A Configurable RMI Mechanism for Sharing Distributed Java Objects

Javanaise is a remote method invocation mechanism that extends the functionality of Java RMI by allowing dynamic caching of shared objects on the accessing nodes.

Daniel Hagimont and Fabienne Boyer
Sirac Laboratory (INPG-INRIA-UJF)

Many distributed programming environments have been designed to support distributed shared objects over the Internet. Most of these environments—Java RMI[1] and CORBA,[2] for example—support client-server applications where distributed objects reside on servers, which execute all methods (remote or local) invoked on the objects. Traditional client-server models do not support client-side object caching and the local access it provides.

We believe that object caching is critical to distributed applications, especially over the Internet, where latency and bandwidth are highly variable. We have developed a configurable and efficient remote method invocation mechanism that provides the same interface as Java–RMI, while extending its functionality so that shared objects can be cached on the accessing nodes. The mechanism, called Javanaise, is based on the caching of clusters, which are groups of interdependent Java objects. We have implemented a prototype consisting of a preprocessor that generates the required proxy classes from the application interfaces and a runtime environment that uses system classes to manage the consistency of cluster replicas cached on client nodes.

In this article, we describe the motivation for our work, the design choices we made for the Javanaise clustering mechanism, the implementation principles for managing Javanaise clusters, and the results from three experiments that compare the performance of Javanaise with Java RMI. The sidebar on page 39 presents "Related Work in Distributed Shared Objects."

Performance Issues

Performance considerations related to the current client-server distribution scheme can limit the benefits of object-oriented application development on the Internet, for example, by leading to the definition of methods that return simple values rather than object references. Consider the case of a directory service that gives access to information about professional services through a get_info() method. Object orientation implies that the get_info() method should return an object reference to information, such as costs

 1089-7801/00/$10.00

and booking information. Especially when the returned data types are complex, specific methods should also allow simple manipulation of the data. However, returning an object reference requires an application to process significantly more remote invocations (each time a client accesses the data), which may saturate the application directory service. Adding more servers to support the directory service may not be easy and in any case would not reduce network traffic.

This situation is summarized in Figure 1. In case 1, the programmer has forced the get_info() method to return values in order to reduce message exchanges, sacrificing object orientation. In case 2, the get_info() method returns an object reference, but also increases the message exchanges and server workload. Even CORBA's object–by–value feature, which allows objects to be passed as parameter values of method invocations, does not resolve this problem because write operations cannot be performed on the client side, which can be useful when a single user has exclusive access to an object during a certain period of time.

We believe that the development of scalable Internet services would benefit from a distribution scheme that supports client- and server-side method invocations, depending on object behavior. Javanaise offers a configurable RMI facility and supports both traditional client-server distribution and a distribution scheme based on object caching. Distribution scheme selection for each program object can be made after application development, at configuration time.

In previous experiments we showed that caching distributed, fine-grained objects can be inefficient because each cached object must exchange a message with the server.[3] The cluster–based caching mechanism proposed in Javanaise is tightly coupled with the data structures managed by the application. This ensures locality yet keeps clustering transparent to the application programmer.

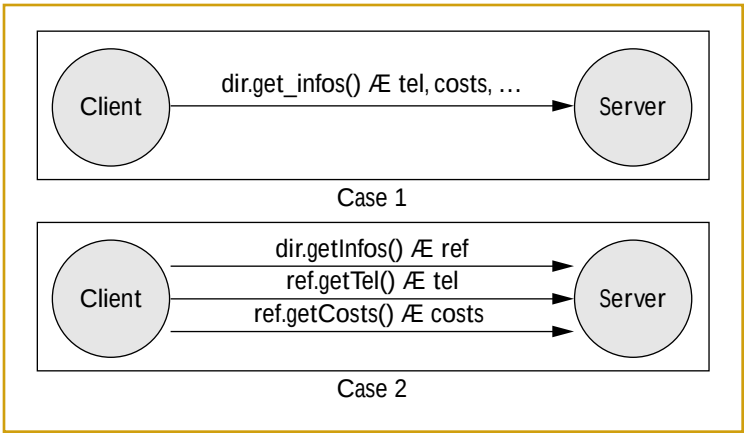

Figure 1. Programming remote method invocations. In case 1, the program invokes simple values, while in case 2 it invokes object references, but at a performance cost.

Design for Cluster Management

Three requirements drove the design choices we made to support cluster management in Javanaise:

- *Accuracy.* Objects within a cluster should be closely related. System services such as binding, naming, and consistency checking are applied to clusters and not to individual objects. Clusters should be composed of interdependent objects (objects likely to be used within the same time interval) to realize savings in system services.
- *Transparency*. Managing cluster replicas requires mechanisms for faulting, invalidating and updating clusters in order to ensure consistency. These mechanisms should be hidden from the application programmer, who should manipulate Java references as if every object were local.
- *Configurability*. Transparency does not imply that cluster management is independent of the application semantic. The programmer should apply different clustering protocols (for example consistency or persistence) in order to tune cluster management.

"This is a Space for a quote"

We designed Javanaise to use *application-dependent clustering*, based on the observation that object–oriented applications tend to manage logical graphs of objects in their data structures. The runtime environment should be able to manage some of these graphs as clusters that correspond to closely related objects according to the application semantics.

A Javanaise cluster is identified by a Java reference to a root object, called a *cluster object*, and by the graph composed of all the Java objects that are accessible from the cluster object (the *transitive closure*). Figure 2 shows the boundaries of this graph, defined by the leaves of the graph and by its *inter-cluster references* (references to other cluster objects). Java objects within a cluster are called *local objects*.

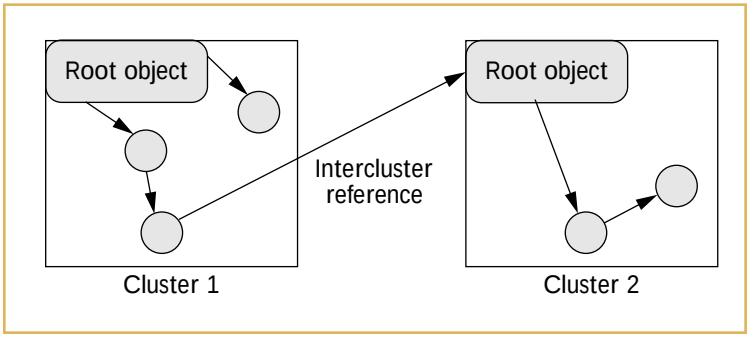

Figure 2. Cluster management. Cluster 1 is defined by its local graph and its intercluster references, which in this case include reference to Cluster 2.

Remote method invocation, as deployed in Java RMI by Wollrath et al.,[4] implicitly enforces application-dependent clustering by requiring reference parameters in the interface of a *remote class* to be references to *remote objects*. An RMI remote object is equivalent to a Javanaise cluster object, and the remote interface of RMI is the basis for our cluster definition. Local objects are accessible only to objects within the cluster. The cost of manipulating these globally visible cluster objects determines application performance. In addition, Javanaise cluster objects are cacheable units and can migrate from one node to another.

"This is a Space for a quote"

Cluster Configuration

An application can be developed in a centralized way, then configured for distribution. The programmer configures clusters by choosing whether to implement the following extensions to the RMI remote interface:

- CacheableCluster (default is noncacheable)
- PersistentCluster (default is nonpersistent)

Figure 3 defines Cluster1 to be both cacheable and persistent. Each cacheable cluster should be associated with a consistency protocol. Javanaise provides several interfaces with different consistency protocols. In the single-writer/multiple-readers protocol (SingleWMultipleRProtocol), exceptions are used to associate a locking policy (reader or writer) with each method.

For example, a method that throws `JAVANAISE_READ_EXCEPTION` requires a lock for entry. When the method is invoked on a cluster instance, the local host acquires a lock and receives a consistent copy of the cluster. Different versions of this protocol allow the cluster to be made consistent either with invalidation-on-write or with update broadcasting. The `SingleWSingleRProtocol` interface corresponds to an exclusive-access policy. We plan to experiment with other consistency/synchronization protocols in the future, according to application-specific requirements.

Javanaise Implementation principles

We implemented a prototype to demonstrate the management of cacheable clusters. The prototype consisted of two main parts: a *preprocessor*, which analyzes the cluster interfaces and generates the appropriate proxy classes, and a *runtime system*, which includes the set of classes used during execution.

Managing Cluster Binding and Consistency

Javanaise manages the binding of dynamically fetched objects from remote nodes. Figure 4 shows the *proxy–in* and *proxy–out* intermediate objects that the pre-processor transparently inserts in the application to bind two clusters (as conceptually described in Shapiro[5]). The proxy objects manage (marshal/unmarshal) onward and backward parameters during intercluster invocations. Proxy–out objects locate the referenced cluster and enable access through a local name. Proxy–in objects

```
public class Cluster1 implements Cluster1_itf,
    CacheableCluster,
    PersistentCluster,
    SingleWMultipleRProtocol {
   public void method1 (Cluster2_itf obj) throws JAVANAISE_READ_EXCEPTION;
   public Cluster3_itf method2 () throws JAVANAISE_WRITE_EXCEPTION;
}
```

Figure 3. Code for a Javanaise cluster. The cluster is defined to be cacheable and persistent and to maintain consistency using the SingleWMultipleRProtocol protocol.

Related Work on Object Caching in Distributed Systems

Wollrath et al.[1] proposed a remote method invocation facility between Java objects called Java RMI. It supports distributed shared objects on the Internet but does not allow objects to be cached and therefore accessed locally. It is possible to manage object replicas using the object serialization facility in Java RMI, but the coherence between the replicas has to be explicitly managed by the application programmer.

Some projects have addressed the problem of object caching, but the proposed solutions are not targeted to the Java environment (for example, see Topol et al.[2]). The Hybrid Adaptive Cache[3] and Thor[4] projects at MIT address the problem of managing client caches in distributed and persistent object storage systems. The objective is to provide hybrid and adaptive caching, which manages both page caches and object caches, according to an object's behavior. Other environments, such as the ObjectStore database management system,[5] use object caching to meet scalability requirements. Javanaise is distinguished from these environments by providing an RMI–based solution that relies on cluster-caching for general-purpose applications.

Krishnaswamy et al.[6] propose an efficient implementation of Java RMI, extending it to benefit from both object caching and the UDP communication protocol (which is faster than TCP). We share the same objectives regarding object caching; however, their extension appears to be deeply integrated within the JDK1.1.5. Thus, their solutions assume the widespread use of this modified environment. Javanaise, on the other hand, is built entirely on a standard Java environment; all its components can be dynamically loaded with application code.

Chockler et al.[7] propose a scalable caching service for Corba objects, based on a hierarchical cache architecture. The service uses domain caching servers to cache objects as close as possible to clients. This contrasts with Javanaise, which allows an object to be cached on the client side.

Some other CORBA platforms provide adaptation features, but do not yet directly address the problem of object caching (for example, see Blair et al.[8] and Singhai et al.[9]). The OpenCorba project[10] aims at providing an adaptable object broker that can reify an object's internal mechanisms and then adapt them at runtime. Allowing object behavior to be dynamically modified at runtime in turn allows different strategies to be used for dynamic placement of objects. OpenCorba can thus be considered as a useful support for the provision of an adaptable RMI.

The Coign project[11] addresses the idea of minimizing communication costs for a distributed application; it uses scenario-based profiling to gather statistics that support automatic and dynamic partitioning of software components in a distributed environment. Scenario-based profiling might offer a way to extend Javanaise so that it could make transparent application-specific decisions about whether or not to cache a given cluster of Java objects; this decision is currently made by the programmer. Mobile-agent–based programming is another emerging paradigm for structuring distributed applications over the Internet.[12] An agent is, roughly, a process with its own context, including code and data, that may travel among several sites to perform its task. It can generally access objects that are exported either by the servers it visits or by other agents running on these servers. We do not think that Javanaise addresses the same kind of applications that a mobile agent does, although one common objective is to increase access locality. Javanaise allows server objects to move closer to the client, while mobile agents allow client objects to move closer to the servers.

References

1. A. Wollrath, R. Riggs, and J. Waldo, "A Distributed Object Model for the Java System," *Computing Systems*, vol. 9, no. 4, Fall 1996, pp. 291–312.
2. B. Topol, M. Ahamad, and J. Stasko, "Robust State Sharing for Wide Area Distributed Applications," *Proc. Int'l Conf. Distributed Computing* (ICDCS), IEEE Computer Society, Los Alamitos, Calif., 1998, pp. 554-561.
3. M. Castro et al., "HAC: Hybrid Adaptive Caching for Distributed Storage System," *Proc. 16th ACM Symp. Operating Systems Principles* (SOSP), ACM Press, New York, 1997, pp. 102-115.
4. B. Liskov et al., "Providing Persistent Objects in Distributed Systems," *Lecture Notes in Computer Science*, Springer-Verlag, Heidelberg, vol. 1628, June 1999, pp. 230-257.
5. Excelon Corp., "ObjectStore 6.0," whitepaper; available online at http://www.odi.com/objectstore/Whitepapers/objectstore.htm.
6. V. Krishnaswamy et al., "Efficient Implementations of Java Remote Method Invocation (RMI)," *Proc. Usenix Conf. on Object Oriented Technology and Systems* (COOTS), Usenix Assn., Berkeley, Calif., 1998, pp. 1-23.
7. G.Chockler et al., "Implementing Caching Service for Distributed CORBA Objects," *Proc. IFIP/ACM Int'l Conf. Distributed Systems Platforms and Open Distributed Processing* (Middleware'2000), *Lecture Notes in Computer Science*, Springer-Verlag, Heidelberg, vol. 1795, Apr. 2000, pp. 1-23.
8. G.S. Blair et al., "An Architecture for Next-Generation Middleware," *Proc. Middleware98*, Springer-Verlag, 1998, pp. 191–206.
9. A. Singhai, A. Sane, and R. Campbell, "Reflective ORBs: Supporting Robust, Time-critical Distribution," *Proc. Workshop Reflective Real-Time Object-Oriented Systems* (ECOOP), *Lecture Notes in Computer Science*, Springer-Verlag, Heidelberg, vol. 1357, June 1997, pp. 55-61.
10. T. Ledoux, "OpenCorba: a Reflective Open Broker," *Proc. Reflection99*, *Lecture Notes in Computer Science*, Springer–Verlag, Heidelberg, vol.1616, July 1999, pp. 197–214.
11. G.C. Hunt and M.L. Scott, "The Coign Automatic Distributed Partitioning System", *Proc. Third Symp. Operating Systems Design and Implementation* (OSDI), ACM Press, New York, Feb. 1999, pp. 187-200.
12. D. Chess, C. Harrison and A. Kershenbaum, "Mobile Agents: Are They a Good Idea?" IBM Research Report, RC 19887, Dec. 1994.

ensure cluster consistency. Each proxy object includes a global name, which allows a cluster to be located when a copy has to be fetched from the local host.

proxy-in and proxy-outobjects. A proxy–in object belonging to a cluster *C* manages *C*'s consistency and invalidates or updates the local copy of *C* (based on its configuration properties). The system

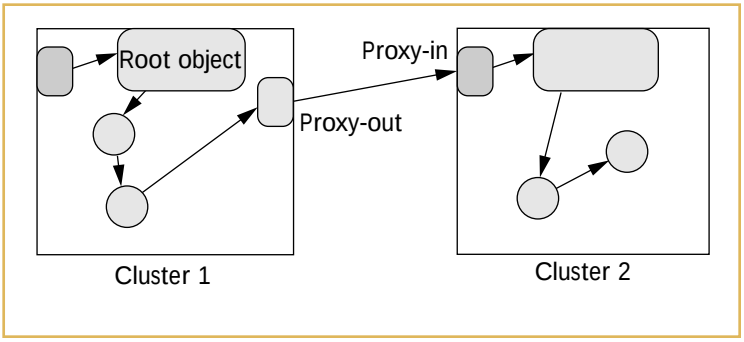

Figure 4. Proxy–in and proxy–out objects. Cluster1 includes a proxy-out object that locates and enables access to Cluster2. The Cluster2 proxy-in ensures Cluster2's consistency.

should ensure that there is one proxy–in object per cached cluster on a given Java VM. This object includes a Java reference to the local copy of the cluster. It implements the same interface as the cluster object, and forwards method invocations to the local copy of the cluster. If a proxy–in fault occurs (its Java reference is null), the runtime system fetches a consistent copy of the cluster.

A proxy–out object includes a Java reference to the proxy–in of the target cluster object (if one exists on the current node). A proxy–out object also implements the same interface as the cluster object and forwards method invocations to the proxy-in pointed to by the Java reference.

"This is a Space for a quote"

If a proxy–out fault occurs (the Java reference is null), the runtime system checks whether the cluster is already cached (due to the binding of another intercluster reference) and fetches a copy of the cluster as needed. The interfaces implemented by the target cluster class are used by the preprocessor to generate the proxy–out and proxy–in object classes.

Cluster consistency. The proxy-in object of a cluster *C* on a given node *N* manages cluster consistency by either invalidating or updating *C* according to the applicable consistency protocol. Invalidation consists of assigning the proxy-in Java reference to null. Updating consists of assigning the reference to a new cluster copy. Clusters may be copied dynamically to a requesting node using the Java serialization mechanism. In either case, assigning the proxy-in Java reference implies the destruction of all the Java objects included in the previous copy of the cluster.

Most consistency protocols require lock management. The locking of a cluster *C* is managed by the proxy-in objects associated with *C* through a master-slave protocol. The master proxy-in of a cluster *C* is created on the node where *C* is created or first loaded from persistent storage. A slave proxy–in object is created when *C* is replicated on another node. The master is invoked each time a concurrency management operation has to be performed. Concurrency management may require invoking operations on the slaves as well (for example, to invalidate replicas).

While the master role may move from one node to another (for example, to support persistence as discussed below), the global name of the cluster allows the master proxy-in to be located. The master knows the locations of all cluster copies (addresses of slave proxy-in objects). In the SingleWMultipleRProtocol, the master knows the list of nodes that obtained a read lock on the cluster (the slaves) if the cluster is locked in read mode. If the cluster is in write mode, it knows the address of the node that hosts the unique copy.

The master handles requests for copies (in read mode) or for the unique copy of the cluster (in write mode). When the cluster is already locked in a conflicting mode, the master sends requests to the slaves (in either read or write mode). The slaves reply when their locks are released.

Managing Reference Parameter Passing

When passed as a parameter of a cluster invocation, a cluster reference must point to a proxy-outobject. Any method invocation from a cluster *C*1 to a cluster *C*2 is intercepted first by the proxy-outof *C*2 in *C*1, and second by the proxy-in of *C*2. A proxy-outmanages any backward-reference parameter passing, while a proxy-in manages onward-reference parameter passing. Either proxy may create a local proxy-outfor a received reference.

All the references to a cluster C_i within a cluster *C* should point to the same proxy–out object. When a reference enters a cluster that has a proxy–out object associated with the referring cluster, the existing object is used. The proxy–in and proxy–out objects know all the existing proxy–out objects of a given cluster.

In Figure 5, a local object in cluster *C*1 performs an invocation (c3 = c2.meth(c1)) on cluster *C*2 (step 1). The invoked method transfers a *C*1 reference to cluster *C*2. The proxy-in in *C*2 creates a proxy-out that refers to *C*1 (step 2). The invoked method also

returns a Java reference to cluster *C*3 (step 3). To store a reference to *C*3 in *C*1, the system creates a proxy-out in *C*1 which refers to *C*3 (step 4). This proxy-out is created by the proxy-out that intercepted the original method call in C1.

Managing Cluster Persistence

Both cacheable and noncacheable clusters can be persistent (that is, they can survive the application that created them). Javanaise provides persistent cluster identifiers along with a location scheme. In the current prototype, persistent global names identify clusters and include fixed and variable parts. The fixed part consists of the IP address of the cluster's storage site, plus a unique identifier. The variable part contains the IP address of the master proxy–in, facilitating location of the cluster if mastership has migrated. After a mastership migration, the variable part of a persistent identifier may become obsolete. In that case the location mechanism queries the cluster storage site, which is always aware of the master location.

Evaluation and Results

We conducted a three-part evaluation of Javanaise. First, we measured the costs of basic operations to better understand the system's behavior. Second, we implemented the traversal portion of the OO1 benchmark,[6] which roughly consists of a traversal of a distributed graph, to compare the benefits cacheable clusters provide over Java RMI. Finally, we ported an existing distributed application to Javanaise to demonstrate the adequacy of the platform with a real application.

The experiments were performed on a pair of PC desktops based on PentiumII processors (400 MHz) running Windows NT4 and the JDK 1.2.2 version of Java VM. These machines were connected through a 100-Mbit Ethernet. All measurements were performed on an isolated network and repeated 10 times; the reported times are the averages of these 10 measurements.

Basic Operations

Table 1 compares the basic costs of Java invocation mechanisms (local and remote) with those of Javanaise. The measurements were made with a small cluster size (75 bytes) iterating on method invocations and dividing the global time by the iteration number.

Row 3 gives the cost of a Javanaise method invocation for a remote cluster not yet cached locally (cold invocation). This operation involves fetching a copy of the cluster from the remote site, installing it on the local host, and invoking the method of the cached copy. Although a Javanaise cold invocation is more expensive than the same cluster invocation with RMI (row 2), row 4 shows that the cost is justified if the same cluster is invoked more than once. While the cost of a Java RMI invocation remains the same, the cost of a Javanaise hot invocation (subsequent invocation) is only 0.077 microseconds. The higher cost of a Javanaise hot invocation (row 4) compared with that of a Java method invocation (row 1) results from the traversal of our proxies.

Figure 6 gives the cost of a Javanaise cold method invocation for different object (cluster) sizes. Because this cost is highly dependent on the cost of object serialization, we measured the objects both as an array of bytes and as a binary tree of small Java objects (the overall tree being the same size). The results indicate that a better implementation of object serialization could greatly improve Javanaise performance.

OO1 Benchmark

The traversal portion of the OO1 benchmark (used to evaluate database systems) implements a tra-

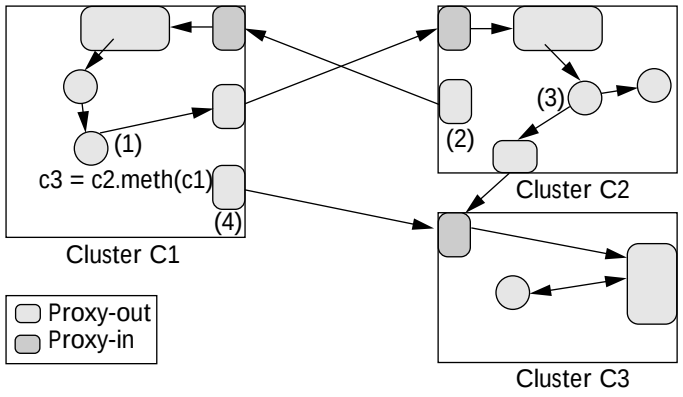

Figure 5. Parameter passing. A proxy-out object includes a Java reference to the proxy-in of the target cluster *C*.

Table 1. Comparative costs for a basic method invocation.

Invocation mechanism	Time in µsec	Variance
Java local method invocation (on an interface)	0.023	0.00013
Java remote method invocation (RMI)	1016.4	11.09
Javanaise remote method invocation (cold)	1313.5	6.02
Javanaise remote method invocation (hot)	0.077	0.00021

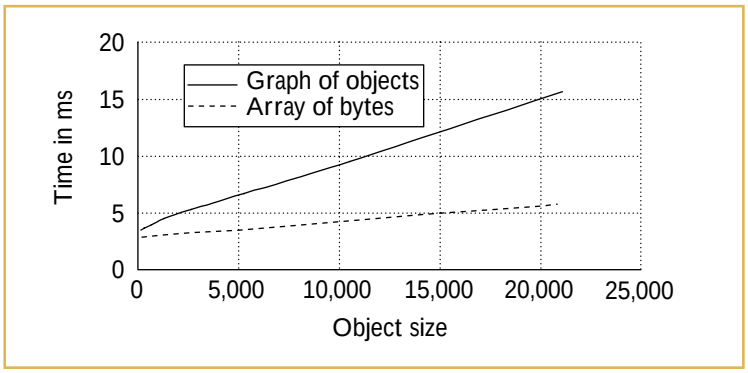

Figure 6. Javanaise method invocation costs for different object sizes. The higher costs for object graphs than for byte arrays indicate that object structure (as well as size) affects application performance.

Table 2. Performance comparison for the OO1 benchmark (objects size = 75 bytes).

Invocation mechanism		Time in ms	Variance
Java RMI	1569	24	
Javanaise (cold)	4465	107.8	
Javanaise (hot)	2.4	0.57	

versal of a 5,000-node graph in which each node has three children. The traversal is done over seven levels (a total of 3,280 nodes are visited). The nodes are equally distributed on two sites. The children nodes are randomly chosen with a probability of 70 percent that a parent and child are on the same machine. (Note that Javanaise would perform better if this probability were smaller.)

We implemented this benchmark with noncacheable clusters (Java RMI) and cacheable clusters (Javanaise). We did not change the source code, but processed the same code with the Java RMI and Javanaise stub generators. In both cases, the nodes were created on two sites. With Java RMI, a reference to a remote object implies a remote method invocation to that site. With Javanaise cacheable clusters, a reference to a remote object implies that a copy of the object is brought to the local host and the method is invoked locally.

Table 2 presents the results. With a cold start, Javanaise performs three times slower than RMI. This inefficiency has two main causes:

- The object graph has a high percentage of interobject references that are local to one site (70 percent). This implies that when a remote object (on site S2) is brought to the accessing site S1, its children on S2 become remote and will have to be fetched. With Java RMI, when the remote object is invoked (on S2), its children on S2 are local and their invocations are very efficient.
- We observed that some nodes of the graph were visited several times—one of the key conditions to Javanaise efficiency. In the Javanaise cold experiment reported in Table 2, we found that objects were reused 1.54 times on average. We varied the traversal depth of the graph to measure the advantage gained by increasing reuse of objects, shown in Figure 7. For example, at a depth of 9 levels, objects are reused about 6 times on average.

The results show that Javanaise can perform better than RMI for the first (cold) travel in the graph depending on the amount of object reuse.

In conclusion, this benchmark shows that Javanaise performs within the same order of magnitude as standard Java RMI and can perform better for a general-purpose workload, especially when objects are intensively reused. Our plans for future work include the study of how caching and remote invocation could be combined to take advantage of efficiencies related to application structure.

Adapting an Existing Application

In our third experiment, we adapted a Java RMI-based distributed application to run on Javanaise. The application is a graphical mail browser that uses a POP server to get electronic mail. It consists of 10,700 lines of Java code and provides traditional facilities such as message folders. When users read or send messages they can archive the messages in different folders for access later. In the original application, all messages and folders are accessible as Java remote objects.

First we had to choose which remote objects should be cacheable clusters. In this application, a message is composed of two parts: the first contains the header of the message, including the sender, the date and a subject; the second contains the body of the message. A folder contains references to a set of messages and copies of the associated headers.

The potential number of remote accesses to be performed on clusters affected our configuration decisions. As folders and messages are not subject to concurrent-write sharing, we decided to configure them as cacheable clusters, as shown in Figure 8 . The messages are downloaded on demand when the user chooses to display messages. We could

manage a folder and its messages in a single cluster, but this would require downloading all the messages stored in the cluster when one message is remotely accessed.

The application's code is dynamically deployed to the accessing node at execution time. A Web server makes a Javanaise application available as an applet that a client starts using an applet viewer. The Javanaise name server registers an entry point for each application user. This entry point is a Java reference to the application's root object. All the objects that compose the application are accessible from this root object, including user preferences, folders and messages.

This experiment let us evaluate the adequacy of Javanaise for the support of an existing application, as only minor modifications to the source code were required (the functional code stayed unchanged). It also demonstrated performance improvement due to object caching versus Java RMI's traditional client-server paradigm. Though a client may access messages and folders only once, most of these data structures must still be copied to the accessing node for display and possible modification. The implementation of the mail application in Java RMI requires these data to be fetched explicitly and therefore requires at least the same number of data transfers.

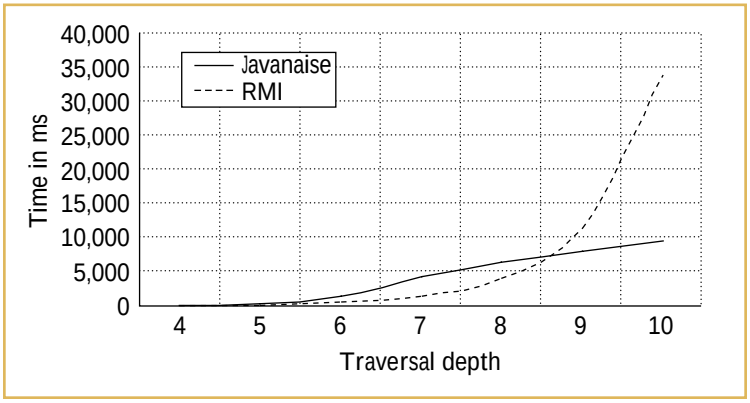

Figure 7. Test results for the OO1 benchmark. Javanaise runs the OO1 benchmark faster than RMI for a traversal depth greater than 8.5.

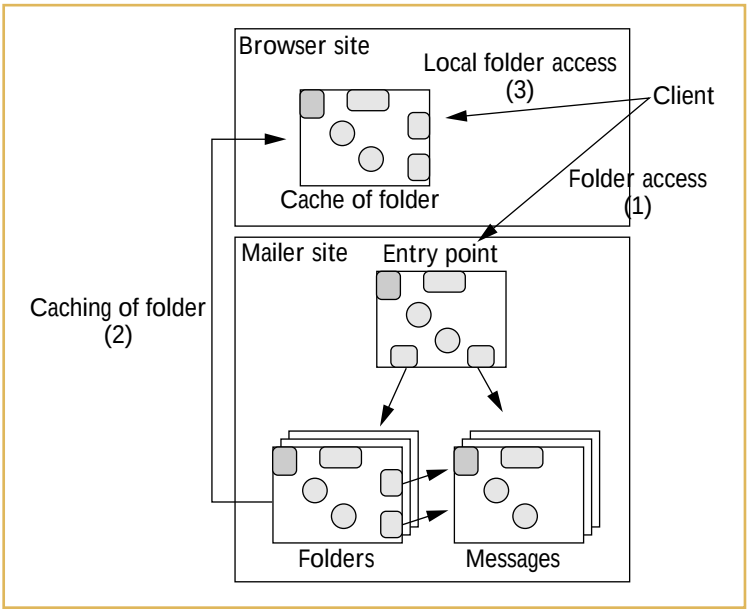

Figure 8. Architecture of the mail application. Folders and messages are configured as cacheable clusters. One folder has been cached on the browser site.

Conclusion and Future Work

Javanaise allows a programmer to develop applications as if they were to be executed in a centralized configuration and to configure them for a distributed setting with only minor modifications to the source code. The programmer can tune an application, relying on either client-server invocations or cluster caching, therefore obtaining better performance as demonstrated in our evaluation.

We hope to improve performance further in future work. For example, Bruneton and Riveill[7] describe a reflective tuning mechanism that could allow dynamic cluster configuration. Configuring cachability dynamically based on the number of remote invocations from a given site would improve performance. A better implementation of object serialization could also help.

We would also like to extend Javanaise to tolerate noncoherent replicas and to reconcile them when application instances reconnect. This would provide support for distributed applications that involve mobile users.

Finally, we plan to experiment with different consistency/synchronization protocols according to application-specific requirements.

References

1. Sun Microsystems, "Java Remote Method Invocation (RMI)," homepage, http://java.sun.com/products/jdk/rmi/.
2. Object Management Group, *The Common Object Request Broker: Architecture and Specification*, Rev. 2.4, OMG formal doc. 2000/11/03; http://www.omg.org/technology/documents/formal/corbaiiop.htm.
3. D. Hagimont et al., "Persistent Shared Object Support in the Guide System: Evaluation and Related Work," *Proc. 9th ACM Conf. Object–Oriented Programming, Systems, Languages and Applications* (OOPSLA), 1994, pp. 129-144.
4. A. Wollrath, R. Riggs, J. Waldo, "A Distributed Object Model for the Java System", *Computing Systems*, 9(4), pp. 291–312, Fall 1996.

5. M. Shapiro. “Structure and Encapsulation in Distributed Systems: The Proxy Principle,” *Proc. Sixth Int’l Conf. Distributed Computing Systems* (ICDCS), IEEE Computer Society, Los Alamitos, Calif., 1986, pp.198– 204.
6. R.G. Cattell and J. Skeen, “Object Operation Benchmark,” *ACM Trans. Database Systems*, vol. 17, no. 1, Mar. 1992, pp. 1-31.
7. E. Bruneton and M. Riveill, “JavaPods: An Adaptable and Extensible Component Platform,” INRIA Research Report 3850, Jan 2000.

Daniel Hagimont is a research scientist at INRIA Rhône-Alpes (Grenoble, France) and a member of the Sirac project, where he leads a group working on distributed systems and applications. He was one of the main designers of the Guide distributed system at Bull-IMAG, and he has worked in the area of operating system support for distributed objects, distributed shared memory, and protection. Hagimont received his PhD from Institut National Polytechnique de Grenoble in 1993.

Fabienne Boyer is a research assistant at Université Joseph Fourier (Grenoble, France) and a member of the Sirac project, where she works more specifically on adaptable distributed systems and the use of reflective features to achieve this goal.. She was one of the main designers of the integrated development environment of the Guide object-oriented distributed system (Bull-IMAG). Boyer received her PhD from Université Joseph Fourier in 1993.

Readers may contact the authors via e-mail at {Daniel.Hagimont, Fabienne.Boyer}@inrialpes.fr.