Towards a Peer-to-Peer Object Middleware for Wide-Area Collaborative Application Development

Carles Pairot, Pedro García, Rubén Mondéjar

Departamento de Ingeniería Informática y Matemáticas
Universitat Rovira i Virgili
Avinguda dels Països Catalans, 26
Tarragona, 43007
{cpairot, pgarcia}@etse.urv.es

Antonio F. Gómez Skarmeta

Departamento de Ingeniería de la Información y las Comunicaciones
Universidad de Murcia
Apartado 4021
Murcia, 30001
skarmeta@dif.um.es

Abstract

In this paper we present DERMI, a decentralized wide-area event-based object middleware built on top of a peer-to-peer substrate. Its main building block is the underlying publish/subscribe event notification system provided by the peer-to-peer layer. By using this methodology, innovative benefits like object mobility and discovery, object replication and caching, distributed interception, high performance synchronous/asynchronous one-to-one/one-to-many notifications and decentralized object location services are provided. Moreover, new programming abstractions (anycall and manycall) are introduced, 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. We believe that such middleware is a solid building block for future wide-area collaborative applications.

Keywords: Wide-area event-based middleware, distributed objects, peer-to-peer overlay networks, collaborative applications.

1. Introduction

Over the years, the Internet network has been growing steadily in number of users and nowadays its ubiquitous nature is well-assumed by everybody. Network bandwidth has increased considerably and the rising of many successful wide-area applications have made it become more and more popular. Apart from network bandwidth itself, computers every day have more overall capacity and its resource sharing capabilities become more and more important.

[1] Los tipos Times y Times NewRoman se asumen equivalentes

Most organizations have a wide variety of heterogeneous hardware systems which run different operating systems and rely on different network architectures. As a result, integration and development of distributed applications is difficult. Middleware systems appeared to provide a useful abstraction layer for building complex distributed applications. They give a common higher-level interface to the application programmer and hide the complexity of dealing with a huge variety of underlying networks and platforms.

The distributed object-oriented middleware frameworks that get the most attention are those that model messaging as method calls. 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. Mature examples of this middleware are CORBA, RMI or DCOM.

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 do not model messages as method calls; instead, they model them as events in an event delivery system. Clients send (produce) and receive (consume) events, or "messages", and producers and consumers do not explicitly know each other. 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's unique advantages and concepts. We believe that the marriage of the best of both worlds is far from being accomplished and constructive synergies still remain to be developed.

Regarding the architectural aspects, both middleware approaches are mostly built on top of centralized client/server models, and this is proven to work well in local-area or even metropolitan-area environments. However in wide-area settings, these platforms clearly suffer from scalability problems, although they can be solved by forming cluster topologies among servers. This option may not be economically viable in all cases. The actual trend is to head towards decentralized models which benefit more from the computing at the edge paradigm, where resources available from any computer in the network can be used and are normally made available to their members.

Perhaps one of the most quickly-growing technologies in computing are peer-to-peer (P2P) networks, which perfectly fit in the computing at the edge paradigm stated before. P2P computing enables applications that are collaborative and communication-focused. High availability comes through the existence of multiple peers in a group, making it likely that at any time there is a peer in the group able to satisfy a user request. This stands in stark contrast to traditional computing models, where high availability comes through complex load-balancing and application fail-over mechanisms. Examples of successful applications of this kind include Napster, SETI@Home, Gnutella, KaZaA or eMule. Lately, many research efforts in this field have been directed towards building efficient, scalable, fault resilient and self-organizing peer-to-peer overlay networks, which aim to provide interesting services such as distributed hash tables (DHTs), decentralized object location and routing, and scalable group multicast/anycast [Dabek et al. 03]. In particular, the DHT abstraction provides the same functionality as a traditional hash table, by storing the mapping between a key and a value, thus supporting the standard interface of `put(key, value)` and `get(key)`. The value is always stored at the live overlay node(s) to which the key is mapped depending on a hashed value of this key. The really innovative aspect of these overlay substrates is that they conform to a specific graph structure (typically organized forming a ring) which makes them to satisfy the condition that the number of hops required for locating any object is *O(log n)*, where *n* is the total number of nodes in the system. Examples of these technologies include [Rowstron et al. 01], [Stoica et al. 01], [Zhao et al.]. All the services provided by these systems give an abstraction layer to upper-level applications which can benefit from them.

In this paper, we go one step further in the integration of object middleware and event-based systems and we present DERMI, a decentralized distributed object middleware that is completely constructed on top of a DHT-based peer-to-peer overlay network substrate. Its main aim is to provide programmers with the necessary abstractions to develop wide-area scale distributed applications. They do not have to take care about solving the common issues of a distributed application, like scaling, routing, replication or caching: our middleware provides them with the necessary APIs to do so. It benefits from the publish/subscribe underlying functionalities provided by the peer-to-peer layer and it thus offers distributed services such as object mobility, caching and replication, distributed interception and high performance synchronous/asynchronous one-to-one/one-to-many notifications. Furthermore, it provides innovative programming abstractions, which include **anycall** and **manycall**.

The rest of the paper is structured as follows: Section II is a summary of related work in wide-area distributed object systems and related technologies, including an overview of Pastry and Scribe, which are the underlying technologies DERMI is built on top of. Section III describes DERMI itself, showing the major benefits and innovative services of our middleware, as well. Section IV presents CSCW utilization applications of our object middleware and finally, in Section V we draw conclusions from this research and present future work trends in the same line.

2. Related Work

A number of companies have advocated peer-to-peer solutions to problems such as distribution of streaming media, web hosting, distributed auctions, etc. There is a renewed interest in a large body of distributed systems research on resource sharing and collaboration in both LAN and WAN environments. In particular, the so called "WAN-OS" projects such as Legion [Lewis et al. 96] or Globe [Van Steen et al. 99] are well suited for supporting arbitrary P2P applications since their goal is to make the Internet look like a single parallel machine by hiding (to the extent desired by the developer) all the complexities associated with vastly different machines, local operating systems, communication protocols, local resource management, access control, and security policies.

The Globe System is aimed to support a big number of users, clients and objects through the Internet. One of the most important features of Globe is its distributed shared object concept, which allows objects to be replicated and distributed between different machines. However, due to this nature, Globe only provides one invocation type: synchronous calls. It has neither support for notifications nor callbacks. DERMI provides synchronous calls, asynchronous one-to-many calls and cleanly supports notifications as it is built on top of an event service.

One of Globe's important hot spots is its wide-area location service which maps object identifiers to the locations of moving objects. Globe arranges the Internet as a hierarchy of geographical, topological, or administrative domains, effectively constructing a static world-wide search tree, much like DNS. Information about an object is stored in a particular leaf domain, and pointer caches provide search short cuts. The Globe systems handles high load on the logical root by partitioning objects among multiple physical root servers using hash-like techniques. As DERMI is based on a DHT P2P overlay network, it performs this hash function well enough that it can achieve scalability without also involving any hierarchy; however, the number of network hops required to get the information varies depending on the number of nodes in the network, whereas in Globe remains constant.

Legion [Lewis et al. 96] provides an object based service model such that objects can be replicated and located arbitrarily transparently. It achieves this by a three level naming scheme which maps human readable names to Legion Object Identifiers (globally unique in time and space), which in its turn map at run-time to address and port of an active instance of the object. Notice that this scheme is similar to the one used by Globe for separating the object name from its address. However, the main difference between both systems is the way objects are considered. In Globe, objects are assumed to be physically distributed over many resources in the system. However, in Legion, objects can be physically distributed over multiple physical resources, but are expected to physically reside in a single address space.

In DERMI, a similar approach to Legion is used, since we use messages (or notifications) as our core communication mechanism. However, we have the notion of an **information bus** [Oki et al. 93], thus benefiting from a decoupled, many-to-many communication style between objects. This distributed information bus is responsible for transmitting to subscribers events thrown by publishers based on their subscriptions, and acts as a unique virtual bus which connects all objects.

Recently, major players such as Microsoft and Sun have announced new initiatives to support complex

P2P applications in their respective operating system environments. In the former case, P2P computing is intended as a part of .NET strategy, which envisions arbitrary services to be deployed over the web via the SOAP interface. In contrast, the JxTA open source project from Sun Microsystems proposes to enable P2P applications by specifying a set of protocols for peers to interact with each other. In addition, Sun also introduced the JINI software system designed for spontaneous systems to connect them to a larger network, using Java to distribute processes among the devices connected to the network. JINI offers services such as object discovery and code mobility intended for LAN environments. Nevertheless, the JINI Distributed Event Service uses Java RMI to notify clients of changes in a remote object, which clearly slows down performance when these changes have to be notified to many clients.

In order to merge the best of both worlds, we could use our DERMI middleware platform for JINI and JxTA approaches. DERMI provides object discovery and mobility, and could be transparently used in wide-area P2P environments as well as in smaller local-area collaboration rings, also providing fault tolerance, caching, replication, synchronous/asynchronous calls and distributed interception techniques.

Related work in the peer-to-peer research field, introduces so many P2P DHT-based overlay network substrates that we could use to build our middleware on top of. However, we needed a publish/subscribe event system which would benefit from the subjacent routing capabilities, which basically consist in that each participating node is assigned a uniform random node identifier (*nodeId*) from a large *identifier space*. Application-specific objects are assigned unique identifiers called *keys*, selected from the same id space. Each key is dynamically mapped by the overlay to a unique live node, called the key's *root*. In order to deliver messages efficiently to the root, each node maintains a *routing table* consisting of the nodeIds and IP addresses of the nodes to which the local node maintains overlay links. Messages are forwarded across overlay links to nodes whose nodeIds are proggressively closer to the key in the identifier space. This mechanism is the so-called **Key-based Routing (KBR)** [Dabek et al. 03]. In the case of Pastry, keys are mapped to the live node with the closest nodeId.

Scribe [Castro et al. 02b] offers an overlay multicast substrate on top of the Pastry routing mechanism. It introduces the concept of a topic (group identifier) to which nodes can subscribe to. Once subscribed, a node will receive all event notifications that fire on that topic. Each group has a unique group identifier (*groupId*). The Scribe node with an identifier (*nodeId*) closer to the *groupId* acts as the *rendez-vous point* for the associated group. This rendez-vous point is the root of the multicast tree created for the group. Group membership is managed by creating a reverse path forwarding multicast tree rooted at the rendez-vous point. In addition to the basic multicast functionality, Scribe maintains the tree structure in the face of high levels of node failures. This is imperative if the system is going to be robust. For further information on both Pastry and Scribe design and features, please refer to [Castro et al. 02b, Rowstron et al. 01].

3. DERMI

DERMI is a distributed event-based object middleware built on top of a decentralized DHT-based P2P overlay network. It benefits from the publish/subscribe underlying functionalities provided by the P2P layer and it models method calls as events and subscriptions under this underlying MOM. There is a prototype implementation available at [DERMI].

Notice that in a centralized client/server model, the event service itself could become a bottleneck in case the number of users increased beyond its threshold capacity. Recall that all events should traverse the event service, which is a centralized communication point. This problem cannot happen in DERMI because of the inherent use of a wide-area scalable notification service such as Scribe, as it does not depend on any centralized point.

So far, DERMI uses Pastry as its routing P2P substrate and Scribe as its MOM infrastructure. It is inspired by the Java RMI object middleware. It provides a *dermi.Remote* interface, a *dermi.RemoteException* class and a *dermi.Naming* class to locate objects in our decentralized registry, which we will talk about later. Mimicing RMI, we provide a *dermic* tool which generates both **stubs** and **skeletons** for our remote objects, which will transparently manage object publications/subscriptions and its inherent notifications. Furthermore, DERMI currently provides many features found in Java RMI, such as remote exception handling, pass by value and by reference, and dynamic class loading.

The main difference between RMI resides in the communication layer located between stubs and skeletons. While in conventional RMI a TCP socket is established between the caller (stub) and the callee (skeleton), DERMI stubs and skeletons both use the event service by making subscriptions and sending notifications in order to communicate the method calls and their results.

Notice however that the peer-to-peer approach also presents its drawbacks. One of them is how to perform the bootstrapping process: how can we find a contact node in the overlay to join? This problem is yet to be solved and although the idea of having a universal ring [Castro et al. 02] expected to be joined by all participating nodes seems to be a good starting point, this is still a hot research topic and much work is still to be done.

3.1. DERMI Services

Once described the DERMI architecture, we now present the innovative services we have built on top of our middleware, which are provided to the application layer. The presented services can be easily constructed because of the decoupled nature of the underlying event infrastructure.

We outline the following services, which are described as follows:

- **Invocation abstractions**, including a*synchronous one-to-many notification*, *synchronous one-to-one notification*, and the new abstractions *anycall* and *manycall*.
- **Decentralized object location**, which includes as well, *object mobility and discovery, object caching* and *replication*.
- **Distributed interception**.

3.1.1. Invocation abstractions

Asynchronous one-to-many notification is a distributed object event service that fits gracefully with our overall model. The *asynchronous calls* are modelled as one-to-many notifications in our middleware. As seen in Fig. 1, all clients (stubs) subscribe to the same topic *hash* (*objectUID* + *MethodID*) and the object server (skeleton) publishes events matching that subscription. Be aware that for doing so, we need to query before our object location service, which will return us the *objectUID* of the object to call. Obviously, this scheme scales better than point to point connections to any interested client, and better performance is achieved by the event system.

In the design of this event system we want to stay close to the programming language chosen. Because of this, our *dermic* tool generates stubs and skeletons using the same naming notations employed in the Java language for asynchronous notifications. The generated stub code creates the appropriate subscription and thus decoupling the object server from clients.

Synchronous one-to-one notification is the mechanism used in DERMI to model *synchronous calls*. They are not implemented the same way as asynchronous calls, because in fact, a synchronous call is simply a direct peer-to-peer call. Therefore, we do not use the event service (Scribe) but instead, we use Pastry's routing capabilities to send a message directly from the object client (stub) to the object server (skeleton). By doing so, we achieve a direct peer communication between both objects, which is more efficient than using the event service to route events to each of them, which would normally incur in *O(log n)* hops. The return results are sent back the same way, thus obtaining a very efficient call, involving only two hops: one for the call and the other one for the returned result. Obviously, these two hops do not include the necessary hops to locate the called object's handle using the object location service. More about the location service will be explained later on.

Anycall is a new and powerful way of doing a remote procedure call (by means of notifications). It benefits from the *anycast* extension implemented in Scribe. Anycast is a service that permits a node to send a message to a nearby member of a group, where proximity is defined using a metric like IP hops or delay.

The proximity-aware spanning trees built for each group by Scribe are used to anycast messages efficiently: messages are delivered to a nearby group member with low delay and link stress.

Our *anycall* implementation uses the *anycast* building block to create a call to the objects that belong to the same multicast group: they can be, for instance, object replicas, which can provide us with a service. The anycall client does not care about which object provides data: it wants its petition to be served by whoever can. So that, an *anycall* means sending an anycast notification to the group, which will make the closest member in the network to the sender, that satisfies a condition, to answer.

Imagine, for example, a kind of CPU intensive application like SETI@Home or United Devices Cancer Research Project. These applications mainly retrieve data units from servers, which are analyzed in our home or office PCs, and the results are subsequently sent back to these servers. An easy approach using an *anycall* could be used for retrieving data units. Now imagine we have several servers which have available data units. We could group them under the topic "AVAIL_DATA_UNITS", thus creating a Scribe multicast group, whose topic identifier would be *hash ("AVAIL_DATA_UNITS")*. Once our client would like to get a data unit, then it would execute *DataUnit du = anycall("AVAIL_DATA_UNITS", getDataUnit)*. This would send an anycast message to the group, and the nearest group member would check whether it has any data unit available. If that was the case, the data unit would be directly returned to the client and the anycast message would not be routed further. In the other case, the anycast message would be routed to another member of the group and so on, until any data unit was found or the root was reached, which would mean that none of the members of the group had available data units. This behaviour would throw a *dermi.RemoteException* back to the client so as it is correctly notified.

A **manycall** is a variation of the *anycall*. It basically

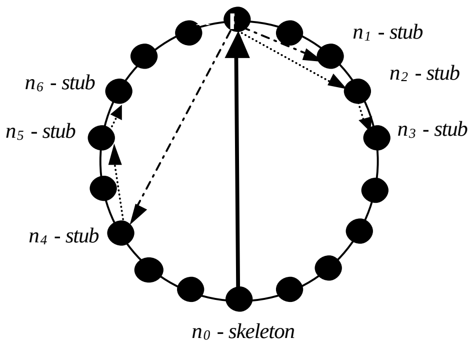

Figure 1. Asynchronous one-to-many notification. Object at node *n_0* sends an asynchronous call which is headed towards the subscribers group root (labelled *R*). The call event is disseminated throughout all clients

works by sending a *manycast* message to the members of a group, i.e. the message is sent to several group members. It is routed further until enough satisfying members are found to satisfy a global condition. It is similar to the *anycall*, in the sense that when an object receives a *manycall* message, it first checks if it satisfies a local condition and subsequently checks whether after calling its method, a global condition (passed along with the message) is met. If it is so, the *manycall* has been successful.

One scenario where a *manycall* could be very useful would be when doing an online voting poll. Let us imagine that we need a minimum of *x* votes to do a certain job. We could simply send a *manycall* to the group and each member would vote *yes/no* depending on its local condition, and after doing so, it would check the global one (if *x* votes have been reached). If this is the case, the voting process concludes successfully, communicating the result to the *manycall* initiator, else the unfavorable result is also told to the client.

3.1.2. Decentralized object location

A scalable, stable and fault-tolerant **decentralized object location** service is needed to locate object references in a wide-area environment such as DERMI. It would be illogic to rely on a centralized naming service which would clearly be an important bottleneck for doing such a common task as object location. As a consequence, we have implemented such facility in our middleware, as explained as follows.

The first approach that comes to our mind is to use the DHT facilities our P2P overlay network substrate provides us with and build our object location service on top of it. As stated in [Cox et al. 02], a P2P lookup service would benefit from the self-organizing and adaptive nature of the underlying layer. However, one problem here would be the higher latencies achieved which do not remain constant and that become augmented as the network size increases. This means that, for example, in a million node network, a maximum number of 5 hops would be required to locate the node that contains the information about where our object is. Despite the fact that this implementation seems not to be as efficient as we would like it to be, it has the advantage that it is completely embedded in our system in a natural way and that we do not need any other external services to do our object lookups.

This P2P location service would basically store object location information in order to locate these objects by using a human-readable name. As found in other wide-area location services [Van Steen et al. 99], our object names would not contain any object's location information in order to decouple the object's current location from its name. An object will have always the same name independent of its location.

Naturally, if the node containing the object's location information fails, object lookups would fail as well, as the node that contains this information is missing. To avoid this problem, data replication mechanisms should be used. When an object handle is to be inserted, this data should be replicated among the *k* nearest nodes to the target node. This way, should the target node fail, information would not be lost and the object's handle could be recovered from any of the *k* nearest nodes. To accomplish this objective in a transparent manner, a persistent and fault-tolerant storage management system like PAST [Rowstron et al. 01b] could be used.

Another more efficient wide-area scalable object location approach would be to use a hierarchical system such as the Globe Location Service. As already explained in Section II, objects are located by means of a dynamic adapting worldwide search tree. Clearly this approach is more efficient than the one presented before, because in the majority of the cases, with only 2 network hops it is possible to locate any object. However, if we only used this system, our middleware would depend on an external hierarchical service for object location, which could be cumbersome if it becomes unavailable for some reason.

We believe both systems could be used for our middleware. For the sake of efficiency, Globe's hierarchical search tree approach is recommended; however if we prefer not to depend on an external location service, we provide a decentralized P2P object location system, which distributes object handles in a more sparse way without involving any hierarchies.

Object mobility refers to the possibility of moving object servers to different locations and continue handling client requests. Object mobility can be easily accomplished in DERMI by simply serializing the object implementation (that inherits from its skeleton) to the remote endpoint. Before serialization, the skeleton removes all its subscriptions from 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 notify 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.

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 our decentralized event service (Scribe) would make the lookup service really accessible to remote locations.

Object replication and Object caching are added functionalities derived from the flexibility of the event bus. Object replication is accomplished generating special stubs that talk message protocols through the event system in order to maintain 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. Our approach is to have all replicas join a multicast group. Once an object wishes to call a method from a replicated object, the stub will send the call to the group as a special *anycall*. This will make any of the object replicas (normally the nearest one) to respond the call, which makes replication totally transparent to the user.

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. In this case, all object caches should join the same multicast group for the source object. This source object would be responsible for updating the caches state simply by periodically sending a multicast message to the group. 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.

3.1.3. Distributed interception

Distributed interception is an interesting service for applying connection-oriented programming concepts in a distributed setting. With this service, it is possible to reconnect and locate type-compatible interceptors at runtime in a distributed application. Our model allows us to create custom skeletons and stubs for remote classes that can intercept calls to a running remote object. Naturally, we demand that a queue of interceptors can be established or removed as well. In order not to change the subscriptions of both interceptor skeletons and intercepted remote objects each time a new interceptor is added or removed, we have extended our event service classes (Scribe) in order to natively support this feature.

Our interceptor implementation benefits from the fact that all events that must be sent to a multicast group in Scribe are first routed to its rendez-vous point. As stated in [Castro et al. 02b], this could provide us with a form of access control, however it can also be used for our interception purposes: each group's rendezvous point will contain a list of pointers to other interceptor objects, which will be updated every time a new interceptor is added or removed. As a consequence, each time an event is sent to a multicast group, this notification first arrives at its rendez-vous point, which will check whether it has interceptors or not. If it is not the case, the event will be normally sent to the multicast group itself; else, the event will sequentially pass throughout all the interceptors which may transform the event and finally it will be routed back again to the rendez-vous point which will, in turn, send the intercepted event to the group members. For further clarity, see Fig. 2.

A fault-tolerance mechanism is also needed in case the rendez-vous point changes. Scribe provides us with callbacks which notify these root node modifications. A simple approach to solve this problem would be to send all interceptor data from the old root to the new one. However, a different strategy should be used in case we wanted to take care of root node failures. We could store this interceptor information in the *k* nearest nodes to the rendez-vous point.

This should be easily achieved using a large-scale distributed storage management system such as PAST [Rowstron et al. 01b].

Normally, the number of interceptors will be low due to their sequential nature, efficiency quickly degrades. Nevertheless, to prevent such problem, and thus, reducing the number of network hops

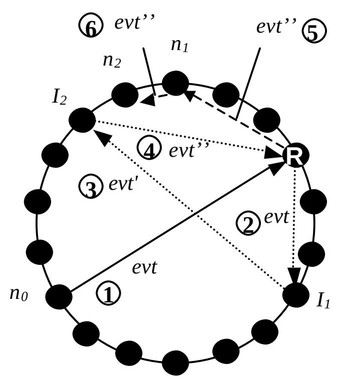

Figure 2. Distributed interception. Object at *n_0* sends an event to group rooted at R. This event is sent to the interceptor queue sequentially, thus transforming it (evt ? evt' ? evt''). Finally the event is sent back to root, which ends up delivering it to group subscribers

between interceptors, we could opt for moving them directly to their source objects, which would intercept events locally. This way, each time a publisher sent an event to its subscribers, the publisher itself would do the interception process locally, thus incrementing efficiency.

By using this interception mobility mechanism, distributed interception would also be supported in *synchronous call* environments where direct peer-to-peer calls are used. Notice that it was not possible to use interception mechanisms in direct *synchronous calls* because of the fact that no publish/subscribe mechanisms are taken into account in these special calls. Therefore, there existed no rendez-vous point to store interceptor pointers in.

Observe that 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 solution does not affect client connections, as they are represented as invariant subscriptions.

4. CSCW Applications of DERMI

The main idea when developing the middleware platform described throughout this paper was to ease the development phase when architecting wide-area distributed applications. In this line, developers can benefit from all the services our underlying layer provides for building these applications. We will focus on possible uses of our architectural model to build collaborative applications. In fact, we have already developed an application on top of DERMI which consists of an Eclipse [Eclipse] plug-in for team programming, called *CoopWork*.

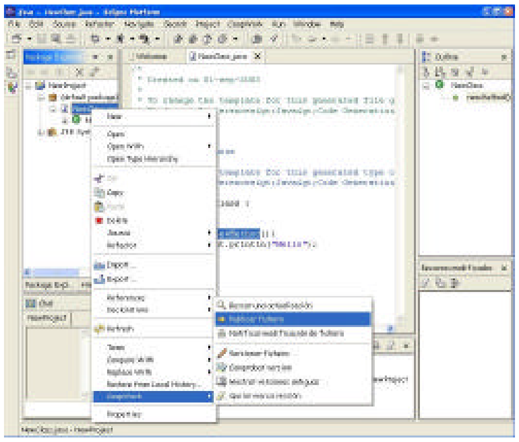

Figure 3. A snapshot taken from *CoopWork*

CoopWork provides a series of functionalities to ease application development for programmer groups. It allows concurrent project modifications by means of tools that permit method blocking / unblocking, file or method publication, acceptation or denial, looking for class updates, version control and even a chat functionality. This development environment is, of course, totally decentralized, thus eliminating any main server dependance. *CoopWork*'s ease of use is also one of its main hot spots. It does not require any kind of difficult installation; just simply register it as a *plug-in* to Eclipse and there is nothing more to configure. Compared to other similar systems, like CVS, *CoopWork*'s learning curve is very low.

In addition to *CoopWork*, we are planning, as well, to develop more CSCW and CSCL applications that will take advantage of DERMI's benefits, one of which is the *PLANET* project, whose main goal is to develop a low-cost multiuser collaborative platform for advanced training in settings like architecture, medicine or scientific simulation. The platform will extend an existing Collaborative Virtual Environment (MOVE) in order to provide advanced interaction and visualization with immersive virtual reality 3D devices such as Head Mounted Displays, gloves and stereoscopic projection systems.

Another objective of the *PLANET* project is the generation and distribution of educational content. To do so, we plan to extend the collaborative platform in order to create a distributed content repository. This repository is planned to be built on top of DERMI and will permit hierarchical access to distributed contents located in different knowledge repositories through content brokers and mediators. We will also integrate collaboration tools in the content's life cycle in order to promote knowledge communities around educational content hierarchies.

Further, we believe that DERMI can be the base infrastructure for a CSCW wide-area component framework, and so, it would seamlessly support the concept of a shared session inhabited by remote users that interact with synchronous and asynchronous components in a controlled and secure environment. This way, the transition from local-area CSCW applications to wide-area ones should not present many difficulties. The middleware's CSCW layer could be easily adapted to match the underlying layer's peer-to-peer nature and so, changes in the application's code should be minimal.

5. Conclusions and Future Work

This paper presents a wide-area decentralized distributed object middleware built on top of a publish/subscribe notification service (Scribe). We argue that the underlying decoupled abstraction fuels innovative services such as object mobility, object replication and caching, distributed interception, object discovery and high performance synchronous/asynchronous one-to-one/one-to-many notification. Whereas many of them can be architected on top of traditional synchronous calls, they fit and scale better with one-to-many asyncronous event services.

Further, we propose several new programming abstractions, such as *anycall* and *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.

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. In particular, connection-oriented programming and aspect oriented programming could use and improve our distributed interceptors and connection service.

DERMI is an open source project with a stable version including samples, documentation and unit tests. We continue development of this environment in order to provide other services like a distributed container model. We also begin to work with class tagged attributes to produce a more elegant code generation mechanism at method call level. We plan as well to replace naming conventions in Remote interfaces with attributes selecting different parameters like replication, notification, caching, etc.

As future work we are implementing an enhanced event system, called the Connection bus, which will provide extended functionalities ideally suited for supporting distributed component interactions. Publisher registration/disconnection events and a meta-information connection service are especially interesting new services to be added in a near future to DERMI.

Furthermore, we are also thinking of introducing new services such as system monitoring and reflection, and devising a security service along with a decentralized persistence service. A P2P common API was proposed in [Dabek et al. 03], which is included in the recent version of FreePastry [FreePastry]. We plan, as well, to move our middleware to this version so as to take advantage of this unified API.

In conclusion, although much work remains to be done, we consider that distributed components, containers and decoupled event systems still have constructive synergies to be explored.

Acknowledgements

This work has been partially supported by the Spanish Ministry of Science and Technology through project TIC-2003-09288-C02-00.

References

[Carzaniga et al. 01] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf. 'Design and Evaluation of a Wide-Area Event Notification Service'. ACM Trans. on Computer Systems, 19. pp. 332-383. (2001).

[Castro et al. 02] M. Castro, P. Druschel, A.M. Kermarrec, and A. Rowstron. 'One Ring to Rule them All: Service Discovery and Binding in Structured Peer-to-Peer Overlay Networks'. Proc. of the 10th ACM SIGOPS European Workshop. (2002).

[Castro et al. 02b] M. Castro, P. Druschel, A.M. Kermarrec, and A. Rowstron. 'SCRIBE: A large-scale and decentralized application-level multicast infrastructure'. IEEE Journal on Selected Areas in Communications, 20. (2002).

[Cox et al. 02] R. Cox, A. Muthitacharoen, and R.T. Morris. 'Serving DNS using a Peer-to-Peer Lookup Service'. Proc. of the 1st International

Workshop on Peer-to-Peer Systems. pp. 155-165. (2002).

[Dabek et al. 03] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. 'Towards a Common API for Structured Peer-to-Peer Overlays'. Proc. of the 2nd International Workshop on Peer-to-Peer Systems. (2003).

[DERMI] DERMI and ERMI website, http://ants.etse.urv.es/ermi

[Eclipse] Eclipse Project website, http://www.eclipse.org

[Eugster et al. 00] P.Th. Eugster, P. Felber, R. Guerraoui, and A.M. Kermarrec. 'The Many Faces of Publish/Subscribe'. Technical Report DSC ID:2000. (2000).

[FreePastry], http://www.cs.rice.edu/CS/Systems/Pastry/FreePastry

[Lewis et al. 96] M. Lewis and A. Grimshaw. 'The Core Legion Object Model'. Proc. of the 5th IEEE International Symposium on High Performance Distributed Computing. (1996).

[Oki et al. 93] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. 'The Information Bus – An Architecture for Extensible Distributed Systems'. Proc. of the ACM 14th Symposium on Operating Systems Principles. pp. 58-68. (1993).

[Rowstron et al. 01] A. Rowstron and P. Druschel. 'Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems'. IFIP/ACM International Conference on Distributed Systems Platforms (Middleware). pp. 329-350. (2001).

[Rowstron et al. 01b] A. Rowstron and P. Druschel. 'Storage Management and caching in PAST, a large-scale, persistent peer-to-peer storage utility'. ACM Symposium on Operating Systems Principles. (2001).

[Stoica et al. 01] I. Stoica, R. Morris, D. Karger, M. Frans Kaashoek, and H. Balakrishnan. 'Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications'. Proc. of the ACM SIGCOMM 2001. (2001).

[Van Steen et al. 99] M. Van Steen, P. Homburg, and A.S. Tanenbaum. 'Globe: A Wide-Area Distributed System'. IEEE Concurrency. pp. 70-78. (1999).

[Zhao et al.] B. Zhao, J. Kubiatowicz, and A.D. Joseph. 'Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing'. UCB Tech. Report UCB/CSD-01-1141.

[Zhuang et al. 01] S.Q. Zhuang, B. Zhao, A.D. Joseph, R.H. Katz, and J.D. Kubiatowicz. 'Bayeux: An Architecture for Scalable and Fault-tolerant Wide-area Data Dissemination'. Proc. of the 11th International Workshop NOSSDAV 2001. pp. 11-20. (2001).