

\mathcal{A} -Globe: Agent Platform with Inaccessibility and Mobility Support

David Šišlák, Milan Rollo, and Michal Pěchouček

Department of Cybernetics
Czech Technical University in Prague
Technická 2, Prague 6, 166 27 Czech Republic
sislakd@feld.cvut.cz
{rollo|pechoucek}@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. They are thus not suitable for experiments with real-world simulation. In this paper we describe architecture of newly developed agent platform \mathcal{A} -GLOBE. It is fast and lightweight platform with agent mobility support. Beside the functions common to most of agent platforms it provides the Geographical Information System service to user, so it can be used for experiments with environment simulation and communication inaccessibility. \mathcal{A} -GLOBE performance benchmarks compared against other agent platforms are also stated in this paper.

1 Introduction

\mathcal{A} -GLOBE is an agent platform designed for testing experimental scenarios featuring agents' position and communication inaccessibility, but it can be also used without these extended functions [1]. The platform provides functions for the 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. Comparison to the others agent platforms can be found in section 3.

\mathcal{A} -GLOBE platform is not fully compliant with the FIPA [2] specifications, e.g. it does not support the inter-platform communication (caused by different message format). This interoperability is not necessary when developing closed systems, where no communication outside these systems is required (e.g. agent-based simulations). For large scale scenarios the interoperability also brings problems with system performance (memory requirements, communication speed).

\mathcal{A} -GLOBE is suitable for real-world simulations including both static (e.g. towns, ports, etc.) and mobile units (e.g. vehicles). In such case the platform can be started in extended version with Geographical Information System (GIS) services and Environment Simulator (ES) agent. The ES agent simulates dynamics (physical location, movement in time and others parameters) of each unit.

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 library manager (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) agent** – simulates the real-world environment and controls visibility among other agent containers (section 2.4);
- **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.

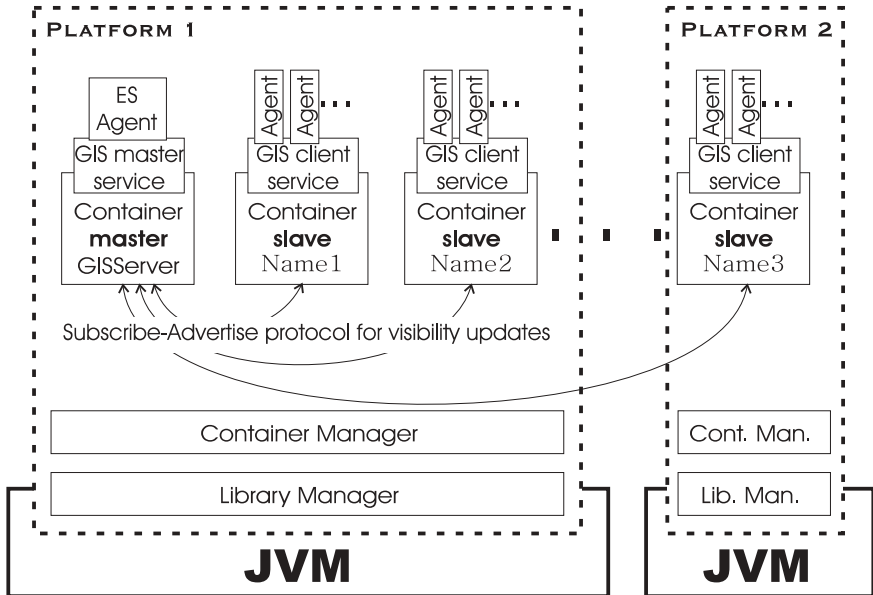


Fig. 1. System Architecture Structure

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 version 1.4 or higher is required). Several platforms can run simultaneously (maximum 1000) on one computer, each in its own JVM instance. When new agent container is started, it can be specified in which platform it will be created and running.

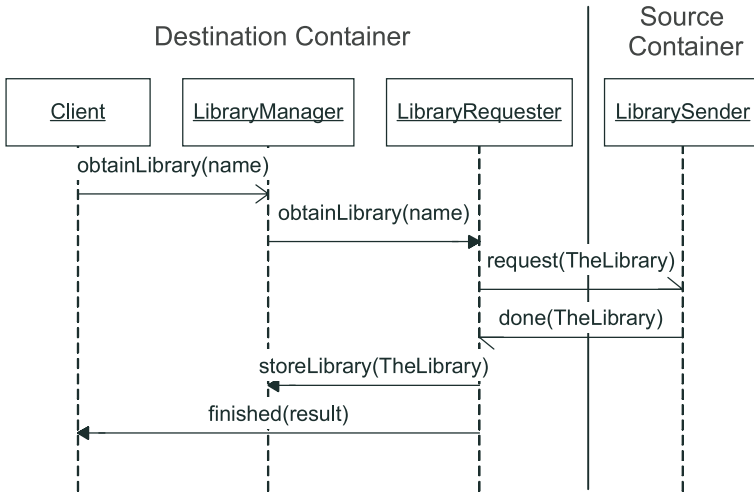


Fig. 2. Library Deployment Sequence Diagram

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

- **Container Manager.** In one agent platform can run one or more agent containers. Container Manager takes care of starting, execution and finishing these containers. Containers are mutually independent except for the shared library manager. Usage of one agent platform for all 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.
- **Library Manager.** The Library Manager takes care of the libraries installed in the platform and monitors which agents/services use which library. Descriptor of each agent and service specifies which libraries the agent/service requires. The Library Manager is also responsible for moving libraries of any agent migrating to other platform when required libraries are not available there. The migration process makes use of the Java programming language features. Whenever an agent migrates or agent/service is deployed, Library Manager checks which libraries are missing on the platform and obtains

them from the source platform. The inter-platform functionality of the Library Manager is realized through the service `library/loader` (this service is present on every agent container). Library deployment sequence diagram is shown on figure 2. The user can add, remove and inspect libraries using the container GUI.

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 a stand-alone applications, but are executed inside the agent containers, each in its own separate thread. The schema of general agent container structure is shown on figure 3. 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.

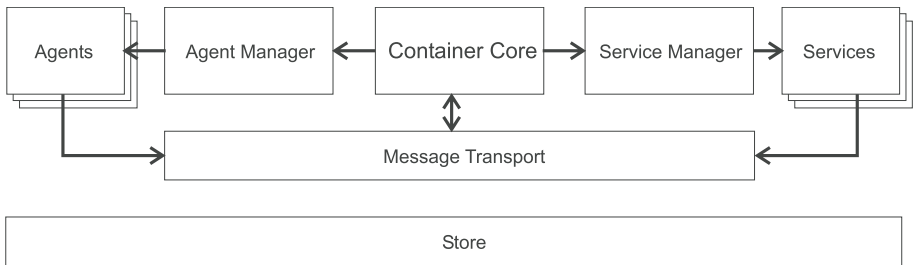


Fig. 3. The Agent Container Structure

The agent container components are:

- **Container Core.** The Container Core starts 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.
- **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 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 user or any agent/service. The user services can be either permanent (started during every container startup) or temporary (started and stopped by some agent). In contrast to agents the services are not able to migrate on other containers.

The *container name* must be unique inside one system build from several containers. This name is also used for determination specific store subdirectory for the agent container and registered to 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 4.

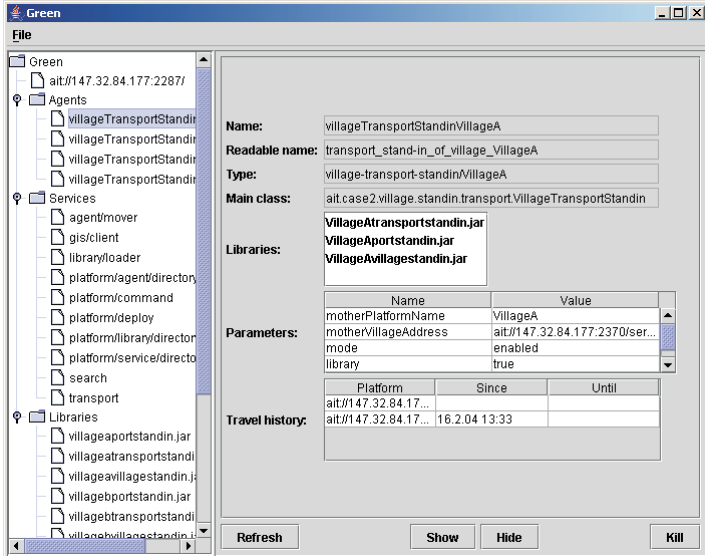


Fig. 4. Container GUI: Agent Information

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.

Message Transport. The *Message Transport* is responsible for sending and receiving messages. Shared TCP/IP connection for message sending is created between every two platforms when the first message is exchanged between them. The message flow inside the platform is shown on figure 5.

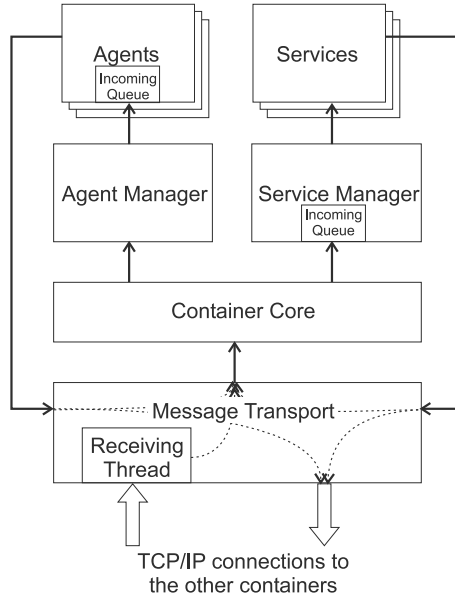


Fig. 5. Message Flow

The message structure respects FIPA-ACL [3]. 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) [4] package is used. For transport, all binary data (e.g. libraries, serialized agents, etc.) are encoded using the open source Base64 coding and decoding routines.

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 agents deal with multiple jobs simultaneously. To simplify a development of such agents, the *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 message *ConversationID*. The *ConversationID* should be viewed as a ‘reference number’.

Agents. The agents are autonomous entities with unique name and 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.

Services. The services are bound to particular container by their identifier. There could be the same service on several containers. Services do not have their own dedicated thread and are expected to behave reactively on response to incoming messages and function calls. Services can be also deployed to remote containers.

The agents (and services or container components) have two ways how 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 on figure 6.

The advantage of 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 information what service name it was connected to. When the agent moves to the new location, the shell reconnects to its 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 platform during agent creation. The service name is unique only within one agent container (services cannot migrate) and is specified by the service creator. An address has the following syntax:

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

2.3 Migration Procedure

In order to successfully migrate, the agent has to support serialization. The migration sequence is shown on 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

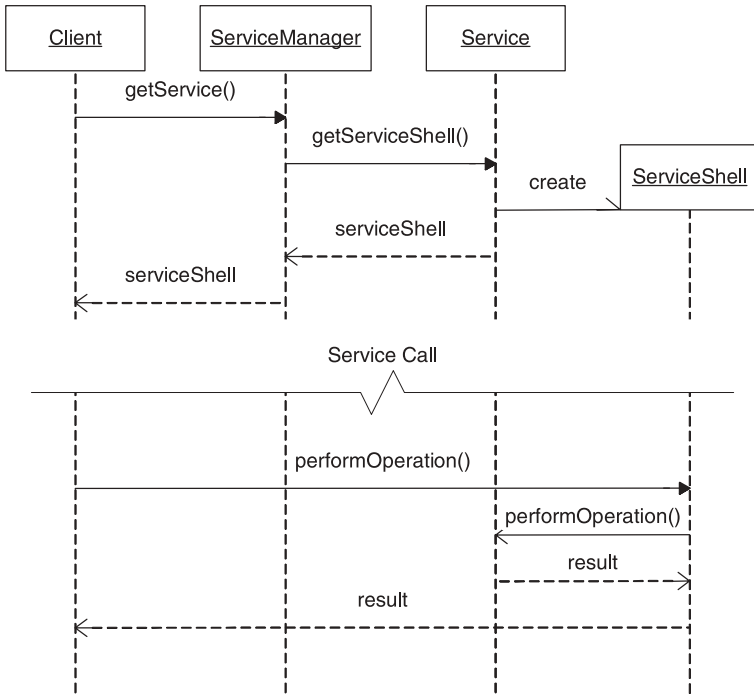


Fig. 6. Service Shell Operation

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.

2.4 Environment Simulator and Communication Inaccessibility Support

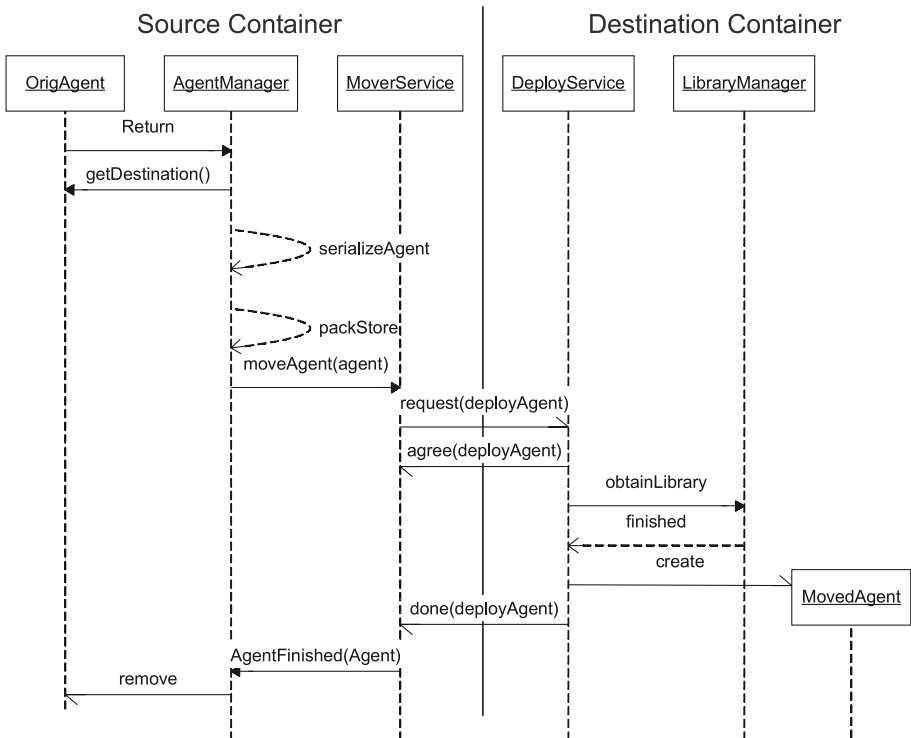
The purpose of Environment Simulator (ES) is to simulate the real world environment. More specifically the ES models the platforms’ mutual accessibility and informs each container about other platforms inside its communication range. Besides visibility, the ES can inform the containers about any other parameters (eg. position, temperature, ...). The ES consists of two parts:

- **ES Agent**, which generates the information and
- **GIS services** that are present at every platform connected in the scenario.

Presence of the ES agent allows the container freely communicate with all other containers. The ES agent architecture allows simulation of complicated container motion and environment dynamics. There are two ES agents implemented:

Table 1. System services description

SERVICE NAME	DESCRIPTION
container/command	Service through which the container core remotely receives commands (show/hide GUI, shutdown)
container/service/directory	Provides searching of service addresses matching some search criteria
container/agent/directory	Provides searching of agent addresses matching some search criteria
platform/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

**Fig. 7.** Agent Migration

- **Manual (Matrix) ES Agent.** This agent provides simple user-checkable visibility matrix, as shown on figure 8. The user simply checks which containers can communicate together and which can not.

	Moon	Earth	Mercury	Command	Mars	Venus
Moon	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Earth	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Mercury	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Command	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Mars	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Venus	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>

Fig. 8. Platform Visibility Matrix

- **Automatic ES Agent.** This agent is fully automatic environment simulator. It computes the positions of mobile agent containers representing mobile units in virtual world and automatically controls accessibility between them. The visibility is controlled by means of simulation of the short range wireless link. Therefore each container can communicate only with containers located inside predefined radius limit. As the containers move, connections are dynamically established and lost.

The GIS services are responsible for distributing this visibility information to all container message transport layers. There are two types of GIS services - one for server side and one for client side, as presented on figure 1.

Server side is container where the ES Agent and `gis/master` service are running. On the client side `gis/client` service is running. After the container startup, the client service subscribes with the `gis/master` to receive the environmental updates (visibility, etc) and this information is than accessible to any container component (agent or service) interested in it. This client services are connected to the message transport layer and control message sending (when agent tries to send message to agent whose container is not visible, this message is returned as undeliverable). If no ES Agent is started, all platforms are connected without any restrictions.

The GIS services in general can be realized at an agent level when one agent in the system will serve as an ES Agent and the others will be informed about the visibility changes via the messages. But in such a case several problems will occur. Agents will have to deal with processing of the incoming messages and no one can be sure that the agents will follow the visibility restrictions they will receive in the message (using the GIS services that are connected directly to the communication layer the simulation correctness is guaranteed). The communication accessibility is computed among the containers that are supposed to represent one physical location where a number of agents can be running. All agents within one container can freely mutually communicate and only the GIS client service is to be informed about the other containers. Using the GIS on agent level, all agents have to be informed about the others (even on the

same container) which may result in high (and mostly useless) communication traffic. The proposed approach is thus more natural and efficient.

2.5 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 to find and resolve communication problems in system during 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 on figure 9. The message transmission status is emphasized by type of line. The color of the message corresponds to the message performative.

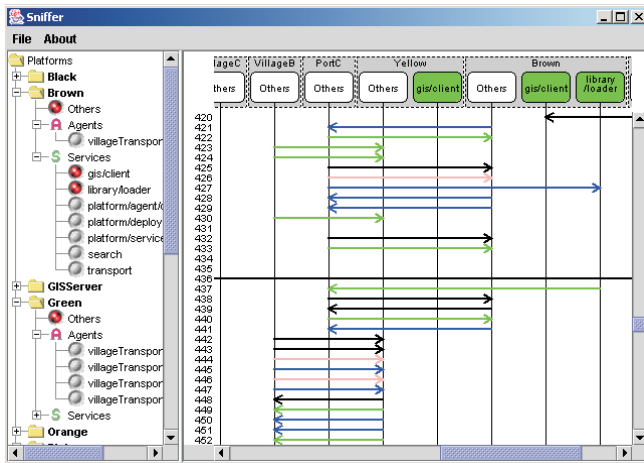


Fig. 9. The sniffer GUI

3 Platform Comparison

This section presents the results of comparison of available JAVA-based agent development frameworks evaluated by an independent expert Pavel Vrba from Rockwell Automation Research Center in Prague [5], 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 paper. Firstly, the particular benchmark criteria, which the agent platform should provide are identified (e.g. small memory footprint and message sending speed). These benchmarks were carried out for

following agent platforms¹ - JADE [6] [7], FIPA-OS [8], ZEUS [9], JACK [10] and \mathcal{A} -GLOBE [11].

Table 2. Message delivery time results for selected agent platforms

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 \leftrightarrow		agents: 10 pairs messages: 100 x \leftrightarrow		agents: 100 pairs messages: 10 x \leftrightarrow	
	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	x	x
FIPA-OS v2.1.0 2 hosts, RMI	12,2	5,14	96,2	5,36	x	x
ZEUS v1.04	101,0	50,67	224,8	13,28	x	x
ZEUS v1.04 1 host, 2 JVM, ?	101,7	51,80	227,9	x	x	x
ZEUS v1.04 2 hosts, TCP/IP	101,1	50,35	107,6	8,75	x	x
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

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, so called *average roundtrip time* (avgRTT) is measured. This is the time period needed for a pair of agents (let 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.

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

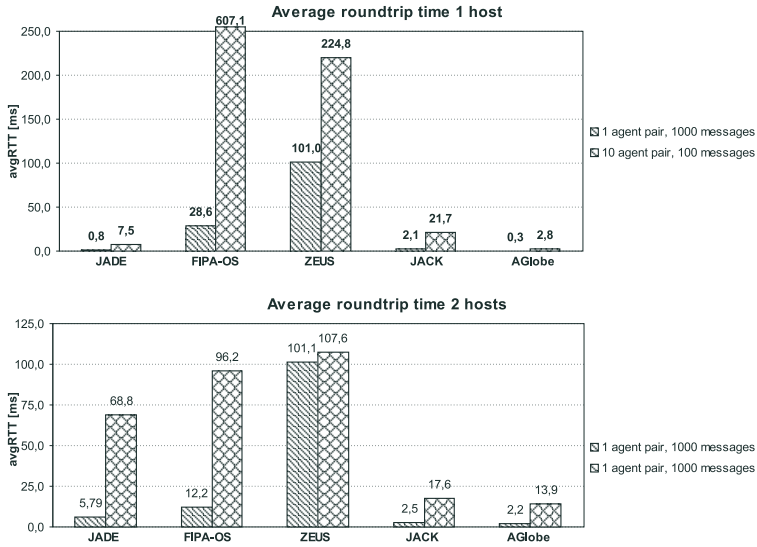


Fig. 10. Message delivery time - serial test results

The overall benchmark results are presented in the table 2. More transparent representation of these results in the form of bar charts is depicted in 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 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 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.

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 the one computer at the same time. Approximate memory requirements per agent can be seen on figure 11.

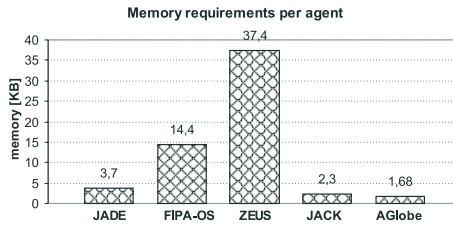


Fig. 11. Approximate memory requirements per agent

4 Simulation

Features of the \mathcal{A} -GLOBE platform were 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 12 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, is limited.
- **Signaling** accessibility, used for coordination messages and position information, is currently assumed to be perfect.

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

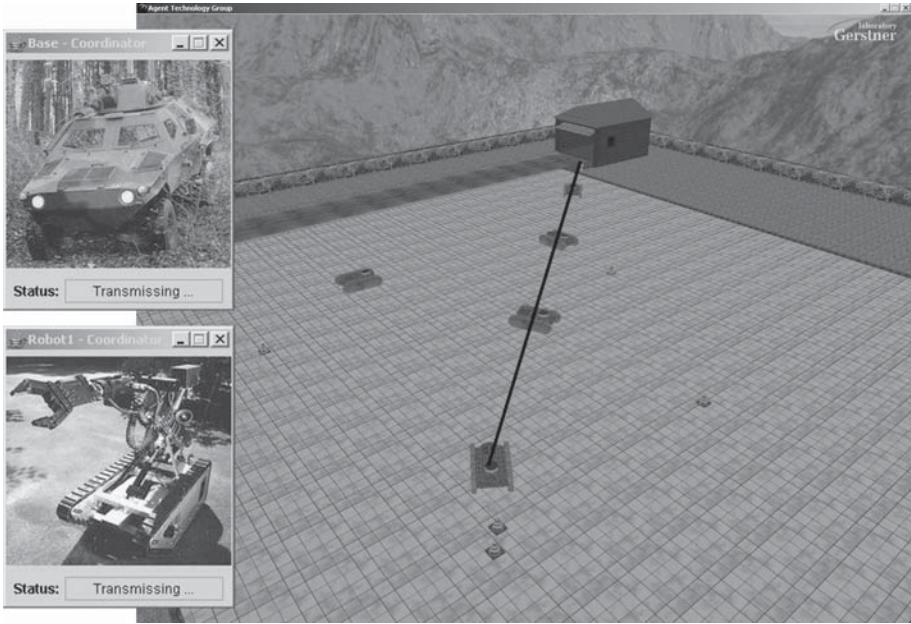


Fig. 12. Relayed communication (link between the robot and base through 3 relays)

5 Acknowledgement

Authors wish to express acknowledgement to Rockwell Automation Research Center in Prague for mutually beneficial cooperation in the platform evaluation process. *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

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

As can be seen, *A*-GLOBE has the best results in all message sending speed benchmarks (table 2) from all selected agent platforms. In comparison with its main competitors, JADE and FIPA-OS, the *A*-GLOBE is at least two times faster than JADE and six times faster than FIPA-OS. *A*-GLOBE has not any communication errors. Also in memory benchmark (figure 11) *A*-GLOBE has one of the smallest memory requirement per agent.

References

1. Pěchouček, M., Mařík, V., Šišlák, D., Reháček, 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)
2. FIPA: Foundation for intelligent physical agents. <http://www.fipa.org>, 2004
3. FIPA-ACL: Fipa agent communication language overview. Foundation for Intelligent Physical Agents, <http://www.fipa.org/specs/fipa00037> (2000)
4. JAXB: JAVA API for XML Binding. <http://java.sun.com/xml/jaxb> (2004)
5. Vrba, P.: Java-based agent platform evaluation. In Mařík, McFarlane, and Valckenaers, editors, *Holonic and Multi-Agent Systems for Manufacturing*. Number 2744 in LNAI, Springer-Verlag, Heidelberg (2003) 47–58.
6. JADE: Java Agent Development Framework. <http://jade.tilab.com> (2004)
7. 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)
8. 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*, Manchester (2000) 355–368
9. 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
10. Fletcher, M., *Designing an integrated holonic scheduler with jack*. In *Multi-Agent Systems and Applications II*, Manchester, Springer-Verlag, Berlin (2002)
11. A-Globe. A-Globe Agent Platform. <http://agents.felk.cvut.cz/aglobe> (2004)