1998

Message Patterns in the Bond Distributed Object System

Ladislau Bölöni

Kyung-Koo Jun

Thomas Danials

Dan C. Marinescu

Report Number:
98-004

MESSAGE PATTERNS IN THE BOND DISTRIBUTED OBJECT SYSTEM

Ladislau Boloni
Kyung-Koo Jun
Thomas Daniels
Dan C. Marinescu

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD-TR #98-004
March 1998

Message patterns in the Bond Distributed Object System

Ladislau Bölöni, Kyung-Koo Jun, Thomas Daniels and Dan C. Marinescu
(Email: boloni, junkk, daniels, dcm@cs.purdue.edu)
Computer Sciences Department
Purdue University
West Lafayette, IN, 47907, USA

March 6, 1998

Abstract

The Bond distributed object system is built on a message oriented structure, using KQML as messaging language. KQML messages in the Bond system are grouped into closed subsets called *subprotocols*. The various semantic relationships between objects (client-server, monitoring, event handling, cooperation) are handled at the application level by building *message patterns*. A message pattern specifies the flow of messages and the specific performatives needed to implement a particular relationship.

The peculiarity which distinguish the Bond object system from other KQML implementations is the fact that in Bond every object, even passive ones can be destinations and sources of messages.

1 Introduction

The Bond distributed object system is built on a message oriented structure, using KQML as messaging language. Bond objects are network objects in the sense that they can communicate with each other, can be instantiated and run remotely. A more detailed presentation of the Bond objects is presented in [3].

This paper presents the mechanisms used by the Bond system to implement the basic functionality of a distributed object system. Being a message oriented system, the functionality is implemented by loosely coupled objects exchanging messages in the KQML format.

KQML messages in the Bond system are grouped into closed subsets called *subprotocols*. The basic notions of communication using messages is presented in Section 2 The various semantic relationships between objects (client-server, monitoring, event handling, cooperation) are handled at the application level by building *message patterns*. A message pattern specifies the flow of messages and the specific performatives needed to implement a particular relationship. Message patterns are presented in Section 3.

In modern distributed systems an executable can be part of multiple relationships at once: can be server in relation with an object, a client for another and a source of events for yet another object. Section 4 presents the mechanism of *waiting slots* provided by the Bond system to facilitate handling multiple message patterns at once.

In Section 5 a side by side comparision presented between the message oriented approach of Bond and the remote method call approach of Corba.

2 Object communication using KQML

The KQML agent communication language was initially designed for communication between intelligent agents. KQML is a metalanguage in the sense that it does not specify the content of the communication but allows agents to express an attitude relative to the content of communication. KQML messages, called performatives allow to encode basic abstractions like asking, replying, achieveing, subscribing or notifying while the content of the messages are partially encoded in the parameters, and partially assuming to be known by both parties of communication, by having access to a common ontology. In the Bond system we use KQML in a somewhat different way than it was originally designed: instead of being the language of the agents, in the Bond system every object (even small internal objects) understand KQML.

In Bond an object in the working memory is said to be active. Only objects in the secondary storage are considered passive and need to be activated before receiving messages.

Sometimes we are asked if this decision does not have a too big overhead on Bond objects. The reason is the misconception that only an object with an active thread can receive messages. As programmers are used to call methods on objects thereby activating them when needed, in the same way there is only one active messaging thread in a Bond executable, which can activate any object by sending it a message. Likewise, there is only one instance of the KQML parser which is called by the messaging thread, so

the objects receive the messages in the parsed form.

If an incoming message requires intense processing extended over a long time a new thread is created for the object or the objects existing thread is instructed to carry out the required processing.

Although all objects understand the syntax of KQML, their level of semantic understanding depends on their type. Different objects can carry out different conversations - a scheduling agent understands different messages than a database server.

For a remote method call system like Corba, the question is what procedures, with what parameters can I call on the remote object? For a message oriented system, like Bond, the question is what kind of messages the remote object understands?. The solution of Corba is the description of the interface to the objects by the IDL language for the static case or the dynamic interface repositories. The Bond solution is based on subprotocols.

A subprotocol is a closed subset of the KQML language, used to implement a specific functionality. Bond objects can implement a number of subprotocols, the only restriction being that if an object implements a subprotocol, it should implement it completely. Some examples of the more important subprotocols in the Bond system are presented in Table 2.

Generally speaking every Bond object implements all the subprotocols implemented above it in the Bond object hierarchy. Every Bond object implements the property access subprotocol. Because the subprotocols implemented by the object is a property of the object, it can be interogated using the property access subprotocol. If two objects need to communicate they first have to establish a common language, the subprotocol both understand. For this end they use the property access subprotocol, which is the minimal common ground of communication. Then, the two objects can communicate using the subprotocols which are implemented by both of them.

An important exception is the interface discovery subprotocol, which permits an object to learn the syntax and the semantics of an unknown subprotocol.

Interface discovery may be used by agents to learn to communicate with agents which provide a known service, but with a different way. A good example would be a user which can connect to the internet using a PPP or an ISDN service. The service (connection to the internet) is the same, however both service providers need a different set of commands and information in order to provide the service. In this situation the service provider can present its service access subprotocol to the user agent, together with the

Subprotocol	Who implements it	Function
Property access	All Bond objects	Supports read/write access to all properties of a Bond object.
Security	Some Bond objects	Used to establish trust relationship between Bond objects.
Monitoring	All Bond executables	Allows a Monitor agent to obtain information about the current state of the executable.
Checkpoint/restart	Some Bond executables	Supports checkpoint/restart facilities
Agent control	All Bond agents	Allows a Bond object to start, stop and control a remote agent.
Interface discovery	Some Bond objects	Allows an object to discover the syntax and the semantics of an unknown subprotocol implemented by another object.
Database access	All database servers and some Bond objects which need database access	Supports creation, deletion and updating of objects from databases.
Scheduling	All scheduling agents	Supports scheduling of a contract
User access	All wrappers	Allows the External agent to control the execution of a legacy program running under the control of a wrapper.

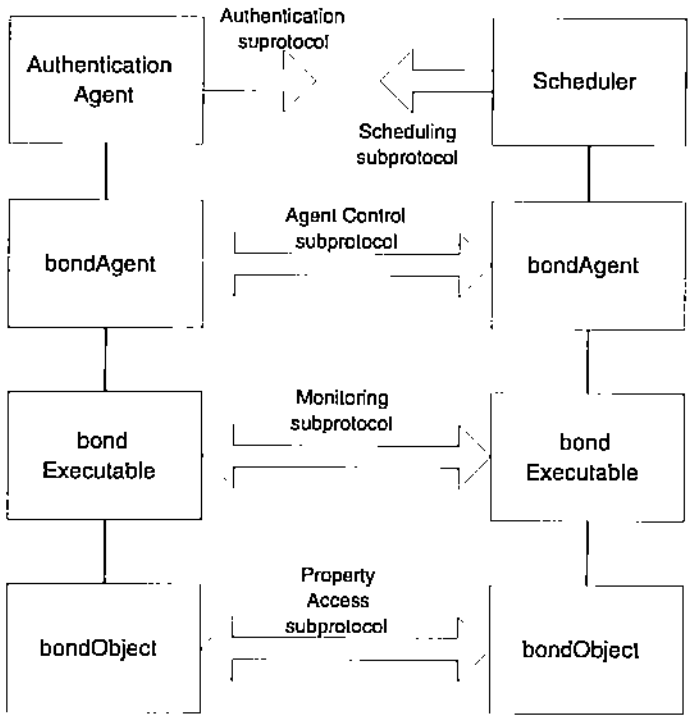

Figure 1: Two different agents can communicate by using as a common ground the subprotocols implemented by both of them

semantics of the messages in the subprotocol. The semantics specifies how the state of the service provider can be modified by different messages and how the changes in the state of the provider are mirrored in the messages it sends to the user. Knowing the syntax and the semantics of the subprotocol, the user agent can select the messages needed to be sent in order to bring the provider in the disered state - for example "Connected" in our case.

3 Message patterns

The operation of a distributed system can be divided into tasks. Examples of tasks are "logging in", "running a program", "calling a function" or "deleting a file". Usually in a distributed system multiple tasks are executed simultaneously. The Bond approach for implementing tasks are the *message patterns*.

A message pattern is a logical sequence of messages between a number of network objects, performed in order to accomplish a task. The messages are the control element of the task, while the processing or data transfer is performed by the objects.

In the following we present a number of message patterns for the most common operations that occur in the Bond distributed system. At this level of abstraction we are concerned only with the control flow of the task. Our assumption is that the objects involved understand the corresponding messages, and are capable and willing to execute the commands.

3.1 Remote procedure call

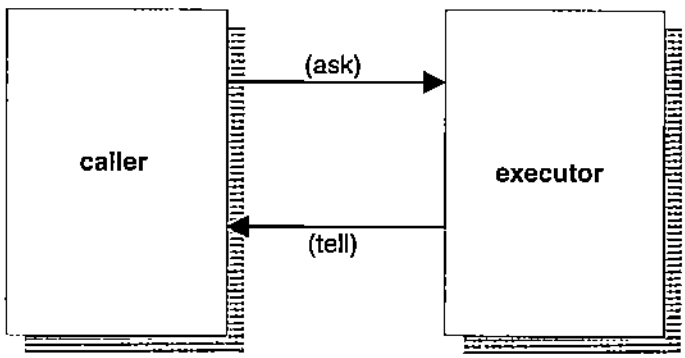

Figure 2: Message pattern for a remote procedure call

The most simple message pattern is used for calling a procedure on a remote site. This is functionally equivalent to the classical RPC protocol. The message pattern for a remote procedure call is presented in Figure 2. The caller sends an ask message with the name of the called function and the function parameters as the parameters of the performative. The destination of the message is the remote agent, which will perform the function call and return the result to the caller.

3.2 Remote method call

Remote method call is used to call a specific method on a remote object. This is functionally equivalent to remote method calls in the Corba or RMI sense. The message pattern is presented in Figure 3. Unlike the remote procedure call case the destination of the message is directly the remote

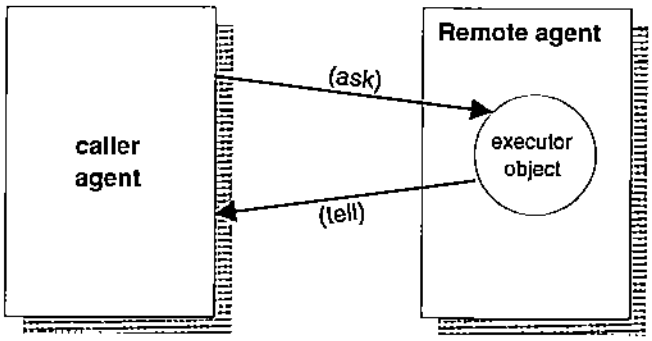

Figure 3: Message pattern for a remote method call

object. We can observe that the messaging abilitity of any object in Bond is a central feature in implementing this pattern.

In a message oriented system like Bond, a relationship like this is usually considered to be a *question/answer* relation instead of a *method call/reply.* This is more a linguistic issue, in pure object oriented languages like Smalltalk, methods are called messages. However in the case of Bond, there is an important advantage that the question/answer relationship is asynchronous by default. After sending a question, the object can continue it's processing and the reply is distributed by the messaging thread to the corresponding *reply waiting* slot (see section 4.2 for details).

3.3 Event handling

Events are an important subclass of messages. Events are generated by an event generator object, and are consumed by an event consumer object (event listener in the Java terminology). While in Java the Event class is used to report user interface events like keypress or mouse click, in the Bond system the events are course grain network events. One important application of events is monitoring agents.

The message pattern for event handling is presented in Figure 4. The event consumer sends a `(subscribe)` performative to the event producer, subscribeing for specific kind of events it is interested in. The event producer sends the events embedded in `(tell)` performatives. There is no reply to the event messages. When the event consumer is not interested in the specific type of event, sends an `(discard)` performative to the even producer.

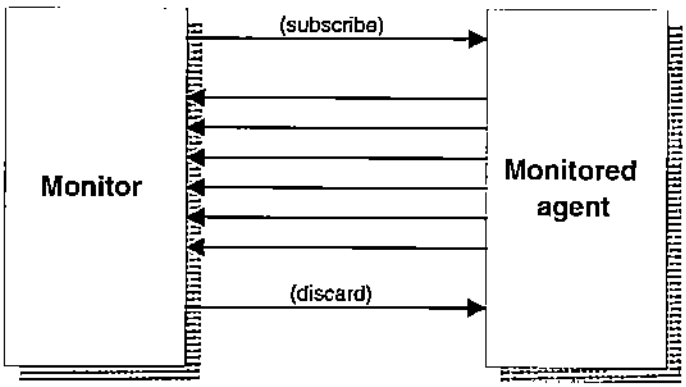

Figure 4: Message pattern for event handling

3.4 Client server relationship

Client server relationship is one of the most important semantic relationship used in today's network systems. The message pattern for the client server relationship is presented in Figure 5. This pattern presents the case of a multithreaded server. In the unlikely case that a single thread server is needed, the message pattern will be similar to Figure 2.

The assumption in this case is that a client sends an (ask) performative to the main thread of the server. If the request is valid, the server will start a new thread for the request, which will handle the subsequent requests of the client. Alternatively the server thread may deny the service by sending a (deny) performative. In this case no new thread will be created.

3.5 Global directory, search

The directory service is one of the basic features of the Bond object system. It's importance is even larger because the Bond system in general does not keep distributed states, and the consistency of the system is based on the fact that agents can find again the lost objects.

The message pattern for the directory service is presented in Figure 6. When an object wants to find an object sends a request to the local directory of the executable. If the requested object is found locally, the local directory builds the shadow of the object. If the requested object is not on the local directory, the local directory sends the request to the central directory. The central directory sends forward

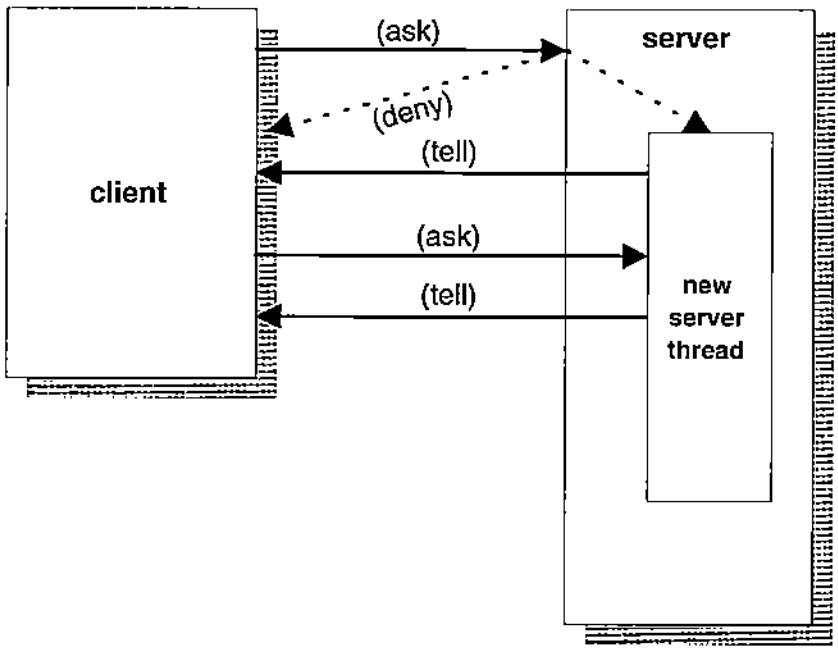

Figure 5: Message pattern for a client server relationship. For each new connection a serving thread is created inside the server and the original request is passed to the new thread

3.6 Searching for services

Searching for services are handled in the Bond system by the same `find()` interface used for finding individual objects. Nevertheless the message pattern is different, because there may be number of objects on the network which can provide the required service. It is the task of a *broker agent* to select from the possible objects who provide the service the object which will provide the service.

The message pattern is presented in Figure 7. The object which requests a service sends a request to the local directory, which transmits it to the global directory, which broadcasts the message to the remote agents. The agents which can provide the required service send their response messages to the broker agent which selects the agent which ultimately provides the service to the requester.

This message pattern is just one way to implement the directory service, suitable for small to medium size systems. For large systems the directory

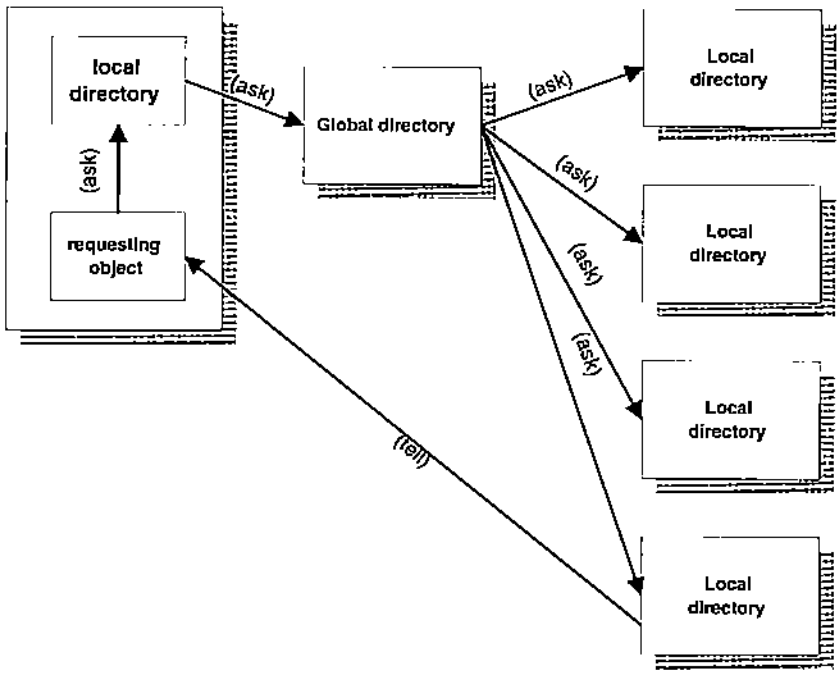

Figure 6: Message pattern for the global directory services - searching for objects

service may be in itself distributed and/or replicated and a different message pattern employed.

3.7 Authentication - creating a trust relationship

A frequent task in a distributed environment is creating a trust relationship. Figure 8 presents a message pattern where an agent presents itself to the controlled agent and an authentication procedure is performed in order to ensure the identity of the agent.

The authentification assumes the existence of a trusted authentication agent. The agent generates a random message and sends a message to the remote agent asking to encrypt it with it's private key. The agent replies with the encrypted message. The encrypted message is sent by the agent to the authentication agent, requesting to decrypt the message according to the assumed identity of the remote agent. The authentication agents reply is compared with the randomly generated message. If they are identical the

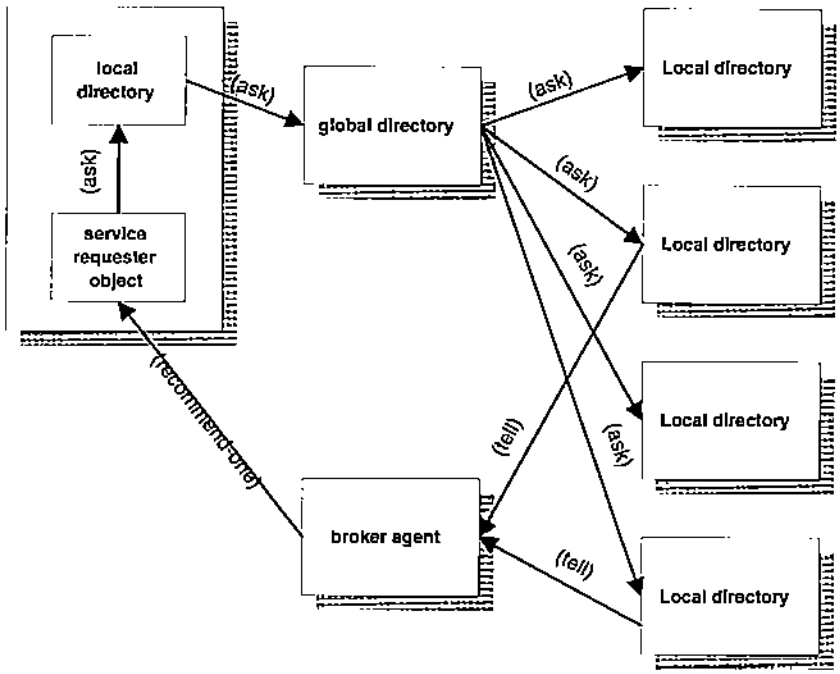

Figure 7: Message pattern for the global directory services - searching for services

trust relationship is established.

This authentication protocol is rather simple, and has some limitations (the existence of the trusted authentication agent being the most important). One of the important advantages is that it doesn't require the agent to have a key, nor encrypting capabilities. On a typical Bond system, most objects have no keys and most messages are travelling on non-trusted channels. It is the decision of the system administrator to set the security to the level required by the application.

3.8 Remote start of a trusted agent

In the previous section we have presented how a trusted relationship can be established between two agents, and we saw that the trusted agents should have a key. This raises the *key distribution problem*, which in the terms of the Bond system can be put in the following way: an agent should acquire a public/private key pair, and it should publish his public key together with

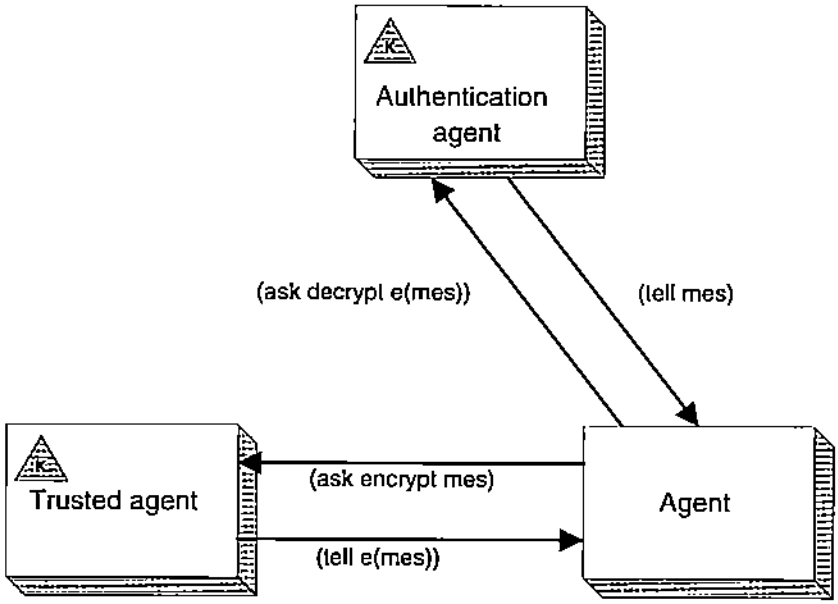

Figure 8: Authentication message pattern

it's identity. The message pattern of one of the possible solutions is presented in Figure 9.

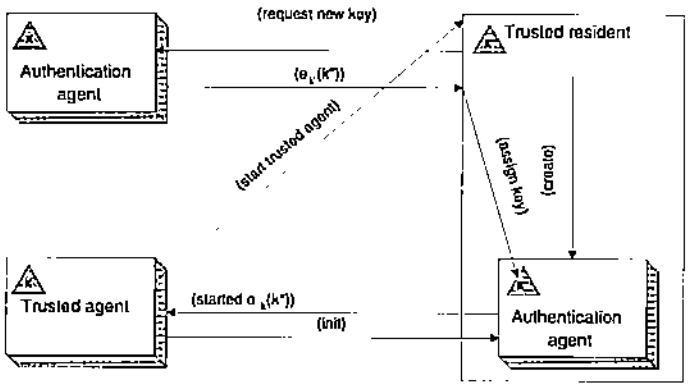

Figure 9: Starting a trusted agent

4 Handling multiple connections

In modern distributed systems a software agent can have different roles: it can be server in some situations, client in others, and at the same time act as an event producer for a monitor agent. In these system the default operating mode is asynchronous, non-blocking mode. An object can have multiple simultaneous connections.

Bond objects receive messages in the `say()` function. The *messaging thread* of the Bond executable is responsable for calling the `say()` function of the corresponding object. It is the objects responsability to process the message, identify it as being a reply to a particular request send some time ago, an event for which the object subscribeed for or a request coming from a remote object.

In order to facilitate the handling of multiple connections the Bond system offers the mechanism of *waiting slots*. When using waiting slots, the messaging thread of the Bond executable acts as a demultiplexer, by presorting the incoming messages according to the semantics of the message.

There are two different kind of waiting slots, having a close relation to the KQML semantics:

-*event waiting slots* which collect the messages coming from a source. Every time when an object subscribes as an event consumer for a remote object, a waiting slot is created. The incoming messages from the remote object are placed in the waiting slot and handled by the `on_notify()` function. The waiting slot is deleted when the object sends an `(discard)` performative, indicating that is no longer interested in the events coming from the object.

-*reply waiting slots* collect the reply for a request. Any message that contains a `reply-with` field will create a reply waiting slot. An incoming message with the parameter `in-reply-to` with the same value is placed in the reply waiting slotn and the `on_reply()` function called. The waiting slot is normally deleted after a successful return from the `on_reply()` function.

Any Bond object can turn on or off the source waiting and reply waiting features by setting the `source_waiting` and `reply_waiting` boolean variables. If they are set to false, the corresponding messages will be handled to the `say` function.

As an observation, the waiting slots are tightly coupled with the KQML semantics, and they function correctly when communicating with other, not Bond KQML agents.

4.1 Event waiting slots

Event waiting slots are used for handling events. They are created whenever an object sends a subscribe message to an event producer. They are deleted when a (discard) performative is sent.

The messaging thread is responsible to place every message representing an event into the event waiting slot corresponding to the particular event source, and to notify the receiving object by calling the on_event function. Figure 10 shows the operation and lifecycle of the event waiting slot. Because there is a separate waiting slot for each event producer, the mechanism helps the object to keep participate in multiple event handling message patterns. The following example shows a typical way to handle incoming events.

```
void on_event(bondShadow o, bondKQML m) {
   // an event arrived from the event producer
   // pointed by the shadow o
   switch(m.type) {
      case EVENT_1:    // handling the event
      case EVENT_2:    // according to its type
      default:
         say(m,o);     // fallback to normal message handling
   }
}
```

4.2 Reply waiting slots

Reply waiting slots are used for the asynchronous matching of the replies with the questions. For every message needing a reply (indicated by the reply-with parameter in the message) a reply waiting slot is created. The execution continues, and it is the messaging thread's task to identify the reply with the question. The answer is put into the reply waiting slot, and the object is notified be calling the on_reply function. This mechanism helps the object to send multiple simultaneous messages without having to wait for a reply, and still to match easily the replies with the questions. As an observation another approach, frequently used in servers, would be to create a separate thread for handling each connection. Reply waiting slots are a more economical approach which does not imply creation of new threads.

Figure 11 shows the operation and lifecycle of the reply waiting slot. The following example shows a typical way to handle replies.

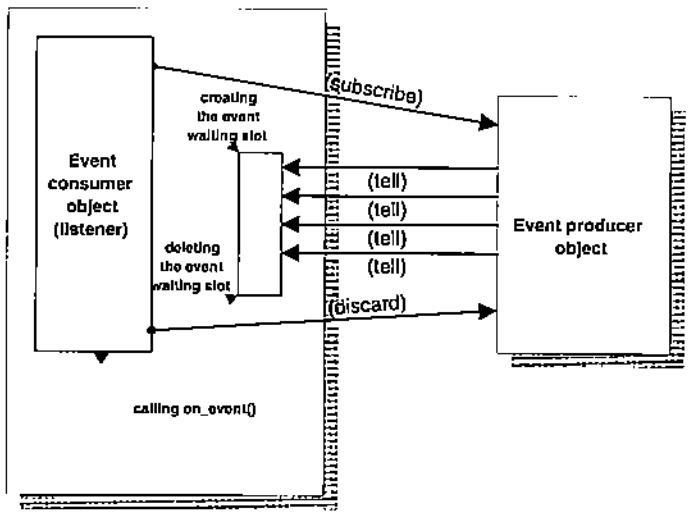

Figure 10: Event waiting slots

```
boolean on_reply(bondKQML message, bondKQML reply) {
    // the reply to "message" was "reply"
    return REPLY_OK; // satisfied, delete the reply slot
}
```

4.3 Syncronous transfer in the Bond system

The messaging is asynchronous by default in the Bond system. If an object wants to implement a synchronous request-reply operation, the `waitReply()` function blocks until a reply arrives or a timeout happens. In this way a synchronous operation can be achived.

```
bondKQML m = new bondKQML("ask :content get :value x)");
remote.say(m);
if (m.waitReply() == false) {
  \\ timeout!!!
} else {
  \\ reply received in the waiting slot
}
```

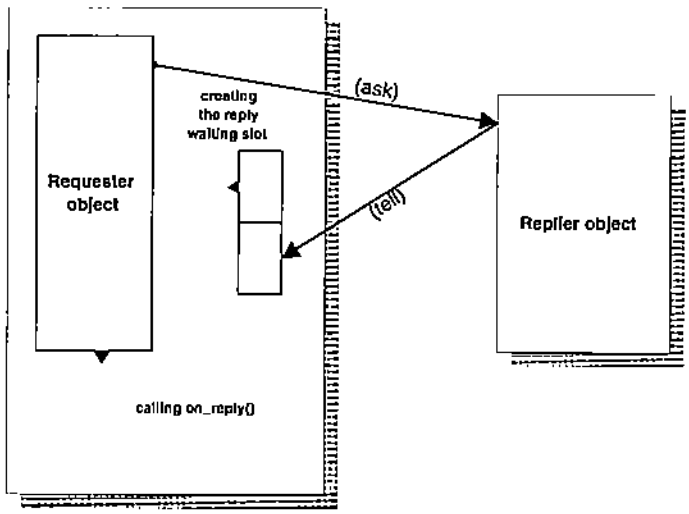

Figure 11: Reply waiting slots

5 Conclusions

There are a number of approaches to distributed object systems today. Some of them are tightly coupled, remote function call based systems like Corba, DCOM or RMI, while others a more loosely coupled message oriented approaches. Although a number of significant vendors offer message oriented middleware like IBM's DSOM or Novell's Tuxedo, there is yet no emerging standard for message oriented middleware systems.

In the Bond system we have chosen the message oriented approach. This paper presented two key concepts the *subprotocols* and *message patterns* used by the Bond system to implement the needed functionality. These are independent upon the transport mechanism used.

The following table presents the solutions provided by Bond to various challenges of a distributed object system design compared with Corba solutions. The solutions provided by both systems are consistent with their philosophy: the abstractions used by Corba are producedure calls, services and interface descriptors while the abstractions used by Bond are messages, message patterns and subprotocols.

From the practical point of view we expect a slight overhead for message based systems needed for message processing. On the other hand, we expect that message oriented systems scale better because of the loose coupling

between objects and asynchronous communication. We also expect that fault tolerance is easier solvable in a message oriented systems like Bond.

implementing...	Bond	Corba
communication channel	multiplexed to the in-box+outbox of the messaging thread	multiplexed on the ORB
messaging	native	messages as procedure calls
remote procedure calls	remote procedure calls as messages	native
event handling	subscribe/unsubscribe + event waiting slots	event service
finding objects	directory service	naming service
finding services	directory service + broker agent	object trader service
remote execution	resident	ORB
remote instantiation	realizing a shadow	life cycle service
static access	shadow + property access subprotocol	IDL + access functions
dynamic access	shadow + interface discovery subprotocol	interface repository service
multilanguage	preference for Java, but implementable in any OO language	yes

References

[1] Ö. Babaoglu, K. Marzullo *Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms* Technical Report UBLCS-93-1, January 1993

[2] L. Bölöni, K.K. Jun and D.C. Marinescu: *QoS and Reliability Models for Network Computing* Department of Computer Sciences, Purdue University CSD-TR #97-051

[3] L. Bölöni: *Bond Objects – a white paper* Department of Computer Sciences, Purdue University CSD-TR #98-002

[4] K. Mani Chandy, A. Rifkin, Paolo A.G. Sivilotti, J. Mandelson, M. Richardson, W. Tanaka and L. Weissman *A World-Wide Distributed*

System Using Java and the Internet IEEE International Symposion on High Performance Distributed Computing, August 1996.

[5] K. Mani Chandy *Caltech Infospheres Project Overview: Information Infrastructures for Task Forces*

[6] M.Lewis and A. Grimshaw *The Core Legion Object Model* Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing, IEEE Computer Society Press, Los Alamitos, California, August 1996.

[7] R. Orfali, D. Harkey, J. Edwards *Instant Corba* Wiley Computer Publishing 1996

[8] M. G. Sirbu, D. C. Marinescu: *Bond - A Parallel Virtual Environment* Proceedings of HPCN Europe '96, pp 722-728, Lecture Notes in Computer Science, Volume 1067, Springer Verlag, 1996

[9] M. G. Sirbu, *The Design of a Metacomputing Environment.* Doctoral Thesis, August 1997.

[10] M. G. Sirbu, D. C. Marinescu *A Scheduling Expert Advisor for Heterogeneous Environments* Proceedings of the HCW'97 Heterogeneous Computing Workshop, pp. 74-82, April 1997

[11] D.C. Marinescu *Software Development for Intranet Applications*, Proc DCIA 98, pp 31-50, January 1998

[12] T. Finin, et al. *Specification of the KQML Agent-Communication Language*, DARPA Knowledge Sharing Initiative draft, June 1993

[13] Y. Labrou, T. Finin *A Proposal for a new KQML Specification* UMBC TR-CS-97-03

[14] P. Hertmann *Illustrated Guide to HTTP*, Manning Publications 1997

[15] W. Grapp, E. Lusk and A. Skjellum *Using MPI*, MIT Press 1994

[16] D. Flanagan *Java in a Nutshell*, O'Reilly & Associates Inc. 1996