Efficient Implementations of Java Remote Method Invocation (RMI) *

Vijaykumar Krishnaswamy   Dan Walther   Sumeer Bhola
Ethendranath Bommaiah   George Riley   Brad Topol
Mustaque Ahamad†

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332

Abstract

Java and the Remote Method Invocation (RMI) mechanism supported by it make it easy to build distributed applications and services in a heterogeneous environment. When the applications are interactive and require low response time, efficient implementations of RMI are needed. We explore both transport level protocols as well as object caching in the RMI framework to meet the performance requirements of interactive applications. We have developed a prototype system that offers new transport protocols and allows objects to be cached at client nodes. We describe the design issues and the implementation choices made in the prototype along with some preliminary performance results.

1 Introduction

Interactive applications that enable widely distributed users to cooperate over the Internet will become increasingly common in the future. Such applications have traditionally been explored in the area of groupware but as increased bandwidths become available into the home (e.g., cable and digital subscriber line (xDSL) networks), electronic commerce and entertainment applications will become interactive. For example, a consumer located at home can utilize a graphical user interface (GUI) to view various retail items he or she is interested in purchasing. Simultaneously, a sales associate located at the retail outlet may also have a copy of the GUI which permits the associate to see what the customer is selecting and may suggest alternatives which are then presented in the customer's GUI. Furthermore, the home consumer may have requested friends located at other homes to also participate in this decision making and therefore they too may be running a GUI and viewing the possibilities and also making suggestions. Many such interactive application scenarios can be constructed easily.

Interactive applications will be supported by shared distributed services. In order for the internetworked computing infrastructure to support the above application scenario, system support is needed to allow the distributed services and client applications to be programmed easily. The use of object technology is becoming an increasingly popular approach for implementing distributed services. This is due to the fact that object technology provides a uniform mechanism for accessing local and remote resources and reduces the complexity of building applications in an internetworked computing environment. The Java language is a popular foundation for building distributed services and applications because it hides the problems that

*This work was supported in part by NSF grants CDA-9501637 and CCR-9619371, and industrial partners of the Broadband Telecom Center, Georgia Institute of Technology.

†Contact author: Email and home page – mustaq@cc.gatech.edu, http://www.cc.gatech.edu/fac/Mustaque.Ahamad/.

arise due to heterogeneity of server and client hardware and software platforms. Remote Method Invocation (RMI) is Java's mechanism for supporting distributed object based computing [18]. RMI allows client/server based distributed applications to be developed easily because a client application running in a Java virtual machine at one node can invoke objects implemented by a remote Java virtual machine (e.g., a remote service) the same way as local objects.

Although RMI enhances the ease of programming for distributed applications, we have found that it does result in significant performance penalties for applications compared to message passing [10]. Such loss of performance is undesirable for interactive applications in a wide-area environment because of the need for interactive response time in the presence of high communication latencies. The additional processing required by RMI will add some overhead compared to message passing but there are a number of techniques that can exploit the communication structure embodied by RMI to provide better performance. For example, it may be possible to exploit the "invocation-response" nature of RMI communications to develop a more efficient communication protocol than the TCP protocol that is employed by RMI (such an approach was used in the implementation of remote procedure call or RPC [2] which is closely related to RMI). Furthermore, when possible, a client may be able to cache the state of remote objects and invoke them locally. In this case, the overhead associated with communication can be avoided when there is significant locality of access.

We explore a number of techniques to improve RMI performance and integrate them into the RMI framework. Since the performance of RMI depends on the underlying communication protocols, we first explore a number of alternate transports that may improve the performance of RMI implementations. We developed a user datagram protocol (UDP) based reliable message delivery protocol that exploits the request-response nature of RMI communications. Also, when object state is cached at client nodes, consistency of the replicated object copies has to be maintained. Consistency protocols for replicated objects can benefit from one-to-many (e.g., multicast) communication and we have developed a flexible multicast transport that is available to RMI implementation. Finally, we extend the reference layer in the RMI framework to cache objects at client nodes. This approach allows clients to transparently invoke remote objects independent of whether they are being cached. When a cached copy of an invoked object is available, the invocation is executed locally. An invalidation based protocol has been implemented to maintain consistency of the cached copies. All this support has been added to the RMI framework by extending interfaces that are provided in the framework. The prototype system we have implemented has allowed us to quantify the benefits of caching.

We briefly review the RMI framework in Section 2. This framework primarily consists of the transport layer and the reference layer. The transport layer provides interfaces for communication protocols that support message passing across sites. Section 3 describes the new protocols that have been added by us to the transport layer. These include a UDP based reliable message delivery protocol and a multicast protocol that is used in maintaining the consistency of cached object copies. We explore design issues for object caching in the RMI reference layer and discuss our implementation is Section 4. Performance studies and their discussion is presented in Sections 5 and 6. We describe related work and conclude the paper in Sections 7 and 8.

2 The Java RMI Framework

The RMI framework [8] in Java allows distributed application components to communicate via remote object invocations. In particular, a client running at one node can access a remote service by invoking a method of the object that implements the service. Thus, the RMI framework enables applications to exploit distributed object technology rather than low level message passing (e.g., sockets) to meet their communication needs. A high level architecture of the RMI framework is shown in Figure 1.

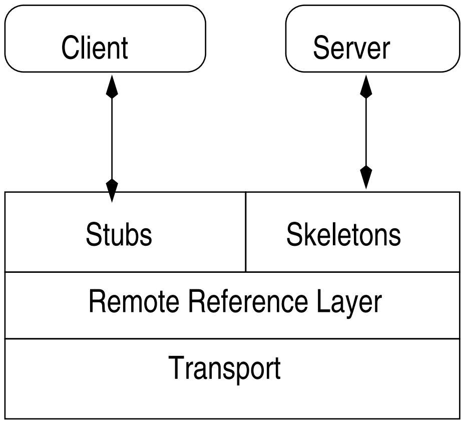

Figure 1: RMI Framework

All objects that can be invoked remotely must implement the interface `Remote`. This interface is just a tag that is used to distinguish remote objects from normal objects. Remote objects implement one or more interfaces and only through these interfaces are they visible to the outside world. The **rmic** tool is used to generate the skeleton and stub classes for a given remote object interface. For a given remote object `impl`, the stub, `impl_Stub`, and skeleton, `impl_Skel`, have the same set of methods that are defined in the interface of `impl`. `impl` provides the actual implementations of the methods defined in its interface.

A server that wants to make an object implemented by it remotely invokable must first *export* the object. This results in the instantiation of the skeleton object, the stub object, the reference object and the transport endpoint object in the server's virtual machine. When this object is bound onto a name server (e.g., **rmiregistry**) via the *bind* or the *rebind* operation, the stub, the client side reference object and the transport object are serialized and moved into the name server. At this point, the object is available for remote invocations from client nodes. To invoke a remote object `impl`, a client must first obtain a reference for it. Such a reference can be obtained in one of two ways. A client can do a *lookup* on the object which results in the instantiation of `impl_Stub` and other needed reference layer objects in the client's VM. All subsequent method invocations by the client to the remote object are routed via these objects. The client can also receive a reference to the object as an argument of an invocation. At the time the object reference is unmarshaled, the `impl_Stub` and other related objects are instantiated to enable the client to remotely invoke `impl`.

A client making an invocation on a remote object actually makes the call to the `stub` object. The *remote reference layer* is responsible for carrying out the invocation. The *transport layer* is responsible for connection setup, connection management and keeping track of and dispatching to remote objects. The `skeleton` for a remote object makes an upcall to the remote object implementation when a request for remote invocation is received at the server VM. Once the invocation is executed, the return value is sent back to the client via the skeleton, remote reference layer and transport layer on the server side, and then up through the transport and remote reference layers, and stub object on the client side. If an exception is thrown while making a call on the server side, this exception object rather than the result is marshaled and sent back to the client. The client side has enough machinery to detect that the received result is actually an exception rather than the result of the call and throws the corresponding exception to the application. A distributed garbage collector running in the server VM keeps track of client references for the remote object `impl`. In particular, client nodes *lease* `impl` for a certain period of time and each client reference increments its reference count by one. If a client's lease ends, the reference gets decremented and when this reference count becomes zero, `impl` can be garbage collected.

3 Efficient Communication Support

Although the RMI transport layer is flexible enough to include several transport protocols, at the time we started this work, only the Transmission Control Protocol (TCP) was available for RMI related communication. The more efficient User Datagram Protocol (UDP) cannot directly be used since it does not guarantee reliable delivery of invocation request and response messages. We take an approach that implements a reliable message delivery protocol based on UDP. We call this protocol R-UDP. Since RMI communications follow a "request-response" pattern, it is possible to exploit this structure in R-UDP to efficiently implement the reliable delivery of messages. In addition to R-UDP, we provide a flexible multicast protocol that can be used by the RMI reference layer when object caching is employed. These two protocols are described in this section. The impact of these protocols on the performance of remote invocations is discussed in a later section.

3.1 R-UDP: UDP based Reliable Protocol

We believe the use of a protocol like TCP for all phases of the RMI communication activity leads to certain inefficiencies. In particular, during the actual remote object invocation phase (after a given object has been located on the remote host and all necessary initialization has been performed), the data flow would typically fall into a "request-response" model, with the client sending a single "request" to the server, followed always by the server sending a single "reply" back to the client. Given this, any explicit acknowledgments used by TCP for requests can be avoided in a reliable protocol that is aware of the structure of RMI communication. A similar argument was used in the implementation of remote procedure calls by Birrell and Nelson in [2]. Another area where we expect to gain some performance improvements is by having explicit control over the behavior of the transport layer, specifically in the buffering and sending of network packets, rather than allowing the underlying protocol to make decisions about when to buffer data and when it is time to send a network packet.

3.1.1 Implementation Details

For communication between a client and server running at different sites, RMI allows for the specification of a "Socket Factory" by both the client and server. Thus, the default classes `Java.net.Socket` and `Java.net.ServerSocket` do not need to be used. We designed and implemented `RMISocket` and `RMIServerSocket` classes as subclasses of `Socket` and `ServerSocket` respectively. Since they are subclassed from the standard TCP socket classes, they must mimic their functionality. These new classes will allow a client/server pair to choose the R-UDP protocol for reliable message delivery on a *setSocketFactory* call. All other socket related processing in the RMI implementation is unchanged. The implementation of R-UDP can be broken down into two activities, (1) connection setup, and (2) reliable sending and receiving of data.

Connection Setup: During the connection setup phase, the server side *accept* method is simply blocked on a receive on the specified well known port address. A client wishing to connect to the server creates a local socket bound to a transient port, assigns a random 64 bit sequence number, and forwards the sequence number and local port number to the server in a datagram marked as a "Connection Request". Upon receipt of the connection request packet, the server creates a local socket bound to a transient port, assigns its own random 64 bit sequence number, and returns the sequence number and local port number to the client in a datagram marked as a "Connection Acknowledgment". When the client receives this packet, the connection is established. Of course, the Connection Request/Connection Ack sequence must be timed out and retransmitted in the event of errors.

Reliable Data Transfer: When either the client or server sends data, the normal `Java.net.Socket` paradigm is used, namely the use of `getDataOutputStream` and the writing of stream data to the returned output stream. Our implementation returns an output stream object of our design, which is a subclass of `ByteArrayOutputStream`. Our stream simply places all data written into a byte array buffer until a call to *flush* is made on the stream. When the flush call is made, the array is passed to a separate thread (the "SendingThread") to be transmitted to the peer. The sending thread is blocked on a Java *wait* call until something is available to be sent, and is started by a *notify* call by the flush method. The data to be sent is placed in a datagram along with the next sequence number, an implicit acknowledgment sequence number (discussed later), and the actual data. The datagram is sent to the peer, marked as "Data packet, no acknowledgment required". The sending thread then blocks on a *wait* until an implicit acknowledgment is received (discussed next) or a constant timeout period has elapsed. If the timeout period elapses without receipt of an implicit acknowledgment, the sending thread re-sends the packet, but the second (and subsequent) tries are marked as "Data packet, explicit ack requested". As previously mentioned, all data transmissions to a peer include an "implicit ack", which notifies the peer of the highest sequence number packet that has been received. This allows for a server "reply" packet to serve as the acknowledgment that a client "request" packet has been received. This works well in the "request-response" data transmission model used for remote object invocations.

A host wanting to receive data from a peer uses the normal paradigm of `getInputDataStream` and reading stream data from the returned object. Our implementation returns an input stream object of our creation, which is a subclass of `ByteArrayInputStream`, and which is managed by a separate "Receiving" thread. The receiving thread is blocked on a datagram receive call, and will fill data in the byte array based on the contents of the received datagram. If an explicit acknowledgment is requested by the peer, an acknowledgment packet is prepared and returned, otherwise the thread just blocks waiting for the next message.

In the interest of brevity, the above discussion glosses over or ignores completely many of the details of a good implementation for reliable data transmission, such as the recognition and processing of duplicate data blocks. By no means is our implementation a fully functional TCP implementation, but is adequate for our needs in testing remote objects.

3.2 Multicast Communication

The RMI design is flexible enough to add server replication for improved scalability and fault-tolerance. We also explore object caching at client nodes to avoid the network latency when there is locality of access. Consistency protocols need to be employed when multiple copies of objects exist either due to replication or caching. Such protocols can benefit from multicast communication. By using multicast as against multiple unicast channels, we stand to gain in terms of better usage of network and server resources. For example, if an invalidation protocol is used to maintain consistency of replicated object copies, it is clearly beneficial to deliver the invalidation request to all the clients using a multicast message. However, we note that the scalability attainable can be limited by the consistency protocols even when multicast is used. In the case of invalidation protocols, for example, if the protocol requires responses from every client caching the object, then the scalability levels attainable are limited (we are exploring other consistency protocols that do not suffer from this problem).

We have implemented a reliable multicast framework along the lines of SRM [6], with a few novel changes. Like SRM, we use application data unit framing, negative acknowledgments, and multicast the retransmit request and response messages to the whole group. However, while SRM aims at *eventually* delivering *all* messages sent to the group, we aim at *eventually* delivering only the *essential* messages to all the members of the group. This is motivated from the fact that the multicast facility is intended to be used primarily by consistency related messages. Thus, we can rely on hints from the consistency protocol in identifying the *essential* messages. For example, if successive multicast messages update the state of the cached objects, the consistency protocol might permit loss of earlier updates as long as newer updates are delivered, that is, a newer update makes an earlier update *inessential*. We do not expend resources towards reliably delivering messages that have been identified as *inessential*. We believe that we stand to benefit significantly from the above relaxed definition of reliability if the fraction of messages identified as *inessential* is reasonably high.

As we mentioned above, a retransmit request for a missed packet is sent to the whole group, and every member which can service this request locally enqueues a response with a random timer associated with it. The response is eventually sent out by the member whose timer expires the earliest. One of the main disadvantages of this scheme is that every member is required to participate in servicing retransmission requests. We propose to provide an option to permit the usage of a separate group address for multicasting retransmission requests and responses [12].

We have implemented the multicast protocol and used it to send invalidation messages in the consistency protocol that has been implemented in the prototype. The performance improvements made possible by multicast communication are discussed in Section 5.

4 Object Caching in the RMI Framework

Caching of remote objects has been shown to lead to better performance in systems that range from file systems to distributed shared memories. Clearly, if there is locality of access, caching a remote object at the client site can improve application performance because methods invoked on the object can be executed locally. In the RMI framework, the reference layer, which comes between the stub/skeleton objects and the transport layer, is responsible for handling remote method invocations. Thus, the reference layer is the natural place for providing alternative implementations of remote method invocations (e.g., using caching). We first discuss the design issues related to object caching at the reference layer and then present implementation details of a prototype system that we have developed for object caching. Our design of caching in the RMI framework was motivated by the following requirements.

1. The decision on whether an object is cacheable or not should be decided by the object provider at runtime, and not at compile time. This allows a single implementation of the object's functionality to be easily reused in different scenarios. This is all the more important in Java, because only single inheritance is available.

2. Cacheable objects should coexist with uncacheable (UnicastRemoteObjects) objects. This means that cacheable objects can have references to uncacheable objects and vice versa.

3. Caching should be transparent to the client[1], i.e. a client should not treat cacheable and uncacheable objects differently. Also, the invocation, failure and garbage collection semantics should be as close to uncacheable objects as possible.

We first describe an abstract model of RMI, and then show how caching can be added to it.

4.1 Abstract RMI

Based on the discussion in Section 2, we have presented a model of a non-caching RMI in Figure 2(a). The server P_s, first creates the server object O. It then exports O using a certain reference layer which creates the other four objects at P_s: (1) C, the client stub, (2) S, the server skeleton, (3) Ref_c, the object that implements the functionality of the client side reference and transport layer, and (4) Ref_s, the object that implements the server side reference and transport layer functionality. Thus, we have combined the reference and transport layer functionality into a single object. The server then binds C to a name server, which results in the marshaling[2] of the C and Ref_c state and unmarshaling at a name server. A lookup request of a client will be sent to the name server which will

[1] There are obviously situations where a client doesn't want to cache, for example, due to local memory limitations. These policies can be expressed in a separate policy object at the client and do not need to be part of the main control flow.

[2] Referred to as serialization in Java.

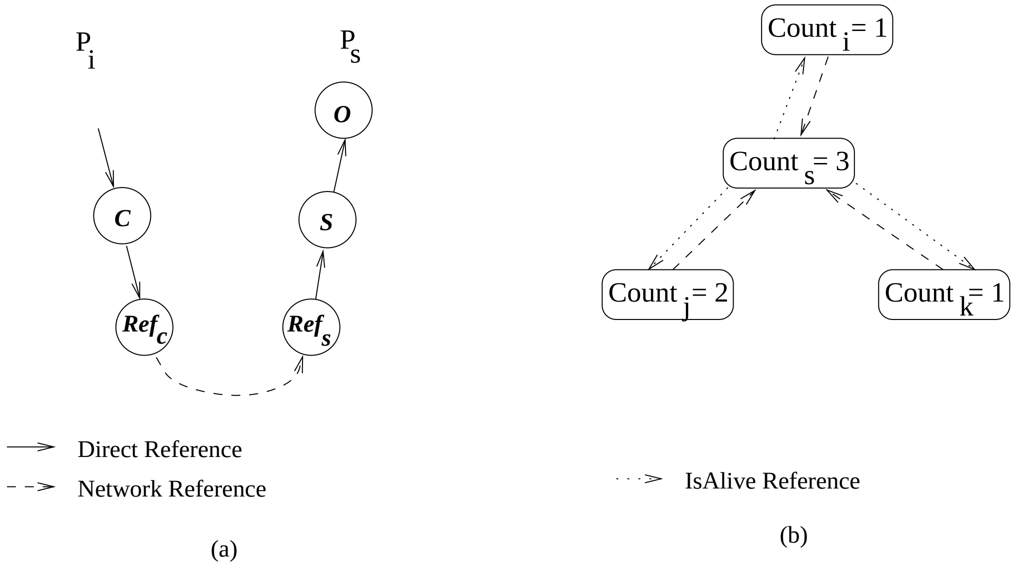

Figure 2: A Model of Non-caching RMI

send the C and Ref_c objects to the client (say P_i) in a similar manner. These objects and the references they hold across them are shown in Figure 2(a) after the lookup operation has been completed. Figure 2(b) shows the reference counting mechanism used for garbage collection. The count at the server, $count_s$, is the number of clients that have a network reference to object O. Each client (say P_i) also maintains a count of the number of references it has to O. P_i creates $count_i$ when it first gets a reference to O. At that time, it also informs the server, which increments $count_s$ and creates an *IsAlive* reference to P_i. If P_i creates more references to O (e.g., by cloning), it only increments $count_i$. When the value of $count_i$ drops to zero, the server is informed, which decrements $count_s$. O can be garbage collected when $count_s$ is zero and there are no local references to O. Note that the *IsAlive* reference is used by the server to detect that a client has crashed, so that $count_s$ can be decremented. This prevents garbage collection from being stalled because some client failed without informing the server.

4.2 Adding Caching

To add caching to this framework, we provide a different reference layer $Cref$, with $Cref_s$ being the server side, $Cref_c$ the client side, and $Cref'_s$ the client caching layer. The creation of the server object, export, binding and lookup are still the same (except for a different reference layer). Figure 3(a) shows the scenario after a lookup has been done at P_i. As there are cases when a process may have a remote reference but may never make an invocation on it (for example, a name server), we only initiate caching at the first invocation. Figure 3(b) shows the scenario after the first invocation. O' is the copy of O cached at P_i. The main differences between Ref_c and $Cref_c$ are

- Initiate caching on first invocation, instantiate cached object, and redirect reference to the cached copy.
- Send every invocation to the cached copy using the direct reference.
- When marshaled (for example, when passed as a parameter to some other remote invocation), do not marshal the direct reference subgraph. This limits

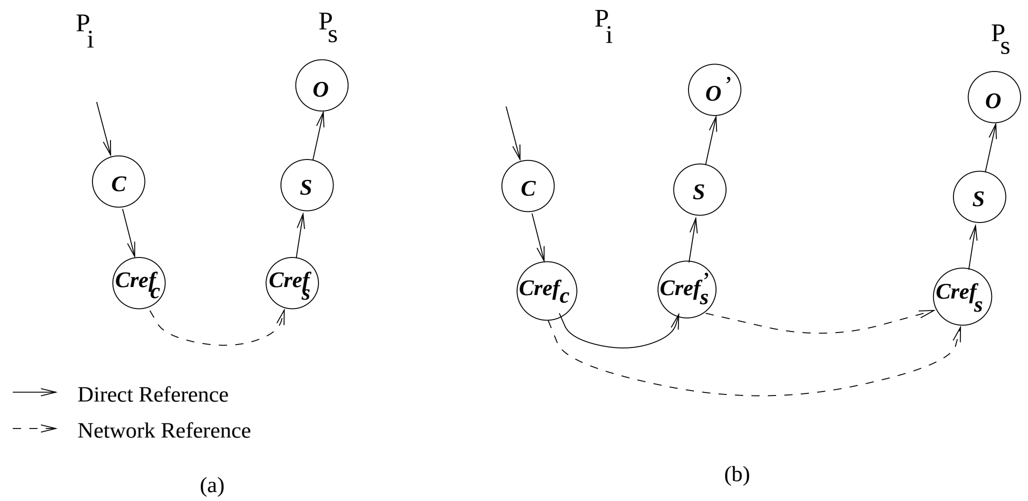

Figure 3: RMI with Caching

the architecture to a two level tree with $Cref_s$ at the root and $Cref'_s$s and $Cref_c$s as the leaves. We can also allow a policy where some $Cref_c$s never initiate caching i.e. their invocations always go to the $Cref_s$.

The job of maintaining the consistency of the cached object copy resides with $Cref_s$ and $Cref'_s$s.

Garbage Collection: A natural question to ask is how does caching effect RMI garbage collection. The answer is, it does not. As a $Cref'_s$ only exists in a virtual machine along with its $Cref_c$, we can still utilize the old system of counting network references from $Cref_c$s to $Cref_s$.

Specifying read and write methods: To initiate a consistency action, $Cref'_s$ needs to know whether a method invocation will only read the object state, or will also write it. We allow object programmers to provide information that can be used to infer if a method's execution only reads the object state or the state is also updated. In particular, the method code should include the throwing of read and write exceptions depending on how the state of the object is accessed by the method. Information about such exceptions is available from the class meta-data that is created by the Java compiler. We extended **rmic** to consult this meta-data to generate another object, `impl_MemFuncStat`, along with the `impl_stub` and `impl_skel` objects. `impl_MemFuncStat` implements a member function *isWriteMethod* that returns whether a certain method reads or modifies the state of `impl`.

Failure Semantics: We consider two types of failures, server, and client. In case of server failure, both noncaching RMI and our caching extension behave the same way, they stop working. Client failures in noncaching RMI don't lead to any problems (except for the GC mechanism detecting the failure and decrementing the counter). For caching clients in our caching RMI, the situation is somewhat complicated and depends on the cache consistency protocol. For an invalidation protocol, the crash of a client which did not have the only valid copy is easy to handle. The problem is when a client with the only valid copy crashes. A simple solution is for the server to use the last version it has as the valid copy and continue from there. This is perfectly acceptable if any updates done by the crashed client which were lost (and updates done by the same client after those lost updates), did not effect some part of the world which can still be seen by the remaining clients. For example, assume that client P_1 updates a cached copy of object O_1 (whose server is at P_1), then updates remote object O_2 which resides at P_2, and then crashes. The update on O_2 is visible but the previous update on O_1 might have been lost because the new O_1 value was not yet sent to the server.

4.3 Implementation of Object Caching

The functionality provided by the $Cref_c$, $Cref_s$ and $Cref_s{}'$ objects shown in Figure 3 has to address a number of problems. First, to cache an object at a client site, the object state and implementation have to be made available to the client. The serialization interface provided by Java is used for transporting object state across nodes. If an object has state that is meaningful only at the server node where the object is instantiated (e.g., open network connections), new serialization methods are allowed by our system that override the default Java serialization methods. The implementation of an object is in the form of the bytecode which can be transferred to client nodes (stub bytecode is already transferred from server to client node in Java RMI). The client side is initially provided minimal code and whenever the system faults on the bytecode, a central code base specified by the server side is contacted and the necessary bytecode is downloaded on demand.

The execution of a method with a cached copy either only reads the object state or it also modifies the object state. The consistency protocol actions that need to be executed depend on whether the method will read or update the object state. We use the the `impl_MemFuncStat` object described earlier for object `impl` to determine the access type.

We employ the standard invalidation protocol to maintain consistency of cached object copies. Thus, when a client invokes a method of an object that can update the object state, the client communicates with the server node. The server keeps track of the clients that have copies of the object and sends them invalidation messages. Once copies at other clients are invalidated, the client that updates the object state is allowed to execute the method with the cached object copy.

We considered the following two approaches for designing a consistency framework that implements the invalidation protocol as well as other consistency protocols.

1. The implementation of a cacheable object extends a consistency object whereby it inherits all the methods of the consistency object which are invoked to maintain the object's consistency.

2. The caching framework maintains a reference for a consistency object and all invocations on the implementation get monitored by this consistency object.

The first approach means that the consistency protocols for a cached object are decided at compile time. The second approach not only allows us to dynamically link an object with a consistency protocol at runtime, it also allows for object caching to be enabled or disabled during the life of the object. Further, consistency levels and hence protocols can be changed depending on the degree of coupling required among the clients. Since the second approach provides more flexibility, we decided to use it in our implementation of object caching. All the consistency objects are derived from a base object called `ConsistencyModel` which has a generic set of methods that are common to all consistency objects. A particular consistency protocol (e.g., server initiated invalidations) is implemented by a specialized object that extends the ConsistencyModel object. Instantiation of this consistency object during the deserialization of the client side caching framework also forks a consistency thread which performs all the consistency actions on the object in a synchronized manner with the application thread using the object.

4.3.1 The Caching Framework

Some of the objects that make up the caching framework on the server and client sides are shown in Figures 4 and 5. On the server side, the reference object, `CacheableServerRef`, maintains state information so as to instantiate an object either in the caching or non-caching mode. It can also dynamically disable or enable caching. The reference layer objects, `CacheableServerRef` and `CacheableRef` are obtained by extending the default RMI reference classes `UnicastServerRef` and `UnicastRef`. When the remote object, `Impl` shown in Figure 4 is instantiated on the server, the object implementor provides enough information regarding the nature of caching, the transport protocol to be used, the consistency algorithm to be used etc. During instantiation, `Impl` calls the *exportObject* method of another class called `CacheableRemoteObject`. This instantiates `CacheableServerRef`, the reference layer object on the server side, `CacheableRef`, the reference layer object for the client, the consistency object (if needed, depending on the protocol), the transport endpoint object for the server VM and a few other objects. The configuration information specified by the user is stored in a `SystemParams` object which is also part of the reference layer. `SystemParams` object stores the name of the implementation, the name of the skeleton, and the *codebase* as its state. When a *bind* or a *rebind* operation is done on `Impl`, the `CacheableRef` object, the transport endpoint object and several other objects are serialized and exported to the **rmiregistry**. When a client does a *lookup* operation, `CacheableRef` and the transport endpoint object are transferred to the client's VM and are instantiated there. A `ConsistencyObject` object is instantiated as a part of the state of the `CacheableRef` object. During its instantiation, `Impl`, the skeleton for it and the `Impl_MemFuncStat` objects also get instantiated at the client. A consistency thread which handles the consistency requests from the server in a synchronized manner with the application thread is also forked during this process. The client side reference layer, from now on forwards all the invocations to the consistency object.

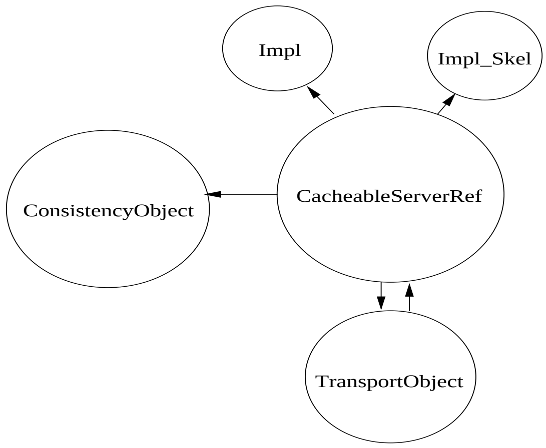

Figure 4: Server side object hierarchy for caching

The `ConsistencyObject` object performs the following sequence of actions before the actual invocation on the cached object is permitted.

- The cached copy is checked for its validity. If not valid then the most recent version of the copy is requested from the server (the server may communicate with another client to receive the latest copy). During this process, the client also acquires a *readlock* for the object that pins the object locally to ensure that its state cannot be invalidated while a method execution is in progress.
- The `impl_MemFuncStat` object is consulted to decide on the nature of the call. If it happens to be a write method, then the required consistency actions are executed and a *writelock* is acquired. This write lock provides atomicity of the method execution when the object state is updated.
- It then does the method invocation on the local object.

Finally when the application thread is about to quit, the inbuilt GarbageCollector is called in to free the resources. The consistency daemon's destruction method is modified so that if the client has the most recent

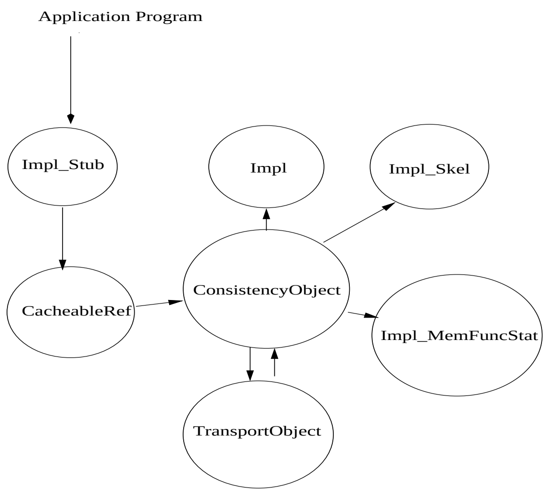

Figure 5: Client side object hierarchy for caching

copy of the object, it is sent to the server before the object is destroyed.

5 Performance Evaluation

We have discussed a number of techniques that could provide improved performance for RMI. The goal of this section is to experimentally evaluate the performance improvements (if any) that are made possible by the alternate transports and by object caching. Our performance studies are preliminary but the experiments conducted by us do provide some evidence of the effectiveness object caching and the use of multicast communication for maintaining consistency of cached object copies.

Our experiments were conducted in two different environments. In the first one, controlled experiments were conducted on a cluster of Sun Ultra 2's connected with a 100 Mbs Ethernet. There were no other applications running on the nodes in the cluster at the time of the experiments and as a result, we were able to reproduce the same results multiple times. We ran each experiment, for which results are presented in this section, six times. We present the average execution for remote method execution. The standard deviation across these experiments is not included because it was insignificant. For example, the maximum standard deviation observed in these experiments was 0.04.

Since caching is more effective when communication latencies are higher, the second environment we use in our experiments is two clusters connected via the Internet. These clusters were at the Georgia Tech and Emory University campuses which are separated by approximately six miles. The cluster at Emory had Sun Sparc 20 machines rather than Ultras. Although we could not control network traffic in the second environment which could impact the results of the experiments, we conducted the experiments late at night when there was minimal interference from other applications. As a result, we were able to obtain repeatable results with small variance across ten runs for most of the experiments. We present the mean times as well as the standard deviation for these experiments. In one case that required communication across several nodes in the wide-area environment, we could not obtain consistent results across different runs due to the variability in the environment. These results are not reported here. Thus, all the results reported here were obtained across a number of runs (at least six for each case) and we present both the mean and the standard deviation for them. In both environments, we used the JDK 1.1.5 distribution of Java with the just-in-time (JIT) compiling feature.

5.1 Object Caching

Object caching allows a remote invocation to be completed locally under a number of conditions. For example, if the valid state of the object is cached locally and the execution of an invocation only reads the object state, the needed consistency actions do not require remote communication. Similarly, if the node caches the object in exclusive mode and the object state is updated by the invocation's execution, other nodes are not notified of the update. If a copy of the object does not exist locally, communication with the server is required. When the object is requested in exclusive mode, the server may have to invalidate other copies before it can return the state of the object to the requesting client node. To measure the costs of invocations under these different conditions, for different size arguments, we measured the costs of completing a remote invocation in the following cases.

- The object is invoked remotely at the server node without caching it locally. Thus, this is the base case where the RMI framework is used to execute the invocation remotely.
- The object is cacheable but at the time it is invoked, its valid state is not available at the client node. In this case, the client must request the current state of the object before the invocation can be executed locally.
- A valid copy of the object is in the cache and the invocation is executed locally. Furthermore, the object is cached in a mode such that the execution of the invocation does not result in communication with other nodes for maintaining consistency. This could be either because the execution of the invocation only reads the object state or in case of an update, the object is cached in an exclusive mode.
- The invocation is executed with a cached copy but communication with other nodes is necessary to maintain consistency. For example, if the state of the object is updated as a result of executing the invocation, read-only copies at other nodes must be invalidated. This is done by communicating with the server which sends invalidation messages to the other clients.

Table 1 shows the results of the experiments that were conducted to evaluate the effectiveness of caching in both cluster and wide-area environments. For the cluster environment, we present average invocation times of ten runs of each experiments. Since controlled experiments were done in the cluster, there was very low standard deviation across the runs (less than 0.04) and Table 1 does not show it for the cluster environment. Clearly, executing an invocation with a cached copy when no communication is required provides much better performance than invoking the object remotely. For example, in the cluster environment when the invocation argument size is 32 bytes, invocation with a cached object completed in 1.53 ms compared to 3.51 ms required when the invocation is executed remotely at the server. Since caching is transparent to the application invoking the object, the invocation arguments are marshaled before the point when the reference layer determines that the object is cached locally. As a result, the execution time in the caching case does include the marshaling and unmarshaling costs.

If the execution of a method with a cached copy does require consistency actions to be executed which result in communication with remote nodes (e.g., invalidation messages), then the execution time of an invocation with a cached copy degrades with the number of invalidation messages. In fact, if communication is required with the server or other clients, executing the invocation with a cached object copy requires more time than its execution at the server. For example, when a valid copy of the object is fetched from the server before locally executing the invocation, the invocation execution time with 32 byte argument size is 5.34 ms compared to 3.51 ms when the invocation is executed at the server. Thus, the benefits of caching to an application will depend on the locality of access and on the mix of invocations that read and update the state of the object. Caching will be effective only when after the caching of an object at a client node, the client executes a number of invocations locally. This will happen when access conflicts at different clients (e.g., object is updated at two clients or one reads it while another one writes the object) are rare.

We now consider the wide-area environment. In this environment, we present both average execution time and the standard deviation for ten runs of each experiment. The improvement in performance for an invocation that executes locally with a cached copy is more dramatic in the wide-area environment. The execution time for 32 byte argument size invocation with caching is 1.58

ms. which is almost 8 times faster than invoking the object at a remote server. Clearly, caching could be more effective when communication overheads are higher. Notice that the costs with cached objects are different in the cluster and wide-area environments. These differences are due to differences in the client hardware (Ultra Sparcs vs. Sparc20s). We were able to obtain consistent results across many runs of an experiment in the wide-area environment when either no messages were sent or messages were exchanged between only two nodes. In the case when the client executing the invocation had to communicate with the server, which in turn had to send an invalidation message to two other clients, we were not able to get consistent results across different runs of the experiments due to lack of control over the network environment. Thus, execution times for this case are not included in Table 1.

The results in Table 1 made use of the TCP transport to send all messages, including invalidation messages that are sent to maintain consistency of cached copies. Since an invalidation message has to be sent to multiple nodes, instead of using separate messages, the server can send a single multicast invalidation message to all nodes that need to invalidate their copies. We used the multicast transport developed by us to send invalidation messages. As shown in Table 2, the use of multicast does improve performance of object invocation when invalidation messages are sent to multiple nodes. For example, when copies need to be invalidated at four client nodes, the use of multicast reduces invocation execution time from 12.6 ms to 9.24 ms when the argument size is 32 bytes. Thus, it is desirable to include a multicast transport to support the communication required by consistency protocols when caching is employed.

The effectiveness of caching (e.g., overall performance improvement for an application when caching is employed) depends on the pattern of method invocations. Our measurements indicate that if there is locality of access (e.g., an object is accessed several times before it gets invalidated), caching can result in significantly better performance, especially when communication latencies are high. To precisely characterize the benefits of caching, actual application or workloads are necessary. We discuss this issue in the next section.

5.2 Reliable UDP Based Protocol

To evaluate the impact of a transport protocol on the performance of remote method invocation, we measured the cost of a remote invocation at the server node when TCP and R-UDP protocols are used as transports. We did these experiments in the cluster environment with various sizes of invocation arguments. These results are presented in Table 3. As can be seen, choosing the R-UDP transport does not provide better performance for remote method execution. In fact, for an invocation that has 32 byte size arguments, its execution at server with R-UDP takes 6.36 ms compared to 3.51 ms with TCP. Although we obtained better round trip message times (an invocation results in a request and a reply) for R-UDP at the transport level compared to TCP, R-UDP does not provide better execution times at the remote method invocation level. There are a number of reasons that can explain why invocation level performance is not improved by R-UDP.

We found that the assumptions made by R-UDP about the reference layer actually do not match what we observed. For example, R-UDP assumes that a `flush()` call is made when the reference layer wants an invocation request to be sent to the server and this is done only once for each invocation. We found multiple calls to `flush()`, including some when the stream had no data that needed to be sent. We fixed some of these problems but our use of several threads to manage the transmission, retransmission and acknowledgment of messages, and synchronization between these threads and the application thread resulted in significant overheads for R-UDP. Currently we are redesigning R-UDP to reduce some of these overheads.

Implementation of remote method execution	Invocation argument size in bytes	Invocation execution time in ms.		
		Cluster environment	Wide-area environment	
			Average	Standard Deviation
At server via RMI support	0	2.54	12.81	0.36
	32	3.51	13.61	0.24
	1024	3.95	15.24	0.49
At client with a valid cached copy, when no consistency related communication is needed	0	0.90	0.93	0.05
	32	1.53	1.58	0.04
	1024	1.61	1.65	0.04
At client, valid copy fetched from server, no other consistency related communication needed	0	4.70	25.89	0.38
	32	5.34	26.98	0.33
	1024	5.45	26.46	0.35
At client, valid copy of object available, two other client copies invalidated via server	0	8.51	-	-
	32	9.16	-	-
	1024	9.19	-	-

Table 1: Caching Performance

Communication generated for executing remote method	Invocation argument size in bytes	Invocation execution time in ms.	
		TCP protocol	Multicast protocol
Client communicates with server, server invalidates one other client	0	6.96	7.15
	32	7.54	7.60
	1024	7.64	7.87
Client communicates with server, server invalidates two other clients	0	8.51	7.28
	32	9.16	7.93
	1024	9.19	8.13
Client communicates with server, server invalidates four other clients	0	11.68	8.40
	32	12.60	9.24
	1024	12.68	9.38

Table 2: Caching Performance with TCP and Multicast Protocol

Invocation argument size in bytes	Invocation execution time in ms. in Cluster environment	
	TCP	R-UDP
0	2.54	5.33
32	3.51	6.36
1024	3.95	6.95

Table 3: RMI performance with different transports

6 Discussion

We have explored a number of techniques for developing efficient implementations of RMI and have integrated them in the RMI framework. The initial performance studies that have been done by us have helped us understand when these techniques may provide improved performance. Clearly, additional performance studies are necessary to quantify the improvements in RMI performance. First, there is lot of room for improving the performance of the new transports that have been added by us. These improvements could come from better thread management at the transport implementation level as well the use of just-in-time compiling to reduce overhead of user-level implementations of the new transports.

The performance benefits of object caching depend on the object access patterns at client nodes. In particular, the locality of access and read-write mix of object invocations play an important role in determining the effectivess of caching. We are exploring a range of interactive applications. In these applications, shared graphical user interfaces (GUIs) and visualizations at participating users are supported by several shared objects. To provide access time that is independent of network latencies, copies of such objects must be created at each participant site. Clearly, caching allows such copies to be made. Furthermore, the current *focus-of-attention* of the interactions only requires manipulations of a small number of the objects. Objects that are not part of the current focus-of-attention are not updated and their copies can be accessed locally to drive the shared GUIs. We feel that the periodic access required to refresh the shared GUIs and localized focus-of-attention would lead to access patterns that are desirable in a caching environment (e.g., most accesses will be read-only and cached copies will be accessed repeatedly). However, we have not implemented and evaluated the applications to quantify the benefits of caching. In our current and future work, we plan to explore several workloads and applications to evaluate the effectiveness of caching.

We were able to add the new transports and object caching by extending the interfaces provided by the RMI framework except a small number of modifications to the interfaces themselves. For example, we had to add a new method to the `RemoteProxy` class which returns the name of the stub for the given class, the `Remote` interface being implemented either by the class itself or by one of it's superclasses.

7 Related Work

We have explored a number of techniques for enhancing the performance of RMI. Communication protocols that exploit the request-response nature of communication in distributed applications include T-TCP [3], VMTP [4] and others. Reliable multicast communication has been studied extensively (Isis and related systems [1], SRM [6], RMTP [16], Log-based [7] and others). Our multicast protocol is designed specifically to meet the needs of object consistency protocols. As a result, it can offer optimizations that are not possible in generic protocols (e.g., messages with newer values of an object make messages containing overwritten values obsolete).

Object caching has been studied in systems such as Spring [15], Flex [11], Thor [14], Rover [9] and others. The Spring distributed operating system presented a generic architecture for object caching. There are several differences in the approaches taken by Spring and by us due to differences in the system environments. For example, separate cacher processes are employed by Spring because of the low overhead of inter-address space communication. Since such inter-address space communication support does not exist in Java, we chose to cache the objects in the virtual machine that invokes the objects. The Flex system that we had implemented previously focused on multiple consistency levels, and several caching design decisions made by it differed from object caching in Java. Also, Flex did not explore transport level support for fast remote invocations. Object replication and caching in Java independent of the RMI mechanism have been explored in systems such as TIE [5] and Mocha [17]. By incorporating caching in the RMI framework, we ensure that applications do not need to differently deal with cached and non-cached objects.

We chose a straightforward protocol for maintaining the consistency of cached objects (similar to one used in the Ivy system for maintainingg coherence of distributed shared memory pages [13]). Considerable work has been done in the area of object consistency and consistency protocols. For example, in distributed file systems and distributed shared memories, a number of protocols have been developed. In our future work, we will explore different consistency levels and consistency protocols by developing a consistency framework similar to the one developed in Flex [11].

8 Concluding Remarks

Interactive distributed applications programmed with Java can run on a wide range of platforms. However, the interactive response time needs of such applications in high communication latency environments require efficient support for communication across sites. We have explored efficient implementations of Java RMI because it allows distributed applications to interact via the remote object invocation mechanism. We were able to integrate a range of performance enhancing techniques in the RMI framework by extending the interfaces provided by RMI. The prototype system we implemented allowed us to evaluate the performance benefits made possible by object caching as well as by multicast communication.

In the future we will undertake detailed performance evaluation of the system using actual applications and workloads. In addition, we will explore fault-tolerance via server replication and other notions of object consistency and associated consistency protocols that provide better scalability.

Acknowledgements

We would like to thank Ken Arnold of Sun Microsystems, the shepherd for this paper, and other anonymous referees. The feedback provided by them has improved the paper greatly.

References

[1] K. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, 1991.

[2] A. D. Birrell and B. J. Nelson. Implementing Remote Procedure Calls. *ACM Trans. on Comp. Sys.*, February 1984.

[3] R. Braden "RFC1644: T/TCP - TCP Extensions for Transactions Functional Specification" July 1994

[4] D. R. Cheriton, VMTP: A Transport Protocol for the Next Generation of Communication Systems *Proc. SIGcomm, pp. 406-415, 1986.*

[5] Michael Condict, Dejan Milojicic, Franklin Reynolds and Don Bolinger. Towards a World-wide Civilization of Objects. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, pages 25-32, Connemara, Ireland, September 1996.

[6] S. Floyd, V. Jacobson, C. Liu, S. McCanne, and L. Zhang. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. *ACM SIGCOMM 95, August 1995, pp. 342-356.*

[7] H.W. Holbrook, S.K. Singhal, and D.R. Cheriton. Log-based Receiver-Reliable Multicast for Distributed Interactive Simulation. In *Proceedings of SIGCOMM '95 , Cambridge, MA, August,1995. ACM SIGCOMM.*

[8] Java Remote Method Invocation Documentation, http://java.sun.com/products/jdk/1.1/ docs /guide/rmi/spec/.

[9] A. D. Joseph, A. F. de Lespinasse, J. A. Tauber, D. K. Gifford, and M. F. Kaashoek. Rover: A Toolkit for Mobile Information Access. In *Proc. of 15th SOSP*, 1995.

[10] R. Kordale, V. Krishnaswamy, S. Bhola, E. Bommaiah, G. Riley, B. Topol and M. Ahamad. Middleware Support for Scalable Services. In *Proc. IEEE Workshop on Community Networking*, October 1997.

[11] R. Kordale, M. Ahamad and M. Devarakonda. Object Caching in a CORBA Compliant System. Usenix Systems Journal, 1996.

[12] S. Kasera, J. Kurose, and D. Towsley. Scalable Reliable Multicast Using Multiple Multicast Groups. Proc. of 1997 ACM Sigmetrics, April 1997.

[13] K. Li, and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM trans. on Comp. Sys.*, Nov 1989.

[14] B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber., U. Maheshwari and L. Shrira. Safe and Efficient Sharing of Persistent Objects in Thor. *ACM SIGMOD*, 1996.

[15] Michael N. Nelson, Graham Hamilton, and Yousef A. Khalidi. A Framework for Caching in an Object-Oriented System. Sun Microsystems Laboratories Technical Report SMLI-TR-93-19.

[16] S. Paul, K. K. Sabnani, J. C. Lin, S.Bhattacharyya. Reliable Multicast Transport Protocol. *IEEE Journal on Selected Areas in Communications, Vol. 15, No. 3,April 1997, Pages 407-421.*

[17] B. Topol, M. Ahamad and J. Stasko. Robust State Sharing for Wide Area Distributed Applications, In *Proc. of International Conference on Distributed Computing (ICDCS)*, May 1998.

[18] Ann Wollrath, Roger Riggs , Jim Waldo. A Distributed Object Model for Java System. *Proceedings of the USENIX COOTS 1996.*