Transparent Mobility of Distributed Objects using .NET

Cristóbal Costa, Nour Ali, Carlos Millán, José Ángel Carsí
Department of Information Systems and Computation
Polytechnic University of Valencia
Camino de Vera, s/n
46022, Valencia, Spain
{ccosta, nourali, cmillan, pcarsi}@dsic.upv.es

ABSTRACT

Nowadays, information systems are becoming more distributed and dynamic in nature, where mobility is a solution for run-time adaptability. However, implementing software with such characteristics is a complex task. This is due to the fact that current middleware technologies do not provide a simple and direct way of implementing distributed objects that can move in a transparent way. In this paper, we are going to present an approach, implemented in .NET Remoting to allow transparent mobility of distributed objects. Our approach is based on separating the distribution and mobility concerns from the source code that contains the application logic in entities called attachments. Thus, attachments are high-level proxies that are responsible for creating communication channels and are capable of managing dynamic location changes without affecting the objects in the case of mobility. This approach has been implemented using a case study. The response time of distributed communication provided by our approach has been tested and compared with the remote communication provided by the primitives of .NET Remoting.

Keywords

Distributed communication, transparent mobility, autonomous mobility, .NET Remoting

1. INTRODUCTION

Currently, distributed systems are built by using middleware services [Ber96a]. The main idea behind middleware is to allow components at different hosts to collaborate in such a way that users perceive the system to be centralized. Information systems are becoming more dynamic at run-time where mobility plays an important role for adapting applications and solving problems such as fault tolerance and load balancing.

However, building mobile and distributed systems is not a simple task. The middleware technologies that are currently available do not provide the sufficient primitives that allow the deployment of distributed components which have a mobile nature at run-time.

.NET Technologies 2006

For example, one of the steps for implementing mobile objects in .NET is serializing the object states using the serializable attribute. However, an object that must be accessible remotely in .NET Remoting cannot be serializable at the same time [Obe02a]. Therefore, .NET Remoting does not allow the direct implementation remote objects mobility. Another drawback found in .NET Remoting is that to implement remote objects, the class must inherit from the *MarshalByRef* class. This limits the inheritance flexibility of remote objects because they cannot inherit from other classes as .NET does not offer multiple inheritance.

In this paper, we are going to present an approach for supporting distributed communication and mobility tolerance in a transparent way for .NET objects. The implementation of this approach is based on a concept called attachments offered by the PRISMA approach. PRISMA is an aspect-oriented component-based approach where attachments allow the transparent communication among components. In order to support the PRISMA approach, a PRISMANET [Per05a] middleware has been implemented. Based on the experience gained from this approach, we noticed that the attachment functionality could be extended to support

transparent distributed communication and mobility for objects. Thus, the implementation presented in this paper can adapt object-oriented applications that were not initially designed to be distributed and mobile in order to obtain this functionality.

Our approach is based on separating the distribution and mobile concerns from the source code (which contains the application logic) in entities called attachments. Thus, the attachments are high-level proxies that are responsible for creating communication channels and are capable of managing dynamic location changes without affecting the objects in the case of mobility.

The structure of the paper is as follows: Section 2 presents some works that offer transparent distributed communication and mobility of objects. Section 3, explains the attachments concept and the implementation of our approach by using a case study of distributed mobile agents. Section 4, evaluates the communication costs introduced by our approach compared with the .NET Remoting framework. Finally, conclusions are presented in Section 5.

2. RELATED WORKS

The work in this paper is focused on providing an approach that allows objects to be accessible remotely and to be moved from one location to another during run-time.

Mobility is classified by Picco [Fug98a] into weak and strong mobility. Weak mobility involves the migration of the code and data of an object. In weak mobility, before interrupting the object for migration, the developer has to make sure that the object's threads have finalized their tasks. Strong mobility involves the migration of the code and the execution state (stack, program counter …). In strong mobility, mobile object execution is only interrupted for migration. Once the object has been migrated to its destination, it continues to execute from the interrupted point. However, strong mobility is difficult to implement as it greatly depends on the .NET CLR internals. In order to interrupt a thread in a transparent way, and to be able to restore it in a new destination, the following actions must be performed. On the one hand, we must be able to obtain the instruction pointer and the execution context of the threads to be moved. On the other hand, we must also be able to restore a thread from an instruction pointer and its thread context. In other words, to implement strong mobility, we must be able to serialize threads, which is not currently available in .NET. For these reasons, our approach is designed to provide weak mobility and not strong mobility.

Approaches that deal with communication transparency have been dealt mostly in Java. The work in [Hic99a] provides a run-time system and a compiler that generates remote references. This work requires having a process on each physical machine. Each of these processes has: a set of caches that maps object IDs to instances, a cache for the local instances, and a cache for each remote process filled with the instance references that are needed locally. A drawback of this approach is that the programmer must indicate where an instance is created, since the objects are always allocated in the same process (physical machine), and there is no way to change the references in the caches.

MobJeX [Rya04a] is a Java-based application framework that allows weak mobility as well as remote accessibility of objects. This is obtained by precompiling the mobile objects in order to generate two interfaces: a remote interface and a local interface. Two classes are also generated: a proxy class which provides a client with the reference to the server, and a serializable class which represents the original class that implements the two interfaces. In our approach, no precompilation is necessary; however, all mobile objects should be serializable classes. Another difference between MobJeX and our approach is that the mobility requests in MobJeX cannot be caused by the same object; they must be caused by a system controller. This eliminates the possibility for mobile objects to be autonomous. Also, if a MobJeX server object is moved a chain of calls is produced in order to find out its new location since the proxy object is not notified of the change directly. However, in our approach, the proxy is updated directly to the new location of the object. Another limitation in MobJeX is that it does not support the declaration of static methods in mobile objects. This is because it only supports interfaces to be shared between the client and the server. In our approach, it is up to the developer to choose between shared interfaces or classes.

Another approach that deals with mobility in Java is the Active Container approach [Cha03a]. This approach provides a compiler that dynamically generates the code for storing objects in containers. The communication among objects is made transparent by calling the active container. However, the mechanism of changing the proxy when the server moves is not described. To move an object, it is also necessary to indicate both the active container of the stored object and the new active container. This reduces mobility transparency and does not allow objects to self-initiate mobility.

One of the few works performed in the context of .NET is [Tro03a]. It provides weak mobility as our

approach. It uses Aspect-Oriented Programming (AOP) to separate the mobility decisions from the objects code in order to allow objects to self-initiate the mobility decisions. Location changes that are caused by mobility are transparent to objects because a module is provided that forwards requests to find out object locations. In our approach, no forwarding requests are needed since the location references are dynamically updated. It is also important to comment that our approach can also use AOP. Thus, the PRISMANET middleware [Per05a] supports mobility and distribution of aspect-oriented components. However, since AOP is not standardized in the .NET framework [Per05a] the work presented in this paper does not use AOP.

3. AN OBJECT-ORIENTED APPROACH FOR TRANSPARENT COMMUNICATION AND MOBILITY

In the following, we present an overview of the background on which our approach is based. We then explain our approach using a case study of mobile agents.

Attachment Overview

3.1.1 The Attachments in PRISMA

PRISMA [Ali05a] is an approach that allows the construction of complex, reusable, dynamic, and distributed architectures by interconnecting architectural elements. Thus, an architectural element must only request and receive petitions through ports of an interface. However, an architectural element instance is unaware of with whom it is interacting, and how the interaction is being performed. This allows the architectural elements to communicate in a transparent way thanks to the attachment functionality.

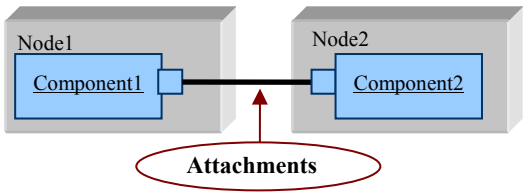

Figure 1 Attachments in a distributed software architecture

Attachments (see Figure 1) are the artefacts that are responsible for the connections among the ports of the architectural elements instances. This way, the attachments can connect architectural elements whether they are distributed or not. In addition, if an instance moves, it is the attachments that change the references and not the architectural element. The PRISMA approach has been implemented using .NET through the PRISMANET middleware [Per05a]. In order to offer the attachment functionality not only to a component based approach but also to object-oriented approaches, the attachments implementation in [Per05a] has been adapted to provide a middleware to connect mobile .NET objects.

3.1.2 Design of the Attachment Approach for .NET Objects

Our middleware permits client objects and server objects to communicate locally or remotely in a transparent way. In addition, the client and server objects can be mobile. Therefore, these objects must be serializable.

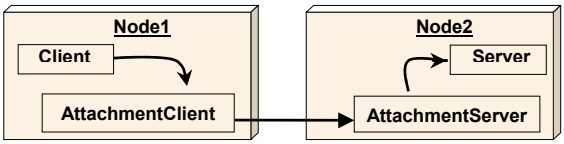

Figure 2 Attachment structure

Figure 2 shows the design of a communication between a client object and a server object in the attachment approach. The communication transparency is performed because each client has a reference to an *AttachmentClient* instance. An *AttachmentClient* instance is always local to the client object. The responsibility of the *AttachmentClient* is to redirect the client's requests to an *AttachmentServer* object. The *AttachmentServer* object is always local to a server. Therefore, depending on whether the server object is local or remote to the client the *AttachmentClient* object may or may not make a remote call. Therefore, for the cases where the server must be accessible remotely, the *AttachmentServer* class inherits from *MarshalByRef* class, as is specified by .NET Remoting technology.

In this approach, the client object always sends requests locally to the *AttachmentClient* object and does not have to take into account the location of the server. Thus, if the server object moves, it is the *AttachmentClient* that must change its references. In addition, as the *AttachmentServer* object is of *MarshalByRef* type the server object does not have to publish its services by .NET Remoting. This solves the problem that objects cannot be both serializable and *MarshalByRef*.

Distributed mobile agent case study

In order to explain the application of this approach, we present a case study of distributed mobile agents. The case study lies in a system composed of several distributed databases, of which we need to collect information. Mobile agents are sent to the databases in order to perform local searches, and then they return to their source host to process the search results.

In many situations the search might have to be done in large and complex systems, such as the Internet, where it is appropiate to use as many agents as sites to search. For this reason, we decided to use a small number of agents that perform the search. These agents are distributed dynamically among databases depending on their search results. The first agent to finish its work moves to the next database and notifies the other agents so that they do not process the same database twice. This solution requires each agent to be capable of moving in an autonomous way and also to be connected with the other distributed agents in order to share services and information.

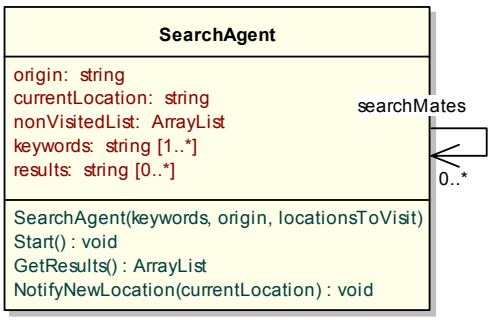

Figure 3 SearchAgent class

The SearchAgent class is defined in Figure 3. Each agent requires a list of keywords for the search, its host origin, and the initial database list where the search is to be performed. The *Start()* method is invoked to search in a current database. After an agent finishes its search, it needs to move to the next unvisited database. Then, it notifies the other agents of its new location by invoking the *NotifyNew Location()* method. It is important to note that each agent could be in a different location each time. Finally, when there are no more locations to visit, each agent returns to its host origin and all the collected data is processed.

Applying the Attachment Approach to .NET Remoting

Our approach provides a lightweight middleware to build distributed applications with the following features:

- Objects can move autonomously among computers without having to take into account how distributed communications with other objects are performed.
- Objects use the middleware to:
  - Register themselves in order to offer their services to other objects,
  - Request the creation of a connection to objects to use their services,
  - Ask for mobility when they need it.
- There is no need for a centralized infrastructure to manage these mobility and object registration services. The infrastructure has been designed in a decentralized way.
- Neither client nor server objects need to precompile code as in other approaches, because reflection and code generation is used.

The communication infrastructure is built on .NET Remoting in a transparent way. The additional communication cost introduced between two objects depends on the network traffic and the derived costs of invocation methods through delegates.

However, this approach requires a few constraints:

- Every computer must run this middleware in order to use mobility and object-registration services.
- A client object needs to know where the server object is located when it establishes the connection. However, location-awareness is provided since connection is established.
- Due to the fact that the middleware provides weak mobility implementation, objects must take care of their threads before moving. When the object is restored in the new location, an initialization method can be provided to initialize new threads at a specific point.
- In order to support the mobility of the object state, both client and server object classes must be marked as *Serializable*.

In the following sections we explain the implementation of our approach using the case study presented in the previous subsection.

3.2.1 The AttachmentManager class

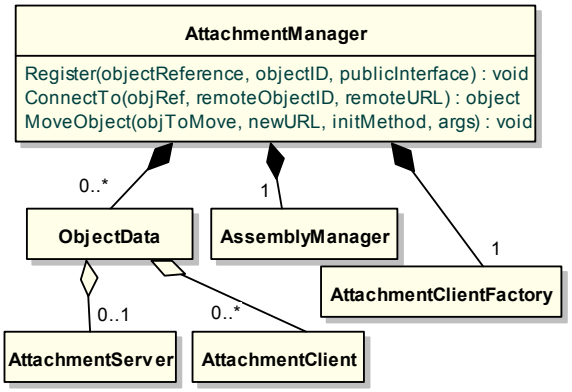

Figure 4 AttachmentManager class

The *Attachment Manager* class (see Figure 4), is the main class of our middleware, and must be running on each computer in order to offer the following services:

- server-behaviour registration services,
- client-behaviour connection services,
- mobility services

- transference of required assemblies when mobility takes place
- dynamic generation of server proxies on demand

For each object that uses the attachment concept, the *AttachmentManager* maintains an *ObjectData* structure that contains information about the attachments that are used. On the one hand, if an object provides services to other objects (that is, it acts with server behaviour), it will have an *AttachmentServer* associated to it. On the other hand, if an object requires services from other objects (that is, it acts with client behaviour), it will have an *AttachmentClient* associated to it.

In our case study, a SearchAgent object has both client behaviour and server behaviour. On the one hand, it needs to notify its new location when it arrives to a new site; i.e. it invokes *NotifyNew Location()* method of other SearchAgents. On the other hand, it must be notified about sites being visited by other SearchAgents; i.e. it provides the *NotifyNewLocation()* method to be invoked remotely.

3.2.2 Server behaviour

A *SearchAgent* object (from now on, the Server object) invokes the *Register()* service of the *AttachmentManager* class in order to be accessible remotely. The following parameters are needed:

- *object reference*: reference of server object, which will be used to create the *AttachmentServer* part.
- *objectID*: custom ID to uniquely identify a server object. This must be known by each client object in order to establish a proper connection.
- *publicInterface*: an optional parameter that allows us to restrict services that would be offered to clients. Otherwise, all services from the server object are provided.

As a result of this invocation, an *AttachmentServer* object is created and made accessible remotely (see Figure 5). This object represents the SearchAgent object and is responsible for offering the following services:

- incoming request services are forwarded towards the server object.
- mobility notification of the server object to client objects that are connected to it.

The *AttachmentServer* is composed by the *AttachmentServerMediator* class, who publishes the services that can be invoked remotely and is responsible for invoking Server methods.

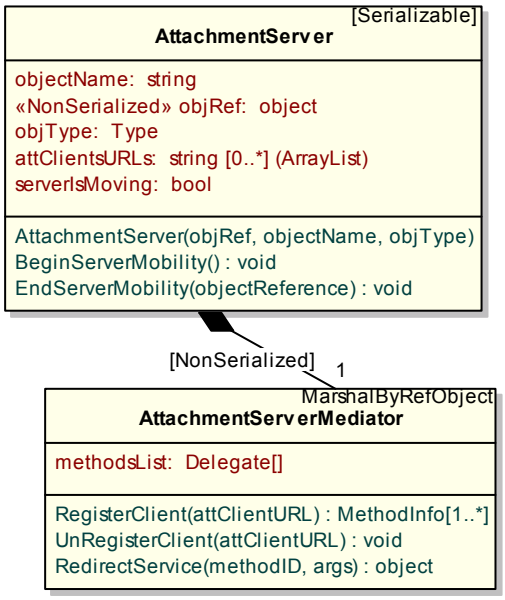

Figure 5 AttachmentServer class

Due to the fact that the method signatures of the Server are not known until runtime, direct call invocation cannot be used. We had to use dynamic method invocation. We decided not to do this through reflection (using Type.InvokeMember()) because it has the worst performance [Gunn04a]. Instead of this, we have used dynamic code calling through Delegates. When *AttachmentServer* is created, a delegate is created for each method provided by the server, following these steps:

1. Method information is obtained by means of reflection at runtime. With this information, a delegate type is created by emitting its MSIL code.
2. This delegate type is instanced and stored in an array.
3. The index of the array where the delegate is stored is used to uniquely identify the method to be executed. We have called it *MethodID*. This index is stored together with related method information in a structure called *MethodInfo*.

Thus, clients forward methods by the invocation of the *RedirectService()* method and by providing the correct *MethodID* of the delegate to be executed. We chose this alternative in order to avoid searches in the delegate list, which can slow method invocation. Clients get all the *MethodID*s and their related information (*MethodInfo* list) when they subscribe to the *AttachmentServer* through the *RegisterClient()* method. Moreover, client subscription to the *AttachmentServer* provides a way to be notified when the server is moving.

3.2.3 *Client behaviour*

A SearchAgent (the client) that wants to call methods from another object (remote or local) needs the reference of this object to do that. This reference is provided by the *ConnectTo()* service of the *AttachmentManager* class. The *objectID* and its current location must be provided in order to get its reference. The reference provided is in fact an *AttachmentClient* that acts as a proxy. From now on, the client object will not have to take care of distributed communications nor location changes of the remote object (the server).

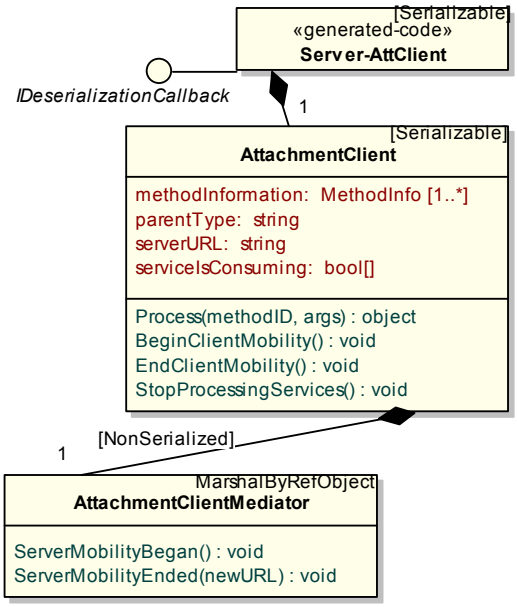

Figure 6 AttachmentClient classes

The creation of the *AttachmentClient* is done in several steps:

- If the client computer does not have the assemblies of the server object, it downloads them from the computer where the server is located at this point in time.
- An *AttachmentClient* object is created. It registers itself in the *AttachmentServer Mediator* of the server object. Thus, it obtains method information about available remote services.
- With this information, a proxy of the server is generated at runtime. The purpose of this proxy is to forward called methods through the infrastructure of attachments in a transparent way. We call it *Server-AttClient*, although its real name will depend on the server type that it represents.
- An instance of the generated *Server-AttClient* is returned to the client object.

The *Server-AttClient* class is generated by emitting MSIL code. It can be created in two ways: by implementing a specified server interface or by inheriting the server type. When it is instanced, a reference to an *AttachmentClient* object is provided, to which methods are forwarded. For each method, the generated code looks like this:

```
void NotifyNewLocation(string currentLocation) {
    object[] args =
         new object[1] {currentlocation};
    MethodID = 2;
    attClient.Process(MethodID, args); }
```

Each method has its related *MethodID* defined at generation time in order to provide it correctly to the *AttachmentServerMediator*. In .NET Remoting, by creating a derived class from the *RealProxy* class, proxies can be built in an easy way instead of emitting MSIL code. However, we cannot use this feature because this infrastructure only accepts objects that inherit from the *MarshalByRef* class. To support mobility, our generated proxy must be serializable, as discussed in section 3.2.4.

In order to minimize generated MSIL code, the *Server-AttClient* class is composed of an *AttachmentClient* class that defines all the functionality of method forwarding and mobility. The *Process()* method is responsible for forwarding the services to be executed to the *AttachmentServer Mediator*. Finally, the *AttachmentClientMediator* class contains the services that *AttachmentServer* is going to invoke in order to notify its mobility, which will be discussed below.

To illustrate, we describe how SearchAgents are created and connected with each other following our approach. First, SearchAgents are created in the host origin and registered in the *AttachmentManager* by providing a different *objectID* for each one. Next, they are connected to each other through the *ConnectTo* service and by providing the *objectID*s obtained in the previous step. Finally, the *Start()* method of the SearchAgents are invoked, so they will begin to move to remote databases to collect information.

3.2.4 *Object mobility*

In order to move an entire object (code + state) to a new host, the *AttachmentManager* class provides the *MoveObject()* service. As mentioned above, there must be an *AttachmentManager* object running at the target host in order to be able to receive the object and restore its state properly. The *MoveObject* service moves the specified object to the new specified computer taking into account the current communication processes. Communication processes are "frozen" while mobility takes place, and they are restored properly when mobility ends. Thus, the other objects to which the moved object was connected to are not aware of the mobility process. Moreover, an object can request to move itself autonomously. In this case, the object thread that

requested the mobility is aborted when the mobility begins.

It is important to note that, in order to provide the objects with a high level of mobility transparency, we considered the objects as black boxes which we do not know anything about (i.e., their threads or the location where remote object references are stored) For this reason, the object to be moved is responsible for finishing all of its executing threads before starting mobility. In other words, the object must reach a secure state before requesting mobility. This cannot be done transparently by the middleware for two reasons. On the one hand, it is difficult to obtain all the running threads of a particular object. On the other hand, it is not possible in .NET (without modifying the CLR) to get the thread execution state (stack and instruction pointer) and to restore it in a new computer. In order to do that, we would need thread serialization capabilities. However, to overcome these limitations, an initialization method and its arguments can be provided to restore the execution state of the object when the mobility process ends.

Mobility is carried out in several steps. First, both the object to be moved and its communication processes (the attachments) are packaged by creating a *MobilityPackage* object (see Figure 7). Second, this object is serialized and transferred to the target host. Then, before deserializing the transferred object, the middleware checks whether the required assemblies are available at the current host. If not, they are downloaded from the host where the object comes from. Finally, the *Unpack*() method is invoked to restore the object and the attachments. If anything fails, the service *UndoMovement*() restores the object to its initial location.

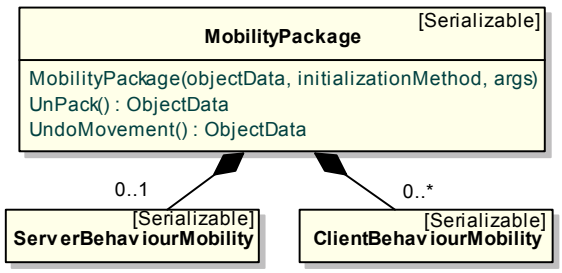

Figure 7 MobilityPackage classes

The mobility process depends on the role of the object to be moved: client or server behaviour. In the case of client behaviour, the *ClientBehaviourMobility* class obtains the *AttachmentClient* data (server location, server type and its unique ID) in order to rebuild it at the target host. This is because *MarshalByRef* objects cannot be serialized, as we stated above. Then, it invokes the *BeginClient Mobility* service of *AttachmentClient* to wait for pending requests to finish properly. Both the *Server-AttClient* and the object are serialized together, so on deserialization the object preserves the *Server-AttClient* reference without forcing the object to provide a setter property to update remote object references. However, as *Server-AttClient* is a dynamic assembly, it must be regenerated at the target host if this was not done before. Finally, at the target host, *EndClientMobility* service is invoked and the connection is restored to the *AttachmentServer* by notifying the new location of the client object.

In the case of server behaviour, each client object must be notified of the server mobility process so that services are not requested during this process. Similar to the *ClientBehaviourMobility* object, the data of the *AttachmentServer* (objectID, objectType and locations of connected *AttachmentClients*) is stored on a *ServerBehaviourMobility* object in order to rebuild it at the target computer. Then, the *ServerBehaviourMobility* object invokes the *BeginServerMobility* service of the *AttachmentServer* to notify the *AttachmentClients* of server mobility. Thus, each *AttachmentClient* blocks the arrival of new requests (by suspending incoming threads) and waits until current processing requests finish. When there are no more requests being processed by the server object, the mobility process can continue. Finally at destination, *EndServerMobility* service is invoked and connection is restored to the *AttachmentClients* by notifying its new location. In such the case that an object has both server and client behaviour, its mobility process will be the union of the above.

Simultaneous mobility is also supported. In other words, an object can move to another host while other objects, that are connected to it, are moved at the same time. Let's suppose that two SearchAgents 'Agnt1' and 'Agnt2' are connected, and 'Agnt1' is being moved. Then, 'Agnt2' also wants to be moved, but if it moves, 'Agnt1' will not be able to connect to it when 'Agnt1' ends its move. In order to do this, a message is left in the host where 'Agnt2' was. When 'Agnt1' ends its move and tries to connect to the last location of 'Agnt2', it will be notified with the new 'Agnt2' location.

In the SearchAgents case study, mobility takes place when an agent finishes collecting data at a certain database. Then, it invokes the *MoveObject*() service by specifying the next unvisited database where it wants to move and the service to be called when the mobility process ends (the *Start* method). When it arrives to the new database, the *Start* method is executed, and the SearchAgent continues its data collecting process.

4. EVALUATION AND RESULTS

Our approach has been implemented to compare the communication costs added by the attachments. We have measured these costs from when a client object requests a service until the results are returned. Without attachments, the average communication costs on a 100Mbit LAN are 0.9030ms. With the attachments (in the same conditions), the average communication costs are 1.0144ms (10.98%). The additional costs introduced, are due to 3 direct calls + 1 delegate dynamic invocation. Therefore, costs are increased because of dynamic invocation costs. For this reason, we also evaluated the performance by using a dynamically generated custom class [Gunn04a] instead of using delegates. This class was invoked by the *AttachmentServerMediator* in order to make direct method calls to the server object. Thus, the average costs have been reduced: 1.0010ms (9.79%). In the case of mobility, the costs are higher: there are communication and processing costs. The object, its related attachments, and the required assemblies are transferred. There are also several notification messages. The most important processing costs are due to the deserialization of transferred data and to the dynamic generation of *Server-AttClient* types.

5. CONCLUSIONS

In this paper, we have presented a lightweight middleware that can be easily included in other middlewares to provide mobility capabilities to its objects. Our approach supports weak mobility by using the attachments concept. Autonomous mobility for distributed objects is provided transparently and simultaneously, so the objects are not aware of the mobility process or the connection process of other objects to which they are linked. Moreover, the communication costs introduced are not very high, so an application can be mobility-adapted easily without slowing its performance. However, there are a few constraints. First, mobile objects must be *Serializable,* and they must manage their own threads before moving. Also, in order to establish the initial connection to a remote object, its current location must be known in advance. Nevertheless, our approach provides location-awareness after establishing a connection. The most common solution to obtain current locations of mobile objects is by having a centralized object that is updated with location changes from which clients can request these locations. However, this is not a decentralized approach and more work has to be done.

Furthermore, attachments add an abstraction layer over the communication infrastructure, so objects do not have to take into account what technology is used. Therefore, even though we implemented our approach in .NET Remoting, in the future we can adapt it to a Service-Oriented infrastructure such as Indigo, so that current running objects do not have to be aware of the underlying technology.

6. ACKNOWLEDGMENTS

This work has been funded by the Department of Science and Technology (Spain) under the National Program for Research, Development and Innovation, DYNAMICA project TIC2003-07776-C02-02.

7. REFERENCES

[Ali05a] Ali, N., Ramos, I., Carsí, J.A. *A Conceptual Model for Distributed Aspect-Oriented Software Architectures*. In International conf. on Information Technology Coding and Computing, (ITCC 2005), IEEE Computer Society, Las Vegas, USA 2005.

[Ber96a] Bernstein, P.A. *Middleware: a model for distributed system services*. Communications of the ACM, Volume 39, Issue 2, ISSN: 0001-0782, 86-98, 1996.

[Cha03a] Chaumette, S. and Vignéras, P. *A Framework for Seamlesly Making Object Oriented Applications Distributed*. In International conf. on Parallel Computing (PARCO 2003): 305-312, 2003.

[Fug98a] Fuggetta, A., Picco, G.P., and Vigna, G. *Understanding Code Mobility*. In IEEE Transactions on Software Engineering, 24(5): 342-361, 1998.

[Gunn04a] Gunnerson, E. *Calling Code Dynamically*. MSDN Library, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncscol/html/csharp02172004.asp, 2004

[Hic99a] Hicks, M., Jagannathan, S., Kesley, R., Moore, J.T. and Ungureanu, C. *Transparent Communication for Distributed Objects in Java*. In ACM Java Grande Conference, 160-170, June 1999.

[Obe02a] Obermeyer, P. and Hawkins, J. *Object Serialization in the .NET Framework*. MSDNLib.: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/objserializ.asp, 2002.

[Per05a] Pérez, J., Ali, N., Costa C., Carsí J.A., Ramos I. *Executing Aspect-Oriented Component-Based Software Architectures on .NET Technology*. International Conference on .NET Technologies, Plzen, Pilsen, Czech Republic, 2005.

[Rya04a] Ryan, C. and Westhorpe, C. *Application Adaptation through Transparent and Portable Object Mobility in Java*. In proc. of 2004 International Symposium on Distributed Objects and Applications (DOA 2004), Agia Napa, Cyprus, 2004, Springer-Verlag LNCS3291

[Tro03a] Troger, P. and Polze, A. *Object and Process Migration in .NET*. The 8th IEEE Intern. Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2003), Mexico, January 2003.