$\mathcal{A}$-globe: Agent Development Platform with Inaccessibility and Mobility Support

David Šišlák, Martin Rehák, Michal Pěchouček, Milan Rollo, Dušan Pavlíček

Gerstner Laboratory©
Czech Technical University in Prague
Technická 2, Prague 6, 166 27 Czech Republic
sislakd@feld.cvut.cz
{rehakm1|pechoucek|rollo|pavlicd}@labe.felk.cvut.cz

Abstract. At present several Java-based multi-agent platforms from different developers are available, but none of them fully supports agent mobility and communication inaccessibility simulation. They are thus unsuitable for experiments with large scale real-world simulation. In this chapter we describe architecture of $\mathcal{A}$-**globe** , fast, scalable and lightweight agent development platform with environmental simulation and mobility support. Beside the functions common to most agent platforms it provides a position-based messaging service, so it can be used for experiments with extensive environment simulation and communication inaccessibility. Simple benchmarks that compare the $\mathcal{A}$-**globe** performance against other available agent platforms are also included.

1 Introduction

In this chapter we present $\mathcal{A}$-**globe** [1], an agent platform designed for fast prototyping and application development of multi-agent systems. $\mathcal{A}$-**globe** provides the same level of services as JADE, COUGAAR, FIPA-OS, JACK (see Section 3 for comparisons and references). Besides presentation of the system itself, we will describe several application scenarios. The main focus of the $\mathcal{A}$-**globe** developers has been given to the following applications domains:

- **simulation**, especially simulation of the multi-agent environment and collective behavior of large communities
- **scalability**, high-number of fully fledged and fully autonomous agents, that are loosely coupled with *lightweight infrastructure*
- **agent migration** persistence and code and state migration within the communication network as much as physical reallocation of the computational host and thus modelling of partial and non-permanent *communication inaccessibility*[2].

The platform provides functions for residing agents, such as communication infrastructure, store, directory services, migration function, deploy service, etc. Communication in $\mathcal{A}$-**globe** is very fast and the platform is relatively lightweight.

$\mathcal{A}$-**globe** platform is FIPA [3] compliant on the ACL level while it does not support the FIPA specification for inter-platform communication, as the addressing and transport-level message encoding were simplified. Interoperability is not necessary for development of the closed systems, where no communication outside these systems is required. This is the case of e.g. agent-based simulations. For large scale scenarios the interoperability also brings problems with system performance where memory requirements, communication speed may become the bottleneck of an efficient collective operation.

$\mathcal{A}$-**globe** is suitable for real-world simulations including both static and mobile units (e.g. logistics, ad-hoc networking simulation), where the core platform is extended by a set of services provided by Geographical Information System (GIS) and Environment Simulator (ES) agent. The ES agent simulates dynamics (physical location, movement in time and others parameters) of each unit.

In this chapter you will learn about the architecture and more technical details about the $\mathcal{A}$-**globe** multi-agent platform (see Section 2), the comparison with different available platforms will be provided (see Section 3). The last section of this chapter will be devoted to simulation and two implemented simulation scenarios (see Section 4).

2 System Architecture

The system integrates one or more agent platforms. The $\mathcal{A}$-**globe** design is shown in Figure 1. Its operation is based on several components:

- **agent platform** – provides basic components for running one or more agent containers, i.e. container manager and message transport layer (section 2.1);
- **agent container** – skeleton entity of $\mathcal{A}$-**globe**, ensures basic functions, communication infrastructure and storage for agents (section 2.2);
- **services** – provide some common functions for all agents in one container;
- **environment simulator (ES) agents** – simulates the real-world environment and controls visibility among other agent containers (section 4.1);
- **agents** – represent basic functional entities in a specific simulation scenario.

Simulation scenario is defined by a set of actors represented by agents residing in the agent containers. All agent containers are connected together to one system by the GIS services. Beside the simulation of dynamics the ES agent can also control communication accessibility among all agent containers. The GIS service applies accessibility restrictions in the message transport layer of the agent container.

2.1 Agent Platform

The main design goals were to develop the platform as lightweight as possible and to make it easily portable to different operating systems and devices (like PDA). The platform is implemented as an application running on Java Virtual Machine (JVM 2 edition 5.0 or higher is required). Several platforms can

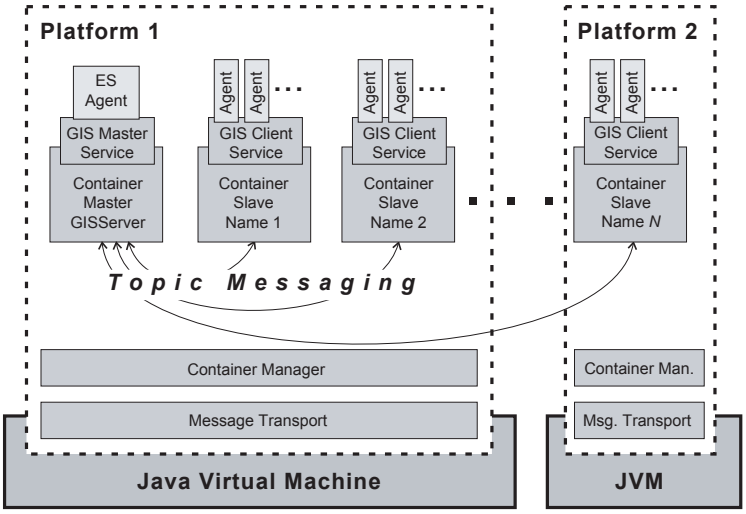

Fig. 1. System Architecture Structure

run simultaneously (maximum 1000) on one computer, each in its own JVM instance. When a new agent container is started, we can explicitly specify in which platform it will be created and run.

The platform ensures the functionality of the rest of the system using two main components:

- **Container Manager.** One or more agent containers can run within single agent platform. Container Manager takes care of starting, execution and finishing of these containers. Containers are mutually independent except for the shared part of the message transport layer. Usage of single agent platform for several containers running on one computer machine is beneficial because it rapidly decreases system resources requirements (use of single JVM), e.g. memory, processor time, etc.
- **Message Transport.** The platform-level *message transport* component ensures an efficient exchange of messages between two agent containers running in a single agent platform (single JVM).

2.2 Agent Container

The agent container hosts two types of entities that are able to send and receive messages: agents and services. Agents do not run as stand-alone applications. Instead, they are executed inside the agent containers, each agent in its own separate thread. The schema of general agent container structure is shown in Figure 2. Container provides the agents and services with several low level functions (message transport, agent management, service management). Most of the higher level container functionality (agent deployment, migration, directory facilitator, etc.) is provided as standard container services.

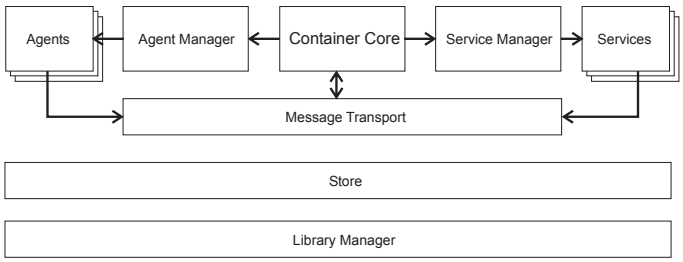

Fig. 2. The Agent Container Structure

The agent container components are:

- **Container Core.** The Container Core starts up and shuts down all container components.
- **Store.** The purpose of *Store* is to provide permanent storage through interface which shields its users from the operating system's filesystem. It is used by all container components, agents and services. Each entity in the agent container (agent, service, container components) is assigned its own virtual storage, which is unaffected by the others. Whenever an agent migrates, its store content is compressed and sent to the new location.
- **Library Manager.** The Library Manager manages the libraries installed in the container and monitors which agents/services use which library.
- **Message Transport.** The Message Transport is responsible for sending and receiving messages from and to the container.
- **Agent Manager.** The Agent Manager takes care of creation, execution and removal of agents on the container. It creates agents, re-creates them after platform restart, routes the incoming messages to the agents, packs the agents for migration and removes agent's traces when it migrates out of the platform or dies.
- **Service Manager.** The Service Manager takes care of starting and stopping the services present in the agent container and their interfacing to other container components. The user can start, stop and inspect the services using a GUI. There are two types of services – user services and system services. The system services are automatically started by the container and form a part of the container infrastructure (agent mover, library deployer, directory services etc.). The system services cannot be removed. The user services can be started by the user or any agent/service. The user services can be either permanent (started during every container startup) or temporary (started and stopped by an agent). In contrast to agents the services are not able to migrate to other containers.

The *container name* must be unique inside one system built from several containers. This name is also used for determination of the specific store subdirectory for the agent container and is registered with the *Environment Simulator Agent*.

Container GUI The *agent container* has a graphic user interface (GUI), which gives the user an easy way to inspect container state and to install or remove its components (agents, services and libraries). The GUI could be shown or hidden both locally and remotely (by message). The GUI screen shot is shown in figure 3.

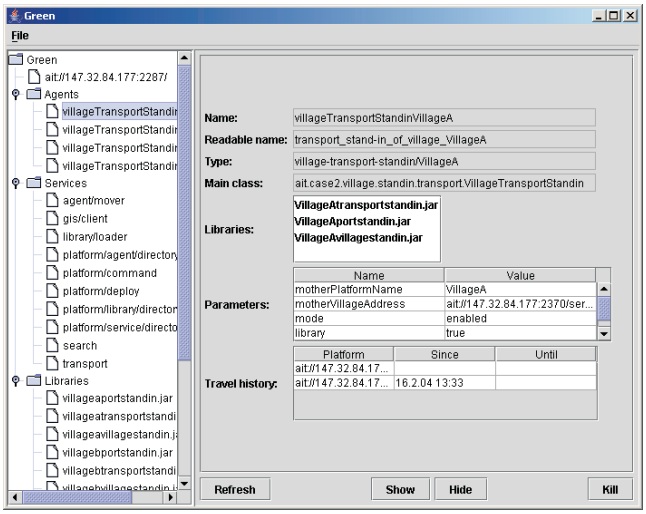

Fig. 3. Container GUI: Agent Information

The window has two parts. The tree on the left side shows names of agents, services and libraries present on the container. The right side shows detailed information about the object selected in the tree. Moreover, the agents and services are allowed to create their own GUI without any restrictions.

Library Manager The *Library Manager* is responsible for the libraries installed in the agent container and monitors the use of these libraries by agents and services. Descriptor of each agent or service specifies which libraries the agent/service requires. The Library Manager is also responsible for automatic library migration when the agent using this library migrates, as described in the dedicated section 2.3. The user can add, remove and inspect libraries using the container GUI.

Every new loaded library in $\mathcal{A}$-**globe** container internally uses the library name constructed from the original library name and SHA-1 hash [4] of the library content. The loaded library is automatically labelled with unique version label constructed as `ver{ver_num_in_the_container}@{container_name}`. This way, two different libraries with same file name can be used in parallel within a single $\mathcal{A}$-**globe** platform. The library can be removed before a class loader opens it. After opening, it can not be removed from the runtime environment. It can be only removed at $\mathcal{A}$-**globe** restart.

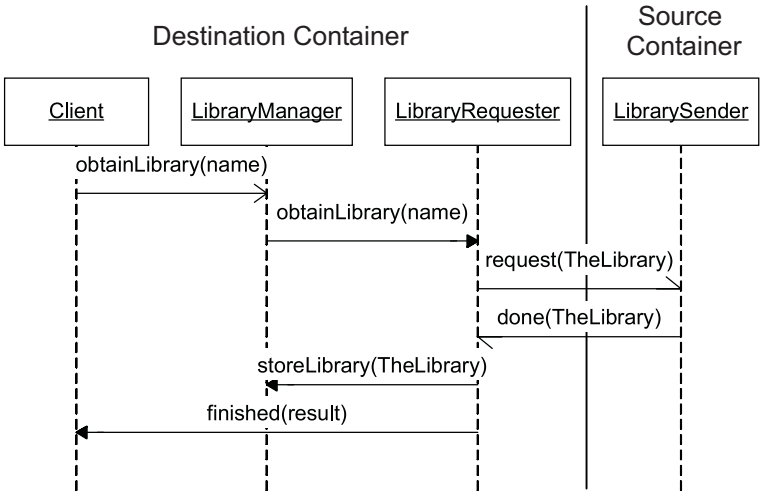

Fig. 4. Library Deployment Sequence Diagram

Class loader is defined for each agent and service. If an agent/service doesn't use any special library, it uses a bootstrap class loader. The bootstrap class loader locates classes only in the name space defined in the path specified by the starting `CLASSPATH` parameter or manifest `CLASSPATH` parameter in the JAR library used by Java runtime. If an agent/service uses specific libraries, it has to define its own class loader which tries to load classes in specified libraries. However, even the agent specific class loader always prefers classes defined by the bootstrap class loader. Therefore, default classes can not be "overridden" - it has no sense to define own `java.lang.String` class because it will never be actually loaded. In **$\mathcal{A}$-globe**, each agent(service) class loader defines agent/service class resolving *name space*. The migration process and the message transport layer always use respective *name space*.

Therefore, several agents with different versions of the same main class can run in a single agent container.

Message Transport The *Message Transport* is responsible for sending and receiving messages. Shared TCP/IP connection for message sending is created between every two agent containers hosted on different agent platforms when the first message is exchanged between them. Messages between two agent containers running in the same agent platform are passed via the platform-level *message transport* component. The message flow inside the container is shown in figure 5.

The message structure respects FIPA-ACL [5]. Messages are encoded either in XML or Byte64 format. Message content can be in XML format or String. The structure of each message content in XML format is described by Document Type Definition (DTD). For coding and decoding XML messages the Java APIs for XML Binding (JAXB) [6] package is used. For transport, all binary data (e.g.

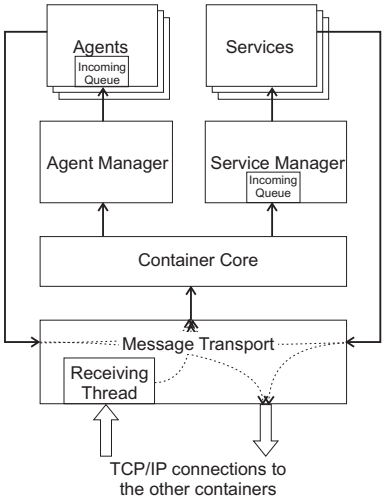

Fig. 5. Message Flow

libraries, serialized agents, etc.) is encoded using the open source Base64 coding and decoding routines.

The *message transport* layer takes care that all message are serialized (marshaled) and deserialized (unmarshaled) in the appropriate class name space depending on the sender and receiver agent/service's class loader.

Agent may receive messages without using conversation discrimination (all messages incoming to this agent are processed in one method), otherwise it must use the *conversation manager* with tasks.

Conversation Manager and Tasks Usually, an agent deals with multiple jobs simultaneously. To simplify the development of such agents, $\mathcal{A}$-**globe** offers *tasks*. A task is able to send and receive messages and to interact with other tasks. The Conversation Manager takes care of every message received by the agent to be routed to the proper task. The decision, to which `Task` a message should be routed, depends on the massage `ConversationID`. The `ConversationID` should be viewed as a 'reference number'.

Agents The agents are autonomous entities with a unique name and an ability to migrate. There is a separate thread created for each agent. A wrapper running in the thread executes the agent body. Whenever an agent enters an error state or finishes its operation, the control is passed back to the wrapper, which handles the situation. The return value of the agent state is used to determine agent's termination type (die, migrate, suspend). Therefore, potential agent failures are not propagated to the rest of the agent container. Agents could be deployed to remote containers.

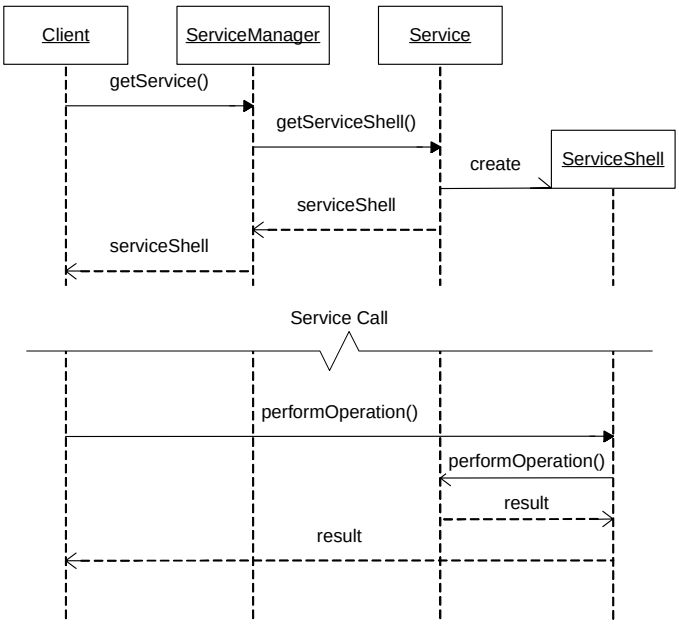

Fig. 6. Service Shell Operation

Services The services are bound to a particular container by their identifier. The same service may be present at several containers and can be also deployed to remote containers. Services handle the requests coming in two forms - as messages or local interface calls.

The agents (and services or container components) have two ways to communicate with a service. Either via normal messages or by using the *service shell*. The service shell is a special proxy object that interfaces service functions to a client. The UML sequence diagram of service shell creation and use is shown in Figure 6.

The advantage of the service shell is an easy agent migration (for migration description see section 2.3): while the service itself is not serializable, the service shell is. When an agent migrates, the shell serializes itself with the information what service name it was connected to. When the agent moves to the new location, the shell reconnects to the same service at the new location. When a service is shut down, it notifies it's shells so that they refuse subsequent service calls.

There are several common services described in the table 1. These services are automatically started by the agent container and provide common functions for all agents.

Agent/Service Naming and Addressing The agent name is globally unique and is normally generated by the platform during agent creation. The service

Service name	Description
`container/command`	Service through which the container core remotely receives commands (show/hide GUI, shutdown)
`container/directory`	It provides extended white pages and directory pages services. It supports inaccessible environment and uses visibility updates provided by ES Visibility servers. The service is automatically started on every client container.
`container/library/directory`	Provides searching of library matching some search criteria
`container/deploy`	Service responsible for starting an agent from agent info record
`gis/master`	Master side of Environment Simulator service
`gis/client`	Client side of Environment Simulator service

Table 1. System services description

name is unique only within one agent container (services cannot migrate) and is specified by the service creator. The address has the following syntax:

`aglobe://platform_ip:port/[agent|service]/name`.

2.3 Agent Migration and Cloning Procedure

In order to successfully migrate, the agent has to support serialization. The *Library Manager* takes care that all necessary libraries are transferred with the agent code. The migration sequence is shown in Figure 7.

All exceptions that might occur during the process are properly handled and the communication is secured by timeouts. If the migration cannot be finished for any reason, the agent is re-created in its original container.

If the `done` message is successfully sent by the agent destination container but never received by the source container, two copies of the agent emerge. If the `done` message is received by the source container, but the agent creation fails at the destination container, the agent is lost. These events can never be fully eliminated due to the possible communication inaccessibility, but maximum caution was given to minimize their probability.

When the migrating agent uses external libraries, the library manager service (see Section 2.2) moves the necessary libraries not available on the new container on behalf of the migrating agent. The migration process makes use of the Java programming language features - serialization and externalization. Whenever an agent migrates or a new agent(service) is deployed, the Library Manager checks which libraries (including the library version) are missing on the container and

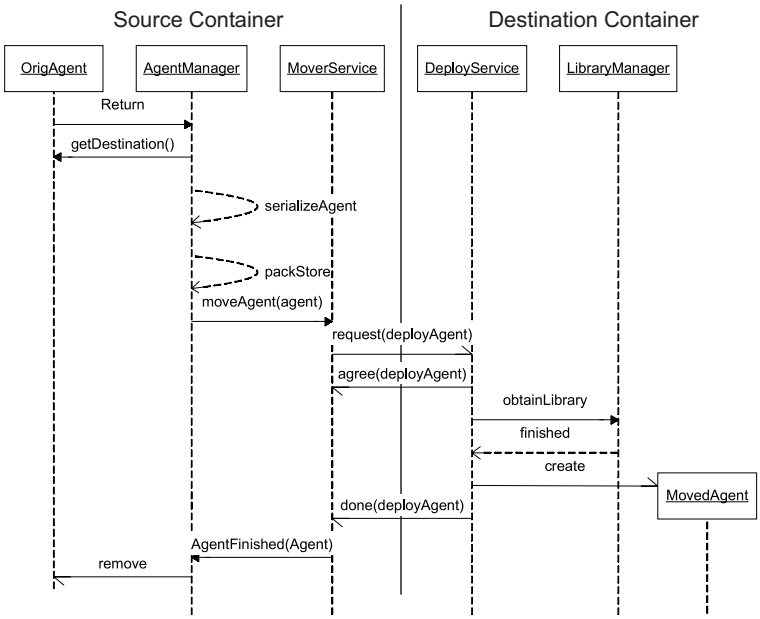

Fig. 7. Agent Migration

obtains them from the source container. The inter-platform functionality of the Library Manager is realized though the service `library/loader` (this service is present on every agent container). Library deployment sequence diagram is shown in Figure 4.

Agent cloning is analogous to the agent migration. The procedure differs only in two points - the clone created on the remote can have a different name, specified by the agent and the original agent is not removed at the end of the operation.

2.4 Sniffer Agent

The Sniffer Agent is an on-line tool for monitoring all messages and their transmission status (delivered or not-reachable target). This tool helps find and resolve communication problems in the system during the development phase.

The sniffer can be started only on an agent container where *gis/master* service is running. After the sniffer starts, all messages between agents and services inside any container or among two agent containers are monitored. Messages can be filtered by the sender or receiver of the message. All messages matching the user-defined criteria are shown in the sniffer GUI, as shown in Figure 8. The message transmission status is visualized by the type of line. The color of the message corresponds to the message performative.

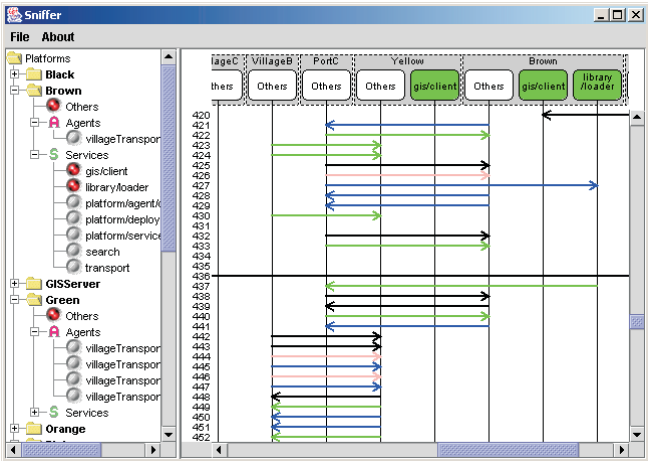

Fig. 8. The sniffer

2.5 Communication Analyzer

The sniffer agent provides a detailed overview of the communication between agents. However, the amount of details provided by the sniffer, combined with the column alignment of agents makes the global view of the interactions in the community difficult to grasp. Quite often, we prefer to clearly see the intensity of interactions between agents, rather than the details of individual communications. This is exactly the service provided by the *communication analyzer*, as shown in Figure 9. Communication Analyzer presents selected agents, filtered by a regular expression, in a circle. Messages exchanged between agents influence the color and the width of the links between communicating agents. In order to keep the image updated, old messages fade away progressively and only the recent ones are visible. Such visualization tool is important especially in situations with limited accessibility, where the fragmentation of the community can be readily perceived. For example in ACROSS scenario (see paragraph 4.2), we can observe the average number of agents participating in auctions or the lack of communication with agents that are not considered to be trustful by others.

3 Platform comparison

This section presents the results of comparison of available JAVA-based agent development frameworks evaluated by an industry expert Pavel Vrba from Rockwell Automation Research Center in Prague [7], which were carried out in a cooperation with the Gerstner Laboratory.

These benchmarks were focused especially on the platform performance which is a crucial property in many applications. Detailed description of the particular features is beyond the scope of this chapter. Firstly, the particular benchmark

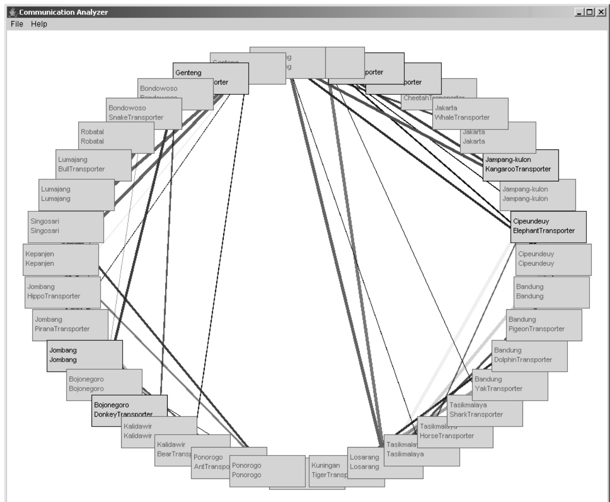

Fig. 9. The Communication Analyzer user interface. Note that two disjoint parts of the agent system can be clearly observed, indicating possible communication failure.

criteria, which the agent platform should provide are identified (e.g. small memory footprint and message sending speed). These benchmarks were carried out for the following agent platforms[1] - JADE [8] [9], FIPA-OS [10], ZEUS [11], JACK [12] and $\mathcal{A}$-**globe** [1]. Shall be noted that the platforms feature various special capabilities that are out of scope of the comparison (e.g. BDI capability in JACK). These features can have an impact on the investigated performance measures.

3.1 Message Speed Benchmarks

The agent platform runtime, carrying out interactions, should be fast enough to ensure reasonable message delivery times. The selected platforms have been put through a series of tests where the message delivery times have been observed under different conditions.

In each test, the so called *average roundtrip time* (avgRTT) is measured. This is the time period needed for a pair of agents (let's say A and B) to send a message from A to B and get reply from B to A. The roundtrip time is computed by the agent A when a reply from B is received as a difference between the receive time and the send time. This message exchange was repeated several times (depending on the type of experiment) and the results were computed as an average from all the trials.

The overall benchmark results are presented in the Table 2. A more transparent representation of these results in the form of bar charts is depicted in

[1] GRASSHOPPER's licence does not allow to use it for benchmarking and other comparison activities.

JAVA-based Agent Development Toolkits/Platforms - Benchmark Results April 2004, Rockwell Automation in Prague						
PIII, 600MHz, 256MB Agent Platform	Message sending - average roundtrip time (RTT)					
	agents: 1 pair messages: 1.000 x ⇆		agents: 10 pairs messages: 100 x ⇆		agents: 100 pairs messages: 10 x ⇆	
	serial [ms]	parallel [s]	serial [ms]	parallel [s]	serial [ms]	parallel [s]
JADE v3.1	0,8	0,36	7,5	0,19	76,3	0,49
JADE v3.1 1 host, 2 JVM, RMI	10,3	4,92	111,9	6,35	1 190,5	7,14
JADE v3.1 2 hosts, RMI	5,79	3,30	68,8	3,71	770,3	2,48
FIPA-OS v2.1.0	28,6	14,30	607,1	30,52	2 533,9	19,50
FIPA-OS v2.1.0 1 host, 2 JVM, RMI	20,3	39,51	205,2	12,50	×	×
FIPA-OS v2.1.0 2 hosts, RMI	12,2	5,14	96,2	5,36	×	×
ZEUS v1.04	101,0	50,67	224,8	13,28	×	×
ZEUS v1.04 1 host, 2 JVM, ?	101,7	51,80	227,9	×	×	×
ZEUS v1.04 2 hosts, TCP/IP	101,1	50,35	107,6	8,75	×	×
JACK v3.51	2,1	1,33	21,7	1,60	221,9	1,60
JACK v3.51 1 host, 2 JVM, UDP	3,7	2,64	31,4	3,65	185,2	2,24
JACK v3.51 2 hosts, UDP	2,5	1,46	17,6	1,28	165,0	1,28
AGlobe v1.0	0,3	0,10	2,8	0,04	28,4	0,09
AGlobe v1.0 1 host, 2 JVM, TCP/IP	2,4	0,33	24,6	0,88	242,7	0,98
AGlobe v1.0 2 hosts, TCP/IP	2,2	0,33	13,9	0,31	96,5	0,44

Table 2. Message delivery time results for selected agent platforms

Figure 10. Three different numbers of agent pairs have been considered: 1 agent pair (A-B) with 1000 messages exchanged, 10 agent pairs with 100 messages exchanged within each pair and 100 agent pairs with 10 messages per pair. Moreover, for each of these configurations two different ways of executing the tests are applied.

In the *serial* test, the A agent from each pair sends one message to its B counterpart and when a reply is received, the roundtrip time for this trial is computed. It is repeated in the same manner N-times (N is 1000/100/10 according to the number of agents). The *parallel* test differs in such a way that the A agent from each pair sends all N messages to B at once and then waits until all N replies from B are received.

Different protocols used by agent platforms for the inter-platform communication are mentioned: Java RMI (Remote Method Invocation) for JADE and FIPA-OS, TCP/IP for ZEUS and $\mathcal{A}$-**globe** and UDP for JACK. Some of the tests, especially in the case of 100 agents, were not successfully completed mainly because of communication errors or errors connected with the creation of agents. These cases are marked by a special symbol.

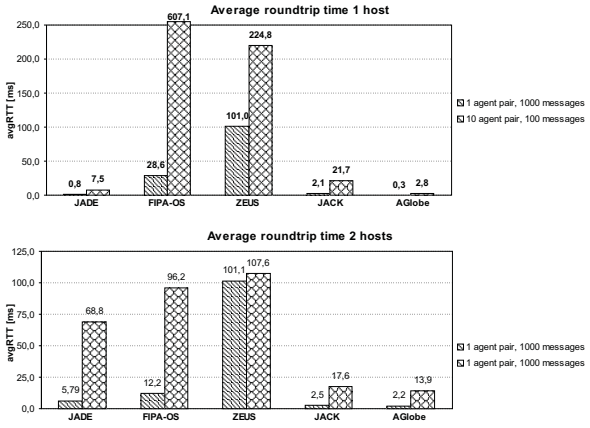

Fig. 10. Message delivery time - serial test results

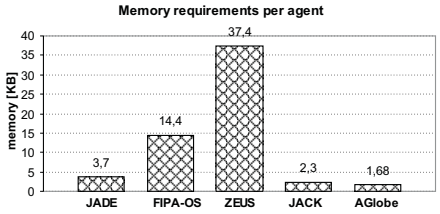

Fig. 11. Approximate memory requirements per agent

3.2 Memory requirements benchmark

This issue is mainly interesting for deploying agents on small devices like mobile phones or personal digital assistants (PDAs) which can have only a few megabytes of memory available. This issue is also important for running thousands of agents on one computer at the same time. Approximate memory requirements per agent can be seen in Figure 11.

4 Simulation

$\mathcal{A}$-**globe** platform is primarily aimed at large scale, real world simulations with fully fledged agents. To support this goal, it includes a special infrastructure for environmental simulation. We will now describe this infrastructure, together with two scenarios implemented using the platform, where we emphasize the concepts used in the simulation of two very different environments for a multi-agent system.

4.1 Simulation support in $\mathcal{A}$-globe

While designing the simulations in $\mathcal{A}$-**globe** platform, we use agents not only to play roles in the simulated world - *actor agents* but we also use them to implement the world where the actor agents act. The agents used for the world simulation are all located in a dedicated master container and are called *Environment Simulation* agents.

These agents only rarely use messages to communicate with actor agents. Instead, they communicate via *topic messaging* - container-to-container messaging specifically reserved for environmental simulation, as shown in Figure 12.

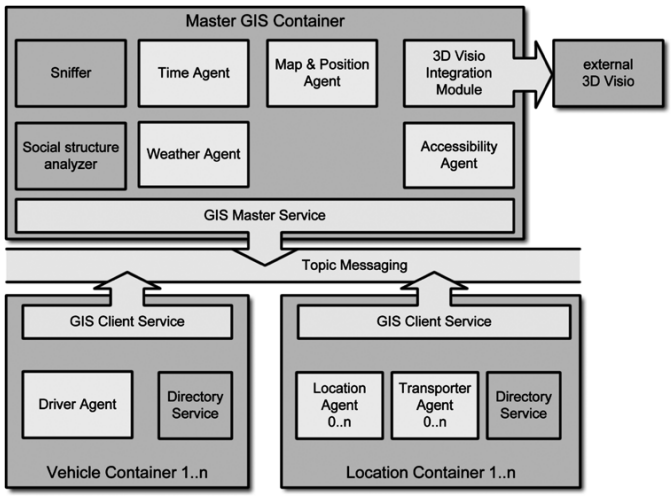

Fig. 12. Topic messaging in $\mathcal{A}$-**globe**

Topic messaging is managed by GIS Service - a special service that is a part of the $\mathcal{A}$-**globe** platform and can be started in a container by specifying an appropriate startup parameter. This parameter value determines whether the container is a *master*, server side container or a *client* - normal container with actor agents.

Client agents subscribe with GIS client service to receive various topics. If such a topic is received by the container, it is distributed to all subscribed agents. Note that all agents in the container receive the same value - this is appropriate in our opinion, as the environment perception shall be identical for all collocated agents. In addition, the agents who wish to act on the environment can submit topics to the GIS service. These topics are then sent to all ES agents in the master container subscribed to receive the topic.

In the nominal configuration, each ES agent manages an internal model of the environment, updates the model with the actions received from actors and submits the environment status to actors in their containers. Each ES agent

can handle one or more topics and one topic can be handled by more then one agent. Specialized ES agents can also subscribe to receive local topics from other ES agents. Typically, many specialized ES agents can receive position information from position agent and use this data to submit appropriate localized environment information to agents.

This approach scales fairly well with the community size. However, when the environment becomes more complex, it is often not economic to handle the environment simulation in the ES agents as the interactions become too cumbersome and internal models too complicated. In this case, the server can use an appropriate GIS server with an ES agent wrapper for simulation purposes. The ES agent(s) is then responsible only for obtaining the information from the appropriate layers of the GIS server and submitting them to corresponding topics. The use of the GIS server is not without a cost - the integration with wrapper agent is rarely flawless and shall be avoided for simple environments.

ES agent can be responsible for nearly any simulation layer, depending on the wishes of the developers. However, a privileged place between ES agents is occupied by *accessibility agents*, who control the existence of communication links between containers holding the actors. Their prominence is caused by the fact that the platform messaging layer is integrated with these agents through predefined topics and any attempt to send a message to an agent in an inaccessible container is automatically unsuccessful.

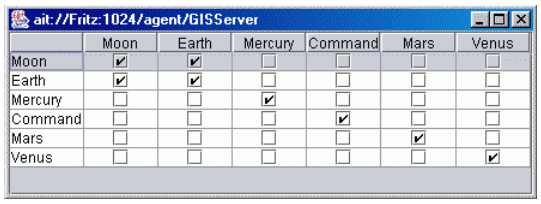

Fig. 13. Platform Visibility Matrix

There are several ES agents implemented and some of them are provided as a part of the $\mathcal{A}$-**globe** platform package for optional use:

- **Manual (Matrix) ES Agent** This agent provides simple user-checkable visibility matrix, as shown in Figure 13. The user simply checks which containers can communicate together and which can not.
- **Distance-based ES Agent** This agent is a fully automatic environment simulator. It receives positions of mobile agent containers representing mobile units in virtual world and automatically controls accessibility between them. The visibility is controlled by means of the simulation of the short range wireless link. Therefore each container can communicate only with containers located inside the predefined radius limit. As the containers move, connections are dynamically established and lost.

Other visibility agents can be implemented for each specific simulation, provided that they respect the ontologies and protocols that apply for them.

4.2 ACROSS – Agent Complex Reasoning Simulation System

To illustrate the concepts of the platform design presented above, we will present the ACROSS scenario that uses the above features to create a relatively rich world where diverse types of agents can interact. Currently, we use this scenario as a common base for multiple research projects, where we need to investigate interactions between a relatively high number of fully-fledged agents.

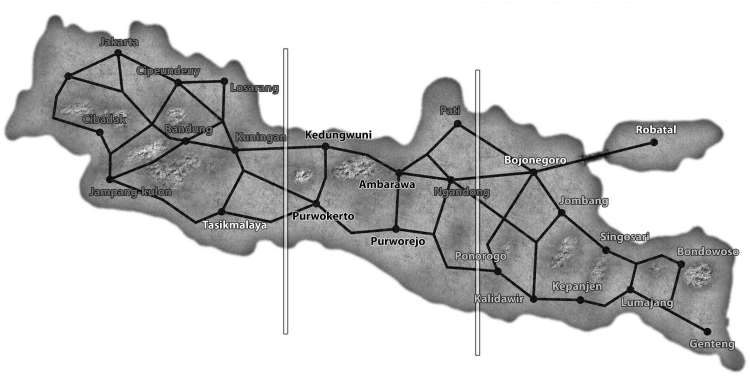

Fig. 14. ACROSS scenario. The geography of the island is modelled after the real Java island in Indonesia, with necessary simplifications.

In our scenario, figure 14, we solve a logistics problem in a non-collaborative environment with self-interested agents. Agents that are part of the scenario have no common goals and their cooperation is typically financially motivated.

We have three types of information about each agent [13]. *Public information* is available to all agents in the system. It includes the agent identity, services proposed to other agents and other relevant characteristics it wishes to reveal. *Semi-private information* is the information which the agent agrees to share with selected partners in order to streamline their cooperation. In our case, resource capacity cumulated by resource type is shared within transporters' alliances (see below). *Private information* is available only to agent itself. It contains detailed information about agent's plans, intentions and resources.

The following types of agents participate in the scenario as actors:

Location Agents: Location agents represent population and natural resources, figure 15 (a). They create, transform or consume resources. As most location agents are unable to completely cover the local demand, they acquire the surplus goods from other locations through one round, sealed bid auctions organized by buyers according to the FIPA CNP protocol [14]. As most

Location agents are physically remote, it is necessary to transport the acquired goods from the provider to the buyer. In order to do so, location agents contract ad-hoc coalitions of transporter agents to carry the cargo, figure 16.

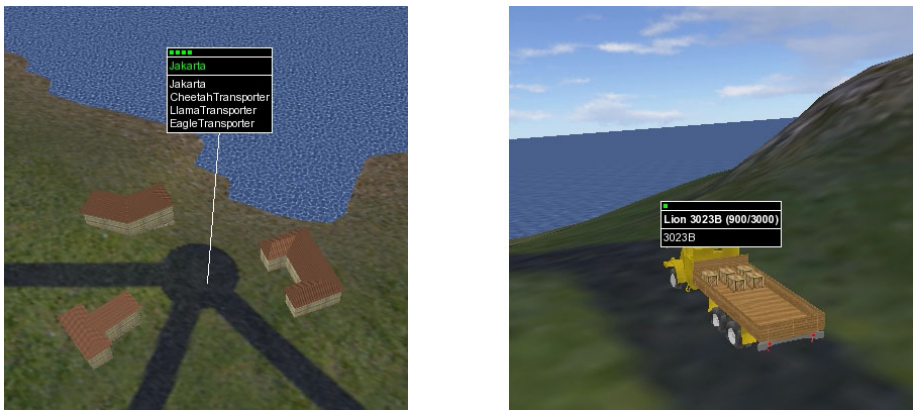

Fig. 15. (a) – Location and 3 Transporter agents in a village container; (b) – a Driver agent in a car container

Transporter Agents: Transporter Agents are the principal agents in our scenario. They use their resources - vehicles, driven by *Driver agents* - to transport the cargo as requested by location agents. As a normal request exceeds the size that may be handled by a single transporter, transporters must form one-time coalitions in order to increase the coverage and thus to be chosen in the auctions. All transporter agents are self-interested and they don't wish to cooperate with all other transporters. They only pick the partners that are compatible with their private preferences. The compatibility is checked using the public information available about the potential partner and agents' private preferences.

While answering the calls for proposals, the agents must form the coalitions relatively fast and efficiently to submit their bid before timeout elapses. Therefore, they use the concept of alliances, discussed in [13], to make the process more efficient. Alliances are groups of agents who agree to exchange the semi-private information about their resources in order to allow efficient pre-planning before starting the coalition negotiation itself. Using the pre-planning, negotiation can directly concentrate on optimization issues, rather than starting from resource query, saving valuable time and messages.

Driver Agents: Driver Agents drive the vehicles owned by Transporter agents, figure 15 (b). They handle path planning, loading, unloading and other driver duties.

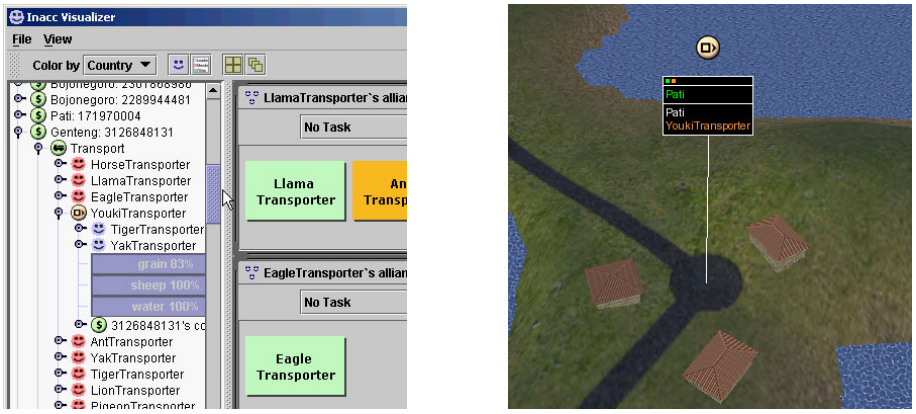

Fig. 16. Location agents contract ad-hoc coalitions of transporter agents to carry the cargo

Numbers of agents actually used can vary from project to project, but the basic configuration uses 25 Location agents, each of them in separate container, 25 Transporter Agents distributed among Location agents' containers and 65 Driver Agents, each with its own container. Besides these "active" agents, we do need several services per container to implement platform functions like GIS, directory or migration management. With the latest optimizations, this configuration runs on a single PC, greatly facilitating the experiments.

Besides the agents mentioned above, several other agents are used for world simulation purposes, as described in 4.1. ACROSS scenario is managed by the following agents:

NodePod Agent simulates the positions and movements of all agent containers (see 2.2) in the simulated world. ACROSS world containers are positioned in the graph. Location agents are placed in a selected node, while the vehicle containers move through the graph following the edges - roads. For each moveable container, at least one agent in this container must be able to communicate the decisions about future directions to the NodePod agent and to handle events generated by the NodePod upon arrival to the graph node. NodePod doesn't take any part in road planning or decision making - it plainly simulates the movements of agent container support on the map following the orders from the Driver agents.

For large scale scenarios, we prefer to handle the movements of agents in a central simulation element, rather than in the container itself. This approach, even if slightly less flexible while adding new agents, pays off thanks to the important savings in the number of messages necessary to run the simulation. In most cases, we require the movements to be smooth, requiring at least 10 simulation steps per second. If the movements are managed in a distributed manner, the system would require 600 messages per second just to report the positions of containers. Besides the sheer number of messages, we must take

into account the fact that many simulation agents require the knowledge of all agent's position in order to generate their output (for example accessibility). Synchronization then becomes an important issue.

Besides the communication with driver agents, the NodePod agent also provides the updated positions of all containers to all other simulation agents in the master container, especially to the Visibility Agent.

Accessibility Agent is an ES agent that simulates the accessibility between the agent containers. It uses the position data received from NodePod ES agent to determine the distance, updates the data with stochastic link failures specified by the configuration parameter and sends the updates to the containers whose accessibility has changed.

We shall note that the two types of inaccessibility - distance based or caused by the link failures - have very different effects on the processes in the community. In the first case, agents who are inaccessible cannot start any direct interaction and this translates into the suboptimal performance of the system, according to the standard economic theories. On the other hand, if the inaccessibility is stochastic, the interactions can indeed start, but the actors must be aware of the possibility that the link can be broken at any time. Therefore, the agents must adopt an appropriate method for inaccessibility resolution, such as use of stand-ins (see [15]), social knowledge or adopted interaction protocols.

Weather Agent maintains the model of the weather in the various parts of the environment. The weather is generated once per each simulation day and submitted to all Location containers. It is then used to adjust the production or consumption of various resources.

Two additional modules are currently integrated with *NodePod* agent. The **3D visualizer** module ensures the selection and formatting of the data for the external visualizers. Besides the pure position data, this module receives the status messages from agents and displays them in the appropriate visualizers. Due to the intensive data flow between this module and external visualizers, we were forced to implement an efficient binary protocol for message sending. The **time module** controls the speed at which the simulation runs. It maps the real time to physical simulation step, therefore influencing the basic pace at which the system runs. Besides this fundamental parameter, we can modify the second parameter, that maps the simulation step to simulation day, used to trigger the recurrent agents' actions, such as production or commercial exchanges.

Commercial Visualizer agent visualizes the auctions, including all bids and selected winners, together with the coalitions of transporters that handle the transportation, as shown in figure 17. It also presents the alliances and their formation described above. In contrast to Sniffer or Communication analyzer agents, this agent is scenario dependent. This makes its integration with other scenarios non-trivial, but the specificity makes the presentation efficient and understandable.

Other ES agents may include for example a Bandit agent, implementing the adversarial actions in the environment, or other various project dependent simulators and visualizers.

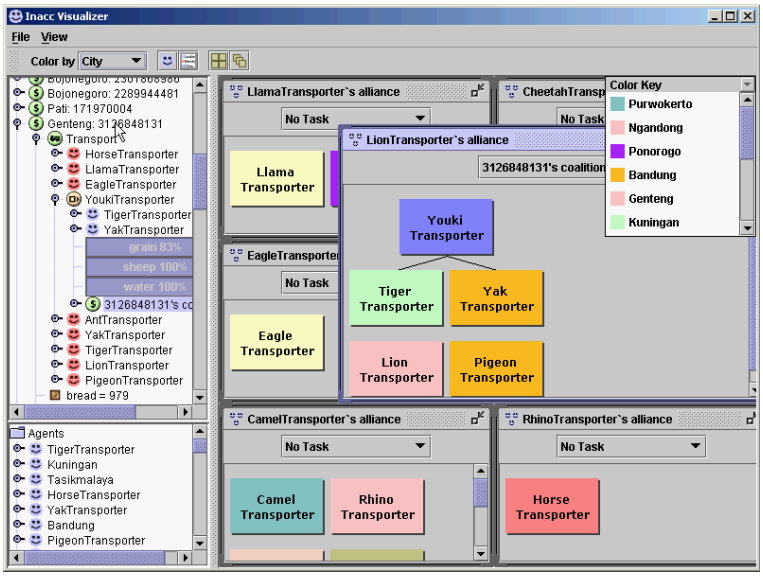

Fig. 17. The commercial visualizer GUI

4.3 Naimt – Naval Automation and Information Management Technology

Features of the $\mathcal{A}$-**globe** platform were also verified on the simulation of identification/removal of mines situated in given area using a group of autonomous robots. This simulation was developed within the Naval Automation and Information Management Technology (NAIMT) project. This software simulation of real-life hardware robots was required to enable scalability experiments and efficient development and verification of embedded decision making algorithms.

The goal of the group of robots is to search the whole area, detect and remove all mines located there. To allow mine removal a video transmission path must be established between the base (operated by human crew that gives the robot a permission to remove the mine) and the robot who has found the mine. Typically, relying via the other robots is necessary, because the video transmission range is limited (e.g. wi-fi connection or acoustic modems in underwater environment). Figure 19 shows an example of robots transmitting a video to base. In this scenario two types of communication accessibility are included:

- **High bandwidth** accessibility, necessary for video transmissions, very restrained.
- **Signaling** accessibility, used for coordination messages and position information, is higher than video accessibility, but remains limited.

All robots in the simulation are autonomous and cooperative. Their dedicated components (coordinators) negotiate in peer-to-peer manner when preparing the

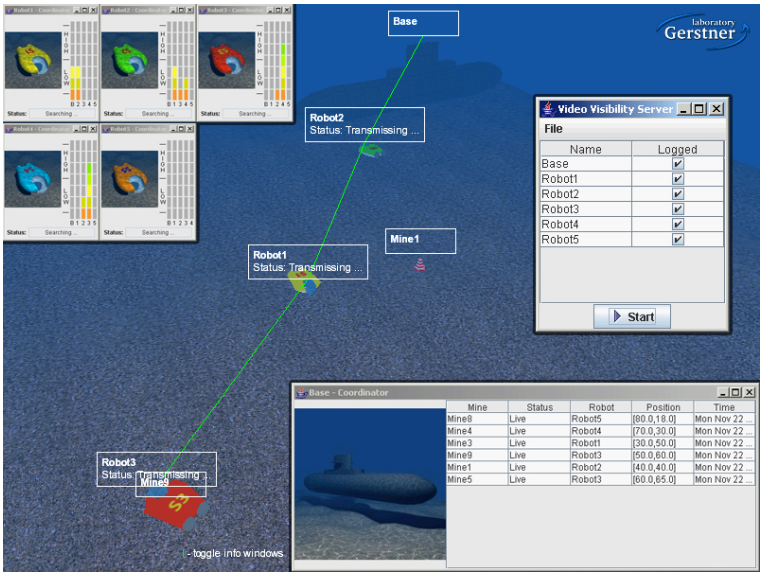

Fig. 18. Relayed communication (link between the robot and base through 4 relays)

transmission path. $\mathcal{A}$-**globe** ES agent and GIS services are utilized during this phase to inform the robot about others within its video transmission range.

Each robot consists of several components, implemented as $\mathcal{A}$-**globe** agents running within one agent container:

- **Robot Pod** simulator, computing robot moves and updating its position with GIS server via GIS service.
- **Mine Detector** simulator, providing the decision-making components with information about found mines.
- **Video** data acquisition and transmission element. This subsystem creates the data feed form the source provided by the simulation and prepares transmission path by remotely spawning one-use transmission agents along the path. Video is then transmitted as a stream of binary encoded messages.
- **Robot Coordinator** implementing search algorithm, transmission coalition establishment and negotiation.

We are using three different approaches to distributed coordination in fixing the ad-hoc data transmission feed. The most straightforward are the approaches relying on a single agent mastering the planning process. Upon finding the mine, it requests the other visible robots to move to a specific positions so that a high-bandwidth transmission link between the mine and the base is established. In two variants of this approach, we may emphasize either the communication quality or the minimization of other robots' actions disturbance. When we optimize the communication quality, relay robots tend to be placed on the join between the mine and the base so that the distance between the relays is minimized and

minimal possible number of robots is used. On the other hand, when we try to minimize the impact on relay robots' own plans, relays are spread in the area between the transmission origin and target, in the proximity of their original areas. In the third approach, the control over the feed planning is not centralized, but rather passed along the communication link relays when the connection is constructed. This approach is well adopted for the environments where the communication is limited and the knowledge necessary for feed building is not common, but rather distributed among robots.

Generality of the $\mathcal{A}$-**globe** technology has been proved when migrating the technology to the robocup soccer environment. The GIS server and ES agents managing the position of the robots have been replaced by the information from the robocup soccer camera. Similarly the nodepod agent has been directly coupled with the hardware of the robocup soccer robots.

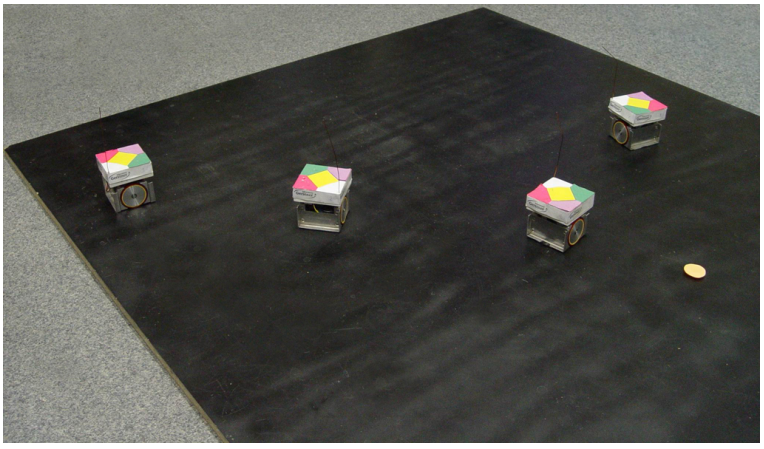

Fig. 19. Relayed communication as simulated by Robocup soccer robots. Robot movements and positions are derived from hardware inputs. Base is in the upper-left corner.

5 Acknowledgement

Authors wish to express acknowledgement to Rockwell Automation Research Center in Prague for mutually beneficial cooperation in the platform evaluation process. $\mathcal{A}$-**globe** agent platform was developed within the project "Inaccessibility in Multi-Agent Systems" (contract no.: FA8655-02-M-4057). The NAIMT deployment has been supported in part by ONR project no.: N00014-03-1-0292.

6 Conclusion

$\mathcal{A}$-**globe** agent platform supports communication inaccessibility, agent migration and deployment on remote containers. These features make $\mathcal{A}$-**globe** a well suited platform for simulation and implementation of physically distributed agent systems with applications ranging from mobile robotics to environmental surveillance by sensor networks. $\mathcal{A}$-**globe** was designed as streamlined lightweight platform which will operate on classical (PC) as well as mobile devices (PDA).

FIPA compliance was not considered a key feature for the simulation-oriented platform the $\mathcal{A}$-**globe** is. However, in the course of the embedding process we are currently planning to implement a dedicated platform service providing FIPA compliance for external communication to cover the gap. Key features the platform currently presents are full mobility support (**??**) and environmental simulation support and platform-level inaccessibility control (4.1). The main driving force of the platform development is the emphasis on easy simulation support and sharp separation between the simulation parts and agent code. Agents that are developed and validated in the simulated environment can be then easily deployed in the real environment, where the GIS service will provide access to the local sensors instead of the simulated values.

The ACROSS scenario is currently exploited with the Agent Technology Group for investigating diverse research concepts in collective decision-making. We study primarily various techniques for coping with agents' communication and coordination in inaccessible and adversarial environments. The **remote presence** techniques include primarily the *stand-in agent* technology [16], [17], [15], while the **remote awareness** concept includes mainly the methods for agents *social knowledge* maintenance and *acquainted models* (e.g. the Tri-base Acquaintance Model [18]).

The ACROSS scenario is used as a benchmark for testing the agents meta-reasoning capacities. The meta-reasoning agents are monitoring the communication exchange in order to reconstruct agents private knowledge. Meta-reasoning in collaborative environments is used mainly for optimization of agents collective behavior [19], [20]. The tri-base acquaintance model has been extended recently for representation of agents' mutual trust and used for formation of trusted and semi-trusted coalitions.

Within the NAIMT scenario, the $\mathcal{A}$-**globe** agents have been used mainly for studying the concept of distributed coordination in partially inaccessible environment. Various techniques of distributed planning, coordination and ad-hoc data transmission processes are currently being investigated [21].

References

1. A-Globe: A-Globe Agent Platform. `http://agents.felk.cvut.cz/aglobe` (2004)
2. Pěchouček, M., Mařík, V., Šišlák, D., Rehák, M., Lažanský, J., Tožička, J.: Inaccessibility in multi-agent systems. final report to Air Force Research Laboratory AFRL/EORD research contract (FA8655-02-M-4057) (2004)
3. FIPA: Foundation for intelligent physical agents. `http://www.fipa.org` (2004)

4. 180-1, F.P.: Federal standard 180-1: Secure hash standard. http://www.itl.nist.gov/fipspubs/fip180-1.htm (1995)
5. FIPA-ACL: Fipa agent communication language overview. Foundation for Intelligent Physical Agents, http://www.fipa.org/specs/fipa00037 (2000)
6. JAXB: JAVA API for XML Binding. http://java.sun.com/xml/jaxb (2004)
7. Vrba, P.: Java-based agent platform evaluation. In Mařík, McFarlane, Valckenaers, eds.: Holonic and Multi-Agent Systems for Manufacturing. Number 2744 in LNAI, Springer-Verlag, Heidelberg (2003) 47–58
8. JADE: Java Agent Development Framework. http://jade.tilab.com (2004)
9. Bellifemine, F., Rimassa, G., Poggi, A.: Jade - a fipa-compliant agent framework. In: Proceedings of 4th International Conference on the Practical Applications of Intelligent Agents and Multi-Agent Technology, London (1999)
10. Poslad, S., Buckle, P., Hadingham, R.: The fipa-os agent platform: Open source for open standards. In: Proceedings of 5th International Conference on the Practical Applications of Intelligent Agents and Multi-Agent Technology, Manchaster (2000) 355–368
11. Nwana, H., Ndumu, D., Lee, L., Collis, J.: Zeus: A tool-kit for building distributed multi-agent systems. Applied Artificial Intelligence Journal **13** (1999) 129–186
12. Fletcher, M.: Designing an integrated holonic scheduler with jack. In: Multi-Agent Systems and Applications II., Manchaster, Springer-Verlag, Berlin (2002)
13. Pěchouček, M., Mařík, V., Bárta, J.: A knowledge-based approach to coalition formation. IEEE Intelligent Systems **17** (2002) 17–25
14. FIPA: Fipa contract net interaction protocol specification. Foundation for Intelligent Physical Agents, http://www.fipa.org/specs/fipa00029 (2002)
15. Rehák, M., Pěchouček, M., Tožička, J., Šišlák, D.: Using stand-in agents in partially accessible multi-agent environment. In: Proceedings of Engineering Societies in the Agents World V (to appear). (2004)
16. Pěchouček, M., Rehák, M., Rollo, M., Šišlák, D., Tožička, J.: Solving coordination inaccessibility in coalition operations. In: Knowledge Systems for Coalition Operatioins 2004, CTU, Prague (2004) 19–36
17. Pěchouček, M., Dobíšek, M., Lažanský, J., Mařík, V.: Inaccessibility in multi-agent systems. In: Proceedings of International Conference on Intelligent Agent Technology. (2003) 182–188
18. Pěchouček, M., Mařík, V., Štěpánková, O.: Role of acquaintance models in agent-based production planning systems. In Klusch, M., Kerschberg, L., eds.: Cooperative Infromation Agents IV - LNAI No. 1860, Heidelberg, Springer Verlag (2000) 179–190
19. Russel, S., Wefald, E.: Do the Right Thing: Studies in Limitted Rationality. The MIT Press, Cambridge, MA (1991)
20. Tožička, J., Bárta, J., Pěchouček, M.: Meta-reasoning for agents' private knowledge detection. In Klusch, M., Ossowski, S., Omicini, A., Laamanen, H., eds.: Cooperative Information Agent VII – Lecture Notes in Computer Science, LNAI 2782, Heidelberg : Springer-Verlag (2003)
21. Bradshaw, J.M., Uszok, A., Jeffers, R., Suri, N.: Representation and reasoning for daml-based policy and domain services in kaos and nomads. In: Autonomous Agents and Multi-Agent Systems (AAMAS 2003), Melbourne, Australia, New York, NY: ACM Press (2003)