SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2007; **37**:799–831
Published online 2 November 2006 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.783

Mechanisms for object caching in distributed applications using Java RMI

John Eberhard*,† and Anand Tripathi

Department of Computer Science, University of Minnesota, Minneapolis, MN 55455, U.S.A.

SUMMARY

Remote Method Invocation (RMI), a mechanism to access remote objects in Java-based distributed applications, uses network communication for each method invocation. Consequently, using RMI in a wide-area environment can cause poor application performance. One solution to improve performance is to cache the objects such that network communication is not necessary for each method invocation. In this paper, we present mechanisms to transparently add object caching to RMI. These mechanisms are compatible with existing RMI applications and use an event-based model to support different consistency policies. The mechanisms also include the ability to adaptively select the consistency policy for an object based on its usage pattern. A novel feature of our mechanisms is the use of a 'reduced object', which is a partial representation of the RMI object. We experimentally evaluate and demonstrate the benefits of our mechanisms. Copyright © 2006 John Wiley & Sons, Ltd.

Received 6 February 2006; Revised 10 July 2006; Accepted 14 July 2006

KEY WORDS: RMI; distributed systems; caching

1. INTRODUCTION

Distributed object systems are commonly implemented using some type of remote procedure call, or in the Java‡ case, Remote Method Invocation (RMI) [1]. One problem inherent with RMI is the high latency of method invocation caused by the necessity of network communication. An approach to overcome the latency problem is to manually cache components of an object's state at the client's node. For example, a programmer could structure a client program so that frequently used

*Correspondence to: John Eberhard, Department of Computer Science, University of Minnesota, Minneapolis, MN 55455, U.S.A.
†E-mail: eberhard@cs.umn.edu
‡Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Contract/grant sponsor: NSF; contract/grant numbers: 0082215 and 0411961

components of an object are cached at the client. When manually replicating the object's state in this manner, the programmer must also ensure that the replicated copies of the object remain mutually consistent. Obviously, programmatically adding caching in this manner is expensive and error-prone. A better solution is the use of mechanisms to transparently add caching to an RMI application. These mechanisms should permit the programmer to choose the caching and consistency policies best suited to the application.

Other environments can also take advantage of caching. On one extreme, in a mobile environment when the client is not connected, operations can be invoked on a cached object. Caching can also be useful in a real-time environment where the object access must meet critical timing constraints.

Mechanisms for caching Java RMI objects must meet certain requirements. First, the object caching mechanisms must be compatible with and transparent to current RMI clients. Second, these mechanisms must support consistency policies tailored to both the semantics and usage of an object. Third, a client should only cache the data that it will access. The primary reason of this requirement is to minimize the communication cost of transmitting a large object between the client and the server. Other reasons include memory constraints of 'thin clients' and the need, driven by security, to not allow the caching of the sensitive components of an object.

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

- identification of issues in caching RMI objects;
- mechanisms, compatible with existing RMI based applications, to cache RMI objects;
- an event-based framework enabling the implementation of a wide range of consistency policies;
- a cost model for determining the cost of a consistency policy, and policy switching mechanisms that permit different policies to be used for different objects, based on each object's usage pattern;
- the notion of *reduced object*, which caches only parts of an object, to reduce the overhead in caching the full object.

We evaluate our caching mechanisms for RMI applications and demonstrate their performance benefits. This evaluation illustrates RMI compatibility and the use of different consistency policies. Furthermore, this evaluation shows the benefits of reduced objects and the benefits of adaptively switching the consistency policy used by the cached object.

The remainder of this paper is organized as follows. Section 2 reviews related work. Section 3 discusses issues and requirements to be addressed when caching RMI objects. Section 4 presents our caching mechanisms and illustrates their support of different consistency policies and adaptive policy selection. Section 5 provides experimental evidence of the benefits of the caching mechanism. Section 6 presents the conclusion.

2. RELATED WORK

Several researchers have investigated caching to improve the performance of RMI and Java distributed object systems. Caching has also been used in distributed object systems. In comparison to existing work, our mechanisms retain RMI compatibility, permitting existing RMI clients to transparently use caching. Our mechanisms also permit different consistency policies to be associated

DOI: 10.1002/spe

with different objects. We also permit several policies to be used for a given object, where the policy is chosen based on the object's usage. We also introduce and support the notion of 'reduced object', in which an object contains a subset of the state of the original objects. Below, we briefly review the existing work most relevant to this paper.

Several researchers have improved the performance of RMI by improving the serialization of Java objects used with RMI [2–4]. While our work does not attempt to improve the base performance of RMI by improving the serialization used by RMI, we use our own custom serialization to minimize the overhead of additional objects used by our mechanisms.

In their work on RMI performance, Krishnaswamy *et al*. [5] describe a caching system used to improve RMI. Their system differs from our work because they cache a serialized version of the object inside the reference layer of the Java RMI implementation. A consistency framework assures that the entire serialized object remains consistent with the object on the server. As with their approach, our system retains RMI compatibility. Unlike their approach, our system caches the objects in their unserialized forms. Furthermore, rather than caching a complete object, our system also permits selective caching of a subset of an object's components.

In a later work, Krishnaswamy *et al*. [6] developed an object caching framework supporting different kinds of consistency management techniques. As with our work, they permit the creation and use of a variety of consistency policies. Our work differs from their approach in that we provide a more flexible framework in which different kinds of consistency policies can be implemented using event-based mechanisms. We also provide the ability to dynamically change the consistency policy used to manage object caching.

Other researchers [7] 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. Lipkind *et al*. [8] 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 [9–11]. As with the mechanisms presented in this paper, several subobjects are used on the client to manage a cached object. These subobjects serve roughly the same purpose as the objects in the caching architecture presented in this paper; for example, subobjects are used for supporting communication and coordination with remote servers to ensure cache consistency. Our mechanisms differ from Globe in the use of byte code analysis to identify write methods instead of requiring the programmer to explicitly identify them. Unlike our approach, Globe does not support the use of Java *wait*/*notify* synchronization and does not permit a cached object method to call another cached object method.

Rover [12] is a distributed object system that provides support for disconnected operations. 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 and conflicting concurrent operations are required to be reconciled using resolver functions. As with most object systems, Rover caches a complete copy of the object.

DOI: 10.1002/spe

In common with many other systems, we use the proxy principle [13] to implement a distributed object system. Recently, Java 1.3 introduced a dynamic proxy mechanism [14] that has been used to implement distributed object systems [15,16]. Similar to GARF [17], we separate the functionality of the object from other concerns. While GARF was focused on providing fault tolerance, our focus is on improving the performance of RMI-based distributed applications.

Thor [18] is a distributed object system with page-based, transactional object caching. It uses a single consistency policy based on optimistic concurrency. Our work differs in that we do not require transactional boundaries and our framework can support multiple consistency policies and dynamic switching between policies at runtime.

Our work applies the principle of binary rewriting for creating objects to support caching. Binary rewriting has been used in the past to remove synchronization [19,20] and also to improve the performance of applet-loaded classes [21]. To achieve our byte code manipulation, we use the Jakarta Byte Code Engineering Library [22].

We presented our initial work in [23]. Caching support for the Java *wait* and *notify* synchronization constructs was presented in this earlier work and is omitted here. In this paper, we have refined and extended several aspects of our caching mechanisms. The most significant changes are our event-based consistency management framework and the mechanisms for adaptively selecting the consistency policy for an object.

3. ISSUES IN CACHING OF RMI OBJECTS

This section presents issues and problems present when caching RMI objects. First, the caching mechanisms must be compatible with RMI. Second, since an object may be cached at multiple clients, appropriate consistency management policies must be supported. Third, since caching involves the placing of data on the client, the Java object model must be understood, including properties that determine which components of the object may be cached.

3.1. Design issues and requirements related to RMI compatibility

Java RMI permits a client to access objects residing on remote servers. To use RMI, a Java interface extending *java.rmi.Remote* is defined for remotely accessible objects. This interface is then implemented by a *Remote* object. This implementation must be enabled to receive remote method invocations either by creating the object as a subclass of *UnicastRemoteObject*, or by using the *exportObject()* method of *UnicastRemoteObject*. Prior to JDK 5.0, the programmer uses the Java *rmic* tool, or RMI compiler tool, to generate the *stub* and *skeleton* classes for each distributed object. In JDK 5.0, when a stub class does not exist, the RMI runtime will use Java's dynamic proxy mechanism [14] to create a dynamic proxy which uses a *RemoteObjectInvocationHandler* as the invocation handler for the proxy.

To enable a client to obtain a reference to an RMI object, Java provides an RMI registry, with which a server registers a remote object. The RMI registry merely stores the stub or proxy for the object. When a client looks up an object in the RMI registry, it receives the stub, which serves as a reference for the *Remote* object. The client invokes a method on the stub, resulting in the invocation of the method on the server. The key feature of the RMI system is that when a *Remote* object is passed as a parameter to a method invocation, the RMI system serializes and passes the stub as a reference for the object.

DOI: 10.1002/spe

When caching a RMI object, two aspects of RMI compatibility must be maintained. First, the client protocol of interaction with the RMI registry should be unaltered. Second, the object returned by the RMI registry as a reference to the *Remote* object should be usable in the same manner as the RMI stub, with the exception that it may cache the RMI object.

When caching an RMI object, we use the term *cache stub* to refer to the object which replaces the RMI stub. When passing a *Remote* object as a parameter to an RMI call, the use of a cache stub in place of an RMI stub should be transparent to the client. In the case where a cache stub is being passed from the server to a client that has not cached the object, the cache stub should include a copy of the object. Otherwise, the client already has a cached copy of the object and including a cache copy of the object is unnecessary.

RMI also provides the ability to load classes from the network using dynamic class loading. A server configures dynamic class loading by using the *java.rmi.server.codebase* property to specify the URL from which to load classes. This codebase information is included when RMI passes a *Remote* object, enabling the client to load classes remotely.

These issues result in the following requirements for RMI compatibility and transparency. When accessing an RMI object, a client application should not have to change in order to use caching. Instead, the appropriate mechanisms should ensure that a suitable representation of the RMI object is available to the client application when the object is requested.

3.2. Design issues and requirements related to consistency management

When a cached RMI object is used, the behavior should be consistent with normal RMI object behavior. To ensure consistency, a proxy must intercept each method invocation and assure the method invocation completes in a consistent manner. The proxy should be designed such that a small set of mechanisms can enable the implementation and use of different kinds of consistency policies. When choosing a consistency policy for an object, an application developer should be able to create a new consistency policy or to select one from a library of existing policies. While it is possible to analyze an application's behavior and assign a single policy for each object, there may be cases where an object may require different consistency policies at runtime, depending on the current usage of the object. A caching mechanism must include the ability to adaptively select the consistency policy best suited to the current usage of the cached object.

3.3. Design issues and requirements related to the Java object model

Caching an RMI object implies that a copy of the object is placed on the client. The placement of a copy of an RMI object causes difficulties due to Java mechanisms for object serialization, object synchronization, nested method invocation, and object creation.

Java object serialization converts a *Serializable* object, including its components and referenced objects, into a *Stream* of bytes which is sent through a communication channel. To serialize an object, Java serialization converts the fields of the object into their serialized form. A field of an object may be marked as *transient* to indicate that it should not be serialized. The Java serialization mechanism will fail if a field references a non-serializable object.

To cache an RMI object, a suitable representation of the object, called 'cached object', must be created. Since the *Remote* interface implemented by the RMI object causes an RMI stub to be serialized,

DOI: 10.1002/spe

a 'cached object' must not implement the *Remote* interface. Furthermore, since transient and non-serializable fields cannot be serialized, the 'cached object' should not include these fields. Since some fields may not or should not be used on the client, the 'cached object' should not include these kinds of fields.

Another factor to consider when caching an RMI object is the synchronization mechanism in Java. All Java objects support object synchronization and coordination through the use of the *wait* and *notify* methods. Because an RMI object may use these mechanisms, caching mechanisms must support the use of the *wait* and *notify* methods.

An RMI object has the ability to invoke a method on another RMI object. If active replication is used in conjunction with 'function shipping', the duplicated method invocation problem [24], or nested invocation problem [25], may occur. The root of this problem is the use of 'function shipping' to update object replicas. As each function is invoked on a replica, each replica then could invoke a method on another object. This latter object then receives multiple method invocations, when it should have received only one. In the consistency policies presented in this paper, this problem does not appear because 'function shipping' is not considered. However, in the case where our mechanisms are extended to use function shipping, either a mechanism must be used to prevent duplicate method invocations or a Globe-like restriction on nested method invocations [25] must be followed by the cached RMI objects.

A final factor to consider is an RMI object's ability to create other RMI objects. When caching is not used, the created RMI objects reside on the server. To maintain the current behavior of an RMI application, created RMI objects must reside on the server. This can be accomplished by requiring that all methods creating RMI objects be invoked on the server.

4. CACHING MECHANISMS AND CONSISTENCY POLICIES

To support caching, we have created the *cache template* and *server template* which respectively represent the RMI object at the client and the server. The cache template and server template contain components which ensure RMI compatibility, provide consistency management, and maintain cached object state.

4.1. Overview of cache template and server template

The cache template and server template, shown in Figure 1, are the basic structures used to cache an RMI object. The topmost objects in the cache template and server template are the *cache stub* and *server proxy*. Their primary purpose is to assure RMI compatibility, especially when references to RMI objects are passed between Java virtual machines.

The next level of objects in the cache template and server template are the consistency managers: *Client Consistency Manager (CCM)* and *Server Consistency Manager (SCM)*. The role of a consistency manager is to intercept a method invocation and perform actions to consistently complete that method invocation. To complete the method invocation, the CCM may need to communicate with the SCM, which it does using the *SCM stub.* Because the SCM may need to communicate with the CCM, a *callback interface* is also provided.

DOI: 10.1002/spe

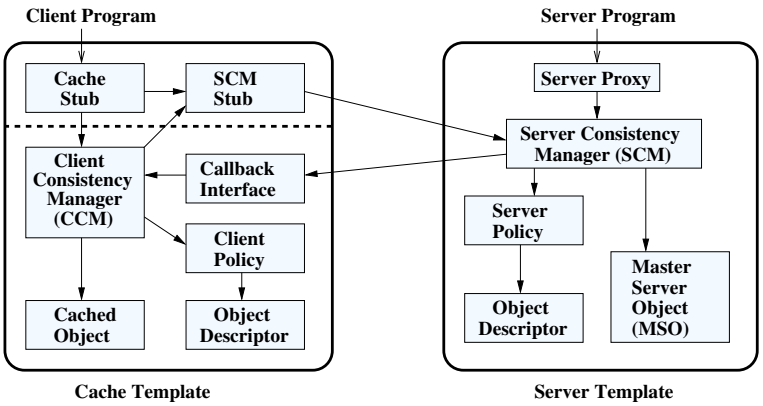

Figure 1. Cache template and server template.

The SCM and CCM are controlled by objects representing the consistency policy used to manage the object. These objects are named *client policy* and *server policy*. In order to implement a consistency policy, these objects must have some knowledge about the object being managed. This knowledge is captured in the *object descriptor*.

The cache template uses the *cached object* to contain the cached state of the RMI object. The object may not contain all the state of the object, in which case we call it a 'reduced object'. The Server Template contains the *Master Server Object* (MSO), which represents the RMI object on the server.

4.2. Mechanisms for RMI compatibility

The objects of the cache template and server template that interface with the application program are the cache stub and server proxy. These objects ensure RMI transparency and compatibility and provide the same Java *interface* as the original RMI object. The classes for the cache stub and server proxy are created from the original RMI classes using a tool that we have developed. These two classes contain a method corresponding to each method in the original RMI object's interface. The generated class for the cache stub extends a base class that contains most of the logic for RMI compatibility.

To assure RMI transparency and compatibility, the class generated for the cache stub extends a Java class named *CacheStubBaseClass*. This class contains a reference to the SCM stub, which is used to contact the server. It may also contain a reference to the CCM. The cache stub contains logic to ensure 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 of only the cache stub and the SCM stub. For example, at the time when the cache stub is sent to the RMI registry, a partial cache template is sent. Clients then receive the partial cache template when they lookup the object from the RMI registry. When the partial cache template is accessed by a client, the cache stub contains the logic needed to contact the server, using the SCM Stub, and to request the remaining components of the cache template. This we refer to as *object fault handling*.

DOI: 10.1002/spe

Local applications at the server access the object using the server proxy. The purpose of the server proxy is to provide RMI compatibility. The RMI system considers the server proxy to be a Remote object that has the cache stub as its RMI stub. Consequently, if the server proxy is passed via RMI, RMI serialization will serialize the cache stub. The cache stub also overcomes a shortcoming of the current RMI implementation. Suppose a server receives, through a parameter of an RMI call, 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 recognizes that it has returned to the server and allows the cache stub to directly access the server proxy, thus avoiding unnecessary communication overhead. This optimization also eliminates the need for a cache template to include the SCM stub, when returning an RMI reference to the server. This can cause a significant reduction in communication cost for applications that return RMI references to the server.

To use the server template, minimal changes to the server application are needed. First, the server proxy object, described above, must be created instead of the original RMI object. 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. This method uses the *export* method of *UnicastRemoteObject* to register the server proxy with RMI and to export the SCM. When the server proxy is registered with RMI, the system uses the corresponding cache stub as the RMI stub. Because our mechanisms rely on RMI serialization to serialize the cache stub, dynamic loading of class files can still be used with our caching mechanisms.

As mentioned earlier, RMI produces a proxy for an exported class either using a Stub class or, beginning in JDK 1.5, by dynamically generating a proxy. We rely on the former behavior to cause the use of cache stub instead of the standard RMI stub. An alternate approach is to use a Java dynamic proxy in conjunction with reflection. Unfortunately, the current RMI system does not permit the use of an invocation handler other than the default *RemoteObjectInvocationHandler*. For this reason, we use the older approach of using precompiled cache stubs.

4.3. Mechanisms for consistency management

As introduced in Section 4.1, the consistency managers (CCM and SCM) are controlled by a consistency policy represented by the client policy and server policy components of the templates. A policy directs a consistency manager using an event–action model.

4.3.1. Event–action model overview

To allow a client policy or server policy object to direct a CCM or SCM, we adopted an event–action based model for consistency control. In this model, a CCM or SCM sends a policy event to a client policy or server policy to indicate that some consistency related actions are required. This event is triggered by a client program invoking a method on the client template or by actions taken by the CCM or SCM. In response to an event, the client policy or server policy returns an action list to direct the CCM or SCM. An Action List is a list of actions that is sequentially executed by the CCM or SCM.

Figure 2 illustrates the typical usage of policy events and action lists. In this figure, two clients have cached the object. In step 1, a client program invokes a method. The CCM uses an event to notify the client policy of the invocation in step 1c, to which the client policy returns an action list in step 1d.

DOI: 10.1002/spe

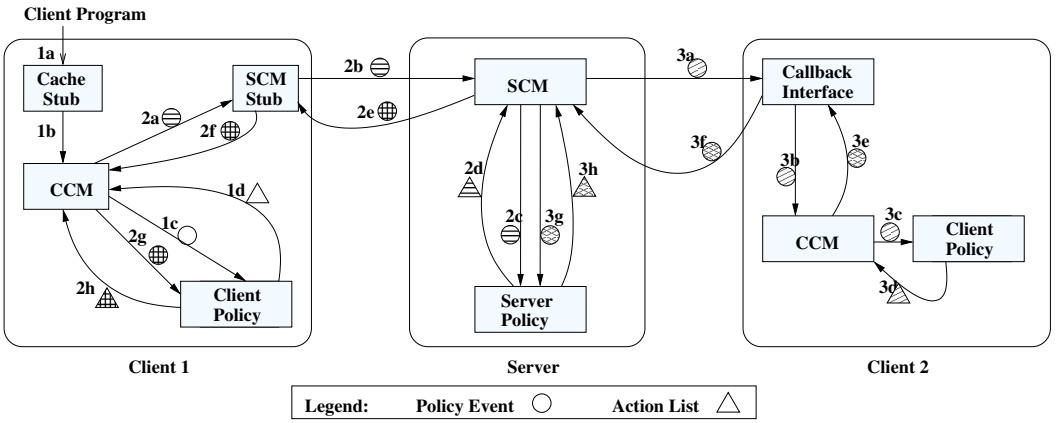

Figure 2. Usage of action lists and policy events.

Suppose, in step 2, that the method cannot be invoked on the cached object. In that case, the action list, from step 1d, directs the CCM to build an event and invoke the method on the server, passing the method parameters and this event, as shown in steps 2a and 2b. In step 2c, the SCM passes this event to the server policy. In step 2d, the server policy then returns an action list to the SCM. The SCM executes the action list and creates an event that is returned in steps 2e and 2f. This event is delivered to the client policy in step 2g, which causes the client policy, in step 2h, to return an action list to direct the CCM.

In some cases, another client must be notified, using the callback interface, to assure the consistency of the RMI object. To accomplish this, in step 3a, the server policy directs the SCM to send an event to the client's callback interface. This event is delivered to the client policy in steps 3b and 3c. Upon receiving the event, the client policy returns an action list to the CCM in step 3d. The CCM executes this action list and creates an event that is returned to the server in steps 3e and 3f. In step 3g, the event is delivered to the server policy, which provides an action list to the SCM in step 3h.

4.3.2. Policy events

A policy event's primary purpose is to convey information about a consistency condition to a policy object. To identify a consistency condition, an event contains a *type*, a *subtype*, and *attributes*. The type indicates the condition that caused the event to be generated. As shown in the steps in Figure 2, the possible types are *MethodCall* (step 1c), *ServerCall* (step 2a), *ServerDone* (step 2e), *Callback* (step 3a), and *CallbackDone* (step 3e). The subtype enables a policy to convey simple information about the event. One use of the subtype is to indicate the consistency policy related state of the cached object. An example of this will be shown later when discussing the Multiple-Reader-Single-Writer (MRSW) policy. Since the event is used to send information between the server and clients, it also includes attributes that a policy can set. Attributes that can be included are a client id, a return value,

DOI: 10.1002/spe

Table I. Actions executable by consistency managers.

Action category	Actions
Invocation flow control	InvokeOnLocal(), InvokeOnServer()
Object state control	SetReturnValue(returnedValue), SetCachedObject(cachedObject)
Policy event construction	AddSubtype(eventSubtype), AddReturnValue(), AddCachedObject(), AddObjectCopy(classname)
Callback control	Callback(List of {clientId, policyEvent})

and a cached object. The client id uniquely identifies a client that caches an RMI object and is always included with a *ServerCall* and *CallbackDone* events. The remaining attributes, described below, are set using actions.

In our initial implementation, we passed an event as an RMI parameter and allowed RMI to handle the serialization of the event. However, we found that the RMI serialization of the event added considerable overhead. To overcome this difficulty, we created routines to convert between an event and a Java string. We pass the string instead of the event as the RMI parameter. As will be illustrated later, this serialization change drastically reduced the overhead associated with the passing of the event.

4.3.3. Action lists and actions

The Action List contains actions, where each action directs the CCM or SCM to perform a consistency-related function. While executing these actions, the CCM or SCM maintains two objects. The first object is the method-return-value, which is returned to the client program after the CCM finishes executing an action list. This value is set when the CCM invokes a method on the cached object or when the SCM invokes a method on the *Master Server Object* (*MSO*). Alternatively, this value can also be set using an action. The second object is a policy event that is created when an action list is executed. Actions are used to set the attributes of this event or cause the event to be delivered to a server policy or client policy.

The actions executable by a CCM or SCM, shown in Table I, can be classified into four categories: invocation flow control, object state control, policy event construction, and callback control. Examples of using these actions are provided later.

- *Invocation flow control.* If an object is cached, one of the primary functions of the CCM or SCM is to invoke a method on the local object, either a cached object or MSO. An *InvokeOnLocal* action causes a CCM or SCM to invoke a method on the local object and set the method-return-value. If a policy determines that a method should not be invoked locally, it must contact the server. The client policy uses the *InvokeOnServer* action to direct the CCM to invoke the method on the server, passing a policy event. The steps that occur after the server is contacted were described above.
- *Object state control.* Two actions are used to set the value returned by the CCM and to change the cached object. The *SetReturnValue* action includes a return value parameter and directs the

DOI: 10.1002/spe

CCM to set the method-return-value. The *SetCachedObject* action directs the CCM to set the cached object using the specified cached object parameter.

- *Policy event construction.* Several actions are used to add attributes to the event created by the CCM or SCM. The *AddSubtype* action sets the subtype of the event. To add the return value of the server method invocation to the event, an *AddReturnValue* action is used. To add the cached object to the event, the *AddCachedObject* action is used. To create a new copy of the object at the server and added it to the event, the *AddObjectCopy* action is used. This action contains the classname of the object to send to the client. The SCM uses this classname to create an object from the MSO.
- *Callback control.* The server policy uses a *Callback* action to trigger actions on a client currently caching an object. This *Callback* action specifies a list of client id/policy event pairs. The processing of a *Callback* action was shown in Figure 2 and takes place in three phases. In the first phase, as shown by step 3a, the SCM uses each client id/policy event pair to send the corresponding client the *Callback* policy event. In phase 2, the client policy receives the *Callback* event (step 3c) and returns an action list (step 3d) to the CCM. The CCM then executes the action list and returns a *CallbackDone* event (steps 3e and 3f) to the server. In phase 3, a *CallbackDone* event is received from each client (step 3g). The server policy then uses an action list (step 3h) to direct the SCM to take additional actions. The *Callback* action is complete after all clients have returned a *CallbackDone* event and the subsequent action lists have been processed.

4.3.4. Object descriptors

A generic consistency policy requires information about the cached RMI object to guide the actions of the CCM or SCM. While the specific information required by the client policy depends upon the consistency policy it implements, we provide an *object descriptor* for use by general purpose consistency policies. The object descriptor categorizes each method of a cached RMI object according to the method's usage of the object's instance variables. The three categories of instance variable usage are as follows.

- *Read-only*. This type of method only reads the object's instance variables.
- *Read–write*. This type of method both reads and writes the object's instance variables.
- *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 cached object.

Our current analysis is a pessimistic analysis that considers a method as a single entity and does not consider different execution branches of the method. A possible area for future research is to use compiler-inspired approaches to determine the conditions under which different instance variables are used by the method. Once this is known, each method can be assigned categories based on a specific precondition.

4.4. Implementation of different consistency policies

The flexibility of our event-based consistency management framework permits the use of a variety of consistency policies. In this section, we illustrate three generic consistency policies that use policy events and action lists. These policies are the *no-cache policy*, the *server-write policy*, and the *MRSW policy*.

DOI: 10.1002/spe

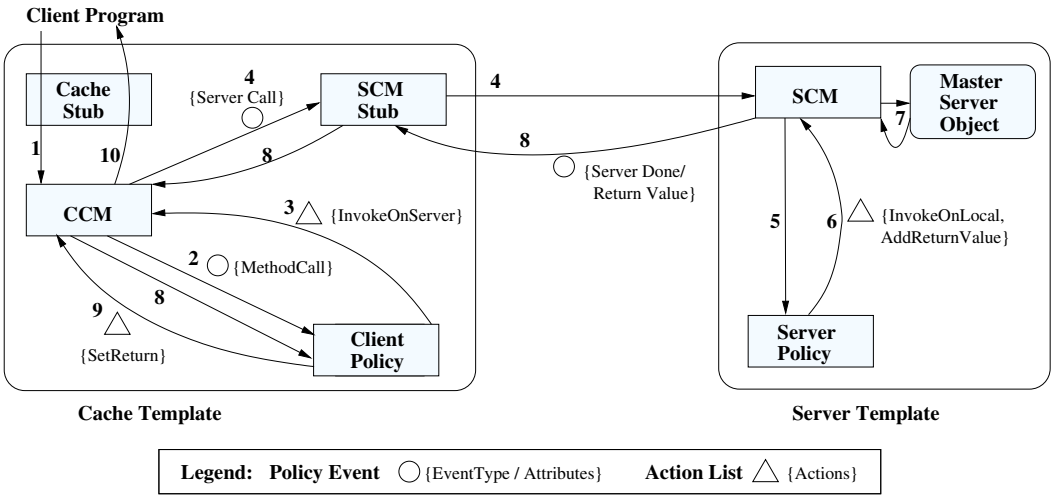

Figure 3. Steps for executing a method using the *no-cache policy*.

4.4.1. No-cache policy

The simplest consistency policy is one where caching is disabled and all client method invocations communicate with the server. We implemented this policy in order to compare the overhead of our mechanisms with standard RMI. This performance data will be given in Section 5.2. Figure 3, shows the steps that occur when a method is invoked and includes the corresponding policy events and action lists. In step 1, the method is invoked and calls the corresponding method of the CCM. In step 2, the CCM notifies the client policy of the method invocation using a *MethodCall* event. In step 3, the client policy returns an action list containing one action, *InvokeOnServer*. In step 4, the CCM executes this action and uses the SCM stub to invoke the method on the server, passing a *ServerCall* event. In step 5, the SCM delivers this event to the server policy, which returns, in step 6, an action list containing *InvokeOnLocal* and *AddReturnValue*. The SCM first executes the *InvokeOnLocal* action by invoking the method on the MSO as shown in step 7. It then executes the *AddReturnValue* action to add the method-return-value to the *ServerDone* event that is returned to the client. In step 8, this event is delivered to the client policy. In step 9, the client policy retrieves the return value from the event and uses it to create a *SetReturnValue* action, which is returned in an action list. The CCM then executes this action list, setting the return value of the method, which is returned in step 10 to the client program.

4.4.2. Server-write policy

A slightly more complex policy, the server-write policy, caches an object for access by read-only methods. In this policy, all methods that modify the object are executed at the server. When the method

DOI: 10.1002/spe

Table II. Client policy responses for server-write policy.

	Policy event type	Method type or *(event subtype)*	Consistency state	Response	
				Action list	New state
1	*MethodCall*	Read-only	Valid	{InvokeOnLocal}	Valid
2	*MethodCall*	Read-only	Invalid	{InvokeOnServer}	Invalid
3	*MethodCall*	Read–write Server-only	Any	{InvokeOnServer}	Invalid
4	*ServerDone*	*(ReadComplete)*	Any	{SetCachedObject, InvokeOnLocal}	Valid
5	*Callback*	*(Invalidate)*	Any	{}	Invalid
6	*ServerDone*	*(WriteComplete)*	Any	{SetCachedObject, SetReturnValue}	Valid

is executed at the server, all cached copies are invalidated to ensure that all clients will view the latest changes to the object.

In this policy, the object is cached in one of two consistency states: *Valid* or *Invalid*. The *Valid* state means that the cached object contains the latest changes to the object. When in this state, the client policy permits read-only methods to be invoked locally. Otherwise, the *Invalid* state indicates that the cached object should not be used because it does not contain the latest changes to the object.

The server-write policy can be represented by tables indicating the responses returned by the client policy and server policy for specific policy events. Table II shows the responses returned by the client policy. Rows 1–3 of Table II describe how the client policy handles a method call. The simplest case, shown in row 1, occurs when calling a read-only method and the consistency state is valid. In this case, the client policy returns an action list containing *InvokeOnLocal*. This causes the CCM to invoke the method on the cached object. For all other method invocations, the method must be invoked on the server. The client policy returns an action list containing the *InvokeOnServer* action to cause the CCM to send a *ServerCall* event to the server.

Upon receiving the *ServerCall* event, the server policy returns an action list as shown in rows 1 and 2 of Table III. If a read-only method was invoked, the server policy recognizes that the client has not cached the object. As shown in row 1, it returns an action list to add an object copy and the subtype *ReadComplete* to the *ServerDone* policy event. The SCM then executes these actions and returns the *ServerDone* event to the client. As shown in row 4 of Table II, the client policy recognizes the event subtype of *ReadComplete* which indicates that the *ServerDone* event contains a cached object. It then uses the cached-object in the event to create a *SetCachedObject* action to update the cached object. It then sets the current consistency state to *Valid* and adds the *InvokeOnLocal* action to invoke the method.

If a *ServerCall* event indicates that a read–write or server-only method was invoked, the server policy returns the actions shown in row 2 of Table III. The first action causes the SCM to initiate the three phases of callback processing. In phase 1, the SCM sends a *Callback* event of subtype of *Invalid* to each of the clients. In phase 2, the client policy receives the *Callback* event and, as shown by row 5 of Table II, sets the cache state to *Invalid* and returns an empty action list. The CCM then returns

DOI: 10.1002/spe

Table III. Server policy responses for server-write policy.

	Event type	Method type	Action list
1	*ServerCall*	Read-only	{AddObjectCopy, AddSubtype(ReadComplete)}
2	*ServerCall*	Read–write Server-only	{Callback({clientId, Policy Event(Invalid)}*), InvokeOnLocal, AddReturnValue, AddObjectCopy, AddSubtype(WriteComplete)}
3	*CallbackDone*	—	{}

a *CallbackDone* event to the server. In phase 3, the server policy receives the *CallbackDone* event and, as shown by row 3 of Table III, returns an empty action list to indicate that no additional actions need to be taken by the SCM. Phase 3 ends after *CallbackDone* events have been received from all the clients currently caching the object.

Once the SCM has completed the three phases of the callback processing, it executes the remaining actions in the action list. It invokes the method, and adds the return value, object copy, and subtype to the policy event. In this case, the subtype is set to *WriteComplete* to indicate that a server-write operation has completed. Once the action list has been processed, the *ServerDone* event is returned to the client. Row 6 of Table II shows the client policy's response to the *ServerDone* event. In this case, the client policy knows that the event contains an updated cached object as well as a return value. The information from the event is used to construct an action list containing *SetCachedObject* and *SetReturnValue*, which will update the cached object and set the return value. The policy also changes its state to *Valid*.

4.4.3. Multiple-Readers-Single-Writer policy

We have also implemented a policy, MRSW, which permits read–write methods to be executed locally on a cached object. In this policy, an object is cached either in read-only mode by multiple clients or in read–write mode by a single client.

This policy considers a cached object to be in one of three consistency states: *Write, Read,* or *Invalid*. The *Write* state indicates the cached object is the only cached copy of the object and invoking write methods is permitted. The *Read* state means that the cached object is valid and read-only methods can be invoked. Otherwise, the *Invalid* state indicates that the Cached Object is stale and should not be used.

As with the server-write policy, the MRSW policy is represented by tables indicating the client policy and server policy responses. A client policy's responses for method calls are shown in rows 1–5 of Table IV. If it is valid to invoke a method in the current consistency state, then an action list containing *InvokeOnLocal* is returned as shown in rows 1, 2, and 3. Otherwise, an action list containing *InvokeOnServer* is returned, as shown in rows 4 and 5. The client-side processing for this action list is the same as for the server-write policy.

On the server, the server policy considers the object to be in one of three consistency states: *NotCached*, *ReaderCached*, or *WriterCached. NotCached* means that the object is

DOI: 10.1002/spe

Table IV. Client policy responses for the MRSW policy.

	Policy event type	Method type or *(event subtype)*	Consistency state	Response	
				Action list	New state
1	*MethodCall*	Read-only	Read	{InvokeOnLocal}	Read
2	*MethodCall*	Read-only	Write	{InvokeOnLocal}	Write
3	*MethodCall*	Read–write	Write	{InvokeOnLocal}	Write
4	*MethodCall*	Any	Invalid	{InvokeOnServer}	Invalid
5	*MethodCall*	Read–write	Read	{InvokeOnServer}	Read
6	*Callback*	*(Invalidate)*	Read	{}	Invalid
7	*Callback*	*(Invalidate)*	Write	{AddCachedObject}	Invalid
8	*ServerDone*	*(Read)*	Any	{SetCachedObject, InvokeOnLocal}	Read
9	*ServerDone*	*(Write)*	Any	{SetCachedObject, InvokeOnLocal}	Write

not cached by any clients. *ReaderCached* means that the object is cached by clients in *Read* state. *WriterCached* means that the object is cached by one client in *Write* state. Table V shows the response of the server policy to policy events. Rows 1, 2, and 3 show the responses to *ServerCall* events. For these events, the policy invalidates, if needed, existing clients using the *Callback* action. In the case where only readers have cached the object, the callback processing is identical to the callback processing for the server-write policy, as shown by row 6 of Table IV and row 4 of Table V. In the case where a client has cached the object in *Write* mode, the client policy will add the cached object to the *CallbackDone* event and set the client cache state to *Invalid* as shown by row 7 of Table IV. Upon receiving the *CallbackDone* event, the server policy completes the callback processing by using a *SetCachedObject* action to update the MSO as shown by row 5 of Table V.

After the callback processing has been completed, the action lists in the first three rows of Table V use the *AddObjectCopy* action to add an object copy to the policy object. It also uses the *AddSubtype* action to return the client's consistency state. If the client should cache the object in *Read* state, then the subtype is set to *Read*. Otherwise the subtype is set to *Write*.

4.5. Adaptive policy selection

The previous section described various consistency policies usable by cached RMI objects. Each policy's performance benefit depends upon the behavior of the clients. For example, while using the MRSW policy with an object that is cached by only one client may result in good performance, using the same policy when several clients concurrently access the object may result in poor performance. In order to achieve optimal application performance, the caching mechanisms should provide a facility to adaptively select the consistency policy used for a cached object. This facility should select the policy that will provide the best performance and should enable the changing of the consistency policy as the behavior of the object changes.

DOI: 10.1002/spe

Table V. Server policy responses for MRSW policy.

	Event type	Method type	Consistency state(s)	Response	
				Action list	New state
1	*ServerCall*	Read-only	*ReaderCached*	{AddObjectCopy, AddSubtype(Read)}	*ReaderCached*
2	*ServerCall*	Read-only	*WriterCached*	{Callback({. . . (Invalidate)}*), AddObjectCopy, AddSubtype(Read)}	*ReaderCached*
3	*ServerCall*	Read–write	*ReaderCached* or *WriterCached*	{Callback({. . . (Invalidate)}*), AddObjectCopy, AddSubtype(Write)}	*WriterCached*
4	*CallbackDone*		*ReaderCached*	{}	*NotCached*
5	*CallbackDone*		*WriterCached*	{SetCachedObject}	*NotCached*

This section describes the mechanisms to adaptively change the consistency policy used for a cached object. Before deciding whether to change the consistency policy, we must be able to determine the cost of a particular consistency policy, given the previous behavior of client access. To estimate this cost, we have defined the *locality window* cost model to estimate a consistency policy's caching behavior and associated cost. Once a decision is made to change the consistency policy, steps must be taken to accomplish that policy change.

4.5.1. Locality window cost model

To assess the cost of various consistency policies, we have developed a cost model to represent the cost of caching an object. When designing our cost model, we considered several factors. First, the recording and reporting of usage data should not cause significant overhead. The information reported by each client should be minimal and should be passed only when the consistency policy requires communication with the server. Second, a policy should be able to estimate its cost based on usage data collected by another policy. Third, we assume that the primary cost of a policy will be the time required for a client to communicate with the server.

Each time a client caches an object, it incurs a communication cost. We call this cost the *retrieval time*. Using the retrieval times experienced by the clients, there are two ways to calculate the cost of a policy. In one method, the cost of a policy is calculated as the sum of all retrieval times experienced by the clients. This method ignores that some communication occurs in parallel and would be useful in environments where minimizing communication is more important than application performance. Because our current focus is to improve the client response time, we use another method that improves system throughput by calculating the policy cost using retrieval times that are known to be sequential. In this method, we organize the retrieval times into groups, which we refer to as frames, where communication could occur in parallel. We then calculate the cost of a policy as the sum of the maximum retrieval time from each group.

DOI: 10.1002/spe

```
class LocalityWindow { Vector<LocalityFrame> localityFrames; }
class LocalityFrame { Vector <ClientUsageData> usageData; }
class ClientUsageData { ClientId clientId; long retrievalTime; Vector<MethodUsage> usage }
class MethodUsage { int methodId; int count }
```

Figure 4. Locality window definitions.

By considering these factors, we have defined a cost model which we call the *locality window model*. A *locality window* represents clients' method invocations on a cached object during a specific period of time. As shown in Figure 4, a locality window consists of a sequence of *locality frames*, where each locality frame represents concurrent access by clients to the cached object while a particular consistency policy is being used. Typically, a locality frame begins when a client starts caching the object and ends when the server revokes all cached copies of an object. A locality frame can also begin when a method is invoked on the server. Each locality frame contains the *usage data* for each of the clients accessing the object. The usage data for each client consists of the client id, the retrieval time, and the id and the count of methods invoked by the client. The usage data is returned to the server when a cached object on a client is invalidated by the server. A locality frame is created at the server by combining the usage data returned by the clients during cache invalidation.

Each locality frame represents the cost of caching an object during a time period. Because we consider communication from different clients to occur in parallel, we calculate the cost of a locality frame as the largest retrieval time in the locality frame. The costs for the locality frames are summed together to calculate the cost of the locality window.

Each policy provides a Java method, *estimateCost()*, which uses a locality window to estimate the cost. In general, a policy estimates this cost by determining the number of locality frames that would have been in the locality window had the policy been used. The basic approach is to examine each locality frame to determine if it would have been combined with the previous frame, or remained as a frame, or divided into several frames. The number of locality frames is then multiplied by the average retrieval time for the policy.

Figure 5 shows the algorithm used to estimate the cost, under a given policy, of executing the methods in a locality window. Given a locality window as input, the goal is to estimate the count of frames in the locality window that would result under the given policy. The algorithm builds an estimated locality window by iteratively processing each frame in the input locality window to estimate and add frames to this the estimated window. In each iteration step, the algorithm maintains a *working frame* for the estimated locality window and examines the *current frame* from the input window as described below.

If the current frame is *compatible* with the working frame, then the current frame is combined with the working frame. Two frames are *compatible* if none of the method invocations in the two frames would cause a cache invalidation. If the frames are not compatible, the algorithm determines if the current frame is a *legal* frame for the policy. A frame is *legal* if none of the method invocations within the frame would cause a cache invalidation. If the current frame is *legal*, then it becomes the working frame and the frame count is incremented because the previous working frame would have been added to the estimated locality window. If the frame is not *legal*, then the frame must be split into multiple

DOI: 10.1002/spe

```
/* Algorithm to estimate cost for a policy */
/* Input: localityWindow          Output: estimatedCost */
long estimateCost(LocalityWindow localityWindow) {
  LocalityFrame workingFrame = UNKNOWN;   int framecount = 0;
  for (LocalityFrame currentFrame : localityWindow) {
    if (compatible(workingFrame, currentFrame) {
      workingFrame = mergeFrames(workingFrame, currentFrame);
    } else {
      if isLegalFrame(currentFrame) {
        workingFrame=currentFrame;  frameCount++;
      } else {
        workingFrame=UNKNOWN;   frameCount += frameSplitCount(currentFrame);
      }
    }
  }
return frameCount * averageRetrievalTime;
}
```

Figure 5. Algorithm to estimate policy cost.

frames that are legal for the policy. Therefore, the algorithm uses the *frameSplitCount()* method to determine the maximum number of *legal* frames needed to represent the current frame. This count is added to the frame count and the working frame is set to UNKNOWN. The working frame is set to UNKNOWN because the ordering of the new frames is not known. This may cause the algorithm to slightly over-estimate the cost. After all the frames have been processed, the cost is calculated using the frame count and the average retrieval time for that policy. This average retrieval time is maintained by our caching mechanisms. It is based on the current application run as well as previous application runs.

Figure 6 shows examples of estimating locality windows from locality windows generated by other policies. In Example 1, a locality window generated by the server-write policy is converted to a locality window for the MRSW policy. In this case, frames 1 and 2 can be combined to form frame A because the MRSW policy can cache an object in read–write mode. Frame 3 is not compatible with frame A and cannot be merged. However, frame 3 is a legal frame for the MRSW policy and is added to the estimated window, as shown by frame B. In Example 2, a locality window generated by the MRSW policy is converted to a locality window for the server-write policy. In this case, frame 1 is a *legal* frame, and is added to the estimated window as frame A. Frame 2 cannot be merged with frame A and it is not a *legal* frame. Because, in the server-write policy, all writes take place at the server, and each write operation must take place in a single frame. Consequently, frame 2 must be divided into frames B, C, and D.

At runtime, the server periodically uses the locality window to estimate the costs for other candidate policies for comparison with the cost of the current policy. If the estimated cost is 20% less, then a policy switch will be triggered.

DOI: 10.1002/spe

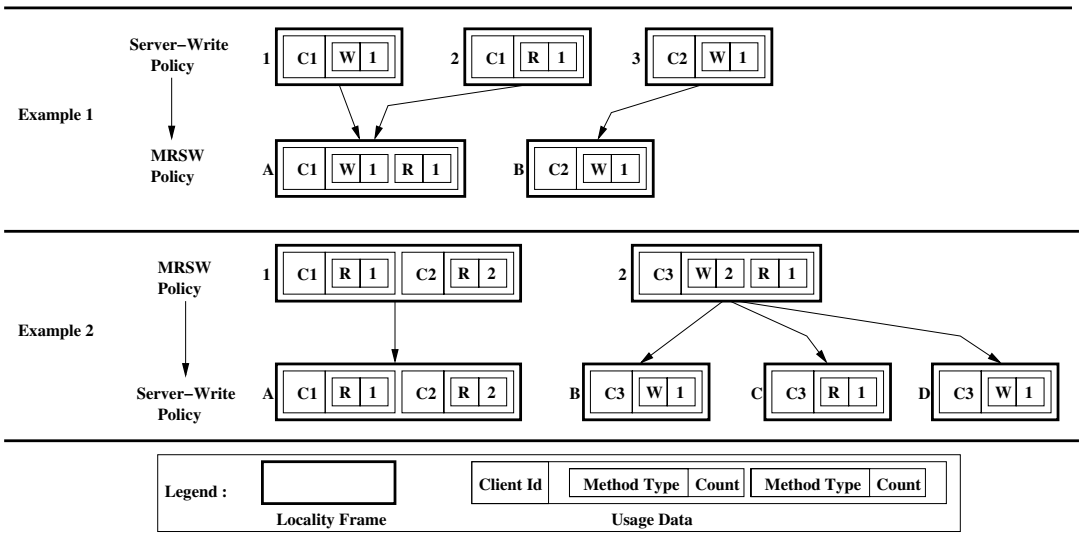

Figure 6. Examples of estimating cost by locality window merging and splitting.

4.5.2. *Policy switching*

Once the server has used the cost model to determine that the policy should be changed, mechanisms are needed to support dynamic policy switching. We have augmented our caching mechanisms to permit the switching of consistency policies. Figure 7 shows steps taken to switch the consistency policy, along with the associated policy events and action lists.

Once the server has decided that the policy must be switched, the current server policy sends a *PolicyRelinquish* action to the SCM, as shown in step 1. In step 2, the SCM receives the *PolicyRelinquish* action and sends a *Suspend* event to every client that is currently caching the object. Each client's CCM then delivers the *Suspend* event to the current client policy. In step 3, the current client policy takes the necessary actions to suspend itself so that it can no longer be used. It then returns an action list to build a *SuspendDone* event. If the cached copy of an object has updates, this action list should contain actions to add those updates to the *SuspendDone* event. In step 4, the *SuspendDone* event is returned to the server and given to the current server policy. In step 5, the server policy returns an action list to update the MSO object with the updates from the client. In step 6, the SCM executes the action lists from the all clients and then discards the current server policy. It then creates a new server policy and asks the new server policy for *Switch* events to be sent to the clients. In step 7, the new server policy provides *Switch* events to be passed to the clients. Each event must include the new client policy to be used by the client. These events are then sent from the SCM to the CCM. In step 8, the CCM retrieves the new client policy from the *Switch* event and replaces the current client policy with the new client policy. In step 9, the CCM returns to the SCM an indication that the switch processing has completed.

DOI: 10.1002/spe

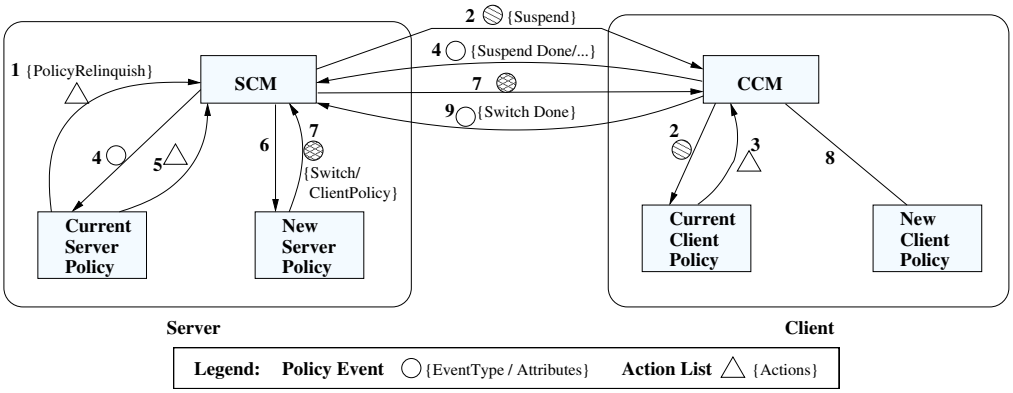

Figure 7. Steps for switching consistency policy.

4.6. Modified server object and cached object

On the server, a MSO represents the RMI object. On the clients, the cached object represents the RMI object. The MSO is a slightly modified version of the original RMI object. The first reason for the modification is to permit the SCM to access the instance variables of the RMI object. The second reason is to replace Java synchronization primitives with method calls to support Java object synchronization, as discussed in Section 3.3 and also in [23]. The tool used to generate the MSO is discussed below.

In order to satisfy the design requirements as outlined in Section 3.3, we create an appropriate representative of the object for use on the clients. This representative object, a reduced object, is a version of an RMI object which is serializable, which may contain a subset of the instance variables of the original RMI object, and which has been modified to support Java object synchronization based on *wait*/*notify*.

When creating a reduced object, the *Remote* interface is removed from the definition of the object and replaced with the *Serializable* interface. This change permits Java to serialize the instance variables of an object when it is transferred to a remote node.

Since only a subset of the instance variables of an object may be used on the client, a reduced object may contain those variables. During application development, our mechanisms can record which methods are invoked on a cached object. The application developer then provides this list of methods to the reduced object generator to create a reduced object containing only the specified methods and the instance variables which they use. The unused methods are disabled. The tool also creates the object descriptor for the reduced object, which marks the unused methods as 'server-only', so then will not be invoked on the cached object.

The final change to create a *reduced object* is to replace calls to the Java synchronization *wait* and *notify* methods with calls to CCM. The CCM will convert those calls into appropriate events that are delivered to the client policy.

DOI: 10.1002/spe

The use of reduced objects provides several benefits. First, communication overhead is reduced as unnecessary object components are not transferred to the client. Second, false sharing is reduced. If a method invoked at the server only accesses instance variables that are not cached, it is not necessary to invalidate clients that have cached other parts of the object. Third, a reduced object can prevent sensitive data from being cached on a client. The object descriptor can be used to identify methods that access sensitive data. By excluding those methods from the list of methods provided to the reduced object generator, the application developer can remove sensitive data from the reduced object.

4.7. Code generation

In order to implement our caching mechanism, we developed several programs to create the Java classes for the objects in the cache template and server template. When enabling an RMI object for caching, we use a script, *GenerateObjects*, to invoke these programs.

The first class files created by *GenerateObjects* are two Java interfaces. The first interface is the 'non-remote', or NR, interface. This interface has the same methods as the *Remote* interface used by the RMI object, but the interface extends the *java.rmi.Serializable* interface instead of the *java.rmi.Remote* interface. The NR interface is needed to enable the creation of the CCM and cached object classes which have the same methods as the RMI object, but must be serializable so that they can be transferred between the client and the server. The second interface is the *SCMInterface*. This is the *Remote* interface which is used by the CCM to communicate with the SCM. When the CCM communicates with the SCM, it must also pass a policy event. Consequently, the *SCMInterface* contains methods with the same names as the methods of the RMI's object remote interface, but each method has an additional parameter to permit a policy event to be passed with the method invocation.

The next class files created by *GenerateObjects* contain the classes for the objects in the server template. These objects are the server proxy, the SCM, and the MSO. The server proxy implements the Remote interface. For each method in the Remote interface, the corresponding method of the SCM is called. The SCM implements the *SCMInterface* discussed above. For each method, code is generated to deliver a *MethodCall* event to the server policy and to execute the resultant action list. The MSO is generated from the original server object class, making the changes described in the previous section.

GenerateObjects then creates the class files for the objects used by the cache template. First, the class for the cache stub is created. For each of the methods in the NR interface, a method is generated which calls the CCM, if it is available. If the CCM is not available, the method will contact the server to retrieve the CCM. Second, the class for the CCM is created. For each method in the remote object, a method is generated which creates a *method call* policy event and delivers it to the client policy. The CCM also contains code to execute the action list returned by the client policy. The next classes created are the reduced object and object descriptor. As described above, the tool to create the reduced object uses a list of methods to create a class file containing those methods and the instance variables used by those methods. The object descriptor is created by analyzing the byte codes of a Java class to determine how each method uses the instance variables of the object. Using this information, the methods of the reduced object are categorized as read-only, read–write, or server-only.

4.8. Policy configuration and creation

For each cached object class, an application developer creates a configuration object which specifies the consistency policies used for objects of that class. A configuration object is created using

DOI: 10.1002/spe

```
public class NoCacheClientPolicy extends ClientPolicyBase
 public ActionList getEventActionList(PolicyEvent event) {
  switch (event.eventType) {
   case PolicyEvent.METHOD_CALL:
    return new ActionList(new ActionInvokeOnServer());
   case PolicyEvent.SERVER_DONE:
    return new ActionList(new ActionSetReturnValue(event.returnValue));
...

public class NoCacheServerPolicy extends ServerPolicyBase {
  public ActionList getEventActionList(PolicyEvent pe) {
    switch (pe.eventType) {
      case PolicyEvent.SERVER_CALL:
      return new ActionList(new ActionInvokeMethod(),
                            new ActionAddReturnValue());
...
```

Figure 8. Sample *NoCacheClientPolicy* and *NoCacheServerPolicy* classes.

a *ConfigurationWriter* program by specifying the cached object class and its associated consistency policy classes and the reduced object classes. The first policy specified is the default policy used for the object. When more than one policy is specified for the configuration object, the adaptive policy selection mechanism will select between these policies at runtime.

Besides using existing consistency policies, an application developer can create client policy and server policy classes which extend a base class. The developer then adds to each class a *getEventActionList* method which returns an action list. Figure 8 shows the implementation of the *NoCacheClientPolicy* and *NoCacheServerPolicy*. As shown earlier, the client policy returns an action list to invoke the method on the server when a method is called. The server policy returns an action list to invoke the method and to add the return value to the policy event. When the SERVER_DONE event arrives, the client policy creates an action list with the *SetReturnValue* action using the return value contained in the event.

5. EXPERIMENTAL EVALUATION

After implementing the mechanisms described above, we evaluated their effectiveness using RMI applications. Using these applications, we evaluate the overhead of our mechanisms, the benefits of caching, and the benefits of caching with reduced objects. We demonstrate the benefits of adaptive policy selection.

5.1. Benchmark development and experiment methodology

To evaluate our mechanisms, we used two different RMI applications and structured a methodology for conducting our experiments. The first application is a single-client medium size object benchmark

DOI: 10.1002/spe

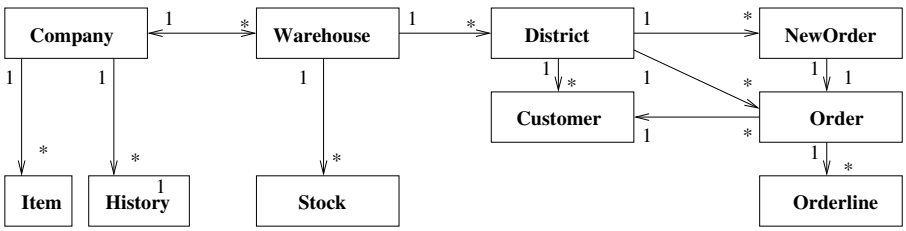

Figure 9. Objects in the rmiBOB benchmark.

similar to the TPC-C benchmark [26]. This application contains a variety of RMI objects with RMI objects passed frequently between the client and the server. This application is used primarily for measuring the benefits of caching and reduced objects. It shows performance benefits for applications where clients do not share objects. The second application is a benchmark with objects representing bank accounts. This application contains a collection of RMI objects, all of which are of the same type, but they are used in a different manner. This application has several clients which access shared and non-shared objects. We use this application to show the performance benefits of adaptive consistency management.

5.1.1. rmiBOB application

To obtain an application with a rich variety of RMI objects, we created rmiBOB from IBM's Business Object Benchmark (BOB) from the book Enterprise Java Performance [27]. This benchmark implements the business logic in the TPC-C benchmark [26]. The benchmark creates a 'company', with associated objects, and measures the performance of executing 'transactions' on the objects.

In developing rmiBOB, we changed the primary objects in the application to be RMI objects. The primary objects in the benchmark are shown in Figure 9. There is one company object that has the items sold by the company as well as a history of payments made by customers. This company has warehouses that stock the items sold by the company. Each warehouse has sales districts, with customers assigned to each sales district. 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 6141 objects, as shown in Table VI.

The benchmark executes five types of 'transactions' 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 for a single client to execute 1000 transactions. To verify that the consistency policies were working correctly, we enabled screen writes for each run and saved the output to a file. At the end of each run, we compared the output to a previous RMI run to ensure that the results were correct.

DOI: 10.1002/spe

Table VI. Number of initial objects.

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

We ran our experiments using two PCs. The server had 1 GB of memory and a 1.6 GHz Pentium®§ processor running Windows®¶ XP and the JDK 1.5 level of Java. The client had 512 MB of memory and a 1.6 GHz Celeron® processor running Linux®‖ 2.4 and the JDK 1.4.2 level of Java. The PCs were connected together using 100 MB Ethernet.

5.1.2. A bank account application

We created a second application in which several clients access a set of bank account objects using *deposit()*, *withdraw()*, and *getBalance()* methods. The benchmark was designed to simulate different object usage patterns by several clients. It does this by creating scripts that dictate the operations to be performed by the clients. The clients run their respective scripts and the server reports statistics about the run.

The script generator for this application uses several parameters to control the behavior of the benchmark. These parameters, and the settings used for our experiments, are shown in Table VII. For our experiments, we varied the number of clients between 2, 4, and 8. Following the 80:20 rule, each client used 80 non-shared (not used by any other client) RMI objects and 20 shared RMI objects. Each client invoked 100 000 method invocations on these RMI objects. For each method invocation, an object is randomly chosen. To emulate locality of access we introduced another parameter in the workload generation to control the probability of accessing an object that was recently accessed. We performed experiments with two values—20% and 80%—for this parameter. To control the sharing of object, we used the values 0, 25, and 50, as the probability of choosing a shared object. We also control the type of method that was invoked by using either 25 or 50% probability of using a *deposit()* or *withdraw()* method.

We conducted our experiments on Sun workstations. The server and three of the clients had 1024 MB of memory and 1 GHz SPARC®** processors. The remaining clients had 512 MB of memory and 502 MHz SPARC® processors.

§Pentium and Celeron are registered trademarks of Intel Corporation in the United States and other countries.
¶Windows is a registered trademark of Microsoft Corporation in the United States and other countries.
‖Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.
**SPARC is a registered trademark of SPARC International, Inc. Products bearing SPARC trademarks are based on an architecture developed by Sun Microsystems, Inc.

DOI: 10.1002/spe

Table VII. Parameters passed to bank account benchmark script generator.

Parameter	Meaning	Settings
Client count	Number of clients accessing RMI objects	2, 4, 8
Shared object count	Number of RMI objects that are accessed by all clients	20
Non-shared object count	Number of RMI objects that are accessed by a single client	80
Client method calls	Number of method calls to RMI objects done by each client	100 000
Locality probability	Probability of accessing a recently used object	20%, 80%
Shared object probability	Probability of a client accessing a shared RMI object. Otherwise, the client will access a non-shared object	0%, 25%, 50%
Write method probability	Probability of the client calling a *deposit()* or *withdraw()* method. Otherwise a *getBalance()* method is called	25%, 50%

5.1.3. *Experiment methodology*

When conducting our experiments, we measured two key metrics. The primary metric was response time. For the rmiBOB application, we measured the average time to execute a transaction. For the banking application, we measured the average time to invoke an RMI method. The second metric was the number of bytes transferred between the clients and the server. We did this by configuring RMI to use a custom socket factory [28] that created a custom socket which recorded the number of bytes transferred.

We made multiple observations of our experiments in order to achieve a confidence level in our results. Using Student's t-distribution, we determined that 10 runs provided a 95% confidence level of obtaining results that were within 5% of the actual mean.

5.2. Overhead of mechanisms

To assure that our caching mechanisms did not add significant overhead, we conducted several experiments using RMI and the no-cache policy. We first conducted experiments to obtain the basic overhead of the mechanisms. We then conducted experiments to gain a detailed understanding of the data transfer and processing overhead.

5.2.1. *Basic overhead*

To observe the basic overhead of our caching mechanisms, we compared the performance of the benchmarks when using RMI and when using mechanisms with the no-cache policy (see Section 4.4.1). Our results are shown in Table VIII. For the rmiBOB benchmark, the response time for using the no-cache policy was about the same as RMI, with 39% fewer bytes transferred. The primary reason for this improvement was the ability to transfer a reference to an RMI object that does not include an RMI stub, as described in Section 4.2. For the bank account benchmark, the overhead of our caching

DOI: 10.1002/spe

Table VIII. Overhead measurements.

		Response time		Bytes transferred	
Benchmark (clients)	Policy	Mean	Overhead	Mean	Overhead
rmiBOB (1)	RMI	115.75 ms		26 062 359	
rmiBOB (1)	No-cache	119.38 ms	3%	15 840 131	−39%
banking (2)	RMI	430.6 µs		13 616 091	
banking (2)	No-cache	494.7 µs	15%	16 757 644	23%
banking (4)	RMI	623.8 µs		27 232 457	
banking (4)	No-cache	787.4 µs	26%	33 514 987	23%
banking (8)	RMI	1186 µs		54 464 057	
banking (8)	No-cache	1511 µs	27%	67 028 150	23%

Table IX. Detailed data transfer overhead for the no-cache policy.

Operation	RMI bytes	NC bytes	Overhead
Object reference retrieval	1125	1190	5.78%
First call	69	240	247%
getBalance calls (499)	33 445	41 507	24.11%
deposit calls (250)	16 756	20 264	20.94%
withdraw calls (250)	16 766	20 260	20.84%
Total	68 155	82 411	22.45%

mechanisms, as measured by increased average response time, was 15% in the two client case, and increased to 27% in the eight client case. The overhead in bytes transferred remained constant at 23%.

5.2.2. *Measurement of data transfer overhead*

To understand the data transfer overhead, we created a smaller version of the bank account benchmark which made 1000 calls to an RMI object. We ran the program using both RMI and the no-cache policy. This application retrieved the object reference as the return value from an RMI call, made one call to the *getBalance* method, then called the *getBalance* method 499 times, the *deposit* method 250 times, and the *withdraw* method 250 times. Table IX shows the number of bytes transferred for the RMI and no-cache policy.

As seen in the table, retrieving the object reference using a cache stub instead of an RMI stub added less than 6% overhead. However, the overhead for the first call was more substantial, requiring an additional 171 bytes to transfer the CCM and client policy objects to the client. The average overhead for the remaining *getBalance*, *deposit*, and *withdraw* calls was an additional 16, 14, and 14 bytes, respectively. This overhead included the cost for transferring an additional policy event parameter on the method call, and for receiving a policy event object as the return value of the method call.

DOI: 10.1002/spe

Table X. Execution and processing times for instrumented banking benchmark.

	getBalance	*withdraw*
RMI stub execution time	401	401
Cache stub execution time	493	475
Cache stub processing time	2	2
CCM execution time	491	473
CCM processing time	22	19
Client policy execution time	4	2
Policy event marshalling time	6	3
SCM stub execution time	459	449
SCM stub processing time	432	429
SCM execution time	27	24
SCM processing time	21	20
Server policy execution time	1	1
Policy event marshalling time	5	3

This low overhead was possible because of our serialization technique which converts a policy event to a string, as described earlier. When we did not use this technique, the overhead for the *getBalance*, *deposit*, and *withdraw* calls was an additional 102, 56, and 56 bytes, respectively.

5.2.3. Measurement of processing overhead

To analyze the processing overhead, we instrumented important classes in our framework and RMI to record the time, in microseconds, when each method was entered and exited. Using the bank account benchmark, we ran the application with just a single client. Compared to the performance of a single client with no instrumentation, the instrumentation added a 4% overhead when using RMI and a 10% overhead when using the no-cache policy.

Using the instrumentation, and repeating the experiment for 10 runs, we measured the average execution time of each component, which is the time between when the component was entered and exited. The actual processing time of each component is the component's execution time minus the execution time of the components it calls. Table X shows our measurements and the relationship between the component execution and processing times. The first two rows show the execution time for the RMI stub and the cache stub. With the instrumented code, the overhead of using our mechanisms was 23% for the *getBalance* calls and 18% for the *withdraw* calls. Examining the cache stub execution time, it consists of the cache stub processing time and the CCM execution time. In turn, the CCM execution time consists of the CCM processing time, the client policy execution time, the policy event marshalling time, and the SCM stub execution time. The SCM stub execution time consists of the SCM stub processing time, which includes the time required for an RMI call, and the SCM execution time. The SCM execution time consists of the SCM processing time, the server policy execution time, and the policy event marshalling time.

When compared to RMI, the *getBalance* call had an overhead of 92 μs. The largest portion of the overhead, 43 μs, was caused by the processing of an action list, as can be seen in the CCM processing

DOI: 10.1002/spe

Table XI. Consistency policies used for caching objects.

Object class	Company	Warehouse	Stock	Customer	OrderLine
Policy	Server-write	Server-write	Server-write	MRSW	Server-write

Table XII. Performance of caching and reduced objects for the rmibob benchmark.

	Response time		Bytes transferred	
Configuration	Mean	Speedup factor	Mean	Reduction factor
Caching	62.7 ms	1.84	9 847 525	2.65
Reduced object	56.1 ms	2.07	7 776 857	3.35

time and the SCM processing time. An addition overhead of 31 μs was caused by the passing of the policy event on the RMI call, as calculated as the difference between the SCM stub processing time and the RMI stub execution time. An additional overhead of 11 μs was encountered when marshalling the policy event. The remaining overhead was caused by the policy objects (5 μs) and cache stub (2 μs).

5.3. Benefits of caching for rmiBOB benchmark

To evaluate the benefits of caching and reduced objects when used with an RMI application having different types of objects, we used the rmiBOB benchmark. By experimentation, we determined the best performing policy settings, shown in Table XI, for each type of object. We then ran the benchmark with and without the use of reduced objects. The results are shown in Table XII. We calculated the speedup factor as the RMI response time divided by the response time with caching enabled. We calculated the communication reduction factor as the RMI communication cost, measured as number of bytes transferred, divided by the cost when caching was enabled. Without using reduced objects, we measured an average transaction cost of 62.7 ms, which is a 1.84 speedup factor from the RMI transaction cost. When reduced objects were used, we measured an average transaction cost of 56.1 ms, which is a 2.07 speedup factor from the RMI transaction cost. The improvement in bytes transferred was much better, with a reduction factor of 2.65 when caching was used. Using reduced objects further improved the reduction factor to 3.35.

5.4. Benefits of caching without object sharing

To understand the best possible caching performance, we conducted experiments, using the bank account application, to determine the benefits of object caching when the objects were not shared. For these experiments, we set the shared object probability, described in Table VII, to zero.

DOI: 10.1002/spe

Table XIII. Performance of bank account benchmark with no shared objects.

		Response time		Bytes transferred	
Clients	Policy	Mean	Speedup factor	Mean	Reduction factor
2	RMI	427.8 μs		13 573 102	
2	MRSW	11.0 μs	39.0	314 554	43.2
4	RMI	621.1 μs		27 145 825	
4	MRSW	11.5 μs	54.0	627 448	43.3
8	RMI	1197 μs		54 291 318	
8	MRSW	21.1 μs	56.8	1 255 355	43.2

The results are shown in Table XIII. The speedup factor ranged from 38.9 to 56.7. The communication reduction factor was about 43.2. As expected, significant performance improvement is possible when caching an object that is not shared by other clients.

5.5. Benefits of adaptive policy switching

Since there are many applications where RMI objects may be shared by different clients, we conducted some experiments using the bank account application to determine the benefits of caching with adaptive policy switching. The adaptive policy switching was configured to initially use the MRSW policy and adaptively switch to/from the no-cache policy. To test different usage patterns, we conducted four sets of experiments. For two sets of experiments, we used a locality setting of 80%. To illustrate the extreme where little locality was present, the remaining two sets used a locality setting of 20%. For each locality setting, we conducted one set of experiments using 25% sharing and 25% write methods and the other set using 50% sharing and 50% write methods. These results are shown in Figures 10–13.

Figure 10 shows the results for 80% locality, 25% sharing, and 25% write. Because of the high locality and low sharing, the MRSW policy performed very well, with a speedup factor ranging from 5.7 to 2.7. Because the MRSW policy performed so well, the adaptive policy mechanisms did not switch the policy. As shown in the figure, the response time overhead of enabling the adaptive mechanism was very small. The communication cost of enabling the adaptive mechanism was larger, yet the cost was still much less than the RMI cost.

Figure 11 shows the results for 80% locality, 50% sharing, and 50% write. Because of the higher sharing, the MRSW policy performed worse than with 25% sharing. In the worst case, the speedup factor for MRSW was only 1.08. In the four client case, the adaptive mechanism caused a slight degradation in response time when compared to MRSW. However, in the eight client case, the adaptive mechanism performed very well, with a speedup factor of 1.40.

Figure 12 shows the results for 20% locality, 25% sharing, and 25% write. In this case, the MRSW policy performed worse than the 80% locality case. On the other hand, the adaptive mechanism performed well, resulting in improvements over MRSW for the 4 and 8 client cases. For these cases, the adaptive mechanism achieved speedup factors of 2.70 and 2.67.

DOI: 10.1002/spe

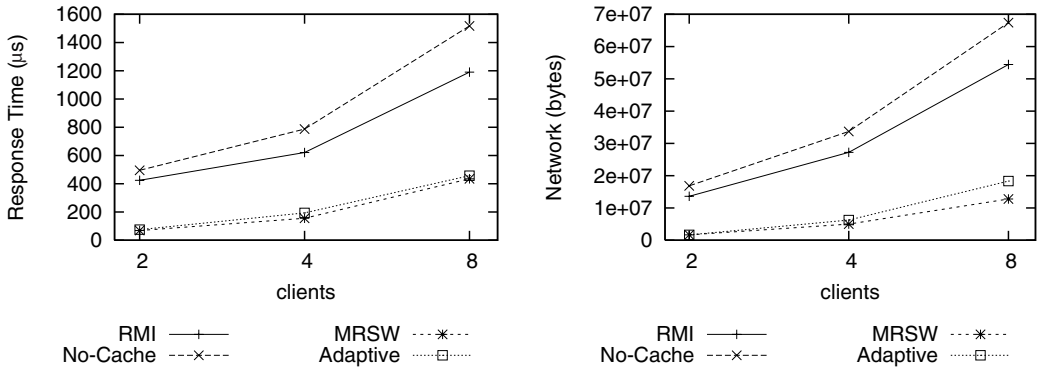

Figure 10. Performance with 25% sharing, 25% write, and 80% locality.

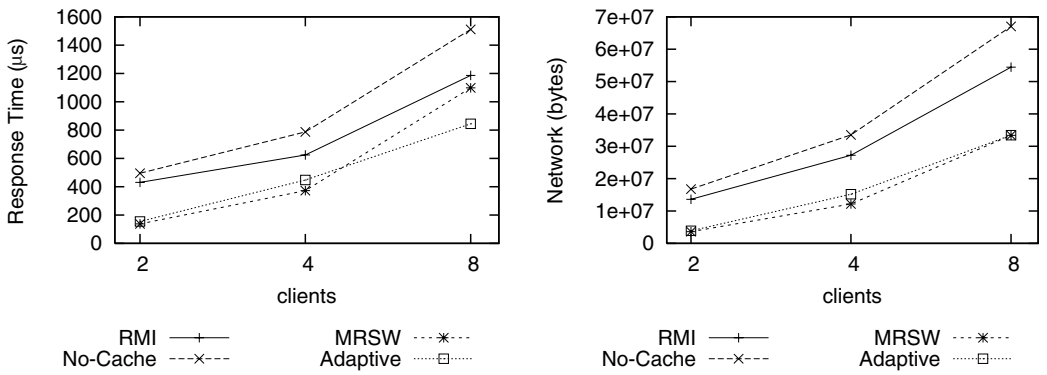

Figure 11. Performance with 50% sharing, 50% write, and 80% locality.

Figure 13 shows the results for 20% locality, 50% sharing, and 50% write. In this case, MRSW performed very poorly. For 4 and 8 clients, the response time was 13% and 63% worse than RMI. In those cases, the adaptive mechanism permitted the caching mechanism to still outperform RMI, with speedup factors of 1.42 and 1.37 in the 4 and 8 client cases.

These experiments illustrate the importance of adaptive policy switching. In general, adaptive policy switching had acceptable overhead when the MRSW policy was being used. When the MRSW policy performed poorly for an object, the adaptive policy switching caused the no-cache policy to be used instead, resulting in better application performance.

DOI: 10.1002/spe

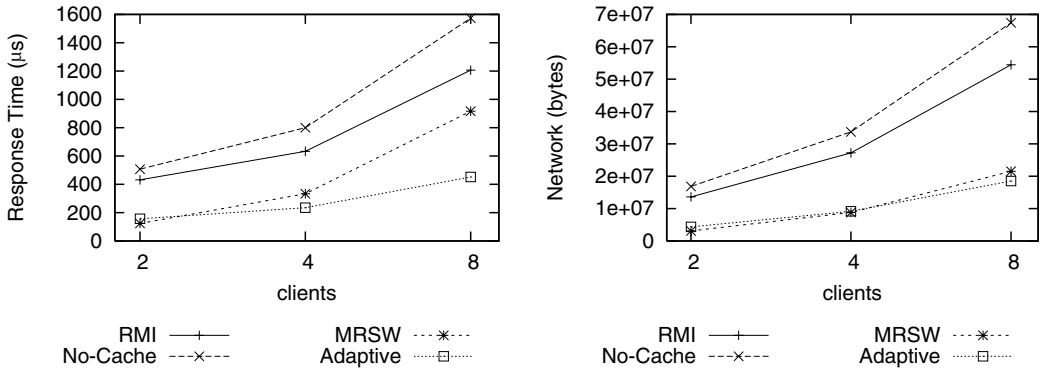

Figure 12. Performance with 25% sharing, 25% write, and 20% locality.

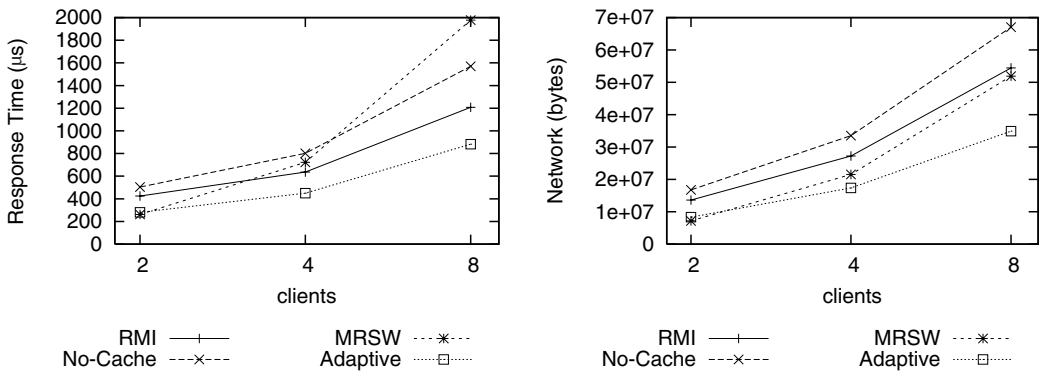

Figure 13. Performance with 50% sharing, 50% write, and 20% locality.

6. CONCLUSION

This paper presented mechanisms for caching RMI objects. These mechanisms are compatible with RMI and transparent to existing RMI clients. These mechanisms ensure the consistency of a cached RMI object using an event-based consistency model. Using this consistency model, we have illustrated the design and implementation of several consistency policies. Moreover, we have also created adaptive mechanisms that use a cost model to adaptively select a consistency policy best suited to the current usage of the object. Our mechanisms also permit the use of reduced objects, which reduces both the amount of data that is cached by clients and the communication overhead of caching the object.

In this paper, we illustrated the performance benefits of the caching mechanisms using RMI applications. When using an RMI application with a single type of object, but with different usage

DOI: 10.1002/spe

patterns, the response time speedup factor was 56.7 in the best case. In the worst case, where 50% of method invocations accessed shared objects and locality was only 20%, adaptive policy switching still resulted in a speedup factor of 1.37. When using an RMI application containing several types of RMI objects, the mechanisms improved the response time of the application by a speedup factor of 2.07 when both caching and reduced objects were used.

These mechanisms provide the base for our continuing research in distributed object caching. While the consistency policies presented here do not support disconnected operation, our framework provides enough flexibility to create consistency policies for use in a mobile environment. In particular, we are utilizing these mechanisms to create consistency policies that utilize the semantics of object methods to ensure the consistency of cached objects. Our results for these semantics-based consistency policies are not discussed here, but a preliminary report is available in [29].

ACKNOWLEDGEMENTS

We wish to thank the anonymous reviewers for their insightful comments that have improved the quality of this paper. This work would not have been possible without IBM's degree work study plan's support of the primary author's PhD program.

REFERENCES

1. Sun Microsystems. Java remote method invocation specification, 2003. http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html [17 August 2006].
2. Ghezzi C, Martena V, Picco GP. Enhancing remote method invocation through type-based static analysis. *Proceedings of the 7th International Conference on Fundamental Approaches to Software Engineering (FASE04)*, Barcelona, March 2004. Springer: Berlin, 2004; 339–353.
3. Maassen J, van Nieuwpoort R, Veldema R, Bal H, Kielmann T, Jacobs C, Hofman R. Efficient Java RMI for parallel programming. *ACM Transactions on Programming Languages and Systems* 2001; **23**(6):747–775.
4. Veldema R, Philippsen M. Compiler optimized remote method invocation. *Proceedings of the 5th IEEE Conference on Cluster Computing*, Hong Kong, 2003. IEEE Computer Society Press: Los Alamitos, CA, 2003; 127–136.
5. Krishnaswamy V, Walther D, Bhola S, Bommaiah E, Riley G, Topol B, Ahamad M. Efficient implementation of Java remote method invocation (RMI). *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, Santa Fe, April 1998. USENIX: Berkeley, CA, 1998.
6. Krishnaswamy V, Ganev IB, Dharap JM, Ahamad M. Distributed object implementations for interactive application. *Proceedings of the Middleware 2000 Conference*, April 2000. Springer: Berlin, 2000; 45–70.
7. Aridor Y, Factor M, Teperman A, Eliam T, Schuster A. A high performance cluster JVM presenting a pure single system image. *Proceedings of the ACM of 2000 Java Grande Conference*, San Francisco, CA, 2000. ACM Press: New York, 2000; 168–177.
8. Lipkind I, Pechtchanski I, Karamcheti V. Object views: Language support for intelligent object caching in parallel and distributed computations. *Proceedings of the 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Denver, CO, November 1999. ACM Press: New York, 1999.
9. Bakker A, Kuz I, van Steen M, Tanenbaum AS, Verkaik P. Design and implementation of the Globe middleware. *Technical Report IR-CS-003*, Department of Computer Science, Faculty of Sciences, Vrije Universiteit Amsterdam, June 2003.
10. Bakker A, van Steen M, Tanenbaum AS. From remote objects to physically distributed objects. *Proceedings of the 7th IEEE Workshop on Future Trends of Distributed Computing Systems*, Cape Town, South Africa, December 1999. IEEE Computer Society Press: Los Alamitos, CA, 1999.
11. van Steen M, Homburg P, Tanenbaum AS. Globe: A wide-area distributed system. *IEEE Concurrency* 1999; **7**(1):70–78.
12. Joseph AD, deLespinasse AF, Tauber JA, Gifford DK, Frans Kaashoek M. Rover: A toolkit for mobile information access. *Proceedings of the 15th Symposium on Operating Systems Principles*, Copper Mountain Resort, CO, December 1995. ACM Press: New York, 1995; 156–171.

DOI: 10.1002/spe

13. Shapiro M. Structure and encapsulation in distributed systems: The proxy principle. *Proceedings of the 6th IEEE International Conference on Distributed Computer Systems (ICDCS)*, Boston, MA, May 1985. IEEE Computer Society Press: Los Alamitos, CA, 1985; 198–204.
14. Sun Microsystems. Dynamic proxy classes, 2003. http://java.sun.com/j2se/1.4.2/docs/guide/reflection/proxy.html [17 August 2006].
15. Hassoun Y, Johnson R, Counsell S. Applications of dynamic proxies in distributed environments. *Software—Practice and Experience* 2005; **35**(1):75–99.
16. van Heiningen W, Brecht T, MacDonald S. Exploiting dynamic proxies in middleware for distributed, parallel, and mobile Java applications. *Proceedings of the 8th International Workshop on Java for Parallel and Distributed Computing*, Rhodes Island, Greece, April 2006. IEEE Computer Society Press: Los Alamitos, CA, 2006.
17. Garbinato B, Guerraoui R, Masouni KR. Implementation of the GARF replicated objects platform. *Distributed Systems Engineering Journal* 1995; **2**(1):14–27.
18. Liskov B, Castro M, Shrira L, Adya A. Providing persistent objects in distributed systems. *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP'99)*, Lisbon, Portugal, June 1999. Springer: Berlin, 1999; 230–257.
19. Aldrich J, Chambers C, Sirer EG, Eggers S. Static analyses for eliminating unnecessary synchronization from Java programs. *Proceedings of the 6th International Static Analysis Symposium*, Venezia, Italy, September 1999. Springer: Berlin, 1999; 19–38.
20. Bogda J, Hölzle U. Removing unnecessary synchronization in Java. *Proceedings of the 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Denver, CO, November 1999. ACM Press: New York, 1999.
21. Krintz C, Calder B, Hölzle U. Reducing transfer delay using Java class file splitting and prefetching. *Proceedings of the 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Denver, CO, November 1999. ACM Press: New York, 1999.
22. Jakarta Byte Code Engineering Library. http://jakarta.apache.org/bcel [17 August 2006].
23. Eberhard J, Tripathi A. Efficient object caching for distributed Java RMI applications. *Proceedings of Middleware 2001 Conference*, Heidelberg, Germany, November 2001. Springer: Berlin, 2001; 15–35.
24. Mazouni KR, Garbinato B, Guerraoui R. Building reliable client-server software using actively replicated objects. *Proceedings of the International Conference on Technology of Object Oriented Languages and Systems (TOOLS)*, Versailles, France, March 1995. Prentice-Hall: Englewood Cliffs, NJ, 1995.
25. Maassen J, Kielmann T, Bal HE. Parallel application experience with replicated method invocation. *Concurrency and Computation: Practice and Experience* 2001; **13**(8–9):681–712.
26. Transaction Processing Performance Council. TPC benchmark C, 2004. http://www.tpc.org/tpcc [17 August 2006].
27. Halter SL, Munroe SJ. *Enterprise Java Performance*. Prentice-Hall: Englewood Cliffs, NJ, 2000.
28. SUN Microsystems. Using a custom RMI socket factory, 2003. http://java.sun.com/j2se/1.4.2/docs/guide/rmi/socketfactory [17 August 2006].
29. Eberhard J, Tripathi A. Object-based commutativity analysis for real-time applications. *Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, Sedonia, AZ, February 2005. IEEE Computer Society Press: Los Alamitos, CA, 2005; 279–286.

DOI: 10.1002/spe