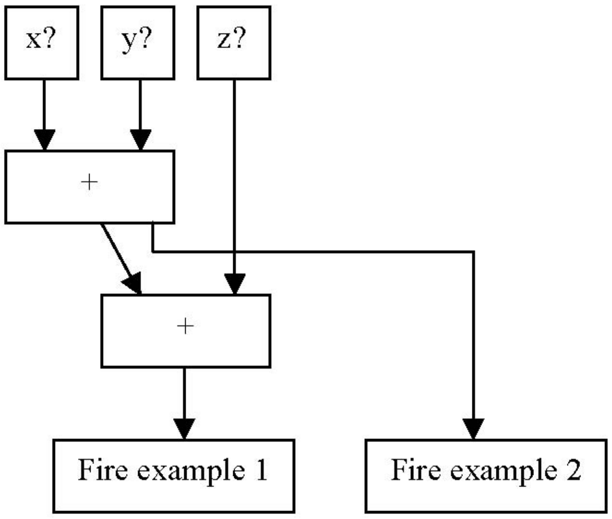


Figure 21 Network version 3

Chapter 4: Global Design

In this chapter I will discuss the requirements and constraints of the system, I will explain how the system is built up from different components, what the responsibilities of these components are, and how they are designed.

4.1. Requirements and Constraints

When developing any system, there are some requirements and constraints that needs to be taken in mind. I will take the problem description to split up the different requirements into workable components.

Design and implement multimodal system for mobile devices (1), capable of reporting crisis and emergency situations via wireless or high speed mobile network and using a map of the surroundings (2), which is expressive enough to handle complex and unexpected situations (3), yet intuitive enough to use without making (a lot of) errors (4). The system should be language independent, allowing the user to add icons, text and different drawings to the report (5). The system should be intelligent enough to assemble and maintain a correct and up to date world model (6). It should detect possible errors in the form of missing, double and wrongly placed icons (7). Furthermore the system should be dynamic in the sense that new concepts and rules can easily be added (8). It also should provide data availability against power down (in client and server side), data reliability and data security (9).

The requirements that can be found in this problem description are discussed below:

1) In terms of mobile devices I will stop only on a Google Android platform, which during this project is still in development and there is no real devices. The application is tested on provided emulator that comes along with the development tools a released to the time of starting of the project.
2) For the purposes of the project I will need a flexible and easy to use solution for the map of surroundings, which in this project will be Google Maps Service.
3) To make the system expressive enough I will need different categories of icons for people, events, transportation and buildings. Within these categories several icons are needed for more specific information. Just placing a man on the map doesn't do much good, if it's not specified he is e.g. a fireman. Even when there is a good icon to represent the concept, I will need a way to add even more information. That's why the icons will have several attributes. In the case of flames these attributes will be the size of the flames, the intensity and the status (increasing, decreasing, under control).
4) To prevent a lot of errors, the GUI should be intuitive and easy to use. It should be clear which category and icon is selected and they should be added to the map with just clicking on the location. To provide extra information, an attribute window will pop up where the values of the attributes can be given. The values can be selected out of a small list, this decreases the chance of making a wrong selection, and eliminates the chance to make an illegal selection. When icons are placed, the user should be able to delete them again, or to inspect or alter its attributes. To prevent placing icons on the wrong location, the user

should be able to easily zoom in and out of the map, to be able to place the icon exactly where it should be.

5) To make the system suited for iconic communication in a crisis situation I need to have icons that represent concepts in a crisis, and map of the surroundings to place the icons on. I will also need a way to allow the users to add a drawing and text to their reports that will be handled by server.
6) In order to assemble and maintain a world model I will collect all information at one server. To create a world model out of this information I could store it all in some sort of database, which will have to be kept consistent at all times. I will choose a client-server implementation for this, where the many clients are the reporters of the crisis, and the server is the part that keeps a consistent world model and distributes all information among its clients.
7) To detect missing, double and wrongly placed icons I need some intelligent agent, that constantly works on the information that's being gathered. This means it will have to work on the database of collected information.
8) In order to make the system dynamic, it's useful to store all its information about icons and their rules in some sort of database. The entries of this database should be easy to edit, and the database should be extendible. If I keep this database in separate files that are read by the system on start up, it's possible to adjust rules, and icons in a way that does not require the entire system to be recompiled.
9) In order to keep system secure and protected against data loss I will need to use techniques to distribute the information in a separate redundant server and/or in clients. As I mentioned before the application is designed to be used in a wireless or mobile networks which does not provide data security I will use some encryption methods for securing the data and also for providing data reliability.

4.2. 3.4 Overview

As said the system will work with one or more servers and multiple clients. The clients collect the information about their surroundings on their map and send it to the server. The server, in turn, will make a consistent world model of all the gathered information and send it back to the clients. The clients can not interact with each other directly, but all communication will be done via the server.

The basic input of our system are the observations of the users. To reduce the ambiguity and to come up with a shared view of the world, the system will provide the set of icons. Nevertheless, the observations can still be somewhat ambiguous because of the following reasons:

- Observers miss objects in the scene; they can overlook certain events or have a different view on what is important to report.
- Observers are remote in time. Crisis events occur and develop over time, so the observations are time dependent.
- Observers are remote in location. Observers are positioned at different locations and can see different things, or from a different angle.
- Observers can either report with the use of the map, or just with icon strings (when these systems are integrated). Both types of messages are supposed to be consistent and complementary.

In Figure 22 an overview of how the different components interact is shown. The XML files contain information about the icon properties and all possible scenarios. They also contain parameters that influence the rules in the Server. The reasoning of the server is only in Jess. Setting up Jess engine including XML parsing and communication with clients is implemented in

Java. For the communication with clients I designed and implemented a separate server that retranslates the messages between Jess servers and clients. The client is done entirely in Java for Android platform, it gets information about icon properties from a XML files and in the client there is a mapping between icons names and corresponding resources in the system. A change in the XML files will cause other properties to appear in the client, and other icons and rule parameters will be used in the server as well. When the XML files are changed, server need to start up again, in order to let the changes take affect. But client need to be recompiled because of the organization of the internal application resources.

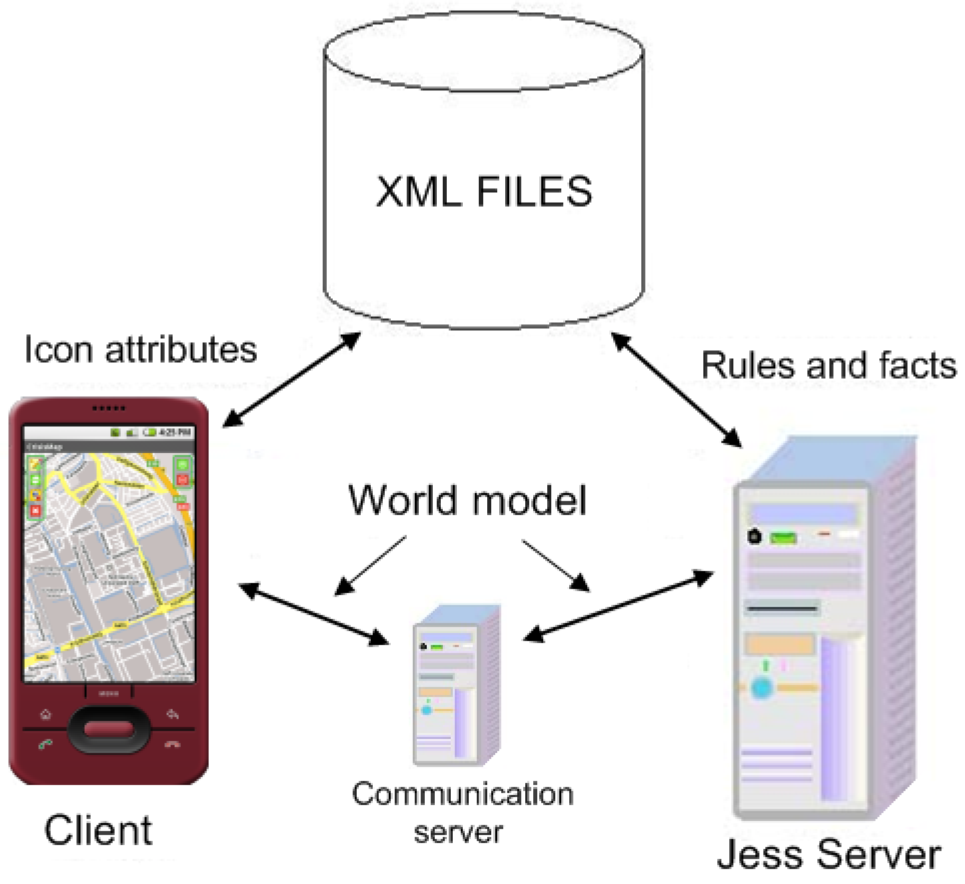


Figure 22 Overview of the components

The server will send some information about scenarios and will suggest some icons that can be placed next.

The scenario information will consist of the predefined scenarios and the probability values that the server has added to them. The values are not in percentages, but are a number that the intelligence of the system will award to each scenario. The higher the number awarded, the more likely it is that the scenario is happening. The scenario information is to give the users of the clients a quick idea of what is happening around them, and to make better predictions of expected icons.

The suggestions for new icons are some feedback for the user. The server has some expectations of what icons it will be receiving next. If an icon is missing, the server will notice this by the information of the XML files. If a fireman is reported, but there is no fire truck, the system will be likely to suggest placing one. The suggestions for placing new icons come from both individual icon relations (icon:flames icon:smoke) and relations between scenarios and icons (scenario:bomb scare icon:bomb). Providing suggestions like these, I believe, will add to a more accurate report. Things the reporter might have missed, or did not find important enough to report will be more likely to be reported now. The user will be actively looking for the concept represented by the icon of the suggestion.

4.3. Design of the XML Files

The XML files will form the basis of all knowledge in the system. In the first place, all the used icons are defined in these files. The icons will be summed up, and each icon will have its own attributes, as defined in these files. Furthermore these files will contain parameters that influence the intelligent behaviour of the system as a whole. Finally the XML files will contain information about emergency scenarios.

In my system I will need two separate XML files. The first one contains all the icons and the inter icon relations. The DTD file is defined as follows:

```
<!ELEMENT iconlist (group*)>
<!ELEMENT group (icon*)>
<!ELEMENT icon (icon_name, slot*, next_icon*, previous_icon*)>
<!ELEMENT icon_name (#PCDATA)>
<!ELEMENT slot (slot_name, slot_value*)>
<!ELEMENT slot_name (#PCDATA)>
<!ELEMENT slot_value (#PCDATA)>
<!ELEMENT next_icon (icon_name, chance, timespan)>
<!ELEMENT previous_icon (icon_name, chance, timespan)>
<!ELEMENT chance (#PCDATA)>
<!ELEMENT timespan (#PCDATA)>
```

Figure 23 DTD of the icon XML file

A valid XML file, as defined by Figure 23, has an element iconlist, which is the main structure. I want to have icons divided in categories, so I make one big list, which can contain multiple groups. The iconlist can contain zero or more groups, as indicated by the *. Each group, in turn can contain zero or more icons. Each icon contains exactly one icon_name, zero or more slots, zero or more next_icons, and zero or more previous_icons. The icons name is defined as #PCDATA, an arbitrary string of characters. The slots of an icon is where its attributes are stored. It contains the name of the attribute and the possible values the attribute can take on. The inter icon relations are defined in the next_icon and previous_icon fields. The fields have the name of the next, respectively previous icon, a chance and a timespan. The chance defines how likely it is that if the icon is placed, a next_icon or previous_icon is required as well. The higher the number in the chance slot, the more likely. The timespan is currently not used, but can add extra information about how much time may pass until the relationship 'expires'.

```
<iconlist>
      <group>
            <icon>
                  <icon_name>policeman</icon_name>
                  <slot>
                        <slot_name>number</slot_name>
                        <slot_value>1</slot_value>
                        <slot_value>2</slot_value>
                        <slot_value>3-5</slot_value>
                        <slot_value>5-10</slot_value>
                        <slot_value>10+</slot_value>
                  </slot>
                  <slot>
                        <slot_name>status</slot_name>
                        <slot_value>busy</slot_value>
                        <slot_value>idle</slot_value>
                        <slot_value>wounded</slot_value>
                        <slot_value>dead</slot_value>
                  </slot>
                  <previous_icon>
                        <icon_name>policecar</icon_name>
                        <chance>2</chance>
                        <timespan>1</timespan>
                  </previous_icon>
            </icon>
      </group>
</iconlist>
```

Figure 24 Fragment of the icon XML file

The second XML file I will use contains the information about scenarios. Its DTD is given below.

```
<!ELEMENT scenariolist (scenario*)>
<!ELEMENT scenario (scenario_name, icon*)>
<!ELEMENT icon (icon_name, chance, slot*)>
<!ELEMENT icon_name (#PCDATA)>
<!ELEMENT slot (slot_name, slot_value*)>
<!ELEMENT slot_name (#PCDATA)>
<!ELEMENT slot_value (#PCDATA)>
<!ELEMENT chance (#PCDATA)>
```

Figure 25 DTD of the scenario XML file

As we can see the scenariolist contains zero or more scenarios, each consisting of multiple icons. In the icon fields is the name of the icon is defined, with its chance of being in the scenario, and possible slots that are relevant. Below, in Figure 26 is a small sample of the scenariolist.xml file.

```
<scenariolist>
      <scenario>
            <scenario_name>carcrash</scenario_name>
            <icon>
                  <icon_name>policeman</icon_name>
                  <chance>3</chance> </icon>
            <icon>
                  <icon_name>crashedcar</icon_name>
                  <chance>5</chance> </icon>
            <icon>
                  <icon_name>roadblock</icon_name>
                  <chance>2</chance> </icon>
            <icon>
                  <icon_name>victim</icon_name>
                  <chance>3</chance> </icon>
            <icon>
                  <icon_name>helicopter</icon_name>
                  <chance>1</chance> </icon>
            <icon>
                  <icon_name>policecar</icon_name>
                  <chance>3</chance> </icon>
            <icon>
                  <icon_name>ambulance</icon_name>
                  <chance>3</chance> </icon>
            <icon>
                  <icon_name>nurse</icon_name>
                  <chance>3</chance> </icon>
            <icon>
                  <icon_name>firetruck</icon_name>
                  <chance>1</chance> </icon>
            <icon>
                  <icon_name>fireman</icon_name>
                  <chance>1</chance> </icon>
      </scenario>
</scenariolist>
```

Figure 26 Fragment of the scenario XML file

For a complete overview of the used XML files see Appendix B: XML Files. To understand how I made the inter icon relations, and scenario icon relations.

4.4. Design of the Jess Component

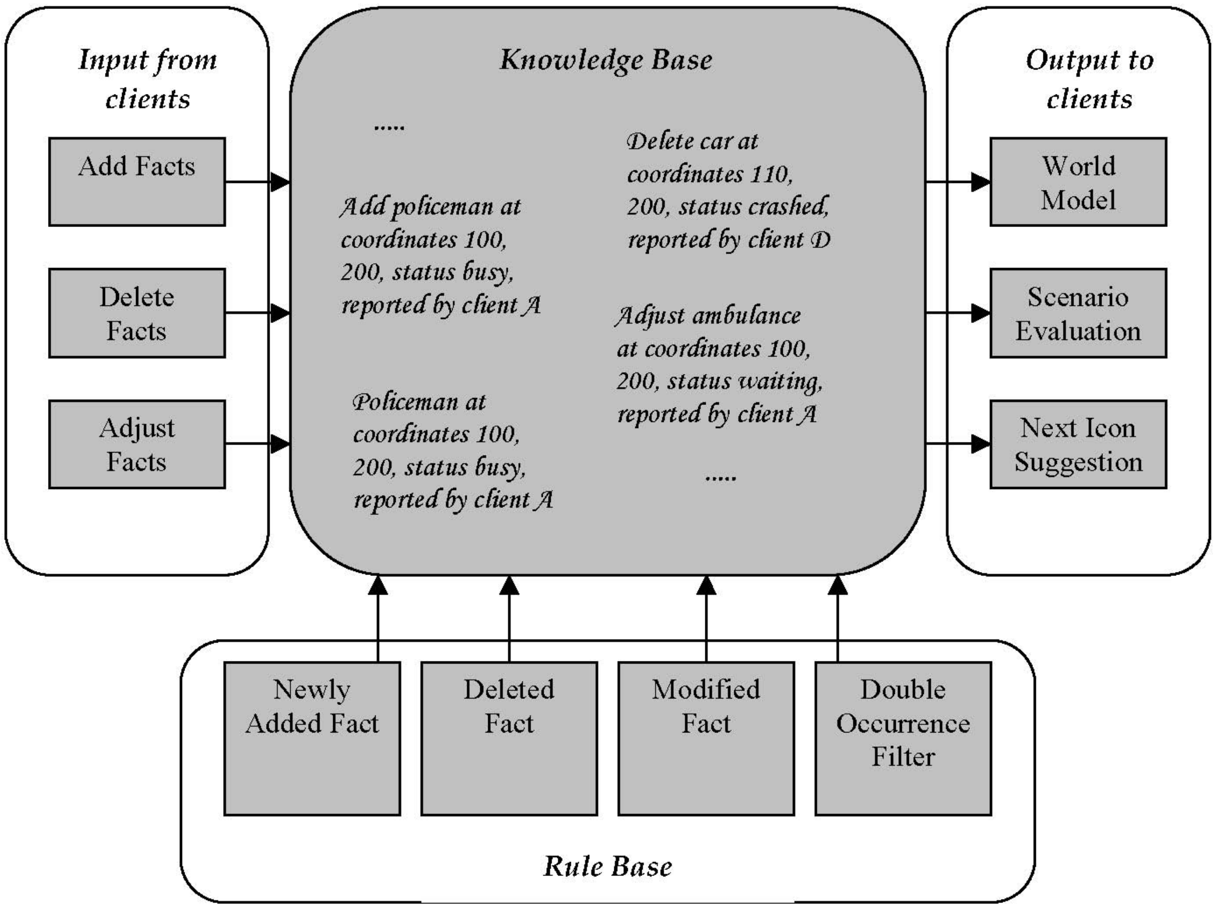
My first problem in designing an expert system is to make the knowledge I need explicit. A common way for knowledge elicitation is to interview experts [Cha05] or to get the knowledge from documentation and manuals. The focus of my project is more on the design and implementation of a report tool. To test my system I defined the rules based on common knowledge about cricises. The rules and concepts was in my case extracted from the news articles in Appendix A.

It can be questioned if a rule based approach is appropriate for our system. A common AI procedure is to start with a deterministic rule based system and as a next step to take a probabilistic approach and to design a Bayesian Belief Network. This will be further discussed in the Recommendations. We have chosen an incremental, prototype based approach, to first prove that the concept of such a system is worth further investigation.

The Jess component will contain the systems World Model, based on reports from the clients. The knowledge base of Jess will be kept consistent and up to date by constantly running the rule base on it. When we look at the functionality of the Jess component, we can see that it works as a virtual blackboard, with some intelligent agent that keeps it clean, consistent and up to date. This agent is defined in the rule base. Every client can write information on it, the reported input is then written down on it without question. Then the agent does its work by performing certain functions on it. The resulting new world model is then send back to the clients. In Figure 27 an overview of the Jess component is given.

As can be seen, the clients input will consist of new facts, deleted facts and modified facts. The facts are in this case obviously the placed, deleted and modified events and icons. When input is received, the rule base will be applied. There are rules for added, deleted, modified and doubly placed events and icons.

The rule for adding new events and icons will add the event's corresponding icons to the list of placed icons that the knowledge base keeps track of. Deleting facts will delete the specified fact, and update the list of placed icons again. The modify rule will search the fact that need modification and edit it. The rule that searches for doubly placed events will scan the knowledge base for events that are very similar and combine them into one fact. This rule only applies if the double events are in the near vicinity of each other, are not reported by the same client and have similar icons in them. The thought behind this is if one client sends two reports for events that are close to each other, there probably are two distinct events, while if two clients both send the same information for them, there is probably just one event happening.



Input from clients
Add Facts
Delete Facts
Adjust Facts
Knowledge Base
.....
Add policeman at coordinates 100, 200, status busy, reported by client A
Policeman at coordinates 100, 200, status busy, reported by client A
Delete car at coordinates 110, 200, status crashed, reported by client D
Adjust ambulance at coordinates 100, 200, status waiting, reported by client A
.....
Output to clients
World Model
Scenario Evaluation
Next Icon Suggestion
Newly Added Fact
Deleted Fact
Modified Fact
Double Occurrence Filter
Rule Base

Figure 27 Overview of the expert system

After the rule base is applied to the knowledge base the resulting world model is send to the clients, together with the scenario information and next icon suggestions. These last two outputs are acquired by performing a function on the knowledge base that calculates the values for the scenarios and suggestions. The Jess components are discussed in more detail in Chapter 4: Implementation.

4.5. Design of the Java Component

The Java component in the system is responsible for everything besides what's included in the XML and Jess components. This means Java is responsible for the interface of the clients, the network, and the integration of the XML and Jess components. I will discuss each of these in this chapter, and in some more detail in Chapter 4: Implementation.

4.5.1. Graphical User Interface

The Graphical User Interface (GUI) of the client is made in Java and XML layout files. All configurations, application windows, services are described in XML files. The icon properties that will be used are also extracted from the XML files. Because of the architecture of the Android application it's impossible to add new icons to the application without recompiling it. For easy building and installing the application I the prepared an ant build XML file.

As is stated in the projects problem description, the goal of the system is to communicate with icons using a map. So obviously the icons and the map need to have their place in the GUI. Besides these elements, there also need to be some placed reserved for control elements. Things that fall in this category are a delete tool, an inspect option and some zooming possibilities.

Since the clients will be receiving some information about the scenario that they are finding themselves in, there needs to be some room reserved for displaying this information as well. Finally we need some place to display any suggested icons. After many prototypes, the final interface of the main screen that I developed can be seen in Figure 28.

As said before, the application is developed to fit on a Android emulator and future real devices, and therefore has a resolution of 320x240 pixels. All of the space is filled up with the map. All buttons and icons are places on the map.

The top left button group (1) is reserved for the main control icons, these are in order: creating new event, requesting world update, disconnecting from server, is closing the application without disconnecting from server. The top right buttons group (2) is reserved for zooming utility buttons. User can manipulate the map by zooming buttons, moving the map with his finger (in the emulator this is the mouse cursor) and zooming with double click on the screen.

To report an event the user can navigate through the map so he can see desired event location on the screen, select the new event button (most top left button) and then to point the event location on the map. The map will center to that point. At the place of that click, on the map will appear an arrow that points the precise location and above will appear the event control icons (4). At the bottom of the screen will appear icon categories (9), as they are described in icons XML file. Although there are only 7 categories in the figure, there is room for 1 extra category. If icon groups are more than places in the group, the group will grow automatically with one or more rows. This is valid also the icons in every group (8) and proposed icons from server (7). With selecting group icon all icons from that group will appear above. To add an icon to current event (5) user have to click (tab) twice on the desired icon, with that icon properties window will appear (see Figure 29), so the reporter can alter or leave the properties to match the Real World. To reduce errors each icon can be added only one to given event, to do so when icon is added to the event then the same icon is deleted from group list for that event, until user removes it from that event. The only icon that can have duplicates is drawing icon.

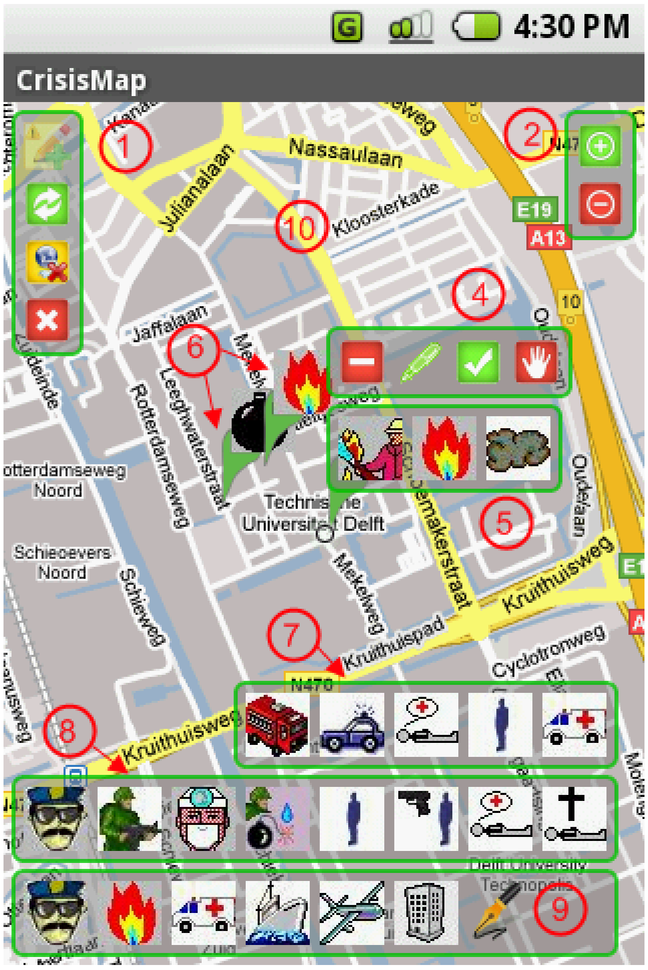


Figure 28 The GUI of the report tool

In any time user can alter icons properties, to remove any given icon, to finish and send the final report to the server or to cancel the report. In order to edit the icon properties, reporter has to select the desired icon from the event and then to select the edit button from event control buttons. On the screen will appear the properties window which will allow him to alter the properties values. He can stop the edit in any given time with selecting the stop option at the bottom of the window. Remove option has two different functions depending whether there is selected icon in the event or not. If there is selected icon the remove option will delete the icon, otherwise it will delete the last added icon. If reporter is finished with his report he can send the final report to server with selecting the approval option. This will remove icons groups, their icons and proposed icons from the screen and also to hide the event icons leaving only the first icon or the icon that corresponds to the suggested scenario and event control buttons.

During the report the reporter in any given time can manipulate the map so he can have clear view of the surroundings. He can also request an update his map with any reported events (6) from other users.

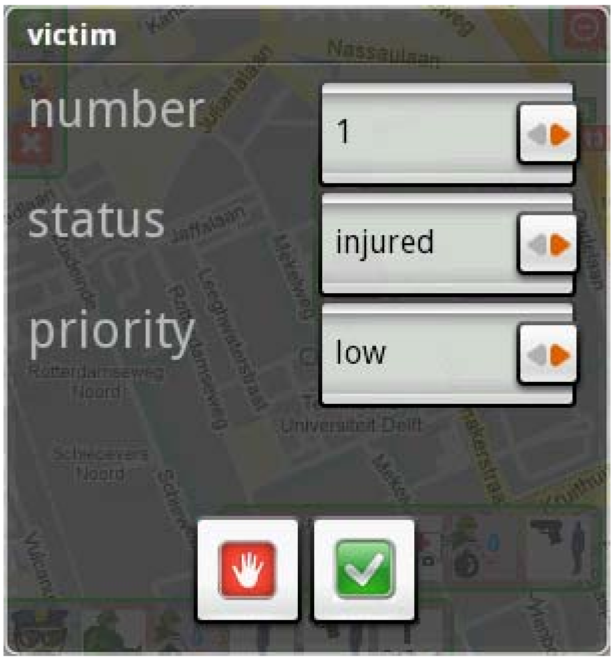


Figure 29 Icons properties window

Before the reporter can user the application he has to register itself in the system. This is done when the application is started for the first time as it’s shown on Figure 30.

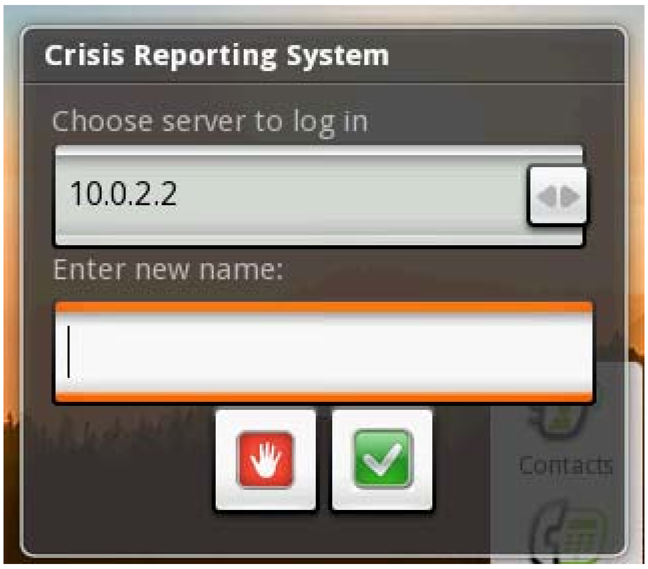


Figure 30 Login screen when the application is launched for the first time

It’s not allowed existing of two reporters with same user names. So when new reporter enters his name, the name is send to Jess servers, the name is checked and verified and the clients is notified if the name is accepted or rejected. If the name is rejected the user can try again to enter a different name. If the application is launched after first successful login, then the login screen is simpler. The user can choose a server to login and an option to change his login name if this is other user. This can be done by selecting the middle button of the screen (see Figure 31).

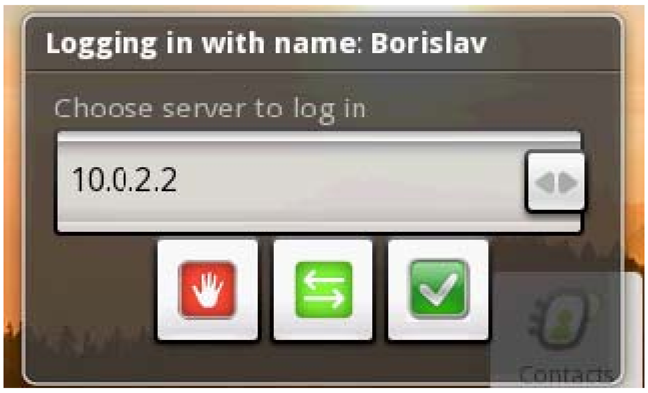


Figure 31 Login screen

4.5.2. The Network

Reasonable decision is to build up the communication backbone on JADE [JADE], which seems to be the most promising framework for mobile networks at the moment, but because of the lapse of the support in Android platform this is not a solution. So a decided to develop mine own network architecture. The result is shown on Figure 32.

Communication channel between clients and Jess servers consists of three parts: through internet from the client to communication servers (CS), from communication servers to Jess servers (JS) through iROS network. Because of the mobile device concept it's impossible to initiate a connection to client. That's why the only one that can initiate a connection is the reporter. Communication is performed via internet through the new fast 3G network (depending on the mobile operator), GPRS, WAP or wireless ad-hoc connection.

I will user a plain socket connection to connect and communication with communication servers. These servers are responsible for resending the messages from reporters to Jess servers and vice versa. Their other purpose is the keep track of connected users and to detect when the user is disconnected to notify the Jess servers. Communication and Jess servers are connected through iROS network. IROS implements asynchronous messaging model known as publish/subscribe model. This model supports publishing messages to a particular message topic. Subscribers may register interest in receiving messages on a particular message topic. In this model, neither the publisher nor the subscriber knows about each other. A good metaphor for it is anonymous bulletin board. The following are characteristics of this model:

- Multiple consumers can get the message
- There is a timing dependency between publishers and subscribers. The publisher has to create a subscription in order for clients to be able to subscribe. The subscriber has to remain continuously active to receive messages.

Using this provides a way of separating the application from the transport layer of providing data. All communication and Jess servers are subscribed to a specific topic. For example CSs are subscribed to System Update Report, System World Update and Communication Servers. The first to topics correspond to messages only for the reporters, which are virtually subscribed to them via the communication server. On the other hand the JSs are subscribed to Jess Servers, System Reports, System Register User, System User Connected, System User Disconnected and System Request World Update.

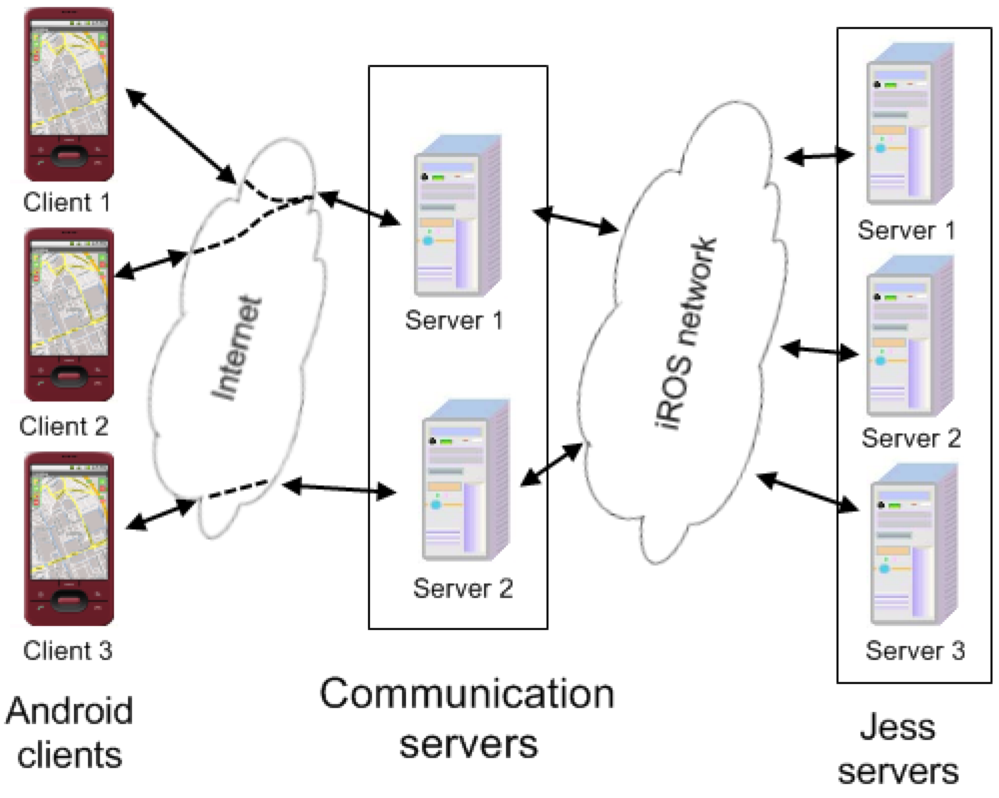


Figure 32 Network architecture

All communication messages are in text format. So they can be read easily if they are intercepted. For that reason will embed verification sequence number and encrypt the message. The message that does not meet the verification will be discarded. For encryption of the messages I will use a simple but efficient algorithm that doesn't require a password or key.

I developed a message format based on EDIFACT specification, which uses very small amount of overhead information. The main concept is that the messages consist of sequence of commands and their arguments, which are interpreted on the other side of the communication channel.

The reporter initiates his virtual connection to the Jess server with a message that notifies his presents and a request to receive the current world status in his area. After that he can send reports, receives icon propositions, server suggestions and disconnect from the server.

4.5.3. Integration

Java is also responsible for the integration of all components with Jess and XML. The XML file about icons is read into Java, and based on the information in them the GUI gets build. The server will read in both the icon information and the scenario information from the XML files. Based on this information the knowledge base and rule base of the Jess engine is build. The integration of the Jess component, is basically importing the right Jess classes and filling the knowledge and rule base dynamically based on the information that was read from the XML files. More information about how this integration is realised is presented in Chapter 4.

Chapter 5: Implementation

In this chapter I will discuss in detail how the system is built. I will show at the hand of diagrams how the different parts of the system work. The most important classes, functions and algorithms will be discussed in detail. Furthermore the Jess component will be explained.

5.1. UML

UML stands for Unified Modelling Language and is a system of diagrams that can specify how systems work. System development focuses on three different models of the system [Bru00]:

1) The functional model is represented in UML by Use Case Diagrams, which specifies the systems functionality from a users perspective.
2) The object model is represented by Class Diagrams and describes the structure of the system in terms of objects, attributes, associations, and operations.
3) The dynamic model is represented by sequence diagrams, state chart diagrams, and activity diagrams. These describe the internal behaviour of the system.

In the next sections I will describe the system according to the first two models, and will use one UML representation per model. First I will discuss the Use Case Diagrams and then the Class Diagrams.

5.1.1. Use Case Diagram

Use cases are used during requirements elicitation and analysis to represent the functionality of the system. Use cases focus on an external view of the system. A use case describes a function provided by the system that yields visible results for an actor. An actor describes any entity that interacts with the system (e.g. a user, another system, the systems physical environment).

In the Use Case Diagram in Figure 33 is shown the use case for logging reporter in the application. He can choose a server to log in, to enter a name to log in or to change previous entered name, to send his information and login or to stop the application.

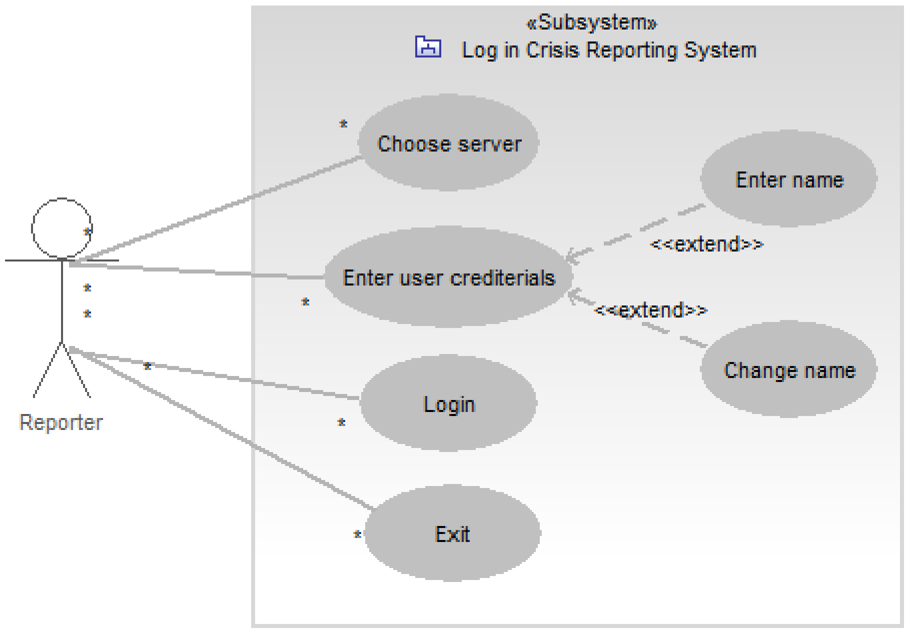


Figure 33 Use case diagram for Logging Reporter

In the Use Case Diagram in Figure 34 is shown the use case for the reporters. They can add events and in order to do that they need to point the location of the event. After that they can add or remove icons from that event and the event is updated after each of the operations. To add or remove icon the icon have to be specified. If specified icon is in new icons then the icon can be added in the event but if the icon is in the current event then the icon can removed or its properties edited. After modification of icon or event the same is updated and the changes are sent back to the server. Depending on the sent data the server responds with different information. Adding a text is other possibility to the user, which also changes the icon representing the text information in the event. Adding drawings to the report also reflects as a change in the event. Users can request an update of the current world status with sending a request message for that. Reporters can disconnected themselves from the server or to stop the application with selecting the appropriate option. The zoom use case corresponds to the functionality of zooming in and out the map of surroundings.

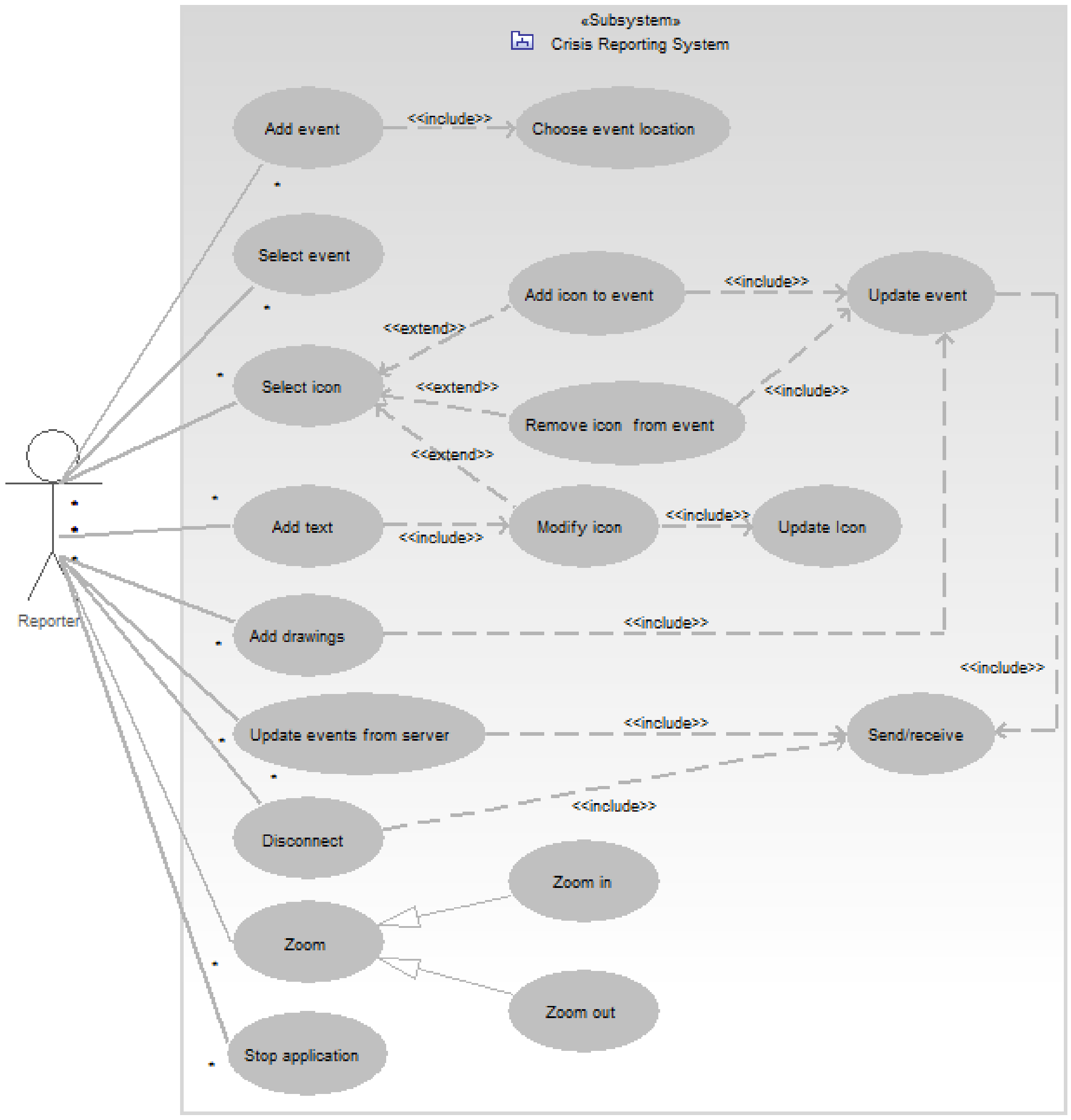


Figure 34 Use case diagram for Reporter

In the next Use Case Diagram (Figure 35) the use case for the client is shown. Just like the reporter gets and inputs information to the client, the client interacts with the server. The client can either send or receive information from the server. In sending will include: send requests to the server; send drawing and send event information. Receiving from the server will include: Receive world status, receive drawings, receive proposed icons that can be added.

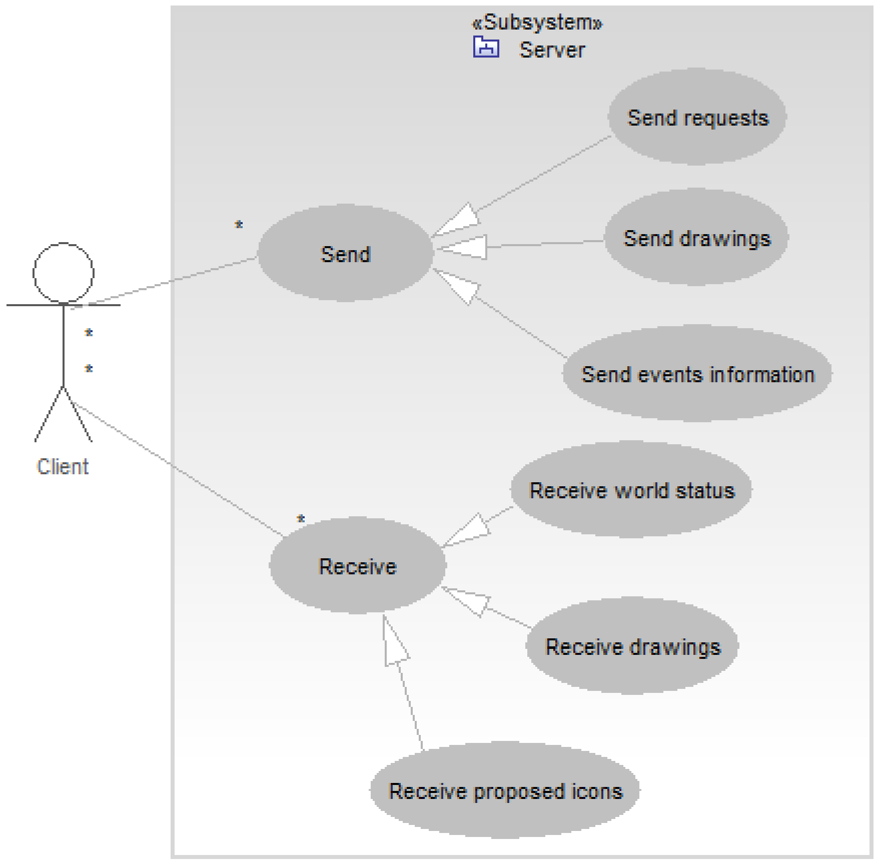


Figure 35 Use case diagram for Client

5.1.2. Class Diagrams

Class Diagrams are used to describe the structure of the system. Classes are abstractions that specify the common structure and behaviour of a set of objects. Objects are instances of classes that are created, modified and destroyed during the execution of the system. An object has a state which includes the values of its attributes and its relationships with other objects. Java programs are build up in classes already and therefore it is easy to create a Class Diagram of it. Because of the size and complexity of the project modules it is difficult to create understandable class diagram for the modules, so the split them in sub modules.

On Figure 36 is shown the class diagram of the GUI of the Android application. The main part of the application is the class CrisisMap, which contains the map and handles all map overlays. The map overlays are used as a drawing surface on the map. I will use them to draw all icons, buttons and drawing on the map. Examples of overlays are: CrisisOverlay, DrawingOverlay, DrawingButtonsOverlay. The last two will be used for creating drawings. On the other hand the CrisisOverlay will handle all functionally concerning managing the icons, reporting events and responding to button click. All images used on the map ether as a button or as a crisis icon are represented by MapIcon class. It will contain an information about icon position on the screen and map, the image itself, the id of the event that icon belongs to, the icon properties and other useful information. PropertiesActivity will represent the window for editing the icons properties, which will be contained in instance of the Properties class. The IconsGroup will be helper class for organizing several icons in a logical group. All icons that will have a buttons functions will have a reference to some of the children of the AbstractAction class. Each icon will have three unique characteristics: name, resource id and global identifier. A mapping between them can be found in IconMapping class. All messages to and from Jess server will be managed by MessageManager class. The incoming messages will be broadcasted from the communication service contained in an instance of the class IncomingMessage. Message broadcasting is the way of communication between the long running processes as services and other parts of the applications in Android. The class that can receive broadcasted messages is called IndentReceiver, an example of that class in this class diagram is the class MessageReceiver.

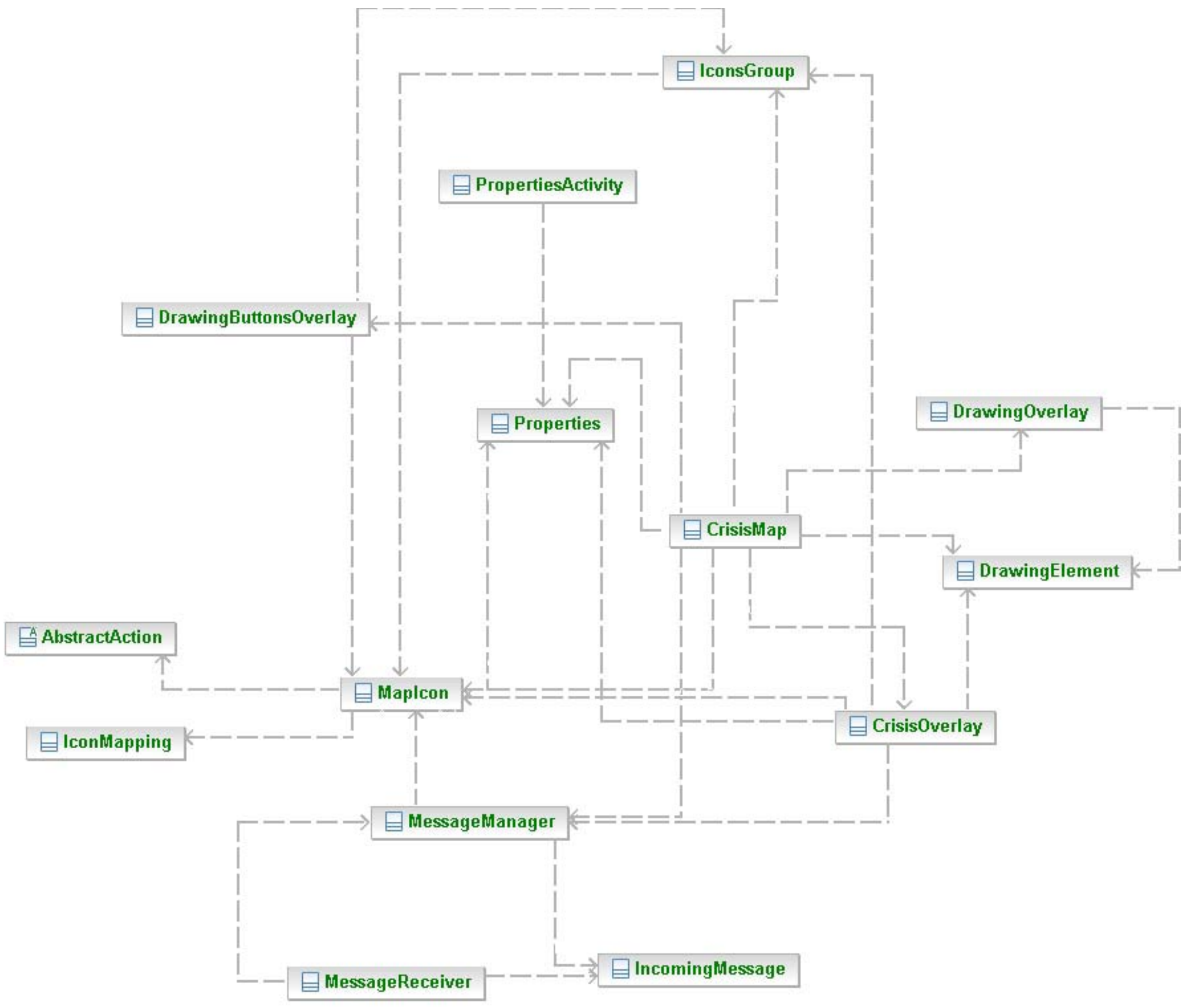


Figure 36 Class diagram of GUI module in the Android application

On Figure 37 is shown the communication server class diagram. The communication service will be responsible for the communication with CS and JS servers. CommunicationService class is the service itself but the thread that will keeps the service alive is the inner class Runner and ServiceHandler is the class that will receive all messages to the service from outside. The inner class IdleConnection will monitor the user network activity and if the user is idle for certain amount of time it will disconnect the user from server. Other important class is the inner class CleanUp that will be responsible for successful delivery of all outgoing messages that are received by CommunicationIntentReceiver class, which will listen for messages from other applications and especially from the GUI of the my application. The class responsible for managing the physical connection with the server is the Communicator class, it will receive, decode, disassemble and verify incoming messages and build and encode all outgoing messages. The incoming messages will be encapsulated in the inner class Message and send to the service. UserIdManager class can be seen also in Jess server and with GaloisLFSR class are responsible for generation and verification of the message sequence numbers embedded in all messages transmitted between the clients and Jess servers. This sequence numbers are important part of the security, because they will authenticate that any given message is from the real client and it's not fake.

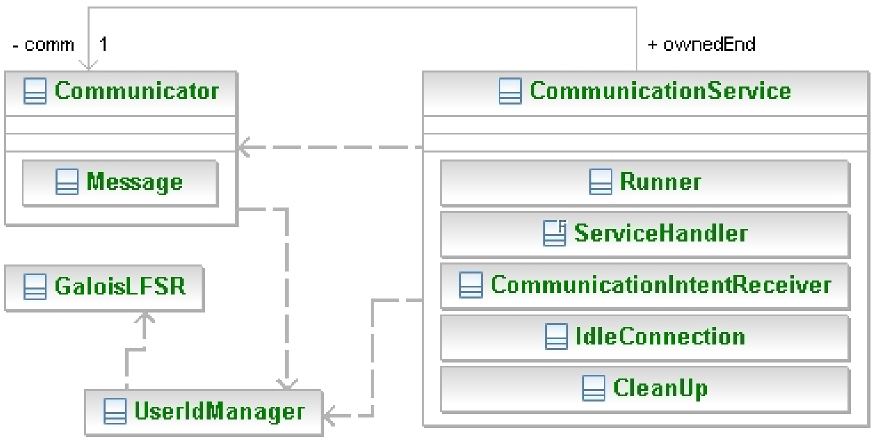


Figure 37 Class diagram of the communication module in the Android application

On the class diagram bellow is shown the communication server. The RetranslatorServer class will represent the actual server that accepts incoming connections from the clients. Each accepted connection will be held by instance of ClientHandler class, which implements the Retranslatable interface. The interface defines methods needed for implementing a bidirectional communication. The exception class will be used for notifying for disconnecting a user from the server.

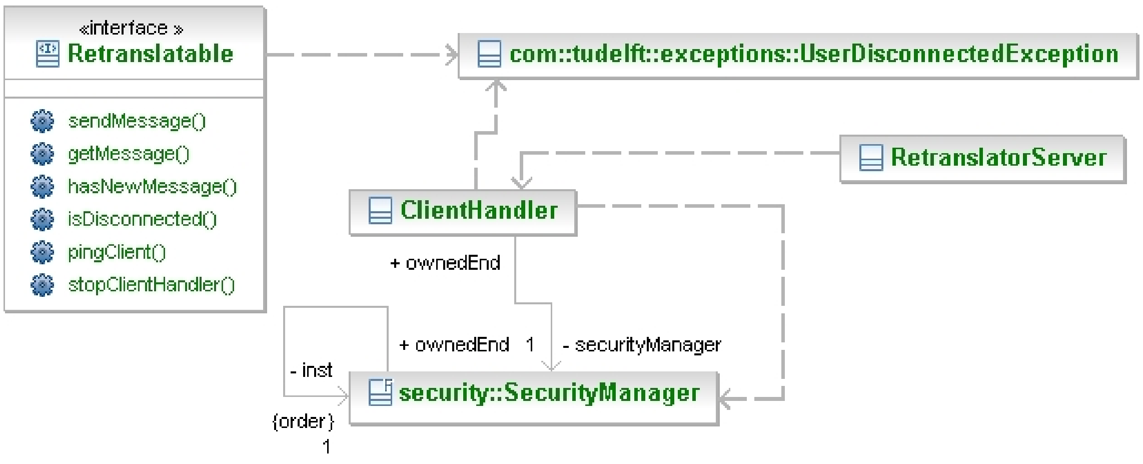


Figure 38 Class diagram of the communication server

On the next few diagrams are shown parts of the Jess server implementation. On Figure 39 are shown the classes responsible for encapsulating and building incoming and outgoing messages. All messages will be encapsulated in instances of the derived AbstractMessage classes. MessageBuilder class will help building a message objects that can be compiled to text that can be send to clients, he will also help to decompile incoming messages. CommandBuilder will compile prebuilt messages to a text command. If something get wrong while compiling a BuilderException will be thrown.

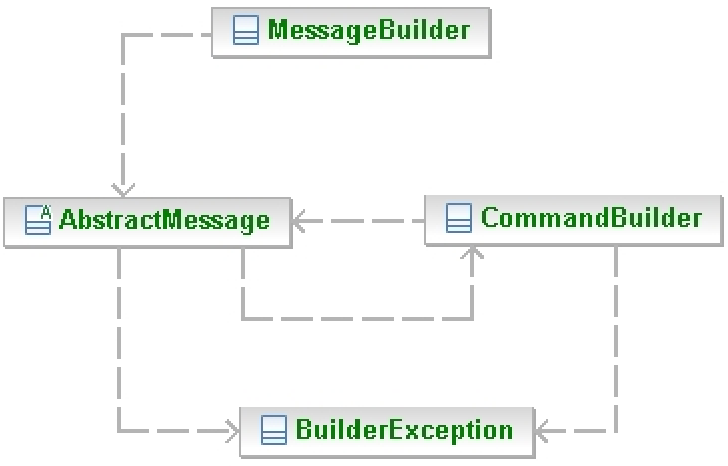


Figure 39 Class diagram of the communication protocol implementation

To make the underlying communication implementation I will use the interface Icommunication shown on Figure 40. Current implementation of the communication interface is IRosProxy class, which handles all requests to IROS network. The abstract class AbstractUserObserver will provide partial implementation for his derived classes. Each of those classes will be handle messages received on a particular iROS topic. For example UserManagementHandler responds to messages connected to user management: registering, approving connection and removing clients to current server, WorldManagementHandler will respond to all requests for world updates, SystemReportsHandler will handle all information sent for currently happening events. JessServiceHandler does not extends AbstractUserObserver because he responds only to other Jess servers connected to the current iROS network. All of the handlers will work with incoming messages that are instances of the IncomingMessage class descendent of Message and AbstractMessage classes. All of the functionally of the Jess server is initialized and combined by JessServer class.

On the class diagram shown on Figure 41 can be found the class AiManager, which is responsible for initiation of Jess engine and keeping it running and up-to-date. Hi also will offer methods for manipulating the knowledge of the server. For initialization of the Jess the AiManager will use the information provided by the XML files and containing in instances of Properties class. The inner classes in Properties are used for organizing and managing the information kept in it. The User class will contains all known information about currently connected users. The functions of the class MapIcon are similar to the class found in the GUI with that difference that here in the server the icon object does not contain an image and only information that is needed for transferring the data from handles to the Jess and vice versa.

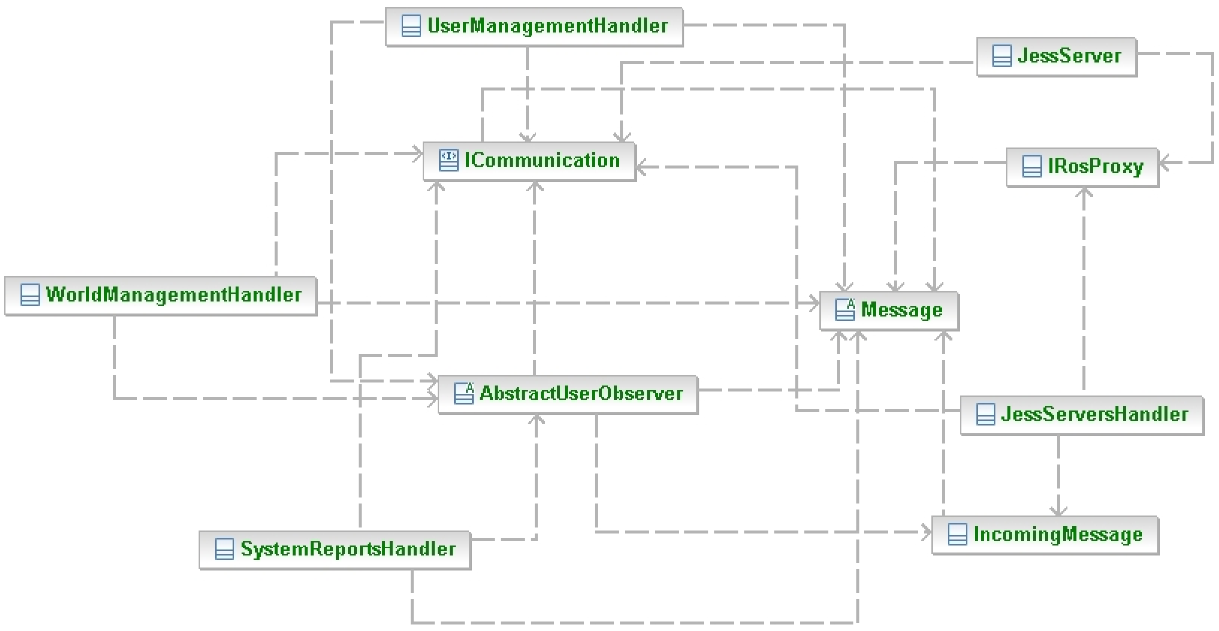


Figure 40 Class diagram of Jess server communication organization

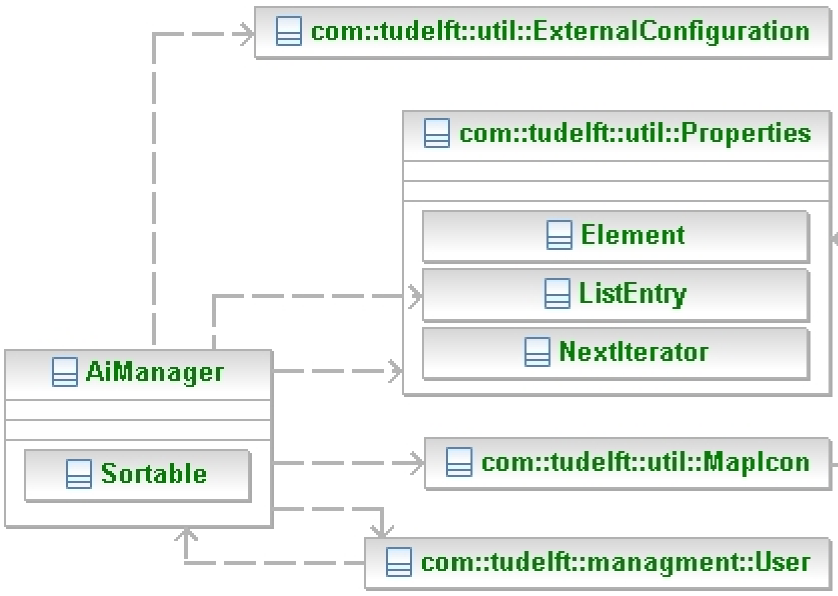

com::tudelft::util::ExternalConfiguration
com::tudelft::util::Properties
Element
ListEntry
NextIterator
AiManager
Sortable
com::tudelft::util::MapIcon
com::tudelft::managment::User

Figure 41 Class diagram of the intelligence of the Jess server

On the last diagram on Figure 42 is shown the security design implementation. The interfaces Cipher and Decipher defines the basic operations that can be performed on a different data times, so the input data to be encoded and respectively decoded. Both interfaces are implemented by Cryptograph class but without actual implementation of the inherited method but only is added some other to help his descendents. The actual implementation is done by CipherIml and DesipherIml classes. The SecurityFactory class is used to create instances of Cipher and Decipher interfaces and in it can be added logic to load external classes that implement those interfaces but with different underlying algorithms. The 'face' of the security implementation is the SecurityManager class, who acts as a proxy to the real implementation and hides all functionality.

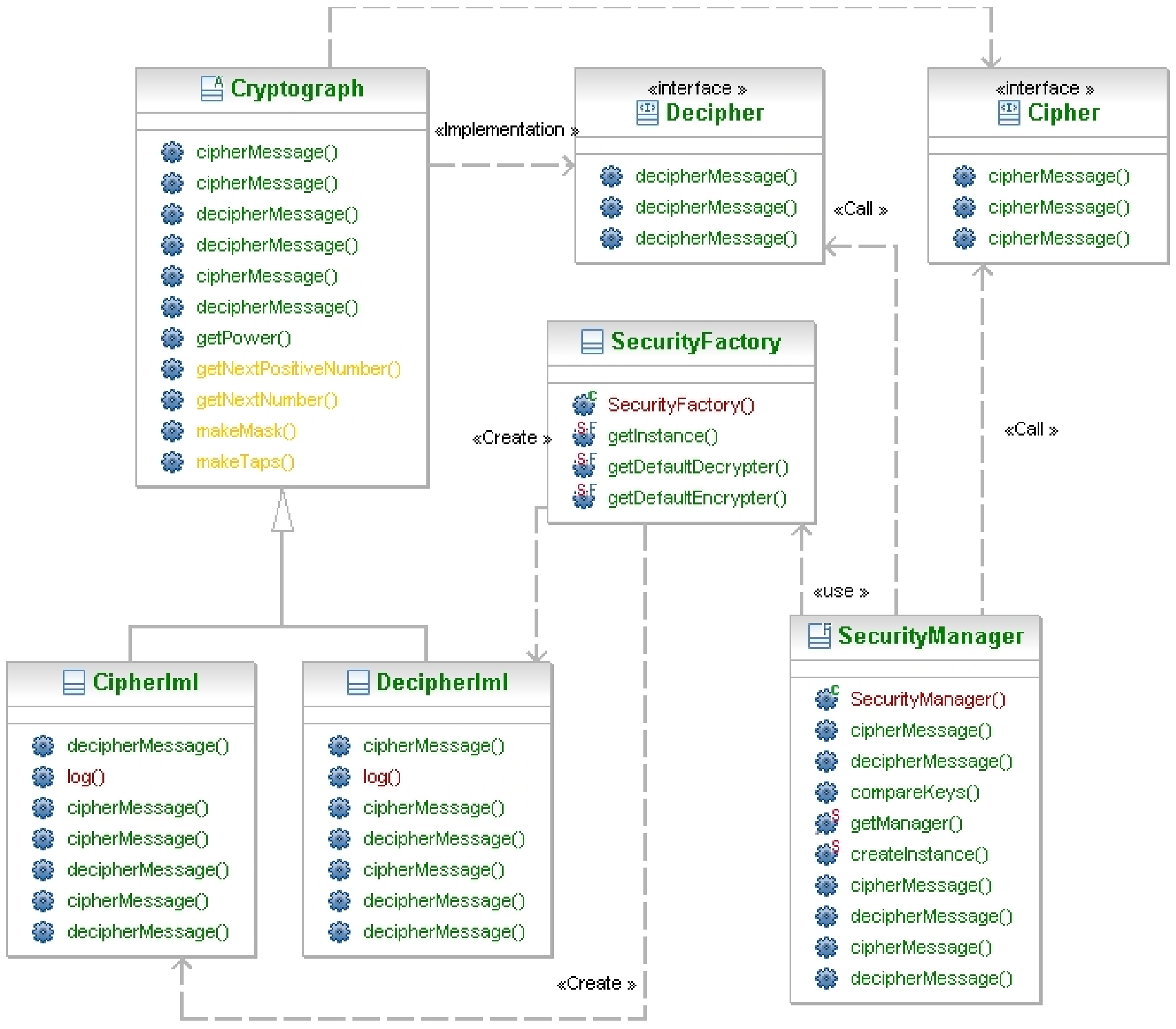


Figure 42 Class diagram of the security implementation

5.2. Jess

In this section I will elaborate on the inner workings of the Jess component. When we look back to Figure 27 we see what functionality needs to be added. All the input that comes from the client will be asserted as facts to the knowledge base. Even for events and icons that need to be deleted, I will assert a fact. The rule base should decide to throw it away or not with its rules for newly added facts, deleted facts and modified facts.

First I will see how I can add facts to my knowledge base, how icons will be deleted, and how modifications take place. After that I will explain how the double events filter works. I will continue with the way the server determines the scenario, and finally I will explain how the system can give suggestions for next icons.

5.2.1. Adding Facts to Jess.

When a client sends its new information, this should be added to the Jess knowledge base. To do this I will add it as a fact. The knowledge base is the collection of all facts that have been inputted to it. In Jess, there are three kinds of facts: ordered facts, unordered facts, and definstance facts.

Ordered facts are simply lists, where the first field (the head of the list) acts as a sort of category for the fact. Here are some examples of ordered facts:

```
(policeman 5199886 4373395,  idle Peter)
(flames 5199806 4373455 expanding huge huge Peter)
```

Figure 43 Example of ordered facts

Ordered facts are useful, but they are unstructured. I want a bit more organization. In object-oriented languages, objects have named fields in which data appears. Unordered facts offer this capability (although the fields are traditionally called slots.) When we rewrite the previous ordered facts to unordered facts we get:

```
(policeman (x 5199886) (y 4373395) (status idle) (supporter Paul))
(flames (x 5199806) (y 4373455) (status expanding) (intensity huge) (supporter Paul))
```

Figure 44 Example of unordered facts

before you can create unordered facts, you have to define the slots they have using the deftemplate construct:

```
(deftemplate template-name
    ["Documentation comment"]
    [(declare (slot-specific TRUE | FALSE)
              (backchain-reactive TRUE | FALSE)
              (from-class class name)
              (include-variables TRUE | FALSE)
              (ordered TRUE | FALSE))]
    [extends template-name]
     (slot | multislot slot-name
         [(type ANY | INTEGER | FLOAT |
                NUMBER | SYMBOL | STRING |
                LEXEME | OBJECT | LONG)]
         [(default default value)]
         [(default-dynamic expression)])*)
```

Figure 45 Deftemplate construct

The <deftemplate-name> is the head of the facts that will be created using this template. There may be an arbitrary number of slots. Each <slot-name> must be an atom. The default slot qualifier states that the default value of a slot in a new fact is given by <value>; the default is the atom nil. The 'default-dynamic' version will evaluate the given value each time a new fact using this template is asserted. The 'type' slot qualifier is accepted but not currently enforced by Jess; it specifies what data type the slot is allowed to hold. Acceptable values are ANY, INTEGER, FLOAT, NUMBER, ATOM, STRING, LEXEME, and OBJECT. Since they are currently not enforced by Jess, we will not use these.

```
(deftemplate policeman "A policeman."
    (slot x)
    (slot y)
    (slot status (default unknown)))
```

Figure 46 Example of a deftemplate for a policeman

The example in Figure 46 would allow us to define facts like this:

```
(assert (policeman (x 5199806) (y 4373455)))
(assert (policeman (x 5199806) (y 4373455) (status idle)))
```

Figure 47 Examples of facts, defined by the deftemplate of Figure 46

Note that the status of the policeman is unknown by default. If we don't supply a default value for a slot, and then don't supply a value when a fact is asserted, the special value nil is used.

Because I want to get all information about which icons I can use from the XML files, need to make deftemplates for every icon in the list. Of course I have to read in the data from the XML file first and then form Jess commands of it. This will all happen in the initialisation phase of the Jess server. When the server is up and running, it will get information from the clients, in the form of the placed icons. These need to be transformed into Jess facts, and asserted into the knowledge base.

Since I will have rules that apply to all icons, rather than just an icon of specific type it would be a good idea to let every type icon extend an overarching *icon* fact. Figure 48 shows how this can be done.

```
(deftemplate icon
    (slot name)
    (slot x)
    (slot y)
    (slot eventID))

(deftemplate policeman extends icon
    (slot number)
    (slot status))

(assert (policeman
         (name policaman)
         (eventID 3435346546)
         (x 5199886) (y 4373395)
         (number 1) (status idle)))
```

Figure 48 Extending from other deftemplates

There seems to be a little bit of overhead, because when I assert a new policeman now, I specifically add the name of the icon, which in this case is the same as the type of the icon. But this overhead will simplify the rules that will work with the icons, because the rule can use an icon as a type but to distinguish the icons byte their *name* slot.

5.2.2. Deleting Facts from Jess

When looking back to Figure 27, we see that the clients can also tell the server to delete some events or icons. Rather than doing this without any questions asked, I would prefer to add a fact that tells the server to delete a certain event or icon. This approach resembles a blackboard where everyone can put sticky notes on with a request. The owner of the blackboard, the server in this case, can then take off the notes and decide what to do with them. The facts that will be asserted in order let the server know what icon to delete is shown in Figure 49. When facts like these are asserted, the rule to delete icons will be triggered if such icon with that name in that event exists. A rule like this is shown in Figure 50. This is the same and with reporters. For each active reporter the server asserts a fact for it the Jess. When a user disconnects a fact is asserted and if there is no reported event from him he can be deleted otherwise he will remain in the memory.

```
(deftemplate icon_to_delete
    (slot name)
    (slot eventID)
    (slot deleted (default FALSE)))

(assert (icon_to_delete (name policaman) (eventID 65741321543)))
```

Figure 49 Template and example for deleting icons

```
(defrule delete-icon
    ?i2d <- (icon_to_delete (name ?name) (eventID ?id) (deleted FALSE))
    ?icon <- (icon (name ?name) (eventID ?id))
    ?event <- (event (id ?id) (icons $?icons))
    =>
    (modify ?i2d (deleted TRUE))
    (modify ?event (icons (complement$ (create$ ?name) ?icons)))
    (retract ?icon))
```

Figure 50 A rule for deleting icons from events

On the Figure 51 is shown the template for deleting user and the rules responsible for the actual removing from memory. The rule will fire when there is a fact user-to-delete and the reporter with the same name and does not exists event that has that reporter name in his supporters list.

```
(deftemplate user-to-delete (slot name))

(defrule remove-idle-user
    ?del <- (user-to-delete (name ?name))
    ?user <- (reporter (name ?name))
    (not (event (supporters $? ?name $?)))
    =>
    (printout t "Found an idle user: " ?name crlf)
    (retract ?user)
    )
```

Figure 51 Template and rule for removing idle users

5.2.3. Modifying Jess Facts

Facts can also be modified. The client will send a list of icons that need to be modified by the server. There is a built-in function to modify facts, which can be seen in the rule on Figure 50. In my system the modification of the icons is done in Java using the method modify (Fact fact, String[] slotNames, Value[] slotValues).

5.2.4. Rules about Double Placed events

Often it will happen that that the server receives double events. If one client reports an event and another client hasn't updated his world yet, this client could report the same event and the server will receive the same or similar information twice. Jess should filter these double occurrences out.

When a double event is reported by only one client there is a big chance that there are actual two distinct occurrences. In this case the Jess engine should not filter one of them out. However, when two clients report a same event in approximately the same place these occurrences are probably the same, and one should be filtered. This is the reason I keep track of which reporter reported which event. The supporters of an event is a list with all the reporters that reported a particular event. At the beginning I will be only the user that reported the event, but during the running of the Jess engine some of the events can be merged and their supporters too. The reporters have a name and a location, which are defined at the start up of the client. The filtering can happen in several ways:

1) Both the occurrences get deleted, and the system adds on occurrence at the average location. This is :

$$newX = \frac{(x_1 + \cdots + x_n)}{n}$$

$$newY = \frac{(y_1 + \cdots + y_n}{n}$$

2) The system calculates which client is closest to its reported event, and deletes all the other occurrences. This might be a reasonable solution because I think the closer reporter to the event will give more accurate report.
3) A combined version of 1) and 2). Although I think that the closer reporter might provide the more accurate report, and I don't think that the other reporter just reports nonsense. That's why I could add weights to the reports of the clients. The closer clients report will get a higher weight that the other reports.

$$newX = \frac{(w_1 x_1 + \cdots + w_n x_n)}{n(w_1 + \cdots + w_n)}$$

$$newY = \frac{(w_1 y_1 + \cdots + w_n y_n)}{n(w_1 + \cdots + w_n)}$$

Now the question remains how to distribute the weights. In practice when this rule is fired there will be one event with at least one supporter already in the knowledge base and one new event is has just been added with one supporter. This means there are two groups, which results in the following formula:

$$newX = \frac{w_1(x_{11} + \cdots + x_{1n}) * groupsize1 + w_2(x_{21} + \cdots + x_{2n}) * groupsize2}{w_1 * groupsize1 + w_n * groupsize2}$$

$$newY = \frac{w_1(y_{11} + \cdots + y_{1n}) * groupsize1 + w_2(y_{21} + \cdots + y_{2n}) * groupsize2}{w_1 * groupsize1 + w_n * groupsize2}$$

Where the weights *w1* and *w2* have either value 1 or 2. The group that is closest to the event gets weight 2 and the other group gets weight 1. I choose to implement this last design.

Besides the filtering I have to check if the events are close enough to each other, if that have similar reported icons and that have to be reported by different users. For the distance I choose 50 meters, computed over the earth surface. The coordinates of the events and the users are stored in the knowledge base in mille degrees and because the earth surface is not flat as we see it. To calculate the distance I used the following formula:

$$t1 - \sin lat1 * \sin lat2$$
$$t2 = \cos lat1 * \cos lat2$$
$$t3 = \cos(lon1 - lon2)$$
$$t4 = t2 * t3$$
$$t5 = t1 + t4$$
$$result = \left(\tan^{-1}\left(\frac{-t5}{\sqrt{(-t5 * t5 + 1)}}\right) + 2 * \tan^{-1} 1\right) * 6366832.7531680002588673358856$$

Figure 52 Formula for calculating the distance between two points on earth surface

In the formula above the variables lat1, lon1, lat2 and lon2 are the coordinates in radians for the first point and second point. "Lat" stands from latitude and "lon" from longitude. The approximation constant at the end is calculated in advance and is used for transforming the result in meters.

The rule that detects probable duplicate events is shown on Figure 53. The rules searches for two evens with different event identifiers, those are closer than 50 meters, are not reported by same reporters and have similar icons (has more that 60% common icons). If such events are found then their information is passed to function that combines them. There the coordinates are calculated and assigned to the first event, its supporters and icons are merged and the second event is deleted and all his belongings icons. The function for combining the events is shown on Figure 54.

```
(defrule filter_double_events
    ?event1 <- (event (id ?id1) (lat ?x1) (lon ?y1) (supporters $?list1)
                (icons $?icons1))
    ?event2 <- (event (id ?id2&~?id1) (lat ?x2) (lon ?y2) (supporters $?list2)
                (icons $?icons2))
    (test (neq ?event1 ?event2))
    (test (< (distance ?x1 ?y1 ?x2 ?y2) 50.0))
    (test (not (has_shared_element ?list1 ?list2)))
    (test (are-they-simillar ?icons1 ?icons2))
    =>
    (printout t "Found double icon " ?id1 ". Matched x: " ?x1 " with " ?x2 " and
y: " ?y1 " with " ?y2 crlf)
    (printout t "The distance between events is " (distance ?x1 ?y1 ?x2 ?y2)
crlf)
    (combine_events ?event1 ?event2 ?list1 ?list2 (create$ ?id1 ?id2)
      (union$ ?icons1 ?icons2) (create$ ?x1 ?y1 ?x2 ?y2))
```

Figure 53 Rule for detecting a double events

```
(deffunction combine_events (?event1 ?event2 ?supporters1 ?supporters2
?ids ?icons $?pos)
    (bind ?supporters_to_add (complement$ ?supporters1 ?supporters2))
    (bind ?supporters_total (insert$ ?supporters1 1 ?supporters2))
    (printout t "The combined supporters are: " ?supporters_total crlf)
    (bind ?x1 (nth$ 1 $?pos))
    (bind ?y1 (nth$ 2 $?pos))
    (bind ?x2 (nth$ 3 $?pos))
    (bind ?y2 (nth$ 4 $?pos))
    (bind ?number_of_supporters1 (length$ ?supporters1))
    (bind ?number_of_supporters2 (length$ ?supporters2))
    (bind ?distance_group1 9999999)
    (bind ?distance_group2 9999999)
    (foreach ?supporter_name ?supporters1
        (bind ?rep (run-query* search_reporters ?supporter_name))
        (while (?rep next)
            (bind ?x (?rep get "x"))
            (bind ?y (?rep get "y"))
            (bind ?n (?rep get "name"))
            (printout t "Found reporter: " ?n " at " "(" ?x ", " ?y ")"
crlf))
        (bind ?dist (distance ?x1 ?y2 ?x ?y))
        (if (< ?dist ?distance_group1)
            then (bind ?distance_group1 ?dist)))
    (printout t "The distance of group 1 is " ?distance_group1 crlf)

    (foreach ?supporter_name ?supporters2
        (bind ?rep (run-query* search_reporters ?supporter_name))
        (while (?rep next)
            (bind ?x (?rep get "x"))
            (bind ?y (?rep get "y"))
            (bind ?n (?rep get "name"))
            (printout t "Found reporter: " ?n " at " "(" ?x ", " ?y ")"
crlf))
        (bind ?dist (distance ?x1 ?y2 ?x ?y))
        (if (< ?dist ?distance_group2)
            then (bind ?distance_group2 ?dist)))
    (printout t "The distance of group 2 is " ?distance_group2 crlf)
```

Continued on the next page....

Continue from the previous page....

```
    (bind ?weight_group1 1)   ;both groups get weight 1 to start with
    (bind ?weight_group2 1)
    (if (< ?distance_group1 ?distance_group2)
        then      (printout t "group 1 is closer and gets a double weight"
crlf)
        (++ ?weight_group1)
        )

    (if (< ?distance_group2 ?distance_group1)
        then      (printout t "group 2 is closer and gets a double weight"
crlf)
        (++ ?weight_group2)
        )
    ;the 'winning group' now has weight 2
    (bind ?new_x (/ (+ (* ?weight_group1 (* ?x1 ?number_of_supporters1)) (*
?weight_group2 (* ?x2 ?number_of_supporters2))) (+ (* ?weight_group1
?number_of_supporters1) (* ?weight_group2 ?number_of_supporters2))))

    (bind ?new_y (/ (+ (* ?weight_group1 (* ?y1 ?number_of_supporters1)) (*
?weight_group2 (* ?y2 ?number_of_supporters2))) (+ (* ?weight_group1
?number_of_supporters1) (* ?weight_group2 ?number_of_supporters2))))

    (printout t "new x: " ?new_x crlf)
    (printout t "new y: " ?new_y crlf)
    (modify ?event1 (lat ?new_x))
    (modify ?event1 (lon ?new_y))
    (modify ?event1 (supporters ?supporters_total))
    (update-icons ?ids ?icons)
    (modify ?event1 (icons ?icons))
    (retract ?event2)
    (assert (event-replacement (oldID (nth$ 2 ?ids)) (newID (nth$ 1 ?ids))))
    )
```

Figure 54 Function for combining events

The code in Figure 54 may seem a bit overwhelming, especially for those who are not familiar with Jess. What the function does is:

- Combine the supporters
- Determine the closest supporter of 'team 1'
- Determine the closest supporter of 'team 2'
- Give the 'winning team' a double weight
- Calculate the new position
- Modify one event to have the combined supporters, new location and icons
- Delete the other events and his belongings icons
- Asserts new fact in the memory that an event with given id has been changed to the new value.

5.2.5. Determining the Current Scenario

At the server side there will be some thoughts about scenario's. The system will read in different scenario's from the XML files, and by looking at which icons are in the world model, it will determine in what scenario we are. The contents of each scenario, and the values the icons get awarded in the XML files. The scenario information will be send back to the reporters as general feedback.

The procedure of determining the scenario for an event is:

- To make a variable for each scenario, and set it to 0.

- Every time an icon gets added, it will award points to the corresponding scenarios.
- When a certain threshold is reached the scenario is believed to be true and set to be the current scenario.
- When another scenario gets a higher score than the current scenario, the current scenario gets exchanged.

To keep track of the likely scenario I will make a variable for each defined scenario for every single event. The code will award points towards each of these variables, if a newly added icon in event would be appropriate for the scenario. The easiest way to do this is simply add a rule for each placed icon, as can be seen in Figure 55.

If I do this for each added icon I would get a score for each scenario. However, there is a problem if I delete icons again. Then I would have to subtract all the awarded points for each scenario. This would mean that in addition of having a rule for every type of icon that gets added, I would also have to make a rule for every type of icon that gets deleted.

```
(bind ?scenario_riot 0)
(bind ?scenario_carcrash 0)
(bind ?scenario_fire 0)
(bind ?scenario_bombscare 0)
(bind ?scenario_shooting 0)

(defrule new_policeman
    ?fact <- (policeman (name ?policeman) (x ?x) (y ?y) (status ?s)
(supporters ?sup))
    =>
    (bind ?scenario_riot (+ ?scenario_riot 4))
    (bind ?scenario_carcrash (+ ?scenario_carcrash 1))
    (bind ?scenario_fire (+ ?scenario_fire 1))
    (bind ?scenario_bombscare (+ ?scenario_bombscare 2))
    (bind ?scenario_shooting (+ ?scenario_shooting 3)))
```

Figure 55 Awarding points to scenarios, with a rule for every icon

```
(deftemplate scenario_chance
    (slot scenario_name)
    (slot icon_name)
    (slot value))
(deftemplate scenario_suggestion
    (slot scenario_name)
    (slot value))
;This will results in facts like:
(assert (scenario_chance (scenario_name riot) (icon_name flames)
        (value 2)))
(assert (scenario_chance (scenario_name riot) (icon_name policeman)
        (value 3)))
(assert (scenario_chance (scenario_name carcrash) (icon_name car)
        (value 5)))
and at start up we will assert the following facts:
(assert (scenario_suggestion (scenario_name riot) (value 0)))
(assert (scenario_suggestion (scenario_name fire) (value 0)))
(assert (scenario_suggestion (scenario_name carcrash) (value 0)))
```

Figure 56 Examples of facts that get asserted at start up of the application

The time to calculate the scenario information is when the client sends his icons. The server will then calculate a new world model, possible scenario and suggested icons. When the server has calculated these things, it can be send back to the client.
To calculate which scenario is going on, Java will fire a Jess function and get back the value of the scenarios. At start up of the application I will add all the scenario_chances, according to the XML file. See Figure 56.

With asserted facts like these I can define a function that calculates the chances for each scenario. I will define 1 variable first, to keep track of what types of icons exist, according to the XML files.

```
(bind $?*all* (create$ policeman soldier bomb flames victim …))
```

Figure 57 The creation of *all* variable

The final function that awards the points to the scenarios and the query for searching them in the memory is shown in Figure 58. It will award points by looking at the icons that are already placed for the particular event, and the values these icons give to the different scenarios.

```
(defquery get-suggested-scenarios
     ; Searches all scenario suggestion for a particular event id
    (declare (variables ?eventID))
    (icon (name ?name) (eventID ?eventID))
    (scenario_chance (scenario_name ?sname) (icon_name ?name)
            (value ?sc-value))
    ?fact-id <- (scenario_suggestion (scenario_name ?sname)
                        (eventID ?eventID) (value ?ss-value))
    )

(deffunction check-suggested-scenarios (?eventID)
    ; With one query we retrieve everything needed
    (bind ?res (run-query* get-suggested-scenarios ?eventID))
    (bind ?value -1)
    (bind ?scenario nil)
    (while (?res next)
        (bind ?sc (?res get "sc-value"))
        (bind ?ss (?res get "ss-value"))
        (bind ?f (?res get "fact-id"))
        (bind ?sname (?res get "sname"))
        (printout t (?res get "name") " appear in scenario " ?sname
                  " with value " ?sc crlf)
        (bind ?v (+ ?sc ?ss))
        (if (< ?value ?v)
            then (bind ?scenario ?sname)
            (bind ?value ?v))
        (modify ?f (value ?value))
     )
    (bind ?res (run-query* get-event ?eventID))
    (while (?res next)
        (modify (?res get "fact-id") (scenario ?scenario))
    )
```

Figure 58 Jess code for checking the suggested scenario for a given event identifier

After executing this function the scenario_suggestion facts are updated, and the suggestion with the highest value can be picked as suggestion.

5.2.6. The Next Icon Predictor

Besides giving the reporter feedback with the newly calculated icons and the most likely scenario, the server will also give some suggestions for icons that might be placed next. This is useful if the reporter forgot to report an icon. According to the placed icons in the current world model, the server will suggest icons to the client that it thinks are missing, or would be a good choice as a next icon. For example, if flames were reported, the server may suggest a smoke icon as well, as long as there is no smoke icon already.

In order to make a reasonable prediction, the server can make use of 3 sources of information:

1) Every icon has a list of probable next icons
2) Every icon has a list of probable previous icons
3) Every scenario has a list of icons.

In the XML file for the icons, besides it's name, attributes, etc, every icon has a list of probable next and previous icons. These icons also have a number attached to them, telling how probable the relations are. These numbers are on a scale from 1 to 5, where 1 would mean possibly and 5 would mean definitely. For example 'fire' would have as probable next icon 'smoke' with probability 5. It would also have 'smoke' as probable previous icon with probability 5, because often smoke is reported first, when the fire cannot be seen yet. The next and previous relations are therefore causal relations.

The server is already keeping track of scenario's. Every scenario has a list of icons that makes up the scenario. When enough of these icons are in the world model, the scenario is probably happening. With this, I have a great source to predict new icons, namely those icons that should be in the scenario that's going on, but are not on the map yet.

To implement a way to suggest new icons, I need to keep track of how many points each predicted icon has been given. Just as in the calculation of the scenarios I will only calculate this when we need to know the results, in order to prevent us from too much overhead in the bookkeeping of the variables. Icons will only be suggested if they are not on the map yet. This will help me in performance, since I only have to calculate the values for icons that are not on the map yet.

I want to calculate suggestions every time the clients send its icons and want to receive an update of the world model. In order to do this, I obviously need a function that will calculate the chances of every icon to be placed next. Since I only need to check for icons that are not on the map yet, we will be using the function *get-placed$* and the variable *$?*placed**, together to calculate which icons have not been placed yet.

The function on Figure 59 looks for all the icons that exist but have not been added in the particular event yet. Then for every icon in this group it starts looking for other icons that can award points to it. Of course the icons that may award points have to be placed themselves. First this is done by looking if aplaced icon has this icon as a next relation, if so the points are awarded to the suggestion. After this I do the same for previous relations, again points get awarded to the suggestion. Finally I look if the suggested icon is present in any scenario. The points awarded for this are of course dependent on the chance that the scenario is actually happening. When this is done for all the icons that are not added yet, I have a list of suggestions and their score. The highest 5 scoring suggestions will be send to the reporter as feedback.

```
(deffunction check-suggested-icons (?eventID)
    (bind $?icons_to_check (complement$ (get-placed$ ?eventID) $?*all*))
    (printout t "Icons to check: " ?icons_to_check crlf)
    (foreach ?x $?icons_to_check
        (bind ?res (run-query* next-of ?eventID ?x))
        (while (?res next)
            (bind ?nvalue (?res get "nr-value"))
            (bind ?svalue (?res get "s-value"))
            (modify (?res get "fact-id") (value (+ ?nvalue ?svalue))))

        (bind ?res (run-query* previous-of ?eventID ?x))
        (while (?res next)
            (bind ?pvalue (?res get "pr-value"))
            (bind ?svalue (?res get "s-value"))
            (modify (?res get "fact-id") (value (+ ?pvalue ?svalue))))

        (bind ?res (run-query* is-in-scenario ?eventID ?x))
        (while (?res next)
            (bind ?scvalue (?res get "sc-value"))
            (bind ?factor (/ (?res get "ss-value") 10))
            (modify (?res get "fact-id") (value (+ (* ?factor ?scvalue)
                  (?res get "s-value"))))
            )
        )
    )
```

Figure 59 Jess code to calculate the icon suggestions for a given event

5.3. Networking and security

Important part of the project is the secured communication between clients and Jess servers. For that purpose I developed a mine own protocol suited for my needs.

For the security of the system I used a simple algorithm for pseudo random number generation based on Galois linear feedback shift register implementation with maximum period of 232 -1 possible combinations. This algorithm can produce random number is the given period range without repeating of any of them.

On that algorithm I developed logic for verifying and encryption of all communication messages. To verify the message I embedded in each message a unique generated sequence number. With the help of that number the receiver can verify the sender. If the received number is not the one expected the message is discarded. This will reduce the risk of receiving unauthorized messages. The sequence numbers are always different in every message. To increase the security I developed and encryption of the messages. The encryption can be done on character or byte level. The best solution is the byte level but if there is a text that is not in 8 bit code then some problems can occur during sending the messages, which is the reason I choose to use the character level encryption. This means that the letters in the message a scrambled.

Chapter 6: Conclusions and Recommendations

In this chapter the results of the project are discussed. After that I will evaluate to what extend the projects goals are reached. Finally I will discuss some possibilities for further research and development of the system.

6.1. Results

This section presents the results of the project. I will present the results made on the interface and the intelligence, and the dynamical aspects of both.

6.1.1. The Interface

The interface of the system was one of the first clear goals of the project. Before I could make a working system I needed some way to let the user feed input to it. Two previous prototypes were developed. They also were for Android platform. I did not add any functionality to these prototypes, but to see how to arrange the icons to be easier for the users to use in emergency situations.

I used a set of icons gadded for previous project on the same project, because the purpose of the project is not the icons itself. But I was curtain with that the icons that I will use have to be similar in type, otherwise is very confusing and annoying when you have to describe something with icons that can't or it's difficult to combine.

While developing the prototypes I started build up the system incrementally. After some basic functionality was added I could build on top of this, delete the changes if I didn't like them and replace them with improvements. Some of the changes in the graphical interface reflected the way of the communication with the server, because as I said before I developed a communication protocol suitable for my needs. After this prototype was designed and implemented, I began with the development of the servers. The servers didn't need a graphical interface.

At the late stages of the development I introduced the security capabilities of the servers and communication server inside the mobile client. The result of this is the final version of the client.

6.1.2. The Intelligence

When I had a first working version of the client I needed to implement a the communication server and Jess server as well. The communication server needed to be able to connect with several clients at the same time, so I designed it to start up a new thread for each client, which could then be handled simultaneously. I designed it internally to create a virtual extension of the iROS network so the Jess servers can send private messages to specific client without all other clients to receive the information. This implementation looks like as encrypted private virtual connection. I could start implementation of the Jess server because I needed the communication server to connect the clients with the intelligence.

When the communication server was complete I started the implementation of the Jess server. I first created some simple implementation as a proof of concept. This first version of the Jess server could receive input from clients and send back some hard coded messages, before I started the intelligence implementation.

In designing the Jess code I first made it possible to add new events and icons to the knowledge base. When that was working we also designed code to delete the events and their icons. After that I developed a way to filter out double events and combining them.

Behaviour that could be added in a matter that relies both on human and machine intelligence is to let the server make suggestions for possible next icons. To just add the icons would be too dangerous, and the location of the to be placed icon is too hard to predict. The compromise I designed is to let the server calculate which icons it expects, and ask the human user to decide if they should really be placed, and where.

The server also makes some assumptions about what scenario is going on. Since this is just an assumption that does not have a big impact on the working of the system and its world model, I decided to keep this intelligence completely at the server side, without human intervention. The adding of these rules resulted in the final version of the server. A server that has a consistent world model at all times, and makes suggestions to the user.

6.2. Conclusions

In this section we will evaluate to what extend the project goals are achieved. We will do so by using the split up we made in the Design chapter:

> Design and implement multimodal system for mobile devices (1), capable of reporting crisis and emergency situations via wireless or high speed mobile network and using a map of the surroundings (2), which is expressive enough to handle complex and unexpected situations (3), yet intuitive enough to use without making (a lot of) errors (4). The system should be language independent, allowing the user to add icons, text and different drawings to the report (5). The system should be intelligent enough to assemble and maintain a correct and up to date world model (6). It should detect possible errors in the form of missing, double and wrongly placed icons (7). Furthermore the system should be dynamic in the sense that new concepts and rules can easily be added (8). It also should provide data availability against power down (in client and server side), data reliability and data security (9).

The conclusions for each requirement will be discussed below:

1) This part of the problem is solved. I made an interface which allows to be used on a mobile device.
2) The system can communicate with the remote server and allow the user manipulate the map of surroundings. The map is dynamically loaded with the help of Google Map service embedded in the used platform.
3) The system is expressive in a sense that there are a lot of concepts that can be reported about. Furthermore the icons that are used to represent the concepts can be given attributes to add more information. The complexity of the system can be further increased by extending the XML files. Unexpected situations would be situations in which new icons are needed that were not implemented yet. This can be done by adjusting the XML files to add the concept and its relation to other concepts. There is a problem with this however, because the server and the clients have to be recompiled in order for the changes to take effect. When this is done, the already reported icons will be lost.
4) It is easy to select the right icon because the icons are distributed over logical icon groups, which can be altered if needed by adjusting the XML files. To provide extra information, an attribute window will pop up where the values of the attributes can be given. The values can be selected out of a small list, this decreases the chance of making a wrong selection, and eliminates the chance to make an illegal selection. When icons are placed, the user is able to delete them again, or to inspect or alter its attributes. To prevent placing

events on the wrong location, the user can easily zoom in and out of the map, to be able to place the icon exactly where it should be

5) During my work on the project I understood that to make the system completely language independent will be really difficult, but managed to limit the use of text at minimum. I developed and implemented algorithms to let the user draw on the map and send the drawing to all other users and server.
6) In order to assemble and maintain a world model we collect all information at one server. The Jess component in the server is responsible for keeping the world model up-to-date. Because the server only has 1 world model that gets adjusted over time, the clients will all be send the same information, which is always the newest. The correctness of this model is dependent on the information the users send. When they report nonsense, the systems world model is worthless. During the user test, both users were sending correct information, and the server fused their world models in a correct new one.
7) To detect missing icons, the system looks at the already placed icons and the possible scenario. From this it gives suggestions to the user, rather than adding icons autonomously. The user can then decide if the suggested icon should be placed or not. Double and wrongly placed icons are detected by the systems double icon filter. When two or more of the same icons are placed very closely to each other, the system combines them, as long as they were reported by different clients. When multiple reports of the same icon are made, the system will take a weighted average of the icons location, and thus incorrectly placed icons will be placed on a better position. If a client reports an icon that is too far from its correct location, and out of the filters range, it cannot be detected.
8) New concepts can easily be added or adjusted by altering the XML files. The relations between the icons can also be adjusted in this way. Adjusting these will result in the system to give other icon suggestions or scenario overviews. There are however still some hard coded functions that are not dynamical, such as the double icon filter.
9) For the security of the system I implemented user verification for every message transmitted through the communication network and also to encrypt the message data. Other thing done to improve the availability against power down is I implemented a sort of clustering and redundancy in the server architecture.

Concluding I can say that all the goals, as stated in the problem description are met. However, there are still many things I would like to see done in a different or more elaborate way. I will discuss these in the next section.

6.3. Recommendations

The development of the CRS system has been a single student effort with a time span of approximately a 6 months. A similar system, like the C2000 project was done by hundreds of people, costing about 700 million Euro, and its development is lasting many years already. From this it should be clear that our constraints on resources have made it impossible to develop and implement every aspect I wanted. In this chapter I will discuss some of the ideas I were unable to work out and implement.

Extending the system by non human observers

At this time we only get input from human observers. To extend the system we could add some non human observers as well. This could be done by sensors, which could for example report about smoke development. We could add smart cameras to the system, which can report about various things like unexpected crowds of people, smoke, or traffic jams. Systems like these could place their own icons on the map and send them. In an ideal case our system could get input from all sorts of security systems. If a fire alarm goes off in a building it could send a report to our system as well, we know the location of the building and that there is probably a fire. Information like this is exactly what we need for CRS. The same goes for burglar alarms, in banks or even houses.

Expanding and improving the intelligence

The current intelligence of the system filters out double icons and gives scenario information and suggestions for icons that could be placed next. Room for improvement lies in the information I are using. When there is a lot of information available about scenarios we could improve the information in the XML files to provide for more realistic calculations of the scenarios and icon suggestions [Cha05]. Besides improving the intelligence by using more reliable and accurate information we can also expand the intelligence. The following aspects could be investigated:

- suggestions for deleting icons
- intelligence over time
- giving the clients roles

Bibliography

[PSc0] Master Thesis of Paul Schooneman, Delft University of Technology

[C2000] The C2000 system, designed in order of the Dutch government, see

[Cha05] MSc Thesis of Jan Chau, still under construction at this time. Delft University of Technology
[CLIPS] Expert System tool, see http://www.ghg.net/clips/CLIPS.html

[JADE] Java Agent DEvelopment Framework, see http://jade.tilab.com/

[JESS] Java Expert System Shell, see http://herzberg.ca.sandia.gov/jess/

[Tat03] Iconic Communication, Iulia Tatomir, December 2003, Bachelor Thesis, Delft University of Technology