Javanaise: distributed shared objects for Internet cooperative applications

D. Hagimont[1], *D. Louvegnies*[2]

SIRAC Project
INRIA, 655 av. de l'Europe, 38330 Montbonnot Saint-Martin, France

Abstract: We have implemented a service for the development of distributed cooperative applications on the Internet. The service consists of a set of system classes and a proxy generator implemented in the Java environment. The service and the applications that use it are dynamically deployed to client nodes at run-time, thanks to Java mobile code. The objects managed by the applications are transparently shared between the client nodes, so that the application developer can program as in a centralized setting.

Our system support relies on object replication on the client nodes. Logically related objects are grouped in clusters, the cluster being the unit of sharing, replication and coherence. One of the main advantages of our proposal is that the object clustering policy is tightly coupled with the application code, thus ensuring locality, while keeping clustering transparent to the application programmer.

This service has been prototyped and validated with simple distributed applications.

1. Introduction

Support for cooperative distributed applications is an important direction of computer systems research, involving developments in operating systems as well as in programming languages. In the 8Os, a model that emerged for the support of cooperative distributed applications is that of a distributed shared universe organized as a set of objects. Distributed object-oriented systems such as Guide [Hagimont94], Emerald [Black87] or Clouds [Dasgupta90] belong to this class of systems. More recently, the growth of the Internet, which is now daily used as a cooperation support, logically leads to consider the deployment of distributed cooperative applications over the Internet.

Today, distributing applications on the Internet is closely linked with the Web (essentially URLs) and Java. Therefore, a first attempt to provide distributed shared objects on the Internet was Java-RMI [Wollrath96] which provides remote method invocation between Java objects. Shared objects are uniquely named with URLs and a mechanism called *object serialization* allows distributed programs to exchange copies of objects (as in Sun RPC [rpcgen88]). However, using the RMI facilities, distributed applications are based on the client-server architecture which does not allow objects to be cached and therefore accessed locally. It is possible to manage object replicas using the object serialization facility, but the coherence between the replicas has to be explicitly managed by the application programmer. We believe that object caching is one of the key features required by cooperative applications, especially over the Internet, whose latency and bandwidth are highly variable.

---

[1] INRIA (Institut National de Recherche en Informatique et Automatique

[2] Université Joseph Fourier, Grenoble

In order to assist the programmer, we propose a new system service which implements the abstraction of a distributed shared Java object space. Objects are brought on demand on the requesting nodes and are cached until invalidated by the coherence protocol. With this system support, the programmer can develop its application as if it were to be executed in a centralized configuration. Then, the application can be configured for an distributed setting without any modification to the application source code. This configuration is performed by annotating the interfaces of the objects that are distributed, specifying the synchronization and consistency protocols to apply to these objects. A prototype of this service has been implemented on top of Java and consists in a proxy-generator which is used to generate indirection objects (proxies) for the support of dynamic binding, and a few system classes that implement consistency protocols and synchronization functions.

The main advantages of our approach are:

- Dynamic deployment. Applications are dynamically deployed to the client nodes from the node that hosts the application; thus we don't require applications to be statically installed prior to execution.
- Transparency. A distributed cooperative application can be developed as if it were to be run centralized. Distribution and synchronization are programmed separately from the application code.
- Caching. Our system support allows shared Java objects to be cached on cooperating nodes, thus enabling local invocation on distributed objects and reducing latency.
- Clustering. Grouping objects in clusters is one of the key techniques for achieving good performance by factorizing system costs. We propose to base clustering on the data structures defined in the application. We claim that applications already manage object groups in their internal structure and that clustering should be mapped on this application grouping.

The rest of the paper is structured as follows. In section 2, we provide an overview of the Java environment which introduces the Java features used in the rest of the paper. We present in section 3 the general motivations for the Javanaise environment. Section 4 presents the overall design choices for this system support. Section 5 describes the implementation principles and the prototype on top of Java is described in section 6. After a discussion of related work in section 7, we conclude in section 8.

2. The Java environment

In this section, we recall the aspects of the Java environment [Arnold96, Sun96] that are relevant to our experiment.

Java is a C++ like object-oriented programming language which is used to generate programs that can be executed on a portable runtime environment that runs on almost every machine type. Since Java is very popular and well known, we will only describe the features used in our experiment.

Code mobility

A key feature of Java is code mobility. Java allows classes to be dynamically loaded from remote nodes. Such mobility of code requires code portability and security enforcement in order to confine error propagation. Code portability is provided by interpretation of byte code. The *javac* compiler does not generate machine code, but a code which is common to (and independent of) any type of hardware and which is interpreted by the runtime of the language during execution. Security is mainly enforced by the safety of the Java language. The Java language does not allow direct access to the address space of the program. Objects are not manipulated through pointers, but through language level references that cannot be forged. A program can only obtain a Java reference in return of an object creation or as a parameter of an object invocation.

Java code mobility is now widely used for the Web. Most of the Web browsers includes a Java virtual machine and an HTML page can include some references to Java programs called Applets. When the HTML

page is downloaded by a browser, the Java program is executed by the Java machine embedded in the Web browser.

Polymorphism and dynamic binding

Another important aspect of Java that we used is polymorphism. Polymorphism refers to the ability to define *Interfaces* (or types) and classes separately. An interface is a definition of the signatures of the methods of a class, which is independent from any implementation. An interface can therefore be implemented by several classes, and it is possible to declare a variable whose type is an interface, and which can reference objects from different classes that implement the same interface.

Java also implements dynamic binding, which is crucial to mobile code. Dynamic binding means the ability to determine at run-time the code to be executed for a method invocation. Since Java allows classes to be dynamically loaded, a variable of an interface type can be assigned to a reference that points to an object, whose class was loaded dynamically. Java postpones the binding of the code (of this variable) until invocation time, thus allowing dynamically loaded classes to be executed.

Object serialization

Java also provides in its last release an object serialization feature [Riggs96] which allows instances to be exchanged between different runtimes. This feature provides a means for translating a graph of objects into a stream of bytes which can be sent as a message over the network or written into a file on disk. The receiver of the message or the reader of the file can deserialize the byte stream, i.e. rebuild the graph of objects. The Java references within the graph are changed, but the structure of the graph is preserved.

Each instance of a class which implements the *Serializable* interface can be serialized. Two inherited methods *writeObject* and *readObject* define, respectively, the default behavior for serializing and deserializing an object. By default, all the objects that are referenced from a serialized object are serialized (they must implement the *Serializable* interface). In order to control the serialization process, it is possible to override these methods by specifying which fields of the object should be transferred and reassigned when the object is rebuilt. This makes it possible, for instance, to stop the serialization recursion.

Garbage collection

The Java runtime environment also includes a garbage collector which implements persistence by reachability. Each object is preserved by the Java runtime as long as it is reachable by at least one execution thread in the virtual machine. If an object is not reachable anymore, all the resources (mainly memory) allocated for that object are collected by the runtime and may be reused for other objects. Garbage collection is transparent to user programs, which only have to deal with object references. Java keeps track of all the Java references which point to one object and the object is garbage-collected when the last reference to the object is reassigned.

Access to system level resources

Java provides access to resources managed at the operating system level such as files or network connections. In particular, Java allows a program to open, read and write files and to store the result of a serialization into a file, therefore managing persistent Java objects on disk. Classes for binding to a remote machine port are also provided, both through a socket interface or through the HTTP protocol in order to query a Web server.

3. Motivation

The main motivation for Javanaise is to provide adequate support for developing and executing cooperative applications on the Internet. Cooperative applications aim at assisting the cooperation between a set of users involved in a common task. An example of cooperative application is a structured editor which allows

documents to be shared concurrently by remote users. We implemented such a distributed editor in former project [Decouchant93].

These applications are characterized by a large amount of shared data structures which are browsed or edited by cooperating users connected from remote workstations. Since these data structures should be brought to the accessing nodes, at least to be displayed and sometimes to be modified, a caching strategy should be used. Defining the unit of sharing and consistency is the key issue to efficiency.

Another important issue for this service is to facilitate the installation and administration of software. In an intranet, application installation is often managed by system administrators who are responsible for installing these applications properly and also ensuring that they do not act as Trojan horses in the intranet. However, requiring any cooperative application to be officially installed (by administrators) is constraining; in addition, it is a difficult issue for administrators to decide that an application can be trusted. An alternative is to allow these cooperative applications to be freely downloaded on a Java virtual machine (just like applets), thus benefiting from a dynamic deployment of the applications with the guarantee that the application cannot corrupt the local host, thanks to Java's type safety.

Finally, we want to allow programmers to develop application as if they were to be run centralized. Then, a programmer can debug and test its application on a single machine, and then after a simple configuration step, run it distributed using our system support.

The three motivations described above (efficiency, easy administration, easy development) constitute the guideline which leads us to the design of Javanaise.

4. Basic design choices

This section presents our design choices for the Javanaise system support. Their translation to implementation principles is presented in the section that follows.

4.1. Managing clusters

The main problem we have to solve is to efficiently manage distributed replicas of Java objects while keeping distribution transparent to the application programmer:

- Managing object replicas requires mechanisms for faulting on objects, invalidating and updating objects in order to ensure consistency. These mechanisms should be hidden to the application programmer, who should only manipulate Java references as if every object were local.
- Previous experiments with the management of distributed fine grained objects have shown that efficiency is closely linked with object clustering [Benzaken90]. A cluster of objects is a group of objects which is supposed to be coarser grain (than a single object). Therefore, since system mechanisms are generally applied to coarse grained resources (e.g. IOs), they are applied to clusters, thus factorizing the costs of these mechanisms for all the objects within a cluster. However, clustering works well only if objects co-located within the same cluster are effectively closely related at execution time.

The mechanisms we want to factorize are naming, binding and consistency mechanisms. In order to be able to dynamically bind a reference to a remote object, we need to associate a unique name with each object, thus allowing the object to be located and brought to the requesting node. In order to implement object binding[3], we need to manage indirection objects that allow object faults to be triggered if the reference is not yet bound. In

[3] without modification to the Java virtual machine

order to manage objects consistency, we need to exchange messages between cooperating nodes to invalidate and update copies according to a consistency model.

Managing clusters of objects is a means for amortizing these costs (indirection objects, messages) over a group of objects that are inter-dependent. Inter-dependence means here that if one object of the group is accessed, most of the objects included in the group are likely to be used in the near future.

4.2. Application dependent clustering

Our previous experiments with object clustering (in the Guide system [Hagimont94]) relied on a system support that allows any object to be stored in any cluster. The system exports to applications a cluster management interface allowing objects to be stored in or migrated to any cluster. From the programmer's point of view, managing clustering is complex and most of the time leads to a default policy which is inefficient and doesn't actually use the flexibility of the clustering interface.

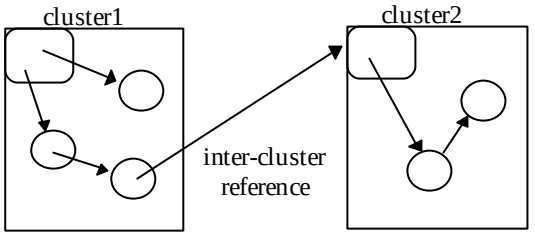

Figure 1. Management of clusters

In Javanaise, we propose to implement what we call *application dependent clustering*. This approach is inspired by the observation that cooperative applications tend to manage logical graphs of objects in their data structures. For example, a cooperative structured editor manages chapters that are composed of sections, themselves composed of subsections and paragraphs. We claim that some of these graphs should be managed as clusters by the system since they correspond to closely related objects according to the application semantics.

In Javanaise, a cluster is an application-defined graph of Java objects. A cluster is identified by a Java reference to a first object (called a *cluster object*) and the graph that defines the cluster is composed of all the Java objects that are accessible from the cluster object (the transitive closure). The boundaries of this graph are defined by the leaves of the graph and by the references to other cluster objects. A reference to another cluster object is called an *inter-cluster reference* (Figure 1). The Java objects within a cluster are called *local objects*.

A cluster object is an instance of a class (defined by the programmer) which has been defined (when the application is configured to be run distributed) as being a cluster class. Only interfaces of cluster objects are exported to other cluster objects, which means that the interface of a cluster object may only include methods whose reference parameters are references to cluster objects. Therefore, local objects in one cluster are only accessible from objects within the cluster.

Our assumption, which is based on our experience with cooperative applications development in the Guide project [Hagimont94, Decouchant93], is that cooperative applications tend to implicitly manage such clusters in their data structures. In section 5.4, we consider some extensions of this model, that allow several entry points in a cluster and dynamic object migration between clusters (reclustering).

4.3. Application programming

The programmer develops applications using the Java language without any language extension nor system support classes (libraries). An application can be debugged and tested locally (on one machine).

Configuring the application for distribution first consists in specifying which classes are cluster classes. The configurator should take into account the data structures managed in the application, i.e. the links between the classes that compose the application. However, this separation between the configuration and the application code makes it possible to experiment with different configurations for the same application without any modification to the application. However, we claim that logical groupings already exist in most cooperative applications.

Since an application is developed centralized, it does not deals with synchronization and consistency problems. A second step in the configuration is to associate a synchronization and a consistency protocol with each cluster. This is done at the level of the interfaces of the cluster classes. The interfaces of the cluster classes are annotated with keywords that define the consistency and synchronization protocols associated with the clusters.

At the moment, we have only implemented a single reader / multiple writers protocol. In the interface of a cluster, it is possible to associate a mode (reader or writer) with each method. When the method is invoked on a cluster instance, a lock in that mode is taken and a consistent copy of the cluster is brought to the local host. However, we will experiment with different consistency/synchronization protocols in the near future.

5. Implementation principles

Here, we describe the implementation principles used to manage distributed shared clusters of Java objects.

5.1. Managing cluster binding

Since a cluster is a graph of Java objects, clusters may be brought dynamically on a requesting node using the Java serialization mechanism.

The problem is to manage dynamic binding of references to objects that may be brought dynamically from remote nodes. Since the unit of naming and caching is the cluster, we have to provide a mechanism for dynamic binding of inter-cluster references.

Our implementation relies on intermediate objects called *proxies* [Shapiro86] which are transparently inserted between the referenced cluster and the cluster which contains the reference (Figure 2). A proxy contains a Java reference that points to the referenced cluster object if it is already there and null if not. It also contains a unique name associated with the cluster, allowing the cluster to be located and a copy to be brought on the local host.

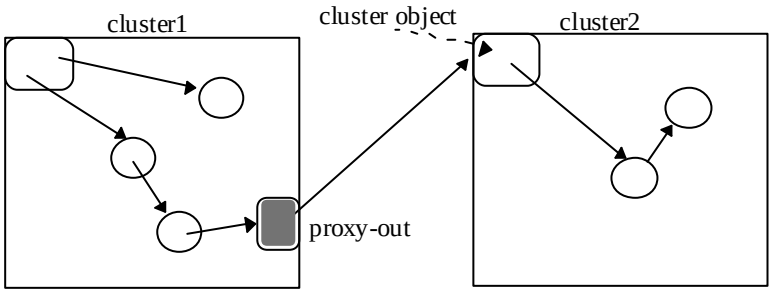

Figure 2. Binding of inter-cluster references

The class of the proxy object is generated from the interface of the cluster class to which it points. This proxy implements the same interface as the cluster object. Each method invocation is forwarded to the actual cluster object if the reference is already bound, i.e. if the Java reference in the proxy is not null. If this Java

reference is null, then a function of the runtime system is invoked in order to check whether the cluster is already cached (subsequently to the binding of another inter-cluster reference). A copy of the cluster is fetched if required and the Java reference in the proxy object is updated.

In the following, we call this proxy a *proxy-out object*. Proxy-out objects are stored in the cluster which contains the reference to the cluster.

5.2. Managing cluster consistency and synchronization

First, the problem is to manage invalidates and updates of clusters according to a consistency protocol. In this section, we only describe the mechanism that we used, independently from the consistency protocol which is applied.

A cluster can be invalidated on one node (Java virtual machine) simply by assigning to null the Java references in the proxy-out objects that reference the cluster. All the Java objects included in the invalidated cluster are then automatically garbage collected by the Java runtime. However, instead of dynamically looking for all the proxy-out objects that point to the invalidated cluster (which would be complex and inefficient), we decided to manage another type of proxy called *proxy-in object*, which is inserted between the proxy-out object and the cluster it points to (Figure 3). A proxy-in object is stored in the cluster which is referenced. Similarly to proxy-out objects, a proxy-in object forwards method invocations to the referenced cluster if its internal Java reference is not null. If this Java reference is null, then a function of the runtime system is invoked in order to fetch a consistent copy of the cluster and the Java reference in the proxy-in object is updated.

Then, a cluster invalidation on one node simply consists in assigning to null the Java reference in its associated proxy-in object.

Therefore, we deal with two kinds of cluster faults:

- proxy-out faults. When the Java reference in a proxy-out object is null, a copy of the referenced cluster is fetched. This copy of the cluster includes a proxy-in object pointing to the cluster object.
- proxy-in faults. When the Java reference in a proxy-in object is null, a copy of the referenced cluster is fetched. This copy of the cluster does not include a copy of the proxy-in object which is already there.

In both cases, the consistency protocol may require the cluster to be invalidated on some other nodes, using its proxy-in object on that node.

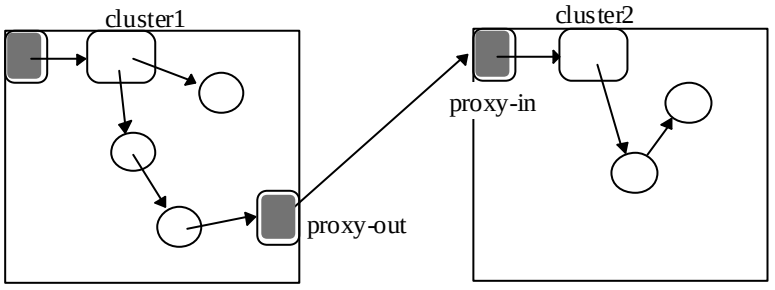

Figure 3. Consistency of cluster objects

At this point, we are able to invalidate and update clusters for consistency management. With the mechanisms described above, we can manage strong consistency for a cluster, ensuring that only one copy of the cluster is accessible on one node.

However, in most cases, a synchronization scheme is associated with the consistency protocol. For example, this is the case for the entry *consistency protocol* [Bershad93] which allows applications to lock objects

and guarantees objects coherence only when a lock is taken. Such a synchronization scheme can be implemented in the proxy-in object.

In our prototype, we allow an access mode (reader or writer) to be associated with each method in a cluster interface, which means that a lock on the cluster must be taken before entering the method. This locking strategy in managed in the proxy-in object which knows which lock is being held on the current node. An invalidation on one node is in this case a lock request to the proxy-in object, that may block until all locks are released on that object.

5.3. Managing reference parameter passing

In the interface of a cluster object, methods may only have reference parameters that are references to cluster objects. When such a reference is passed at execution time, the system must ensure that a reference which enters a cluster will point to a proxy-out object. This is ensured by the proxy-out objects for onward parameters and the proxy-in objects for backward parameters (Figure 4).

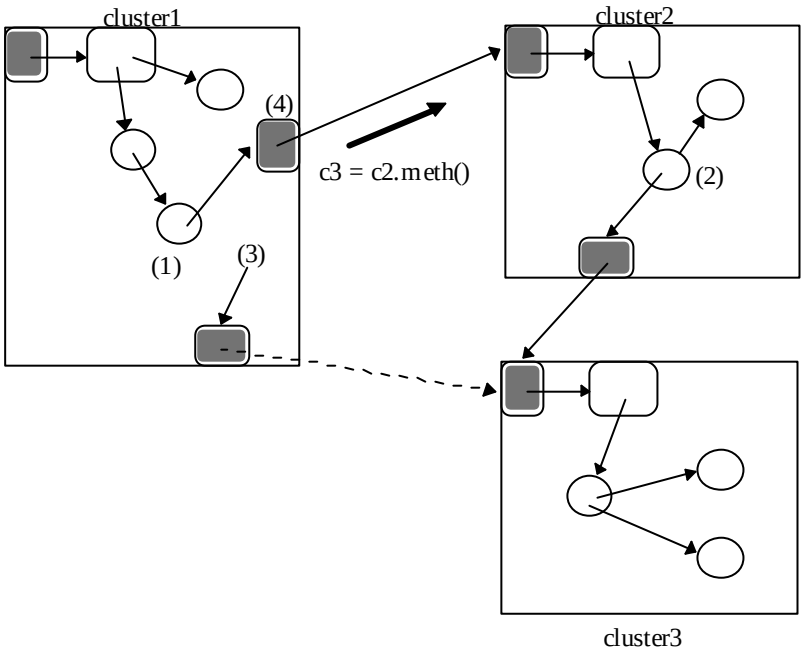

Figure 4. Parameter passing

In Figure 4, a local object in *cluster1* (1) performs an invocation (*c3 = c2.meth()*) on *cluster2*. The invoked method returns a Java reference stored in *cluster2* (2), which is a reference to *cluster3*. In order to be able to store a reference to *cluster3* in *cluster1*, the system must create a proxy-out which points to *cluster3* (3). This proxy-out object is created by the proxy-out in *cluster1* which is associated with the reference to *cluster2* (4).

An onward reference parameter would be managed similarly by the proxy-in object in *cluster2*.

When managing proxy-out objects for entering parameters in a cluster, we need to guarantee that all the references to a cluster *C* within the cluster point to the same proxy-out object. This is especially important when comparing two variables that contain cluster references (within one cluster). To do this, we manage in each cluster a table which registers the proxy-out objects which already exist in the cluster. When a reference enters the cluster and if an associated proxy-out object already exists in the cluster, then this proxy-out object is used

and no additional proxy-out object is created. Therefore, we avoid having two proxy-out objects associated with the same external reference in one cluster.

5.4. Other clustering issues

In this section, we consider two variants of our clustering model which we plan to investigate : clusters with several entry points and reclustering. Both variants preserve the model in the sense that they don't require hardcoding a clustering policy in the application code. The clustering policy remains based on the data structures and the interfaces managed in the application.

We first consider the management of several entry points in clusters. Until now, we have imposed the restriction that any interaction with a cluster should take the form of a method invocation on the cluster object (the unique entry point of the cluster). This implies that reference parameters in the interface of a cluster are always references to cluster objects. However, we could allow cluster interfaces to include local object reference parameters. This would require the ability to dynamically install proxy-out and proxy-in objects, thus managing several entry points in clusters. It would not be difficult to extend our prototype in order to implement this variant of the model. We plan to experiment further in order to evaluate its interest.

The second variant is to provide support for reclustering. Reclustering means here the ability to move a local object from one cluster to another. We propose to allow a local object to be migrated when its reference is passed as parameter of an inter-cluster invocation. The interface of the cluster may include a "move" statement associated with the reference parameter, meaning that the local object should be moved to the destination cluster (similarly to the *call-by-move* feature of Emerald[Jul88]). Then the local object in the source cluster must be transformed into a proxy-out object, and the proxy-out object (if there is one associated with this local object) in the destination cluster must be changed into the migrated local object.

These two variants constitute interesting perspectives since they preserve the spirit of our clustering model.

6. Prototype on Java

We have implemented a prototype of this system support and we describe it in this section.

6.1. Overall architecture

The overall architecture is illustrated on figure 5.

In our prototype, we assume that the code of Javanaise, the code of the application and the persistent clusters are stored on a node called the Home Site of the application. The code of Javanaise and the code of the application are dynamically deployed to the client nodes when the application is invoked. An application is made available on the Home Site through a Web server and identified and located with a URL. The application is launched using an Applet viewer, the code of the application and of Javanaise being downloaded on client nodes just like an Applet. Clusters are then fetched on demand by the requesting nodes and shared following the consistency protocol.

The Javanaise system support consists of:

- the proxy generator which generates proxy-out and proxy-in classes from the interface of a cluster. Proxy-out and proxy-in classes provide mechanisms for reference binding and consistency management.
- some system primitives (*Javanaise client*) that are available on the client nodes and used by the proxies on these nodes. *Javanaise client* maintains a table of the clusters (proxy-in objects) that are already present on the local host.

- some system code (*Javanaise server*) that participates in the binding and consistency protocols on the home site. The Javanaise server maintains a table which registers the locations and locks held for all the clusters.

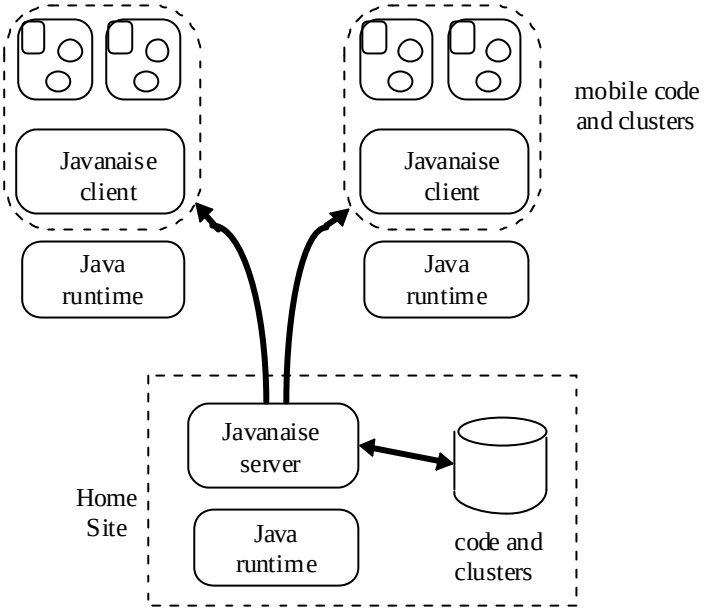

Figure 5. Architecture

6.2. Generation of proxies

Proxies are generated from the interface of a cluster class using a proxy generator. Let's see the Java code skeleton of the proxy-out and proxy-in classes generated from the cluster interface below.

```
public class Cluster1 implements Cluster1_itf {
  public void method1 (Cluster2_itf obj);   : read
  public Cluster3_itf method2 ();           : write
}
```

This definition describes the interface of the cluster class *Cluster1* which implements the *Cluster1_itf* interface. Two methods are defined, the first one taking a onward reference parameter and the second returning a reference parameter. The first method requires a read lock before execution and the second a write lock.

Below is the proxy-out class generated from the above interface definition. The name of this class is actually *Cluster1*, because when a program manipulate a reference to a *Cluster1* instance, it is in fact a reference to a proxy-out object which implements the same interface.

This class defines two instance variables, *object_id* which is the unique identifier of the object the proxy-out refers to, and *proxy-in* which points to the proxy-in object. The identifier is unique in the context of the application; our prototype currently manages shared objects between applications located on a single Home Site, but it can be extended to manage different Home Sites simply by using URLs.

```
public class Cluster1 implements Cluster1_itf {
  int                  object_id;
  Proxy_in_Cluster1    proxy_in;

public Cluster1 () {
  proxy_in = new Proxy_in_Cluster1();
  object_id = proxy_in.my_id();
}

public void method1 (Cluster2 obj) {
  if (proxy_in == null)
     proxy_in = (Proxy_in_Cluster1)      javanaise_client.get_proxy_in(object_id,
read);
     proxy_in.method1(obj);
}

public Cluster3 method2 () {
  if (proxy_in == null)
     proxy_in   =   (Proxy_in_Cluster1)   javanaise_client.get_proxy_in(object_id,
write);
     Cluster1 p = proxy_in.method2();
     p = clone_proxy-out(p);    // clone proxy-out for backward parameter
     return p;
}
}
```

The first method is the constructor. When a user program invokes the creation of a *Cluster1* object, it actually creates a proxy-out object which in turns creates a proxy-in object which creates the real instance of *Cluster1* (see below in the proxy-in class). If a constructor with parameters is defined, a constructor is generated accordingly in the proxy-out and proxy-in classes and implements the same interface.

The two other methods check the binding of the reference to the cluster. If the reference has not already been bound, *Javanaise client* is invoked is order to get a copy of the cluster (including the proxy-in object) with a lock in the corresponding mode[4]. Then the invocation is forwarded to the proxy-in object.

Notice that the *method2* method returns a reference to a *Cluster3* instance. The reference returned by this method is a reference to a proxy-out object in the invoked cluster. Then, we must create a clone of this proxy-out object, which will be stored in the cluster receiving the reference. This clone gets created only if a proxy-out object for the received reference doesn't yet exist in the receiving cluster. The table which registers the existing proxy-out objects in the cluster is stored in the cluster itself. In this code skeleton, we omitted the code related to the management of this table (to keep it readable).

Below is the generated proxy-in class. The proxy-in class also defines a variable *object_id* for the unique identifier of the object and a variable *object* which points to the real cluster instance. It also contains a variable *lock* which indicates the lock that is held on the local host on this cluster (read, write or none).

[4] The locking mode is specified here in order to get both the copy of the cluster and the lock on that cluster in a single request.

```
public class Proxy-in_Cluster1 implements Cluster1_itf {
    int                 object_id;
    Real_Cluster1       object;
    int                 lock;
    int                 num_readers;

public Proxy_in_Cluster1 () {
    object = new Real_Cluster1();
    object_id = javanaise_client.register_id(this);
    num_readers = 0;
}

public void method1 (Cluster2 obj) {
    if ((object == null) || (lock == none)) {
      object = (Real_Cluster1) javanaise_client.get_object(object_id, read);
      lock = read;
    }
    Cluster2 p = clone_proxy_out(obj);      //   clone   proxy-out   for   onward
parameter
    num_readers ++;

    object.method1(p);               // effective call

    num_readers --;
    if ((lock == read) && (num_readers == 0)) {
      javanaise_client.release_lock(object_id, read);
      lock = none;
    }
}

public Cluster3 method2 () {
    if ((object == null) || (lock != write)) {
      object = (Real_Cluster1) javanaise_client.get_object(object_id, write);
      lock = write;
    }

    object.method1(obj);                    // effective call

    javanaise_client.release_lock(object_id, write);
}
}
```

The constructor of the proxy-in class creates the real instance of *Cluster1* which has been renamed in *Real_Cluster1* (by the proxy generator) since the proxy-out class has been renamed in *Cluster1*. Then, *Javanaise*

client is invoked in order to allocate a unique object identifier[5]. A reference to the proxy-in object is passed in order to initialize Javanaise internal tables (detailed below).

The two methods checks whether a copy of the cluster with the requested lock is present. If not, a copy and/or a lock are requested to *Javanaise client*. The locking policy currently implemented allows multiple readers and one writer at a time.

A lock held on a cluster is managed in the proxy-in object of the cluster. The proxy-in object invokes Javanaise client in order to request the lock. The proxy-in object also includes primitives (upcalls not present in the above skeleton) that may be invoked by Javanaise client in order to check the status of the lock on the cluster and block a lock request from a remote node until the lock is explicitly released. The methods in the proxy-in object which manipulate the lock variable are synchronized.

6.3. Management of consistency and synchronization

In order to manage consistency and synchronization, two tables are maintained, one in *Javanaise client* and one in *Javanaise server*.

First, in *Javanaise server*, we need to be able to locate any cluster and any lock. *ServerTable* is a table which keeps track of all the clusters that have an image in memory (see next section for clusters stored on disk). This table associates with each cluster (known by its *object_id*) the locations (node addresses) of all the images of the cluster in memory. If the cluster is in read mode, the table gives a list of the nodes that obtained a read lock on the cluster. If the cluster is in write mode, the table gives the address of the node which hosts the unique copy.

This table is used in order to locate one copy (in read mode) or the last copy of the cluster. When a collision on a lock occurs (the cluster is locked in write mode), the request is sent to the writer node and *Javanaise client* responds only when the lock is released.

The second table (*ClientTable*) is managed in *Javanaise client*. This table records the clusters that are cached on the local host. This table associates with each cluster (*object_id*) the Java reference to the proxy-in object which represents the cluster. Using the *ClientTable*, *Javanaise client* can check the status of the cluster on this node and wait for a lock to be released by the proxy-in object (with the *release_lock* primitive).

All the interactions between Javanaise clients and Javanaise server are based on message passing using the socket interface. A message always includes a header object which describes the message type. This header may be followed by (i.e. point to) a cluster object which is a graph of Java objects. These objects, the header and the cluster, are serialized in order to obtain a flat string of bytes which is sent on the socket. In order to be serializable, each object of the application must implement the *Serializable* Java interface, which means that it inherits default methods that are invoked to serialize the object. The default serialization behavior is to flatten the object state and to do it recursively following every Java reference in the object state. To stop this recursion, we redefined these serialization methods for the proxy-out objects : we only save the *object_id* field of the object and reset to null the Java reference when the object is deserialized.

6.4. Management of persistence

In Javanaise, clusters are persistent, which means they survive the application that created them. Clusters are stored on disk on the Home Site of the application, one file per cluster. Recall that each cluster is identified with an *object_id* (an integer) which is a unique identifier.

In Javanaise server, the *ServerTable* keeps track of all the clusters that have an image in memory on one of the client hosts. When a cluster is requested and the cluster is not represented in memory, the cluster is read from its storage file. The byte stream read from the file is deserialized and sent to the requesting client (and the

[5] In the current prototype, this allocation is forwarded to Javanaise server which ensure the identifier uniqueness.

ServerTable updated). When an application terminates on a client node and if some modified clusters are no longer used (cached on any node), they are serialized and copied back to their storage files on the Home Site.

6.5. Deployment of an application

Applications in Javanaise are deployed dynamically to the client nodes just like Applets. A Javanaise application is made available as an Applet through a Web server and a client typically starts the application using an Applet viewer. This Applet contains the *main* entry point of the program.

An initialization primitive is provided, which initializes the Javanaise environment, but also returns a Java reference to a name server object. This name server allows symbolic names to be associated with Cluster objects' references. The *register* method registers a cluster reference with a given symbolic name and the *lookup* method returns the cluster reference associated with a symbolic name. In the implementation, passing a cluster reference to the *register* method is actually passing a proxy-out object reference to the name server (which registers a copy of the proxy-out object). Getting a cluster reference from the name server is actually getting a proxy-out object.

We have described the prototype of the Javanaise runtime we have implemented on top of Java. In the rest of the paper, we relate our work to previous experiments and conclude with our continuation perspectives.

7. Related work

A first attempt to provide distributed shared objects on the Internet is Java-RMI [Wollrath96] which provides remote method invocation between Java objects. A Java program can query a Web server with a URL in order to obtain a Java reference. A stub is dynamically loaded and bound to the application, thus allowing remote invocation on the referenced object. Additional object references may be returned and are similarly dynamically bound to remote instances. A mechanism called *object serialization* allows distributed programs to exchange copies of objects. However, using the RMI facilities, distributed applications are based on the client-server architecture which does not allow objects to be cached and therefore accessed locally. It is possible to manage object replicas using the object serialization facility, but the coherence between the replicas has to be explicitly managed by the application programmer.

One of the emerging paradigm for structuring applications over the Internet is agent-based programming [MAF97]. An agent is roughly a process with its own context, including code and data, that may travel among several sites in order to perform its task. Generally, an agent can invoke objects exported either by the servers it visits or by other agents running on these servers. This paradigm seems very adequate for information search engines or electronic commerce, but it does not provide support for information sharing between remote machines. Mobile agents are relevant to Javanaise since Javanaise applications are dynamically deployed on remote nodes just like agents do.

W3Objects [Ingham95] is a project which aims at defining an object-oriented framework for developing distributed application on the Internet. They propose to extend the Web using object-orientation techniques in order to make the integration of complex resources and services feasible. Their proposals mainly rely on extensions to HTTP in order to provide remote object invocation between heterogeneous resources. This idea is very promizing since it is open to the integration of a wide range of already existing applications, but it does not address the issue of caching which is one of the most difficult in the Web.

Most of the work described in this paper was influenced by the ideas developed in the Guide project [Hagimont94], a former project of the proposing team. The Guide system aimed at providing a distributed shared object space for the development of cooperative applications on a local area network. Memory management in the Guide system was structured in two layers: the storage memory composed of permanent objects stored in clusters on distributed disks and the execution memory composed of clusters in use (i.e. mapped) by running applications. In Guide, a cluster was mapped on the accessing node at first use of an object stored in the cluster. Subsequent accesses from other nodes to this cluster's objects were forwarded to and performed on that mapping

node using a traditional remote invocation scheme. Object clustering was managed by applications using the interface exported by the runtime environment. Therefore, compared to the Javanaise runtime, Guide was not managing distributed replicas of clusters and it let applications manage their own clustering policies, which led to inefficient default clustering policies. Instead, Javanaise, which targets distributed applications on the Internet, relies on clusters caching on the client nodes and its clustering policy is implicitly derived from the applications structures.

Our work on Javanaise can also be related to another object-based system of the 80s which is the Clouds system [Dasgupta90]. Clouds provides essentially the same paradigm than Guide, i.e. a distributed shared object space, but it differs by the management of object granularities. Clouds objects are coarse-grained since an object is implemented by a virtual address space. Clouds objects are developed using the CC++ (Clouds C++) language, but Clouds allows the management of finer grained objects (C++ objects) within a Clouds object [Dasgupta91]. Therefore, Clouds objects can be compared to Javanaise clusters which contain finer grained Java objects.

Javanaise can be seen as a grant child of these systems. Its main contribution is to study the application of a similar paradigm (a distributed shared object space) to cooperative applications on the Internet. This paradigm has been integrated into the Java world in order to allow dynamic deployment of shared applications to the client hosts.

8. Conclusion and Perspectives

In this paper, we presented our experience with providing a runtime environment for the development of cooperative application on the Internet. In our environment, applications are developed using the Java language, and made available on the Internet through a Web server just like an Applet. Applications are dynamically deployed to client nodes, thanks to Java mobile code, and the Java objects managed by the applications are transparently shared between application instances.

In order to allow for efficient object caching on client nodes, our runtime manages clusters of Java objects, the cluster being the unit of sharing, caching and consistency. Clusters are persistent and stored on disk on the Home Site of the application. Clusters are brought on demand on the client nodes and shared following a specified consistency and synchronization protocol. Our claim is that object grouping in clusters can be derived from the application structure by specifying which classes correspond to clusters, other classes defining local objects within clusters.

We have implemented a prototype of the Javanaise runtime, composed of a preprocessor which generates required proxy classes from the interfaces of cluster classes, and of system classes which manage consistency of cluster replicas cached on client nodes. Our prototype was validated using simple applications and we are currently porting a structured cooperative editor [Decouchant93] on top of this runtime.

The perspectives of this work are first to validate and evaluate this system support through full scale cooperative applications such as the editor mentioned above. Then, this preliminary experiment opens many challenging issues.

We first would like to experiment with the model extensions proposed in section 5.4, i.e. managing cluster with several entry points and dealing with reclustering issues. Second, we would like to investigate the combination of using invoking a cluster replica locally and of invoking a cluster remotely using an RMI-like mechanism. The choice between these two mechanisms should be based on the semantic of the cluster objects and the amount of write sharing.

Bibliography

[Arnold96] K. Arnold and J. Gosling. *The Java Programming Language*, Addison-Wesley, 1996.

[Benzaken90] V. Benzaken, C. Delobel, “Dynamic Clustering Strategies in the O2 Object-Oriented Database System”, 4th International Workshop on Persistent Object Systems: Design, Implementation and Use, September 1990.

[Bershad93] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon, "The Midway Distributed Shared Memory System", *Proc. 38th IEEE Computer Society International Conference (COMPCON'93)*, pp. 528–537, February 1993.

[Black87] A. Black, N. Hutchinson, E. Jul, H. Levy, L. Carter, “Distribution and abstract types in Emerald", *IEEE Transactions on Software Engineering*, 13(1), January 1987.

[Dasgupta90] P. Dasgupta, R. Chen, S. Menon, M. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R. LeBlanc, W. Appelbe, J. Bernabeu-Auban, P. Hutto, M. Khalidi, C. Wilkenloh, "The Design and Implementation of the Clouds Distributed Operating System", *Computing Systems*, 3(1), pp. 11-45, Winter 1990.

[Dasgupta91] P. Dasgupta, R. Ananthanarayanan, S. Menon, A. Mohindra, M. Pearson, "Language and Operating System Support for Distributed Programming in Clouds", *Proceedings of the 2nd Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS)*, March 1991.

[Decouchant93] D. Decouchant, V. Quint, M. Riveill, I. Vatton, *Griffon: A Cooperative, Structured, Distributed Document Editor*, (93-20), Bull-IMAG, May 1993.
http://sirac.imag.fr/~hagimont/papers/93-20-griffon-RT.ps.gz

[Hagimont94] D. Hagimont, P.Y. Chevalier, A. Freyssinet , S. Krakowiak, S. Lacourte, J. Mossière et X. Rousset de Pina, "Persistent Shared Object Support in the Guide System: Evaluation & Related Work", *Ninth Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, Portland, October 1994.

[Ingham95] D. Ingham, M. Little, S. Caughey, S. Shrivastava, "W3Objects: Bringing Object-Oriented Technology to the Web", 4th International World-Wide Web Conference, Boston, December 1995.

[Jul88] E. Jul, H. Levy, N. Hutchinson, A. Black, "Fine-grained mobility in the Emerald system", *ACM Transactions on Computer Systems*, 6(1), February 1988.

[MAF97] Mobile Agent Facility Specification, OMG TC Document orbos/97-09-20, September 1997.

[Riggs96] R. Riggs, J. Waldo, A. Wollrath, K. Bharat "Pickling State in the Java System", Computing Systems, 9(4), pp. 313-329, Fall 1996.

[rpcgen88] rpcgen - An RPC Protocol Compiler, Sun Microsystem, Inc., 1988.

[Sandakly97] F. Sandakly, P. Poyet, "Java and code mobility in B&C Applications, A Case Study", *Workshop on Persistence and Distribution in Java (Perdis European Project)*, Lisbon, October 1997.
HTTP://www.perdis.esprit.ec.org/events/java-wkshp-971020/sandakly-javaBC.html

[Shapiro86] M. Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principle, *Proc. of the 6th International Conference on Distributed Computing Systems*, pp. 198-204, 1986.

[Sun96] Sun Microsystems, “JDK 1.1 Documentation”, Sun Microsystems,
URL: http://www.javasoft.com/products/jdk/1.1/docs/index.html

[Wollrath96] A. Wollrath, R. Riggs, J. Waldo, "A Distributed Object Model for the Java System", Computing Systems, 9(4), pp. 291-312, Fall 1996.