Table of Contents

JIT'99

II

Evaluation of Java Messaging Middleware as a Platform for Software Agent Communication

Frank Kargl, Torsten Illmann, and Michael Weber

Distributed Systems Department, University of Ulm, 89081 Ulm, Germany
frank.kargl@informatik.uni-ulm.de
torsten.illmann@informatik.uni-ulm.de
weber@informatik.uni-ulm.de
WWW home page: http://www-vs.informatik.uni-ulm.de/

Abstract. In this document we introduce an infrastructure for personal agent communication and coordination. An essential part is the so called AgentBus, built on top of existing messaging systems that allows flexible communication between agents. We show how messaging differs from other communication mechanisms and describe our evaluation of several Java messaging systems like JMS, Corba Event and Notification Service or Softwired's iBus with respect to functionality and performance. We also describe the special requirements of agent communication and the design and performance of our AgentBus.

1 Introduction: The CIA Project

Our research group is currently studying different aspects of software agent systems in a project called CIA [1]. We are developing an infrastructure where software agents can easily be integrated. Java 2 is the platform for all our prototype implementations. In this project all agents belonging to one user form a so called AgentCluster. This cluster supports the agents with all kinds of commonly needed services. For all communication agents use the so called AgentBus that is implemented on top of an exchangeable messaging system. Therefore we have evaluated and tested different Java messaging systems.

2 Agent Communication

When we talk about agent communication, we find a lot of diverse communication patterns. Agents communicate with their users, with other agents (belonging to the own or another user), with services found at various places etc. We will first give an overview over different communication models in general and messaging in particular.

2.1 Messaging Oriented Middleware

"Traditional networking systems" like many Internet services (e.g. WWW) are build upon the client-server paradigm. A dedicated server offers services under

a specific address. A client uses these services typically by sending a request to the server and yielding an appropriate response. These systems use a more or less direct addressing scheme. Peers or servers are contacted either by their address (e.g. 134.60.240.13) or by a indirect addressing using statically bound names (e.g. www.uni-ulm.de). Application specific protocols (e.g. http) are used for communication. Resolution mechanisms like portmappers or more dynamic name-servers (e.g. CORBA name-service) don't change the direct connection between the peers or the client and the server.

Sometimes it is desirable to decouple this strict relationship. Especially in an object-oriented environment with dynamic communication patterns, you can use so called *Message Oriented Middleware* (MOM). Typical applications that use MOM are characterized as follows [2]:

- Multiple client applications may be interested in the same object-initiated information.
- It is better to make the data objects actively share their information because clients may come and go dynamically.
- The objects can not afford to suspend execution while a given message is being transmitted to each interested party.

These messaging systems are peer-to-peer facilities where clients can send messages to and receive messages from any client. Clients connect to messaging agents that support creation, sending and reception of messages. Each system provides a way of addressing messages. When using MOM there are typically two messaging styles offered [3]:

- *Point-to-Point* (PTP) systems use message queues that are associated with specific clients. Messages are addressed to queues. Clients extract messages from their queues.
- In a *Publish/Subscribe* (Pub/Sub) system clients take the role of either Publishers (Producers) or Subscribers (Consumers). Publishers send their messages to some named entity (e.g. a channel) from which clients can extract them. Most systems are capable of broadcasting or multicasting a message to many destinations at once (see Figure 1).

If messages are delivered *asynchronously* to clients as they arrive, we speak of *push* communication. If a client must (*synchronously*) request each message, it is called *pull* communication. Sometimes an immediate response to a message sent by one client is expected. Some messaging systems implement this as a so called synchronous request-reply communication which is similar to the communication mechanisms in traditional client server systems while still preserving the other advantages of a messaging system (no direct addressing etc.)

2.2 Requirements for Agent Communication

We have identified various aspects of agent communication that suggest the usage a messaging system as a base for a software agent communication infrastructure. We use these criteria as guidelines in our evaluation of different products:

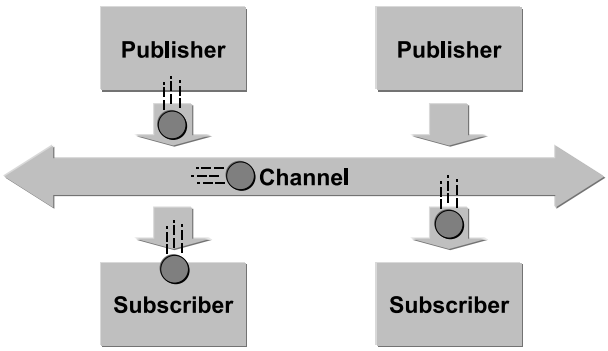

Fig. 1. Publish/Subscribe Model

- **Location transparency**. The composition of agents at one place may vary frequently. Mobile agents may come or go and certain agents may be started or shutdown by their user. Thus a mediating middleware is preferable above direct addressing.
- **Topic addressing**. An agent can not know which other agents are interested in the information the agent wants to make public or which other agents can respond to its request. Thus an addressing style based on topically named communication channels and not on single agents is preferable.
- **Message filtering**. Furthermore not all agents want to interprete all messages. So some kind of filtering mechanism should be realized directly within the messaging system.
- **Persistent messages**. Mobile agents or agents hosted on mobile systems like notebooks or PDAs may not always have a direct connection to the messaging system. Thus some kind of disconnected operation mode is needed where important messages are stored in a persistent manner within the messaging system and are delivered to the agents whenever they reconnect.
- **Quality-of-Service**. Not all messages have the same importance and therefor different qualities of service need to be implemented. Some messages e.g. may need high throughput but only few reliability whereas others have to be delivered with an exactly-once semantic.
- **Timing constraints**. Often an information has a certain lifetime. It is only valid after some initial date and no longer than it's expiration date. Such and similar timing constraints should be respected by the messaging system.
- **Secure communication**. Agents often deal with sensitive information about their user (like credit card numbers etc.). Secure and confidential communication should be implemented directly within the agent communication infrastructure.
- **Distributed architecture**. For reasons of scalability and resilience, the messaging system should work without any central components, like message dispatchers or naming services.or at least these components should work redundantly.

– **Portability** As we want to use different platforms for our system, ranging from PDAs to PCs and workstations, the messaging system must be platform independent or easily portable.

Because of the last item and various other reasons a first decision was to use Java for our implementation prototypes. Next we have evaluated and tested common messaging systems for Java.

3 Messaging Systems for Java

3.1 Java Message Service

Sun has specified a special Messaging API called *Java Message Service* (JMS). The current version is 1.0.1 [3] . Sun plans to use JMS as the standard mechanism for asynchronous bean invocation with EJB. JMS provides a common way for Java programs to create, send, receive and read messages from various messaging systems. JMS therefore defines a common set of enterprise messaging concepts and facilities. These concepts are implemented by a specified messaging product and may be accessed using so called JMS Providers. Often these are written in 100% pure java and applications using JMS are thus portable among a wide range of platforms.

JMS has two messaging domains: the PTP and the Pub/Sub domain. JMS-compliant applications can only be ported directly across different JMS providers within their communication domain. Messages are produced by a MessageProducer and consumed by MessageConsumers. A JMS messages consist of a header and a body part. The header contains administrative information (Destination of message etc.) as well as simple QoS requirements (expiration dates, priority etc.). In addition to predefined header fields messages may contain user defined properties. Using Message Selectors, which are some kind of search expressions based on these properties, users can identify to the system what messages exactly they are interested in. The Body of a JMS message may be of type StreamMessage, MapMessage, TextMessage, ObjectMessage or ByteMessage, containing either a stream of Java primitive values, an associative map of name/value pairs, a `java.lang.String`, any serializable Java object or a stream of uninterpreted bytes. Many vendors have announced or already implemented JMS support for their messaging systems. We have evaluated one JMS implementation called FioranoEMS 3.1 [8].

Requirements: In respect to the requirements from the last chapter, JMS delivers topic based addressing, but no further filtering capabilities. Quality-of-Service and Security are not within the scope of the JMS API but JMS providers may implement their own proprietary extensions. JMS supports so called persistent messages with expiration dates and a transaction concept. As they are pure java JMS providers available, portability is good. Although JMS is only a specification and many different implementations are possible, many JMS providers (like Fiorano) use central components as dispatchers. Redundancy is possible nevertheless.

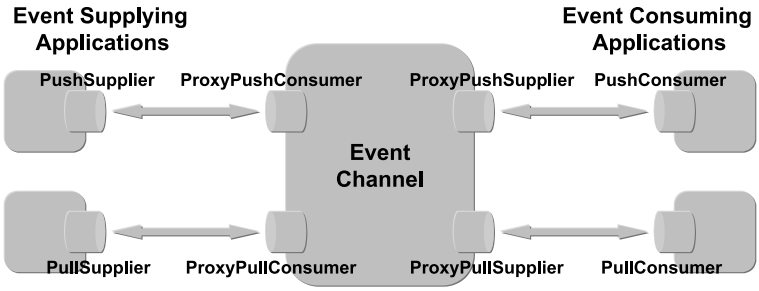

Fig. 2. CORBA Event Channel

3.2 CORBA Messaging

The basic CORBA mechanism provides for synchronous execution of operations within remote objects (or remote method invocation). CORBA uses direct addressing via object references and is no messaging system in the sense used above. Nevertheless the OMG [4] has specified two additional services for CORBA that implement messaging systems: the **CORBA Event Service** [6] and the **CORBA Notification Service** [5].

CORBA allows a special kind of communication called event-style. When using event-style communication in push style, event-supplying applications send events to event-consumers by invoking a push operation on the latter, passing the event as an operation parameter. Pull style communication is similar, except for the consuming application invokes a pull operation on the supplier, which will return an event if one is available.

Although the mechanism described above provides basic support for event-style communication it does not support decoupled, asynchronous, multicast communication. However it is used to define an intermediary agent known as the Event Channel that satisfies these requirements (see Figure 2). An Event Channel is a standard CORBA object; communication with the Event Channel takes place using standard CORBA requests. It supplies platform and language independent, mixed push/pull, many-to-many communication within CORBA. In most implementations the Event Channel is implemented as a central dispatcher object, so scalability problems are likely to occur.

The **CORBA Event Service** suffers from some deficiencies [2]:

- It has no filtering capabilities. This may lead to heavy load or congestion as every consumer connected to a given channel receives a copy of every event delivered to the channel.
- Clients may want to specify their different quality of service requirements like fast, best-effort versus slower, guaranteed delivery.

So the OMG issued a Request for Proposal for an enhanced successor to the Event Service. The result is called **CORBA Notification Service** and is back-

ward compatible with the original Event Service. It has two major improvements to overcome the named deficiencies:

- Clients can associate a set of filters with proxy objects. The proxy object will forward only those events that match at least one of the constraints associated with at least one of its filters.
- Clients can define quality of service requirements on a per event, a per proxy or a per channel base.

Quality of Service parameters that can be specified include reliability, priority, time constraints and user defined properties. Additional features of the Notification Service enhance its scalability, performance and usability. E.g. channels may inform event suppliers about the types of event in which consumers have interest in receiving and vice versa channels can inform consumers about types of events that suppliers intent to emit. In addition, the service defines special transactional proxies for transactional event transmission and a repository for the definition of application specific event types.

Requirements: The Event/Notification Service fulfills most of our requirements for agent communication. Topic based addressing can be realized using one channel object per topic. Filtering and Quality-of-Service were added with the Notification Service. Security may be realized by using IIOP via SSL connections. As CORBA is one of the leading industry standards many implementations (incl. pure Java solutions) are available. Event/Notification Services are often implemented as single dispatcher objects which has negative impact on scalability and resilience.

3.3 Softwired's iBus

iBus (current version 2.0.1) by Softwired Inc. [7] is a 100% pure Java messaging system. It provides publish/subscribe style of communication transmitting any kind of serializable Java objects. Events are sent either using IP multicast (many-to-many communication) or TCP PTP connections. In normal operation iBus provides asynchronous push of events using different channels named by URIs. There is also a fault tolerant multicast request/reply mechanism that allows synchronous request/reply communication. A request by one client is delivered to all potential repliers. After return the client gets an array of all responses supplied.

iBus has a flexible Quality of Service framework with an extensible protocol stack, allowing each application to tailor the QoS exactly to their needs. Applications can even supply their own stack modules implementing new and unexpected QoS characteristics. Applications may e.g. decide what kind of acknowledge mechanism they want to use (positive, negative, none at all), they can include a crypt stack module for online encryption of all iBus traffic or they can replace the IP multicast module with an ATM multicast module. iBus has no central components (like a naming service) and there is no central performance bottleneck. Clients can join or leave channels anytime and at any place within

an IP multicast Intra-/Internet enabling what Softwired calls spontaneous networking.

Requirements: Most of our requirements are addressed with iBus. Topic based addressing is done using named channels. Flexible Quality of Service requirements are handled by the dynamic stack framework, although some aspects like timing constraints or security aspects aren't handled by the provided stack modules. Message persistence is announced as a separate product. As iBus is a pure java solution, portability is good.

4 CIA AgentBus

The CIA AgentBus is the central (and only) communication mechanism in our agent infrastructure. It may reside on top of any of the above messaging systems.

It consists of several agent channels that provide a topic based information exchange between clients. Agents, channels and the whole AgentBus are named using a URL-like notation. An example may be:

`cia:<qos>://frank.kargl@de:<pid>/dates/business/diary-agent`

You can specify optional QoS parameters, your name, country and the optional personal id to identify your personal Agent Cluster. `/dates/business` denotes the name of a channel for exchanging date information. Finally `diary-agent` is the name of a specific agent communicating via this channel.

The primary design principal was to keep usage of the AgentBus by agent programmers as easy as possible. Agent programmers should focus on writing good agents and not on dealing with complex communication systems. For communication agents simply establish new AgentChannels or join existing ones by creating a new AgentChannel object with a specified AgentURL. The AgentBus knows three communication mechanisms:

- Asynchronous ChannelEvents that are directed to a channel and are seen by all consumers on this channel.
- Asynchronous ChannelMessages that are addressed to a specific agent on a channel and that are delivered only to this agent.
- Synchronous multicast request/reply. A client sends his request to a channel where a number of repliers may process it. Each replier may return a result. All supplied results are delivered to the client as an array.

Any of these mechanisms may be used in parallel within the same channel. The following code example illustrates how easy channels are created/joined and events are sent:

```
// create new AgentBusFactory using ibus implementation
AgentBusFactory myABF = new AgentBusFactory("ibus");
// create new AgentURL
AgentURL myAURL =
    new AgentURL("cia://frank.kargl@de/dates/business/diary-agent");
// create new Channel
```

```
AgentChannel myAC = myABF.newChannel(myAURL);
// create new Message
ChannelEvent myCE = new ChannelEvent("Test Message");
// send event
myAC.sendEvent(myCE);
```

Reception of events works via event handlers. Messages and Requests/Replies are used analogous. There's also a mechanism for event persistence. A special agent associated with each channel records all events for a specified lifetime and may retransmit them to any agents that join this channel at a later point in time. Encryption, reliability etc. may be encoded in the QoS specification in the AgentURL.

After comparing the different messaging systems described above we decided to use iBus as a first platform for implementing the AgentChannel. In fact the AgentBus design is partly influenced by the iBus architecture. We think iBus is a very lean and portable (100% java) concept with good built-in capabilities. More importantly we are able to integrate own features like new security mechanisms or new communication patterns using custom stack modules. As we don't want to depend on a single messaging system the AgentBus totally wraps all specific aspects of iBus. There is a factory for creating new AgentBus object instances that can work with any other implementation. We plan to implement at least three other alternatives for comparison: one based on the CORBA Notification Service, one JMS based solution and a completely independent implementation of a messaging system based on an ATM network that allows advanced QoS applications like video conferencing.

5 Performance Comparison

This chapter provides results of performance measurements with implementations of a Corba Event Channel (Visibroker for Java 3.4), iBus 2.0.1, JMS (Fiorano EMS 3.1) and our AgentBus. We have not tested a Corba Notification Service for availability reasons. Our AgentBus is included into the tests because we want to measure the produced overhead compared to the iBus implementation. Since iBus allows to specify different QOS functionality, it is tested in a reliable and an unreliable case. The test environment allows to send events from one machine to another using the different services. It is done on a 10 Mbit Ethernet dedicated LAN with a PC 300 MHz PII as sender and a PC 366 MHz PII as receiver. In case of Visibroker and Fiorano EMS where a central dispatcher is required, we use an extra PC 350 MHz PII.

The first scenario computes the average transmission time of different-sized events. In that scenario, the event size is varied from 1 kB to 1 MB while the number of transmitted events remains constantly 500. The result illustrated in Figure 3 shows that in all cases the average transmission time increases with the size of events. The best results are provided by the unreliable iBus implementation. The reliable iBus resides on the second place. Here, iBus profits from not having a central dispatcher as the others. Comparing the others, the Fiorano

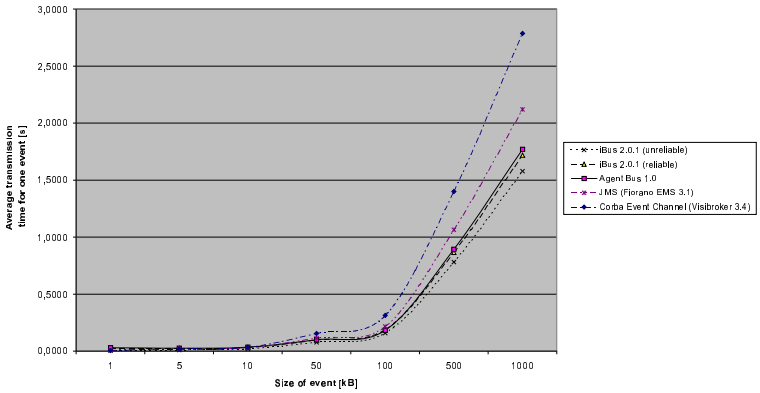

Fig. 3. Average transmission time of different-sized events

EMS System returns better results than Visibroker. As expected, the transmission time of AgentBus events is insignificant higher than reliable iBus events since the test is based on top of the reliable iBus.

In the second test scenario, we measure the bandwidth reached during the transmission of events whereas the number of sent events varies from 10 to 10000 and the event size remains constantly 10kB. The results illustrated in Figure 4 show that the reached bandwidth depends on having an extra dispatcher or not. The iBus and AgentBus only reach about 200 kB/s bandwidth for a small number of events whereas Visibroker and Fiorano EMS constantly return results between 300 kB/s and 350 kB/s. When raising the number of events higher than 5000 the results change. The iBus (and respectively AgentBus) reaches results around 500 kB/s whereas a maximum of about 810 kB/s was reached without using a messaging middleware at all. Visibroker and Fiorano EMS do not depend on the size of transmitted events. This is probably due to the dispatcher (Corba Event Channel or JMS kernel) which caches the results and dispatches them with a constant bandwidth.

Furthermore, we tested a small multicast scenario with one sender and two receivers. We recognized that the results of iBus improved due to its usage of IP multicast. But since the performance results do not improve significantly we do not present them here. To summerize the measurements, our results show that the iBus implementation is well scalable for either a lot of traffic or large data items. Visibroker and Fiorano JMS returns good results for smaller systems.

6 Conclusion & Outlook

Messaging Systems are a new and exciting way to realize new forms of networking in a very dynamic manner. Especially when used with software agents this

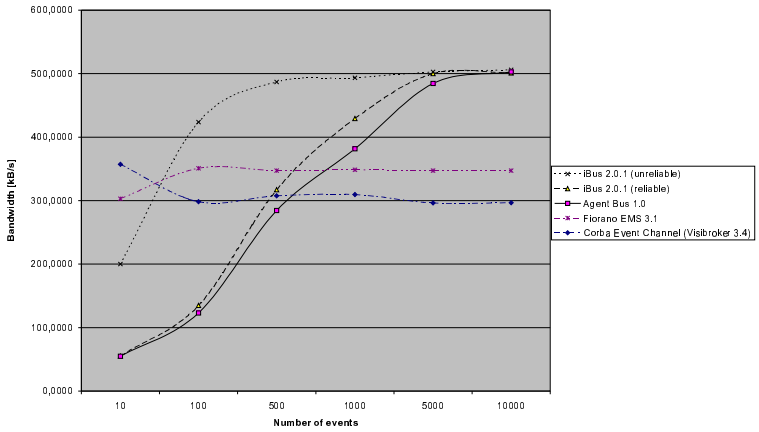

Fig. 4. Bandwidth of event transmission

will open possibilities for new applications that don't need any configuration for finding their communication partners. Quality of Service, Encryption etc. should be integrated directly into the middleware without changing the agents or applications.

With the CIA Agent Bus we have designed a powerful yet easy to use messaging system that is esp. suitable for agent communication. As it may reside on virtually any available Java messaging solution, we are very flexible in choosing and comparing different products. As we have demonstrated with our tests, our current base, iBus, can be adapted to different needs (performance vs. reliability) very well. Our future work will integrate more messaging products like Corba or JMS into the AgentBus. On the other hand we will add functionality like security or resilience features to the AgentBus.

References

1. Kargl, Illmann, Weber: CIA - a Collaboration and Coordination Infrastructure for Personal Agents. DAIS 99, Juni-July 1999, Helsinki, Finland
2. Notification White Paper. IONA Technologies PLC, 1998, Dublin, Ireland
3. Hapner, Burridge, Sharma: Java Message Service, Version 1.0.1. Sun Microsystems Inc., JavaSoft, 1998, MointainView, USA
4. Object Management Group. `http://www.omg.org/`
5. Telefonica, Hewlett-Packard: Joint submission to Notification Service RPC. 1998
6. CORBAservices: Common Object Services Specification, 4. EventService Specification. OMG, 1997, Framingham, USA
7. Softwired Inc. `http://www.softwired-inc.com/`
8. Fiorano Software, Inc. `http://www.fiorano.com/`