The AAA Agent-based Message Oriented Middleware

N. De Palma¹, L. Bellissard², D. Féliot, A.Freyssinet³, M. Herrmann³, S. Lacourte³

SIRAC project and AAA-DYADE⁴
INRIA Rhône-Alpes,
655, Avenue de l’Europe
F-38330 MONTBONNOT SAINT-MARTIN, France
Tel: +33 4 76 61 52 00, Fax: +33 4 76 61 52 52
Email: Luc.Bellissard@inrialpes.fr

1 Introduction

The past years have witnessed the decentralisation of the computing resources within most of the organisations. Reasons are numerous but the main one is the rise of small, powerful and affordable computers. As a result, the computing environment has become more and more distributed and heterogeneous while departments or even end-users have developed very efficient solutions for their particular needs. Consequently, interoperability concerns with the rest of the organisation global computing system have not been taken into account. From the organisation point of view, reducing the cost of software development is a major concern, thus leading to tremendous efforts on the integration and reuse of software components. One needs to address how the geographically spread applications can inter-operate, forming clusters of distributed applications. So the biggest challenge is to develop software that connects legacy components over the network in an inexpensive and flexible way to meet evolving requirements.

Various middleware technologies like CORBA[9], DCOM[6][11], RMI[12] are providing solutions but some application domains are not fully covered by those technologies. For example, “24 by 7” environments[8] request hardware or software failure recovery, guaranteed transmission, flexible cooperation between software components and scalability. We are currently involved in a commercial application dealing with Internet security. The application is a firewall, originally designed to work on a single node controlling an entire Intranet network. The problem with this application is related to the evolution of the firewalls functionalities and to the co-ordination and control of several firewalls together on a same Intranet, each of them acting as a gateway to one department of the organisation. The following requirements are essential to this application:

- **R1: Legacy system**: interact with the existing firewall software

---

¹ Affiliated to Joseph Fourier University, Grenoble
² Affiliated to INPG (National Polytechnic Institute of Grenoble)
³ Affiliated to Bull S.A.
⁴ DYADE is a joint consortium between Bull S.A. and INRIA (Institut National de Recherche en Informatique et Automatique)
• **R2: Flexibility**: add new functions to an existing firewall with external, added value software components

• **R3: Glue software**: add software for the co-ordination and control of distributed firewalls.

• **R4: Autonomous execution**: firewall software are working independently from the other with weak synchronization.

**Message Oriented Middleware**

Message Oriented Middleware[3] (MOM) are the alternative to classical middleware. MOM are based on asynchronous messages as the single structure for communication, coordination and synchronization, thus allowing desynchronized execution of components. Reliable communication is guaranteed by message queueing techniques (e.g. MQSeries[4]) or specific communication protocols (e.g. iBus[7]) that can be added independently from the programming of software components. However, current solutions only provide to programmers a communication interface (e.g. JMS[15]) and recovery applies only to messages but not necessarily to software components themselves.

In this paper, we introduce a Message Oriented Middleware which combines the previously stated communication properties as well as a programming model of distributed, persistent software entities called agents into a single homogeneous platform. The platform handles not only reliable communication but also recovery of software failures, while the programming model of the software entities remains simple and flexible.

2 Distributed Programming Model

2.1 **Agent Paradigm**

The agent paradigm is close to the reactive object paradigm[1] in an asynchronous world. Agents are active objects which can run concurrently and have their own state. They behave according to an event/reaction model. An *event* is a typed data structure used for exchanging information with other agents. Information means in this particular case any data structure that can be passed by value or agent references. Once an agent receives an event, it reacts accordingly thus changing its state and/or communicating with other agents. This event/reaction model is based on a message-passing middleware which ensures asynchronous and anonymous communication properties. Sending and reacting to events are the only way for agents to communicate but an agent never directly designate another agent. Events are sent through « Roles ». Each Role is bound to a set of agent that will receive any events sent through them. This is how anonymous communication is achieved, the binding process is done by adding new listening agents to a Role. As a consequence, agents do not share their state which makes them real autonomous entities. The following agent definition in a Java-like syntax is an illustration of how an agent is programmed.
This example shows that an agent is composed of a state (identified by its attributes) and a set of rules. These rules describe how an agent reacts to events (i.e., reaction1 is triggered when event1 and condition1 occur) and how an agent communicates and exchanges information with another agent.

- **Anonymous communication**: anonymous communication means that agents do not know events targets. As shown in Figure 1, an agent triggers events through a role. Roles dynamically manage the relationships between agents and events in order to achieve the communication. An agent must subscribe to a role before receiving the corresponding events. In addition, subscription can be performed by the agent itself or by a third party. This property improves application flexibility. It is a way of separating concerns: agents interconnections can be changed without modifying the agent source code.

- **Asynchronous communication**: The asynchronous communication schema decouples producers of information from consumers. They do not need to be both ready for execution at the same time.

### 2.2 Agent Infrastructure Properties

Complex services provisioning in a distributed environment require a flexible, scalable and reliable infrastructure, mainly because component should be able to continue processing after an event is sent even though the destination component is not available. Components can be unavailable because they are stopped (e.g., end-users decision or administrative operations), busy or because hardware or network failures have occurred. In order to cope with application unavailability, the agent infrastructure provides recovery facilities, a reliable communication mechanism and ordering and reordering facilities for event delivery. Let us detail each of these properties.

#### 2.2.1 Agents Properties

One of the requirements of a glue software system is its reliability. Execution structures must be able to resist to failures, in particular node failures. For this reason, any agent reaction is atomic and agent are persistent entities, both enabling the recovery of a consistent agent state in case of a failure.

- **Persistence**: Agents are persistent. This means that the agent lifetime is not bounded to the duration of the execution. The agent state is not lost when the execution node stops or crashes. However, agent persistency is not sufficient for retrieving a consistent state after failures.

- **Atomicity**: Therefore, agent reactions are atomic. This property ensures that a reaction is either fully executed or not executed at all: if the reaction is fully executed, the agent new state is committed and all events triggered during the reaction are actually delivered to their
destination. If a failure occurs, the system rollbacks to a consistent state by retrieving the agent’s initial state and removing any events triggered during the reaction.

2.2.2 Communication properties

At the programming level, agents communicate by triggering events. At the infrastructure level, notification of events are transformed into messages. Messages are the units of information carried out by the communication system implemented within the infrastructure. Remember that events are the only way for agents to communicate, to synchronise and to share information. The communication mechanism is based on message queueing system, which provide asynchronous, reliable and ordered communications.

- **Asynchronous**: The asynchronous property allows the work to be performed as soon as applications are ready. As a consequence, applications can be designed and implemented in a time-independent manner. This property enables a deferred access to applications and a loose coupling of software components with each other.

- **Reliability**: The reliability property allows the work to be performed despite application unavailability due to transient failures. This property provides a recovery mechanism for messages lost during network failure or system crash. Message recovery capabilities enable lost messages to be re-sent with no involvement from the application. Events once sent are guaranteed to be delivered. This property is the basis of message queueing systems.

- **Anonymous**: communication is anonymous for the agent programmer since all message are sent through roles, which identifies a set of receiver agents. Agents must subscribe to the role in order to receive this event.

2.3 Programming Agents

2.3.1 Implementation principles

Agents are implemented by Java objects. The object class `Agent` defines the behaviour common to all agents. Reliability and configurability properties are provided by this particular class. As a consequence, these properties are inherited by all agents.

The agent programming is basically restricted to the implementation of the reactive behaviour with the function member `react`. In order to communicate, agents use the `sendTo` method, which is in charge of transforming the event into a message that can be handled by the communication system. Programmers do not have to deal with networking details for describing communication between agents, working with agents in a local process or in remote processes is the same. Since java serialization is used to ensure agent persitency, they must be serializable.
The example above shows an agent which reacts to events called Event1 and Event2. The `react` method dispatches events to their handler. According to what has been stated in section 2.2.1, the execution of the `react` method is atomic, the agent state is persistent and public attributes are configurable. As we see, event Event3 is sent through a role. This event will be received by all agents which have subscribed to the role.

### 2.3.2 Subscription

The notification of an event is issued by a source agent and then it is transmitted to a listening agent. The source agent creates the event and asks for its delivery using a role. Roles manage the subscription of listening agents and achieve the communication. An event sent through a role will be received by all agents registered to this role. Without going into too much detail, the subscription of agents to other agents is performed with a similar model as the Java Beans[13].

![Subscription procedure](image)

The subscription procedure is simple: a listening agent calls the `addListener` method on the source agent. This `addListener` method sends a subscription event of class `addListenerEvt` to the source agent. The source agent reacts to the subscription event by calling the `addListener` method on the appropriate role. The role's `addListener` method is then finally called to achieve the subscription. The example

```java
public class MyAgent extends Agent {
    int localState = 10; // the agent's state
    Role someRole; // will be set by agent's subscription

    public void reaction1(int i) {
        // do something ...
    }

    public void react(AgentId from, Notification evt) {
        if (((evt instanceof Event1) && condition1)
            reaction1(((Event1) evt).parameter);
        else if (((evt instanceof Event2) && condition2))
            sendTo(someRole, new Event3(...));
            localState += 1;
            ...
        else super.react(from, evt);
    }
}
```
above illustrates the subscription. All events sent through a role will be transmitted to all registered agents.

2.3.3 Creation and Installation of an Agent

We have already mentioned that agents are Java object inheriting from a particular class. Creating an agent starts with the creation of an object considered as a local image of the real agent, called an avatar, which will be installed later. In particular, at the creation of an agent, the information related to the future node of execution must be provided. This local image is then allowed to return an identifier. This AgentId will be used from now on by whoever needs to interact with the agent. The final step is the effective installation on a computer node and the coupling of the local image to the Agent infrastructure, i.e. the MOM. The deploy operation is in charge of this final process. Until the deploy operation is called, the local image is not considered as an agent, thus it cannot react to any event.

The following is an illustration of the creation of a new agent and its deployment on a computer node.

```java
... MyAgentClass theNewAgent; // the agent object, never to be used after its deployment
AgentId theNewAgId; // the agent ID
theNewAgent = new MyAgentClass(nodeNumber1); // creates the object on a node
theNewAgId = theNewAgent.getId(); // retrieves its ID
theNewAgent.deploy(); // installs the Agent on a remote server
...
sendTo(TheNewAgId, someEvent); // later transmission of an event to the newly // created agent
```

3 The Agent Anytime Anywhere (AAA) Runtime architecture

3.1 Overview

The runtime infrastructure is based on a message oriented middleware, also called a software bus which implements message queueing features. Intuitively, message queueing is a technique for sending messages from one program to another by directing messages to a queue as an intermediate storage point. However, the AAA platform provides also the Agent runtime infrastructure, in charge of handling agent execution and the coupling of agents with the MOM.

The basic behaviour is that roles send events using the `sendTo` primitive provided by the bus. Events are encapsulated in messages. The bus forwards messages over the network and stores them in a queue. The bus gets the first message in the queue and calls the agent reaction which handles the event. Agents and the message bus are implemented using Java. Thus, we benefit from the Java Serialisation service to exchange information from node to node, without any heterogeneity problem.

3.2 Inside the Agent infrastructure

3.2.1 Overview

The architecture of the agent runtime can be divided into three main entities: `agent server, Agent and the software bus`.

`Agent Server` : it is a single process that manages a set of agents. Servers can be hosted by the same host or not. All servers are interconnected, they can communicate with each other, thus implementing a part of the software bus. Each server has an engine, implemented within a single thread of execution, which is in charge of activating agents reaction according to the events received.
**Agents**: It is the unit of execution. An agent is a passive Java object whose reaction is activated by the server engine component whenever an event is delivered. This solution can be to some extent constraining because reaction should be time-bound. On the other hand, the implementation of the engine as a single thread is a lightweight execution structure able to manage a large number of agents at a time.

**Software Bus** (Message queue): It is the communication infrastructure, made of representatives contained in each agent server. Each of those are implemented in their own thread of execution, thus handling communication, reliability and the causality properties.

### 3.2.2 Ensuring the infrastructure properties

The agent infrastructure must ensure the atomicity of agent’s reaction, agent persistency and guarantied event delivery.

**Persistency**: Saving agent’s state is achieved by using Java “Serialization”. The engine saves agent state in between each reaction. This is made possible because a single engine controls the execution of every agent reaction of a server.

**Atomicity**: The engine ensure also agent’s reaction atomicity. Intuitively, the engine starts a transaction to run the agent reaction and save it state.

**Guarantied event delivery**: Reliability communication is ensured by the software bus. All events sent during a reaction are added in a persistent queue during a transaction. An outgoing event stays in this queue until it has been transmitted into another persistent queue used to store incoming event on the target agent server side. Event flows through persistent queue is achieved during transaction (see 3.2.2). When a failure occurs, events cannot be lost since there are stored either in the outgoing event persistent queue or in the incoming queue. As soon as communication is reestablished, all pending events are resent if necessary.

### 3.2.3 Agent Servers

The AAA platform is a distributed infrastructure composed by a set of **agent servers**. Several **agent servers** can run on a node. Each agent server contains its set of agents and a “local bus”. Local busses perform event transmission and dispatch incoming events to local agents. The local bus is self-sufficient for local communication (i.e.: event notification involving agents located in the same agent server). However events can be sent to agents hosted by other servers. In this case, the local channel transmits the event via the bus of the distant agent server as shown in the following figure. Notice that all local busses can communicate with each other directly. This distributed infrastructure avoids performance problems due to a centralised communication point.

The local bus structure is composed of a **channel** and an **engine**. Basically the **channel** is responsible for event target localisation and for event transmission over the network. The **engine** dispatches event reception to agents by executing the **react** method of the event target.

A server is included in a process. Two threads of execution are associated to respectively the **engine** and the **channel**. The first one triggers agents reaction according to the received event while the second is in charge of transmitting and receiving events from node to node with the reliability and causal ordering properties.
This implementation choice has some impact. Agents of a same server do not really run concurrently because they are activated one at a time, but the event-reaction model fits well to such an implementation. In addition, the single threaded implementation allows a lightweight execution structure, thus providing atomicity of reaction as well as persistency in a simple and low-cost way. The following program shows the various steps for enacting an agent. Actually, parallelism may be achieved if needed by installing multiple agent servers on a same node and deploying the agents into those servers. This can be done without changing the agents’ program.
The engine’s basic behaviour is to take the first waiting event in the incoming event queue, to run the appropriate reaction on the related agent and to save it state. The engine runs all reaction in a transaction. More precisely, any events sent during a reaction, is stored in a temporary queue. When the react is committed, these events are transfered to the channel persistent queue. At this time, the channel starts another transaction to retrieve the event from the persistent queue and send them through the network.

3.2.4 Channel

The channel is responsible for event transmission to the target agent server. The channel provides causal ordering, communication reliability and message queueing features; On reaction commitment, all events which have been sent during the reaction are guarantied to be delivered despite nodes or network failures and agents unavailability.

![Figure 5 Message Queue Internals](image)

On event notification, the channel encapsulates the event in a persistent message which contains additional logical clock information (required for maintaining causality) and both identification of sender and listener. Then, the channel locates the event target and transmits the message to the distant channel via internal queues ensuring reliable queueing. On incoming event, the distant channel starts a transaction for transmitting the message to the engine. As we saw, this message will be retrieved by the engine in order to achieve the reaction. Internal queues used by the channels can be restored easily if a failure occurs. The ordering of messages in the queues is rebuilt from matrix-clock information stored in the persistent messages. Therefore, messages and causal ordering are not lost despite failures. When a message cannot be sent to its target, a fault is suspected. The channel algorithm attempts to resend periodically the message with increasing period. Furthermore, when an agent server is restarted, an (IsAlive?) broadcast message is sent to all agent servers.
### 3.3 Integration of legacy components

The agent execution model is completely defined by the agent server implementation, in particular user-defined multi-threading must be carefully used in order to avoid liveness problem of the core components: the engine and the message queue. In addition, the only way agents communicate is using the message bus, so communicating with external processes or components should be also carefully done.

Two classes frameworks are included in the agent infrastructure to overcome those issues: the task framework interfacing agents and threads and the proxy framework for creating gateways to external components. Let us detail the latter. The Proxy framework is used to provide an agent: that is a surrogate of the external process. In addition, several classes helps the proxy agent in the adaptation of the external communication according to what the external process handles, and in the insertion of all external messages into the event transmission mechanisms, the message queue.

The **Driver** objects are here to connect message from the **proxy** agent to the communication stream with the external process, but they also receive external message, transform them as event and insert the event in the **Channel** in such a way that persistency and causality is preserved. Drivers can be subclassed according to the external process, but mainly they provide a safe way to integrate external data into the agent server. The objects, **OutputStream** and **InputStream** are the one using the desired communication mechanisms like sockets, pipes, IIOP communication, … Today, several types of **proxies** have been implemented: TCP and UDP socket proxies, File proxies that store and retrieve information from a file, RMI and CORBA proxies querying RMI/CORBA servers and thus recreating synchronous invocation, …

**Figure 6 Reliable delivery algorithm**

**Figure 7 External communication architecture**
4 Evaluation and Performances

4.1 Performance Issues

Measures have been conducted to evaluate the overhead induced by the properties of the agent platform, compared to the lower level mechanisms used to implement the platform. We have also been interested in profiling the time spend in some presumably costly operations, in particular the atomic reaction to events.

The first measures evaluate the percentage cost of each operation on a single machine, when two agents communicate with each other by sending an event. The experiment platform is an Intel’s Pentium Pro 200MHz running Windows NT 4.0 Workstation, with SCSI hard disk. The Agent platform is running on Java JDK 1.1.6. The size of the event is here of 40 bytes of user defined data and the agent size is of 40 bytes of user defined attributes. The following table summarizes the percentage cost of each operation involved in an atomic reaction: start of the transaction, cost of the destruction of the received event from the channel message queue, cost of sending the event to the queue prior to transmission to another agent, cost of the agent state storage prior to transaction commit and cost of the commit operation.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Percentage time of the total reaction execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start of an atomic reaction</td>
<td></td>
</tr>
<tr>
<td>Begin transaction</td>
<td>4%</td>
</tr>
<tr>
<td>Cost of event delivery</td>
<td></td>
</tr>
<tr>
<td>Removing the event from the message queue</td>
<td>6.7%</td>
</tr>
<tr>
<td>Cost of reliable and ordered delivery</td>
<td></td>
</tr>
<tr>
<td>Sending an event to the queue</td>
<td>30%</td>
</tr>
<tr>
<td>⇒ Cost of the matrix clock update</td>
<td>⇒ 45% of above percentage</td>
</tr>
<tr>
<td>⇒ Cost of the event persistent storage on disk</td>
<td>⇒ 55% of above percentage</td>
</tr>
<tr>
<td>Cost of Agent persistency</td>
<td></td>
</tr>
<tr>
<td>Persistent storage of the agent on disk</td>
<td>40%</td>
</tr>
<tr>
<td>⇒ Cost of disk access and Serialization</td>
<td>⇒ 95% of above percentage</td>
</tr>
<tr>
<td>Cost of atomic reaction commitment</td>
<td>18%</td>
</tr>
<tr>
<td>End transaction</td>
<td></td>
</tr>
</tbody>
</table>

Table 1 Profiling of an agent reaction execution time

This table shows the distribution of the cost of a null reaction execution– null reaction corresponds to single transmission of an event to another agent -. It shows that disk access ensuring the agent persistency and the reliable message queueing consumes up to 55% of a reaction time; this is the price of reliable communication and processing.

The following measures are evaluating the impact of the event size on performances, when agents are communicating locally or remotely. Those measurements have been conducted under the same experimental conditions as above, but the Java JDK which was here version 1.1.7. Homogeneous Intel-based machine (Pentium Pro 200MHz) communicate through a 10Mb/s Ethernet local area network.

The first graphs measure the communication and reaction time for an event transmission between two agents, the size of the event being gradually increased. Comparison have been made between remote
and local event communication (graphs *remote* and *single*) to a standard TCP/IP socket based communication. This graph tries to evaluate the overload induced by the reliability and the causal ordering ensured by the agent platform.

![Figure 8: A comparison between TCP and AAA event communication](image)

This figure shows that the cost of a reliable remote event communication through the bus is at least three time greater than a TCP remote communication. This is the cost of the agent infrastructure properties.

A second set of measures has been conducted to compare the cost of a local event communication (graph *single*) to the cost of an agent store on disk plus an event persistent save (*compute graph*). A local event communication should be composed by the serialization of the sender agent as well as the event sent; then their storage on persistent media. Since local communication involves only one agent server, the event deserialization cost can be avoided. Furthermore no TCP communications are involved.
This figure shows that a local event communication is greater from 50 to 100 ms than the cost of agent and event serialization. This overload is mainly due to the matrix clock update required to ensure causality and its persistent storage, and the atomicity of reactions.

The last set of measures evaluate the cost of a remote event communication (graph \textit{remote}) through the bus. This figure shows that the event transmission cost is above the combination of one TCP-based communication plus two event stores and the agent store (graph \textit{compute}) – note that an event transmission corresponds to a first persistent store of the agent in the sender agent message queue, then the commit of the reaction which provokes a storage of the agent, third the effective transmission through TCP sockets and finally a second persistent store of the event on the receiver machine. – This graph shows that a remote event communication is greater than the compute graph. Furthermore, the difference between the remote and the compute graph seems to be independant of the distribution since we have the same difference as in figure 9.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure9.png}
\caption{Evaluation of single node communication with AAA}
\end{figure}
Other measures are still being conducted to evaluate the cost of the causal ordering property. Since the algorithm is based on matrix clock, we assume this should have a bad impact on the scalability of the agent platform. This is why we are exploring ways to interconnect several agent platforms together.

4.2 Real Application Experiments and Lessons learned

Experiments have been undertaken in cooperation with Bull S.A., one of them dealing with network security. Agents are used in a firewall application for various purposes. The first one is the management of logged auditing information. Firewalls must take into account huge log files storage and management. These files spend a lot of disk space while they cannot be delete because of the need of potential post-analysis. The firewall produces a log which will be compressed and analysed by the traffic analysis application. This application allows an administrator to customize the firewall filtering functionalities to produce reduced logfiles.

Agents are used to provide new filtering functionalities to the firewall software, that can be added and customized lately, according to customers requirements. From this experiment, a set of predefined agents for the analysis of audit information were written, then combined in multiple ways according to each customer requirements.
That experiment has shown the interest of the event/reaction agent model was interesting for the processing of a data flow (an event is received to start an analysis process. Then events containing the information are received one after the other, while the agent is processing their content. At the end, according to the result of the analysis, events are transmitted to other agent either to tell a processing has been done, or simply to continue or refine the processing by another agent.) Moreover, because of the anonymous communication between agent and the subscribe/publish mechanism, agents of the same class could be easily used in multiple context by simply customizing their attributes. For instance, a filtering agent can be customized by positioning its attributes value containing the pattern recognition expression for audit information.

The second role of agents is the coordination of multiple firewalls, for example to maintain the consistency of filtering rules, to correlate suspicious actions coming through various firewall. For example, one agent is in charge of spying mail delivery to user of the Intranet. If each agent notices mail deliveries from a same sender and if such an action is detected on the various firewalls, then a coordinator agent asks every involved firewall for the rejection of any request from this sender: in such a case, the sender can be suspected of mail spamming. The modification of filtering rules is also performed by agents to ensure no message loss and the consistency of rules of the firewall set (consistency is preserved because of guaranteed delivery, causal ordering and atomic reaction; once the update of rules is asked, it has to be performed in a particular order).

The reliability property, the transparent distribution, the flexible infrastructure and agent model proved to be a good answer to the enhancement of a legacy system. In the firewall case, part of the agent technology are gradually being shipped in the commercial version of the firewall because it has been considered as a flexible solution to the firewall evolution issue.

5 Related work

MOM have been studied and prototyped in the last decade, often referred to as Software busses[8][7]. They provide a minimal communication semantics between objects, based on untyped information structures and anonymous communication. This is the basis for continuous operation, evolution of applications and integration of legacy components. Information Bus [8] is a perfect representative of such systems but mechanisms have been included to recreate the application semantics on the distributed communication platform: introspection of objects to check type conformity of communicating objects, higher programming effort to interface legacy software with the bus and development of a publish/subscribe mechanism in order to provide binding between service provider and requester. The Java Messaging Service (JMS)[15] interface is the current industrial specification of a software bus behaviour. Additional properties to communication, such as reliable or guaranteed
delivery of messages, have also been inserted in such platforms either using specific network protocol [7] or message queueing features [4]. But the lack of a distributed programming model enabling flexible and location independent objects has always been the reason MOM are used only as an interconnection means, and not a real reliable distributed programming environment. Some attempts have been made with Java InfoBus[14], where objects are programmed with the Java Beans model, but the bus is neither distributed nor reliable.

6 Conclusion

This paper presents a distributed programming model for asynchronous active objects and an implementation of distributed communication infrastructure based on message passing features. The combination of both provides a flexible and reliable environment for distributed application execution.

The communication infrastructure is based on a Message Oriented Middleware that provides asynchronous, reliable and guaranteed transmission of messages. The implementation is based on a message queueing system in order to resist to network or server crash or failure. In addition, an ordering property has been made available in order to facilitate the programming of distributed entities.

The distributed programming model of the software entities, called agents, is based on asynchronous event passing and atomic reaction to those events. At the execution level, agents are tightly connected to the message oriented middleware in order to benefit from its properties, while others are provided like persistency, flexible interconnection through a publish and subscribe mechanism, remote creation and deployment and migration as well as other dynamic reconfiguration features.

This platform has been used with a particular application dealing with network security. This experiment has shown the efficiency of the agent model and the flexibility of the agent platform in terms of extension and interconnection of a legacy system, in order to change a monolithic single node application into an extensible distributed one. In addition, the experiment system had strong requirements concerning reliable interactions, which is one of the main interest of the presented agent platform.

There are still work to conduct around this project, the main one being the enhancement of performances and the other one the scalability of the system. One of the property of the platform is to maintain causal deliveries of message in a distributed environment, and matrix-clock based algorithm have been accordingly used. However, those algorithms are known to be poorly scalable (talking of 1000 of nodes) which requires thinking of interconnection solutions between clusters of agent servers. Finally, there are also ongoing work around construction environments of agent-based application[2], which combined with the presented technology, will provide a complete distributed programming and execution solution for asynchronous and reliable applications.

Acknowledgment

The authors would like to thank the members of the SIRAC team for their support and friendship. A special thank is addressed to Roland Balter, Jacques Mossiere and Michel Rivell for their comments on versions of this paper. This work is supported by the EC under the C3DS ESPRIT LTR 24962 project.

7 References


