Efficient Object Caching for Distributed Java RMI Applications*

John Eberhard** and Anand Tripathi

Department of Computer Science, University of Minnesota,
Minneapolis, MN 55431
{eberhard, tripathi}@cs.umn.edu

Abstract. Java-based[1] distributed applications generally use RMI (Remote Method Invocation) for accessing remote objects. When used in a wide-area environment, the performance of such applications can be poor because of the high latency of RMI. This latency can be reduced by caching objects at the client node. However, the use of caching introduces other issues, including the expense of caching the object as well as the expense of managing the consistency of the object. This paper presents a middleware for object caching in Java RMI-based distributed applications. The mechanisms used by the middleware are fully compatible with Java RMI and are transparent to the clients. Using this middleware, the system designer can select the caching strategy and consistency protocol most appropriate for the application. The paper illustrates the benefits of using these mechanisms to improve the performance of RMI applications.

1 Introduction

Distributed object systems are commonly implemented using some type of remote procedure call, or in the Java case, remote method invocation (RMI) [20]. One problem with using RMI is the high latency associated with invoking a method on an object. This problem is especially severe in wide-area networks, due to large latencies dictated by the distances between sites. To overcome the latency inherent in remote method invocation, the programmer is forced to structure applications so that interactions are reduced, or the programmer must rely on other schemes to reduce latency related problems.

One approach to overcome the latency problem is to cache objects at the client's node. For example, a programmer could structure a client program so that a copy of the object resides at the client. When manually replicating the object in this manner, the programmer must decide the granularity of caching. The programmer must also ensure that the replicated copy of an object remains consistent with its other copies. Obviously, programatically adding caching in

* This work was supported by NSF ITR 0082215.

** Author is currently employed by IBM Rochester, Rochester, MN 55901

[1] Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

R. Guerraoui (Ed.): Middleware 2001, LNCS 2218, pp. 15–35, 2001.

this manner is expensive and error-prone. A better solution is the use of a middleware to transparently add caching to an RMI application. This middleware should permit the programmer to choose the caching and consistency protocols best suited to the application.

A middleware that improves the performance of Java RMI through the use of caching must meet certain requirements. First, for caching to be useful, it must be compatible with RMI and transparent to clients currently using RMI. Second, object caching mechanisms must support consistency protocols tailored to both the semantics and usage of an object. Third, when using caching, only the data needed by the client should be cached. One technique to do this is to selectively choose which objects should be cached. Another technique is to cache an object that contains only a portion of the data of the original object. To accomplish this, we introduce the concept of "reduced object."

Based on these requirements, we have designed a set of object caching mechanisms, integrated into a middleware system and a set of tools, for adding caching to any existing Java RMI application. This paper describes the following contributions of our work.

- A set of caching mechanisms transparent to and compatible with the existing RMI clients of an application.
- These mechanisms support integration of different kinds of consistency protocols by the server.
- These mechanisms retain semantics of Java's *wait/notify* synchronization mechanisms.
- A set of tools to create the objects necessary to support caching in Java RMI. These tools create the Java classes necessary to cache an RMI object.
- The notion of *reduced object* is introduced to support caching only parts of an object as needed by a client, to reduce the overheads in caching the full object.
- We present the results of our experimental evaluation of this system using two different kinds of applications, which demonstrate the benefits of these mechanisms for RMI applications.

As shown in this paper, adding caching to an RMI application can be accomplished using a middleware system that easily permits caching and consistency policies to be chosen for RMI objects.

The remainder of the paper is organized as follows. Section 2 presents related work. Section 3 presents the requirements guiding the development of the RMI caching middleware. Section 4 presents the middleware and associated caching mechanisms. Section 5 illustrates the benefit of this middleware using two different RMI applications. Section 6 presents the conclusions.

2 Related Work

Several researchers have investigated caching to improve the performance of RMI and Java distributed object systems. Other researchers have used caching

in distributed object systems. In comparison to existing work, our mechanisms retain RMI compatibility, permitting existing RMI clients to transparently use caching. Our mechanisms permit different consistency protocols to be associated with different objects. We also support reduced objects, which contain a subset of the state of the original objects. Below, we briefly review the work most relevant to this paper.

In their work on RMI performance, Krishnaswamy, et al., [14] describe a caching system used to improve RMI. Their system differs from our work because they cache the serialized version of the object inside the reference layer of the Java RMI implementation. All interactions with the object take place on the serialized object. A consistency framework assures that the entire serialized object remains consistent with the object on the server. Like their approach, our system retains RMI compatibility. Unlike their approach, our system caches the objects in their unserialized forms. Furthermore, rather than caching complete objects, our system permits caching portions of objects.

In a more recent work, Krishnaswamy, et al., [13] developed object caching for distributed Java applications. Their work is similar to our work in that they permit a variety of consistency protocols to be used to manage cached objects. Their work is concerned with using the quality of service specified by the client to guide caching decisions. While we take a server centric approach, the client can still provide information about its environment. However, unlike our work, they do not examine the impact of caching portions of objects.

Other researchers[3] have implemented Java caching using distributed shared memory. These approaches have required changes to the underlying Java virtual machine and would not be appropriate in a heterogeneous wide-area environment, nor would they be usable by existing Java RMI clients. Unlike their approach, our work does not require changes to the Java virtual machine.

Lipkind, et at., [16], describe a Java distributed object system with caching. Their architecture is based on "object views", where the programmer explicitly states how an object is to be used. The information from the object views is used to optimize the behavior of a distributed shared memory system running on a cluster of workstations. The work presented in this paper differs from their work in that we use RMI for client-server communication, thus permitting its use in a wide-area network.

Another distributed object system is Globe[4][22]. Like the system presented in this paper, several objects are used to represent a cached distributed object. These objects serve roughly the same purpose as the objects in the caching architecture presented in this paper, i.e., a local cache object, a consistency object, and a communications object. However, with their local cache object, they currently do not have a mechanism to use a subset of the instance variables of the classes. The use of reduced objects that contain only a subset of the instance variables of an object is a contribution of our work.

Rover is a distributed object system that caches objects. Its primary goal is support for disconnected operation. It has a single consistency policy that uses optimistic concurrency control to execute methods on a cached object. Requests for method execution are queued for eventual execution at the server. Like most

object systems, Rover caches a complete copy of the object. In contrast, our work provides support for integration of different consistency protocols. As mentioned earlier, our work uses reduced objects to minimize the amount of state cached at the client.

Like many other systems, we use the proxy principle[19] to implement a distributed object system. Like GARF[7], we separate the functionality of the object from other concerns. While GARF was focused on providing fault tolerance, our focus is on providing caching to improve the performance of an existing distributed object system.

A well-known object system is Thor[17]. Thor uses page-based, transactional object caching that uses a single protocol based on optimistic concurrency. Our work differs in that we do not require transactional boundaries and we support multiple consistency protocols. Instead of caching objects as a physical page, our work uses the logical relationships between objects to direct caching decisions.

Our work applies the principle of binary rewriting to creating objects that support caching in a middleware system. Binary rewriting has been used in the past to remove synchronization[1][5] and also to improve the performance of applet-loaded classes[12]. We extend the use of binary rewriting to the creation of objects for caching. As described below, we have developed tools to analyze the byte code in a Java class file and to manipulate that byte code to create the objects in our system. To achieve our byte code manipulation, we use the JavaClass API written by Markus Dahm[6].

3 Background and Requirements

Before undertaking the caching of Java RMI objects, the nature of such objects must be understood. Once this is understood, requirements can be developed to assure efficient and usable caching. These key requirements are RMI compatibility, flexible caching and consistency policy support, and minimal caching. These requirements, their motivating factors, and corresponding challenges are explained below. We also briefly discuss the issue of cache replacement.

3.1 Object Model

A Java RMI object is an object that implements an interface extending the *java.rmi.Remote* interface. The object is accessed at the client using the methods of the interface. To cache an RMI object, the structure of a Java RMI object must be understood.

When viewed in the strictest sense, an object contains fields that are either native types or references to other objects. The Java serialization mechanism serializes an object so that it can be transmitted to another JVM environment. When moving the object to another system, it is a simple matter to copy the native types. However, moving a reference implies also moving the referenced object. Some fields of the object and some referenced objects cannot be serialized. A field designated as *transient* cannot be copied to a client, since the Java serialization mechanism will not serialize a *transient* field when the object is

serialized. Also, an object that is not serializable cannot be cached. An example of this type of object is a FileReader object, which cannot be serialized because it refers to an open file. When a Java RMI object, or *Remote* object, is serialized, the object is replaced with a remote reference.

An object is cached by sending to the client the portions of the object that are expected to be used by the client. Once the object has been cached, the object and associated objects may be changed using methods of the interface. A method manipulates the fields of the object and referenced objects. A native field is manipulated using reads or writes to the field. A reference field can be modified to refer to another object, or a method can be invoked on the referenced object to change that object. If an object is cached, the caching mechanism must have some means of assuring that changes to fields and referenced objects are consistent with its other copies, including the primary copy of the object residing at the server.

Any middleware system that caches RMI objects must be aware of the object structure and provide mechanisms to assure the consistency of the fields of the cached object as well as the serializable objects referenced by the cached object.

In summary, the characteristics of RMI objects that affect caching are the following:

- transient variables cannot be cached
- non-serializable objects cannot be cached
- remote objects are serialized as RMI stubs and accessed via RMI stubs
- the serializable portion of an object as well as the serializable and non-Remote objects referenced by the object can be cached
- the system must assure that the cached object, including referenced objects, remain consistent with other copies

3.2 RMI Compatibility

A key requirement is RMI compatibility. Compatibility should be evidenced in the following ways. Caching should easily be added to an existing RMI application. Objects should be transparently cached at the client in a manner fully compatible with RMI. It should be possible to use caching with some *Remote* objects, yet access other *Remote* objects using RMI. Objects should be accessed using RMI, if caching the object will not improve performance. Security concerns may also prevent an object from being cached.

Because the current goal of our work is to determine the benefits of caching, we require the use of RMI as the underlying communication mechanism. Because prior work has established that RMI can be improved using other techniques[14][18], we wish to show that the benefits of our work are solely due to caching, not other factors.

Meeting these requirements poses several challenges. To be compatible with RMI, a suitable replacement for the standard RMI stub is needed. This stub requires more logic than a normal RMI stub in order to determine whether the stub should be passed as a reference, or if it should contain a cached RMI object. Furthermore, a tool similar to the RMI stub generator, *rmic*, needs to be designed to create these new stubs

3.3 Flexible Usage of Consistency Protocols

Another key requirement is that the caching mechanism should permit the integration of different kinds of consistency protocols, where each object can be assigned a consistency protocol best suited to its behavior. When copies of an object are located at several locations, in general, some mechanism must assure that the various copies are consistent with each other. This can be accomplished by assigning a consistency manager to each object cached on a client. Because it is possible for the object to be accessed at the server, a consistency manager is also required at the server. These consistency managers should be able to use a variety of consistency protocols, permitting each object to use a protocol suited to its semantics and usage.

Because objects differ widely in their usage and semantics, they have different consistency requirements. Consequently, consistency managers should not be tied to a specific consistency model or management protocol. When designing protocols, it should be possible to use a given protocol with a wide variety of objects. For each object in an application, the application developer should be able to select or create a protocol that meets the object's consistency and performance requirements.

Consistency protocols use different and often conflicting techniques. Examples of these techniques[2][15] include *cache invalidation* versus *cache update*, *write through* versus *write back*, and *data shipping* versus *method shipping*. Each of these techniques performs well in some situations and poorly in other situations.

To support multiple consistency protocols, a proper design must permit consistency managers to mediate access to an object and the cached copies of the object. These consistency managers also need to be able to access the state of the object. Developing consistency managers which can be generally applied to a wide variety of objects, yet at the same time have intimate knowledge of the structure of the object is a challenge to design and implement.

3.4 Minimize Cached Data

As mentioned earlier, a Java object may reference other objects. When caching an object, caching all objects referenced by the object is not desirable. Only those remote objects expected to be used should be cached. Other remote objects can either continue to be accessed using RMI or the appropriate mechanisms should be available to transparently cache the object either when it is first referenced or when it is first used. Furthermore, when caching an object, it may not be desirable to cache all contents of the object, since not all instance variables of an object may be used at the client. Consequently, the object cached at the client should only contain those instance variables that will be used.

The selective caching of objects and their instance variables provides efficiency in two ways. First the network overhead of transferring the unused data is eliminated. More important, the overhead of maintaining the consistency of the data is also eliminated. Another reason not to cache an instance variable

or object at the client is security. For example, if an instance variable or object contains sensitive data, it should not be exposed to the client or the network.

To accomplish the selective caching of instance variables of an object, we have developed the concept of a *reduced object*. A reduced object is a version of an object where unused instance variables, as well as methods that use those variables, have been removed. To effectively use reduced objects, several challenges had to be met. Mechanisms to generate the reduced object had to be created, including mechanisms to determine what instance variables should be included in a reduced object.

3.5 Cache Replacement

To simplify our design, we do not consider the issue of cache replacement. We assume that the cache will be large enough to hold all objects that will be cached. Gray and Shenoy suggest that because of the high latency of web access, a "cache everything" strategy should be used when caching web pages[8]. The same factors that influence the "cache everything" strategy for web pages will influence the use of large caches for distributed objects. While we do not consider the eviction of objects from the cache, in our design, objects in the cache are removed by the Java garbage collector when they are no longer in use.

4 Caching Mechanisms

This section presents the mechanisms of our system satisfying the requirements outlined in the previous section. These mechanisms are based on structures called the *cache template* and *server template*, which respectively represent the RMI object at the client and the server. The objects within the cache template and server template work together to manage the global state of the object as well as enable RMI compatibility. We also develop mechanisms that permit the development of consistency protocols that are applicable to a wide variety of objects.

4.1 Caching Structures

To assure consistency, all access to the copies of the object that reside at the clients as well as access to the server object must be mediated and coordinated. While some systems provide either virtual memory mechanisms, such as those available in software distributed shared memory systems like DOSA[10], or reflective mechanisms, like in MetaJava [11], to assist in managing fine grained access to an object, these mechanisms are not available in standard Java. To assure consistency in our system, we use interposition, in which an object manager is interposed between a copy of an object and the users of the object. We use interposition on both the client and the server. The middleware described in this paper uses two primary structures, one on the clients and one on the server, to ensure that all access to the object is mediated. These structures are described below.

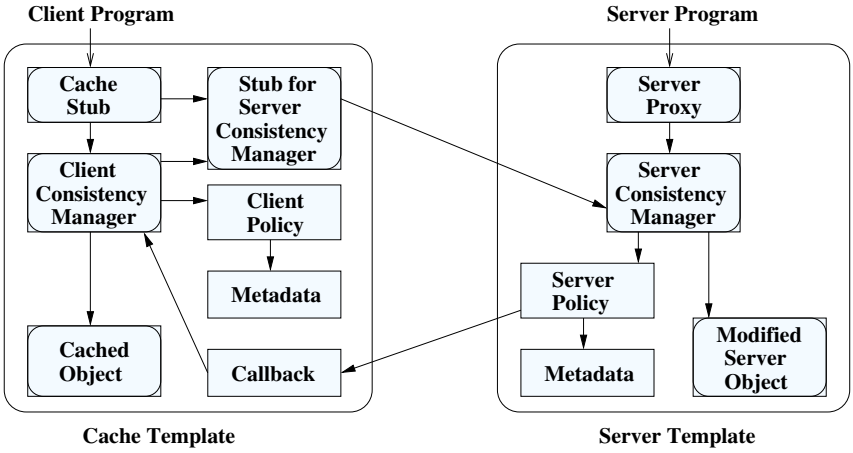

Fig. 1. Cache Template and Server Template

Cache Template. The *Cache Template* is our mechanism to cache RMI objects at a client. Its root object, the *Cache Stub*, replaces the standard RMI stub. The cache template consists of a cacheable version of the RMI object, along with accessory objects that manage the client's access to the cached object. As shown in Figure 1, the cache template consists of seven objects: the *Cached Object,* the *Client Consistency Manager*, the *Metadata*, the *Client Policy*, the *Stub for Server Consistency Manager*, the *Callback,* and the *Cache Stub*. Each of these objects plays a role in satisfying the caching requirements previously described.

The central object in the cache template is the cached object. The cached object is a copy of the object that resides at the server. This copy implements non-Remote versions of the interfaces of the object being cached. These non-Remote versions of the interfaces are identical to the interfaces of the object being cached with the exception that the interfaces no longer extend the *java.rmi.Remote* interface. This must be done because Java RMI will not serialize an object that implements the *java.rmi.Remote* interface (instead it will attempt to serialize the RMI stub for the object).

The cache object may be a reduced object. As mentioned earlier, a reduced object contains a subset of the instance variables of the original RMI object. A tool generates the classes needed for a reduced object. This tool is provided a list of methods that should be enabled in the reduced object. The tool determines which instance variables are needed by those methods and creates a new class that contains only those variables. Because the reduced class must implement all the interfaces of the object, all the method signatures are present. However, unsupported methods are changed so that they throw a "Not Available" exception if called. As explained below, the other objects in the cache template assure that an unsupported method will not be called. The reduced object is also changed

so that it properly supports the Java wait/notify mechanism. This is described below in Section 4.3.

Because the access to the cached object is controlled by other objects in the cache template, if any of the methods of the original object were declared as *synchronized*, that synchronization is no longer needed. Any synchronized methods of the original object have their synchronization removed from the cached object. This implies that the client consistency manager and client policy, described below, are responsible for the correct ordering of any methods invoked on the cached object. This includes the assurance that the state of the cached object is consistent.

Since other objects in the cache template need to update the fields of the object, the fields must be accessible. This is accomplished by changing the access privileges of any private fields in the RMI object to *package private*.

Since the cached object must be kept consistent with the object at the server, a direct reference to the cached object is not provided to the client program. Instead, a *client consistency manager* is interposed between the client program and the cached object. Like the cached object, the client consistency manager implements a non-Remote version of the interfaces of the object being cached. The client consistency manager works in conjunction with a *Client Policy* to maintain the consistency of the cached object. For each method invoked on the object, the client consistency manager consults the client policy, which determines what actions should take place to handle the method invocation. The client policy directs the actions of the client consistency manager using an *ActionsList*, which is discussed below. For a reduced object, the client policy assures that a method that is not supported by the reduced object will be executed on the server instead of on the client. In summary, the purpose of the client consistency manager is to enable the use of a generic policy with a specific object.

The client policy needs information about the object to guide the actions of the client consistency manager. This information is contained in the metadata object. The metadata object describes each method of the reduced object in terms of the instance variables which are accessed by the method and the manner in which each instance variable is used. Using the information about how each method uses the instance variables of the object, the metadata also classifies a method into one of the following five categories:

- NONE
  This type of method does not use any instance variables of the reduced object. Consequently, this type of method may be safely invoked at any time.
- IMMUTABLE
  This type of method only uses immutable instance variables of the reduced object. Since immutable instance variables never change, they are always valid if the object has been cached.
- READ-ONLY
  This type of method only reads instance variables of the reduced object. Since this type of method may read instance variables that may be changed

by other clients, the policy must assure that the instance variables read by the method are consistent.

- READ-WRITE
  This type of method both reads and writes instance variables of the reduced object. Since this type of method may change an object, the policy must assure that the invocation of this type of method will cause the object to remain consistent.
- SERVER-ONLY
  This type of method must be executed on the server because some instance variables used by the method are not available in the reduced object. A client policy will cause this type of method to be executed at the server.

The metadata for the reduced object is created by a tool which analyzes the byte codes of a Java class to determine which instance variables of the object are used by each supported method. Since an instance variables is only accessed using the "PUTFIELD" and "GETFIELD" byte codes, this analysis is fairly straight forward. Using this information, the methods of the reduced object are categorized into one of the above categories.

The client consistency manager communicates with the server using a *Stub for the Server Consistency Manager*. This object is a standard Java RMI stub for an object on the server, the server consistency manager, which is described below. It implements an interface that contains a modified version of all *Remote* methods of the object being cached. For each method, two changes are made. First, an additional *client context* parameter is added to the interface. The client context passes information to the server about the context in which the method is being called. This parameter is set by the client policy. One example of the context, which is always passed to the server, is the identity of the client. The second change to the interface is that an actions list is returned. This actions list permits the server to control the behavior of the client consistency manager. More details about the actions list are provided below.

Because the server may need to communicate with the client consistency manager, the cache template contains a *Callback* object to which the server may send requests. The callback object accepts remote requests from the server and forwards those requests to the client consistency manager. This permits the server to send requests such as cache updates and cache invalidations.

The client program accesses the object using a *Cache Stub*. The main purpose of the cache stub is to ensure RMI transparency and compatibility. The cache stub replaces the RMI stub found in standard RMI. The cache stub contains logic to assure that it is serialized in the correct form when passed via an RMI method. When passed to a client, the cache stub causes the sending of either an entire cache template or a *partial cache template*, consisting only of the cache stub and stub for server consistency manager. For example, if the cache stub is being sent to the RMI name registry, then the partial cache template is sent. Clients then receive the partial cache template when they lookup the object from the name registry. If the partial cache template is accessed by a client, the cache stub contains the logic needed to contact the server, using the stub for the

Table 1. Primary Function of Objects in Cache Template

Object	**Primary Function**
Cached Object	Contains state of cached RMI object
Client Consistency Manager	Manages access to the cache object based on the client policy
Client Policy	Defines in an object-independent manner how object is managed
Metadata	Describes the cached object
Callback	Permits the server to contact the client
Stub for Server Consistency Manager	Provides communication with the server
Cache Stub	Provides RMI compatibility and object faulting

server consistency manager, and request the object. This we refer to as *object fault handling*.

The cache stub also overcomes one of the shortcomings of the current RMI implementation. Suppose, a server receives through an RMI call, either as a parameter or return value, a reference to an RMI object residing at the server. If the server uses this reference, which is an RMI stub, to access the object, the communication overhead of RMI will be incurred. However, if a cache stub is received on the server, its deserialization routine will use the ID of the cached object to determine if it resides on the server. If so, it will allow itself to directly access the consistency manager for that object, thus avoiding unnecessary communication overhead.

The classes for the cache stub, client consistency manager, cache object, and stub for server consistency manager must be created for each class of object being cached. To facilitate the creation of the appropriate classes, we have developed suitable code generators. These generators use the Java class file of the remote object as input and produce the necessary classes.

The objects in the cache template permit the caching of an RMI object and control when a method is invoked on the cache object, assuring that the cache object is in a consistent state. In summary, the objects in the cache template and their functions are given in Table 1.

Server Template. The *Server Template* is the mechanism used to manage the primary object residing at the server. As shown in Figure 1, it consists of five objects. The original remote object is replaced by a modified server object. Like the cache template, the server template also has consistency manager and policy objects. To assure RMI compatibility, the server template also has a proxy object.

The modified server object is a slightly modified version of the original RMI remote object. A modified version of the object is needed for many of the same reasons that the client contains a modified version of the object. These reasons include changing the access privileges of instance variables and replacing synchronization primitives with primitives described in section 4.3.

Table 2. Primary Function of Objects in Server Template

Object	Primary Function
Modified Server Object	Maintains state of the object at the server
Server Consistency Manager	Manages access to the modified server object
Server Policy	Dictates in an object-independent manner how object is managed
Metadata	Describes the server object
Server Proxy	Provides RMI compatibility

```
public void sampleMethod(String inputParm) {
  actionsList = clientPolicy.getActionsList(methodId);
  executeActionsList(actionsList);
}
```

Fig. 2. General Structure of a Method of the Client Consistency Manager

The server consistency manager plays the same role as the client consistency manager on the client. It manages method requests from the server and clients. Like the client consistency manager, on each request it consults a server policy. The server policy returns an actions list to direct the behavior of the server consistency manager. Like the client policy, the server policy uses a metadata object to guide its decisions.

At the server, the object is accessed using the *Server Proxy*. The purpose of the server proxy is to provide RMI compatibility. If it is passed via RMI, RMI serialization will serialize the cache stub of the cache template. In other words, the RMI system considers the server proxy to be a Remote object that has the *Cache Stub* as its stub.

To use the cached template, minimal changes to the server are needed. First, a modified server object must be created instead of the original RMI objects. Second, instead of exporting the object using the export method of UnicastRemoteObject, the application must export the object using the export method of the CacheableObject class, which we provide.

In summary, the objects in the server template and their functions are shown in Table 2.

4.2 A Framework for Consistency Management

As mentioned earlier, the purpose of the client consistency manager is to intercept, via interposition, each method invocation. It consults the client policy for the actions to take to handle the method invocation. Figure 2 provides an example of the structure of a method in the client consistency manager. This method retrieves an actions list from the client policy and executes the actions in this list.

An *ActionsList* is a list of actions that are to be executed by a client consistency manager. We have identified several actions used to control a client consistency manager. Some of these actions are listed in Table 3.

Table 3. Actions Executable by Client Consistency Manager.

Action	Description
1. Invoke method	Invoke the method on the locally cached object.
2. Get actions list from server	Contact the server, passing the method id and the client context. Returns an action list.
3. Invoke method on server	Invoke the method on the server using RMI, passing method arguments and the client context. Returns an action list.
4. Set return value	Sets the return value of the method.
5. Update variable	Updates an instance variable of the cached object to a value.
6. Update policy state	Updates the state of the client policy. (This is explained in Section 4.2)

When a client policy is consulted by a client consistency manager, it considers the state of the cached object as well as the characteristics of the method. If the client policy determines that a method can be safely invoked (for example, the method is an IMMUTABLE method), it returns the "Invoke Method" action. Otherwise it may return an action to contact the server, either action 2 or 3 in Table 3. For these actions, the server consistency manager returns an additional actions list to be executed by the client consistency manager. For example, this list could contain actions to set the return value of the method, to update the instance variables of the cached object, and to update the state of the client policy. For example, this state could be used to track which instance variables of the cached object are valid.

Examples of actions executed by the client for certain scenarios are given in Table 4. For example, in scenario 1, a method only uses immutable instance variables. The client policy returns the action to invoke the method locally on the cached object. In scenario 2, a method uses instance variables that are only accessible at the server. The client policy returns an action to invoke the method on the server. After executing the method, the server returns an action list to set the return value of the method. In scenario 3, the object state used by the method is invalid. The client policy returns an action to get an actions list from the server. The server returns actions to update the instance variables of the cached object, update the policy state to indicate that the cached object is valid, and finally to invoke the method on the cached object.

The goal of the consistency mechanisms is to enable actions to be taken both before and after the method on the cached object is invoked. The client consistency manager is directed by the client policy as well as the server policy objects. The policy objects at the client and the server cooperate to assure that the consistency and synchronization requirements associated with the object are met.

Table 4. Examples of actions executed by client consistency manager for particular situations. Actions in SMALL CAPITAL LETTERS indicate actions from the client policy. Actions with an asterisk (*) and in *italics* were received from the server.

Scenario	**Actions Executed by Client Consistency Manager**
1. Method uses immutable instance variables	INVOKE METHOD LOCALLY
2. Method uses instance variables only available at server	INVOKE METHOD ON SERVER * *Set return value*
3. Method uses instance variables that are not valid at the client	GET ACTIONS LIST FROM SERVER * *Update variable* * *Update policy state* * *Invoke method locally*

4.3 Wait and Notify Support

This middleware assures consistency by controlling the principal entry points into a cached object, namely the method entry and exit. However, an object may be entered and exited in another manner. This is through the Java *wait* and *notify* mechanisms. The system must ensure that the semantics of *wait* and *notify* are preserved. To accomplish this, our tools modify the cached object (as well as the server object) such that the invocations of the *wait* and *notify* methods are respectively replaced by invocations of the *handleWait* and *handleNotify* methods of the consistency manager.

When a *handleWait* method of the client consistency manager is invoked, it asks the client policy object for an actions list. This actions list typically contains an action to inform the server that a client object has executed the *wait* method.

The *handleNotify* method must be implemented differently. Java semantics dictate that when a thread issues a *notify,* a waiting thread will awake and acquire the monitor lock after the current thread releases the monitor lock. In the most common case, this occurs when a synchronized method returns. For this reason, the *handleNotify* method sets a flag to indicate that *notify* has been invoked. After the method returns, if this flag is set, the client consistency manager executes an actions list that it obtains from the client policy.

Figure 3 shows the sequence of method calls that take place in order to handle *wait* and *notify*. In general, a *wait* invoked on a client causes that client to wait. That client is awakened after another client invokes *notify*. For clarity, interactions with client policy and server policy are not shown. In step 1, a method of the cached object is called on client A. Where the original object called *wait*, the cache object now invokes the *handleWait* method, as shown in step 2. In step 3, the client consistency manager notifies the server that there is a client waiting, returning any object state that has been modified at the client. The method invocation of client A then blocks at the server. In step 4, the client consistency manager of client B invokes a method on the cached object. In step 5, this method invokes the *handleNotify* method in the place where the original object called *notify*. As described above, a flag is set to indicate that a notify is pending. In step 6, the object returns from the method that invoked

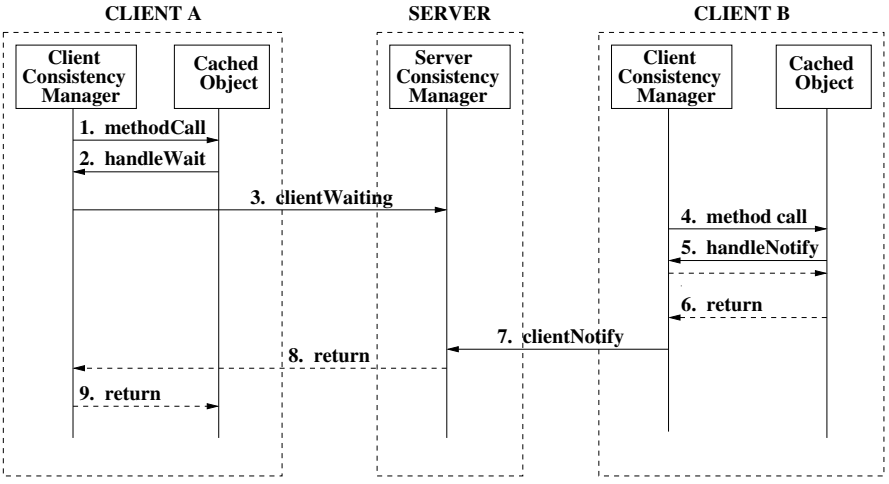

Fig. 3. Sequence of Method Calls for Wait Notify

handleNotify. At this point, as shown in step 7, the client consistency manager notifies the server that a notify has occurred, and passed any updates that have been made. The server consistency manager then wakes up the waiting thread of client A as shown in steps 8 and 9. The thread then continues executing after the *handleWait* call in the cached object.

5 Experimental Results

To evaluate our mechanisms, we used two different RMI applications. The first is a simple producer-consumer application that is used to determine compatibility of our system with the Java wait/notify mechanisms. The second is a medium size object benchmark similar to the TPC-C benchmark[21]. In this benchmark we show the use of two different consistency policies and reduced objects. We use these two applications to determine the baseline performance of RMI. We then compare this baseline with the performance of these applications after caching has been added.

5.1 Distributed Producer-Consumer Application

We use a simple producer-consumer application to evaluate our support for Java wait-notify. In this application, a server exports a buffer object that is accessed by two clients. The *produce* and *consume* methods of the buffer object are shown in Figure 4.

To cache this object, we designed client policy and server policy objects that ensure that the buffer can only be cached at one client at a time. These policy objects were also designed to minimize communication with the server, when the

```
synchronized public void produce(Object item) {
    while (produceAt >= (consumeAt + items.length)) wait();
    items[produceAt % items.length] = item;
    produceAt++;
    notify();
}
synchronized public Object consume() {
    while (consumeAt == produceAt)  wait();
    Object item = items[consumeAt % items.length];
    consumeAt++;
    notify();
    return item;
}
```

Fig. 4. Producer Consumer Object

Table 5. Performance of Producer Consumer Application

Buffer Size	RMI	Caching
64	4311 ms	3893 ms
512	4202 ms	3450 ms
2048	4281 ms	2824 ms

buffer methods are invoked by a producer and a consumer. A *wait* method is processed as shown in Figure 3. However, the sending of a *clientNotify* message is deferred until either a *wait* method is called or until the object has not be used for 100 milliseconds. At that time, the cached object is invalidated and the changes to the object are returned to the server. Overall, this protocol permits the producer to produce many objects before the cached object is returned to the server. The server then updates the buffer at the consumer and the consumer can consume many objects before releasing the object. We call this policy the *delayed notify* policy.

For our experiments, the producer creates 4096 string objects. We measure the time from when the producer deposits the first item to the time when the consumer retrieves the last item. We verify correctness by assuring the sequence of objects consumed is the same as the sequence of objects produced.

We evaluated the performance of our caching mechanism in a LAN environment, using various buffer sizes in the object. In this environment, the server, producer, and consumer were connected via a 100 Mbps switched Ethernet connection. We compared the performance of RMI with the performance of our caching system. The results of our experiments are shown in Table 5.

As seen in the table, the use of caching improves performance as the buffer size increases. This is because with larger buffer sizes, network interactions are reduced. With a buffer size of 64, the performance improved by 10% and with a buffer size of 2048, the performance gain was 34%.

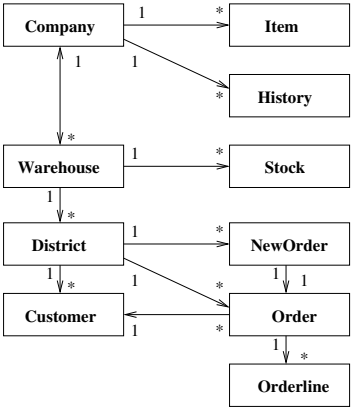

Fig. 5. Objects in the rmiBOB Benchmark

5.2 Object Benchmark

To measure the performance of our mechanism, we created rmiBOB, a port (using Java RMI) of the Business Object Benchmark (BOB) benchmark from the book Enterprise Java Performance [9]. The benchmark implements the business logic in the TPC-C benchmark[21]. The benchmark creates a "company" and associated objects. The benchmark measures the performance of a set of "transaction" operations on the objects.

We ported the BOB benchmark to RMI by changing the primary objects in the application to be RMI objects. The primary objects in the benchmark are shown in Figure 5. There is one company object that has the items sold by the company as well as a history of payments made by customer. This company has warehouses that stock the items sold by the company. Each warehouse has sales districts, with customers assigned to each sales districts. Each district has orders that are placed by customers, and each order contains a number of order lines. For our tests, we used an initial population of 6,141 objects, as shown in Table 6.

The benchmark executes five types of "transaction" operations on these objects. The transactions are executed such that for every 23 transactions the following number of transactions are executed: 10 new order, 10 payment, 1 order status, 1 delivery, and 1 stock level.

We modified the benchmark to run 1000 transactions. To verify that the consistency protocols were working correctly, we enabled screen writes and saved the output to a file. At the end of the run, we compare the output to a previous RMI run to assure that the results are correct.

5.3 Baseline Performance

To get an idea of the overhead of the architecture, we performed two experiments. We first measured the performance when the benchmark used RMI. We

Table 6. Initial Number of Objects

Class	Count
Company	1
Item	1000
History	300
Warehouse	1
Stock	1000
District	10
Customer	300
NewOrder	210
Order	300
Orderline	3019
Total	6141

Table 7. Performance Comparison of RMI and the Middleware with Caching Disabled

configuration	average transaction	server calls	average server call
rmi	187.947 ms	-	-
middleware with no caching	187.561 ms	79632	2.589ms

then measured the performance when the benchmark used our mechanisms with caching disabled. We disabled caching by using a client policy that causes all method invocations to go to the server. The performance of these two configurations is shown in Table 7. The reader will notice that our mechanism performs slightly better than the RMI version. This improvement is due to the benefit, described in section 4.1, of directly accessing the object when a cache stub is returned to the server.

5.4 Caching Using Consistency Protocols

To experiment with our mechanisms, we implemented two simple consistency protocols. These protocols ensure serial consistency by assuring that all writes and reads are coordinated.

The first protocol we implemented is a *server-write* protocol. In this protocol, all methods that change the state of an object are considered write methods. All write methods go to the server, which invalidates all clients using the object. The updates to the object are returned to the client which called the method. Other clients are updated when they access the object.

The second protocol we implemented is a *multiple-readers / single-writer* (MRSW) protocol. If a client accesses an object in write mode, it implicitly obtains a lock for the object and can write the object locally. If the server or another client then accesses the object, the lock is recalled by the server using the callback object and the updates are returned to the server.

We experimented with these protocols to select the most appropriate protocols for each object. In our application, since all the objects of the same class

Table 8. Consistency Protocols Used for Caching Objects

Object Class	Protocol
Company	Server Write
Warehouse	Server Write
Stock	Server Write
Customer	MRSW
Orderline	ServerWrite

Table 9. Performance of Caching and Reduced Objects

configuration	average transaction	server calls	average server call
caching	126.060 ms	15931	9.744 ms
reduced object	117.524 ms	14681	8.90 ms

were used in the same manner, we used the class as the basis for selecting the protocol. We selected the combination shown in Table 8 as the best mix. The results of this combination is shown in Table 9. Using this mix, we measured an average transaction cost of 126.06 ms, which is 33% less than the RMI transaction cost.

5.5 Benefits of Reduced Object

We then performed an experiment to determine the benefit of reduced object caching. We first ran our benchmark using a client policy that recorded which methods where being used on the cached objects. Then, using the list of methods used by each object as input to our tools, we created reduced objects for the objects in Table 8. The results of the experiment are shown in Table 9. The use of reduced objects had the following impacts. The average transaction cost fell to 117.524 milliseconds, which is 37% less than the RMI cost. When compared to caching without reduced objects, the number of server calls dropped by 8% and the average cost of a call to the server dropped by 9%. This improvement is due to the removal of false sharing between the client and the server. Since reduced objects were used, the cached items were not invalidated when the server used portions of the object that were not cached at the client. Since the client cache was not invalidated, the client did not need to contact the server to request a valid copy before executing a method locally.

6 Conclusion

In this paper, we have presented requirements and mechanisms for efficient caching of objects for Java RMI applications. Using these mechanisms, caching can be easily and transparently added to existing RMI applications, while preserving RMI compatibility. This includes the ability to support the Java *wait* and *notify* mechanisms. No changes are required to existing clients and only

minimal changes are required at the server. The central mechanism on the client is the *cache template* containing three important objects: a cache stub ensuring RMI compatibility, a client policy implementing a consistency protocol, and a *reduced object* containing a subset of the state of the object being cached. On the server, similar objects exist in the *server template*.

Using these mechanisms in conjunction with appropriate consistency and caching policies, we illustrated the benefits of caching using two RMI applications. Our first experiment used a *delayed notify* policy to improve the performance of a producer-consumer application by 34%. Our second experiment involved a medium sized object benchmark. We showed how *server write* and *multiple reader single writer* protocols could be used on the same application to improve performance by 33%. We also show how using a *reduced object* improved performance by 37%.

These mechanisms serve as the basis of our ongoing research in caching of Java objects. While our experience showed performance improvements in a LAN environment, we can expect better performance when using caching in wide-area networks. We plan to expand our work to include consistency protocols suitable for use in a wide-area network environment, including the support of distributed collaborative applications. We also plan to expand our work to permit the reduced object to change dynamically as the client's usage of the object changes.

References

1. Aldrich, J., Chambers, C., Sirer, E. G., and Eggers, S. Static analyses for eliminating unnecessary synchronization from Java programs. In *Proceedings of the Sixth International Static Analysis Symposium* (Venezia, Italy, Sept. 1999), pp. 19–38.
2. Archibald, J., and Baer, J.-L. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems 4*, 4 (Nov. 1986), 278–298.
3. Aridor, Y., Factor, M., Teperman, A., Eliam, T., and Schuster, A. A high performance cluster JVM presenting a pure single system image. In *Proc of 2000 Java grande conference* (San Francisco, CA, June 2000), pp. 168–177.
4. Bakker, A., van Steen, M., and Tanenbaum, A. S. From remote objects to physically distributed objects. In *Proceedings of the 7th IEEE Workshop on Future Trends of Distributed Computing Systems* (Cape Town, South Africa, Dec. 1999), pp. 47–52.
5. Bogda, J., and Hölzle, U. Removing unnecessary synchronization in Java. In *Proceedings of the 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (Denver, Colorado, Nov. 1999), pp. 35–46.
6. Dahm, M. Byte code engineering with the JavaClass API. Tech. Rep. B-17-98, Freie Universitat Berlin, July 1999.
7. Garbinato, B., Guerraoui, R., and Masouni, K. R. Implementation of the GARF replicated objects platform. *Distributed Systems Engineering Journal 1*, 1 (Mar. 1995).

8. Gray, J., and Shenoy, P. Rules of thumb in data engineering. In *Proceedings of the 16th International Conference on Data Engineering* (San Diego, CA, Feb. 2000), pp. 3–12.
9. Halter, S. L., and Munroe, S. J. *Enterprise Java Performance*. Prentice Hall PTR, 2000.
10. Hu, Y. C., Yu, W., Cox, A., Wallach, D., and Zwaenepoel, W. Runtime support for distributed sharing in typed languages. In *Proceedings of LCR2000: The Fifth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers* (Rochester, NY, May 2000).
11. Kleinöder, J., and Golm, M. MetaJava: An efficient run-time meta architecture for Java. In *Proceedings of the Fifth Workshop on Object-Orientation in Operating Systems (IWOOOS '96)* (Seattle, Washington, Oct. 1996), pp. 54–61.
12. Krintz, C., Calder, B., and Hölzle, U. Reducing transfer delay using Java class file splitting and prefetching. In *Proceedings of the 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (Denver, Colorado, Nov. 1999), pp. 276–291.
13. Krishnaswamy, V., Ganev, I. B., Dharap, J. M., and Ahamad, M. Distributed object implementations for interactive application. In *Proceeeding of Middleware 2000* (New York, New York, Apr. 2000), pp. 45–70.
14. Krishnaswamy, V., Walther, D., Bhola, S., Bommaiah, E., Riley, G., Topol, B., and Ahamad, M. Efficient implementation of Java remote method invocation (RMI). In *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)* (Sante Fe, New Mexico, Apr. 1998).
15. Levy, E., and Silberschatz, A. Distributed file systems: concepts and examples. *ACM Computing Surveys 22*, 4 (Dec. 1990), 321–374.
16. Lipkind, I., Pechtchanski, I., and Karamcheti, V. Object views: Language support for intelligent object caching in parallel and distributed computations. In *Proceedings of the 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (Denver, Colorado, November 1999), pp. 447–460.
17. Liskov, B., Castro, M., Shrira, L., and Adya, A. Providing persistent objects in distributed systems. In *Proceedings of ECOOP'99* (Lisbon, Portugal, June 1999), pp. 230–257.
18. Nester, C., Philippsen, M., and Haumacher, B. A more efficient RMI for Java. In *Proceedings of The ACM 1999 Java Grande Conference* (San Francisco, June 1999), pp. 153–159.
19. Shapiro, M. Structure and encapuslation in distributed system – the proxy principle. In *Proceedings of the Sixth International Conference on Distributed Computing Systems* (May 1986).
20. SUN Microsystems. Java remote method invocation specification, 1997. http://www.javasoft.com/products/jdk/1.1/docs/guide/rmi/spec/rmiTOC.doc.html
21. Transaction Processing Performance Council. TPC benchmark C. http://www.tpc.org/cspec.html, 1999.
22. van Steen, M., Homburg, P., and Tanenbaum, A. S. Globe: A wide-area distributed system. *IEEE Concurrency 7*, 1 (Mar. 1999), 70–78.