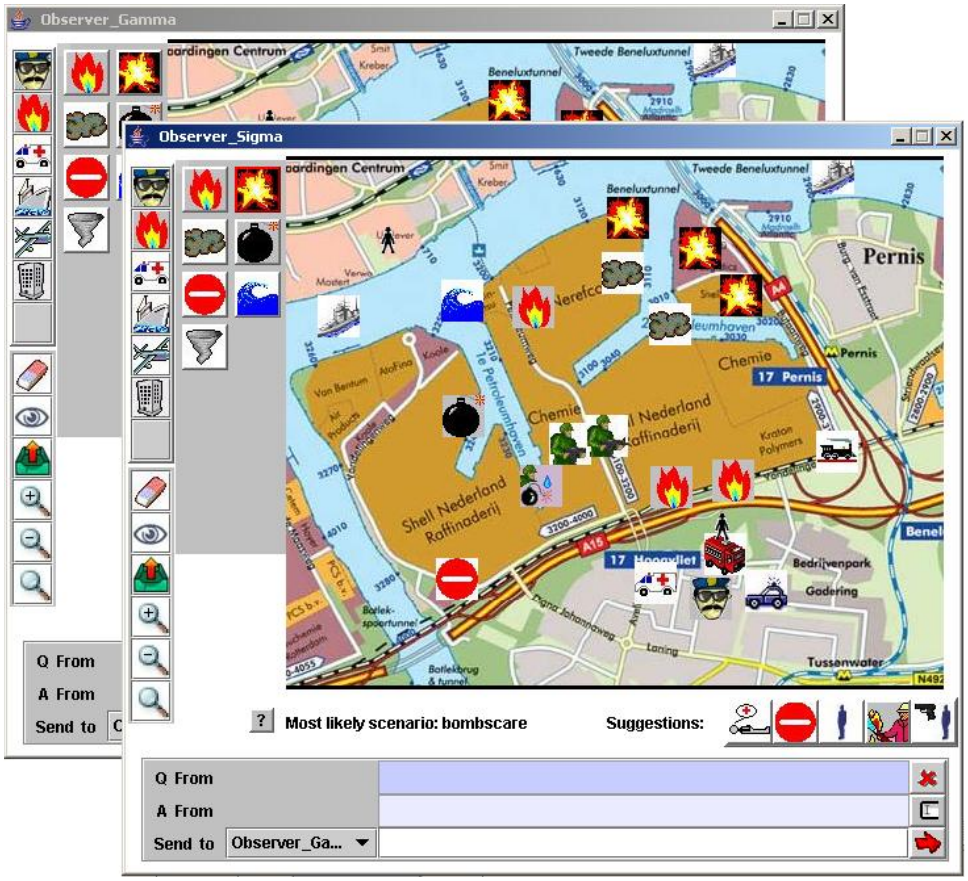
An icon based observation tool for crisis situations modeled with Cougaar



Most likely scenario: bombscare

Suggestions:

Q From

A From

Send to Observer_Ga...

UNIVERSITATEA
Transilvania
BRAŞOV

Project report by
Marius Paltanea,
February 3 to May 3,
2005 TU Delft

TU Delft
Delft University of Technology

Acknowledgments

First of all I would like to thank my supervisor from the TU Delft University of Technology, Dr. Drs. L. J. M. Rothkrantz for his support and confidence in regard to the development of the project. His ideas and suggestions shaped the new improved look and functionality of the original IconMap application.

Then, I would like to thank several people belonging to the MMI staff, who helped me develop the application. They are Bogdan Tatomir, who coordinated and guided my work throughout the whole period and also kindly offered his help every time I needed it , Paul Klapwijk, who helped me get past some technical issues and last but least, Paul Schooneman, whose IconMap application was the basis on which my whole project was founded upon.

A special thanks goes to Hakan Elgin from the DECIS LAB, who introduced me with the Cougaar framework and also gave me a lot of useful tips regarding this state-of-the-art programming environment.

I would also like to thanks Prof. Dr. Livia Sangeorzan from the "Transilvania" University of Brasov for making possible my coming to TU Delft.

Abstract

Crisis and disasters have always been with us. Time and again they have caught us unprepared to deal with the rapid and unexpected changes that they are inducing to the environment. The increasing number of crises and disasters has motivated more and more academics, consultants, and practitioners to engage in crisis-related activities. The project on crisis management researched at the Man Machine Interaction group at TUDelft University is such an activity, focused on developing tools that can help people better coordinate their efforts towards resolving a crisis.

The icon based observation tool for crisis situations has been designed as an alternative means of communication. The graphical interface and the use of intuitive icons should prove very useful when trying to report about a crisis.

The Cougaar framework was used to wrap the application, giving it the shape of a distributed agent-based application. Cougaar is an open-source, Java-based architecture for the construction of highly scalable distributed agent-based applications. The Cougaar architecture distinguishes itself from other architectures by its agent model, based on a distributed blackboard.

This paper presents the design and implementation of the icon based observation tool, insisting on the Cougaar elements used to model it as a distributed agent-based application.

Table of contents

I. Preview

1.1 Overview of the ISME application

1.1.1 Introduction

The ISME (Icon Based System for Managing Emergencies) is the graduation project of Paul Schooneman at the Man Machine Interaction group of the faculty Electrical Engineering, Mathematics and Computer Science at the Delft University of Technology. The research is done as part of the Combined Systems [Comb] group at DECIS LAB, a collaboration of Delft University of Technology, University of Amsterdam, THALES Nederland, and TNO. This collaboration is focused on the research of decision support systems, seeking to order information in complex and chaotic situations.

The goal of this thesis was 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. Generally, 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.

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, which he can then place on the map. Then the Structured World Model gets send to a central server, which collects reports of all the agents in the field. This server will fuse the ideas and form its own structured world model that gets send back to the agents, along with suggestions that the agent might have forgotten to report. The agent will see these suggestions, forcing him to observe the situation again, to see if he missed anything.

1.1.2 Interface

As said in the introduction communication will be realized by means of placing icons on a map. Since the application is developed for crisis situations it should be able 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. Therefore, the icons are 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.

1.1.3 Intelligence

One of the requirements for the system was to be 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, the Artificial Intelligence module of the system is 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 Notes on design

The following set of requirements was taking into account while designing the original ISME application:

- To make the system suited for iconic communication in a crisis situation it should use icons to represent concepts in a crisis and maps of the surroundings to place the icons on.
- To make the system expressive enough it should use 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, there should be a way to add even more information. That's why the icons will have several attributes. For instance, in the case of flames these attributes could be the size of the flames, the intensity and the status (increasing, decreasing, under control).
- 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.
- In order to assemble and maintain a world model all information should be collected at one server. In order to create a world model out of this, the information could be stored in some sort of database, which will have to be kept consistent at all times. A client-

server implementation was chosen 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.

- To detect missing, double and wrongly placed icons, there was the need of some intelligent agent, which should constantly work on the information that has been gathered. This means it would have to work on the database of collected information.
- 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 this database is kept 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.
- The real system, except for the server, has to run in the field during a crisis situation. This means that the type of hardware needed 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.

1.2 Adapting the ISME Application for the Cougaar framework

1.2.1 Problem Description

The problem which was solved to a certain degree in the ISME thesis work which was focussed on the interface and intelligence of the system was defined as follows:

Design and implement a system that is suited for iconic communication in a crisis situation, 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 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.

The work on this current particular project left from the assumption that the ISME system has been implemented according to the above statement and was meant to solve the following problem:

Adapt the ISME application so that it can be easily used by agents in a Cougaar framework. The system should be aware at all times of the users' position and constantly update them with information about the ever changing world model. In addition to that, users should be able to communicate directly with each other using a simple question-answer interface.

As stated above, the goal of this assignment was to transfer the ISME application to Cougaar in order to simulate the real-time distribution of updates from the central server to the users of the application. In the case of the original ISME application, a particular user would send the "field" information from his client application to the application's server. The server

would then process this information and send back the updates to the user. In the Cougaar approach, the sending of the "field" information works on the same principle (from client to server). However, the response of the server is not directed solely towards the sending-user, but also to all the other users of the client application. This way, all the users will be constantly updated about the status of the crisis' development.
In order to achieve this, the main task was to replace the socket-based communication between the server and the client applications, used in the original ISME application, with the complex mechanisms of message transmitting implemented by the Cougaar technology.

In addition to that, the Cougaar version had to be designed always bearing in mind the possible scenarios in which the users of the application would be involved. In regards to this, for the case of a real crisis situation, knowing the position of a particular user who placed some icons on the map would prove very useful information. The next logical step would be to allow users to communicate directly with each other. For instance, a field observer who has just placed some icons could be contacted by other users, in order to give them some additional information about the situation he is witnessing.

The idea of assigning roles to users comes soon after. In order to prevent chaos from taking over the "crisis management system" itself (a situation which could easily happen if all users would have the right to modify the map as they please) they could be assigned roles. For instance, field reporters should only be allowed to add, remove or modify icons in the vicinity of their location. Field coordinators should be people with some experience of managing a crisis situation and therefore their sent information could be given more credit. They should also be given the opportunity to get some feedback from other field reporters, who had just sent their updates. A special category of users should be the central coordinators, who would not necessarily have to be on the field, but who would have the important role of monitoring the whole development of the situation. They should be the ones taking the decisions after having acquired enough information to get a clear picture about what is happening, either by inspecting the map or by questioning the field reporters about what they are witnessing.

As some of these new ideas were implemented, the Cougaar version of the Icon Map application had to suffer some modifications both in the user interface and in the intelligence of the system.

II. Overview of the Cougaar framework

2.1 Introduction to Cougaar

Cougaar (for Cognitive Agent Architecture) is an innovative software architecture that enables building distributed agent-based applications in a manner that is powerful, expressive, scalable and maintainable. In fact, Cougaar is a code baseline that has successfully demonstrated its utility at constructing dynamic, complex, distributed applications. Perhaps of even more significance than the software that implements its concepts, Cougaar represents a methodology, a tried and powerful approach towards designing and building distributed applications.

Cougaar was developed for DARPA (Defense Advanced Research Projects Agency) under the Advanced Logistics Program or ALP. The focus of the developers has been to make the Cougaar platform survivable, that is, to enhance the Cougaar platform with components offering:

- Robustness: A Cougaar application should survive the loss of any individual components and/or hardware substrate with minimal loss of functionality. This includes automatic recovery of lost agents, as well as various mechanisms to conserve resources and to use redundancies efficiently.

- Security: A Cougaar application should be capable of repelling various sorts of electronic attacks, should maintain information integrity, and should avoid exposing communications as much as possible.

- Scalability: The Cougaar infrastructure should not have any intrinsic scalability issues. It should be possible to implement Cougaar applications which scale to the degree that the application logic allows.

What kinds of problems are well suited to a Cougaar solution? While Cougaar was developed to handle a problem with all the complexities listed above, it is of potential value in any domain bearing any of the above complexities. For example, any of the following problem categories would benefit from being modeled in Cougaar:

- Problem domains that entail hierarchical decomposition and tracking of complex tasks
- Complex application domains involving integration of distributed separate applications and data sources
- Domains involving the generation and maintenance of dynamic plans in the face of execution
- Highly parallel applications with relatively loose-coupling and low-bandwidth communications between parallel streams
- Domains too complex to model monolithically, best modeled by emergent behavior of components

We should note that while Cougaar was designed to address the requirements of military logistics planning, we have seen applications of Cougaar to many different domains—some only tangentially related to military logistics, others completely separate. Nonetheless, this document will contain a flavor of military logistics in many of its examples and illustrations, as this is the domain to which Cougaar has been applied most broadly and successfully. It is important to keep in mind that the Cougaar technology is a domain independent architecture for large scale distributed agent systems.

2.2 Architecture "Quick Look"

In a nutshell, Cougaar is a large-scale workflow engine built on a component-based, distributed agent architecture. The agents communicate with one another by a built-in asynchronous message-passing protocol. Cougaar agents cooperate with one another to solve a particular problem, storing the shared solution in a distributed fashion across the agents. Cougaar agents are composed of related functional modules, which are expected to dynamically and continuously rework the solution as the problem parameters, constraints, or execution environment change.

2.2.1 Top Level: Cougaar Society

A Cougaar Agent is an autonomous software entity that has been given behaviors to model a particular organization, business process or algorithm. Multiple agents often collaborate as peers in a Peer-to-Peer (P2P) distributed network. The complexity of each agent can range from simple embedded sensors to a highly complex artificial intelligence application. Cougaar is a framework for developing distributed multi-agent applications. The Cougaar architecture includes components to support agent-to-agent messaging, naming, mobility, blackboards, external UIs, and additional (pluggable) capabilities. Developer write components, also called "plugins", which are loaded into agents to define their behavior.

A Cougaar Society is a collection of Agents that interact to collectively solve a particular problem or class of problems. The problems are typically associated with planning, where the plan objective and constraints may be continually changing and re-planned in the face of execution. A Cougaar Community is a notional concept, referring to a group of Agents with some common functional purpose or organizational commonality. Thus a Cougaar society can be made of one or more logical communities, with some Agents associated with more than one community and other Agents not associated with any community. The society shares a DNS-like Namespace that allows all agents to resolve references to one another, and which may be monolithic or distributed/redundant. All Agents in a given society run the same Cougaar core software baseline, written in Java, though different Agents may contain additional "jars" to give them particular behaviors, model particular entities, or embody particular capabilities.

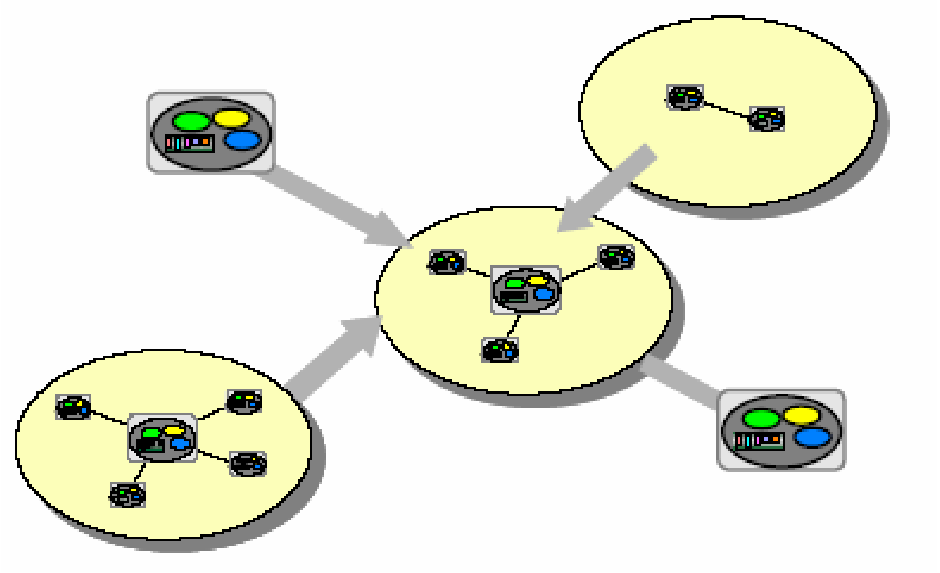


Figure 1 - Cougaar Society

2.2.2 Zoom In: Cougaar Community

When we zoom in on a Cougaar community, things look similar to the view of a society: it is composed of other communities, perhaps sub-communities, and Agents. A community is not a software architecture concept, but a notional design concept that helps in designing and constructing the society out of a logical grouping of pieces. A community tends to speak a "common dialect," meaning the Agents in a community may communicate about objects and issues that only this community knows about and using terms and activities only this community understands. There is typically a lot of traffic among Agents of a community, and fairly limited and constrained traffic between communities. A community typically has some notional interface of what it provides to the society: it performs the following services, it takes these inputs and produces these outputs, etc. Thus the concept of a community, during both design and implementation, provides an effective logical circumscription of a specific domain model and the suite of functionality that operates over that domain model.
Communities need not be distinct: communities are often hierarchical (e.g. as in departments in an org chart), or overlapping as a given Agent may be a member of any number of communities, each usually denoting a different semantic grouping.

2.2.3 Zoom in: Cougaar Node

When we zoom in further, we reach a Cougaar node. A Node is a single Java Virtual Machine (JVM) instance that may contain and maintain multiple Agents. In most cases there is a 1:1 correspondence between the node and the hardware platform, but this is configuration efficiency rather than a requirement. The grouping of Agents into a node is not necessarily domain related, but rather based on equitable sustainable sharing of computer resource requirements among all Agents. In some cases, Agents from different communities will share a single machine and node to be efficiently collocated near a shared data source, like a database. It is often convenient to think of a node as a special class of unnamed Cougaar communities where the logical grouping is by physical machine locality.

As in other Cougaar communities, the membership can change over time as Agents are created, moved to other machines, and decommissioned.

All Agents on the same node share the same CPU, the same pool of memory and disk, and compete for incoming and outgoing bandwidth traffic. The node serves as a router of messages for the Agents it is hosting: messages to other Agents in the same node are passed directly within the same JVM, efficiently short-circuiting the message transport layer; messages to Agents in different nodes pass through a MessageTransport layer that passes the message through the network to the receiving node which then routes it to the appropriate Agent within that node.

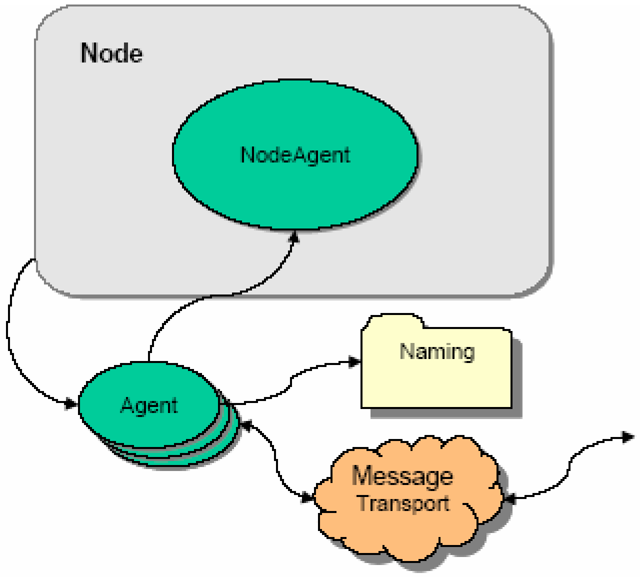


Figure 2 - Node Structure

2.2.4 Zoom in: Cougaar Agent Internals

As we zoom in yet further, we view the Agent and its internal components. An Agent consists of two major components: a partitioned distributed Blackboard, and Plugins. The Plugins are software components that provide behaviors and business logic to the Agent's operations, and operate by publishing content and subscribing to objects on the Blackboard.

A Blackboard is an agent-local memory store that supports publish/subscribe semantics. Components within the agent can add/change/remove objects from the blackboard and subscribe to local add/change/remove notification. Agent domains monitor the local blackboard and can send messages to other agents and alter the blackboard when the local agent receives messages.

The primary benefit of an agent blackboard is that it abstracts the message transport from the plugins. The blackboard defines an asynchronous publish/subscribe API with pluggable domain-specific behavior. This frees developers to concentrate on the domain-specific issues of their application.

Cougaar blackboards also support transactions, persistence, rehydration, and dynamic reconcilliation. Additional details can be found in the Cougaar Architecture Guide.
All access to the Blackboard is transaction-controlled. Blackboard transactions cover only membership of objects in the logical collection—Transactional safety is not guaranteed for sub-object changes, only for addition and removal of objects from the Blackboard. In addition, the mechanism that delivers packages of Add and Remove events also delivers Change events (with details) which may be used to track sub-object-level changes in-band with Add and Remove events. Cougaar blackboards are agent-local to assure scalability. A globally shared blackboard (e.g. JavaSpaces or JMS) is a single point of failure and a considerable performance bottleneck.
The Blackboard contents are segmented into sets of logically-related objects by Domains. A Domain is, in effect, a specification of the language used by plugins to communicate with each other and with related plugins in other Agents. Each Domain has a set of LogicProviders, which are plugin-like components that act as translators into other Domains' business logic and/or Agent messaging.

The Agent is responsible for management of queues containing messages to and from other Agents, scheduling the execution of Plugins and managing the subscription mechanisms of Agents.

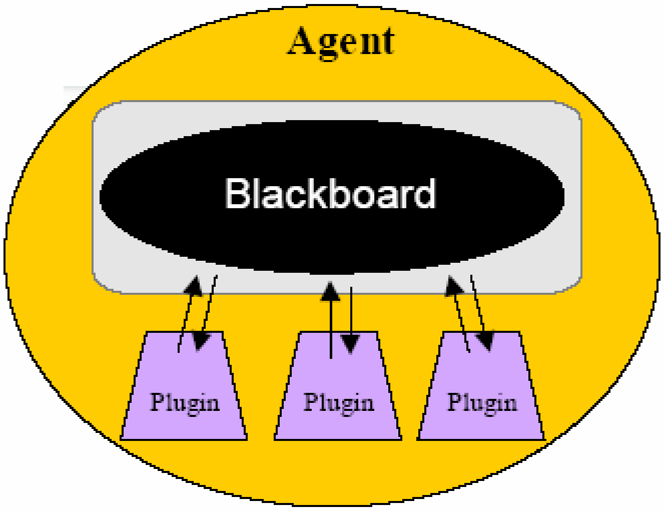


Figure 3 - Agent Structure

2.2.5 Zoom in: Cougaar Blackboard Contents

As we zoom in to our finest level of detail, we see the detailed contents of the blackboard. Plugins publish and subscribe objects to the blackboard. In principle, these could be any objects, as defined by the application's business logic. For example, the Cougaar planning Domain implements the following sorts of objects:

- Tasks represent a requirement or request from one agent to another to perform or plan a particular operation.
- Assets represent resources to which tasks are allocated. A distinguished type of asset is an OrganizationalAsset or an EntityAsset, which represents a proxy to another Agent.
- PlanElements contain dispositions of tasks. Tasks may be allocated (through an Allocation PlanElement) to an asset resource, or expanded (through an Expansion PlanElement) into subtasks, or aggregated with other tasks (through an Aggregation PlanElement).

The blackboard of a Cougaar Agent is part of the distributed blackboard managed in a distributed fashion by the whole Cougaar society. Each Agent owns its blackboard and its contents are visible only to that Agent. All sharing of blackboard state is done by explicit push-and-pull of data through inter-Agent tasking and querying. In this way, Cougaar is able to maintain fine-grained state in individual Agents while sharing only high-level synopsis information around the society, making the management of information scalable and efficient.

2.3 Cougaar Communications

2.3.1 Agent Naming Services

Cougaar includes support for distributed agent naming services. These naming services are used by the Cougaar message transport to route message over multiple network protocols to mobile agents. Application developers can also use the naming services to dynamically discover agents at runtime.
In this section we discuss the five different types of distributed naming capabilities have been identified:

- "Name Generation" constructs a globally unique agent name.
- The "White Pages" is a table that maps names to network addresses (e.g. DNS).
- The "Yellow Pages" is an attribute-based directory (e.g. a categorized phone book).
- "Local Discovery" uses LAN-based IP multicast to locate nearby agents.
- "Peer-to-Peer Search" allows an agent to search adjacent agents for resources.

2.3.2 Name Generation

All agents in a Cougaar society are required to have a unique name. An agent's name is used to route messages to the agent and may hold special meaning in the developer's application. Most Cougaar developers select their own agent names to match the role of the agents. For example, an agent that models the Boston Department of Motor Vehicles may be named "dmv.boston.ma" by its developer. Note that the Cougaar infrastructure doesn't assume that this name has any particular meaning, just like an Operating System doesn't care what you name your files so long as they are unique.

A runtime name generator can be used to randomly select a name when the agent is created. This is typically accomplished by hashing the local host's IP address with a large random number. A random name may make sense for an anonymous embedded sensor.

2.3.3 White Pages

The white pages is a distributed table which maps agent names to network addresses. The primary function of the white pages is to support the Cougaar message transport and other network-aware components.

For example, a white pages lookup of "AgentX" may return a set of network entries such as the agent's RMI message address (rmi://test.com:1234/xyz) and the agent's servlet port (http://test.com:8800). This is similar to DNS (Domain Name Service) name-to-address resolution.

The predecessor of the white pages is the Cougaar Naming Service, which has existed in different forms over the lifetime of the Cougaar project. The current white pages implementation is the third full redesign and included several high-level goals:

- Must be scalable, to support thousands of agents, running on hundreds of hosts on a wide area network.
- Must be robust, with no single point of failure, multiple servers to survive overloads, and persistence to support restarts
- Must be efficient, utilizing an integrated caching and garbage collection scheme
- Must be cleanly integrated into Cougaar, leveraging the Cougaar message transport for message protocols, quality of service, and security

Reflection on how the naming service was used prompted us to split the naming service into two separate services. A phone book analogy was adopted, where the two services are: A "white pages," which maps names to network addresses; and a "yellow pages," which supports more complex attribute-based searches.

The white pages service has been modeled after DNS. Agent names now support Internet host name semantics with the '.' separator character. A hierarchical name space will support better cache control and help distribute the data to multiple naming server agents. The white pages is now an effectively agent-based application that runs within Cougaar, whose job is to support the message transport and other Cougaar network-aware components.

2.3.4 Yellow Pages

The Yellow Pages is a directory service that supports attribute-based queries. This service allows agents to register themselves based upon their application's capabilities, and allows agents to discover other agents based upon queries for these capabilities.
For example, an agent that models an inventory warehouse might register in the yellow pages with an attribute-value pair of 'role=inventory'. At runtime another agent could query the yellow pages to list all agents where 'role=inventory'. This query may be limited by geographic or other application-specific constraints, such as "I prefer geographically close entries."
In prior versions of Cougaar this could be implemented by using the JDNI-based naming service, which supported both white-pages-style and yellow-pages-style queries. The Yellow Pages also supports entries with custom data structures with detailed information, as opposed to the white pages' limited network-address entries. Lastly, the Yellow Pages often has multiple application-specific concepts of locality that don't necessarily match the white pages' network-based layout, such as geographic locality as opposed to LAN/WAN locality.

2.3.5 Local Discovery

Local discovery allows an agent to discover other LAN-local agents without knowing their names or IP addresses. This typically uses LAN-local IP multicast with UDP.
For example, a new node with a couple agents may be started on a rebooted host. The agents can send a generic "Is anybody out there?" multicast to find the other agents on the LAN. Each agent on the LAN then replies to the
multicast with its name and network-address information.

2.3.6 Peer-to-Peer Search

Peer-to-peer search allows an agent to discover resources that reside on adjacent (peer) agents. This typically uses a hop-based search mechanism, analogous to JXTA search or the Gnutella protocol.
For example, an agent could send out a peer-based request for a specific file named 'test.mp3'. This request must be scoped by either the "time-to-live," a peer-based hop count, or some other limit. The result of the search is a list of etching agents and perhaps additional information pertaining to this search.

2.3.7 MessageTransport

When a node starts an Agent, it makes available to the Agent a MessageTransportServer instance that provides MessageTransport and NameServer functionality. The MessageTransportServer instance is usually constructed on behalf of the node by the static methods of the Communications class. Communications uses System Properties to determine the class of MessageTransport to construct, create an instance, and then start it.

The choice of MessageTransport usually implies a single specific related NameServer class and instance which is, in turn, constructed and started.
The MessageTransport class provides an API for sending messages to arbitrary Agents by name (MessageAddress, usually a ClusterId) and for registering a MessageTransportClient (usually an Agent) with the Transport so that it can receive messages from other sources. MessageTransport implementations are usually fairly complex in order to achieve good throughput to multiple peers which may vary considerably in distance/latency, bandwidth, and connectedness (e.g., periodically connected). The default RMIMessageTransport uses multiple queues served by a pool of threads to guarantee proper, in-order delivery of messages. Messages are always one-way and asynchronous—any response will be in the form of another message.

2.4 Cougaar Servlets

Cougaar includes support for handling HTTP and HTTPS requests by invoking user-developed server-side request handlers, called “servlets”. Developers can use servlets to generate HTML views for browsers, send binary data back to a remote client, interact with local or remote Swing-based UI clients through HTTP, and other applications.

Servlets are similar to plugins: each agent has a separate set of servlets that can use a ServiceBroker to access the Cougaar services within that agent. Servlets are bound within an agent to a unique URL path, such as “/test”. Agents themselves are registered with a globally-unique URL-based “/$name”, such as “/$TRANSCOM”, that matches the naming-services registration. Together these create a globally unique URL-path to that agent's servlet: “/$TRANSCOM/test”.

All nodes in the society create web-servers with a unique “scheme://host:port” address, such as “http://foo.com:8800”. A remote HTTP-based client can send the “/$TRANSCOM/test” path to any server in the society, such as “http://foo.com:8800/$TRANSCOM/test”, and the request will be redirected to the node that's running agent “TRANSCOM”.

Components have access to a “ServletService” through the ServiceBroker, which allows the component to register and un-register servlets. A component can have its servlet as an inner class or as a separate class. Additionally there are some helper classes included in Cougaar to simplify the design, such as a component that loads simple servlets.

III. The Data Model

To model our Cougaar observation tool we used 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:

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

The diagrams presented in this chapter mainly describe how the structure and functionality of the original application was adapted in order to be plugged in the Cougaar agents. The communication part, which is the main issue regarding Cougaar, will be properly analyzed in the next chapter.

3.1 The Use-Case Diagram

The following diagram represents the use case for the observers. They can add icons, delete icons, inspect or modify icons, enter and modify their position on the map, add other observers as contacts and talk to them, view the world model, view the suggestions, view the scenario information, zoom, send and receive. Some tasks, such as adding icons require some additional actions. To add an icon, the icon that needs to be placed has to be selected, a location has to be given and the attributes have to be specified. To delete an icon the icon has to be specified first. To modify or inspect an icon, the icon has to be specified, then the current attributes will be given, after which the new attributes can be specified. To enter the position on the map a location has to be given, while in order to change the position also requires to inspect the observer icon. Finally, to add a new contact, an observer icon needs to be specified and inspected.

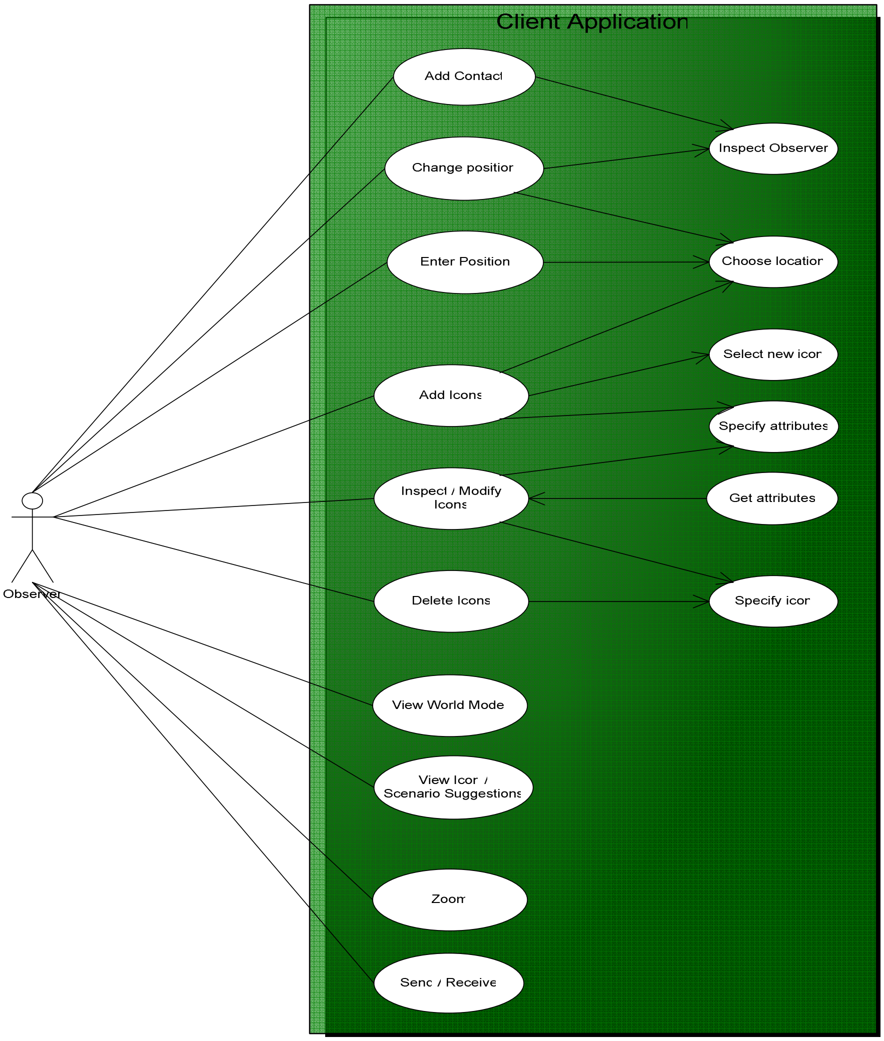


Figure 4 – Use-case diagram

3.2 The Class Diagram

Class Diagrams are used to describe the structure of the system. The following diagram highlights the different layers and components of the Cougaar application.

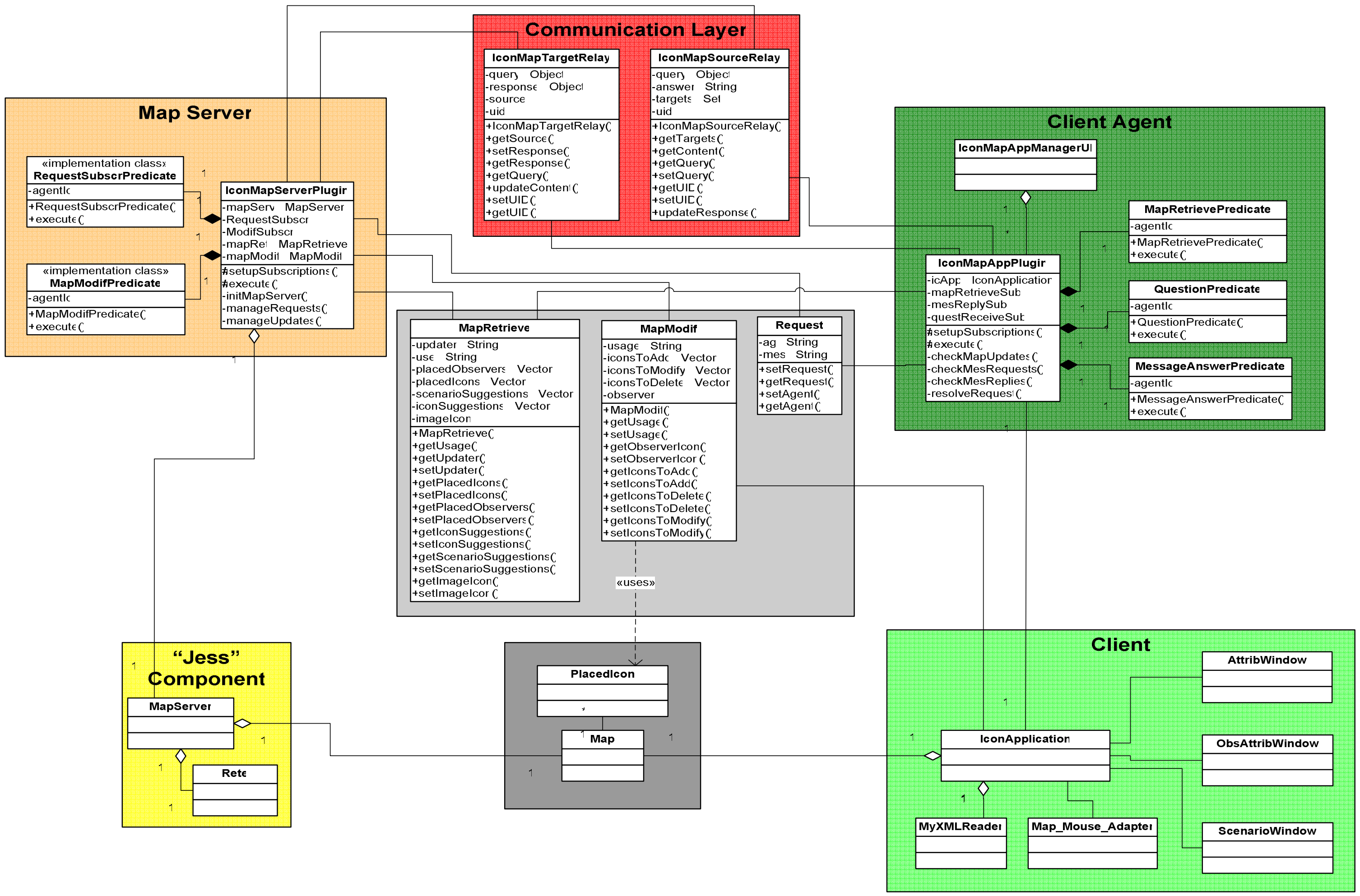


Figure 5 – Class diagram

The directed relations in this figure can be read as: x has a y. For example IconApplication has a MyXMLReader. The undirected relations are actually directed in both ways. IconApplication has a ScenarioWindow and ScenarioWindow has an IconApplication. In this case the obvious relation is that the IconApplication, which is the main part of the client, has a ScenarioWindow. The other way around is true as well, because a ScenarioWindow has to know who its parent is to give back the focus to the parent when the window is closed.

Note that the IconApplication and the MapServer do not have a direct relationship. The IconApplication does not have a MapServer and vice versa. They communicate through the network, using their associated plugins, via the
Cougaar message transport service.

The IconApplication class, which makes up the main part of the client, has a ScenarioWindow is which the scenario information will be displayed, an AttributeWindow which is used to specify or show the attributes of a selected icon. It also has a PlacedIcon which is a class that defines everything we need to know about an icon that is placed (location, attributes, etc.). Furthermore the IconApplication has a MyXMLReader to help it read in the information from the XML files. It also has a Map_mouseAdapter, which is a class that handles the clicks that were made on the map. And finally it has a Map. Map is a class that stores the background image, the icons that were placed, etc. and it has some functions for zooming.

The MapServer class also has a Map. This is not the same instance as the Map on the IconApplication, but via the network these two instances will be kept synchronized. The ServerThread is the class that handles the connections with the iconApplications. Each serverThread instance will support the connection with one iconApplication, but the mapServer can have many different serverThreads (in fact, as many as the computational power of the server can handle). Finally there is the Rete class. The Rete class is the main class for the Jess component, all the reasoning is done via this class.

In order to integrate the above classes in Cougaar, plugins were required for loading the information into the agents. The IconMapServerPlugin uses an instance of the MapServer class and directs the actions of the server, according to the specifications from the messages received from the clients.

On the other side, there is an IconMapAppPlugin, which uses an instance of the IconApplication class, deals with all the messages received from the central server and also with messages coming from other clients who use the Icon Application. Alternatively, the IconMapAppPlugin could be divided into two plugins, in order to take care separately of the two kind of messages mentioned before. The IconMapAppPlugin can be loaded and unloaded at runtime by the IconMapAppManagerUI, a class which extends a Cougaar servlet class.

Both the IconMapAppPlugin and IconMapServerPlugin use implementations of the Cougaar UnaryPredicate class, which basically act as a filter, selecting from an agent's blackboard only those entities that pass several imposed constraints.

Another layer of the application is the communication layer. This is composed of two classes implementing the Cougaar Relay Source and Relay Target classes, which are used for transporting messages. The messages themselves are contained in objects having the type of one of the following classes (depending of the context): Request, MapModif and MapRetrieve. All these three classes have to implement the Serializable interface, since this is a requirement for any object that is sent via the Cougaar message transport system. An instance of the Request class is created every time a client IconApplication is loaded and it needs the map from the server. MapModif objects are used to transmit information about the changes in the map from the IconApplication to the MapServer. In their turn, the MapServer's replies to the IconApplication come in the form of MapRetrieve objects.

More details about all these classes and their functionality can be found in the implementation chapter.

3.3 The Sequence Diagram

In order to clarify the functioning of the different objects together, the next section describes two different functions of the system, using sequence diagrams to visualize the functions and messages which are exchanged between objects.

Usually, sequence diagrams are used to represent the flow of events in a system. The objects in the system interact with each other by sending messages. When a message is received, this results in some action at the receiving end. Actions that will be executed are operations that may result in the sending of new messages to other objects. Arguments may be passed along with the message to give more detail of what actions need to be undertaken.

The next diagram shows the message flow between the IconApplication and the MapServer.

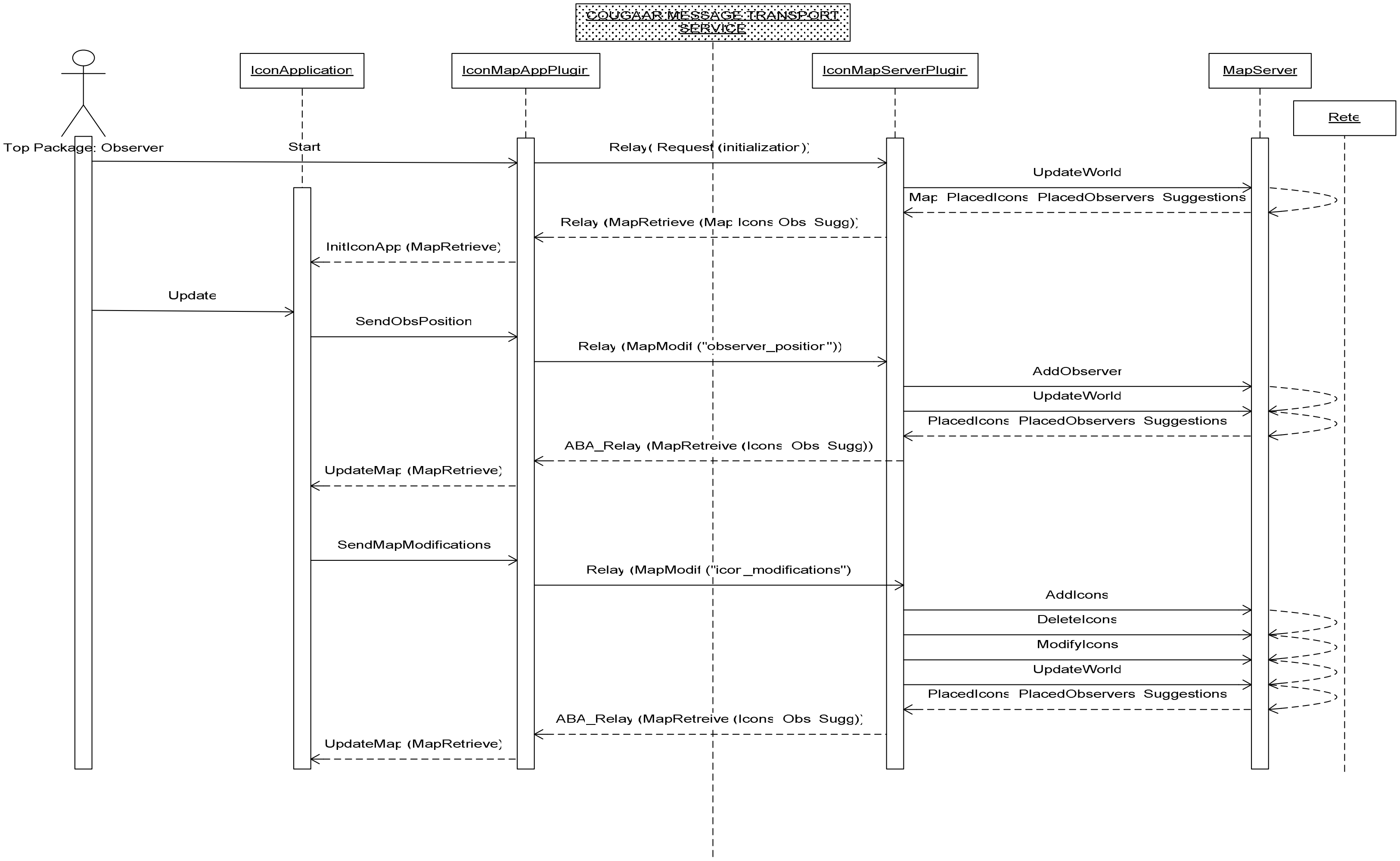

COUGAAR MESSAGE TRANSPORT SERVICE
IconApplication
IconMapAppPlugin
IconMapServerPlugin
MapServer
Rete
Top Package: Observer
Start
Relay(Request (initialization))
UpdateWorld
Map PlacedIcons PlacedObservers Suggestions
Relay (MapRetrieve (Map Icons Obs Sugg))
InitIconApp (MapRetrieve)
Update
SendObsPosition
Relay (MapModif ("observer_position"))
AddObserver
UpdateWorld
PlacedIcons PlacedObservers Suggestions
ABA_Relay (MapRetreive (Icons Obs Sugg))
UpdateMap (MapRetrieve)
SendMapModifications
Relay (MapModif ("icon_modifications"))
AddIcons
DeleteIcons
ModifyIcons
UpdateWorld
PlacedIcons PlacedObservers Suggestions
ABA_Relay (MapRetreive (Icons Obs Sugg))
UpdateMap (MapRetrieve)

Figure 6 – Sequence diagram of sending and receiving from the server

The first thing to be mentioned is that all messages are automatically transported by Relays (which will be discussed later) through the Cougaar Message Transport Service.

The client application can only be started through the use of the IconMapAppManagerPlugin (which does not appear on the diagram since it's not really involved in message transmitting), a servlet which attaches to the default Cougaar servlet server. When the user selects to start the application, the IconMapAppPlugin instantly sends a message to the IconMapServer in the form of a Request object asking for initialization. The MapServerPlugin queries its agent's blackboard for received messages containing Request objects. If it finds entities fitting this description, it asks for the map, placed observers, placed icons, scenario and icon suggestions that are stored in the MapServer at that particular moment. This information is gathered in a MapRetrieve object which is placed on a new message to be sent back to the requestor. When this last message has been received by the IconMapAppPlugin, the information it contains is being and based on that the IconApplication is initialized. From this point on, all messages will leave directly from the IconApplication.

Another step which must inevitably occur in the flow of events is the sending of the user's initial position. This is sent as a MapModif object. The procedure is similar to the one described above for the initialization request with two slight differences: as the IconMapServerPlugin receives the position of the new user, it transmits it to the MapServer, which introduces it in the Jess knowledge base; secondly, after the MapServer has made the corresponding updates, it creates a MapRetrieve object which will be sent to all the other agents (which have the IconMapAppPlugin) instead of only the sender of the message as in the case of the request event.

After the two previous steps have been successfully accomplished (the client application has the map and the position of the owner has been sent) the client can start doing his updates using the interface of the IconApplication. The map modifications (adding icons, modifying icons, deleting icons, modifying the owner's position on the map) are again stored in a MapModif object which is forwarded from the IconMapAppPlugin to the IconMapServer. The updates contained in the MapModif object are sent to the MapServer, which reasons about them using the Rete system and creates the new world model that will be broadcasted to all the reporters.

The next diagram shows the flow of events which is generated by the communication between two client applications. As said in the beginning of the chapter, the communication is not its main concern. However, in order to explain the following diagram, a sneak preview on Cougaar Relays is required.
Relays consist of two interfaces – Relay.Source and Relay.Target - that blackboard objects can implement so that data from a source blackboard can appear on target blackboards and responses from the objects on the target blackboards can appear within the objects on the source blackboard.

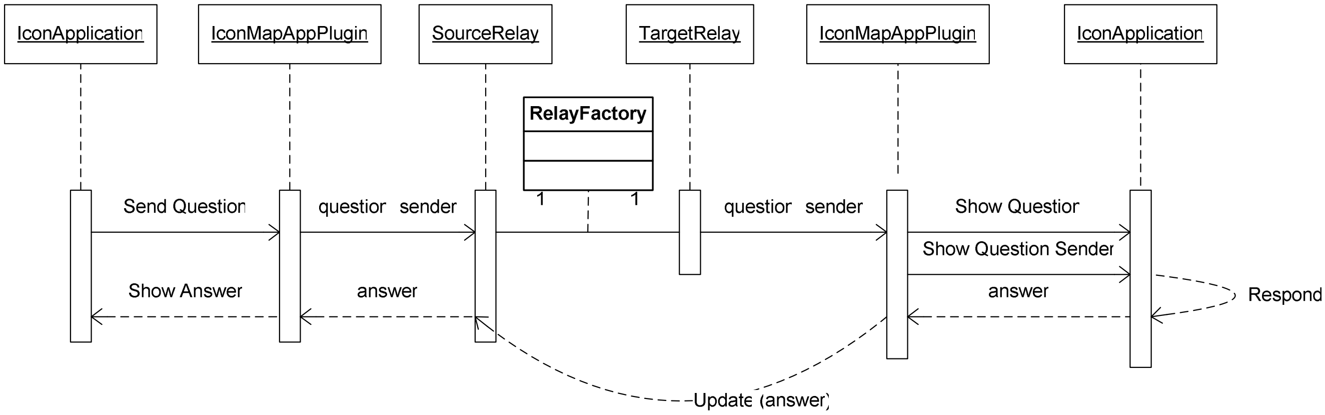


Fig7 – Sequence diagram of sending and receiving between clients

A first client (agent) sends a question to a second client (agent) and then expects an answer. The first client writes the question using the interface provided by the IconApplication. The IconMapAppPlugin wraps this question in the form of a message – a Source Relay object – which is sent to the second agent. The corresponding Target Relay object should be available at the other end – on the second agent's blackboard. When this happens, the second agent's IconMapAppPlugin extracts the question from the Target Relay and sends it to its IconApplication, where it will be displayed. The question will remain visible till the second client puts down the answer. The answer is then undertaken by the IconMapAppPlugin, which updates the Relay object by placing a response on the first agent's Source Relay.

IV. Cougaar Model

4.1 Configuration of the Cougaar society

The most common way of representing the structure of a Cougaar society is by means of XML files. The contents of the XML files is read and interpreted by the Cougaar XML parser at startup.

4.1.1 Defining a community

Usually, Cougaar agents are grouped in communities. In addition to providing a way to define simple groups of agents, communities and their associated entities may also be associated with attributes that can be used to select a specific community member or subgroup.
A Cougaar community consists of one or more Entity objects. An Entity (community member) may be an agent or another community. Each Entity is constructed from two basic components, an identifier and optional attributes consisting of JNDI-based name-value-pairs that can be used to define characteristics such as "Role". These attributes provide the foundation for a flexible query and abstract addressing mechanism that is tightly integrated with the Cougaar blackboard and inter-agent messaging infrastructure.

In this particular application, all agents are members of the community called Observers (so in this case the society is equivalent to the community, at least at an intuitive level), whose main purpose is to facilitate the broadcasting of information from the server agent to the observer agents. This community is automatically created during the startup of the society, using a community definition file and the CommunityPlugin. The CommunityPlugin within an agent looks for the standard community definition file called communities.xml on the configuration path during startup and attempts to join all communities containing the respective agent as a member.

The communities.xml file has to follow a predefined structure, which can be described by the following Document Type Declaration (DTD) file:

```
<!DOCTYPE Communities [
<!ELEMENT Communities (Community+)>
<!ELEMENT Community (Attribute+, Entity*)>
<!ATTLIST Community Name CDATA #REQUIRED>
<!ELEMENT AttributeID EMPTY>
<!ATTLIST AttributeID ID CDATA #REQUIRED>
<!ATTLIST AttributeID Access (manager|member|associate|world)
#IMPLIED>
<!ELEMENT Entity (Attribute*)>
<!ATTLIST Entity Name CDATA #REQUIRED>
<!ELEMENT Attribute EMPTY>
<!ATTLIST Attribute ID CDATA #REQUIRED>
<!ATTLIST Attribute Value CDATA #REQUIRED>
]>
```

Fig 8 – Tthe structure of a community configuration file

The Observers community uses the "CommunityManager" attribute (a community level attribute) in order to explicitly make the MapServer agent the manager of the community. This means that the community can only be created / managed by the MapServer, since without the server application running, the whole system would have no functionality. If another agent (an Observer agent) starts before the MapServer, its join request will not be processed until the MapServer agent has started and created the Observers community.
The Observers community is an open community, meaning that virtually every agent can join it or leave it without restriction. After being created, the community is constantly monitored by the MapServer agent. Since when a node is terminated, the community is not automatically updated about the (possible) left agents, the MapServer periodically checks the state of the community.

Another important attribute (an entity level attribute this time) is the "Role" of the agent in the community. This is an essential piece of information when messages need to be broadcasted to multiple agents using the Attribute Based Address mechanism. In this case, an attribute of this type acts as a filter applied to the members of the community and helps selecting only the desired agents supposed to receive the message.

This is what the Observers community would look like, should it contain the MapServer agent and two more Observer agents:

```
<?xml version="1.0" encoding="UTF-8"?>
<Communities>
 <Community Name='Observers' >
        <Attribute ID="CommunityType" Value="Domain" />
        <Attribute ID="CommunityManager" Value="MapServer" />
        <Entity Name="Observer_Alpha">
                <Attribute ID="EntityType" Value="Agent" />
                <Attribute ID="Role" Value="Member" />
        </Entity>
        <Entity Name="Observer_Omega">
                <Attribute ID="EntityType" Value="Agent" />
                <Attribute ID="Role" Value="Member" />
        </Entity>
        <Entity Name="MapServer">
                <Attribute ID="EntityType" Value="Agent" />
                <Attribute ID="Role" Value="Member" />
        </Entity>
</Community>
</Communities>
```

Fig 9 – The “Observers” community file

Another important Cougaar configuration file is the alpreg.ini file, which defines a host and a port for the naming service. The default host is set as “localhost” but this should be used only when all the Cougaar nodes are simulated on the same machine. Since the application can be run with multiple nodes spread across multiple hosts, the alpreg.ini files on each physical machine should specify the same host (different than localhost).

The following excerpt shows the aspect of the alpreg.ini file:

```
[ Registry ]
address=ahost
alias=AlpFDS
port=8000
```

Fig 10 – The alpreg.ini file

4.1.2 Virtual Machine parameters

Running a Cougaar society also requires the setting of several Java virtual machine parameters. These can either be specified in the Cougaar script that starts the nodes or in the Cougaar society configuration file.
In this case , nodes specify Java command line parameters, which are extracted by the bin/Cougaar script. This duplicates the “-D” system properties in the bin/Cougaar scripts, but allows easier control over per-node “- D”s, and consolidates configuration management in the XML file. For example:

```
. . .
<node name="anode">
<vm_parameter> -Dorg.cougaar.society.file=xml_filename </vm_parameter>
<vm_parameter> -Dorg.cougaar.node.name=anode
</vm_parameter>
<vm_parameter> -Dorg.cougaar.install.path=$COUGAAR_INSTALL_PATH
</vm_parameter>
<vm_parameter> -Dorg.cougaar.core.node.InitializationComponent=XML
</vm_parameter>
<vm_parameter>-Xms100m</vm_parameter>
<vm_parameter>-Xmx300m</vm_parameter>
<vm_parameter>
 -Dorg.cougaar.core.logging.log4j.category.org.cougaar.lib.web=DEBUG
</vm_parameter>
<vm_parameter> -Dorg.cougaar.core.mts.destq.retry.initialTimeout=500
</vm_parameter>
<vm_parameter> -Dorg.cougaar.core.mts.destq.retry.maxTimeout=10*1000
</vm_parameter>
<vm_parameter>
-Xbootclasspath/p:$COUGAAR_INSTALL_PATH/lib\javaiopatch.jar
</vm_parameter>
 <vm_parameter>
-Djava.class.path=$COUGAAR_INSTALL_PATH\lib\bootstrap.jar </vm_parameter>
<vm_parameter>
-Dorg.cougaar.system.path = $COUGAAR_INSTALL_PATH\sys
</vm_parameter>
<class>org.cougaar.bootstrap.Bootstrapper</class>
</node>
. . .
```

Fig11 – Virtual machine settings for running a node

The settings in the example above are the most common.
The value of "-Dorg.cougaar.society.file" specifies which XML file will be used to load the society.
The value of "-Dorg.cougaar.node.name" specifies the name of the node
The value of "-Dorg.cougaar.install.path" specifies the path to the the Cougaar directory
The value of "-Dorg.cougaar node.InitializationComponent" specifies which type of file will be used to load the society (usually XML file)
The "Xms…m" and "Xmx…m" settings are used to configure the memory usage.

There are a lot of optional Cougaar settings that can be specified as VM parameters. In the example above for instance, the setting "-Dorg.cougaar.core.logging.log4j.category.org.cougaar.lib.web=DEBUG" specifies that the logging level should be set as DEBUG (Plugins can obtain a logging service and record their separate messages at seven different levels: DETAIL – DEBUG – INFO – WARN – ERROR – SHOUT – FATAL; the default level is WARN, which discards all DEBUG and

INFO statements). The "-Dorg.cougaar.core.mts.destq.retry.initialTimeout" and "-Dorg.cougaar.core.mts.destq.retry.maxTimeout" are settings for the Message Transport Service. The first one specifies the initial delay between resending the messages that were not accepted by their destination, while the second one specifies the maximum amount of time that can be allocated for trying to resend these messages.

The rest of the arguments in the example above are standard and they specify the location of the Cougaar JAR files (the COUGAAR_INSTALL_PATH/lib directory) and the third-party JAR files (the COUGAAR_INSTALL_PATH/sys directory).

4.1.3 Configuring the agents

The second step in configuring the society for the application is to define the structure and functionality of the agents. The agent is the principal element in the Cougaar architecture.
An agent typically models a particular organization, business process or algorithm.
These are also specified in XML files, which are loaded at society startup. The XML society configuration files must contain the name of the society, the name of the host(s), the name of the node(s) running on the host, the names of the agent(s) started by the node, and finally, the plugin component(s) loaded into the agents.
Usually, each node has its own XML configuration file.

Within an "<agent>" or <"node"> tag, "<component>" tags are used to specify components. The format is:

```
<component class=CLASS
[ insertionpoint=INSERTION_POINT ]
[ name=NAME ]
[ priority=PRIORITY ]> ( <argument>ARGUMENT</argument> )*
</component>
```

Fig12 – Configuring a component

The default INSERTION_POINT point is the standard plugin insertion point "Node.AgentManager.Agent.Component".
The default NAME is the CLASS followed by paranthesis around a comma-separated list of ARGUMENTS, for example "AnyClass(A,B,C)". Component names are only used to distinguish two components with identical classnames and argument lists.
A component can specify zero or more "<argument>" tags, which are passed to the component at runtime through the "steParameter(Object o)" method as a list of Strings. If zero arguments are specified then "setParameter(Objet o)" method is not called.

The next excerpt shows the general configuration of a Cougaar society.

```
<society ..>
  <host ..>
    <node ..>
     <agent ..>
      <!-- typical plugin -->
      <component class=”MyPlugin”/>
      <!—an agent-level component with parameters -->
      <component
         class=”AnotherExample”
         insertionpoint=”Node.AgentManager.Agent.Component”
         priority=”HIGH”>
         <argument>a=b</argument>
         <argument>green</argument>
      </component>
     </agent>
     <!— optional: node-agent component(s) -->
     <!— optional: agent(s) -->
    </node>
    <!— optional: more nodes -->
   </host>
   <!— optional: more hosts -->
 </society>
```

Fig13 – General configuration of a society

The Cougaar society modeled for this particular application contains two types of Agents: the MapServer agent, which loads the server application (MapServer) of the ISME and the Observer agents, which load the client application (IconApplication) of the ISME. The Observer agents can have in their turn different roles specified as arguments, according to which the functionality and looks of the IconApplication would be modified. Loading these applications into the agents is done according to the Cougaar specifications by plugins.

4.1.4 The MapServer agent

The application makes use at the moment of only one MapServer agent. It is assumed that the server of the application will be able to run continuously, without disruption. In a real situation, this agent should be running on a node placed on the physical machine with the highest connectivity and the least probability to crash. As recommended in the ISME thesis paper, a great improvement to the security and robustness of the network would be to have a number of servers that are physically not located near each other. Should a server crash, the Observer agents could still receive information from the other working servers. These servers would have to be kept up to date by synchronizing them with the other servers. At all times, all the servers should have the same information. This requirement would be rather easily assured by using Cougaar, since the servers could be grouped in the same community.

This means that the communication between them would be realised in a very elegant and reliable fashion, through the robust and secure Cougaar infrastructure.

The MapServer agent uses the capabilities of the predefined org.cougaar.core.agent.SimpleAgent Cougaar class. Functionality is added to the agent by loading of the following plugins:

§ IconMapServerPlugin - the "core" of the MapServer agent
- gets as an argument the name of the image file that will be loaded as map
- is used to load the MapServer application of the ISME
- monitors the joining / leaving of the agents from the Observers community
- takes care of the received messages from Observer agents
- sends back replies or updates to the Observer agents

§ CommunityPlugin – a predefined plugin needed by an agent in order to join a community; in the case of the MapServer agent (which was nominated as the manager of the community in the communites.xml file) this plugin is also responsible for creating the community

§ CommunityViewerServlet – a servlet used for monitoring the community activity

§ HistoryServlet – a servlet which shows the blackboard changes of the agent

When the node containing the MapServer agent is run, the IconMapServerPlugin automatically creates an instance of the MapServer class from the ISME application; in other words, the server starts working.
As stated above, the MapServer agent is also the manager of the whole community. Therefore, all incoming requests to join the community from Observer agents are automatically resolved by the Cougaar infrastructure at the level of the MapServer agent.
The configuration of the MapServer agent:

```
<agent name='MapServer' class='org.cougaar.core.agent.SimpleAgent'>
  <component  name='IconMapServerPlugin'
   class='nl.decis.combined.iconmap.IconMapServerPlugin'
   priority='COMPONENT'
   insertionpoint='Node.AgentManager.Agent.PluginManager.Plugin'>
    <argument>map5.jpg</argument>
  </component>
  <component name='org.cougaar.community.CommunityPlugin'
   class='org.cougaar.community.CommunityPlugin'
   priority='COMPONENT'
   insertionpoint='Node.AgentManager.Agent.PluginManager.Plugin'>
  </component>
  <component
   name='org.cougaar.community.util.CommunityViewerServlet'
   class='org.cougaar.community.util.CommunityViewerServlet'
   priority='COMPONENT'
   insertionpoint='Node.AgentManager.Agent.PluginManager.Plugin'>
  </component>
  <component
   name='org.cougaar.pizza.servlet.HistoryServlet'
   class='org.cougaar.pizza.servlet.HistoryServlet'
   priority='COMPONENT'
   insertionpoint='Node.AgentManager.Agent.PluginManager.Plugin'>
  </component>
</agent>
```

Fig 14 – The configuration of the “MapServer” agent

4.1.5 The Observer agent

The number of Observer agents is variable. For this simulation, the agents should be run on different nodes (JVMs), since in a real situation there would be a one-on-one relation between a node (PDA) and an Agent. Therefore, the Observer agent simulates the field reporter in a real live crisis situation. He is allowed to start / stop the IconApplication which is installed on his PDA, if he is registered as a member of the Observers community. His role in the community is however not specified in the society configuration file but is instead taken from the community file.

The Observer agent uses the capabilities of the predefined org.cougaar.core.agent.SimpleAgent Cougaar class. Functionality is added to the agent by loading of the following plugins:

§ IconMapAppManagerUI – a java servlet used for starting / stopping the client application

§ IconMapAppPlugin - the “core” of the Observer agent;
- loaded in the Observer agent at runtime by the IconMapAppManagerUI
using a special Cougaar service called the AgentContainmentService which

manages the adding / removal of components into agents.
- loads the IconApplication
- sends messages to the MapServer agent
- interprets messages received from the MapServer agent
- sends messages to other Observer agents
- replies to messages received from other Observer agents

- CommunityPlugin – a predefined plugin needed by an agent in order to join a community

- CommunityViewerServlet – a servlet used for monitoring the community activity

- HistoryServlet – a servlet which shows the blackboard changes of the agent

The configuration of an Observer agent:

```
<agent name='Observer_Sigma'>
   <component  name='IconMapAppManagerUI'
      class='nl.decis.combined.iconmap.IconMapAppManagerUI'
      priority='COMPONENT'
      insertionpoint='Node.AgentManager.Agent.PluginManager.Plugin'>
   </component>
   <component name='org.cougaar.community.CommunityPlugin'
     class='org.cougaar.community.CommunityPlugin'
     priority='COMPONENT'
     insertionpoint='Node.AgentManager.Agent.PluginManager.Plugin'>
   </component>
   <component
    name='org.cougaar.community.util.CommunityViewerServlet'
    class='org.cougaar.community.util.CommunityViewerServlet'
    priority='COMPONENT'
    insertionpoint='Node.AgentManager.Agent.PluginManager.Plugin'>
            </component>
  <component
   name='org.cougaar.pizza.servlet.HistoryServlet'
   class='org.cougaar.pizza.servlet.HistoryServlet'
   priority='COMPONENT'
   insertionpoint='Node.AgentManager.Agent.PluginManager.Plugin'>
  </component>
</agent>
```

Fig 15 – The configuration of the "Observer" agent

4.1.6 Server configuration

The servlet server includes many bult-in servlets that are always loaded, providing basic support for listing servlet paths and locating agents.
Let's assume that

- the Cougaar society for the application contains two nodes, IconMapNode1 and IconMapNode2 that will be run on the same JVM on localhost
- IconMapNode1 contains the MapServer agent
- IconMapNode2 contains two Observer agents called Observer_Kappa and Observer_Sigma (so they contains the internally registered, user-developed servlet with path /IconMapAppManagerUI)
- IconMapNode1 starts before IconMapNode2
- the Observers community is specified and has three members: MapServer (manager), Observer_Sigma and Observer_Gamma

By default, two nodes on the same host will race to create a shared local naming registry. Therefore, at runtime, both nodes will attempt to connect to the default Cougaar Apache Tomcat server http://localhost:<port>/.
In this context, several system properties are directly responsible on how the nodes will run on the server.
The scanRange sets a limit for the server's search for an open port. First the given HTTP/HTTPS ports are tried, then they are both incremented by one, tried again, etc. This property has a default value of 100 and can be modified by using it as a VM parameter: -Dorg.cougaar.lib.web.scanRange=intValue
The default HTTP port is 8800 and can also be modified by using it as a VM parameter: -Dorg.cougaar.lib.web.http.port=intValue.
Additionally, the HTTPS port can be specified as a VM parameter: -Dorg.cougaar.lib.web.https.port=intValue (defaults to -1; the typical value is 8400)

So, with the assumption above, IconMapNode1 would start on localhost, port 8800. The server would then try to find a port for IconMapNode2, which should be 8801 (assuming it's not occupied by another node started previously).

Having the two nodes running, the built-in servlets include:

http://localhost:8800 – this URL generates a simple help page, plus links to some built-in servlets

http://localhost:8800/agents - this URL lists the name of the node running on the 8800 port, as well as all the agents it contains: IconMapNode1 and MapServer; it also has a link to the list of all agents on the root (agents on localnode IconMapNode1 plus agents from IconMapNode2)

http://localhost:8801/agents - this URL lists the name of the node running on the 8801 port, as well as all the agents it contains: IconMapNode2, Observer_Kappa and Observer_Sigma; link to the list of all agents on the root (agents on localnode IconMapNode2 plus agents from IconMapNode1) http://localhost:8801/$Observer_Sigma - generates a page for agent Observer_Sigma with links to other built-in servlets

http://localhost:8801/$Observer_Sigma/list - generates a page ("List of Observer_Sigma servlets") which lists the servlet paths that are registered in agent Observer_Sigma. In the example, both the built-in servlets will be listed ("/agents" and "/list"), as well as the user-developed servlets loaded in Observer agents ("/IconMapAppManagerUI", "/communityViewer", "/history")

http://localhost:8801/$Observer_Gamma/IconMapAppManagerUI
- invokes agent Observer_Gamma's "/IconMapAppManagerUI" servlet. This is how agent-level servlets are invoked.

http://localhost:8800/$MapServer/communityViewer - generates a page which displays the existing communities (local and remote); since the MapServer agent is a member of the Observers community, this community will appear under Local Communities

http://localhost:8800/$MapServer/communityViewer?community=Observers
- lists the members of the Observers community which are currently connected: MapServer, Observer_Sigma, Observer_Gamma

4.2 Notes on Plugin Implementation

4.2.1 Plugin - Blackboard relation

Plugins are the essential "compute engines" of each Agent. They are self-contained elements of software that can be loaded dynamically into Agents.
Plugins communicate only with the Agent infrastructure, reacting to the Blackboard events and publishing results to the Blackboard. Plugins are unaware of other Plugins, and therefore cannot be dependent on the presence of other Plugins. Plugins may be specialized by domain so that an Agent operating in a specific domain will use only those Plugins that are relevant and specific to its operation. Plugins bring functionality to the Agent, while the society of Agents provides structure and order.

Plugins do not usually communicate directly with the Blackboard. Instead they are given a proxy object called a Subscriber which manages most of these interactions. This separation of functionality allows Plugin developers to either extend one of several base classes or to write their own Plugin classes from scratch without risk of damaging the delicate interactions between the infrastructure and the Subscriber.

Blackboard Transaction may be represented as a collection of "add object," "remove object," and "change object" messages to be applied atomically to the Blackboard. Rollback is not supported. Pending change events are not visible even to the entity making the changes until the end of the Transaction. It is important to note that it is add/remove/change events which are transaction-controlled, never the internal state of any blackboard objects. This implies that either blackboard objects should be immutable, or that the application must be certain that only one component may modify and/or examine internal state at a time (e.g., via synchronize or transaction-controlled features).

A Subscription is logically a "slice" of the Blackboard as specified by a Predicate that selects the objects of interest. In addition, most Subscriptions both track changes to the Subscription's members since the previous Transaction and maintain a Collection (Java Collection API) of the subscribed elements.
A Predicate is an implementation of the utility class UnaryPredicate that has an execute(Object) method which returns true if and only if the object should be considered part of the set. Predicates are run often—it is a good idea for predicates to exit as quickly as possible and be as inexpensive as possible. Predicates are only executed once per object change.

In the following section the application's two main plugins, IconMapServer and IconMapAppPlugin will be analyzed.

4.2.2 The Cougaar ComponentPlugin

It is recommended that a plugin should extend the ComponentPlugin, which is an abstract class that extends BlackboardClientComponent. These two classes provide basic services and APIs needed by plugins that will use the blackboard.

```
import org.cougaar.core.plugin.ComponentPlugin;
public class MyFirstCougaarPlugin extends ComponentPlugin {...
```

Fig16 – Plugin declaration

Now the plugin should determine what services it needs. Some services are already provided by the BlackboardClientComponent base class, such as the BlackboardService, (provides Plugins with the ability to specify and interact with objects / data of specific interest to that plugin), AlarmService (allows plugins to access the current time of the system as well as set Alarms to be awoken at a specific system time or after a certain amount of real time), AgentIdentificationService (allows all components in an agent to discover of which agent they are a subcomponent), and SchedulerService (provides plugins with a way to register with a scheduling service to be awoken under specific circumstances). However, other services must be requested. Services can be requested directly from the plugin's service broker or the plugin can rely on load-time introspection to set the services for them.
The plugins for the current application extended from the CombinedComponentPlugin, which was in turn an extension of the ComponentPlugin with several additional loaded services: the LoggingService (provides a standardized logging service to all components), the UIDService (provides unique identifiers – UIDs – for blackboard objects that implement the UniqueObject interface) and the CommunityService (provides methods enabling a client

to create / join / leave a community, modify community or member attributes, obtain a list of communities, search for communities and community members based on attributes) .

Then the plugin should override the state methods provided by the infrastructure where necessary. These methods include initialize, load, start, suspend, resume, stop, halt and unload. Note that it is not necessary to override these methods if the plugin does not have specific work that needs to be done during these states.
The IconMapAppPlugin overrides the load method in order to get the role of the agent in the Observers community, prior to initializing its IconApplication

```
public void load()
{
  super.load();
  Community comm = comserv.getCommunity("Observers",null);
   ...
    if ((comm.hasEntity(this.agentId.getAddress())))
      role = (String)comm.getEntity(this.agentId.getAddress())
         .getAttributes().get("Role").get();
    shout("Role="+role);
   ...
  }
}
```

Fig 17 – Excerpt from the IconMapAppPlugin's load method

The load method of the IconMapServerPlugin is where the MapServer application is initialized.
It should be noted that the load state method is executed immediately after the plugin component has been inserted in the agent. Considering this, the plugin's general initialization settings should be made inside this method, while the setupSubscriptions method (which can also be seen as an initialization routine since it's executed only once) should be left only for subscription initialization. The unload state method is executed when the plugin is being removed from the agent and it's generally overridden to unload the loaded services.

4.2.3 Predicates & Subscriptions

Now the plugin is ready to decide what kinds of objects it is interested in or needs to complete its job. In order to collect a view of interesting objects, the plugin must define a predicate defining the object(s) of interest. The predicate will allow the infrastructure to fill in the plugin's subscription with blackboard objects that pass the predicate restrictions. Predicates can either be written in the same class as the Plugin or in a separate class implementing the UnaryPredicate interface.

The IconMapServerPlugin uses two Predicates to subscribe to the MapServer agent's blackboard objects:

- RequestSubscrPredicate – queries all incoming Target Relays placed on the blackboard by Observer agents and collects the ones containing initialization requests in the form of Request objects
- MapModifPredicate – queries all incoming Target Relays placed on the blackboard by Observer agents and collects the ones containing messages as MapModif objects

The IconMapAppPlugin uses the following Predicates to subscribe to the Observer agent's blackboard objects:

- MapRetrievePredicate – queries all incoming Target Relays placed on the blackboard by the MapServer agent collects the ones containing messages as MapRetrieve objects
- QuestionPredicate – queries all incoming Target Relays placed on the blackboard and selects the ones that were not placed by the MapServer (the ones that are supposed to contain questions from other Observer agents)
- MessageAnswerPredicate – collects all Source Relays on the blackboard

The following excerpt shows the implementation of a Predicate (the MapRetrievePredicate)

```
class MapRetrievePredicate implements UnaryPredicate {
   private MessageAddress agentId;
   public MapRetrievePredicate(MessageAddress agentId)
   {
     super();
     this.agentId = agentId;
   }
   public boolean execute(Object o)
   {
     boolean sw = false;
     if (o instanceof IconMapTargetRelay)
     {
         sw = ((IconMapTargetRelay) o).getQuery() instanceof
         MapRetrieve;
     }
     return sw;
   }};
```

Fig 18 – The implementation of the MapRetrivePredicate

Each class extending the ComponentPlugin must implement its two abstract methods:

- setupSubscriptions () – called only once as a pre-execute
- execute() – called every time changes have occurred on the agent's Blackboard

The basic principle on which a plugin works can be described as follows:

- the plugin creates subscriptions in its setupSubscription() method;
- when the subscriptions receive updates, the "awken" execute() is called
- execute() does some operations based on the updated subscriptions
- when execute() is finished, it goes back to "sleep" and the BlackboardService forwards andy changes made to the subscriptions to the rest of the agent

Subscriptions are requested in the setupSubscriptions method of a plugin. When a subscription is created, it is immediately filled with objects matching the subscription's predicate. Consequently, already existing objects can immediately be enumerated with the elements() method of the subscription. In addition, some of these objects may be on the added list of an incremental subscription. Both plugins create in their setupSubscription() method a subscription for each Predicate mentioned before.

The following diagram shows how the Subscription mechanism really works (taking as an example the IconMapServerPlugin):

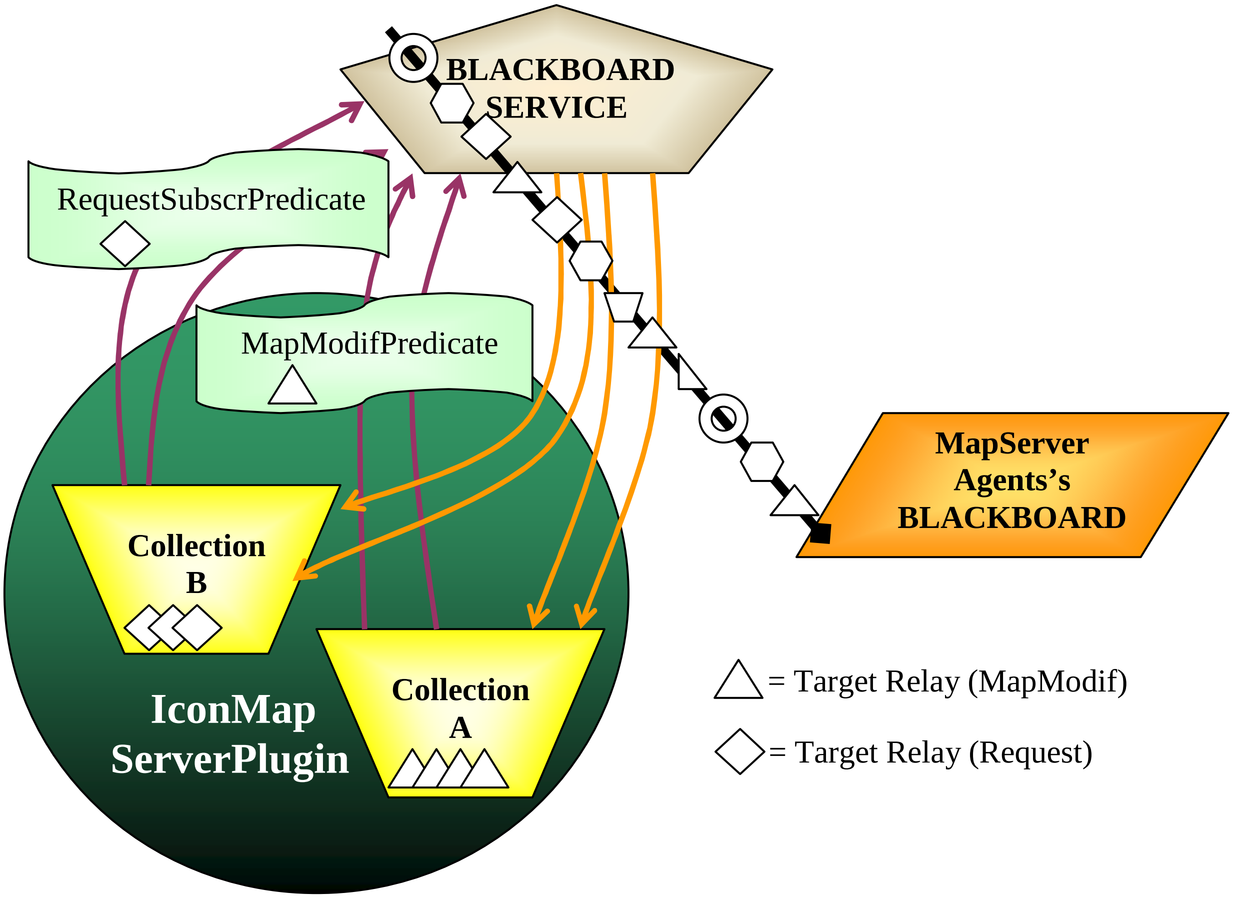


Fig 19 – The Subscription mechanism

ModifSubscr causes objects that are selected by MapModifPredicate to be placed in Collection_A;
RequestSubscr causes objects that are selected by RequestSubscriptionPredicate to be placed in Collection_B.

The following excerpt shows how a subscription is created in the setupSubscription() method:

```
private IncrementalSubscription mapRetrieveSub;
protected void setupSubscriptions() {…
mapRetrieveSub = (IncrementalSubscription)
      blackboard.subscribe(new MapRetrievePredicate(this.agentId));
…}
```

Fig 20 – Setting up subscriptions

4.2.4 The Execute Method

Based on updated subscriptions, the execute() methods of the plugins has to make the necessary changes. In the case of the IconMapServerPlugin, these changes concern the managment of the initialization requests and of the updates received from Observer agents. The execute() method of the IconMapAppPlugin deals with the updates received from the MapServer as well as with messages from other Observer agents.
The following excerpt from the IconMapServerPlugin code will better show the (usual) way of implementing the execute() method.

```
protected void execute() {…
  manageRequests();
…}

private void manageRequests() {
  Enumeration new_requests = RequestSubscr.getAddedList();
  while (new_requests.hasMoreElements())
  {
      IconMapTargetRelay requestMes= (IconMapTargetRelay)
             new_requests.nextElement();
     // get the sender of the request
      String requestor = requestMes.getSource().getAddress();
    // get the verb of the request
      Request req = (Request) requestMes.getQuery();
shout("Recieved request "+req.getRequest()+" from "+requestor);
// THE REPLY TO THE “requestor”
. . .}
```

Fig 21 – Excerpt from the IconMapServerPlugin’s execute method

As can be seen, the result of the RequestSubscr subscription (with the RequestSubscrPredicate matching the Target Relays containing Request objects) is a collection of such blackboard objects which is placed in a Enumeration. Since IconMapTargetRelays (an implementation of a Target Relay which will be discussed in the communication chapter) objects were expected from the named subscription, the elements of the Enumeration are converted to this type of object. Then, the sender of the Relay as well as the Request object contains within the Relay are extracted from the IconMapTargetRelay object. Having acquired these information, the plugin is then ready to send back the reply to the agent who asked for the initialization data.

Generally, the "transaction" mechanism for Plugins is defined by the execute() method. Plugins have "Container level transaction safety" during the scope of the execute() method, thus, within the execute() method, it is sufficient to simply invoke the subscribe method. In some circumsctances, a Plugin may need to provide "Container level transaction safety" outside of execute() cycles.
This also happens is the case of the IconMapAppPlugin, when several transactions need to be managed outside the execute() method. In order to safely invoke the subscribe method outside of the execute() cycle, the following code template has to be used, providing container level transaction safety.

```
myBlackboardService.openTransaction();
mySubscription = myBlackboardService.subscribe(myPredicate);
myBlackboardService.closeTransaction();
```

Fig22 – Blackboard transaction

Another reason for which a code as the sample above has to be used is that a Plugin may have multiple active threads of execution which do not rely on the execute() method being called. To allow other threads safe read and write access to their Subscriptions, the Plugin must explicitly call openTransaction() and closeTransaction().

Because the community is not automatically updated when an Observer agent has left, the IconMapServePlugin has to compare from time to time the state of the community. If it finds discrepancies between the current stored record of connected agents and the list of community members as received from the Community Service it can decide which agents must be removed from the community. Since this operation shouldn't run continuously - in order to keep the node's resources free - the IconMapServerPlugin implements a Timer which periodically launches this operation.
Each Cougaar Agent has two Timers for allowing Plugins to request rescheduling at specific times: a "real-time" system clock and an "execution-time" planning clock. Plugins can use real-time Alarms to manage computer resources or to run expensive simulations once every few minutes. Plugins have simple access to this functionality through the "wake" family of methods on the standard Plugin base classes. Timers and Alarms have a millisecond-level granularity, but have no specific variance – that is, a Plugin will be wakened as soon after the alarm instant as possible as constrained by load, other Plugin function, etc.

4.3 Inter-agent communication

4.3.1 Relays & AttributeBasedAdresses

Under most circumstance, Cougaar developers do not need to be concerned with the communications needed to support their application, since logic providers handle them. Cougaar has two features that address this issue: Relays and AttributeBasedAddresses (ABAs). These mechanisms are independent but frequently used together.

Relays provide a general mechanism for blackboard objects of one agent to have manifestations on the blackboard of other agents.
A Relay is essentially an object wrapper that can be published to an agent's local blackboard and then automatically forwarded to the blackboard of multiple remote agents. While the publisher of a Relay may explicitly identify the recipients, the Relay is often created with an ABA that targets all members of a named community. When a community-based Relay is published, each agent that is a member of the specified community will receive a copy of the Relay payload on its local blackboard. Furthermore, each agent that later joins or leaves the community will have a copy of the payload added/removed from its blackboard until such time as the publisher removes the Relay source.

Relays consist of two interfaces that blackboard objects can implement so that data from a source blackboard can appear on target blackboards and responses from the objects on the target blackboards can appear within the objects on the source blackboard. Objects on the source blackboard implement the Relay.Source interface while those on the target blackboards implement the Relay.Target interface.
The essential features of a Relay.Source are that it has a list of target addresses to which its content should be sent and that it has content to send. The source must also furnish a factory that can be used at the target to construct an object to be published to the target's blackboard from the content object. The source implementation may also retain responses from the targets, but that depends on whether responses are necessary.
The essential feature of a Relay.Target is that it represents the content in some way. Beyond that, the target must furnish a response and the source address if responses are to be used.
The Relay created for the application consists of two classes: IconMapSourceRelay and IconMapTargetRelay, implementing the Relay.Source and the Relay.Target interfaces.

A sender agent will always place a certain message on an IconMapSourceRelay; the message will be available at the receiver's end as the contents of an IconMapTargetRelay.
The Relay mechanism is used in two ways in the application. The first one assumes that the sender doesn't expect an answer from the receiver. For instance, when the MapServer agent agent sends updates to Observer agents, it doesn't need to get any response from them. In this case, the two interfaces,
Relay.Source and Relay.Target, could be implemented by the same class.
In the second case, the responder updates the same Relay received by the sender with his response. This can be exemplified by the communication between two Observer agents. When an Observer agent wants to pose a question to another Observer agent, it creates a

IconMapSourceRelay containing the question. After the second agent has received the question (wrapped in a IconMapTargetRelay), it sends back the answer and indicates this by modifying the state of the IconMapSourceRelay to changed.

AttributeBaseAddresses allow messages to be sent to agents based on their attributes rather than their names.
Attribute-based Addresses are extensions of MessageAddress objects meant to specify the recipient(s) of a multicast message based on the attributes of an agent within a community. Most commonly, the attribute is the agent's role in that community, but other attributes can be used. Attribute-based addresses are commonly used with Relay objects especially when multiple agents might have a given role and want to receive the same message. The AttributeBasedAddress class is used for this form of Attribute-based addressing, delivering sensor data among agents in a community. Such an address is constructed from the name of the community within which the attribute has meaning, the name of the attribute, and its value. This Attribute information is stored in the NameServer, and must have a corresponding context for AttributeBasedAddresses mapped to Agent addresses. AttributeBasedAddresses are ideal for sensor messaging among sensors with the same attribute in a community.

4.3.2 Data flow

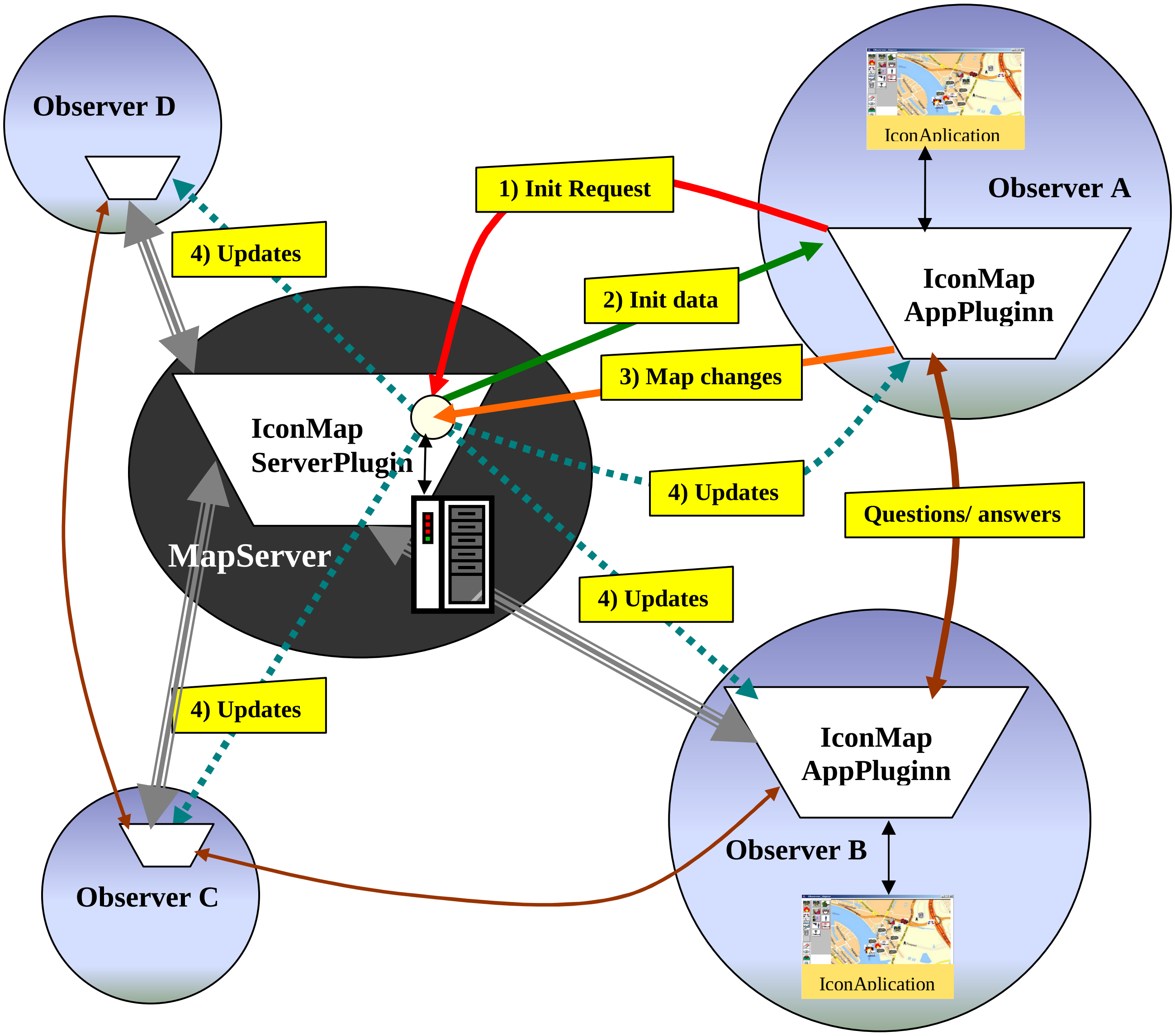


Fig23 – The application's data flow

The application makes use of both the Relay and the Attribute Based Addresses. The Relays are represented by continuous lines. The combination of Relays and ABAs is represented by the dashed lines.
The communication between the agents of the community submits to one of the following:

- Community join request:
  This request is automatically resolved by the CommunityPlugin which was described in a previous section. This plugin is part of the Cougaar community infrastructure which provides fine grained asynchronous events to interested agents when changes occur in

community state. The IconMapServerPlugin registers with its local community service to receive callbacks when changes in community state occur. Support for distributed change events is an important scalability feature, as it eliminates polling by agents for the purpose of detecting changes in community membership or attributes. Scalability is addressed by fact that the MapServer agent exercises flow control in its dissemination of community updates. The rate at which status updates are transmitted is throttled to minimize network traffic during periods of high activity, and all changes generated between updates are aggregated into a single message.

- Initialization request

As a client chooses to start the IconApplication from the associated servlet, the IconMapAppPlugin sends a message to the IconMapServerPlugin, thus informing the MapServer agent that it needs the initialization data. If the server is running, it will receive the message form the Observer agent in the form of a Target Relay placed on its blackboard. This Relay is extracted from the blackboard and its queried for its contents. If the contents is an Request object, then information send back an Object to the client containing the initialization information.
The following sample code shows the creation and the publishing of the IconMapSourceRelay object in the IconMapAppPlugin class:

```
public void setupSubscriptions()
{
  …
   Request req = new Request("init",this.agentId.getAddress());
   IconMapSourceRelay iconMapRelay =
     new  IconMapSourceRelay(uids.nextUID(),
           MessageAddress.getMessageAddress("MapServer"),req);
   blackboard.publishAdd(iconMapRelay);
   shout("Initialization request sent to Map Server");
   …
}
```

Fig 24 – Sending a initialization request to the MapServer

It is obvious that the initialization request should be sent only once by the Observer agent, so that is why this operation is taken care of in the setupSubscriptions() method of the IconMapAppPlugin. The IconMapSoureRelay constructor gets as parameters an unique blackboard object identifier, the name address of the receiver (MapServer) and the contents (the Request object previously created).

- Initialization reply

The MapServer checks its blackboard for IconMapTargetRelays that contain initialization queries sent by Observer agents. The reply will be also in the form of a source Relay containing the initialization information (the map in its currently updated state – placed icons and placed observers)

```
…
Enumeration new_requests = RequestSubscr.getAddedList();
while (new_requests.hasMoreElements())
{
// EXTRACTING THE REQUEST
IconMapTargetRelay requestMes= (IconMapTargetRelay)
                                   new_requests.nextElement();
String requestor = requestMes.getSource().getAddress();
Request req = (Request) requestMes.getQuery();
shout("Recieved request "+req.getRequest()+" from "+ requestor);
…
// SETTING UP THE REPLY
mapRet = new MapRetrieve("init");
…
IconMapSourceRelay replyMes = new
IconMapSourceRelay(uids.nextUID(),
                       MessageAddress.getMessageAddress(requestor),
mapRet);
blackboard.publishAdd(replyMes);
}
```

Fig25 – MapServer's response to request initializations

The next initialization request is extracted from the list of IconMapTargetRelays. Then the sender and the Request object, are in turn extracted from the converted object. The reply contains a MapRetrieve object.

- Initialization data received

The IconMapAppPlugin queries the blackboard for IconMapTargetRelays that contain MapRetreive objects wrapping the initialization information expected from the MapServer.

- Sending map modifications

Map modifications can concern either sending the new/modified reporter position or the reporting of icons. When the client starts its icon application he must introduce his location on the map. Then, as he changes his physical location, he should inform the MapServer about this. Clients report the crisis situation they are witnessing by placing, modifying, deleting icons on their IconApplication map. All these changes take the form of a MapModif object that must take the way of the MapServer agent. Again, an IconMapSourceRelay is used to wrap this information and send it.

- Broadcasting updates

The MapServer checks for any updates received from the Observers; if it finds any such updates, it processes the information received and then broadcasts a message containing the new updates to all the agents currently connected to the community. In this case, the Relay mechanism is combined with the AttributeBasedAddress mechanism, meaning that the reply message is not only being sent to one entity, but to all the members of the Observers community matching the attribute “Role” = “Member”.

```
mapRet = new MapRetrieve("update");
….
MessageAddress target =
AttributeBasedAddress.getAttributeBasedAddress("Observers","Role","Member");
Relay iconMapRelay = new IconMapSourceRelay(uids.nextUID(), target,
mapRet);
blackboard.publishAdd(iconMapRelay);
```

Fig 26 – Broadcasting the updates to the members of the community

- From Observer agent to Observer agent

We assume there are two clients that want to communicate with each other: client A and client B. They appear in the Cougaar community as two Observer agents. When client A wants to contact client B, he writes a question in his IconApplication interface. This message is passed to the associated IconMapAppPlugin which wraps it in an IconMapSourceRelay and sends it to Observer B’s IconMapAppPlugin. This pluing queries the blackboard for IconMapTargetPlugins coming from other Observer agents. When it finds such a message, it extracts the contained question and places on client B’s IconApplication. The IconMapSourceRelay is now in a waiting state, expecting to receive the response from Observer B. When client B eventually places an icon, Observer B’s IconMapAppPlugin is immediately informed and updates the IconMapSourceRelay with the response.

Therefore, the IconMapAppPlugin acts both as a sender and receiver depending whether the agent that acts as a sender or as a receiver.

The following excerpt represents the code that deals with the already updated IconMapSourceRelays. As can be seen, these are selected from the subscription by the getChangedList() method.

```
Enumeration newMessageReplies = mesReplySub.getChangedList();
IconMapSourceRelay message = (IconMapSourceRelay)
                                                          newMessageReplies.nextElement();
// RESOLVE THE “message”
blackboard.publishRemove(message);
```

Fig 27 – Getting the message replies

V. Conclusions and Recommendations

Cougaar proved to be a very efficient tool in adapting the icon based observation application as a distributed agent-based application. The implementation and testing of the application have shown that this framework can be successfully used to wrap swing applications that can afterwards communicate with each other by means of the Cougaar infrastructure.
A lot of tests have been carried out during the process of building the application: agents running on a single node on a single, agents running on multiple nodes on a single, agents running on multiple nodes on multiple hosts. There has also been some successful testing related to the communication between the Cougaar Icon Map application and the Routing application. In regard to this, an interesting future work could be to fully integrate the Routing application to Cougaar. This could be done either by rewriting a Java version of the Routing application and then adding it as a plugin into a Cougaar agent or by placing the output data of the Routing application in a Cougaar LDM (Logical Data Module) plugin so that it could be accessed by the other agents.
Further improvements to the application could concern:

- the decomposing of the Observer agent's "core" plugin into more specialized plugins, where they can deal with specific problems
- the specialization of the Cougaar Observer agents according to their functionality in a real live situation (e.g field observers that can only add icons; field coordinators that can add, delete icons; remote supervisors that can analyze the whole picture of the crisis and contact different field agents etc.)
- designing the model with more than one MapServer agent
- port it to PDA's (assuming that they will be able at some point in the future to support JDK1.4 as required by the Cougaar version used for the application)

Bibliography

MSc Thesis of Paul Schooneman: Icon based System for Managing Emergencies, Delft University of Technology
http://www.kbs.twi.tudelft.nl/Publications/MSc/2005-Schooneman-MSc.html

Crisis simulations: Exploring tomorrow's vulnerabilities and threats, Arjen Boin, Celesta Kofman-Bos, Werner Overdijk; SIMULATION & GAMING, Vol. 35 No. 3, September 2004
http://sag.sagepub.com/cgi/reprint/35/3/378

BBN Technologies, The Cognitive Agent Architecture (Cougaar) Open Source Project - CougaarForge
http://cougaar.org

BBN Technologies, Cougaar Architecture Document V 11.0, 8 March 2004, http://www.cougaar.org/

BBN Technologies, Cougaar Developers Guide V 11.0,
8 March 2004, http://www.cougaar.org/

BBN Technologies, Cougaar tutorial, 6 December 2004
http://tutorials.cougaar.org/

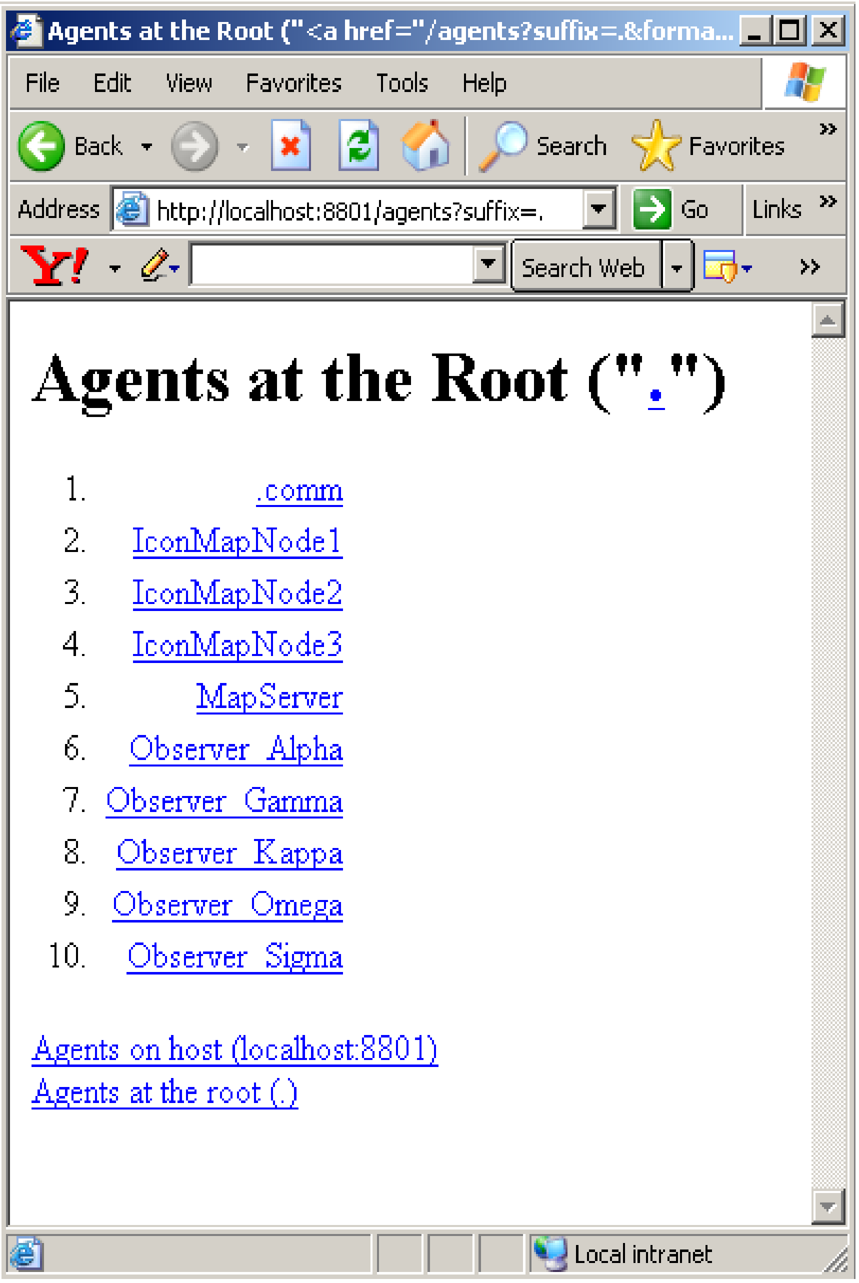
DECIS Lab, Cougaar forum
http://wiki.decis.nl/combined/project/tiki-index.php?page=Cougaar

DARPA UltraLog, Cougaar Agent Communities, Dr. Douglas C. MacKenzie, Ronald D. Snyder; Open Cougaar, New York, 2004,
http://www.mobile-intelligence.com/oc04-comm.pdf

BBN Technologies, Firewall support project, Sebastian Rosset, 16 September 2004
http://fwsupport.cougaar.org/index.html

Appendix

Display of three nodes running on the same host

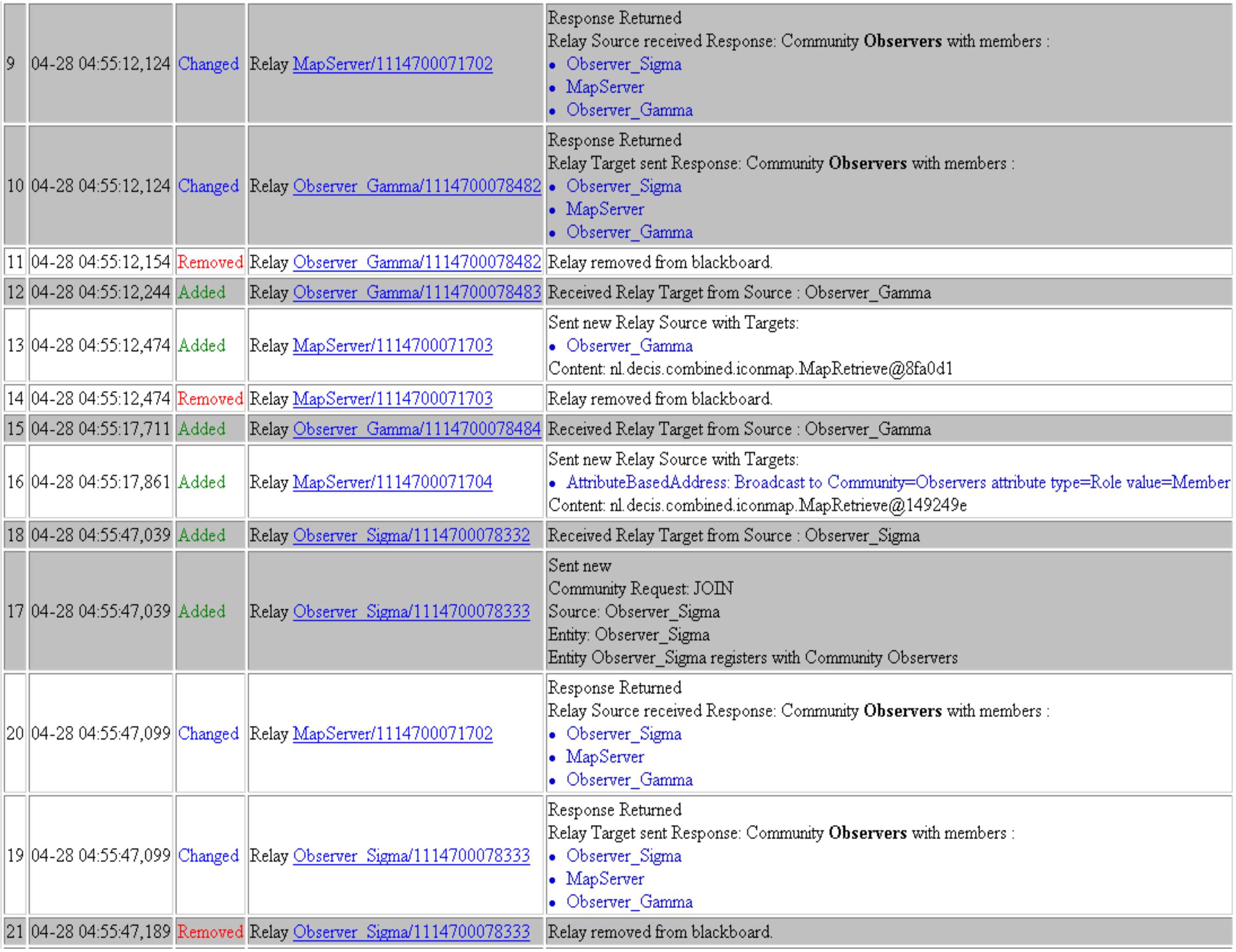

Agents at the Root ("<a href="/agents?suffix=.&forma...
File Edit View Favorites Tools Help
Back
Search
Favorites
Address http://localhost:8801/agents?suffix=.
Go
Links
Search Web
Local intranet

Agents at the Root (".")

1. .comm
2. IconMapNode1
3. IconMapNode2
4. IconMapNode3
5. MapServer
6. Observer_Alpha
7. Observer_Gamma
8. Observer_Kappa
9. Observer_Omega
10. Observer_Sigma

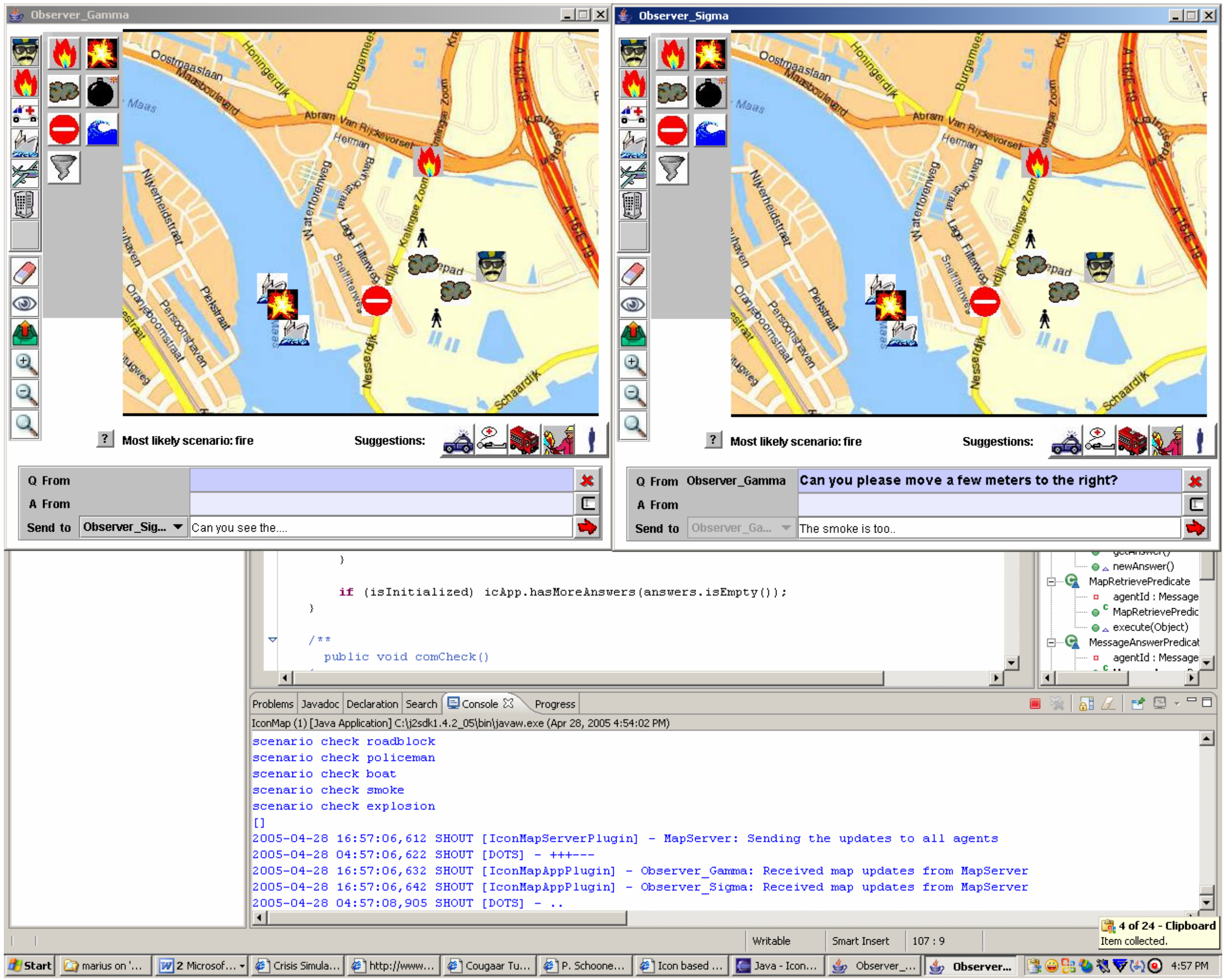
Agents on host (localhost:8801)
Agents at the root (.)

MapServer agent's history servlet (shows all the changes in the agent's blackboard)



9	04-28 04:55:12,124	Changed	Relay MapServer/1114700071702	Response Returned Relay Source received Response: Community **Observers** with members : • Observer_Sigma • MapServer • Observer_Gamma
10	04-28 04:55:12,124	Changed	Relay Observer_Gamma/1114700078482	Response Returned Relay Target sent Response: Community **Observers** with members : • Observer_Sigma • MapServer • Observer_Gamma
11	04-28 04:55:12,154	Removed	Relay Observer_Gamma/1114700078482	Relay removed from blackboard.
12	04-28 04:55:12,244	Added	Relay Observer_Gamma/1114700078483	Received Relay Target from Source : Observer_Gamma
13	04-28 04:55:12,474	Added	Relay MapServer/1114700071703	Sent new Relay Source with Targets: • Observer_Gamma Content: nl.decis.combined.iconmap.MapRetrieve@8fa0d1
14	04-28 04:55:12,474	Removed	Relay MapServer/1114700071703	Relay removed from blackboard.
15	04-28 04:55:17,711	Added	Relay Observer_Gamma/1114700078484	Received Relay Target from Source : Observer_Gamma
16	04-28 04:55:17,861	Added	Relay MapServer/1114700071704	Sent new Relay Source with Targets: • AttributeBasedAddress: Broadcast to Community=Observers attribute type=Role value=Member Content: nl.decis.combined.iconmap.MapRetrieve@149249e
18	04-28 04:55:47,039	Added	Relay Observer_Sigma/1114700078332	Received Relay Target from Source : Observer_Sigma
17	04-28 04:55:47,039	Added	Relay Observer_Sigma/1114700078333	Sent new Community Request: JOIN Source: Observer_Sigma Entity: Observer_Sigma Entity Observer_Sigma registers with Community Observers
20	04-28 04:55:47,099	Changed	Relay MapServer/1114700071702	Response Returned Relay Source received Response: Community **Observers** with members : • Observer_Sigma • MapServer • Observer_Gamma
19	04-28 04:55:47,099	Changed	Relay Observer_Sigma/1114700078333	Response Returned Relay Target sent Response: Community **Observers** with members : • Observer_Sigma • MapServer • Observer_Gamma
21	04-28 04:55:47,189	Removed	Relay Observer_Sigma/1114700078333	Relay removed from blackboard.

Client applications



Most likely scenario: fire

Suggestions:

Q From

A From

Send to Observer_Sig... ▼ Can you see the....

Most likely scenario: fire

Suggestions:

Q From Observer_Gamma Can you please move a few meters to the right?

A From

Send to Observer_Ga... ▼ The smoke is too..

```
        }

        if (isInitialized) icApp.hasMoreAnswers(answers.isEmpty());
    }

    /**
      public void comCheck()
```

```
newAnswer()
MapRetrievePredicate
    agentId : Message
    MapRetrievePredic
    execute(Object)
MessageAnswerPredicat
    agentId : Message
```

IconMap (1) [Java Application] C:\j2sdk1.4.2_05\bin\javaw.exe (Apr 28, 2005 4:54:02 PM)

```
scenario check roadblock
scenario check policeman
scenario check boat
scenario check smoke
scenario check explosion
[]
2005-04-28 16:57:06,612 SHOUT [IconMapServerPlugin] - MapServer: Sending the updates to all agents
2005-04-28 04:57:06,622 SHOUT [DOTS] - +++---
2005-04-28 16:57:06,632 SHOUT [IconMapAppPlugin] - Observer_Gamma: Received map updates from MapServer
2005-04-28 16:57:06,642 SHOUT [IconMapAppPlugin] - Observer_Sigma: Received map updates from MapServer
2005-04-28 04:57:08,905 SHOUT [DOTS] - ..
```