A Java-based system support
for distributed applications on the Internet

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. This service provides the abstraction of a distributed shared object space. 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.

One of the main characteristics of cooperative applications is that data must be copied to the accessing nodes, at least to be displayed and sometimes modified. So our service relies on object caching. It provides the same abstraction and interface as the RMI service of Java, but it extends the functionality by allowing shared objects to be cached on the accessing nodes. Objects are copied on demand to the requesting nodes and are cached until invalidated by the coherency protocol.

This service has been implemented on top of Java. It has been evaluated by comparison with RMI using a benchmark and validated by porting an existing mail browser application.

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 80's, a model has emerged for the support of cooperative distributed applications, that of a distributed shared universe organized as a set of objects. Distributed object-oriented systems such as Emerald Black87], Clouds [Dasgupta90] or Guide [Hagimont94] belong to this class of systems. More recently, the growth of the Internet, which is now used daily for cooperation, logically leads to 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. This is mainly because they provide a global naming scheme and machine independent code (to deal with heterogeneity). Therefore, a first step to provide distributed shared objects on the Internet was Java-RMI [Wollrath96] which provides remote method invocation between Java objects. Remote object references may be exchanged 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, where latency and bandwidth are highly variable.

In order to assist the programmer, we propose a new system service called Javanaise which implements the abstraction of a distributed shared Java object space. This service provides the

---

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

[2] Université Joseph Fourier, Grenoble

same abstraction and interface as the RMI service of Java, but it extends the functionality by allowing shared objects to be cached on the accessing node. Objects are copied on demand to the requesting nodes and are cached until invalidated by the coherency protocol. With this system support, the programmer can develop his application as if it were to be executed in a centralized configuration. Then, the application can be configured for a distributed setting without any (or with minor) 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 the Javanaise service are:

- Dynamic deployment. Applications are dynamically deployed to the client nodes from the node that hosts the application; thus we do not require applications to be installed on a machine prior to execution.

- Caching. Our system support allows shared Java objects to be cached on cooperating nodes, thus enabling local invocation on distributed objects and reducing latency. This is one of the key requirements of cooperative applications.

- 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. This also enables adaptation of existing centralized applications in order to run distributed.

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. Our experience and evaluation of the prototype is reported in section 7. After a discussion of related work in section 8, we conclude in section 9.

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. 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 retains reachable objects. 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.

Remote method invocation

Java provides a service called RMI which allows for objects to be invoked from a remote Java virtual machine. A class may be defined as being a *Remote* class, which means that instances of this class may be invoked remotely. RMI provides a stub compiler which generates stubs and skeletons from Remote classes definitions. A name server (called *Registry*) allows a remote object reference to be exported on one site. Then, a client Java program on any machine can fetch this reference and remotely invoke a method on the object. Subsequently, other remote references may be returned to the client as parameter of this invocation.

An important characteristic of RMI is that the interface of a Remote class is restricted to include Java reference parameters to either Remote object or Serializable objects. Passing a Remote object reference ensures that the object can be invoked from any other machine; the

object is thus passed by reference. Passing a Serializable object reference allows RMI to provide a copy of the object; the object is thus passed by value.

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 a former project [Decouchant93].

Cooperative 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 copied 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 motivation for Javanaise is to allow remote access to applications without requiring them to be installed on the machine prior to execution. An alternative to installation is to allow these 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. Dynamic deployment of applications also allow users to access the same application (including some persistent data) from any machine connected to the Internet, as long as a Java virtual machine runs on that machine.

Finally, we want to allow programmers to develop applications as if they were to be run centralized. We want to minimize the impact of distribution on application development. Thus, a programmer should be able to develop, debug and test its application on a single machine, and then after a simple configuration step, run it distributed using our system support. Porting an existing centralized application should also be simple and require only minor modifications to the source code.

The three motivations described above (caching, 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 [Hagimont94]. 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.

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 copied 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 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 does not actually use the flexibility of the clustering interface.

In Javanaise, our goal is to allow application specific clustering policies while keeping it transparent to the application programmer. 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*.
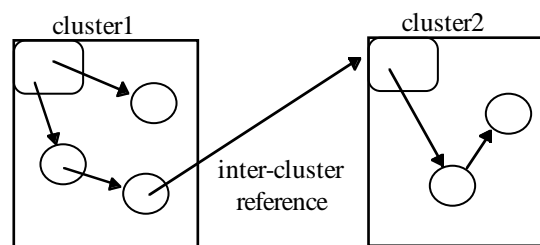

Figure 1. Management of clusters

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.

At this step, it is important to notice that RMI implicitly enforces application dependent clustering. Indeed, since RMI imposes that any reference parameter in the interface of a Remote class is a reference to a Remote object, a Remote object is completely equivalent to a cluster object. A Remote object can create many local (non-remote) objects but cannot export references to these local objects.

---

[3] without modification to the Java virtual machine

With RMI, as for Javanaise, configuring the application for distribution consists in specifying which classes are Remote (resp. Cluster). Then, only Remote objects are globally visible and the applications only pay for objects declared Remote.

The basic difference between Javanaise and RMI is that Javanaise provides cluster caching. This means that a cluster is loaded on the client node the first time it is used, and then accessed locally. Furthermore, Javanaise enforces objects consistency whenever objects are cached on different machines, as will be shown in the next section.

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 (just like an RMI user specifies which classes are Remote). 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 writer / multiple readers 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 copied dynamically to a requesting node using the Java serialization mechanism.

The problem is to manage dynamic binding of references to objects that may be fetched 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.
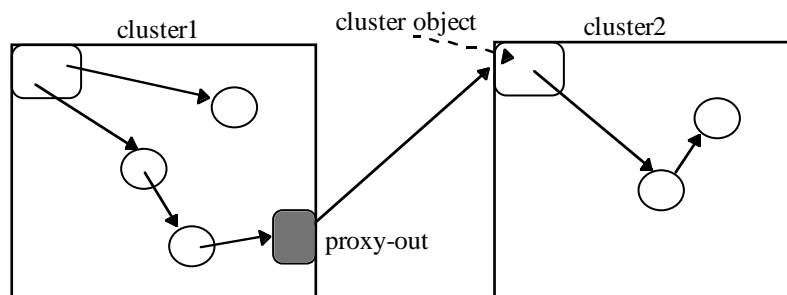

Figure 2. Binding of inter-cluster references

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.

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.
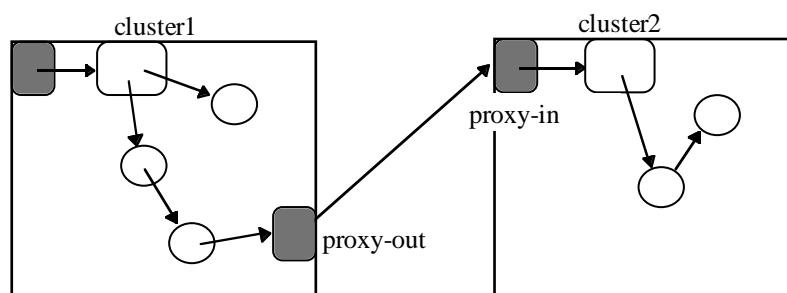

Figure 3. Consistency of cluster objects

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.

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).
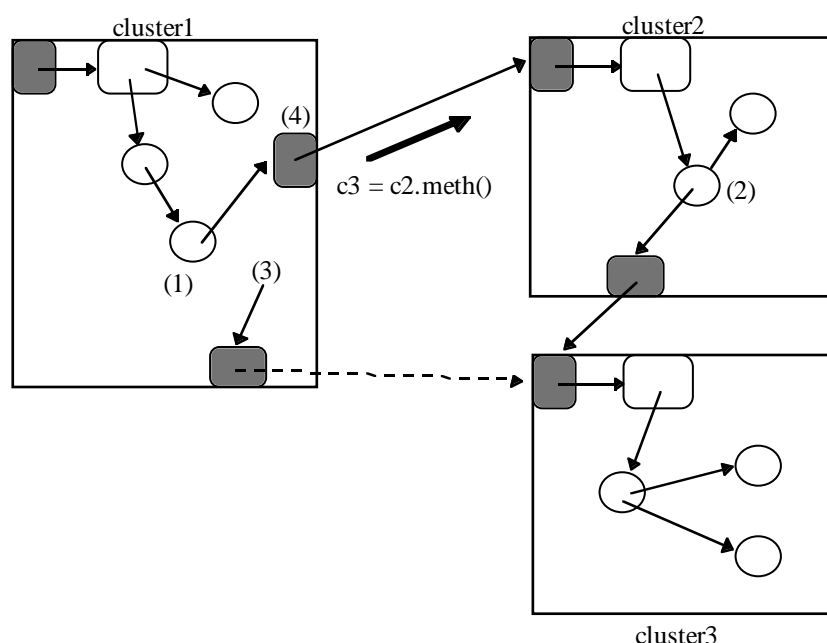

Figure 4. Parameter passing

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.

6.   Prototype on Java

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

6.1.   Overall architecture

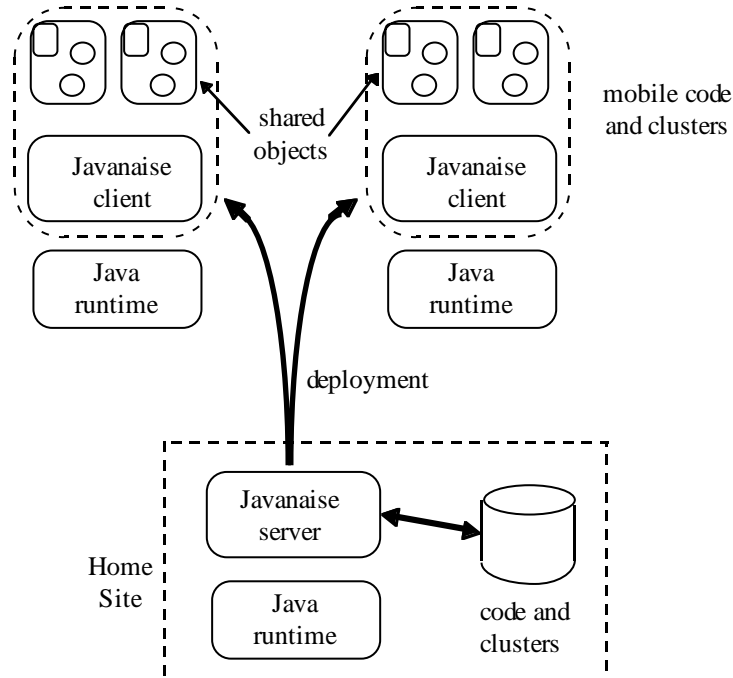The overall architecture is illustrated on Figure 5.

Figure 5. Architecture

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.

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[4] 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.

```
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;
}
}
```

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 distributed shared objects for applications stored on a single Home Site, but it can be extended to manage different Home Sites simply by using URLs.

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

---

[4] In the current prototype, proxy classes are generated from a text definition of the interface of the class. In the next prototype, we will generate our proxies directly from the actual class object, as RMI does.

(including the proxy-in object) with a lock in the corresponding mode[5]. 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 does not 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).

```
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);
}
}
```

[5] 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.

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[6]. 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

---

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

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 *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.   Experience and Evaluation

In this section, we present our evaluation of the Javanaise service. This evaluation consists of three steps. First, we measured the costs of basic operations in order to be able to understand precisely what is happening in the system and to better explain our other measurements. Second, to compare globally Javanaise with RMI, we designed and implemented a variant of the well-known Cattell benchmark [Cattell92], which roughly consists in a traversal of a distributed graph. Finally, to demonstrate the adequacy of the platform, we have ported an existing (centralized) application to Javanaise.

All the performance figures below were measured on a pair of Bull-Estrella machines based on PowerPC 604 machines (100 Mhz) running AIX and we used the JDK 1.1.2 version of the Java Virtual Machine. These machines are connected through a 10 Mbit Ethernet network[7].

In this section, we will use the term "object" to designate a Javanaise cluster or an RMI Remote object.

7.1. Basics costs

Table 6 gives the costs of the basic operations that are relevant to our experiment.

(1)	Java method invocation	1.45	microsecs
(2)	RMI invocation	4.00	millisecs
(3)	Javanaise invocation (cold)	5.00	millisecs
(4)	Javanaise invocation (hot)	4.50	microsecs

Table 6. Basic costs

Line 3 gives the cost of a method invocation in Javanaise in the case the object (a cluster) is remote, i.e. not yet cached locally (cold invocation). We made this measurement with a small

---

[7] Our measurements were done on a local area network in order to be able to compare Javanaise with RMI without network performance variation.

object size (75 bytes). This operation involves fetching a copy of the object from the remote site, installing it on the local host and invoking the method of the cached copy. This must be compared to the cost of invoking the same object with RMI (line 2). A Javanaise cold invocation is slightly more expensive than a RMI invocation. However, line 4 shows that it will be amortized if the same object is invoked more than once; while the cost of a RMI invocation remains the same, the cost of a Javanaise invocation falls down to 4.5 microseconds. The difference between the cost of a Javanaise hot invocation (line 4) and the cost of a Java method invocation (line 1) is due to the traversal of our two proxies.
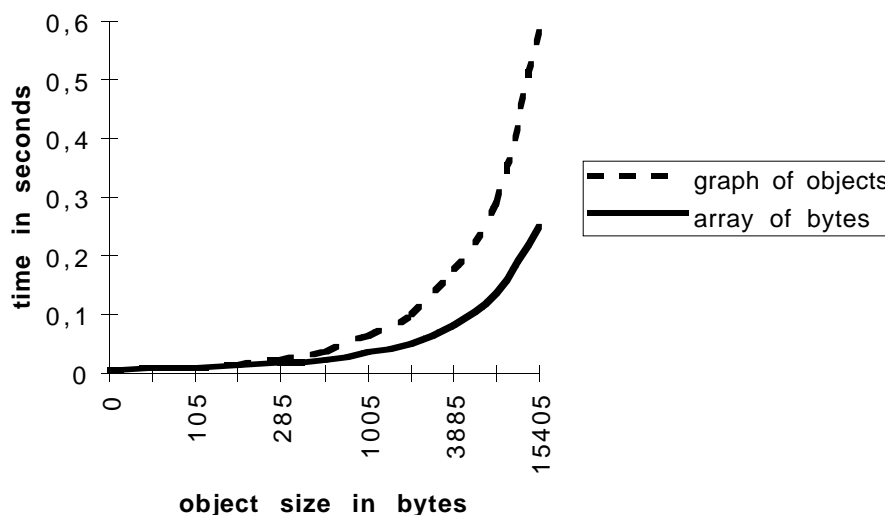

Figure 7. Javanaise method invocation for different object sizes

Since the cost of a Javanaise cold method invocation is highly dependent on the size of the remote object (a cluster), we measured it for different object sizes. The results are given on Figure 7. In order to know the cost due to object serialization, we performed the measurements with, for each object size, the object being an array of bytes and the object being a binary tree of small Java objects (the overall tree being the same size). The results indicate that object serialization is very expensive and that a better implementation of object serialization could greatly improve Javanaise performance.

However, the impact of objects sizes on Javanaise performance is not critical. Recall that we aim at supporting cooperative applications, for which most of the data structures should be copied to the accessing nodes (at least to be displayed and sometimes to be modified). An implementation of such an application using RMI would anyway copy these data to the accessing node.

7.2. Cattell Benchmark

The main motivation for Javanaise is to implement object replication on the client nodes since this is one of the key features required by cooperative applications. However, for the performance evaluation of our system support, we wanted to compare it to RMI for a general purpose workload which does not have strong requirements on object caching.

To perform a global comparison of Javanaise and RMI, we designed and implemented a variant of the Cattell benchmark, a benchmark used by practitioners of database systems. This application implements a traversal of a graph of 5000 nodes in which each node has three children nodes and the traversal is done down to seven levels (a total of 3280 nodes are visited). The nodes are equally distributed on two sites. The three children nodes are randomly chosen with a probability of 70% to belong to its father's site[8].

---

[8] Notice that a lower locality of inter-site references would benefit to Javanaise.

This benchmark has been implemented both on top of RMI and Javanaise. Notice that we did not change any line of source code, but we processed the same code with both the RMI and Javanaise stub generators. In both cases, the nodes are created on the two sites. With RMI, using a reference to a remote object implies a remote method invocation to that site. With Javanaise, using a reference to a remote object implies bringing a copy of the object on the local host and invoking the method locally.

(1)	RMI	6.4	seconds
(2)	Javanaise (cold)	29	seconds
(3)	Javanaise (hot)	19	milliseconds

Table 8. Results of the Cattell benchmark

Table 8 presents the results. In this benchmark the object size is 75 bytes. The results are very disappointing for Javanaise for a cold start. It performs 5 times slower than RMI. This inefficiency has two main reasons:

- first, the object graph has a high percentage of inter-object references which are local to one site (70%). This implies that when a remote object (on site 2) is brought to the accessing site (on site 1), its children on node 2 become remote and will have to be fetched. With RMI, when the remote object is invoked (on node 2), its children on node 2 are local and their invocations are very efficient.

- second, we observed that since many nodes are visited, it happens that some nodes are visited several times (i.e. the object is visited while already loaded) and this is one of the key condition to Javanaise efficiency. In the previous test-bed, we measured that in the average, object are reused 1.54 times. Increasing the amount of reused objects would advantage Javanaise. This is illustrated on Figure 9. In order to vary the amount of reuse of objects, we varied the depth of the traversal of the graph. The consequence is that, for a depth of 9, objects are reused about 6 times in the average.
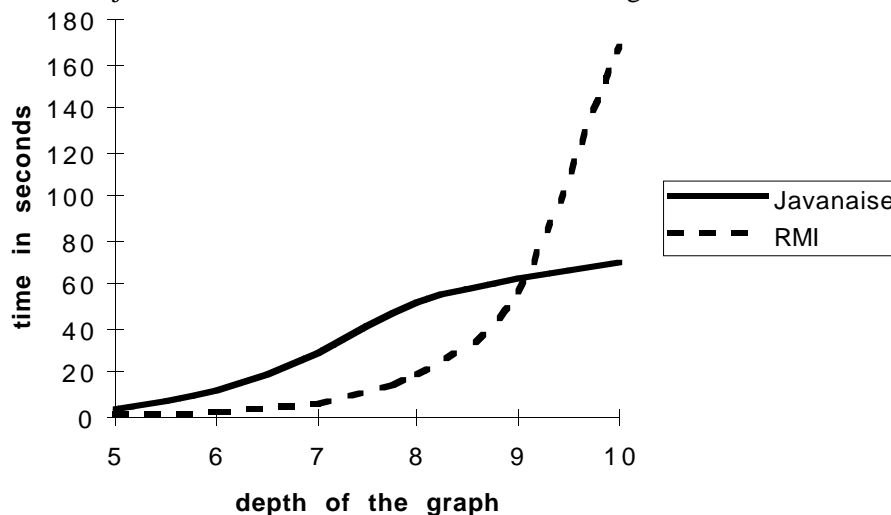

Figure 9. Cattell benchmark with different ratios of reused objects

The results show that according to the amount of reused objects in the graph, Javanaise can perform better than RMI for the first (cold) travel in the graph.

In conclusion, while this benchmark does not correspond to the typical cooperative applications we are targeting, it shows that the performance of Javanaise is within the same order of magnitude as standard Java-RMI and can even perform better for a general purpose workload, especially when objects are intensively reused. In the near future, we plan to study how caching

and remote invocation could be combined in order to use the most efficient mechanism according to the application structure.

7.3. Porting an existing application

In this section, we describe our experience on porting an existing centralized application to Javanaise in order to allow access to it from any remote machine.

The application is a graphical mail browser using a POP server to get the electronic mail. This application consists of 10700 lines of Java code. It provides traditional facilities such as folders for messages. When a user reads and sends messages, he may archive the messages in different folders and browse old messages in his folders. In the original centralized application, all the messages and the folders are stored in files on the site where the application is installed. This application is supposed to be launched only on that site.

Our goal is to allow remote access to this application from any machine connected to the Internet[9]. With today's technology, this can be achieved in two ways:

- redirecting the application display from the application site to the screen of the connection site (the site from which the user connects). This solution is not acceptable if the distance between the two sites is very long. The amount of messages sent to update the display on the connection site would slow down the application.

- run the application locally (at the connection site) and use the POP server to bring the new messages to the running application. Using this solution, it is not possible to browse or archive messages in folders which are stored on the application site. Moreover, it would require the application to be installed on the connection site.

More generally, systems and applications do not currently allow users to keep their working space available while they are on the move. Javanaise can be used to provide such facility[10].

By supporting this application on top of Javanaise, the application can be deployed to any machine on which a Java virtual machine runs, without any prior installation of the application. Furthermore, Javanaise allows remote access to the folder environment of the user, the user's folders being copied to the connection site on demand.

The main modification we made to this application concerns the storage of persistent data, i.e. the set of folders and the address book managed by the application. These data structures are kept persistent thanks to Javanaise persistence facilities, without requiring explicit saves/loads to/from files by the application. We just had to remove this function from the original code. The graphical interface of the application and its interface with the POP server were kept unchanged. The architecture of the application is described in Figure 10.

On the Home site of the application, an entry point of the application is registered in the Javanaise name server. This entry point is a Java reference to the Javanaise object which is the root of the application. From this root object, all the objects which compose the application are accessible. The code of the application is composed of the graphical interface of the application and two packages which respectively manage the address book of the user and his folders. The code of the application and the classes that compose the Javanaise runtime are mobile, i.e. they are dynamically deployed to the accessing node at execution time. The address book and the folders are Javanaise objects that are loaded from files (on the Home site) on demand and copied dynamically to the requesting node.

---

[9] Provided that the user is authenticated with a login password.

[10] Even if this application is not a real cooperative application, allowing remote access to a user's applications is a very important feature which has the same requirements as cooperative applications. Moreover, since Javanaise manages distributed shared objects, an user can launch his mail browser from home, without quitting the first instance of the application running on his office workstation.
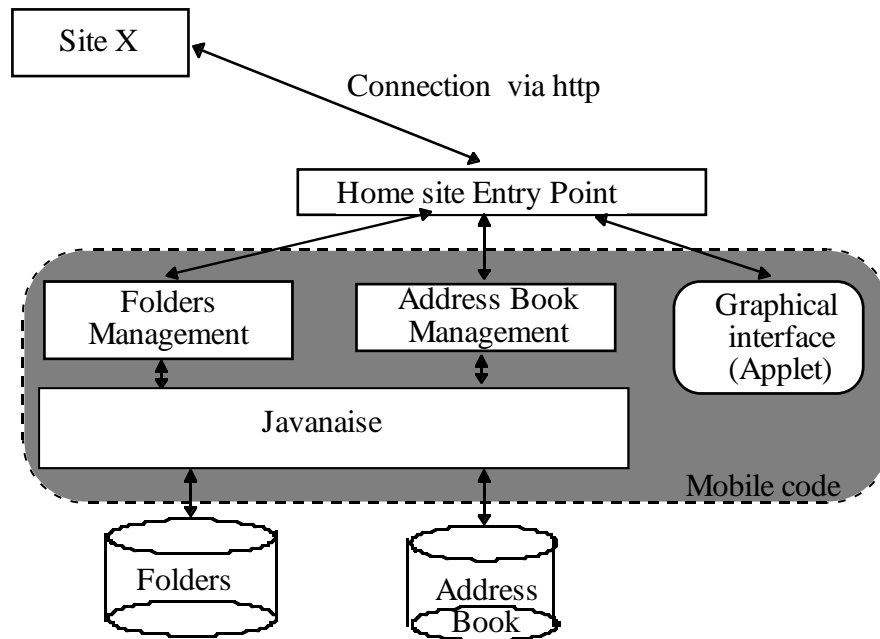
Figure 10: architecture of the mail application

In this application, a mail message is composed of two parts. The first contains the header of the message, including the sender, the date and the subject of the message. The second part contains the body of the message. A folder has a name and contains a set of messages. Folders and messages are Javanaise objects. A folder object contains references (inter-cluster references) to the message objects that implement the messages stored in this folder.

In order to be able to display a folder, including its message headers, without having to download all the messages, we had to adapt the application as follows. A message header is replicated; it is stored in the message it belongs to and also in the folder in which the message resides. Hence, when a folder is displayed, it is possible to show the headers of the messages included in this folder without having to download all the messages. The messages are effectively downloaded when the user chooses to display the individual message.

Except for the two modifications described above (relying on Javanaise persistence and replicating message headers in folders), the code of the application is kept unchanged. We just had to specify which classes are Javanaise clusters and to use our stub generator in order to allow clusters to be dynamically loaded on the accessing node.

To conclude this section, it has been very easy to port this application on top of Javanaise. It took actually two days to complete this port. We are currently continuing these experiments and working on porting a cooperative editor which will allow cooperation between multiple users at the same time.

8. Related work

In this section, we first relate our work to current projects which address the issues of supporting distributed applications on the Internet. Then, we compare Javanaise to older projects which can be seen as Javanaise's ancestors.

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.

Another approach for sharing Java objects on the Internet is the management of distributed shared tuple spaces. Several projects (JavaSpaces [Sun97], Jada [Ciancarini97]) proposed Java toolkits that implement this abstraction, inspired by the Linda language. Cooperation between distributed applications is achieved via tuple spaces. By exchanging tuples through tuple spaces, applications can exchange data or synchronize. Access to a tuple space is performed using a set of operations which allow tuples to be inserted, extracted or read from the tuple space. The drawback of this approach is that every cooperation with any remote application must be explicitly implemented using the set of tuple management primitives. We believe that application level objects should be implicitly shared between applications, as in the Corba (or RMI) philosophy.

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 promising 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.

The design of Javanaise was also influenced by the Perdis European project [Shapiro97] which aims at providing a distributed shared memory (DSM) based system support for cooperative applications (in the area of the building industry) on large scale networks. However, with a DSM, object sharing between heterogeneous sites with dynamic deployment of applications is much more difficult and it is not addressed in the framework of the Perdis project.

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 as Guide, i.e. a distributed shared object space, but it differs by the management of object granularity. 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.

9.   Conclusion and Perspectives

In this paper, we have 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 on 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 copied on demand to the client nodes and shared following a specified consistency and synchronization protocol.

We have implemented a prototype of the Javanaise runtime, composed of a preprocessor which generates the 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 has been evaluated by comparison with RMI using a benchmark and validated by porting an existing mail browser application.

A first perspective of this work is to continue the evaluation of our service through the support of full scale cooperative applications. We have identified several applications that we plan to port on top of Javanaise. Otherwise, we would like to investigate the integration of other mechanisms within Javanaise.

In particular, we would like to:

- combine the invocation of clusters replicas locally and the invocation of clusters remotely using an RMI-like mechanism. The choice between these two mechanisms should be based on the semantic of the cluster objects, the amount of write sharing, or the amount of references to that cluster within the application.

- allow disconnected operation of applications on top of Javanaise. Javanaise should be extended in order to tolerate non-coherent replicas and to reconcile them when application instances reconnect.

Bibliography

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

[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.

[Cattell92] R. G. G. Cattell and J. Skeen, "Object Operation Benchmark", *ACM Transactions on Database Systems*, 17(1), March 1992.

[Ciancarini97] P. Ciancarini, D. Rossi, "Jada: A Coordination Toolkit for Java", Technical Report C, Department of Computer Science, University of Bologna, Italy, 1997.

[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.

[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.

[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.

[Shapiro97] M. Shapiro, S. Kloosterman, F. Riccardi, "PerDiS - a Persistent Distributed Store for Cooperative Applications", Proceedings of the 3rd Cabernet Plenary Workshop, IRISA Rennes, France, April 1997.

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

[Sun97] Sun Microsystems, "JavaSpace Specification", Sun Microsystems.
URL: http://chatsubo.javasoft.com/javaspaces/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.