Experience paper: Implementing a Multi-Agent Architecture for Cooperative Software Engineering

Alf Inge Wang*

January 17, 2000

Abstract

The paper describes experiences we have earned from implementing a multi-agent architecture used to support cooperative software engineering. Before starting to implement a multi-agent architecture, important decisions and considerations must be taken into account. You have to decide how to provide efficient inter-agent communication support, what language should the agents talk, should the agents be stationary or mobile, and what technology should be used to build the architecture. This paper describes how we implemented our multi-agent system, and the experiences we gained from building it.

Keywords: *Cooperative Software Engineering, Agents, Multi-agent system, KQML, XML, Aglets, JATLite, CORBA.*

1 Introduction

The last couple of years, distributed computing and agent technology have become more and more popular. When researchers are developing prototypes, the choice of technologies and how to use different technologies is getting more and more complicated. This paper describes experiences from designing and implementing a Multi-Agent System (MAS) that provides support for Cooperative Software Engineering (CSE). CSE is a research area where focus is on how to support participants in cooperative processes working in distributed organisations. In our MAS, agents are used to provide support for cooperative processes as coordination and negotiation. The agents communicate through a defined language, and the MAS offers the infrastructure for the agents to talk to each other as well as to users.

*Dept. of Computer and Information Science, Norwegian University of Science and Technology (NTNU), N-7491 Trondheim, Norway. Phone: +47 73 594485, Fax: +47 73 594466, Email: alfw@idi.ntnu.no

Before we started to implement a multi-agent system, a theoretical study on distributed architectures and agent technologies was conducted by Joar Øyen, a diploma student at the Department of Computer Science at the Norwegian University of Science and Technology. The outcome of this study was a report [18] that gave us guidelines for building a multi-agent architecture. We have used these guidelines as a starting point for our implementation.

We have been searching for related work that focuses on experiences from implementing multi-agent system using different types of technology. It seams like there is nothing or at least very little published covering these issues. There are a lot of publications on high-level descriptions of multi-agent architectures, such as [3, 9, 2], but details about technology used or experiences from implementing these systems are left out.

We have used the multi-agent paradigm to implement our system, but there are alternative approaches. JavaSpaces [12] based on Java RMI and JINI, recently introduced by SUN, provide an alternative approach for exchanging distributed objects. JavaSpaces technology is 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 you can invoke on a JavaSpace:

1. **write** Write an entry into a JavaSpace
2. **read** Read an entry from a JavaSpace that matches some specified parameters
3. **take** Read an entry from a JavaSpace that matches some specified parameters, and remove it from this space
4. **notify** 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 as multi-agent systems as 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.

The rest of the paper is organised as following. Section 2 briefly describes the architecture we wanted to build and what components the architecture consists of. A scenario is also presented to show how the prototype can be used. Section 3 outlines the requirements to the technology to be used to implement our architecture. Section 4 describes technological guidelines to be followed when implementing a multi-agent architecture. Section 5, describes experiences we have achieved from implementing a multi-agent architecture in two versions DIAS I and DIAS II. Section 6 concludes the paper.

2 CAGIS Multi-Agent Architecture for Cooperative Software Engineering

This section is a short introduction to the CAGIS[1] Multi-Agent Architecture for Cooperative Software Engineering. This is an architecture that uses agents to represent cooperative participants in a cooperative effort and supports coordination, negotiation and communication of agents. A more detailed description of this architecture can be found in [16].

2.1 CAGIS Multi-Agent Architecture components

Software agents are useful for supporting cooperative activities, since software agents can act as human agents on behalf of humans. Our architecture uses this property of software agents to model data- and control flow, which can be used to implement a self-optimising or self-improving process. The main components in this architecture are agents, workspaces, Agent Meeting Place (AMP) and repositories:

- **Agent** A piece of autonomous software created by and acting on behalf of a user. The agent is set up to achieve a modest goal, with the characteristics of autonomy, interaction, reactivity to the environment, as well as pro-activeness. Agents are grouped into three main groups. The first group, *System agents*, are used to execute administrative tasks of the multi-agent architecture. This can be to monitor human and software agent activity, deal with repositories and managing AMPs (see the third point). The second group, *Local agents*, assist users in work within local workspaces. The last and most important agent group, *Interaction agents*, help users in their cooperative work (coordination, negotiation, communication). Note that mediation agents are also provided to suggest solutions for locked negotiation processes based on prior experiences.
- **Workspace (WS)** A temporary container for relevant data in a suitable format, together with the processing tools. Workspaces can be private as well as shared.
- **Agent Meeting Place (AMP)** AMPs are where software agents meet and interact. AMPs are built on underlying communication mechanisms, but provide agents with more intelligent means to facilitate their interaction. An AMP can also work as a market place where agents can "trade" information and services. The main purpose of the AMP is to facilitate cooperative support for software agents.
- **Repository** A persistent storage of data that can be local, global or distributed. Repositories can be accessed either by tools or by agents. One specific repository is very important in the CAGIS multi-agent architecture; the experience base. An experience base can be used as the community memory, and a mediator agent can utilise prior stored experiences to solve negotiation conflicts.

[1] CAGIS (Cooperative Agents in Global Information space) is the name of a Norwegian research project (1998-2000) with main focus on software support for distributed cooperation for human problem solvers. More information about the CAGIS project can be found at:http://www.idi.ntnu.no/∼cagis.

Next subsection will give an example of how the CAGIS multi-agent architecture can be used.

2.2 Example of a multi-agent architecture

Figure 1 shows how the CAGIS multi-agent architecture can be used in a simplified software development scenario. This example describes a coordinated software development process involving multiple departments of an organisation. Each department is represented by a workspace[2], and an AMP is used as a place where the departments can cooperate through software agents.

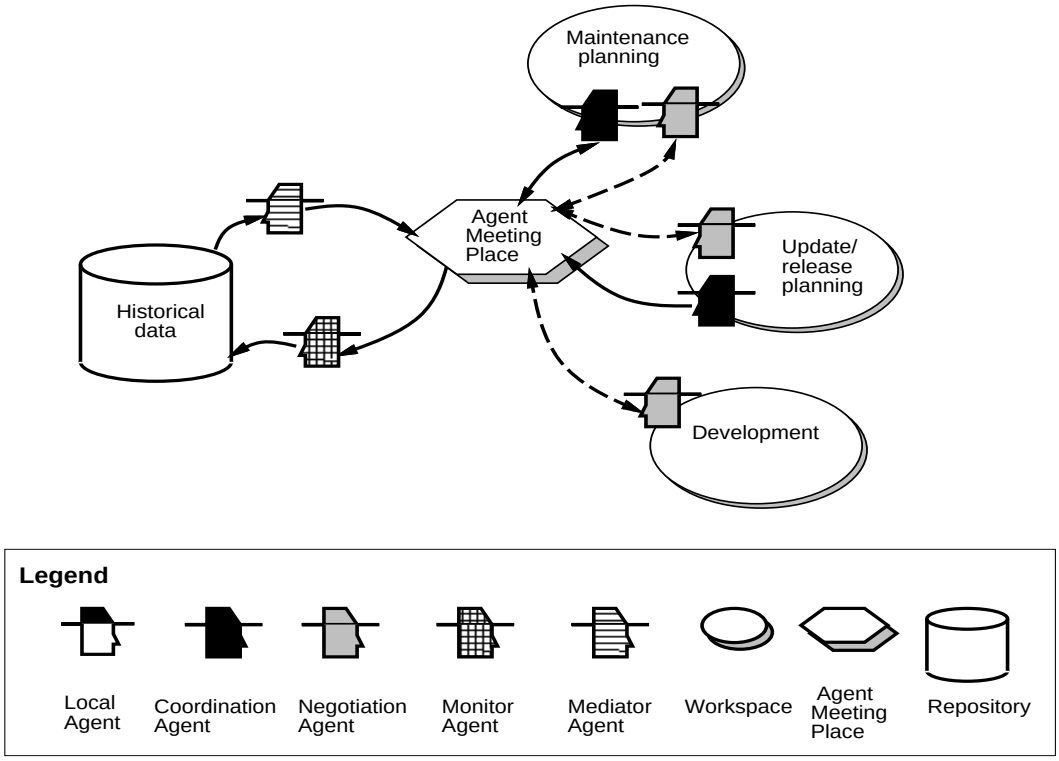

Figure 1: An example of an agent architecture application

This scenario demonstrates how it is possible to support a situation where limited human resources in the development department cause trouble for a *Maintenance planning department* and an *Update/release planning department*. Both departments are competing for human resources in the *Development department*, because the two departments want to serve request and complaints from customers, and to improve the software product respectively. If a negotiation process between for instance *Maintenance planning* and *Update/release planning* takes to much time, the mediator agent (shown in figure 1) will break into the negotiation process to make an agreement. The mediator agent can base its judgement on experiences from prior projects (e.g., we lost a lost the biggest customer because of too many bugs in the product last year), on company guidelines or through interaction with the company's management.

[2]Workspaces are used in this architecture to group people. It is however possible to have personal workspaces.

In the suggested solution, workspaces are places where humans and agents communicate and share information and tools. AMPs are also places, but they are indented to be virtual markets where agents meet and interact to facilitate cooperative support for applications and users. In addition, repositories are special places where information and knowledge can be permanently stored and later retrieved.

3 Requirements to the technology to be used

This section will outline the implementation requirements for the multi-agent architecture described in section 2. These are the requirements for a prototype of the system and not requirements for a full-fledged system. The requirements given in this section should be quite general, and should also be applicable to most multi-agent systems supporting mobile agents.

3.1 General requirements

The overall goal of the proposed CAGIS multi-agent architecture was to be flexible and tailorable to many different needs. The main requirement is therefore an open architecture that can evolve together with the real world it is supposed to support.

Since a prototype of the architecture will be built in a short period of time, it is important that the cost of the implementation is kept at a minimum, that is free and proven technologies should be used whenever appropriate. This means that the solution must be feasible today.

Performance is not important for this prototype architecture.

3.2 System infrastructure

The organisations that the CAGIS multi-agent systems are intended to support, are inherently distributed, a heterogeneous environment, and are continually changing. The infrastructure that is going to be the foundation of the system must thus define an open, flexible, and malleable environment, which encompasses hardware platforms, operating systems and networks.

A run-time environment to implement workspaces and AMPs must be provided by the infrastructure. Such an environment must provide a number of services to its inhabitants. Name registration and service advertising allows agents to be aware about each other's properties. In addition, event mechanisms will allow monitoring agents to register their interest in specific events.

3.3 Agent implementation and configuration

Again, an open multi-platform solution is required. This includes the ability to ship binary executables between different platforms, so that the agents can be mobile.

Agents are going to be built by experts, but it must be possible to tailor agents to the specific needs of the different users. Some sort of agent scripting or agent configuration is therefore required to let users fine-tune their system.

3.4 Agent communication

To be able for agents to communicate, we need a format to represent information, as well as some conversation policies on how agents communicate. The architecture also requires facilities for users to communicate with agents, and agents access to external entities like repositories, workspaces, tools, AMPs etc..

The basic communication mechanisms (streams, messages, events, etc.) are provided by the underlying system infrastructure, and the communication mechanisms uses these services to provide more high-level facilities for inter-agent communication. *First*, agents must be able to communicate with each other in a language that all involved parties understand (not only syntax of the language, but also semantics and pragmatics of the conversation). *Second*, agents must collaborate with each other to reach common goals. This collaboration is controlled by process models that agents must interpret. *Third*, agents also need mechanisms for coordination and negotiation to handle many common situations. Coordination mechanisms can be used to control the concurrency between multiple agents and to exchange instructions that tell agents what to do. The system should provide a negotiation model with a defined context and multiple negotiation strategies. Agents are thus given the possibility to find out what is negotiated for and to select an appropriate strategy to use.

3.5 Knowledge sharing

Communication mechanisms like coordination and negotiation operate in a cooperative context and do therefore need to share common goals. The common goals represent some form of group awareness or community memory, and are contained in a work context that consists of information resources and control structures.

Common information can be stored in three different places in the architecture:

1. **In repositories** The main part of knowledge is contained in repositories.
2. **In AMPs** Meta-information about agents are contained in the AMPs, but the actual information is contained inside the agents themselves.
3. **In agents** Agents may be used as information carriers of process information and results.

The most important common requirement is however that information is stored in a formally defined format that makes it accessible by agents.

3.6 Design proposal for the prototype

Figure 2 shows an overall design proposal for how the prototype should be structured. We have used a multi-tier architecture based on the agent, places, and things paradigm. The lower part of the figure (component infrastructure and agent infrastructure) will define a foundation, based on available standard implementations, which will provide functionality and services to the prototype of the multi-agent architecture. In the shown architecture, the facilitators are central components both in workspace and AMP. Facilitators simplify the implementation of agent communication and the coordinator, moderator, and monitor-components. The reason for this is that the interaction between various entities can be controlled from one central point. The drawback with this solution is of course that the facilitators might become bottlenecks of the system.

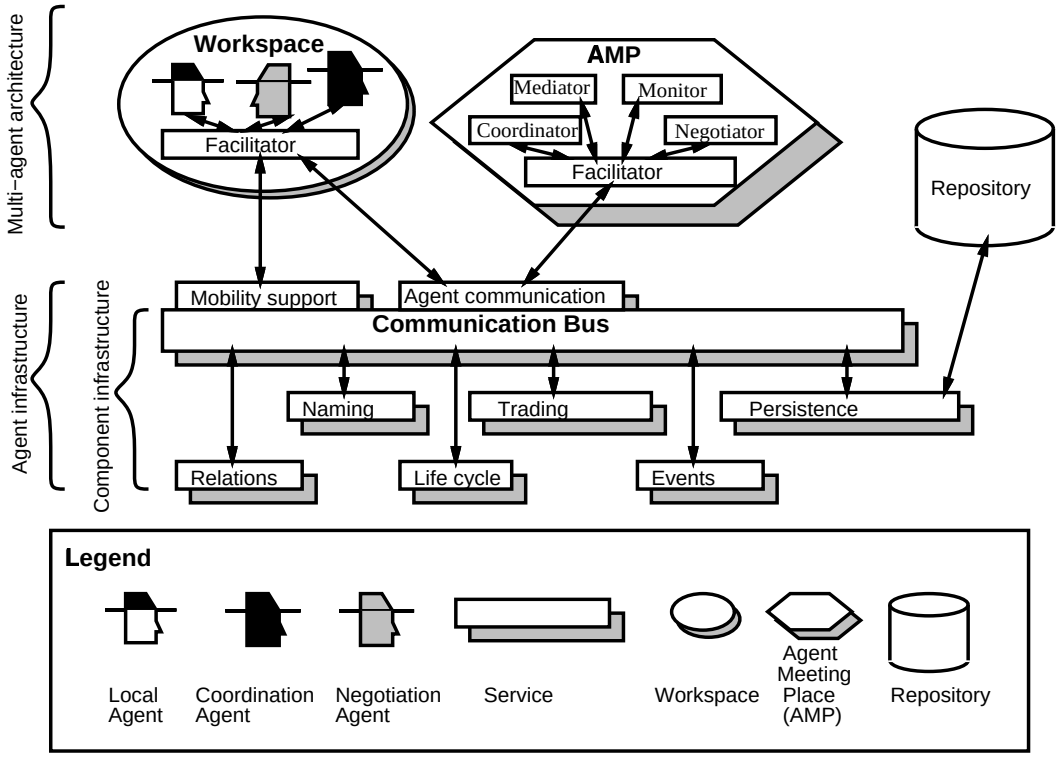

Figure 2: Design architecture

4 Technological guidelines

This section will give advice on what technology to use to implement a prototype of the multi-agent system proposed in section 2 according to the requirements to the technology used described in section 3.

1. **Use Java and Java IDL [13] as the component infrastructure** because (1) Java provides code portability, garbage collection, object-orientation, (2) Java is a broadly accepted standard, (3) Free implementation is available now, (4) Many agent-related technologies are closely associated to Java.

2. **Use the facilitators in the design architecture to implement the trader, event, lifecycle, and relationship services as needed**. By choosing Java IDL for the component infrastructure, we have to use this CORBA implementation for the rest of the system as well. This gives us a major drawback, because Java IDL only provides the naming service of the CORBA services shown in figure 2. Since the design shown in figure 2 has several services that are not supported in Java IDL, these services must be developed in Java if required. These services can be replaces, if Java IDL will offer them in the future.

3. **Use Aglets to implement the mobility support for the agents**. Mobility support should ideally be implemented by OMG's Mobile Agent Facility, but due to the fact that this standard is under construction, Aglets will be recommended instead [3]. Aglets [10, 14, 11] are Java objects, developed at IBM's Research Laboratory that can move from one host on a computer network to another. Aglets servers provide distribution of aglets to aglets viewers. The viewers are the users graphical interface to the aglets. From this interface the users can create, activate, dispatch, and retract agents.

   Using Aglets should provide an easy transition to the Mobile Agent Facility later, because Aglets are one technology that is used as a basis for the Mobile Agent Facility. Another advantage with the Aglets-technology is that user-interfaces for controlling the operation of the agents are provided as a part of the technology.

4. **Use KQML and JATLite for agent communication and as a foundation for implementing workspaces and AMP**. The Java Agent Template Lite (JATLite)[8] is a technology that provides a Java implementation that lets agents communicate with each other, possibly by using Knowledge Query and Manipulation Language (KQML). We recommend using KQML [4] as the agent communication language, because it is an extensible standard that has many of the advantageous features that an agent communication language should have. JATLite is also a technology that can be used to implement workspaces and AMPs, because each JATLite-router defines an environment that facilitate the administration of and communication between a group of related agents.

5. **Use XML to represent information and work-products in the architecture**. eXtensible Markup Language (XML)[7] does not put any restrictions on the format of the information is shall represent. Also tools to support XML are available in Java.

How the recommended technologies are used to implement the various parts of the design architecture is illustrated in 3.

[3] At the time these technological guidelines was worked out, OMG's standard for Mobile Agent Facility was not finished, and there was not any implementation of OMG's Mobile Agent Facility.

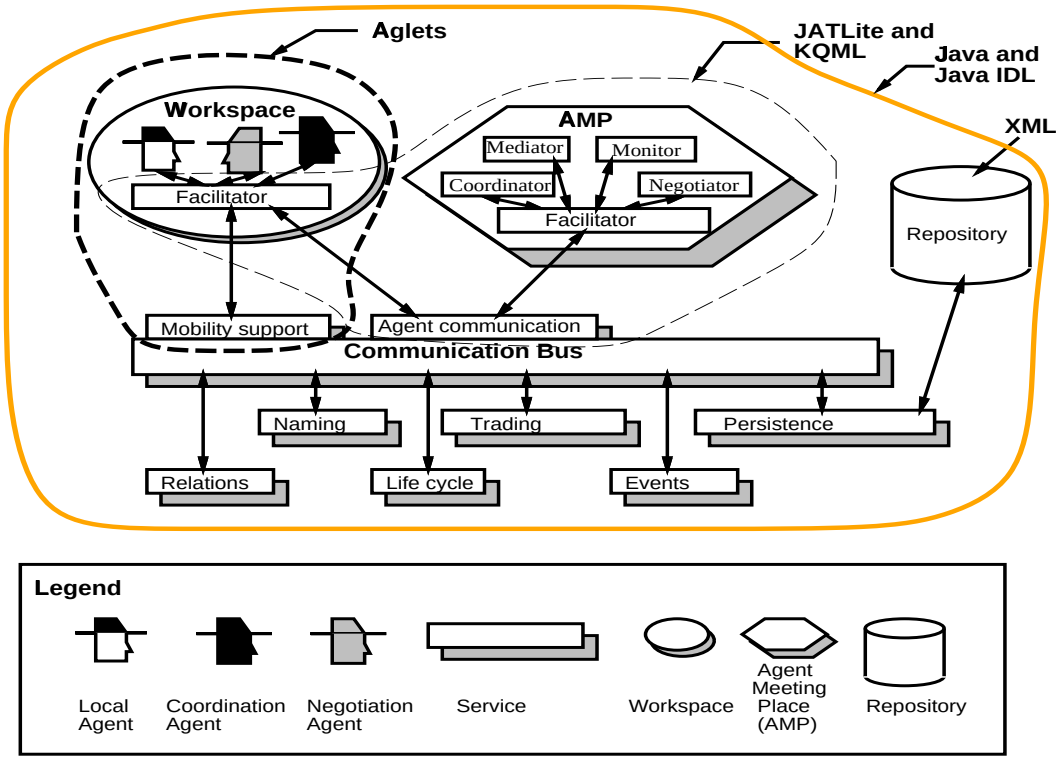

Figure 3: Implementation of the design architecture

It must be noted that figure 3 only loosely denoted where the various technologies should be used, and more experience is needed to decide exactly which of the technologies that are best in the specific situations.

5 Experiences from implementing a MAS based on the guidelines

The implementation of our multi-agent architecture is named Distributed Intelligent Agent System (DIAS). We have implemented two versions DIAS namely DIAS I and DIAS II. Both implementations were developed by last year students at the Dept. of Computer and Information Science at the Norwegian University of Science and Technology. This section describes the work they have done and experiences from this work.

5.1 DIAS I

Spring 1999, four students used about 1000 hours to make the first version of the implementation of our multi-agent architecture[15]. These students were skilled in Java-programming and XML. However, the area of programming agents and implementing a multi-agent system was totally new to them. DIAS I used the requirements to technology (described in section 3) and technological guidelines (described in section 4) as a starting-point for the prototype.

5.1.1 Choice of technology

We discovered that not everything from section 3 and 4 could be used directly when implementing the system. The first problem we encountered was how to combine JATLite and Aglets into one implementation. JATLite was well suited to implement facilitators for workspaces, AMPs and generally for agent communication. Unfortunately, JATLite does not support mobile agents. Since we wanted to have support for mobile agents, we needed to find a way of making JATLite mobile. The Aglets framework was suggested to be used to support mobility, but then we had to integrate JATLite with Aglets. We found that this was not an easy task. We chose to use Aglets as the main implementing standard for agents and agent communication, and just to use parts of the KQML layer in JATLite. This solution makes it possible to use mobile agents in all parts of the architecture. It was possible to use parts of JATLite because JATLite is open-source. The parts missing in Aglets compared to JATLite was quite easy to implement using the functionality found in Aglets. Another reason for using Aglets was that the documentation was better than JATLite.

The requirements to technology and technology guidelines suggested that we should use CORBA (Java IDL) as the communication bus in our architecture. When we worked with the first DIAS-project, the mobility support over the ORB was not possible because the standardisation Mobile Agent System Interoperability Facility (MASIF) was not approved yet by the OMG. Since we wanted to keep the mobility support for agents, Aglets was chosen for communication over TCP/IP. Aglets was chosen because Aglets was one of the agent languages that MASIF is based on. This should make the transition to MASIF easier on a later stage.

KQML was recommended as communication language for our architecture. Since KQML is the most used standard for agent communication, we chose to use KQML for our implementation. Parts of the KQML layer in JATLite were used to give the architecture KQML support. Since the content in a KQML message can be whatever wanted, the designer is free to choose the representation format of the content of a KQML message. XML is a good choice when the knowledge is going to be stored in a web-server or in a file-system. It is also possible to use another format for knowledge representation, but for our prototype we chose XML. XML has also become a well know standard for information representation.

5.1.2 Experiences of use

Although the first version of DIAS was not very advanced, the architecture was sufficient to demonstrate parts of the scenario described in section 2.2. Negotiation agents for the groups *Maintenance planning* and *Update/release planning* used to negotiate about limited human resources were implemented. The experiment showed that such negotiation processes were supported in our architecture through negotiation agents and an Agent Meeting Place. The agents were able to move between workspaces and the AMP, and they were able to communicate using KQML as a communication language. The following KQML performatives were using in our agent-language (these performatives were sufficient at least to demonstrate our scenario):

- **ask-if** Sender wants to know if the sentence is in the Virtual Knowledge Base (VKB) of the receiver.
- **insert** The sender asks the receiver to add content to its VKB.
- **register** The sender can deliver performatives to some named agents.
- **tell** The sentence is in the VKG of the sender.
- **unregister** A deny of a register performative.
- **untell** The sentence is not in the VKG of the sender.

XML was used to represent the contents of KQML-message.

We found some shortcomings with the DIAS I implementation. **First**, it was rather hard to implement new agents. The reason for this was that in order to implement your own agent, you had to know how the Aglets framework works. The DIAS I implementation did not manage to provide a high-level agent-API, so the agent-programmer did not need not be concerned about the low-level technological issues in Aglets. Because of this, it was very hard to use the architecture to create support for various scenarios. **Second**, the implementation was lacking of high-level support for inter-agent communication and inter-AMP communication. The latter meant that to support several AMPs, the programmer of the agents had to hard-code how to deal with the different AMPs and agents. This was not desirable. **Third**, the DIAS I implementation did not provide support for integrating the multi-agent system with other systems.

Because of the shortcomings described above, we decided to continue with a DIAS II project (see next section)

5.2 DIAS II

After finishing implementing DIAS I, two students continued the DIAS project as diploma thesis's, and spent about 1000 hours to extend the original implementation to DIAS II [6]. The DIAS II project focused on making the original DIAS implementation better. The following problems were addressed: agent security, integration with other systems through CORBA, and making a higher-level agent API.

5.2.1 Choice of technology

In the first version of the DIAS implementation (developed in Java JDK 1.1), agent security was not addressed at all. In JDK 1.2, the enhanced security model for fine-grained resource access has been added. Because of this, JDK 1.2 would be a natural choice for DIAS II. Unfortunately, the Aglet Software Development Kit, ASDK 1.1 does not support JDK 1.2. This meant that we had to abandon the new security features in JDK 1.2 if we would like to keep Aglets to support mobility. By comparison, we found that ASDK actually had almost the same security features as in JDK 1.2 included, so we chose to use JDK 1.1.

Since we wanted to add possibility to integrate our multi-agent system with other systems, CORBA support was needed for our architecture. One way of giving our architecture CORBA support was to change Aglets with other mobile agent implementations supporting CORBA. Both Odyssey [17] and Voyager [5] supports CORBA had provide interesting functionality for distributed systems and interoperability between different communication facilities. Another alternative was to continue using Aglets and to implement CORBA support into it. We chose to do this, because it required less work and the Aglets implementation have better support for security. OrbixWeb [1] was used as a CORBA implementation, because of its functionality and availability at the University. External systems can now communicate with our multi-agent systems in the Agent Meeting Places with CORBA support. The CORBA support is implemented according to OMG's Mobile Agent System Interoperability Facility (MASIF). MASIF is a standard to make it possible for interoperability between various multi agent systems, and have four areas that are standardised: Agent management, Agent transfer, Agent and agent system names, and Agent system type and location syntax. The rest of the implementation of DIAS II uses the same technology as in DIAS I.

5.2.2 Experiences of use

The agent-API for DIAS II is totally different compared to the first version of DIAS. For the first version, you had to write low-level Aglets-code to make an agent. In DIAS II, you don't have to know that the architectures use Aglets technology at all. No knowledge of Aglets is needed, and the following areas are covered with high-level methods in the agent API:

- **Create/kill agent** These methods are called when creating new agents or when you want to kill an agent. Agents will be moved back to where they were created before it will be killed. Note that agents can also be cloned.
- **Message handling** Methods for sending/receiving KQML messages to/from other agents. The architecture will take care of sending messages to the correct receiver. The sender will always receive a acknowledgement when the receiver has received the message.
- **Register/unregister agents** These methods are used to register/unregister agents in AMPs.
- **Move agents** Methods for moving an agent to another AMP. If an agent cannot find what he is looking for in an AMP, the AMP can suggest the agent to move to another AMP.
- **Information queries** Various methods offer the agents the possibility to ask AMPs for information about what agents are connected, what type of ontology is used, what properties have other agents etc.

Another major change of the architecture is how agents communicate, and how AMPs can communicate. In DIAS I you had to explicitly state how the communication between

agents should be performed. In DIAS II, the system takes care of looking for agents, using advanced communication agents. Agents are located according to agent ID number, AMPs ID number or/and ontology of AMPs. If for instance neither the agent ID number or AMPs ID number is know, the ontology is used to find a matching agent.

6 Conclusion

Implementing a multi-agent system is not an easy task. The technologies available, makes it easier to build robust systems in rather short time. It is however important to know what technologies are available and what to choose before starting building a multi-agent system. It is also important to know what technologies are possible to combine, before starting on the work. Our experience with building a multi-agent system is that you should use the standards available when possible. Using OMG's MASIF standard will make it possible for a system to communicate with other multi-agent systems as well to other applications through CORBA. KQML is the most widely used agent communication language used, and makes it possible for agents on different systems to communicate. XML offers a convenient information wrapping that is useful for different purposes in an multi-agent system, as information/knowledge representation, small repositories etc. Using Java as the programming language, makes it possible for the system to run in an heterogeneous environment, and most agent standard implementations as well as XML tools are implemented in Java as well.

We are now using our CAGIS multi-agent architecture for cooperative software engineering to make support for various scenarios. In doing this, we want to see how general our architecture is and recognise the shortcomings of our architecture. We are also considering technology as for instance JavaSpaces to implement DIAS III.

7 Acknowledgement

We will give a big thank to Joar Øyen who has provided us with in-depth information about distributed technology in theory and in practise. We would also like to thank Geir Prestegård, Snorre Brandstadmoen, Anders Aas Hanssen, and Bård Smidsrød Nymoen for implementing DIAS I. Anders Aas Hanssen and Bård Smidsrød Nymoen were also responsible for the DIAS II implementation.

References

[1] Sean Baker. *CORBA Distributed Objects - Using Orbix*. ACM press and Addision-Wesley, 1997. ISBN 0-201-92475-7.

[2] Frances Brazier, Pascal van Eck, and Jan Treur. Modelling Competitive Co-operation of Agents in a Compositional Multi-Agent Framework. *International*

Journal of Cooperative Information Systems, 6:67–94, 1997. Special Issue on Formal Methods in Cooperative Information Systems: Multi-Agent Systems.

[3] Frances M.T. Brazier, Catholijn M. Jonker, Frederik Jan Jurgen, and Jan Treur. Distributed Scehduling to Support a Call Centre: a Co-operative Multi-Agent Approach. *Applied Artificial Intelligence Journal*, 13:65–90, 1999. Special Issue on Multi-Agent Systems.

[4] Tim Finin, Yannis Labrou, and James Mayfield. *Software Agents*, chapter KQML as an agent communication language. MIT Press, Cambridge, 1997. Ed. Jeff Bradshaw.

[5] Graham Glass. Objectspace, overview of voyager: Objectspace's product family for state-of-the-art distributed computing. White paper, ObjectSpace, 1999. Availeble on web: http://www.objectspace.com/products/documentation/VoyagerOverview.pdf.

[6] Anders Aas Hanssen and Bård Smidsrød Nymoen. DIAS II - Distributed Intelligent Agent System II. Technical report, Norwegian University of Science and Technology (NTNU), Janurary 2000. Technical Report, Dept. of Computer and Information Science.

[7] Steven Holzner. *XML Complete*. McGraw-Hill, 1998. ISBN 0-07-913702-4.

[8] Heecheol Jeon, Charles Petrie, and Mark R. Cutkosky. JATLite: A Java Agent Infrastructure with Message Routing. To appear in IEEE Internet Computing.

[9] Catholijn M. Jonker, Remco A. Lam, and Jan Treur. A Multi-Agent Architecture for an Intelligent Website in Insurance. In *Cooperative Information Agents III, Proceedings of the Third International Workshop on Cooperative Information Agents, CIA'99*, volume 1652 of *Lecture Notes in Artificial Intelligence*, pages 86–100, 1999.

[10] Danny B. Lange. Java Aglet Application Programming Interface (J-AAPI) White Paper - Draft 2. Technical report, IBM Tokyo Research Laboratory, February 19 1997. Availeble on web: http://www.trl.ibm.co.jp/aglets/JAAPI-whitepaper.html.

[11] Danny B. Lange and Yariv Aridor. Agent Transfer Protocol – ATP/0.1. Technical report, IBM Tokyo Research Laboratory, March 19 1997. Availeble on web: http://www.trl.ibm.co.jp/aglets/atp/atp.htm.

[12] SUN Microsystems. JavaSpaces TM Specification. White paper, SUN Microsystems, January 25 1999. Availeble on web: http://www.sun.com/jini/specs/js.pdf.

[13] SUN Microsystems. Java(tm) idl. web: http://java.sun.com/products/jdk/idl/, Updated 12 december 1999.

[14] Mitsuru Oshima, Guenter Karjoth, and Kouichi Ono. Aglets Specification 1.1 Draft. Technical report, IBM Tokyo Research Laboratory, September 8 1998. Availeble on web: http://www.trl.ibm.co.jp/aglets/spec11.html.

[15] Geir Prestegård, Anders Aas Hanssen, Snorre Brandstadmoen, and Bård Smidsrød Nymoen. DIAS - Distributed Intelligent Agent System. Technical report, Norwegian University of Science and Technology (NTNU), April 1999. Technical Report, Dept. of Computer and Information Science, 387 p.

[16] 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.

[17] Jim White. Mobile Agents White Paper. Technical report, General Magic, 1996. Availeble on web: http://www.ai.univie.ac.at/∼paolo/lva/vu-sa/html/white_whitepaper/.

[18] Joar Øyen. Guidelines for Implementing Software Agent Architectures. Technical report, Norwegian University of Science and Technology (NTNU), December 1998. Technical Report, Dept. of Computer and Information Science, 161 p.