Enterprise Java Beans, Distributed Artificial Intelligence and Group Communication

Technical Report No IC/2002/46

Ion Constantinescu, Steven Willmott, Monique Calisti

Laboratoire d'Intelligence Artificielle, Department Informatique,
Swiss Federal Institute of Technology, IN (Ecublens), CH-1015 Lausanne, Switzerland.
{ion.constantinescu,steven.willmott,monique.calisti}@epfl.ch

Abstract. Today's Middleware is facing increasing pressures to operate in more open, loosely coupled and complex environments. E-business systems require seamless transactions between multiple enterprises and software components are becoming more opaque in an effort to hide their complexity. In parallel, interactions between components are becoming increasingly sophisticated. This paper describes:

- The integration of a well known standards compliant agent platform with a transactional J2EE application server to create a hybrid middleware able to make use of high level agent communication and other features.
- An solution for Group Communication (GC) (a known Distributed Systems problem) using the hybrid middleware.
- The extension of GC to illustrate how new challenges in distributed applications might be dealt with by a convergence of Distributed Systems theory, Distributed Artificial Intelligence and Middleware solutions.

The purpose of the paper is not to eulogise any particular technology but to highlight the relevance of research in the areas of Distributed Systems, Distributed Artificial Intelligence and Middleware and illustrate how they might co-evolve.

Keywords: middleware, distributed systems, distributed artificial intelligence, multiagent systems, group membership

1 Introduction

Today's middleware solutions serve a wide variety of needs and are fast developing to meet new challenges. There is a general shift towards more loosely coupled, flexible and dynamic systems and the definition of middleware is evolving from systems which operate only in closed intranet environments to include inter-enterprise scenarios. Increasingly, users are requiring their middleware to support interactions with third parties (in business to business e-commerce, for example [18]), to support increasingly complex interactions (such as market interactions for example [60]) and provide more sophisticated tools to manage complex systems. There appear to be three major domains which are needed to develop robust, flexible solutions required in the future:

- **Evolving Existing Middleware**: Middleware is already evolving rapidly with increased moves toward message oriented [49, 32], asynchronous [11] and dynamic [31] frameworks. (See Section 2.1.)
- **Distributed Systems (DS)**: Concerned with the formal analysis of problems arising in distributed applications. Developing understanding and ensuring predictable behaviour is likely to become ever more important as we rely more heavily on complex distributed systems. (See Section 2.2.)
- **Distributed Artificial Intelligence (DAI)**: which has developed a significant body of work on complex system-system interactions (such as negotiation [45], [62] and markets [8]), complex reasoning (distributed planning [9] and distributed constraint satisfaction [61]) and modelling systems with highly autonomous components [29]. (See Section 2.3.)

There is increasing cross-fertilisation between these areas with work on Agent Oriented Software Engineering [30], agent architectures using standard middleware (see the FIPA [22] standards, for example) and the clear links between middleware solutions and DS theory. This paper's main objective is to highlight the potential for increasing convergence between these three fields and illustrate how this convergence might come about. This is achieved by:

- Describing a prototype integration of a standards compliant agent platform Jade [54] with the Enterprise Java Beans (EJB) based middleware system application server JBoss [55]. (Section 3.)
- Using the Group Communication (GC) problem to illustrate how standard distributed systems applications can be implemented in such a hybrid middleware. (Section 4.)
- Showing how extensions of the GC directly relevant to the changing needs of distributed applications might form a paradigm which integrates today's middleware and standard Distributed Artificial Intelligence approaches. (Section 5.)

Section 2 begins the paper with brief reviews of emerging application requirements and the technology areas which might contribute to solutions. Section 6 concludes the paper.

2 The Evolution of Distributed Applications

There is no doubt that the transition from single machines, to LAN based environments, to corporate intranet and finally to Internet environments is dramatically changing the needs of distributed applications. Although there are many individual trends in application requirements there are three groups of particular interest with respect to the context of this paper:

1. **Increasing Openness**: with increasing business to business integration (ebXML [18], BizTalk [5] for example), electronic commerce, automated market places [47] and the like, the boundaries between systems belonging to

different organisations are becoming increasingly blurred. This leads to the need for solutions which work in environments were components may be under the control of different organisations. Good example of such openness are frameworks such as GNUTELLA [51] and JXTA [33], which are designed to provide a uniform environment across thousands or millions of network nodes - each of which may be controlled by a separate individual.

2. **Increasing Autonomy**: distributed applications are becoming more complex, multi-layered and of larger scale, making them difficult to design and manage. This promotes a strong trend towards increasing delegation of control to individual processes (often into hierarchical structures [52]) and greater use of abstraction ([2, 7]) to hide complexity in the system. These developments mirror the trend toward asynchronous middleware [49, 32, 11].
3. **More Complex Coordination**: the primary consequence of both of the previous trends is an increase in the complexity of both intra-organisational (multiple departments) and inter-organisational (multiple companies) interactions required to build applications. Components are likely to be more autonomous and opaque (hiding internal processing and structure), heterogeneous (designed and operated by different organisations) and require dynamic discovery (see UDDI [16] and JINI [31] for example).

The first point raises issues with the assumptions underlying much of distributed systems theory and current middleware: that processes in the system are benevolent (do not have malicious intentions) and have been designed to be part of the organisation (that their interfaces are likely to be compatible). The second point suggests that individual processes are likely to exhibit more complex behaviour (since they may be composed of many other systems or take into account complex factors before taking decisions). The third point raises issues in proving different properties (e.g. processes termination), work in all circumstances and have predictable (or optimal) outcomes as well as questions in how to design such complex interactions.

Sections 2.1 to 2.3 briefly covers developments in the three technology areas which relate to these requirements.

2.1 Middleware

Interesting trends in middleware include the following:

- **Message Oriented Middleware (MOM)**: MOM systems are becoming increasingly popular (with systems such as IBM MQ Series [49] or Microsoft Message Queuing [42]), and standardisation initiatives such as the Java Message Service [32] or CORBA event services [11] are broadening usage.
- **Synchronous and asynchronous operations**: many implementations provide different service qualities for both method invocations and message deliveries, including supporting different levels of synchronicity (e.g usage of tupples in JavaSpaces [31] or asynchronous message deliveries with or without acknowledgement timeouts for COSEvent or JMS).

- **Dynamic discovery of interfaces, protocols and data types**: Interfaces to different services are increasingly defined in an reflective manner so that objects can be easily inspected and reconfigured at runtime (see OMG CWM [14] and Jini [31]).
- **Dynamic federation and composition of resources**: As the number and availability of data sources for a particular purpose increases, distributed applications need to leverage them by using resources in a more decoupled way (e.g. being able to create new services by composing exiting ones or to search and negotiate for a given service). Examples of this can be seen in the CORBA trading service [12] and initiatives such as UDDI [16].

Middleware Systems are therefore already being extended to deal with some of the challengers identified in Section 2. The infrastructural support for notions such as service negotiation, dynamic discovery and service composition is generally very basic, however, and each of these areas hides complex computational problems in their own right.

2.2 Distributed Systems

Distributed Systems research is also undergoing considerable change, some relevant trends include:

- **Group processing** [1, 57]: provides a useful abstraction for the design of redundant and fault tolerant systems by grouping a number of processes to provide a single logical function or service.
- **Distributed Transactions and Atomic Multicast**: Distributed transactions are a proven concept mostly used for method invocation middleware [10]. Another concept used in distributed systems is atomic multicast. Equivalences between the two were proven in [35], [48], [27]. Atomic multicast has particular relevance for MOMs which makes it an interesting candidate for future developments.

DS developments therefore mirror middleware developments to an extent.

2.3 Distributed Artificial Intelligence

Distributed Artificial Intelligence (DAI) work tackles a broad range of problems. The key abstraction used in most of this work is the notion of an "agent" as an autonomous process, with research problems related to the interactions between such agents. The precise definition of "agent" used often varies according to the problem being solved (formal definitions of agenthood can be found in [46], [38], [59]). Beyond the individual definitions however, it is important to recognise that it is the solution techniques developed in DAI which are fundamentally important over and above the varying definitions of "agent". DAI's most important contributions in the light of the discussion here appear to be:

- **A fundamentally asynchronous model**: One thing most agent definitions do agree upon is that agents are "autonomous", that is they *operate without the direct intervention of humans or others, and have control over their actions and internal state* [59].[1] Further the mode of interaction is usually via asynchronous message passing of one form or another. This underlying of a fully decoupled asynchronous system assumption therefore forms the basis for most work in the domain.
- **High level communication and Interaction**: Agent communication languages and protocols allows both human-system and system-system interactions, to take place at a higher (semantic) level. This framework facilitates the definition of common problem solving strategies, which can be more easily applied to heterogeneous systems. This "abstraction" level also provides a basis for defining service descriptions, actions and common mechanisms.
- **Coordination techniques**: A large amount of work has been done on how to ensure that groups of agents are able to operate together in a coherent manner and achieve their individual or collective goals (see [28] for a review of coordination approaches).

Accompanying this theory is a wide range of toolkits [54, 53, 43], development environments [56] and methodologies [58]. These tools implement important aspects of the theory and provide developers with the means to create agent based systems. Current trends in implemented systems are:

- **Standardisation**: In recent years there have been significant efforts to standardise some aspects of agent technology - particularly by the Foundation for Intelligent Physical Agents (FIPA - see Section 3.1 for more on this) and OMG MASIF [40] (although this effort seems to have stagnated for the time being). Further efforts include developments in semantic web technology for XML based ontology markup such as the DAML [15] and OIL [19] initiatives.
- **Stability**: Many of the agent systems available today are in their third or fourth generation and are therefore becoming more mature. The feature set provided by most platforms is also relatively constant (and are reflected in the FIPA standards - see Section 3).
- **Methodology**: A recent trend is an increasing emphasis on the use of agents as a system design paradigm [30] for the specification and modelling of complex systems.

Despite these moves, many agent systems still suffer from a lack of maturity, remain untested in outside the research laboratory and could not be considered stable enough for commercial use.

3 FIPA MAS / J2EE Integration

This section outlines a prototype middleware based on the integration of a J2EE application server middleware JBoss [55] and one of the best known FIPA stan-

[1] As we will see in and Section 3.2, Section 3.4, this raises interesting issues with respect to implementation in J2EE frameworks.

dards compliant agent platforms Jade [54]. In Section 4 we discuss how this hybrid middleware can be used to tackle the Group Communication Problem.

3.1 Comparing the J2EE and FIPA Models

This section outlines the differences between the two approaches to be integrated.

Characteristics of J2EE The Java^{TM} 2 Platform, Enterprise Edition (J2EE) defines a standard for developing multi-tier enterprise applications. J2EE is a component-based approach supplying most of the tools needed by such an application (security, naming, transactions, etc.) and also defining clear roles for designers involved in the development of such applications (logical design, appearance design, application assembly, application deployment and so forth).

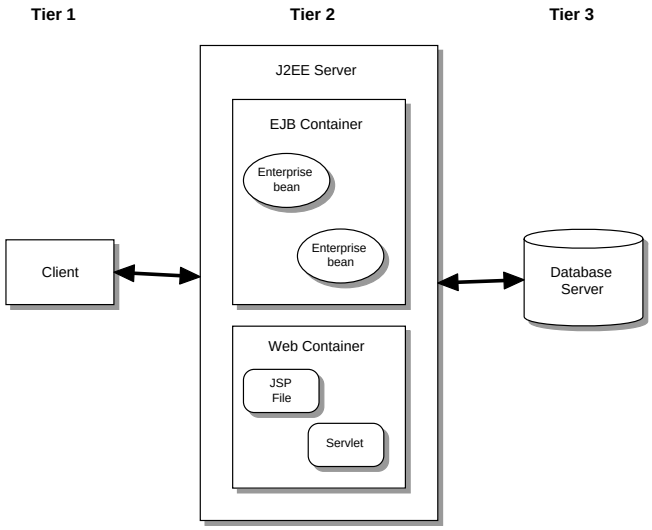

Fig. 1. The J2EE model (taken from [24]) describes a three tier architecture and provides tools and components for the middle one: - servlets and JSP files interact with the user and operate on EJB components implementing the business logic of the application which in turn interacts with a back-end enterprise information system.

As shown in Figure 1, the J2EE application model defines three tiers for building applications: the user interface tier (client or web), the business logic tier and enterprise information system tier. The user interface tier is used for accessing and manipulating the business logic of the application. It can run either on the client machine, on the application server or on both. The business tier implements the logic of the application by taking requests from the interface tier doing different operations, possibly sending requests to the information system tier and returning the results to the user interface. The enterprise information

tier handles enterprise data sources like mainframes, enterprise resource planning systems, databases, etc.

Application components are installed in containers through which they can access platform facilities such as:

- **Security:** web components or enterprise beans can be configured to restrict the access to system resources to authorised users only.
- **Transactions:** specification of relationships among methods that make up a single transaction so that all methods in one transaction are treated as a single unit.
- **Naming:** JNDI lookup services provide to enterprise applications an unified interface to different types of naming and directory services
- **Distribution:** remote connectivity manages low-level communications between clients and enterprise beans or between enterprise beans with different network locations.

Characteristics of the FIPA Architecture FIPA is a standards body for software agents and has been active since 1996. FIPA documents define an abstract architecture, and concrete instances in terms of agent management, message transport and communication:

- **Naming and directories:** agents register upon startup into the local Agent Management System (white pages). They also can advertise their services and capabilities through the use of the Directory Facilitator (yellow pages).
- **Message transport:** FIPA standards define three communications channels that can be used for message transport - CORBA/IIOP, HTTP and WAP. Messages can be routed, passed through gateways and stored in message buffers. Messaging is provided to components inside a FIPA platform as the Message Transport Service (MTS).
- **Communication model:** the communication model relies on the exchange of communicative acts with well defined semantics based on speech act theory [3]. These high level messages (such as "inform", "request" etc.) are used to wrap content in one of several content language and make use of application level domain ontologies.
- **Interaction protocols:** agents can make use of a library of interaction protocols which define sequences of communicative acts. They range from simple request response to protocols for negotiations. These basic interactions can be composed to achieve complex behaviours.

Jade [54] is a middleware platform that complies with the FIPA model. It offers tools for development and debugging, control and management of agents and provides also support for code mobility.

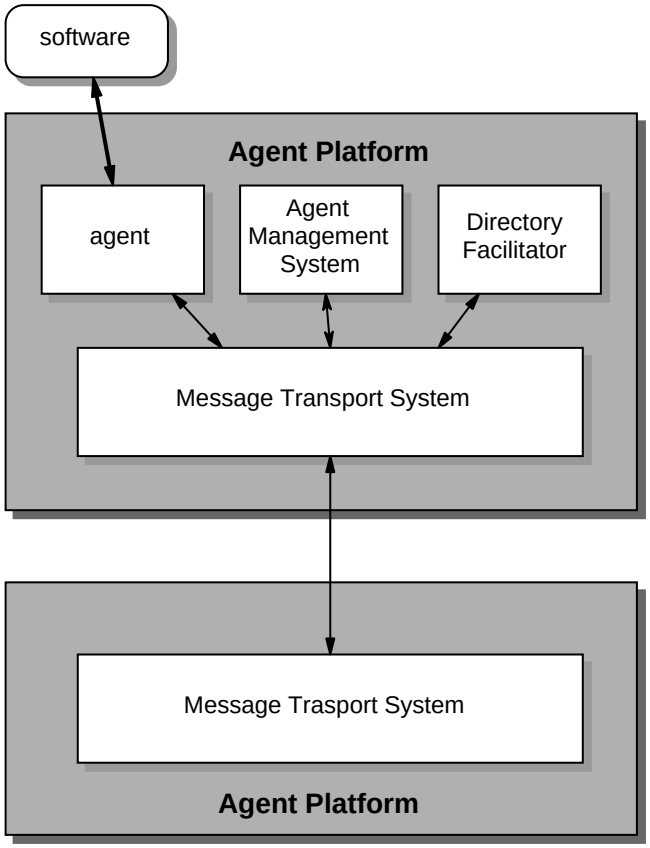

Fig. 2. The FIPA model (taken from [20]) provides for three groups of services: message transport (both local and non-local), management (naming, white pages, life-cycle management) and directories (yellow pages). Finally legacy software can be incorporated into FIPA environments using agent wrappers.

Comparison: Leaving aside the extensive debate on the fundamental conceptual differences between agents and objects (see [58]), the J2EE and FIPA models are very similar in terms of basic architecture and the services they provide. As expected:

- J2EE's strengths lie in: persistence, reliability, high availability and performance (effective memory and CPU usage).
- FIPA's strengths lie in: semantic model, ontology support, standard interaction protocols as well as direct support for autonomy and coordination.

The challenge is therefore to build a middleware which provides access to FIPA's high level features (and hence easy re-use of DAI theories) whilst preserving and enhancing J2EE's strengths.

3.2 Integration Issues

Integrating Jade and JBoss is more complex than simply integrating two different software systems. It is important to consider the models these systems are based upon and the intrinsic characteristics of each system. Among the most important characteristics of the JADE model we have:

- FIPA agent communication apparatus (FIPA-ACL, interaction protocols, content langauges etc.).
- FIPA compliant interfaces to directory services, management services and directories.
- A single thread per agent by an internal scheduler s.t. actions (behaviours) can be scheduled by the agent itself.
- An unique identifier for each agent.

The J2EE framework includes the following premises:

- A single thread based model managed by the AS rather than the agent (invocation based).
- EJB container managed life-cycle (instantiating, loading and saving from secondary storage).
- Concurrent execution for entity beans (i.e. multiple copies are spawned in the case of concurrent requests.
- Message delivery based on transactions and automatic or application acknowledgement of messages.

Although in most areas the fit is good - FIPA interfaces fit with J2EE, message delivery by transactions is not contraditory to Jade's operation, two interesting areas are:

- Threaded model (Jade) versus non threaded model (J2EE).
- The notion of a persistant (singular) entitiy.

In the hybrid architecture described in this paper we sided with J2EE in both cases to respect the J2EE constraints. For the second point (identity), since the AS ensures that state is maintained even if mutliple instances of beans there appears to be little problem. For the first point (threads) however there are some interesting issues related to the autonomy of the agents - these are discussed in Section 3.4.

3.3 Hybrid Jade/J2EE Architecture

The architecture is composed of three main components: a FIPA compliant message transport service (MTS), an Agent Management "Core" (AMC) and a number of end agents (see Figure 3). More concretely the platform provides a service (the MTS) which acts as an interface with other FIPA compliant agent platforms, an enterprise application which keeps track of agents currently on the platform, their types and the ways to contact them and finally "end" agents in form of either enterprise applications or dynamically registered agents on remote transient containers (these are provided by Jade and explained further below).

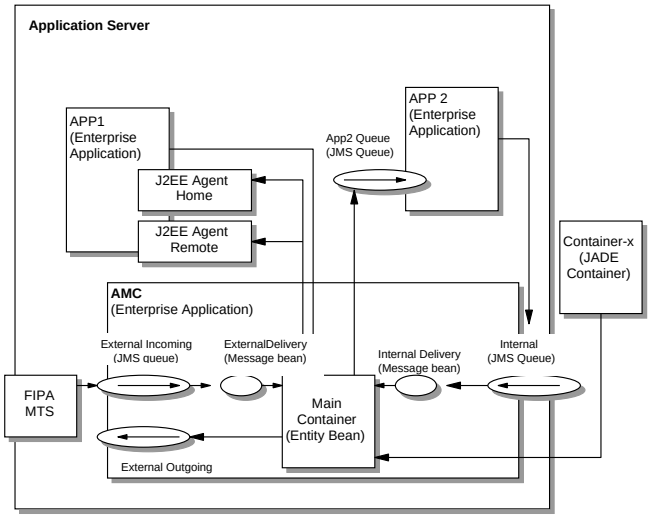

Fig. 3. The hybrid JADE/J2EE is composed of a FIPA Message Transport Service connected to a enterprise application (the AMC), which keeps track of agents on the platform. On top of this applications can be implemented as collections of agents hosted either on the Application service itself or on remote (transient) platforms (JADE containers).

The MTS Service is an asynchronous message delivery facility which complies to the FIPA MTS standard [21]. It is able to receive messages from remote agents over different channels (HTTP/IIOP) and with different content types (string, binary, XML).

The MTS service links with the AMC via two JMS queues - ExternalIncoming (EI) and ExternalOutgoing (EO). These are used to deliver messages from remote platforms to the AMC and from the AMC to external platforms.

The MTS may be linked to the platform in one of two ways: tightly linked - by installing the MTS in the application server, loosely linked - by having the AS provide two message queues and having the MTS fetch and deliver from/to these.

The advantages of the first approach are minimalisation of overhead and following the particular rules enforced by MTS's management of the queues for remote message delivery. The main advantage of the second approach comes from the fact that installation of a JMS service is very dependent on the AS. Sun's J2EE blueprints and the JMS specification standardise interfaces and behaviour for message delivery and factories and destinations management but they do not specify the management interfaces to the services itself. The current implementation is closely linked (support for loosely linked configurations is planned).

The AMC Application's main role is to provide agent management support for the platform. More concretely it has to keep track of types and addresses of agents currently on the platform and provide life-cycle support to agents and management tools. The AMC uses three JMS queues - ExternalIncoming (EI), ExternalOutgoing (EO) and Internal (I), two message driven beans (MDBs) - ExternalDelivery (ED) and InternalDelivery (ID) and an entity bean named MainContainer.

As described above, messages between remote agents and the AMC are delivered through the EI and EO queues. Messages from internal agents are delivered to the AMC either directly or through the I queue. The ED and ID beans are used as bridges between the EI and I queues and the MainContainer. The MainContainer entity bean uses transactions in the invocation methods which affect the life-cycle of agents in order to consistently record the current state of the agents on the platform into persistent storage .

Jade is a distributed platform which can run several "containers" in separate Java Virtual Machines and on separate network hosts. Agents in different containers can communicate via RMI over IIOP and appear to be logically on the same platform. To preserve this feature, the AMC supports Jade's MainContainer remote interface for directly handling invocations from Jade containers. The only major requirement such that existing usual containers directly use the AMC is that they use RMI over IIOP as the underlining transport.

The AMC has also a web interface based on JSP pages for managing the agents.

The End Agents supported by the platform can be included in two main categories:

- "Enterprise agents" which are enterprise applications that have to be deployed into the AS and installed into the MainContainer and for which particular platform resources might also have to be configured.
- "Standard" agents which are hosted on normal Jade containers, register and deregister dynamically only at runtime and have a more transient nature. (These correspond to agents which could be written now in the standard version of Jade.)

This combination allows seamless mixing of different types of agents. Currently there are two kinds of enterprise agents:

- **Entity agents:** these implement the J2EEAgentHome and J2EEAgentRemote interfaces such that messages are delivered by passing them as an argument on the invocation of a particular known method. They are identified by the AMC internally using the location of the home interface and the name of the agent. The name is also used as a parameter for the finder method of the J2EEAgentHome interface when trying to locate the agent. This kind of agents are useful when support for a large number of agents of the same type is needed.

– **Message driven agents:** these use an agent dependent message queue for receiving messages from the AMC and use the Internal queue to send messages to the AMC. Each agent is identified by the AMC internally using the location of his QueueFactory, of his QueueDestination and the agent name. Note that in fact this kind of agents do not necessarily need to be implemented as an enterprise applications because usually JMS services allow message delivery also outside the AS.

In the near future, the idea is to develop a third type of "enterprise" agent based on session beans. These agent types correspond to the primary types of Java Beans commonly used in J2EE environments so that agents have full access to all J2EE functionalities.

3.4 The Notion of Autonomy, J2EE and Jade

As stated in Section 3.2, the J2EE model limits the types of behaviour which could be implemented in "enterprise" agents. This is even clearer from the descriptions of entity and message driven agents in the previous section. Comparing to common definitions of agents (such as [29]), the agents are:

– Situated - receiving messages and events from user actions and other agents in the environment.
– Social - they are able to interact with other agents in the environment (in this case using standard protocols, communication languages, ontologies etc.)
– Responsive - able to take complex decisions and perform complex actions based on events in the environment.

The agents are however purely "reactive" - that is, they *do not act spontaneously* without being activated by an event or message. They would therefore be considered as agents by some definitions of agent and not by others (see [26] for a debate on the reactive v's deliberative issue). Definitions which do not allow purely reactive systems generally require a notion independent action based upon taking actions *pro-actively* without being prompted by any external event.

This is highlighted by the difference between programming a standard Jade agent (which has its own thread) and an enterprise agent (which does not). With the standard Jade agent it is possible to schedule events based on time and looping behaviours which cannot be done with enterprise agents.

Interesting points are therefore: how can autonomy be modelled (or "allowed" in a J2EE/Agent hybrid)? Is it in fact desirable to enable autonomous action ? There appear to be two main ways by which "full autonomy" could be achieved in a hybrid architecture:

1. Violate the J2EE constraints and allow agents to have their own threads.
2. Add an external timer which periodically sends trigger (time) events to enterprise agents. On receipt of the event the agents can evaluate if actions of any kind are needed (a rough equivalent to schedulers automatically available in multi-threaded environments but very symilary to **??**).

These examples highlight that the divide between deliberative / reactive agents is in fact rather fuzzy and depends on the level the system is described at. In many systems, actions are almost always taken in response to changes in the environment and hence have a reactive aspect. The utility of introducing "full autonomy" (pure pro-activity) is clearly application dependent. However, the authors believe that as distributed applications develop the standard J2EE model of assuming an external transaction trigger (usually a user) for all actions will need to be augmented. The seems likely to take the form of additional support for triggers or schedulers based on time or environemtn sensors (e.g. Gizmo Co's inventory running low on widgets). These types of triggers are already built into database systems and could usefully be brought into the J2EE model more formally.

4 Implementing a Group Communication service

To illustrate the points made in the previous sections, this section considers the Group Communication (GC) problem [23, 13, 6, 17]. We first consider the GC problem as usually described in the DS literature (Section 4) and then in Section 5 go on to possible extensions of GC, which correspond to the requirements for future distributed applications in Section 2.

Group communication is a well known distributed systems paradigm. It addresses the reliable transmission of a message to a determined set of "processes". This abstraction is usually used for solving higher level problems such as replication, fault tolerance, atomic multicast, notifications, etc.

Formalisations of the problem consider a Group Communication Service (GCS) used by a number of processes distributed across a network s.t.:[2]

1. A number of processes $p_i \in S_P$ (know as GCS processes) are considered as forming a *group* $g_j \in G$, where S_P is the system consisting of all processes and G the set of all groups.
2. Processes can *join* and *leave* the group dynamically at any time (due to network failures, process crashes or their own volition).
3. Each process in group g_j must to keep a consistent view $v(g_j)$ of which other processes are members of g_j at any one time.
4. The processes in g_j are able to ensure reliable delivery of messages to all members of g_j.

The first three statements define the Group Membership problem [13, 6, 17] which underlies the Group Communication problem [1, 57]. The problems involved in creating a GCS system can be broken down as follows:

– Each process p_i must maintain a current view $v(g)$ of the groups it is a member of.

[2] Note that from here onwards descriptions a considerably simplified for presentation purposes.

- Each process p_i must be able to:
  - *Join* groups it is not already member of.
  - *Leave* groups it is a member of.
  - *Send* messages to and *receive* messages from groups it is a member of.
- Mechanisms must exist such that:
  - Processes can ensure that all members p_k of a group g_j have the same view $v(g_j)$ of the group.
  - All messages are delivered to all members of g_j (atomic multicast).
  - Agents which have failed or joined can be detected and removed from or integrated into g_j.

These features are implemented by the protocols and systems which provide the GCS. (Typically a number of other actions such as flush and block [57] are also required but these are less relevant here).

4.1 Implementation in Hybrid Jade/J2EE

Mapping these operations and solutions found in existing literature into an agent framework is relatively straight forward. In our prototype implementation we considered partitionable systems, primary-component computed based on a majority quorum, sending view delivery and weak total order [57]. This could be extended to include other variations.

The mapping into the agent framework is as follows:

- Each GCS process p_i is implemented by an agent $a_i \in S_A$, where S_A is the system consisting of all agents.
- Each agent has a belief $Bel(currentview, v(g_j))$ about the current view of each group $g_i \in G_{in}$ the agent where G_{in} is the subset of G, a_i is a member of.
- Agents may also establish beliefs $Bel(currentview, v(g_{j'}))$ about the current view of each group $g_{j'} \in G_{out}$ the agent is *not* member of (i.e. G_{out} is the complement of G_{in} in the set of all groups w.r.t. a_i's membership).
- The agent may establish beliefs $Bel(hascurrentview, a_s, v(g_j))$ about the views other agents $a_s \in S_A$ might have of particular groups $g_j \in G$.
- Each agent is able to query the FIPA yellow pages directory service (DF) to obtain a list of current agents in the system S_A and which groups they belong to using the FIPA standard request protocol.
- Agents send messages to a group $g_i \in G$ by multicasting to their current stable view $v(g_i)$ over the multicast message service in J2EE or Jade. The message is wrapped in a FIPA inform communicative act.
- Joining is achieved by sending inform messages containing the required view to all members of the group (established through a view $vg_{j'}$ generated by asking the DF for a list of agents in S_A) the agent wishes to join (see Figure 4).

– View consensus: If agents realise that their view of a group g_j is inconsistent with that of at least one other agent for some reason (due to a failure, new agent joining etc.) they propagate their current views as described above. In the current implementation consistency is established using a vector clocks [34, 39, 44] approach which the agents implement in their reasoning.
– Detection of failed or disconnected agents is carried out using a simple ping protocol (using the FIPA standard Query protocol) which all agents periodically send to members of their respective groups (see Figure 5).

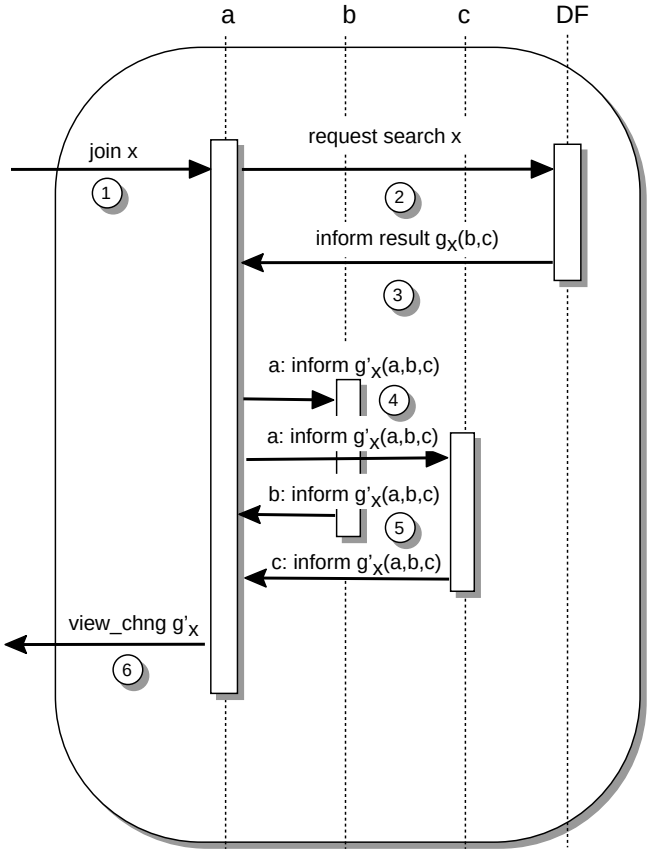

Fig. 4. AUML [4] message sequence diagram for simple joining protocol.

One example is the sequence of messages that are generated when a new GCS agent a wants to join a given group g_x:

1. Agent a receives a request to *join* group g_x (effectively from outside the system).

2. Agent a searches the DF for group g_x.
3. The DF replies with the result group g_x currently contains agents b and c.
4. Agent a then sends to b and c a new inform *membership* message expressing a belief that the view has changed: $Bel(currentview, v(g_x))$ such that $v(g_x) = \{a, b, c\}$.
5. Agents b and c reply with the same view.
6. As a result a decides to install the new view and delivers a *view change* message to the upper layer (back outside the system).

This is a simplified version to the join procedure described in [17] (although with the addition of the DF step). The next example shows how agent a carries on failure detection, decides that agent b has crashed and creates a new view with agent c:

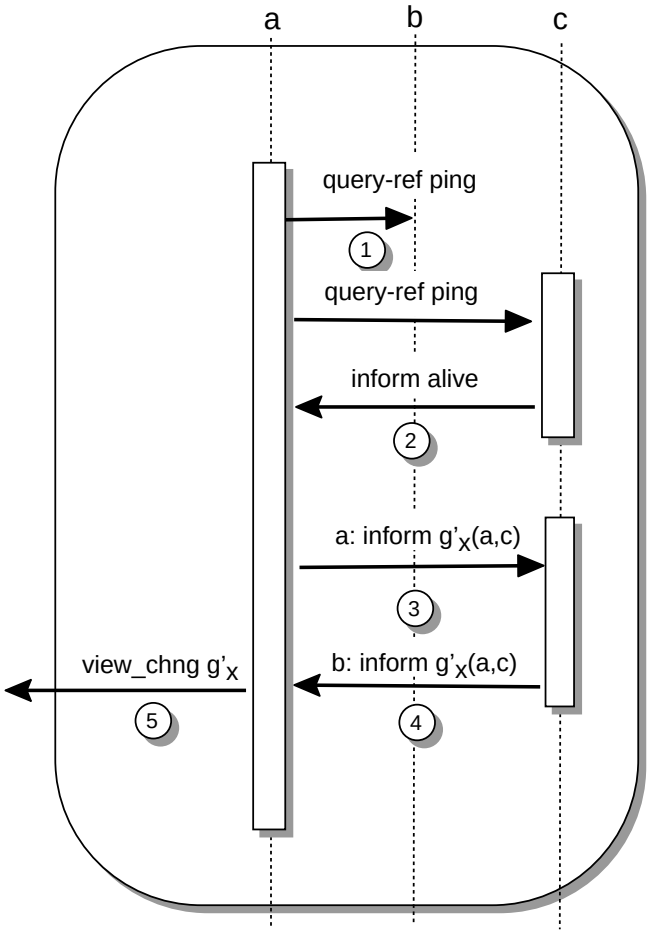

Fig. 5. AUML message sequence diagram for failure detection protocol.

1. Agent a sends query-ref ping messages to agents b and c.
2. Agent c replies with an inform alive.
3. After a timeout agent a decides that b has crashed and sends a new view to agent c containing a and c.
4. Agent c decides the same and sends to a the same view containing a and c.
5. Finally agent a install the new view and delivers a *view change* message to the upper layer.

Seen like this, the semantics of communication provided by the agent framework (Jade/FIPA) can be used to give a simple account of the interactions involved in implementing a GCS. The semantics make explicit what each agent (process) is entitled to believe at any one time and the exercise highlights a number of implicit assumptions in the standard protocol including that every agent is expected to accept the beliefs of others.

The implementation of the GCS service relies on the architecture described in Section 3. In particular, the GCS framework has been modelled as an enterprise application using a message oriented model. The interface with the AMC is realised through two JMS Queues - the Internal queue defined by the application server architecture and a custom GcsQueue installed in the AMC for delivering messages to the GCS agent. Because it uses JMS the GCS is able to acknowledge and commit certain sequences of messages. Transaction commit on queues can be done by the GCS when for example safe prefix message are received.

The GCS contains also a failure detection subsystem. Since (as discussed in Section 3.4) only invoking threads are available inside the application server no pro-active behaviour can be implemented using the core J2EE tools however. So due to the cyclic nature of the failure detection system this is implemented as an external component. Communication with the GCS was also done through the GcsQueue.

5 New Perspectives on the Group Membership Problem

As indicated in Section 2 here are a number of trends in distributed application requirements which present challenges for standard middleware. This section discusses a number of extensions to the standard GCS problem to illustrates how hybrid architectures such as the one described in Section 3 may by useful in addressing some of the points raised in Section 2.

5.1 Group Communication with a Moderator

One of the most significant problems created by increasingly open environments is the possibility that malicious agents (processes) will attempt to subvert protocols and services to achieve their own ends or simply disrupt normal usage. Good examples of such behaviour in GCS systems are sending large amounts of messages, sending oversized messages or sending messages with undesirable content (unsolicted advertisements for Gizmos). This type of problem is not in

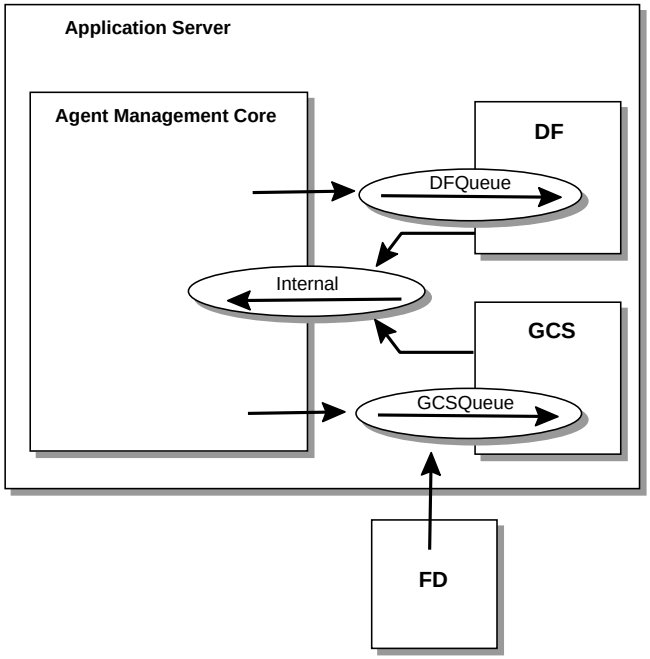

Fig. 6. The implementation of the GCS using the hybrid architecture described in 3 is realised as a message consumer which makes use of the AMC and relies on another enterprise application - the directory facilitator (DF). It also contains a failure detector (FD) implemented as an agent external to the application server.

fact a protocols level failure or a process failure, it is an application level abuse of a service. One way to deal with such attacks is to introduce moderators into the system.

An extension of the GCS implementation described in the previous section to include moderators is underway. At the time of writing however it is unfortunately *not* complete, hence the description is an illustrative outline of how the system might work. The extension can be described as follows:

- Introducing the notion of a *group policy* $p(g_j)$ for a group g_j which states usage rules for the group and the penalty for violation. A simple example would be the
  - Rule: Agents must not exceed a maximum message send rate r_j for the group g_j.
  - Penalty: immediate expulsion from the group and a ban on re-entering for a specified time t.
- Introducing the notion of *moderator status* for a group g_j which can be assigned to agents S_A.
- Ensuring that all agents in the group are aware which agents are moderators, either by including this information in view propagation messages or through additional messages.

- Agents with moderator status for a group g_j and which are currently members of g_j:
  - Have the goal of ensuring that $p(g_j)$ is enforced.
  - May monitor activity in the group and detect violations of the group policy $p(g_j)$ usage. This could be achieved by recording average send rates over time for all agents in the group (the moderator receives messages since it is itself a member of the group, we assume that mechanisms are in place to prevent malicious agents sending to everybody in the group except the moderators).
  - May enact an expulsion by:
    * Sending a new view change message which excludes the agent a_v violating the rule to all members of the group and containing an explicit preference to the exclusion.
    * Sending a message to a_v citing its violation.
    * Since other agents are aware of the status of the moderator they:
      · Are more likely to accept this view change (in preference to an attempt to remain in the group by a_v for example).
      · Create a belief $Bel(excluded, a_v, t)$ representing the fact that a_v is excluded from the group until time t.
      · Reject attempts by a_v to re-join the group until t has expired.
- One could also imagine an appeall proceedure wherby the agent negotiates with the moderator which excluded it to regain entry.
- Clearly there are a number of variations on this: different types of policies (message size, content etc.), assigning one or n moderators to the group, mechanisms for moderator election, agreement between independent moderators, different types of monitoring and excluding agents.

Value of the Agent Aspects The scenario highlights the utility of the agent model for GCS since:

- The strong notion of identity in the agent model [20] is exploited in assigning the role of the monitor and in identifying the violating agent, both at the time of the exclusion and as a permanent reference during the exclusion period t.
- The agents lend themselves well to expressing authority and this is exploited in the way agents recognise the authority of the moderators in excluding the violating agent above attempts of the violating agent to remain in the group. More complex notions of social relationships and social structure are well studied in work on (for example) computational organisations [25] and social laws [41].
- The model makes it easy to mix application level (correct usage of the GCS) with implementation level concerns. The violating agent is *not* violating the actual GCS protocol itself, nor has it failed, it is misusing the system according to the commonly agree policy.

- The use of an explicit policy raises the description of the system to a higher level that (for example) extensions of the protocol given in [17] would. This makes it possible to link to more advanced work on polices such as that described in [50] and [37].
- The model clearly makes it easier to take advantage of more complex DAI techniques for reaching agreement such as negotiation variants (see [36] for an overview), markets [8], distributed planning [9] or distributed constraint solving [61].

Value of the J2EE Aspects The value of implementing this type of extended service in a hybrid architecture such as the one described in Section 3 rather than on a standard agent platform (such as JADE) lies in the robustness an high performance of the hybrid architecture. The J2EE component of the architecture ensures that agents and messages are persistent (no message loss), highly available and efficient in system resource usage.

For a general purpose GCS system to be of use in industrial applications these requirements must be met and therefore the integration experiment appears to be a valuable one. It is unlikely that environments which did not provide guarantees of persistance, robustness (in particular resistance to message loss) could be considered for mission critical services.

5.2 Further Extensions

Other examples of interesting extensions to the GCS include:

- Negotiation at the view resolution stage to enable more complex decision making as to who might join the group. Again this corresponds to dealing with the consequences of openness (this time attemtping to screen gorup membership in advance).
- Use of formal ontologies in group descriptions to allow agents to more easily identify groups of interest. This corresponds to dealing with increased heterogeneity.
- Inserting third parties (moderators, trusted third parties etc.) directly into the message flow. That is - a message for the group first passes through a number of other agents before being delivered, or acknowledgements are needed from several parties that particular messages were received (to record the fact of message delivery).This is related to increasing the complexity of interactions and processes in the system.

5.3 A Final Note on Autonomy

The GCS extensions are also instructive on the points made about the autonomy of "enterprise" agents. In the moderator example, the moderator agent is clearly embedded in its environment (by monitoring traffic), *reactive* to that environment (by taking action when necessary) but is also acting in a goal driven

manner. That is, action is taken when the goal of rule adherence by agents is violated. Therefore, while the final action of excluding another agent is triggered by an incoming message (as sensory input) the decision to act is based on a higher level goal. While this is still not equivelant to scheduling an action completely independently it illustrates that complex decisions can result from simple stimuli (in this case many stimuli over time).

The requirement for a trigger can also be seen in the algorithm description for standard GCS (see Section 4.1). In this case the agent only makes group join actions in response to user prompts from what is effectively outside the system.

6 Conclusions

This paper presents a perspective on the future evolution of distributed applications and analyses contributions that the areas of middleware, distributed systems and distributed artificial intelligence may be able to make. There are clear pressures in each of these fields to co-evolve:

- DAI systems have long been concerned with complex interactions, intelligent autonomous processes but must begin to address concerns of scalability, reliability, provability.
- Distributed Systems theory needs to be further generalised to fields considering hazards such as malicious processes, more complex interactions and more opaque, unpredictable behaviour.
- Middleware solutions are becoming more loosely coupled and flexible but are likely to face great challenges as they are expected to cope with some of the more advanced requirements imposed by tomorrow's distributed applications.

We also present a small piece of this convergence puzzle with the integration of a standards compliant agent system with a J2EE platform to create a robust middleware able to access high level agent communication tools. The resulting hybrid platform was then used to implement a robust "agentified" extended Group Communication Service illustrating how access to Distributed Artificial Intelligence theory may open doors to tackling some of the key challenges faced.

The work presented here is preliminary and there is undoubtedly much more to be said on the convergence between Middleware, Distributed Systems and Distributed Artificial Intelligence. We hope however that the discussion herein represents a useful contribution to the debate on future Middleware solutions.

Acknowledgements

The authors gratefully acknowledge the useful comments made by Giovanni Rimassa on earlier drafts of this paper. Also many thanks to all the JADE team for their precious collaboration.

References

1. Yair Amir. *Replication Using Group Communication Over a Partitioned Network.* PhD thesis, 1995.
2. T. Anker, G. Chockler, D. Dolev, and I. Keidar. Scalable group membership services for novel applications, 1998.
3. John L. Austin. *How to Do Things With Words.* Oxford University Press, Oxford, 1962.
4. Muller J. P. Bauer B. and Odell J. An Extension of UML by Protocols for Multi-agent Interaction. In *Proceedings of ICMAS 2000, Boston, MA, 2000*, 2000.
5. BizTalk. http://www.biztalk.org/.
6. Tushar Deepak Chandra, Vassos Hadzilacos, Sam Toueg, and Bernadette Charron-Bost. On the impossibility of group membership. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC'96)*, pages 322–330, New York, USA, 1996. ACM.
7. Anawat Chankhunthod, Peter Danzig, Chuck Neerdaels, Michael F. Schwartz, and Kurt J. Worrell. A hierarchical Internet object cache. In *Proceedings of the 1996 Usenix Technical Conference*, San Diego, CA, 1996.
8. S. H. Clearwater. *Market Based Control: A paradigm for Distributed Resource Allocation.* World Scientific, Singapore, 1996.
9. S. E. Conry, R. A. Meyer, and J. E. Searleman. Multistage Negotiation in Distributed Planning. In A. H. Bond and L. Gasser, editors, *Readings in Distributed Artificial Intelligence*, pages 367–384. Morgan Kaufmann, 1988.
10. Common Object Request Broker (CORBA). http://www.corba.org/.
11. COSEvent. http://www.omg.org/technology/documents/formal/event_service.htm.
12. COSTrading. http://www.omg.org/technology/documents/formal/trading_object_service.htm.
13. F. Cristian. Reaching agreement on processor-group membership in synchronous distributed systems. *Distributed Computing*, 4(4):175–188, 1991.
14. Common Warehouse Metamodel (CWM). http://www.omg.org/technology/cwm/index.htm.
15. DAML. Darpa Agent Markup Language: DAML+OIL specification v 1.7. Technical report, DAML Project, 2001.
16. Universal Description Discovery and Integration (UDDI). http://www.uddi.org/.
17. D. Dolev, D. Malki, and R. Strong. An asynchronous membership protocol that tolerates partitions, 1993.
18. ebXML Consortium. Whitepaper: Enabling Electronic Business with ebXML. Technical report, UN/CEFACT and OASIS, December 2000.
19. D. Fensel, I. Horrocks, F. Van Harmelen, S. Decker, M. Erdmann, and Klein M. OIL in a Nutshell. In *Proceedings of the Workshop on Applications of Ontologies and Problem-Solving Methods, 14th European Conference on Artificial Intelligence (2000) 16.1-16.4.* 2000.
20. FIPA. FIPA Agent Management Specification (00023). Technical report, Foundation for Intelligent Physical Agents, 2000.
21. FIPA. FIPA Agent Message Transport Service Specification (00067). Technical report, Foundation for Intelligence Physical Agents, 2000.
22. FIPA. FIPA Web pages. Technical report, Foundation for Intelligent Physical Agents, 2001.
23. M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. Technical Report 2, 1985.
24. The J2EE Tutorial for J2EE SDK version 1.3. http://www.javasoft.com/j2ee/tutorial/.

25. M. S. Fox. An Organisational View of Distributed Systems. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-11(1):70–80, 1981.
26. S. Franklin and A. Graesser. Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents. In *Intelligent Agents III. Agent Theories, Architectures and Languages (ATAL'96)*, volume 1193 of *LNCS*, Berlin, Germany, 1996. Springer-Verlag.
27. R. Guerraoui and A. Schiper. Transaction model vs virtual synchrony model: bridging the gap. In *Theory and Practice in Distributed Systems*, pages 121–132. Springer-Verlag, 1995.
28. N. R. Jennings. Coordination Techniques for Distributed Artificial Intelligence. In F. M. P. O'Hare and N. R. Jennings, editors, *Foundations of Distributed Artificial Intelligence*, pages 187–210. John Wiley & Sons, 1996.
29. N. R. Jennings, K. Sycara, and M. Wooldridge. A Roadmap of Agent Research and Development. *Journal of Autonomous Agents and Multi-Agent Systems*, 1:275–306, March 1998.
30. N. R. Jennings and M. Wooldrige. Agent-Oriented Software Engineering. In J. Bradshaw, editor, *Handbook of Agent Technology*. AAAI/MIT Press, 2001.
31. Jini. http://www.jini.org.
32. Java Message Service (JMS). http://www.javasoft.com/jms/.
33. JXTA. http://www.jxta.org.
34. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, july 1978.
35. Mark C. Little and Santosh K. Shrivastava. Understanding the role of atomic transactions and group communications in implementing persistent replicated objects. In *POS/PJW*, pages 17–28, 1998.
36. A. Lomuscio, M. Wooldridge, and N. Jennings. A classification scheme for negotiation in eletronic commerce. In F. Dignum and C. Sierra, editors, *Agent-Mediated Electronic Commerce: A European Perspective*, pages 19–33. Springer Verlag, 2000.
37. E. Lupu, M. Sloman, and N. Yialelis. Policy based roles for distributed systems security, 1997.
38. Pattie Maes. Artificial life meets entertainment: Lifelike autonomous agents. *Communications of the ACM*, 38(11):108–114, November 1995.
39. Friedemann Mattern. Virtual time and global states of distributed systems. In M. Cosnard et. al., editor, *Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B. V., 1989.
40. D. Milojicic, M. Breugst, I. Busse, J. Campbell, S. Covaci, B. Friedman, K. Kosaka, D. Lange, K. Ono, M. Oshima, C. Tham, S. Virdhagriswaran, and J. White. MASIF: The OMG Mobile Agent System Interoperability Facility. In K. Rothermel and F. Hohl, editors, *Proceedings of the 2nd International Workshop on Mobile Agents*, volume 1477 of *Lecture Notes in Computer Science*, pages 50–67. Springer-Verlag: Heidelberg, Germany, 1998.
41. Y. Moses and M. Tennenholz. Artificial Social Systems. *Computers and Artificial Intelligence*, 14(6):533–562, 1995.
42. Microsoft Message Queuing (MSMQ). http://www.microsoft.com/msmq/.
43. Charles Petrie. Agent-based engineering, the web, and intelligence. *IEEE Expert*, 11(6):24–29, December 1996.
44. Michel Raynal and Mukesh Singhal. Logical time: A way to capture causality in distributed systems. Technical Report RR-2472, 1995.
45. Rosenschein, J. S. and Zlotkin, G. *Rules of Encounter*. MIT Press: Cambridge, MA., 1994.

46. Stuart J. Russell and Peter Norvig. *Artificial Intelligence. A Modern Approach.* Prentice-Hall, Englewood Cliffs, 1995.
47. Tuomas Sandholm. eMediator: a next generation electronic commerce server. In Carles Sierra, Gini Maria, and Jeffrey S. Rosenschein, editors, *Proceedings of the 4th International Conference on Autonomous Agents (AGENTS-00)*, pages 341–348, NY, June 3–7 2000. ACM Press.
48. André Schiper and Michael Raynat. From group communication to transactions in distributed systems. *Communications of the ACM*, 39(4):84–87, 1996.
49. IBM MQ Series. http://www.ibm.com/software/ts/mqseries/.
50. M. Sloman. Policy driven management for distributed systems, 1994.
51. The GNUtella Protocol Specification. http://www.clip2.com.
52. C. A. Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison Wesley, 1997.
53. FIPA-OS Open Source Development Team. FIPA-OS Agent Platform. Technical report, Open Source Project, 2001.
54. Jade Open Source Development Team. Java Agent Development Environment. Technical report, Open Source Project, 2001.
55. JBoss Open Source Team. JBoss Open Source J2EE Server v2.2, 2001.
56. Zeus Open Source Development Team. ZEUS Agent Development Environment. Technical report, Open Source Project, 2001.
57. R. Vitenberg, I. Keidar, G. Chockler, and D. Dolev. Group communication specifications: A comprehensive study, 1999.
58. M. Wooldridge, N. R. Jennings, and D. Kinny. The GAIA methodology for Agent-Oriented Analysis and Design. *International Journal of Autonomous Agents and Multi-Agent Systems*, 3, 2000.
59. M. Wooldridge and N.R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.
60. F. Ygge and H. Akkermans. Power Load Management as a Computational Market. In M. Tokoro, editor, *Proceedings of the Second International Conference on Multi-Agent Systems (ICMAS'96)*. AAAI Press, 1996.
61. M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. Distributed Constraint Satisfaction for Formalising Distributed Problem Solving. *Proceedings 12th IEEE International Conference on Distributed Computing Systems.*, pages 614–621, 1992.
62. G. Zlotkin and J. S. Rosenschein. Negotiation and Task Sharing Among Autonmous Agents in Cooperative Domains. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI'89)*, pages 912–917. 1989.