

USING JAVASPACES TO IMPLEMENT A MOBILE MULTI-AGENT SYSTEM

Alf Inge Wang*

Dept. of Computer and Information Science,
Norwegian University of Science and Technology (NTNU),
N-7491 Trondheim, Norway.

ABSTRACT

This paper investigates how a framework for sharing objects in a distributed setting can provide the same support for mobile agents as a framework made specific for this purpose. The paper describes experiences from migrating a multi-agent system using the JavaSpaces framework from Sun and that was originally implemented using the Aglets agent framework from IBM. Our mobile agent system is named CAGIS DIAS and consists of four main parts: agents, agent meeting places, workspaces and repositories. In this paper, we describe how we had to modify the agents and the agent meeting places to use JavaSpaces as the underlying technology for mobility and agent coordination. Our experiences indicate that the JavaSpaces technology is useful for building mobile agent systems. In addition, the experiences shows some advantages and short comings of using this approach.

Keywords: Mobile Agents, Distributed objects, JavaSpaces, Aglets.

1 INTRODUCTION

In the last couple of years many researchers have focused on distributed and mobile technology, resulting in various implementations. Most of these implementations are based on Java, and various technologies are used to provide distribution of data and executional components. Some systems only provide distribution of data, while others enable mobility of both data and programs. In 1999, we implemented a mobile, multi-agent architecture using the Aglets framework from IBM in Japan [8, 4] as underlying technology for mobility. When our implementation was finished, we discovered that IBM was not going to proceed further development of the Aglets framework. This made it hard to continue to use Aglets, due to problems with integration of other products and new versions of Java. We therefore decided to look for alternative technologies to provide distribution and mobility support. We then initiated a small project where two last year master students were going to migrate our mobile, multi-agent system from Aglets to Sun's JavaSpaces. In this work, the goal was to investigate: "Can a framework for sharing objects in a distributed setting provide the same support for mobile agents as a framework made specific for this purpose?".

The JavaSpaces technology was chosen because it provided

simple high-level API and concepts, and provided a framework for mastering a dynamic, distributed environment of Java-objects. JavaSpaces was not originally designed to support mobile agents, but provide mechanisms for transferring Java objects between machines in a network. This paper describes how we implemented a mobile, multi-agent architecture using JavaSpaces, reports the experiences we gained from using the JavaSpaces and identifies some advantageous and disadvantageous with this approach.

The rest of the paper is organised as following: Section 2 describes the core mechanisms in JavaSpaces, section 3 describes our mobile, multi-agent architecture, section 4 describes how we implemented the agent architecture using JavaSpaces, section 5 reports experiences from using JavaSpaces to implement a mobile agent system, section 6 relate our to similar research, and section 7 concludes the paper.

2 JAVASPACES

JavaSpaces [6] from Sun is based on Java RMI and JINI, and is a framework for handling and managing exchange of distributed objects. JavaSpaces technology uses a unified mechanism for dynamic communication, coordination, and sharing of objects between Java technology-based network resources like clients and servers. A JavaSpace is a virtual space between providers and requesters of network resources or objects. This allows participants in a distributed solution to exchange tasks, requests, and information in the form of Java technology-based objects. There are four primary operations on a JavaSpace (illustrated in figure 1):

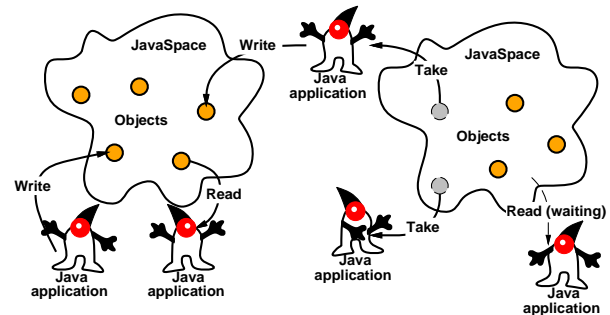


Figure 1. The four primary JavaSpaces operations

*Tel: +47 73 594485, Email: alfw@idi.ntnu.no

1. **Write** an entry (Java Object) into a JavaSpace.
2. **Read** an entry from a JavaSpace that matches some specified parameters.
3. **Take** an entry from a JavaSpace that matches some specified parameters (removing it).
4. **Notify** a specified object when entries that match some specified parameters are written into this JavaSpace.

An *entry* is in JavaSpace terminology a typed group of objects, expressed in a class for the Java platform. JavaSpace technology offers much of the same functionality required in a mobile multi-agent system (MAS); movement of objects, message handling, sharing of objects, etc. Maybe the most useful functionality in this respect, is the support for searching for objects with certain properties.

3 MOBILE SOFTWARE AGENT ARCHITECTURE FOR COOPERATIVE SOFTWARE ENGINEERING

Our mobile software agent architecture is named CAGIS Distributed Intelligent Agent System (DIAS), and can be used to implement process support for cooperative software engineering (CSE). By cooperative software engineering we mean large-scale software development and maintenance work where in a distributed organisation where people must cooperate to produce their products. CAGIS DIAS can be used to implement support for distributed, cooperative activities between roles such as resource negotiation, brainstorming, voting, coordination of artifacts etc. The first version of the multi-agent architecture was implemented in Java, using the IBM Aglets framework to provide mobile agents, KQML was used for inter-agent communication, and ORBIX CORBA was used to offer communication to other applications and other agent systems. The CAGIS DIAS consist of the following four main components (also shown in figure 2):

- **Agents:** An agent is set up to achieve a modest goal, characterised by autonomy, interaction, reactivity to environment, as well as pro-activeness. We have identified three main types of agents: (1) *User agents* that interact with users and perform the tasks they are defined to do for the users, (2) *Interaction agents* that assist with cooperative work between workspaces (negotiation agents, coordination agents), and (3) *System agents* that give system support to other agents and Agent Meeting Places (monitor agents, mediation agents, etc.).
- **Agent Meeting Place (AMP):** AMPs are where agents meet and interact. AMPs support agents in doing efficient inter-agent communication. There can be different types of AMPs for different purposes. Each AMP will have a defined ontology, which the agents have to follow. We can perceive special AMPs for negotiation, coordination, information exchange, selling and buying services etc.

- **Workspaces:** A workspace is a temporary container for relevant data (artifacts, models etc.) in a suitable format to be accessed by tools, together with the processing (work) tools. It can be private, as well as shared. Files stored in a repository can be checked in and out to a workspace.
- **Repositories:** Repositories can be global, local, or distributed, and are persistent storage of data. Experience Bases are one specific type of repository, that we can use in our multi-agent architecture to support community memory.

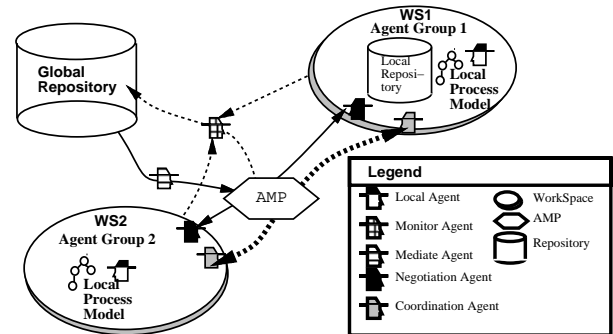


Figure 2. The CAGIS DIAS Architecture

A more detailed description of the multi-agent architecture can be found in [10, 9, 3].

4 IMPLEMENTING A MOBILE, MULTI-AGENT SYSTEM USING JAVASPACE

This section describes how we implemented a mobile, multi-agent system using the JavaSpaces platform. There were two main challenges when changed the underlying technology of our mobile MAS from Aglets to JavaSpaces. First, we had to change the way we implemented mobile agents to adopt to how JavaSpaces manages movement of Java objects. Second, we had to change our Agent Meeting Places to use JavaSpaces as the underlying mechanism for moving objects in a distributed environment.

4.1 MOBILE AGENTS IMPLEMENTED USING JAVASPACE

The JavaSpaces technology makes it possible to distribute virtual spaces over several machines in a network, and read, take and write object from/to these spaces. In addition, a notification service is provided to detect new objects appearing in a JavaSpaces. We used the four basic operations in the JavaSpaces framework (read, take, write and notify) to facilitate agents to move around in a network.

Java objects in a JavaSpace are inactive. To be activated, a Java object must be read or taken (see section 2) from the JavaSpace and then executed. We use a *Receiver Agent* to check if there are agents in the JavaSpace waiting to be executed. The Receiver Agent registers a *listener* in the

JavaSpace which detects incoming mobile agents (user or interaction agents) and *notifies* the agent. All mobile agents have some defined characteristic to make it possible for the listener to recognise them. When a listener has recognised that a new mobile agent have appeared in the JavaSpace (as a passive object), this agent is *taken* out of the JavaSpace. The Receiver Agent will then activate the mobile agent by invoking a *startAgent* method defined in the mobile agent code. The classes UserAgent and InteractionAgent have a super class called SuperAgent with properties like agent ID, agent Type, and destination ID. In addition, the SuperAgent class has the methods startAgent() and disposeAgent(). The Receiver Agent therefore register listeners for the Java objects with the class SuperAgent as a super class. It is also possible to use listeners that look for specific agent properties defined in an agent ontology.

4.2 AGENT MEETING PLACES IMPLEMENTED USING JAVASPACE

The Agent Meeting Places (AMPs) are responsible for enabling agents to exchange information and services, and make it possible for agents to move between AMPs. We have chosen to run one JavaSpace for each AMP, used to transfer agents from one AMP to another.

Agent migration between AMPs Agent migration between two AMPs (A and B) can be split in five steps like shown in *figure 3*. The agent that want to move (Agent A), must first notify a *Dispatcher Agent*. The Dispatcher Agent must be informed of the agent’s destination specified by the name of JavaSpace (unique) of the AMP. The Dispatcher Agent notifies (1) AMP A that Agent A is leaving, and will *write* (2) the agent to the specified JavaSpace (AMP B) as a Java object with preserved state. A Receiver Agent on AMP B, will *listen* (3) for incoming mobile agents in the JavaSpace. Agent A will be *taken* (4) from the JavaSpace and invoked in the AMP B. Finally, the Receiver Agent will inform AMP B that Agent A is *present* (5). In addition to the five steps above, the current location of the moved agent will be registered in the AMP where the agent first time was initiated (origin AMP). An agent is identified with an agent ID and origin AMP ID. In this way, it is always possible to locate an agent’s whereabouts if the agent ID is known.

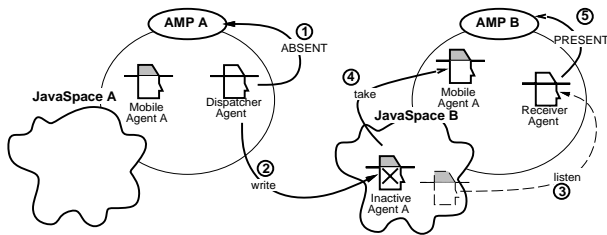


Figure 3. Moving Agent A from AMP A to AMP B

Communication among agents Communication among agents is managed by system agents called *Communication*

Agents. If the two agents are situated at the same AMP, the message is exchanged directly. If the agents are situated at different AMPs, the message will be exchanged according to the five steps shown in *figure 4*. The figure shows how Mobile Agent A in AMP A can send a message to Mobile Agent B in AMP B. First, Mobile Agent A *deliver* (1) the message to the Communication Agent in AMP A. The Communication Agent need to know the agent ID of the receiver. From the receiver’s ID, the Communication agent can locate Agent B (as described in last paragraph). Next, the Communication Agent *write* (2) the message as a *message object* into the JavaSpace belonging to the AMP where the receiver is located (here AMP B). The Communication Agent at the receiver side, will use listeners to *listen* (3) for incoming message objects. As soon as the message object arrives in the JavaSpace, the Communication Agent *takes* (4) it out of the JavaSpace. The Communication Agent will then look for the receivers ID and *deliver* (5) the message to Mobile Agent B. If Mobile Agent B has left AMP B before the message has been delivered, the Communication Agent has to locate Mobile Agent B’s new location and the steps 2-5 are repeated. This is not a big problem, since the mobile agents usually stay at one AMP for some time before moving to another one.

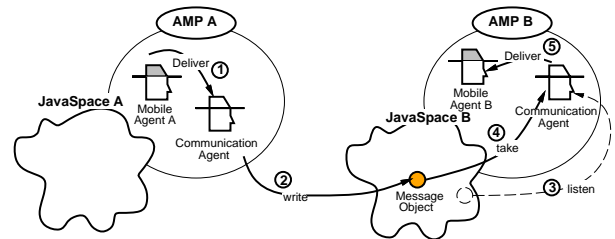


Figure 4. Message exchange between agents at different AMPs

Other system agents at AMPs There are also other system agents running at AMPs that have not been mentioned above. A *Manager Agent* is responsible for creating and destroying AMPs, for exchanging agent information between AMPs, and for initiating other system agents. *Registration agents* are responsible for sending information about an agent’s location to its origin AMP. This information is received and managed by a *Localisation Agent*. In addition there are other optional system agents to provide services for monitoring agent activity, managing repositories etc.

5 EXPERIENCES FROM USING THE JAVASPACE IMPLEMENTATION OF THE MAS

To test the JavaSpace implementation of our mobile, multi-agent platform, we implemented parts of a conference organising scenario [7, 1]. In this scenario we have focused on how to provide distributed cooperative support for organising conferences where the program committee mem-

bers participate in decisions using mobile agents. Mobile agents are therefore used for allocating reviewers to papers, by letting the reviewers select the papers they are interested in. In case of conflicts between reviewers, the agents will follow a negotiation strategy to make sure that all papers have allocated three reviewers. In the same way we have also implemented similar support for group papers into sessions, and assign timeslots for sessions.

We discovered two different ways to implement a mobile agent allowing the reviewers to select the papers they wanted to review. The first solution was to implement a mobile paper selection agent that moved itself from one user to another. This means that the selection agent operates in a sequential manner, and only one user can select papers at a time. When moving around from one user to another, the selection agent can warn the user if her/his selection has been selected by too many already based on knowledge from earlier selections. The other solution was to implement a mobile paper selection agents that work in parallel. This means that all users select papers they want to review at the same time. In the JavaSpaces framework, the operation *read* is used to move multiple copies of a Java object from a JavaSpace to an application. This approach was used to implement the parallel selection agents. If the operation *take* is used in a JavaSpace, only one instance of the Java object will be moved from the JavaSpace to the application. This is because the Java object will be removed when taken. We used this approach to implement the sequential selection agent. A screenshot of a mobile paper selection agent is shown in figure 5. To implement the allocation process of the paper (including negotiation between reviewers), both the serial and the parallel approaches were used.

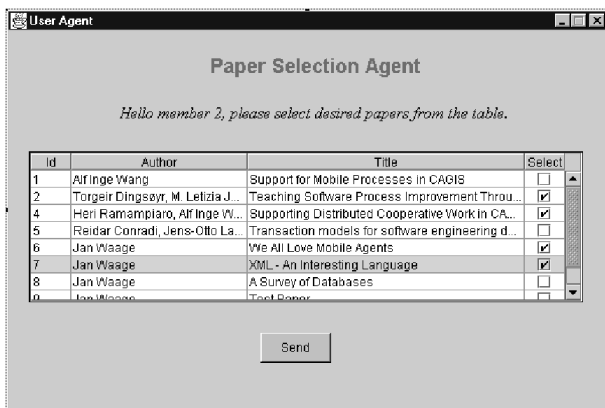


Figure 5. Screenshot from a mobile Paper Selection Agent

The code for the mobile agents to support the scenario above was simple. It consisted of a part to implement the GUI (used the Swing library), a part to define how the agent should move (e.g. a sequential route or in parallel etc.), and a part containing the logic (control) of the agent.

When the implementation of agents to support the sce-

nario was finished, we ran the scenario on different machines in a network. The mobile agents travelled around prompting different users for input through a GUI, and the papers, sessions and timeslots were allocated according to the defined negotiation strategy. From running the scenario we discovered advantages and disadvantageous compared to our old Aglets implementation. First, the configuration of the JavaSpace based agent system was a lot simpler and could be done dynamically, since JavaSpaces automatically detects new AMPs were added to the network. Also when AMPs crashed or lost network connection, this was detected and managed by the agent system. JavaSpaces is based on JINI, and JINI provides services for dynamically detecting network resources. However, to run several JavaSpaces on the same machine was very demanding on computer resources (both CPU and memory) compared to Aglets. In practise, this means that we have to spread AMPs out on different machines instead of running several on one. In the documentation of JINI, it is not written explicitly over what distances JINI can detect network resources. We discovered that from our machines running in Norway, that JINI was able to detect machines running JavaSpaces in Germany and in the USA. To ensure connection over long distances, we have implemented a special AMPs that are used to exchange agent information, agents and Java objects between sites distributed in WANs. In Aglets, we did not have the network distance problem because all connections between Aglets servers had to be explicitly specified.

Our experiments with JavaSpaces discovered one problem with implementation of mobile agents. In the prototype we included some JComponents from the javax.swing package. Our experiments showed that there is a problem with serialisation of JComponents. This problem can be solved by replacing JComponents with ordinary Java components.

6 RELATED WORK

In [5], McKim and Pomerleau presents a JINI technology based mobile agent framework that uses JavaSpaces to support agent coordination and management. This framework also defines agent places where agents are executed. The main difference between this approach and our approach is that they mainly use JavaSpaces as a general facility for storing agents, while in our approach every agent meeting place (AMP) has its own JavaSpaces for coordination of agents. McKim and Pomerleau also state that their framework is not expandable to the entire Internet due to JINI technology discovery protocols. Our framework has overcome this problem as described in previous section.

In [2], Cabri et. al describes the Mobile Agent Reactive Spaces (MARS) coordination architecture, based on the definition of Linda-like tuple spaces. These tuple spaces are exploited by agents to coordinate themselves with other application agents and to access local resource. The tuple spaces acts similar to blackboards where applications

can share data and coordinate their actions. A MARS tuple space is realized as a Java object, which implements the MARS interface. This MARS interface is an extension of the JavaSpace interface with two additional operations defined: readAll (retrieve all matching tuples from space) and takeAll (to extract all matching tuples from the space). The main difference between the MARS architecture and our approach is that the MARS architecture provides an infrastructure for coordinating agents in general, while our approach coordinates agents in a specific way (in meeting places).

7 CONCLUSION

The paper we have presented how we migrated our mobile agent system from Aglets to JavaSpaces. In our first agent system implementation (with Aglets), two last year students spent about four months to finish the implementation based on some fixed requirements. The second implementation (using JavaSpaces) was also executed by two last year students, and they used three and a half months based on the same requirements. The students spent about the same time on studying the technologies before starting on the implementation (two months). Both implementation featured the same functionality, but there was one significant difference. It was harder to configure and initiate the Aglets version of the agent system, because the underlying JINI technology in JavaSpace automatically took care of detecting running JavaSpaces. In the Aglets implementation, the detection of Aglets servers had to be hard-coded using URLs or similar. Our experiences show that JavaSpaces can well be used to provide the same functionality as that provided in other mobile agent frameworks like, e.g. Aglets. It is necessary to make some adaptations in order to use JavaSpaces as a basis for mobile agents. These adaptations are quite simple to make and does not require much extra work. In addition, the API in JavaSpaces is very high-level, making it efficient to provide distributed services. The Aglets framework in comparison provides a low level API, requiring the programmer to also configure the distribution of Aglets servers. For static systems this is not a considerable amount of work. However, if you want to develop a dynamic system that can handle a changing environment, a lot of effort must be spent on changing the system to adapt changes. By using JavaSpaces, JINI deal with dynamic changes of the environment providing the system developer to consider the important parts of the system.

By using the JavaSpaces framework, the agent architecture also gets a mechanism for sharing objects for free as described in the last section. Future work will explore the

possibilities of sharing objects will give.

ACKNOWLEDGEMENT

We want to thank to Terje Salvesen and Jan Waage for implementing the mobile agent system using JavaSpaces, and to Maria Letizia Jaccheri, Heri Ramampiaro and Reidar Conradi for giving useful comments. The CAGIS project is sponsored by the Norwegian Research Council's Distributed Information Systems (DITS) Programme.

REFERENCES

- [1] Steinar Carlsen. *Conceptual Modeling and Composition of Flexible Workflow Models*. PhD thesis, Dept. of Computer and Information Science, Norwegian University of Science and Technology, Trondheim, Norway, December 15 1997.
- [2] Franco Zambonelli Giacomo Cabri, Letizia Leonardi. MARS: A Programmable Coordination Architecture for Mobile Agents. *IEEE Internet Computing*, 4(4):26–35, July/August 2000.
- [3] Anders Aas Hanssen and Bård Smidsrød Nymoen. DIAS II - Distributed Intelligent Agent System II, January 2000. EPOS TR 372 (diploma thesis), 324 p., Dept. of Computer and Information Science.
- [4] Danny Lange and Mitsuru Oshima. *Programming and deploying Java mobile agents with Aglets*. Addison-Wesley, 1998.
- [5] John McKim and Steven Pomerleau. Code Mobility and Dynamic Composition Using Jini Technology. web: <http://servlet.java.sun.com/javaone/javaone99/pdfs/e622.pdf>, June 17 1999.
- [6] Sun Microsystems. JavaSpaces TM Specification. White paper, Sun Microsystems, January 25 1999. Available on web: <http://www.sun.com/jini/specs/js.pdf>.
- [7] T. W. Olle, H.G. Sol, and A. A. Verrijn-Stuart. Information Systems Design Methodologies: A Comparative Review. North-Holland, 1982.
- [8] Mitsuru Oshima, Guenther Karjoth, and Kouichi Ono. Aglets Specification 1.1 Draft. Technical report, IBM Tokyo Research Laboratory, September 8 1998. Available on web: <http://www.trl.ibm.co.jp/aglets/spec11.html>.
- [9] Geir Prestegård, Anders Aas Hanssen, Snorre Brandstadmoen, and Bård Smidsrød Nymoen. DIAS - Distributed Intelligent Agent System, April 1999. EPOS TR 359 (pre-diploma project thesis), 396 p. + CD, Dept. of Computer and Information Science, NTNU, Trondheim.
- [10] Alf Inge Wang, Chunnian Liu, and Reidar Conradi. A Multi-Agent Architecture for Cooperative Software Engineering. In *Proc. of The Eleventh International Conference on Software Engineering and Knowledge Engineering (SEKE'99)*, pages 1–22, Kaiserslautern, Germany, 17-19 June 1999.