HDE - A Heterogeneous Object-oriented Distributed Environment

Part II

Adrian Mircea
Computer Sharing Romania
15-17, Calea Dorobanti,
71131 Bucharest
ROMANIA

Ion Stoica
Distributed Processing Systems Laboratory
Research Institute for Informatics
8-10 Averescu Avenue,
71316 Bucharest
ROMANIA

Abstract: The "object-oriented" and "distributed" paradigms have now matured and are present in several areas of computing, from operating systems, databases, programming languages, graphical interfaces to application environments. These topics have profound impact on heterogeneous computing environments requiring new methods of organization, exploration and design. The paper describes the architecture and some implementation strategies of HDE (Heterogeneous Distributed Object-oriented Environment), an integrated support environment for designing, implementing and executing distributed applications built up from object entities. The objects may be mobile, distributed, replicated, fragmented and persistent. From the execution point of view, the objects may be either passive or active with an implicit, and at once explicit control of parallelism. The object model allows a language independent implementation, although a C+ + extension is considered for the application development. HDE may be thought as (just another) distributed object-oriented system prototyped on heterogeneous OS networks (basically UNIX/AIX, OS/2 HDE kernels are under development), allowing the building of fault-tolerant distributed applications that efficiently exploit the parallelism, replication and/or distribution, with a great degree of flexibility and a reasonable transparency-performance trade-off.
The paper has been split into two parts and the introduction has been maintained for bridging them. Chapters are therefore numbered in their continuation from the first part (see Vol. 1, No. 4/December 1992).

Keywords: Distributed object-oriented operating systems, reliable broadcast protocols, fault-tolerance.

Adrian Mircea was born in Romania in 1954. He received his M.Sc. degree in computer science from the Polytechnical Institute of Bucharest in 1979. He worked at the Research Institute for Informatics in Bucharest from 1979 to 1991 as senior researcher and head of Distributed Processing Systems Research Laboratory. Since 1991 he has been technical manager at Computer Sharing Romania, there managing a research project for a multiplatform development environment (UNIX/AIX,OS/2, Windows). His topics of interest include distributed processing systems, CASE systems, object-oriented design and programming. He has been author or co-author of more than 40 papers published in national and international journals. He is a member of DECUS.

Ion Stoica was born in 1965. He graduated the Polytechnical Institute of Bucharest-Computer Science Department in 1989. Since graduation he has worked at the Research Institute for Informatics in Bucharest. He works in the Distributed Processing Systems Laboratory and his main research activity covers distributed systems and neural networks. He is preparing a doctoral thesis on neural networks at the Polytechnical Institute of Bucharest.

Introduction

The "object-oriented" and "distributed" paradigms, considered separately, have now matured and are present in several areas of computing, from operating systems, databases, programming languages, graphical interfaces to application environments. The combination of these two models makes the challenge of the '90s, which will be the cooperative, team working way of computing for both the application development and the actual application processing, by allowing sharing, distribution and replication of objects (not only files, memory or other classical resources). Besides, they impose new methods of organization, exploration and design for both the system and applications.

The paper is the result of years of experience in developing both classical distributed system support and object-oriented environments. It describes the architecture and some implementation strategies of HDE (Heterogeneous Distributed Object-oriented Environment), an integrated support environment for the design, implementation and execution of distributed application built up from object entities.

There are a lot of important DO-O OS (distributed object- oriented operating systems) prototypes in the research community or industry, some of which pioneered the field and some others that have matured it (see Part I overview). One of the main features associating or making them distinct is the

object model, which covers the aspects of distribution and mobility, the granularity level or the execution structure. Although there is a general agreement on the terminology, as some other system presentations do, we will however attempt to explain our own acceptance of the related terms. So, for the full understanding of HDE, the meaning of HDE object attributes must be accepted as stated in section III.

These attributes refer to: migration (solving the problem of object mobility), fragmentation (as a way of distributing processing inside an object along the class hierarchy path), distribution (which denotes only the existence of consistent multiple copies of the object), replication (as a way of solving the fault-tolerance problem) and persistence. Generally, replication is used for enhancing data availability under failure conditions, and that is the semantics of HDE replicated objects. The use of replication for higher efficiency in sharing data by having a local consistent copy of it reflects the semantics of HDE distributed objects (in the previous case the copy may not be on the user's site).

HDE allows the structuring of applications as collections of both passive and active objects, with a different degree of granularity and parallelism. The parallelism may be hidden to the object user, but also may be explicitly available to the class hierarchy implementer.

In order to support the features related with the distribution (in a general acceptance) of the objects, the HDE architecture has a custom protocol set, generally available to the user only via the HDE object behaviour. An informal presentation of HDE protocol architecture is made in Part I. These protocols are oriented towards the group communication paradigm (at both site and object levels, with a variable degree of reliability).

A peculiar feature of HDE is its possibility to get integrated into an application (normally built up only fromHDE objects and contexts) programs written in languages and with rules unrelated to HDE. These external programs are encapsulated in the so-called "foreign objects" of which many features are common to those of the normal HDE objects.

Section IV presents the basic strategies (irrespective of the underlying OS) of the HDE class and object management implementation.It details HDE object types and emphasizes the mechanism for naming and handling classes and objects. The presentation is made by direct reference to the HDE kernel interface.

An HDE application may be written in C, C++ or DC++ (or a combination of them) with the largest transparency of the distribution aspects in DC++. Section V informally presents the DC++ syntactic extensions and some semantic ones. It flexibly encapsulates the active/passive trade-off of the DO-O languages and allows new paradigms for distributed application design (see the distributed data structure [Tanenbaum 89], the replicated workers [Kaashoek 89a], etc.). To sustain these ideas a DC++ implementation of LINDA [Carriero 89] mechanisms is proposed in section VI.

IV. Class and Object Management

This section presents the basic strategies of the HDE class and object managment implementation. It details the HDE objects types and puts an emphasis on the mechanism for naming and handling the classes and objects. The presentation makes direct reference to the HDE kernel interface.

IV.1 Class and Method Registration

An HDE class represents a collection of functions (methods) that manipulates a common set of data. Two of the methods have special meanings, constructor and destructor. The constructor allocates the resources and initializes on object creation, while the destructor releases the allocated resources when the object is destroyed. To be accessible to any potential client (in the same domain) an HDE class must be registered. When a class is registered, the following information goes to the HDE kernel:

- class name. This is a string used as a class identifier;
- list of base classes which the current class is derived from;
- reference to the class constructor. This reference is used by the kernel to invoke the constructor method when an object is created;
- reference to the class destructor;
- format of the parameters passed on to the constructor invocation. This information is used by the kernel for parameter marshalling and unmarshalling in heterogeneous systems, and for dynamic parameter type checking.

In order to register an HDE class in C language, the following call must be used:

```
int RegisterClass(char *pchClassName, char
*pchSupperclassList,
.                                   void * (*pfConstructor)(),
int (*pfDestructor)(),
.                            char *pchParameterFormat);
```

After a class is registered, all class methods that are allowed to be invoked by a client must also be registered. On method registration, the following information must be specified:

- method type;
- method name;
- class name. This is a class which the method is associated with ;
- reference to the method. This reference is used by the kernel to execute the method when invoked;
- format of the parameters passed on to the method invocation.

A method type tells HDE kernel how to execute the method. The following types of methods can be distinguished in HDE:

- READ-specifies that the method invocation modifies none of the instance variables. This parameter is very important for distributed objects composed of more than one copy. In this case the copy consistency is automatically maintained, because a method of READ type does not modify the copy which it is executed upon.
- WRITE - specifies that the method changes some instance variables (changes the object state). Here, for distribu.ed objects, special mechanisms must be provided in order to maintain copy consistency when changed.
- SAVE - specifies that the current method saves the object state (instance variables) on a permanent storage. This method can be invoked explicitly by client or automatically by HDE kernel (for some object types of which semantics asks for this).
- RESTORE - specifies that the current method restores the object state from the permanent storage. As for the previous type, this method can be invoked explicitly by the client or automatically by the HDE kernel.

In C, a method is registered using the following call:

```
int RegisterMethod(int iMethType, char
*pchMethodName, char *pchClassName,
.                       int (*pfMethod)(), char
*pchParameterFormat);
```

Note that an HDE class exports only a set of methods and no variables. In this way, the HDE system enforces a "perfect" encapsulation, so that the client is able to modify or read a variable using but methods.

When a class is registered, the HDE kernel creates a descriptor called class **descriptor (CD)**. This descriptor keeps all the information needed for invoking any class registered method. In a CD, each registered method has associated an identifier and a reference to its implementation (this information is supplied on class registration). The symbolic name is used by the client to logically identify the desired method, while the method reference is used by the HDE kernel for performing the method invocation. Also, CDs are hierarchically organized to allow class inheritance. To illustrate this, suppose that an object O of class C is instantiated and method M of this object is invoked. First, HDE searches for the method M through the registered methods of class C. If the method can't be found, then HDE keeps searching through the ancestor classes (within the same context and later, within other contexts on the same site or on remote sites, if the object is fragmented) until the method is found.

IV.2 Object Creation

An HDE object is an instance of some pre-registered class and is created by calling on the HDE kernel by means of the following parameters:

- object type. This parameter describes what type of an object is to be created (i.e. distributed, replicable, etc.);
- site name;
- object name. This is a string used as an object identifier;
- class name. This is a string that identifies the class where the object is instantiated;
- format of the parameters passed on to the constructor. This describes the parameters format which must be provided to the constructor;
- list of parameters passed on to the constructor;

The "site name" parameter identifies the site

where the object will be instantiated. If this parameter is missing, the kernel tries to instantiate the object on a local site. If the desired class is not available to the local site, the kernel tries to find a site (for which the class is available) in the domain, where the object can be created.

When an object is instantiated, the HDE kernel creates a Remote Object Descriptor (ROD) on the class resident site (where the object is actually created) and a Local Object Descriptor (LOD) on the client resident site (Figure 12.a). After an object is created, the kernel returns a LOD reference (that actually identifies the object) to the caller. Using this reference (which is called the "object handle"), the client can transparently access the object, irrespective of the object type and location (which may be changed with a migration). If the object class server and the client are resident on the same site, only the LOD is created (Figure 12.b).

To create an HDE object in C language, the following call must be used:

```
HOBJ CreateObj(int iObjType, char *pchSiteName, char *pchObjName,
.                    char *pchClassName, char *pchParameterFormat, ...);
```

IV.3 Object Naming

In order to take full advantage of the distributed system environment, it is mandatory that more than one client share the same object.

For this, HDE can associate a symbolic name with an HDE object. This can be done on object creation by supplying the "object name"

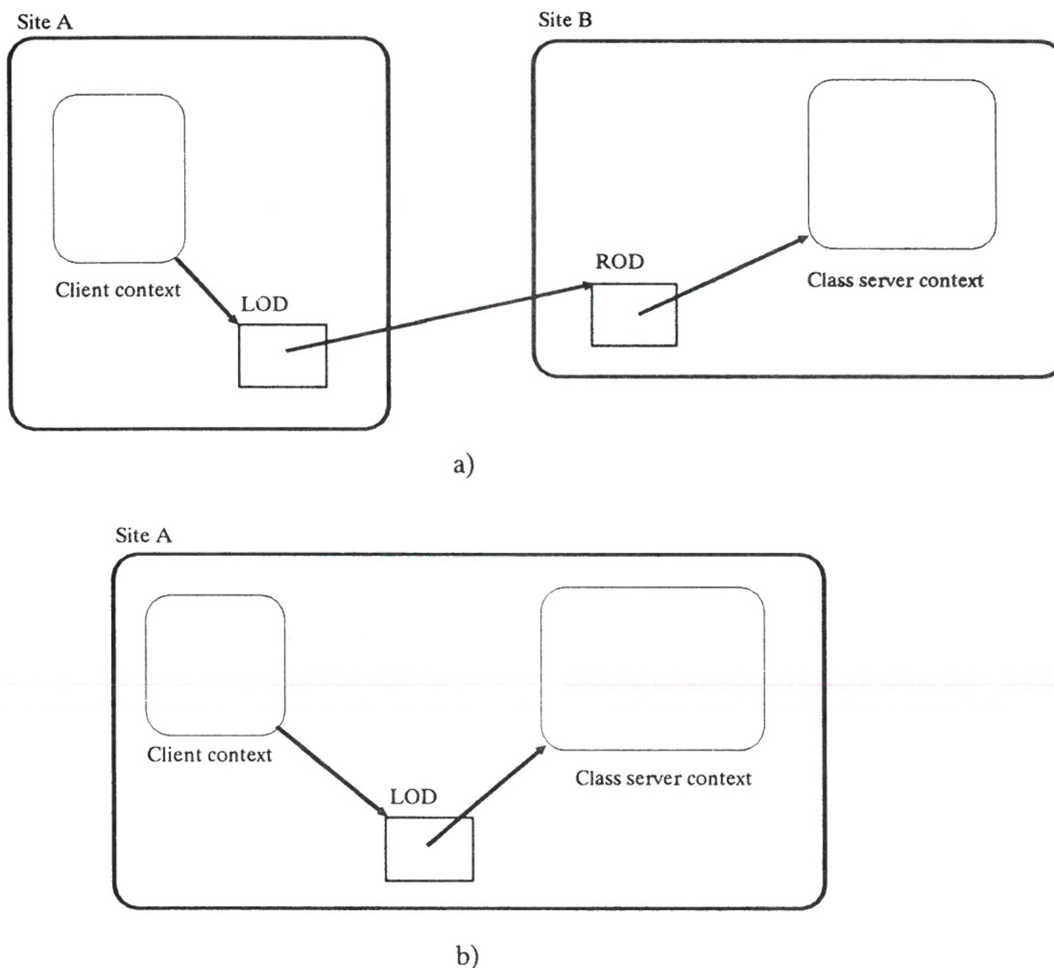

Figure 12. Object creation

parameter, or later by using a call for registering the name (**RegisterName**).

Once an object name registered, any client in the same application (or domain) can access the object. An object that can be invoked by any client in the same application is called a **shared** object, while an object that can be invoked by any client in the same domain is called a **global** object. A global object may or may not be shared.

The **global object name** is unique across all sites within the same domain, while the **shared object name** is unique only inside the same application. A global object can be referred from another domain by appending the domain name to the symbolic name.

For invoking an object method, one should specify in some way the object, the method (see the next section) and pass on some parameters. Now, we will dwell upon identifying the object. As already pointed out, an object can be referred to by using the local handle or the symbolic name . HDE provides a service (**LinkObj**) for associating a local handle with an existing global object. When this function is used, a LOD is automatically created on the client resident site.

Let's take an example for realizing how the mechanism works. Suppose a client C1 instantiates a global object O, and a client C2 (in the same domain) wants to get access to the object. If C2 uses the object symbolic name to access it the following algorithm holds:

* if O is not resident on the same site as C2 client or LOD is not already present on its site, HDE broadcasts a message to all sites within the domain, asking about object O;
* the site on which the object O is resident responds to the caller site with a message containing all the necessary information to locate the object;
* finally, the kernel delivers the message to the destination.

Using a symbolic name is a flexible way of accessing an object, but it implies some overhead to search through all LOD and ROD lists in order to acquire the reference to that object. This method can be used when message traffic between client and object is not so heavy.

That is why, HDE offers a better alternative to getting access to the desired object through a local handle. In this case, the client, before invoking any method of object O, must query the kernel for the local handle of object O (making use of the system function **LinkObj**).

This is a more efficient approach to invoking an object method, because the object LOD is much faster to obtain (actually, as mentioned above, the local handle can be the address of the LOD structure). We say that a client requiring the local handle for a global object **is linked** to that object.

IV.4 Method Invocation

When a client invokes an object method it must specify the following information:

- object identifier. This can be either the object handle or the object name;
- method identifier. This can be either the method handle or the method name;
- format of the parameters passed on to the method. This information is used by the kernel for marshalling and unmarshalling parameters in heterogeneous systems and for dynamic parameter type checking.
- list of parameters passed on to the method;

As already said, an object can be identified by its local handle or by its symbolic name. In a similar way, a method can be identified using a symbolic name or a method selector. The method selector can be obtained calling the **QueryMethSel** HDE service. Using the method name, the kernel is summoned to search through the class hierarchy for finding out the desired method. On the other hand, using the method handle is rather complex, but the operation is more efficiently.

As an example, in C language, a method is invoked using the following call (here both object and method are identified by their symbolic names):

```
int ExecMethod(char *pchObjName, char *pchMethodName,
                    char *pchParameterFormat, ...);
```

IV. 5 Object Migration

A migrable object is an object that can be moved from one site to another. An object can only migrate between two sites in the same domain. HDE requires that on the destination site a class server context for migrable object is present. When an object is migrated, the HDE kernel

automatically invokes object methods of SAVE and RESTORE type.

The object migration is transparent from a client's point of view. So far, two ways of referring an object have been identified: by handle or by its symbolic name, so that the following situations should turn up.

First, let us take an object with no symbolic name and only referred through the local handle and suppose that the object migrates to another site. This object is only known at the client's that has instantiated it. In this case, after migration, the HDE kernel automatically updates the necessary information in LOD on the client site. So, the handle to LOD remains unmodified. Now, let us consider an object that could be identified by using both local handle and symbolic name. This object can be referred by more than one client at the same time. In this case, there will be made a different approach. In this situation, when the object migrates, a broadcast message containing the new resident site name and the new remote handle is sent to all the sites in the same domain. As a result, on each site that contains a virtual object associated with the migrated object, the LOD information is automatically updated. Also, HDE provides a complementary solution to updating the LOD contents upon object migration. If the object is not found on the site specified in LOD, a fault appears. The HDE kernel treats it by sending a broadcast message to all the sites in the domain for querying the new location of the object. LOD is then updated with correct information.

IV.6 Object Types

– Distributed Objects

A **distributed object** consists of one or more copies resident on different sites in the same domain. Since this kind of object can be referred by more than one client at the same time, this object must be a global object (with a global name associated with it). When a distributed object is referred (linked) by a new client, HDE automatically creates a new object copy on the client resident site (if no one exists). One main problem to solve is how to keep all object copies consistent when the state of one copy is changed. Thus, if a client invokes a method which modifies some instance variables of one copy, the change must be with all the object copies. When registering a method, and in order to get high performances, the programmer must specify whether that method modifies (WRITE type) or not (READ type) the object state. HDE handles each type of method differently.

So, if a method not changing the object state (READ type) is invoked, then HDE simply delivers the message to the local server that has instantiated the copy. In this case, the client communicates only with the local copy. On the other hand, if a method changing the object state (WRITE type) is invoked, then the HDE kernel broadcasts the current method to all distributed object copies, using a reliable atomic broadcast protocol. As previously pointed out, this protocol guarantees that all copies receive messages in the same order. Thus, all distributed object copies execute the WRITE type methods in the same sequence and so the copy consistency is maintained.

When a new client refers an existing distributed object (Figure 13.a), a copy is migrated (if not there) on the local site. So, new problems arise as to keeping the copy consistency during migration. For this reason, the distributed object class must implement the **Save** and **Restore** methods that are automatically invoked by the kernel. The general algorithm creating a new copy is described below:

* Create a new distributed object class instance on the local site (Figure 13.b).
* Send a message to invalidate all the distributed object copies by using the RABMP service. After having been received this message, an object copy will ignore all further messages (Figure 13.b).
* Invoke **Save** method and transfer the state to the client site (Figure 13.c).
* Invoke **Restore** method on the client site for the newly created object copy (Figure 13.c).
* Send a broadcast message (RABMP) to validate all the distributed object copies (Figure 13.d).

– Replicated Object

A replicated object is an efficient tool for designing fault-tolerant systems. Akin to a distributed object, a replicated object consists of one copy or more copies from different contexts (possibly resident on different nodes in the same domain). One of them is called **master copy,** while the others are called **slave copies**. Any client that communicates with a replicated object sends the messages only to the master copy.

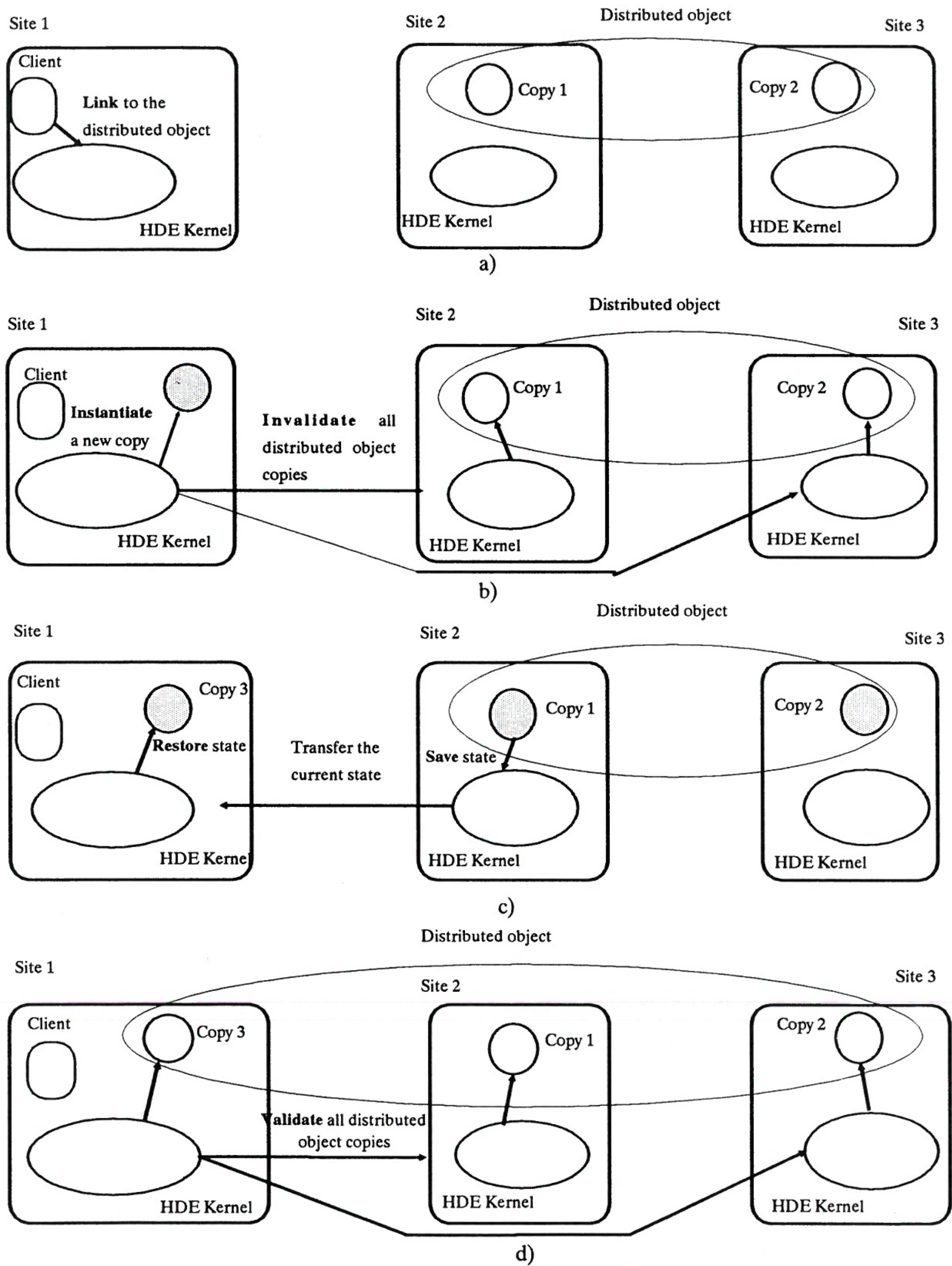

Figure 13. Linking to a distributed object

The slave copies are used transparently by the system to attain a high degree of reliability. We have already detailed the protocols upon which replicated object is built (ROBP). Here, we will only present some implementation elements for this kind of object.

When a replicated object is instantiated, the resilience degree must be specified. Remember that the resilience degree specifies the maximum number of crashes (copy destructions) that can occur at the same time and the object still works. Figure 14 presents a replicated object with the resilience degree 2 (it has 2 slave copies). As one can notice the master copy ROD keeps the references to all slave copies RODs, while the slave copies RODs maintain the back-references to the master copy ROD. These references are used internally by the kernel for the communication protocol and for the recovery process implementation. From a client's point of view, the replicated object is represented only by its master copy. Thus, in LOD, only the reference to the master copy ROD is kept.

When a copy is destroyed, HDE automatically reconfigures the replicated object. Thus, if the master copy crashes (Figure 14.a), then one of the slave copies becomes the new master (Figure 14.b) and the reference to the object contained in LOD is changed. Afterwards, HDE checks if the number of slave copies is larger than or equal to the resilience degree. If not, the HDE kernel will negotiate with other sites in the same domain, and if possible, new copies are being created until the number gets equal to the required resilience degree (Figure 14.c). When a new copy is created, an algorithm similar to the one creating a new distributed object copy appears. So, also for this type of object, the **Save** and **Restore** method must be implemented for being invoked by the HDE kernel during the new copy creation process. As one can see, the recovery mechanism is transparent from the client's point of view, because only LOD contents is modified and not the handle or the symbolic name that represent the object reference inside the client's program.

– Persistent Objects

A persistent object is an object that saves its current state to the permanent storage, whenever it is changed. This is done automatically by HDE kernel invoking the **Save** method. Thus, when a method that changed the internal object state (WRITE type) is invoked, the kernel invokes the **Save** method for recording the new object state on the stable storage. When a context terminates, the persistent objects survive. Therefore, a persistent object must have a global name to be referred independently of any specific context execution. For persistent replicated objects, only the master copy is actually a persistent object, so the other copies do not use the save/restore methods (if not explicitly used in the user code).

In addition to the replicated-object paradigm, the persistent objects provide a natural extension for designing fault-tolerant systems.

When a node crashes, the objects are destroyed from the volatile storage, but the last persistent object state remains recorded on the stable storage. So, when the site is repaired and activated, the persistent object state is automatically recovered from a previously saved state by using the **Restore** method invocation.

– Foreign Objects

HDE has been designed to be an open system, i.e. it allows the integration of other non-HDE programs into an HDE application. For that, HDE let such non-HDE contexts be defined as special kinds of objects named **foreign objects**. The idea of associating such an external context with some sort of objects is the following.

Any program consists of some files on the stable storage, i.e. executable files, data files, configuration files. When that program runs, one or more processes are instantiated. At the limit, we can view the files on the stable storage like a program description, and the process like a program instance. Moreover, processes can communicate with one another by using a specific IPC-like mechanism provided by the operating system. On the other hand, some operating systems provide a de-facto standard interface that can be used by any application intending information exchange. For example, in OS/2 any applications (possibly from different vendors) which conform with such a standard as DDE (Dynamic Data Exchange) specification, can interchange data. So, the problem will be to incorporate these facilities into HDE system.

The HDE solution is presented in Figure 15. Suppose an application that offers a clear service interface to another application.

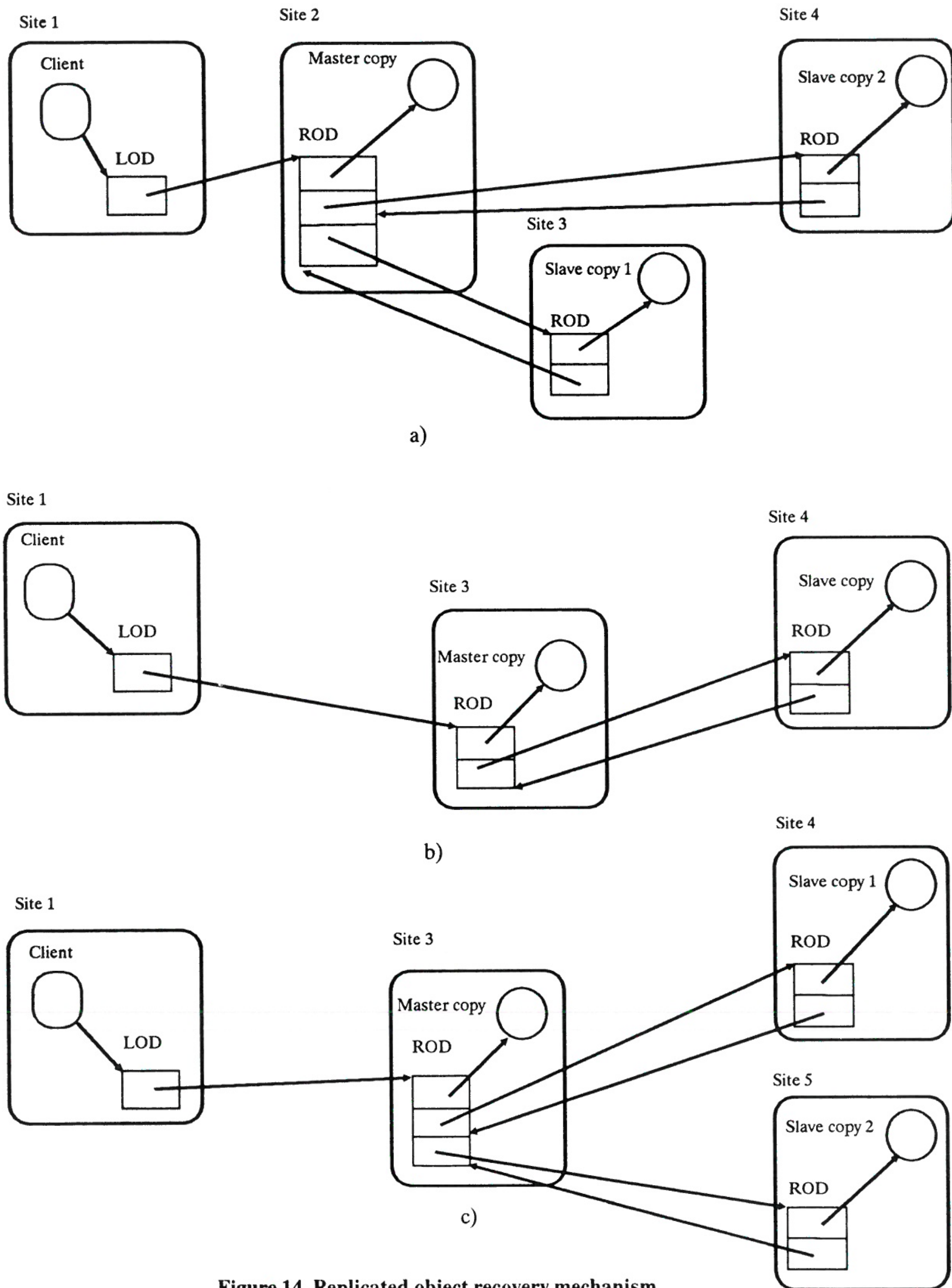

Figure 14. Replicated object recovery mechanism

Those services can be used by any application that conforms with the interface specifications. This application can be viewed as a server providing some services, while the applications that use those services can be viewed as clients. Our goal is to give HDE clients transparent access to the foreign server services. To achieve this goal, a special class for the foreign server has to be implemented. This class must implement an appropriate interface to the HDE-client and a specific interface to call the foreign application services. When an object is instantiated, a new application copy must be created. Therefore, this HDE-class must support the following method implementation:

* the constructor method instantiates a new program copy. The HDE-object must keep in an instance variable the reference to the new application instance. For example, this can be the process id returned by the system on the new instance creation.
* the destructor method removes (terminates) the current program copy from the memory.
* the other methods translate the arguments into the appropriate format and call the specified service.

Thus, the HDE-client may transparently access the foreign application services by invoking the HDE-object appropriate method.

IV.7 HDE Kernel services

This section is an informal presentation of the HDE functions provided by HDE kernel which must be called in C++/C programs or optionally used in DC++ for handling the HDE- objects. They are also used by HDE when translating a DC++ program. The functions for handling the parallelism of active objects in C++ are tightly related to the underlying OS support, so they are not discussed here. In fact, HDE promotes the use of active objects in DC++.

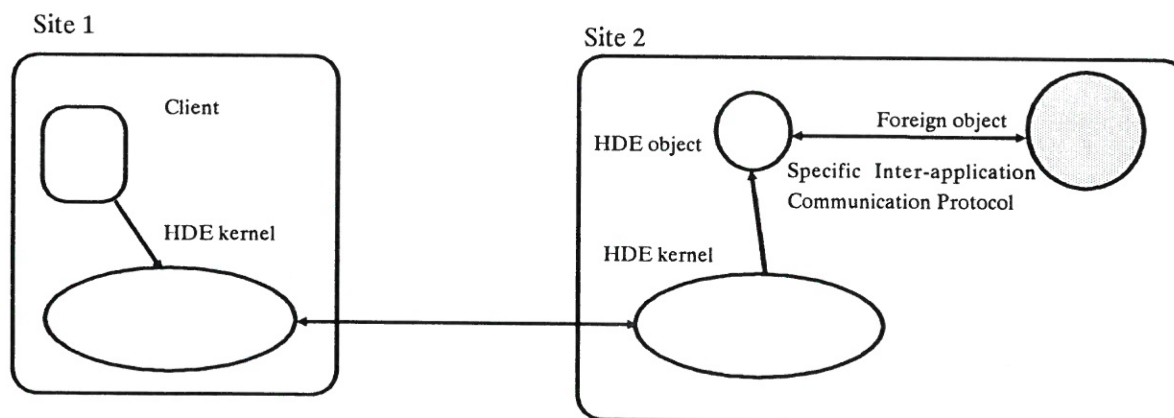

Figure 15. Foreign-object integration in an HDE - application

Table 5. Class and Object Functions

RegisterClass(ClassName, SuperClassList, ConstructorId, DestructorId, format)

– this function registers the specified class "ClassName". The "SuperClassList" parameter contains the list of superclasses from which the current class is derived. The "ConstructorId" and "DestructorId" represent the reference to the class constructor and destructor implementation, while "format" parameter describes the type of parameters that must be passed on to the constructor when an object is created.

CreateObj(ObjType, SiteName, ObjName, ClassName, format, parameters...)

– this function instantiates an object of type "ObjType" from the class "ClassName" on the site specified by "SiteName" parameter. The "parameters" and "format" represent the parameters passed to the class constructor and their format . If the "SiteName" is NULL, then the object is created on the local site. The name specified by "ObjName" is automatically associated with the created object. This function returns a local handle which identifies the object.

DestroyObj(ObjHandle)

– this function destroys the object referred by handle "ObjHandle".

RegisterName(ObjHandle,ObjName)

– this function associates a symbolic name with an object (identified by the handle "ObjHandle"). This name is known by all objects in the same application (HDE_SHARED type) or domain (HDE_GLOBAL type).

DeregisterName(ObjName)

– this function deregisters the name "ObjName".

LinkObj(ObjName)

– this function returns a local handle to the global object "ObjName". Using this handle, any methods of the object "ObjName" may be invoked. All objects from the same site obtain the same local handle for the same object.

UnlinkObj(ObjHandle)

– this function is used in conjunction with the previous one. When LinkObj function is called, a reference counter, associated with object, is automatically incremented. When UnlinkObj is called, the reference counter is decremented. An object can be destroyed only if its reference counter is one.

MigrateObj (ObjHandle,siteName)

– this function is used to migrate an object from resident site to a remote site specified by "siteName".

CopyObj(ObjHandle,siteName)

– this function has the same meaning as MigrateObj function except that the source object is not destroyed.

SaveObj(ObjName)

– this function saves the object state to the permanent storage by invoking the save method.

RestoreObj(ObjName)

– this function restores a previously saved object state from the permanent storage by invoking the restore method.

LocateObj(ObjHandle)
LocateObj(ObjName)

– this function returns the site identifier to where the object is located.

GetObjName(ObjHandle)

– this function returns the symbolic name from the object handle.

Table 6. Method Functions

RegisterMethod(MethodType,ClassName,MethodName,MethodId,format)

– this function registers a method of type "MethodType" for the class "ClassName". The "MethodName" is a symbolic name which identifies the method across network. The "MethodId" is the reference to the specified method, while "format" describes the type of passed parameters, and the type of returning result. When a class is registered, all methods of that class must also be registered.

DeregisterMethod(ClassName,MethodName)

– this function deregisters the method "MethodName" of class "ClassName".

QueryMethSel(ClassName,MethodName,format)

– this function is used by the client to obtain a handle to the desired method.

ExecMethod(ObjName,MethName,format,parameters...)
ExecMethod(ObjName,MethHandle,parameters...)
ExecMethod(ObjHandle,MethName,format,parameters...)
ExecMethod(ObjHandle,MethHandle,parameters...)

– this function invokes a method (specified by the handle "MethHandle" or by the name "ObjHandle") of a remote object (specified by handle "ObjHandle" or by name "ObjName"). The "parameters" contain the values needed invoke the method and receive the results. In plain C these functions have different names.

Table 7. Group Functions

CreateGroup(GroupName)

– this function creates a group with name "GroupName".

DestroyGroup(GroupName)

– this function destroys the group called "GroupName".

AddObjToGroup(GroupName,ObjHandle)
AddObjToGroup(GroupName,ObjName)

– this function adds an object identified by its handle "ObjHandle", or by its name "ObjName" to the group "GroupName".

RemoveObjFromGroup(GroupName,ObjHandle)
RemoveObjFromGroup(GroupName,ObjName)

– this function removes an object identified by its handle "ObjHandle", or by its name "ObjName" from the group "GroupName".

V. DC + +

V.1. Language Extension

We propose here an extension of C + + (DC + +) that integrates the services provided by HDE. The extensions are both syntactic (involving some type modifiers for classes, objects and methods) and semantic. It is out of the scope of the paper to describe the full syntax and semantics of DC + +.

– *virtual*

This keyword is also present in C + + standard language. In DC + + it can precede a class declaration, denoting that the class is implemented by another context. An object instantiated from a virtual class is a virtual object. All virtual class methods are simple invocations of the corresponding methods from the server class. In fact, each virtual class method implements a stub (like in RPC) which makes the method invocation mechanism transparent to the client. So, from the client's point of view the server class is represented by the virtual class that gives total transparency to method invocations. The semantics of a virtual object is modified (the state of the object cannot be accessed directly even if it is publicly defined).

The following five keywords (**shared, distributed, global, persistent and replicated**) are applied to the object declaration and can be viewed as type modifiers in the standard C language (e.g. register, auto, static).

– *shared*

This modifier denotes a HDE-object that can be accessed through method invocation by more than one client at the same time, or by the same client from different threads. A shared object must have a global symbolic name unique over the application. This is used like an object reference by any client that wants to access it. By default, this name is assumed to be the same as the object identifier. For example, suppose we want to instantiate a shared object from the Point class. For that we must simply write:

```
shared Point pt(10, 10);
```

If the object exists (being created before) then the HDE kernel creates, if necessary, a LOD on the local site and returns the handle (in fact the mechanism is similar to a **LinkObj** function invocation). From the application point of view, a shared object is similar to a global variable in a traditional language (but with concurrency control explicitly handled by the structure of passive or active objects).

– *distributed*

This modifier denotes a distributed HDE-object. In some respects, a distributed object is similar to a shared object. Like a shared object, it can be simultaneously accessed by more than one client. But, while a shared object is represented internally by a single copy, a distributed object has a copy on each client resident site. For example, to dynamically instantiate a distributed object , the following code may be written:

```
distributed Point *ppt;
...
ppt = new Point(10, 10, RED);
```

– *global*

The **global** keyword precedes a class declaration, by specifying that the class may be remotely used to instantiate a virtual object. It also specifies that the class must be registered. Once a class is registered, it can be accessed from any other client context (or from its own).

When this keyword precedes an object declaration, it has almost the same meaning as **shared** keyword. Unlike shared modifier that defines an object with a unique symbolic name inside the same application, the **global** modifier defines an object with a unique symbolic name over the entire domain. So, a global object may be shared by multiple applications concurrently. The global modifier applied to an object instantiation is composable with other modifiers (distributed, replicated, persistent).

– *persistent*

The **persistent** keyword is used to instantiate a persistent HDE object. Remember that a persistent object is an object that saves its state on the permanent storage whenever a method changing the object state is invoked. This is done automatically by the HDE kernel through object Save method invocation. In a similar way, a persistent object is created using a statement such as:

```
persistent Point pt(10, 10, RED);
```

– *replicated*

The **replicated** keyword is used to instantiate HDE

replicated objects. This modifier is especially important, because it gives the user the opportunity to specify the **replicated** object resilience degree. In this respect, the desired resilience degree of the replicated keyword is put in brackets. So, in order to create an object which must survive at two simultaneous crashes, the programmer must declare the object as follows:

```
replicated(2) Point pt(10, 10, RED);
```

By default, the replicated keyword without brackets sets the resilience degree at the value 1. For example, when a replicated object with resilience degree equal to 1 must be instantiated, the programmer must simply write:

```
replicated Point pt(10, 10, RED);
```

which is the same as:

```
replicated(1) Point pt(10, 10, RED);
```

Like in C language, the HDE-modifiers can be combined to form more complex modifiers. For example, on instantiating a persistent and replicated object, the programmer might write:

```
persistent replicated Point pt(10, 10, RED);
```

As an example, for this DC++ declaration, the following C++ code is generated (the constants used in the code are self-explanatory):

```
Point pt(HDE_REPLICATED | HDE_PERSISTENT,
NULL, "pt", "Point", format, 10, 10, RED);
```

Obviously, not all combinations are admitted (e.g. it has no meaning to combine **shared** and **distributed** modifiers in the same declaration). The next table gives the list of all valid combinations of object instantiation modifiers.

	shared	global	distributed	persistent	replicable
shared	–	–	–	X	X
global	–	–	X	X	X
distributed	–	X	–	X	–
persistent	X	X	X	–	X
replicable	X	X	–	X	–

– at

This keyword is used on the instance creation for specifying the desired site where the object must be created. For example, to instantiate a **shared** object on "site_1" we must write:

```
shared Point pt(10, 10, RED) at "site_1";
```

– read, write, save, restore

These keywords are used to define a public method of type READ, WRITE, SAVE or RESTORE. When the method is registered, the proper type is specified. By default (if none of these keywords is specified), the method type is assumed to be WRITE, so that by default a distributed object ensures the copies consistency.

– process

This keyword defines a special method representing a basic process (thread of execution) that is executed in background (relative to other method execution). This method has no name or parameters (and so, its operation cannot be invoked as a normal method is) and starts after the object constructor class ends. An HDE object can have at most one process method.

– parallel

This keyword allows the execution of its method in parallel with the other method invocations. Also, there may be more than one invocation of the same method in progress (running in parallel). The methods without parallel or process attributes are executed sequentially in the same thread. If a class has at least one parallel or process type method, any object instantiated from this class is said to be active. On the contrary, a passive object is an object instantiated from a class with only sequential methods. The differences between active and passive objects are from both the execution structure (see Figure 19) and the external behaviour points of view. A passive object executes all its methods sequentially, in the same thread of the context, which is allocated to all its passive objects.

An active thread is allowed to have a collection of methods sequentially executed (but not with respect to other objects) and the rest of them executed in parallel. So, the problems of synchronization and concurrent access to the object state may be efficiently and flexibly handled by the class implementor if choosing some methods that critically modify the state to be sequentially executed.

An active object has one thread allocated to its process method (which is started after the constructor execution), one thread for the group of sequentially executed methods and a pool of threads for the methods executed in parallel.

So, an HDE context may have one or more active and passive class instances (objects).A trade-off must be reached to obtain a level of granularity for some context complexity. Figure 16 shows the HDE kernel/context interaction thread, which is the one that schedules the methods calls (upcalls) in the HDE kernel.

To summarize, an HDE context may eventually have the following threads executing in parallel:

- one thread for all the passive objects;
- one thread for each active object, each one executing the sequential methods of that object;
- one or more threads for each parallel method;
- one thread for each active object, each one executing the process method;
- one general scheduling thread.

The programmer can solve the synchronization (including the critical section) problems by explicitly using the synchronization mechanisms provided by HDE. But, without them, an HDE context may implicitly ensure its consistency if the following rules are obeyed:

- all context global non-object variables (if somebody insists on using them) must be modified only via passive objects;
- all active object state variables must be modified only by the sequential methods of the object;
- instead of the main C++ function, a special HDE main function is used (the context may start its parallel threads only after HDE main returns);

The active object internal parallelism may be used only if the object is a shared or global one. So, HDE active objects allow an easy and flexible implementation of a communication paradigm called distributed data structure(DDS) [Kaashoek 89a]. But an HDE active object is more than a distributed data structure (in its original acceptance) because beyond its internal parallelism (hidden or not to the user), it allows the fragmentation of some functions (HDE acceptance of a fragmented object via its class hierarchy installed on different computers) and replication.

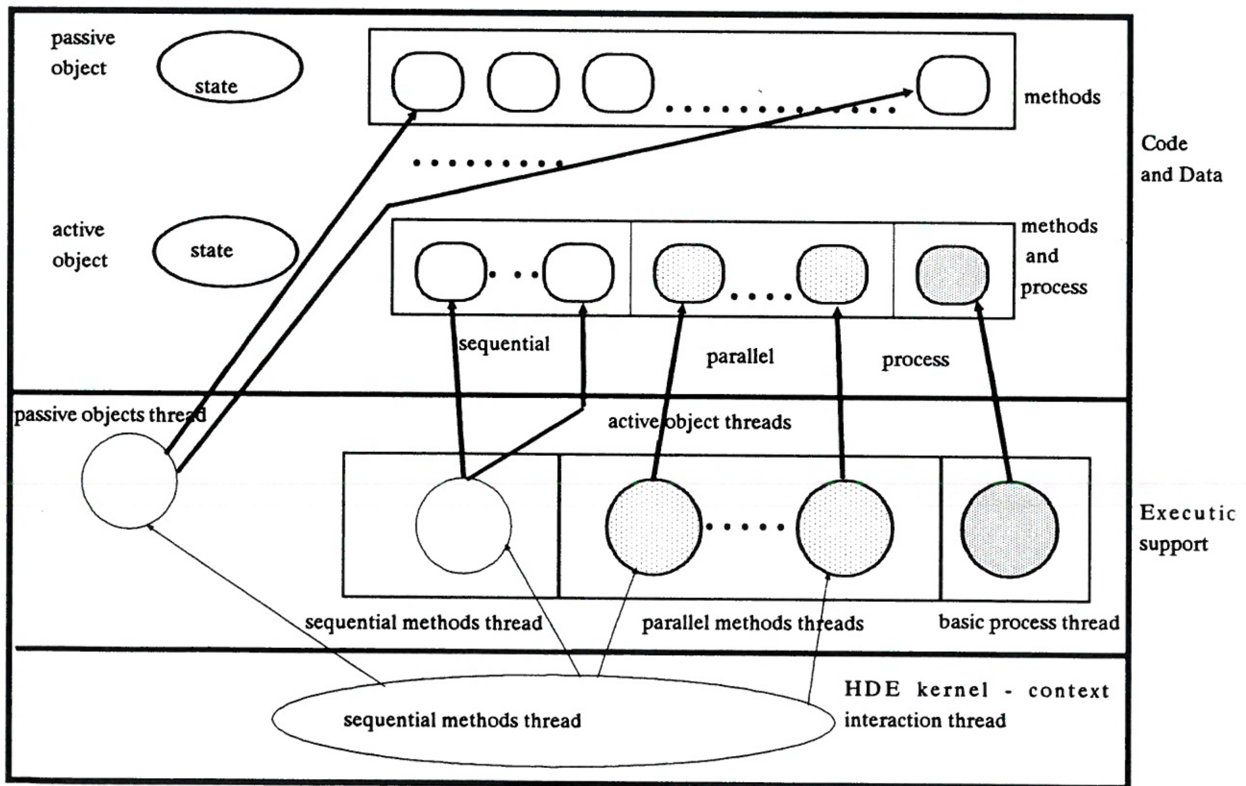

Figure 16. HDE Context Structure

V. 2 A Simple Example

We give here a simple example illustrating DC+ + facilities for distributed programming. First, the C+ + program is presented (Figure 17.a) and then a distributed alternative written in DC+ + is proposed (Figure 17.b, c). The C+ + program defines and implements two classes ("Pixel" and "Point" classes), instantiates an object (called "pt") and invokes an object method. In the DC+ + distributed version of that program both classes are implemented by one module (Figure 17.b), while the object instantiation and method invocation take place in another module (Figure 17.c). As the following section explains, this code may be converted to the standard C+ + code, using the DC+ + translator.

```
// Pixel class definition
class Pixel {
  int x;  // horizontal position
  int y;  // vertical position
public:
  Pixel(int xPos, int yPos)    // class constructor
          {x = xPos; y = yPos;};
  int GetX(void) {return x;}; // get current horizontal position
  int GetY(void) {return y;}; // get current vertical position
  void SetPos(int xPos, int yPos) // set new point position
          {x = xPos; y = yPos;};
};

// Point class derived from Pixel
class Point : public Pixel {
  int colour;
public:
  Point(int, int, int);                  // class constructor
  int GetColour(void) {return colour;};// get current point colour
  void SetColour(int col)               // set point new colour
    {colour = col;};
};

Point::Point(int xPos, int yPos, int col) : Pixel(xPos, yPos)
{
  colour = col;
}

main()
{
  Point pt(10, 10, RED) ; // the "pt" object is created
  pt.SetPos(10, 12);
}
```

Figure 17.a. C+ + Program code

```
// Pixel class definition
global classPixel{
  int x;
  int y;
public:
  Pixel(int, int);
  read int GetX(void) {return x;};
  read int GetY(void) {return y;};
  void SetPos(int xPos, int yPos)
     {x = xPos; y = yPos;};
};

// Point class derived from Pixel
global class Point: public Pixel {
  int colour;
public:
  Point(int, int, int);
  read int GetColour(void) {return colour;};
  void SetColour(int col) {colour = col;};
};

Pixel::Pixel(int xPos, int yPos)
{
  x = xPos, y = yPos;
}

Point::Point(int xPos, int yPos, int col) :
        Pixel(xPos, yPos)
{
  colour = col;
}

hde_main()
{
  // initialization code
       ...
}
```

Figure 17.b. DC+ + Server code

```
virtual class Pixel
{
public:
   Pixel(int, int);          // class constructor
   int GetX(void);           // get current horizontal position
   int GetY(void);           // get current vertical position
   void SetPos(int, int); // set new point position
};

virtual class Point:public Pixel
{
public:
   Point(int, int, int);     // class constructor
   int GetColour(void);  // get current point colour
   void SetColour(int);   // set point new colour
};

main()
{
   Point pt(10, 10, RED); // the "pt" object is created
   pt.SetPos(10, 12);
};
```

Figure 17.c. DC + + Client code

In the server module code the "Pixel" and "Point" class definitions are preceded by the **global** keyword, that is, both classes should be registered (to be accessible from other contexts). In the client module (Figure 17.c), both classes are preceded by the **virtual** keyword that is, the class is implemented by another module and is called a virtual class (not to be taken for C + + virtual modifier).

V. 3 DC + + /C + + Code Mapping

In this section we present the mapping of DC + + language constructions to plain C + +, detailing the previous example.

Class Registration

A class declaration preceded by the **global** keyword is automatically registered when the class server module is run. Here is the C + + code which performs this task.

First, the call functions of the constructor methods of each class are defined:

```
// function to call the pixel class constructor
void *C_Pixel(int x, int y)
{
   return (void *)new Pixel(x, y);
}
```

```
// function to call the point class constructor
void *C_Point(int x, int y, int col)
{
   return (void *)new Point(x, y, col);
}
```

This level of method encapsulation is needed because some C + + implementations [Stroustrup 87] have no chance to obtain the address of the methods (for their registration).

Now, the "Pixel" class can be registered in the main part of the server context:

```
// register class Pixel
RegisterClass("Pixel", NULL, C_Pixel, format, NULL);
```

Here, "Pixel" is the symbolic class name, C_Pixel represents the reference to the constructor and "format" is a string describing the passed parameters format on the constructor invocation. In this case, the format must specify that two integers are passed on as parameters and a pointer to the instantiated object is obtained. The second parameter is a string containing a list of all superclass symbolic names. Here, this parameter is NULL because there is no ancestor (base class) for Pixel class. The last parameter is a reference to the class destructor (in our case it is NULL because it is not implemented). When the registration is performed, the HDE kernel creates the class descriptor (CD) automatically. The Point class is registered in the same way:

```
// register class Point
RegisterClass("Point", "Pixel", C_Point, format, NULL);
```

Here, the second parameter contains the symbolic name of the "Point" ancestor class. If there are more ancestors than one, their symbolic names are delimited by blanks. After having executed this function, CD is created and linked to the Pixel CD.

Usually, C + + has more than one constructor for the same class, using the overload mechanism. This is possible because C + + takes care of calling the correct method (i.e. constructor) for the given argument. HDE also provides a similar mechanism. For example, if you want to define another constructor method for the Point class, you may use the same class register function:

```
// register class Point
RegisterClass("Point", "Pixel", C_Point1, format1, NULL);
```

One must therefore define another function (C_Point1) for calling the Point constructor

method with arguments specified by "format1" parameter. When a Point object is instantiated, HDE compares the arguments passed and calls the appropriate constructor.

Method Registration

When a class is registered, all class methods that a client is allowed to invoke (which are in the public section of the class declaration) must also be registered. To register a method, the same approach as registering a class must be taken. First, a function invoking the desired method is defined (the explanation is the same as for constructor/destructor methods). For example, to register SetColour and GetColour methods (of Point class), the following code may result:

```
// function to SetColour method
void Point_SetColour(Point *ppt, int col)
{
  ppt->SetColour(col);
}
// function to GetColour method
int Point_GetColour(Point *ppt)
{
  return ppt->GetColour();
}
```

Second, the class method can be registered:

```
// register SetColour method
RegisterMethod(WRITE, "Point", "SetColour",
Point_SetColour, format);
// register GetColour method
RegisterMethod(READ, "Point", "GetColour",
Point_GetColour, format);
```

Note that the "format" parameter has the same meaning as in the class registration function.

Virtual Classes

A virtual class describes the appropriate global class (possibly implemented by another context) interface, accessible to the client. For each virtual class, the preprocessor modifies the definition and generates a specific implementation as follows:

```
// Pixel class definition. This class is implemented under
another context
class Pixel : HDEClass // derived from HDEClass
{
public:
  void Pixel(char fCreate, long objType, char *objName,
  char *siteName, int x, int y);
  void ~ Pixel(void);
  int GetX(void);
  int GetY(void);
  void SetPos(int, int);
// Point class definition
virtual class Point: Pixel
{
public:
  void Point(char fCreate, long objType, char *objName,
  char *siteName, int x, int y, int colour);
  void ~Point(void);
  int GetColour(void);
  void SetColour(int);
}
```

One can notice that the first class constructor parameter is a flag called **fCreate**. This flag is true for the first virtual class level in the class hierarchy and false for the others. This is required for overriding the C++ standard constructor method calling strategy. The base class constructor method is always called before the derived class constructor. For example, when an object from class Point is instantiated, the class Pixel constructor is called first and the constructor of class Point is called afterwards.

On the other hand, the first idea that might come up is that when a virtual class constructor is called, then the corresponding real class constructor is to be invoked. So, when the Point local constructor is called, HDE automatically invokes the corresponding constructor via C_Point function. When C_Point invokes the real Point constructor, the real Pixel constructor is also invoked. In this way, a virtual Point object is created. As a conclusion, when a local Point object is instantiated, then the local Pixel and Point constructors are invoked, and consequently, virtual Pixel and Point objects are created via HDE messages. But this situation is unacceptable because two real objects are created for a local Point object: a Point object and a Pixel object. The solution adopted is to filter the remotely created object invocation. Thus, when a local Point object is instantiated, only the constructor of the global Point class must be invoked, while the global class Pixel constructor invocation must be ignored. Practically, fCreate selects all virtual classes which

the real class constructor must be invoked for. For example, when a Point class instance is created then the fCreate parameter is true for the "Point" class constructor and false for the "Pixel" class constructor. Each invoked class constructor verifies the fCreate parameter and, if true, then the virtual class constructor calls CreateObj function that invokes the global class constructor. So, the preprocessor generates the following C++ code for Point class constructor:

```
// local Point class constructor
Point::Point(char fCreate, long objType, char *objName,
char *siteName, int x, int y, int colour) :
        Pixel(FALSE, 0L, NULL, NULL, 0, 0)
{
  if (fCreate)
        // instantiate an object, and get the handle to local
          object descriptor
        iHlod = CreateObj(objType, siteName, objName,
         "Point", format, x, y, colour);
}
```

A similar problem as for class constructor method must be solved for destructor method. So, when a virtual class destructor method is invoked (using the C++ delete operator), also the real object (at the server) must be destroyed. Because C++ uses a different approach to invoke the destructor method (first the object's base class destructor is invoked and later, the ancestor class destructor), the preprocessor generates the following code:

```
// local Point class destructor
Point::~ Point(void)
{
  if (iHlod) {
        // destroy the real object
        DestroyObj(iHlod);
        iHlod = NULL;
  }
}
```

This code ensures that DestroyObj is called only once when an HDE-object is destroyed.

All virtual classes must be derived from a base class called HDEClass, which describes the communication protocols and contains some useful instance variables. One of them is **iHlod**, which contains a handle to the local object descriptor returned when the real object is created. Also, the HDEClass contains another instance variable named iErr, the last error code resulted from a method invocation. This is a public (in the current context) variable, so the programmer should test it by using the following code:

```
if (pt.iErr != HDE_OK)
    // an error occurred
    ... handle the error
```

Object Creation

When an object is instantiated a virtual object is automatically created (from the virtual class) on the local site. The virtual object is, somehow, like a null proxy object (it has no actual state). From the user's point of view, there are no differences between a virtual class and any other common C++ class. For example, in order to instantiate a Point object, the programmer must simply write:

```
Point pt(10, 10, RED); // Instantiate an object from class Point
```

The preprocessor expands this statement by adding the proper parameters needed for calling the **CreateObj** functions as follows:

```
Point pt(0L, NULL, NULL, "Point", format, 10, 10, RED);
```

On the object creation we may have the following situation:

- The client and server modules are on the same site and the class hierarchy is implemented in the same context. In this case, the local object handle (iHlod) points to ROD as shown in Figure 18.a.
- The client and server modules are resident on different sites and the class hierarchy is implemented in the same context. That means that there are no ancestor virtual classes (for the Point class) inside server. In this situation iHlod contains the reference to LOD (Figure 18.b).
- The client and server modules are resident on different sites and the Point class hierarchy is implemented by different servers (Figure 18.c). In our example, that means that Pixel is a virtual class implemented by another server. So, the server which implements Point class becomes a client for the server that implements Pixel class. In this case, such an object has been called fragmented object (this term differs from the one used in SOS [Shapiro 89] project) because the methods and instance variables are implemented by different server modules.

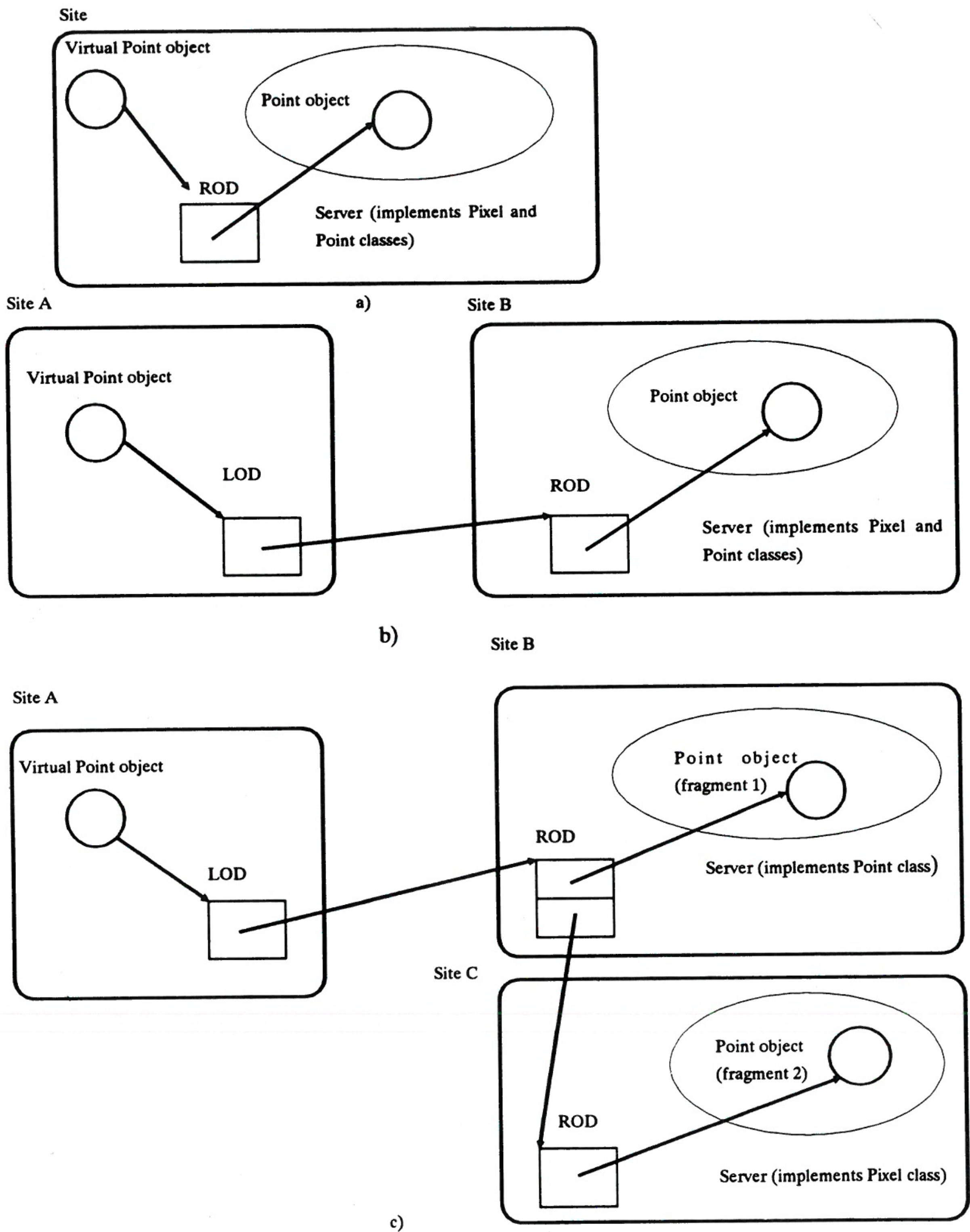

Figure 18.Point object possible implementations

Method Invocation

Let's consider the simple method invocation:

```
col = pt.GetColour();
```

In order to yield the desired behaviour, the preprocessor expands the clients virtual class methods so that the actual class methods should be invoked. In this case the following code will be used:

```
// local class GetColour method implementation
int Point::GetColour(void)
{
   int colour;
   ExecMethod(iHlod, "GetColour", format, &colour);
   return colour;
}
// local class SetColour method implementation
void Point::SetColour(int colour)
{
   ExecMethod(iHlod, "SetColour", format, colour);
}
```

We will further use a symbolic name for denoting the desired method. Some other possibility would be to query a method selector when the virtual class is instantiated. Thus, the method selector could be used for replacing the symbolic name of the method. Following this approach, the code generated by precompiler is rather complex, but the operation is more efficient. We will exemplify with local Point class.

```
// Point local class after expandation
class Point : Pixel
{
   METHSEL sGetColour, sSetColour;
public:
   ..... // method prototypes
};
```

In order to get the method selectors, the constructor method will be modified.

```
// new constructor implementation
Point::Point(char fCreate, long objType, char *objName,
char *siteName, int x, int y, int colour):
      Pixel(FALSE, 0L, NULL, NULL, 0, 0) {
   if (fCreate)
      iHlod = CreateObj(objType, siteName, objName,
      "Point", format, x, y);
   // query GetColour method selector
   sGetColour = QueryMethSel("Point", "GetColour", format);
   // query SetColour method selector
   sSetColour = QueryMethSel("Point", "SetColour", format);
}
```

Obviously, the implementation of the virtual class methods will be correlated to the use of method selector. For example, the SetColour method will be generated as follows:

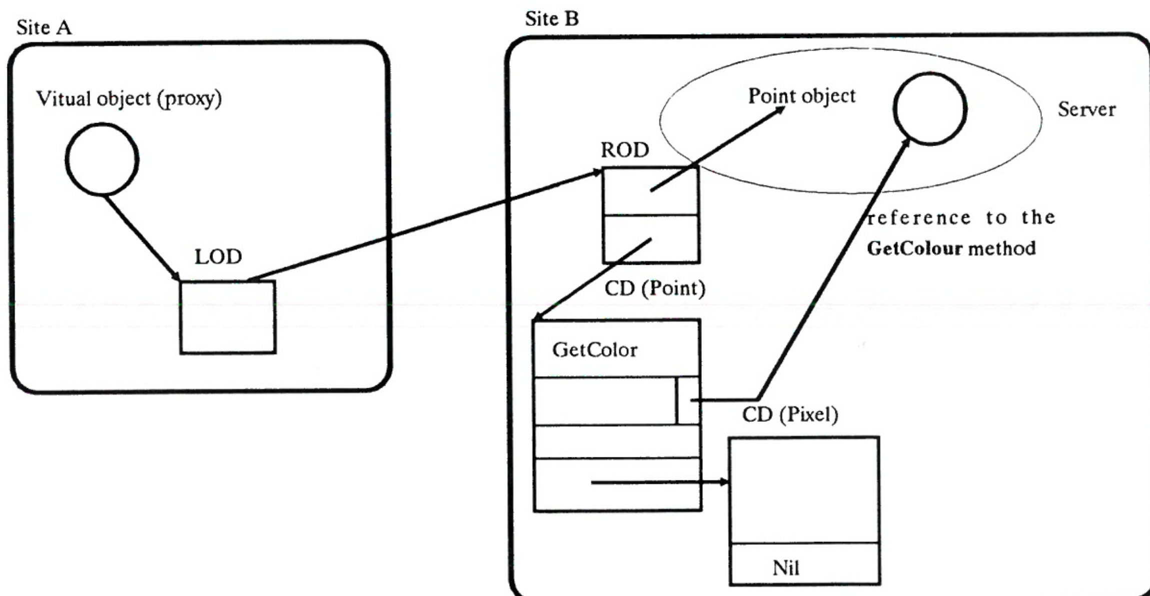

Figure 19. HDE-object method invocation

```
// local class SetColour method implementation
void Point::SetColour(int colour)
{
  ExecMethod(iHlod, sSetColour, colour);
}
```

Figure 19 presents a method invocation path from client to server when all virtual classes are implemented in the same server. Provided that some classes in the server are virtual (they are implemented by another server), the messages will be delivered hierarchically to the server actually implementing the required class.

VI. LINDA Mechanisms Implementation in HDE

LINDA [Carriero 89] is a parallel programming language that supports both the distributed data structure paradigm and the replicated worker programming style [Tanenbaum 89]. A distributed data structure is a data structure that can be concurrently handled by one or more processes. From the process point of view, all the operations that are allowed to be executed on a distributed data structure are atomic (indivisible). The replicated worker programming style assumes that a worker which usually is a process that has to do some work on a set of data is replicated. By sharing a distributed data structure, multiple workers can do their jobs in parallel, thus increasing the application execution productivity. In LINDA, the distributed data structure is implemented like a global content addressable shared memory called Tuple Space (TS). Elements contained in TS are called tuples. On TS five atomic operations are defined:

- out - put a tuple in TS;
- read - read a tuple from TS. This is a synchronous operation, and so, if the tuple is not in TS, it waits until some worker inputs a tuple (using out operation) that matches the required tuple;
- in - read and remove a tuple from TS. This is also a synchronous operation.
- rdp - read a tuple from TS without waiting (asynchronous). If the required tuple is not in TS, this operation returns false (otherwise true).
- inp - read and remove a tuple from TS. As in the previous operation, if the tuple is not in TS, this operation returns a false value, without waiting.

This section briefly presents how LINDA mechanisms can be implemented in HDE using DC++. TS is implemented as an HDE active distributed object that supports as methods all the five operations (Figure 20.a). Both rdp and read methods are preceded by the read modifier (in HDE a method is assumed to implicitly have the write modifier) because they do not modify the object. So, these methods are performed efficiently on the local object copy. Both synchronous methods (in and read) are implemented as parallel methods (in different threads) in order to avoid other method invocation. They call repeatedly the corresponding asynchronous methods (inp and rdp), until the required tuple is found. This is a simple but inefficient implementation, because the processor time is wasted in inp and rdp methods invocation. The other solution, more performant, but not so simple, is to use the HDE explicit synchronization primitives.

In LINDA, a typical worker takes some work from the TS, performs it and invests the results back in the TS. This takes place repeatedly in a loop, until all work is finished. Figure 20.b is an example of such a worker that is encapsulated in an HDE active object. In the constructor method, either a distributed TS object named "ts" is created (if not existing) or a reference to an existing object is obtained (if already created). In the process method, the worker asks for some work to do by invoking TS in method, performs it and invests the results back in the TS by invoking out method. Also, in the main loop, it increments an internal counter (by invoking IncTuples() method) that keeps the total number of jobs done.

The work to be performed enters TS by a process called master (Figure 20.c). After that, the master waits until the work is completed and gets the results one by one. Also, the master instantiates all workers in the domain.

In this example the workers do not generate some further work or create other workers, although HDE supports all the needed mechanisms for this.

The application structure, built up from objects in the master context, worker contexts and TS context, is presented in Figure 21. One may notice that in the master context and in each worker context, there are virtual objects for the distributed TS unique object. Also, on each worker and master site, a copy of TS context is migrated. Although on one site there are more than one worker, only one TS copy will exist.

```
// Tuple class definition
global class TupleSpace {
   ... instance variables ...
public:
   BOOL inp(...);
   read BOOL rdp(...);
   void out(...);
   parallel void in(...);
   parallel read void read(...);
};

BOOL TupleSpace::inp(...)
{
    // asynchronously read & remove a tuple from TS
   ...
}

BOOL TupleSpace::rdp(...)
{
   // asynchronously read a tuple from TS
   ...
}

void TupleSpace::out(...)
{
   // put a tuple in TS
   ...
}

void TupleSpace::in()
{
   while (!inp(...));
   // read & remove a tuple from TS
}

void TupleSpace::read()
{
   while (!rdp(...));
   // read a tuple from TS
   ...
}

hde_main()
{
    ; // no module initialization code here
}
```

Figure 20.a. Tuple Space Implementation

```
// Tuple class definition
virtual class TupleSpace {
public:
TupleSpace(...);
   ~TupleSpace(...);
void rdp(...);
   void inp(...);
   void in(...);
   void out(...);
   void read(...);
};
// Worker class definition
global class Worker {
   int processTuples;
   distributed TupleSpace &ts;
public:
   Worker(void) {processTuples = 0;};
   void IncTuples {processTuples++;};
   read int GetStatus(void) {return processTuples;};
   process MainWork(void);
};
// Worker class constructor method
Worker::Worker(void)
{
   ts = new TupleSpace(...);
   processTuples = 0;
}
// Worker class destructor method
Worker::~Worker(void)
{
   delete(ts);
}

// Worker class process code
Worker::MainWork(void)
{
   while (TRUE) {
           ts.in("to_do", ...);
           ... process touple .....
           ts.out("done", .....);
           IncTuples();
   }
}

hde_main()
{
   ; // no module initialization code here
}
```

Figure 20.b. Worker Class Implementation

```
// Tuple class definition
virtual classTupleSpace{
public:
   TupleSpace(...);
   ~TupleSpace(...);
   void rdp(...);
   void inp(...);
   void in(...);
   void out(...);
   void read(...);
};

// Worker class definition
virtual class Worker {
public:
   Worker(void);
   ~Worker(void);
   int GetStatus(void);
};

main()
{

   distributed TupleSpace ts; // Creates the tuple space object using the same name "ts" as in worker objects.
                              // This provides the same distributed object from TupleSpace class to be used
                              // in the whole application.
   Worker wk1 at "Site_1";    // creates some workers on different sites
   Worker wk2 at "Site_2";
   ...
   Worker wkn at "Site_N";
   for (each work to be done)
          ts.out("to_do", ...);
   // get result loop
   while(exist touples in TS) {
          ts.in("done", ...);
    }
   // all workers created by this context are automatically destroyed by the kernel when the program ends
}
```

Figure 20.c. Master Module Code

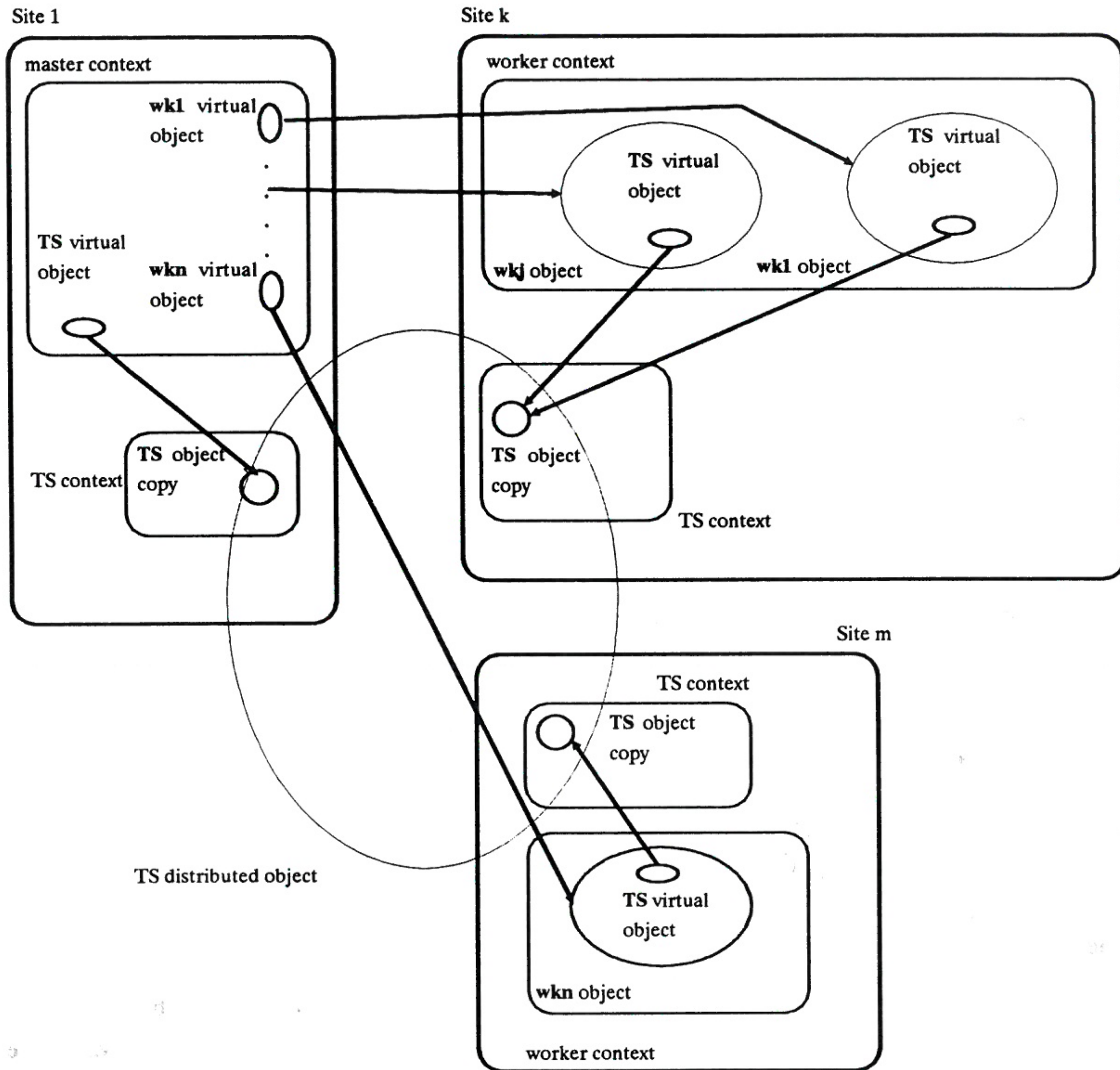

Figure 21. HDE structure of a Linda like application

VII. Concluding Remarks

The paper has described the architecture and some basic (operating system independent) strategies and mechanisms of the HDE (a Heterogeneous Distributed Object-Oriented Environment), a system which tries to merge uniformly the " object-oriented" and "distributed" paradigms and which strives to be an environment for designing, implementing and executing O-O distributed applications.

The system has language independent mechanisms for supporting distribution like attributes of objects: mobility, fragmentation, replication (both for fault-tolerance data availability and user residence copy existence) and persistence. However, HDE demonstrates a high degree of simplicity and transparency by allowing the use of all these features in a C++ extension (DC++).

One of the aspects which differentiates various DO-O OS and which makes them apart from the classical object model is the exploitation of and the access to parallelism. A derived problem is that of active objects. HDE allows the handling of parallelism at variable degree of granularity (from the object methods up). The parallelism of a class hierarchy implementation is explicit (but with high level language constructs, i.e. DC++), while at the user level is transparent.

For supporting these features, HDE relies on a private protocol architecture oriented towards the group communication. The protocols present different degrees of reliability. Its services are implicitly used by HDE application via the HDE object handling (fragmented, distributed and replicated objects), but may also be explicitly used.

The future work of the HDE group will be the completion of HDE implementation on a heterogeneous platform (now limited to UNIX/AIX and OS/2). Also, the CASE-HDE application for designing, implementing and testing DC++ user applications will be one major goal.

REFERENCES

ALMES, G., BLACK A.P. and LAZOWSKA E.D., **The Eden System: A Technical Review,** IEEE TRANS. ON SOFTWARE ENGINEERING, January 1985.

BAL, H.E., KAASHOEK, M.F., and TANENBAUM, A.S., **A Distributed Implementation of the Shared Data-Object Model,** Proceedings of the Workshop on Distributed and Multiprocessor Systems, Fort Lauderdale, FL, October 1989.

BALTER, P., BERNADT, J., DECONCHANT, D. and KRAKOWAK S., **Modèle d' Execution du Systeme Guide, Rapport Technique no. R-3,** IMAG, decembre 1987.

BLACK, A., HUTCHINSON, N., MCCORD, B. and RAJ, R., **EPL, Programmer's Guide,** Eden Project, Dept. of Computing Science, Univ. of Washington, Seattle, Washington, January 1984.

BLACK A., HUTCHINSON, N., JUL, E., LEVY, H. and CARTER, L., **Distribution and Abstract Types in Emerald,** IEEE TRANS. ON SOFTWARE ENGINEERING, January 1987.

CARRIERO, N. and GELENTER, D., **Linda in Context,** COMMUNICATIONS OF THE ACM, April 1989.

HUTCHINSON, N., **Emerald: An Object-Based Language for Distributed Programming,** PhD Thesis, 1987.

A.K Jones and E.F. Gehringers (Eds.) **The Cm* Multiprocessor Project: A Research Review,** Dept. of Computer Science, Carnegie-Mellon University, July 1980.

KAASHOEK, M.F., TANENBAUM, A.S., HUMMEL, S. and BAL, H., **An Efficient Reliable Broadcast Protocol,** ACM OPERATING SYSTEM REVIEW, October 1989.

KAASHOEK, M.F., BAL, H. and TANENBAUM, A.S., **Experience with the Distributed Data Structure Paradigm in Linda,** Proceedings of the Workshop on Distributed and Multiprocessor Systems, Fort Lauderdale, FL, October 1989.

LEBLANC, R.J. and APPELBE, W.E., **The Clouds Distributed Operating System,** Proceedings of the 8th Int. Conf. on Distributed Computing Systems, San Jose, CA., June 1988.

LISKOV, B.H., ATKINSON, R., BLOOM, T., MOSS, E., SHAFFERT, C., SCHEIFLER ,B. and SNYDER, A., **CLU Reference Manual,** Technical Report LCS/TR-225, MIT, October 1979.

LISKOV, B.H., **On Linguistic Support for Distributed Programs,** IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, May 1982.

LISKOV, B.H., **The Argus Language and System, in Distributed Systems: Methods and Tools for Specifications,** LNCS 190, SPRINGER-VERLAG 1985.

SHAPIRO, M., **Prototyping a Distributed Object-Oriented OS on UNIX,** SOR-60, Project SOR, INRIA, Rocquencourt, France, May 1989.

SHAPIRO, M. and GOURHANT, Y., **FOG/C++: A Fragmented-Object Generator,** Usenix C++ Conference, San Francisco, CA., April 1990.

STROUSTRUP, B., **The C++ Programming Language,** ADDISON-WESLEY PUBLISHING COMPANY, July 1987.

TANENBAUM, A.S., **Computer Networks,** Prentice Hall, Englewood Cliffs, NJ., 1988.

WULF, W. A., LEVIN, R. and PIERSON, C., **Overview of Hydra Operating System Development,** Proceedings of the 5 th ACM Symposium on Operating System Principles ACM/SIGOPS, Austin, TEXAS, November 1975.