Not for distribution or attribution: for review purposes only

IEEE Internet Computing Paper Submission

Title: Event-Based Object Middleware: Towards a connection bus for distributed components

Authors: Pedro García López, Carles Pairot Gavaldà, Antonio Gómez Skármeta

Contact Address:

Pedro García López,PhD
Department of Computer Engineering and Mathematics
Office 238
Campus Sescelades
Avinguda dels països catalans, 26
43007 Tarragona, Spain
Phone: +34 977 558 510
Fax: +34 977 559 710
Email: pgarcia@etse.urv.es
Web: http://www.etse.urv.es/~pgarcia/

Event-Based Object Middleware: Towards a connection bus for distributed components

Pedro García López, Carles Pairot
Universitat Rovira i Virguli

Antonio Gómez Skármeta
Universidad de Murcia

Abstract

Object-oriented middleware is a mature technology that provides powerful abstractions for developing complex distributed applications. Event-based or message oriented middleware has received considerable attention for its decoupled nature that naturally fits asynchronous one-to-many interactions and highly dynamic distributed applications. Nevertheless, the marriage of the best of both worlds is far from been accomplished.

In this paper, we present ERMI, a distributed object technology that constructs on top of publish/subscribe event systems. This approach provides innovative benefits like object mobility, object replication, distributed interception, system reflection, object discovery and high performance asynchronous one-to-many notification. We have also developed a prototype event connection bus that extends publish/subscribe systems to include new services focused on distributed components like: publisher registration, publisher disconnection events, event interception and a meta-information connection service. We believe that such middleware is a solid substrate for future distributed component infrastructures.

Keywords

Event-based middleware, distributed objects, component connection and assembly.

1. Introduction

Object oriented middleware is a mature technology for developing distributed applications. Its widespread acceptance in distributed settings has led to a considerable level of sophistication and support by different vendors.

The distributed object-oriented frameworks that get the most attention are those that model messaging as method calls. These systems are often called remote procedure call (RPC) systems. The major benefit of these systems is that they make remote procedure (or method) calls appear to be local procedure calls (LPCs). This represents a powerful abstraction that considerably simplifies development of remote applications.

Message oriented middleware (MOM) has recently received considerable attention because of its decoupled nature that nicely solves asynchronous one-to-many interactions and highly dynamic distributed environments. In contrast to RPCs, MOMs don't model messages as method calls; instead, they model them as events in an event delivery system.. All applications communicate directly with each other using the MOM. Messages generated by applications are meaningful only to other clients because the MOM itself is only a message router

Nevertheless, distributed object-oriented frameworks and MOMs are still almost isolated worlds that do not fully benefit from each other unique advantages and concepts. We believe that the marriage of the best of both worlds is far from been accomplished and constructive synergies still remain to be developed.

However, object middleware is evolving to incorporate concepts from MOMs and we can outline interesting examples found in existing systems. COM connectable objects, J2EE Message Driven Beans, and OMG Corba Component Model sources and sinks permit distributed components to be activated by and trigger events to a messaging middleware.

In this paper, we go further in the integration of object middleware and event-based systems. We propose a distributed object middleware that is completely constructed on top of a publish/subscribe notification middleware. The underlying messaging middleware provides a decoupled abstraction that permits innovative distributed services like object mobility, object replication, distributed interception, system reflection, object discovery and high performance asynchronous one-to-many notifications.

Furthermore, while developing this object middleware, we have found important lacks in traditional event middleware. As a consequence, we have also developed a prototype event connection bus that extends publish/subscribe systems to include new services focused on distributed components like: publisher registration, publisher disconnection events, event interception, and a meta-information connection service. We believe that such middleware is a solid substrate for future distributed component infrastructures.

The rest of the paper is organized as follows: Section 2 describes ERMI (Event Remote Method Invocation), a complete object middleware constructed on top of publish/subscribe event systems. This section describes the major benefits and innovative services of this middleware and it also discusses major drawbacks related to this technology. Section 3 presents the Connection bus, an extended event system including new services for distributed components. Section 4 presents and compares related work and finally, in Section 5, we draw conclusions from this research and present future work trends in this line.

2. ERMI- Event Remote Method Invocation

ERMI is a distributed object middleware constructed on top of publish/subscribe event systems. This design decision implies modelling method calls as events and subscriptions over the underlying messaging middleware. We justify this approach because of the clear benefits and synergies obtained with the combination of both conceptual models:

- For object middleware developers, the main advantages derive from the decoupled nature of the MOM layer. This decoupled nature fuels innovative services like object mobility, object replication, asynchronous one-to-many notification, object discovery and system reflection.
- For MOM middleware developers, the main advantage resides in the higher-level abstraction obtained by using object middleware and method calls. Low–level event programming and message protocols are thus handled by the middleware (object stubs and skeletons).

ERMI Architecture

For the sake of simplicity, we have inspired our design in the Java RMI object middleware. Nevertheless the proposed model is generic and could be developed in any language and distributed technology.

To mimic RMI, we have developed quite similar APIs and tools with the aim of simplifying the learning curve for new developers. In this line, we provide an *ermi.Remote* interface, *ermi.RemoteException* class, and *ermi.Naming* class to locate objects in the registry. We also provide an *ermic* tool that generates *Stubs* and *Skeletons* for remote objects. Furthermore, ERMI currently provides many features found in RMI like Remote Exception handling, pass by value and by reference, and dynamic class loading.

The main difference with RMI resides of course in the communication layer located in Stubs and Skeletons. While in RMI a TCP Socket is established between the client (stub) and the server (Skeleton), ERMI Stubs and Skeletons both connect to an event service, and they must use subscriptions and events to communicate the method calls and results.

As we can see in figure 1 (left side), a *synchronous* call from the stub to the skeleton, implies two different subscriptions: a skeleton subscription for the remote call (object UID) and a stub subscription (object UID + RESULT + stub ID) for the result of the call. Both skeletons and stubs also send notifications that match their respective subscriptions. Although this solution seems complex, it is the only way to mimic synchronous blocking calls on top of an asynchronous non-blocking event middleware. In each method call, Stubs must then wait until the result event comes from the event service.

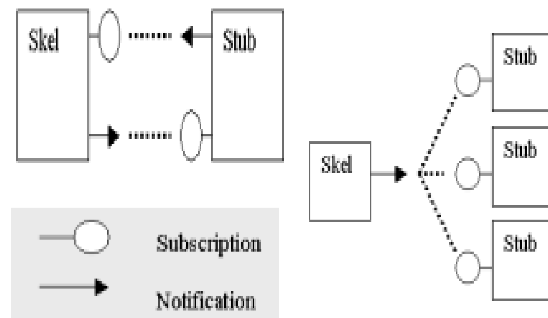

Figure 1. Architecture of synchronous and asynchronous subscriptions and notifications

In the other hand, *asynchronous* calls naturally fit with the underlying event middleware. As we can see in figure 1 (right side), an asynchronous notification call to n clients, will imply that all stubs will subscribe to the same topic (object UID + MethodID) and the skeleton will notify this event, matching the appropriate subscription. This decoupled model permits one-to-many notifications in a way that it is natural for the underlying asynchronous messaging middleware.

It is obvious that the main architectural difference is the existence of a new intermediary: the event service. Clients and server do not directly talk to each other, but instead make use of the underlying decoupled abstraction. The new intermediary implies a clear requirement: all client and server references first need the location of the event service to begin any communication. Because of that, we have decided to include event service location in every object reference along with its UID and class metadata.

ERMI extended functionality

Once described the ERMI internal architecture, we now present the innovative services that we have built on top of our middleware. The presented services can be easily constructed because of the decoupled nature of the underlying event infrastructure.

We outline six interesting services:

- Object mobility
- Distributed interception
- Asynchronous one-to-many notification.
- System reflection and monitoring
- Object discovery
- Object replication and Object caching

Object Mobility refers to the possibility of moving object servers to different locations and continue handling client requests. Object mobility is easily accomplished in ERMI by serializing the object implementation (that inherits from Skeleton) to the remote endpoint. Before serialization, the Skeleton removes all subscriptions to the Event service, and, upon arrival to the remote endpoint, the Skeleton reconnects to the event service and turns to create the subscriptions in the new location. Object clients (stubs) remain unaware of these changes since they maintain their current subscriptions unaffected.

In traditional object middleware, the strong coupling between clients and servers through TCP connections would require to advise all clients to reconnect to the new server location or use instead ad-hoc remote proxies. The first solution does not scale for a high number of clients and the second one is only an ad-hoc façade not suitable for unexpected scenarios. Our decoupled approach permits flexible object mobility and it could be used in different settings like server load balancing, spontaneous systems, agent systems and for highly dynamic and manageable remote services.

Distributed interception is an interesting service for applying connection-oriented programming [9] concepts in a distributed setting. With this service, it is possible to reconnect and locate type-compatible interceptors at runtime in a distributed application. Again, our decoupled model allows us to create custom Skeletons and Stubs for remote classes that can intercept calls to a running remote object. We however need to change the subscriptions of both Interceptor skeletons and Intercepted remote objects. We even demand that a queue of interceptors can be established or removed, so our skeletons must have a relatively complex message protocol able to permit such requirements. As we will explain further in section 3, distributed interception can be simplified if event systems would give us the required subscription filter functionality.

Again, distributed interception is hard to implement in strongly coupled object systems where both clients and servers must be notified of object changes. If a TCP connection is established among many clients and an object server, the insertion of a remote interceptor would imply that all clients should reconnect to the new interceptor, and to bind this interceptor to the remote server. Our decoupled solution is clearly more scalable and do not affect client connections (represented as subscriptions).

Asynchronous one-to-many notification is a distributed object event service that fits gracefully with our overall model. As explained in figure 1.b, all clients (stubs) subscribe to the same topic (UID + METHODID) and the object server (skeleton) publishes events matching that subscription. Obviously, this scheme scales better than point to point connections to any interested client and better performance is easily achieved by event systems.

Several messaging infrastructures employ multicast IP to route these events and thus improving the overall traffic and architecture.

```java
public interface Sprite extends ermi.ERemote
{
    public int getX()
        throws ermi.RemoteException;
    public void setX(int x)
        throws ermi.RemoteException;
    public void addSpriteListener(SpriteListener sL)
        throws ermi.RemoteException;
    public void removeSpriteListener(SpriteListener sL)
        throws ermi.RemoteException;
}
```

Figure 2. Java naming conventions in asynchronous notifications.

In the design of this event system we want to stay close to the programming language chosen. Because of this, our *ermic* generates stubs and skeletons using the same naming notations employed in the Java language for asynchronous notifications. As we can see in the Figure 2, the Sprite interface includes methods for registering client interest in Sprite state changes. The generated stub code creates the appropriate subscription and thus decoupling server from clients

As stated before, this object service is usually solved in distributed object middleware (as JINI) with the use of synchronous calls to all the interested clients. This approach is clearly less scalable and hinders the asynchronous nature of one-to-many notifications.

System reflection and monitoring can be easily achieved when all object communications traverse the event service. First of all, the event service can obtain a snapshot of the relation graph of clients and servers from a running system. In section 3 we will describe an extended service that can give us even more meta-information about the distributed applications and components. In the other side, system monitoring is straightforward simply by listening to events in the messaging middleware. Complex event filtering can also be applied to isolate concrete parts of a distributed application.

In traditional object middleware, system reflection and monitoring requires ad-hoc external intermediaries that must be notified of both client-server connections and distributed interactions among running components. In our case, the intermediary is the underlying event middleware, that is in fact ideally aimed for monitoring and filtering purposes.

Object replication and Object caching are added functionalities derived of the flexibility of the event bus. Object replication is accomplished generating special stubs that talk message protocols through the event system in order to main consistency and data among the remote object replicas. There is not a central object server so any of the clients could fail and the state is preserved. Object caching is also accomplished generating special stubs for object caches. These caches listen for state changes in a central object server in order to maintain a local cache of the object data. Object state changes are still routed to the central server to maintain consistency. Both object replication and object caching benefit from the event bus as the communication channel to establish message protocols and transmit state changes to interested stubs.

It is obvious that both object replication and caching can be easily constructed on top of existing object middleware. Nevertheless, both services share a common requirement: they need an efficient communication channel to route state changes or consistency protocols among the interested parties. Whereas this communication channel can be architected on top of one-to-one synchronous calls, it fits better with asynchronous one-to-many event systems.

Finally, *object discovery* consists of using predefined Object UIDs to locate remote objects in an event bus. In this case, clients locate objects servers with Ids associated to the object subscription. Object discovery is an extremely useful functionality in highly dynamic spontaneous scenarios such as wireless or mobile networks.

Object discovery is usually solved in existing systems by means of the so named lookup services. In this line, JINI lookup service employs UDP broadcast to automatically discover services in the local environment. In fact, this is a nice solution that involves a one-to-many channel like UDP broadcast. Although it is a good solution in local area networks, it is not appropriate for remote endpoints where UDP multicast or broadcast are not available. In these situations, using a distributed event service (TIBCO for example) would make the lookup service really accessible to remote locations.

Major drawbacks

Our approach presents two major drawbacks that must be considered:

- Performance loss in synchronous calls
- The event middleware can become a bottleneck

The first problem is in fact the more severe for the use of our model. Modelling synchronous calls on top of an asynchronous service implies an important penalty in performance. Our initial tests prove that Java RMI spends an average of 10 ms in a method call while ERMI (over our Connection bus) can spend from 15 to 35 ms.

It is however clear that a simple client-server synchronous application will perform better in a traditional object middleware. Event-based object middleware has sense when the distributed applications need special features like server mobility, code replication, dynamic systems, system monitoring and reflection, or distributed interception. There are usage scenarios where this middleware can be specially appropriate like agent systems, mobile environments or dynamic spontaneous systems like the ones solved by JINI middleware. In this scenarios, the performance loss in synchronous calls is then justified by the other services.

The second problem is the possible bottleneck that can be produced when all communications are routed through the Event middleware. This is a less important problem, since Event systems may not strictly follow a centralized client/server approach. Many systems like TIBCO construct on top of Multicast IP and server federations in order to scale to a high number of publishers and subscribers.

Furthermore, we have also developed a decentralized ERMI version called DERMI [7]. DERMI works on top of a peer-to-peer overlay network and it thus achieves complete system decentralization. This implementation works on top of the Pastry overlay network and extends the Scribe [2] multicast mechanism to provide the services presented in this paper. It also provides other programming abstractions such as anycall or manycall, which allow the programmer to make calls to groups of objects without taking care of which of them responds until a determinate condition is met. See [5] and [7] for further information about DERMI

3. The connection bus

The initial version of ERMI has been constructed on top of existing messaging middleware like Elvin and the Java Message Service. Due to design limitations found in traditional event middleware, we have decided to create an extended messaging service: the Connection bus. This new event system provides extended functionality ideally suited for supporting distributed component interactions.

First of all, the connection bus can be constructed on top of a topic or subject-based event system. Our bus offers topic-based filtering in order to improve event delivery times and thus obtaining high performance. Nevertheless, this service could use content-based event addressing like Elvin and thus allowing more powerful filtering services.

Apart from the basic publish / subscribe functionality provided by event systems, we require at least four additional services:

- Publisher registration
- Publisher disconnection events
- Event interception
- A meta-information connection service

These four services are specially designed to support distributed object middleware. We believe that event systems may include these functionalities in order to provide better support for distributed component interactions.

Publisher registration refers to the case when it only exists one publisher for a given topic. This is a typical scenario in object middleware when the object server or client are the only publishers in a given topic. The primitive *registerPublisher(topic, Info)* informs the event service that a given client will be the only publisher in a topic, and it can also include additional data in a dictionary (*Info*) like type or method information.

The information provided by *registerPublisher* to the Event system is very useful for two main reasons: first of all, the messaging middleware can optimize event delivery from the publisher to all subscribers benefiting from the underlying network topology. Furthermore, this information contained in the event system is key for the connection service to define component relations and interactions.

Carzaniga's Object of Interest is possibly the more related concept to publisher registration found in the literature. Nevertheless, Carzaniga's approach is focused on optimising event delivery from publisher to subscribers. It thus not provide meta-information for defining component connections

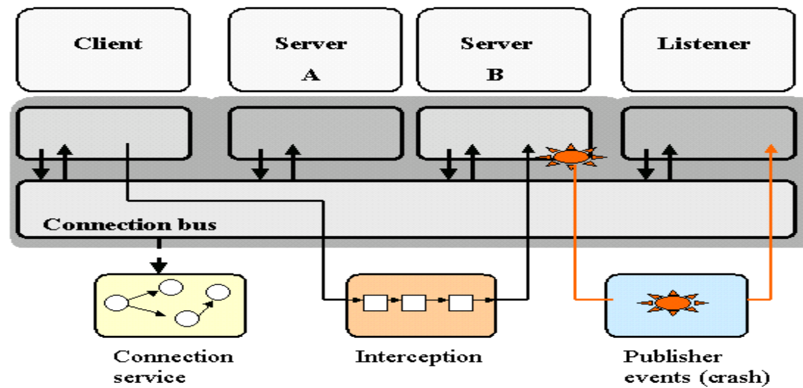

Figure 3. Connection bus services

Publisher disconnection events is an interesting service very related to publisher registration. When a single publisher exists in a topic, subscribers may be interested in obtaining information events about the current state of the publisher. The event system can thus notify interested subscribers of events like publisher disconnection, publisher deregistration, or publisher crashes and connection errors (see figure 3)

This service is extremely useful in object middleware, when clients are interested in server crashes or server disconnections. Upon arrival of these events from the connection bus, clients can then perform sanity actions or recovery mechanisms. Publisher disconnection information is even more important if we consider that ERMI servers and clients are not tightly coupled but use instead the underlying decoupled middleware.

Event interception is a service specially designed to support the distributed component interception feature presented in Section 2. The new primitive *addInterceptor(topic, Interceptor)* permits that any event sent by a publisher is first routed to one or more interceptors arranged in an ordered queue. Once these filters perform event transformations or custom actions, the resulting event is then routed to the targeted subscribers. As we see, the event traverses the interceptor queue and then is routed to all subscribers (figure 3).

This service considerably simplifies object skeletons and stubs in our ERMI middleware when performing distributed interception. The burden of distributed interception is then placed in the event middleware. Furthermore, as we explained in Section 2, the event system can locate interceptors in selected endpoints in order to improve the overall performance and also reducing event roundtrips.

Finally, the *meta-information connection service* is a key actor of our overall architecture. Because all component communications are routed through the event intermediary, this is a suitable point to store component relations and connections at runtime.

In fact, most of the connection information is already stored in the event system by means of subscription information. The event middleware can tell us what clients are connected to a server thanks to their existing subscriptions. Furthermore, the added type information included in the *registerPublisher* primitive allows us to know what is the class type that binds a server and a client. With all this information, the event system can give us a complete snapshot of the distributed connections and interactions between components at runtime.

The information provided by the connection service can be very useful to better understand and introspect a distributed environment. This information could serve to relocate components, to perform load balancing, to change topologies, or even to internally optimise event dispatching depending on client location and bandwidth.

We believe that the proposed connection bus extends existing event systems with useful functionalities for distributed components. Nevertheless, the decoupled nature of the event middleware remain unaffected. Furthermore, the four services could also be used by non-object middleware in distributed applications.

4. Related Work

The increasing popularity of the publish/subscribe paradigm has led to interesting research in the last years. In this line, several projects have applied event-based decoupled abstractions to heterogeneous domains.

ECO (Events-Constraints-Objects) [8] is an interesting approach to integrate events with distributed objects. Objects in ECO communicate by announcing events and by processing those events which has been announced. With the keywords *event*, *outevents* and *inevents* every object specifies its event dispatchers and interests. Furthermore, *constraints* (pre and post) permit to insert named conditions which control the propagation and handling of events.

ECO defined an innovative and elegant object model that considerably improves development of distributed applications. Furthermore, the constraint abstraction is a powerful concept for solving synchronisation and timing restrictions. Because constraints are processed at the start and at the end of the event handler, they permit some kind of event interception.

Nevertheless, ECO is not a complete distributed object middleware in the sense that is focused in triggering and handling of events, and not in remote method invocations. This implies a lower level abstraction than ERMI, that still requires considerable effort by programmers in order to handle and produce remote events. We move the event burden to the underlying skeleton and stub middleware, and we define our interceptors at method level. Pre and post constraints is a good idea that will influence considerably our interceptor design but in our case focused on method invocation. To conclude, ECO provides interesting abstractions, but is still focused on event handling and does not offer ERMI and Connection bus extended functionalities for distributed components.

Regarding object mobility on top of publish/subscribe systems, there exist several approaches like Siena [1] and JEDI [3]. Again, these systems are more focused on event handling in remote endpoints. An object can be moved to remote locations with explicit methods (*moveIn* or *moveOut*) and continue handling events in the remote endpoint. Our approach focuses on object server mobility and implies that after moving a server, it will continue handling client method invocations or producing asynchronous one-to-many invocations. Our design is focused on moving stubs (pass by reference) or skeletons and implementations (object servers) in order to allow object mobility. Again, the event handling burden is solved by our middleware.

Regarding interceptors, we have been influenced by Java Distributed Event Architecture. More concretely, we have refined the so called “Distributed Event Adapters” in order to apply this concept to remote method invocations and distributed component interception for both synchronous and asynchronous calls. ECO pre and post constraints also constitute an interesting contribution to this concept.

Siena [1] is also an important influence for our overall work. Siena’s advertisements influenced the design of our connection bus *registerPublisher* feature. Nevertheless, Siena advertisements are focused on improving event dispatching from publisher to subscribers, whereas our primitive includes additional information (class and type information) that is useful for our meta-information connection service.

Finally, the GridCCM [4] project is an interesting approach for developing parallel and distributed applications in the CORBA component model platform. GridCCM constructs on top of the underlying MPI (Message Passing Interface) and it defines an elegant architecture for distributed and parallel components. Nevertheless, we believe that an extended publish /subscribe system like our connection bus is conceptually superior to the MPI middleware for gluing distributed components. Furthermore, our ERMI and DERMI implementations solve problems in different settings than the GridCCM platform.

5. Conclusions

The paper presents a distributed object middleware constructed on top of a publish/subscribe notification middleware. We argue that the underlying decoupled abstraction fuels innovative services like server mobility, object replication and caching , distributed interception, system reflection and monitoring, object discovery and high performance asynchronous one-to-many notification. Whereas many of them can be architected on top of tradicional synchronous calls, they fit and scale better with one-to-many asyncronous event services.

Furthermore, we propose several extensions to traditional event systems in order to better support event-based object middleware. The principal extensions are: publisher registration, publisher disconnection events, event interception and a meta-information connection service. Our prototype event service that offers such functionalities is called the connection bus. This name is inspired in the connection-oriented programming paradigm focused on component assembly. We believe that our approach creates a seamless connection service that can improve component assembly in highly dynamic environments.

Our proposed middleware is specially useful in dynamic settings where the decoupled abstraction can show its real benefits. System introspection and monitoring, distributed interception, mobile servers, agent systems and spontaneous mobile environments are suitable scenarios for our ERMI middleware.

One of the interesting points of our approach is that our conceptual model does not imply a client/server event service. We have also developed a decentralised version of ERMI on top of a peer to peer overlay network, demonstrating its application in heterogeneous settings.

We foresee interesting research in the confluence of decoupled event services and distributed component infrastructures. We also believe that many settings can really benefit from this decoupled model. More concretely, connection-oriented programming and aspect oriented programming could use and improve our distributed interceptors and connection service.

ERMI, DERMI and the Connection bus [5] are open source projects with stable versions including samples, documentation and unit tests. We continue development of these three environments in order to provide other services like persistence, security and a distributed container model. We also begin to work with class tagged attributes to produce a more elegant generation code mechanism at the level of method calls. We also plan to replace naming conventions in Remote interfaces with attributes selecting different parameters like replication, notification, caching, etc. Our future work implies constructing a component container platform for distributed components on top of the ERMI/DERMI implementation and to explore new application settings like agent systems and collaborative applications.

To conclude, although much work remain to be done, we consider that distributed components and decoupled event systems still have constructive synergies to be explored.

6. Bibliography

[1] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf. "Design and Evaluation of a Wide-Area Event Notification Service". ACM Transactions on Computer Systems, 19(3):332-383, Aug 2001.

[2] M. Castro, P. Druschel, A-M. Kermarrec and A. Rowstron, "SCRIBE: A large-scale and decentralised application-level multicast infrastructure", IEEE Journal on Selected Areas in Communication (JSAC), Vol. 20, No, 8, October 2002.

[3] G. Cugola, E. Di Nitto, and A. Fuggetta, “The JEDI event-based infrastructure and its application to the development of the OPSS WFMS". In Transaction of Software Engineering (TSE), vol. 27, num. 9, September 2001.

[4] Alexandre Denis, Christian Pérez, Thierry Priol, and André Ribes. Process Coordination and Ubiquitous Computing, chapter Programming the Grid with Distributed Objects, pages 133--148. CRC Press, 2003.

[5] ERMI, DERMI and CBUS site. http://ants.etse.urv.es/ERMI/

[6] P. Eugster, R. Guerraoui, and C. Damm. On objects and events. In Proceedings for OOPSLA 2001, Tampa Bay, Florida, October 2001.

[7] Carles Pairot, Pedro García, Antonio F. Gómez Skarmeta. DERMI: A Decentralized Peer-to-Peer Event-Based Object Middleware. Submitted to IEEE ICDCS 2004.

[8] G. Starovic, Vinny Cahill, Brendan Tangney: An Event Based Object Model for Distributed Programming.OOIS'95, 1995 International Conference on Object Oriented Information Systems, 18-20 December 1995.

[9] C. Szyperski. “Component Software. Beyond Object Programming.” Second Edition. Pearson Education. 2002.