Explicit Batching for Distributed Objects

Eli Tilevich[1] and William R. Cook[2]

[1] Computer Science Department, Virginia Tech
tilevich@cs.vt.edu
[2] Department of Computer Sciences, The University of Texas at Austin
wcook@cs.utexas.edu

Abstract. Although distributed object systems, like RMI and CORBA, enable object-oriented programs to be easily distributed across a network, achieving acceptable performance usually requires client-specific optimization of server interfaces, making such systems difficult to maintain and evolve. Automatic optimization techniques, including Batched Futures and Communication Restructuring, are weaker and more unpredictable. This paper presents Batched Remote Method Invocation (BRMI) for Java, a system for explicit batching of remote operations on multiple objects, with support for array cursors, custom exception handling, and chained batches. With BRMI, clients can optimize their access to servers without custom server changes. The applicability of BRMI is demonstrated by rewriting several RMI client applications to use BRMI. Benchmarks of these applications and several micro-benchmarks demonstrate that BRMI outperforms RMI and has significantly better scalability when clients make multiple remote calls.

1 Introduction

Distributed object systems enable object-oriented programs to be easily deployed across a network. Examples of such systems include the Common Object Request Broker Architecture (CORBA) [1], the Distributed Component Object Model (DCOM) [2], and Java Remote Method Invocation (RMI) [3]. These systems generalize the notion of an object reference to support remote proxies [4], or references to objects on remote machines. When a method is invoked on a remote proxy, the method call is transferred between machines for execution. Distributed object systems abstract away the distance between objects, allowing them to interact as if they were local.

Powerful abstractions are good, except when they abstract away an essential aspect of the problem being solved. Distance is one of the essential problems in distributed computing, because of the need to manage bandwidth and latency of remote communication [5]. Most object-oriented programs use many small methods and frequent method invocations between objects, which when combined with the high latency of remote communication, leads to poor performance. Performance can be improved significantly by following design patterns for distributed systems [6]. However, these patterns require writing special-purpose

classes for each client interaction. As a result, such programs are difficult to maintain and evolve over time. Alternatively, systems can be optimized using mobile code, although this introduces additional complexity and potential for security problems.

Ideally the system would automatically optimize the communication patterns while preserving an abstract model of distributed objects. One way to do this is *implicit batching*, in which the system automatically combines communications into batches. Implicit batching is based on the observation that two remote calls to the same remote server can often be combined into a batch without affecting program semantics. Implicit batching was introduced in the Thor database system [7] but has been implemented in other systems [9, 10]. These systems work by modifying the program to combine remote calls into a batch, at runtime, if the program behavior is not affected by this change. There are several important situations that prevent creation of batches; in particular, retrieving multiple data fields, exception handling, and iterators all pose problems for implicit batching. As a result, implicit batching is not as effective as manual optimizations, and programmers cannot easily predict if optimizations will be applied.

This paper presents *explicit batching*, a new approach for optimizing distributed object systems. Explicit batches allow calls on remote objects to be invoked in a batch, while automatically transfering arguments and return values in bulk, in effect creating appropriate Data Transfer Objects on the fly. One advantage of explicit batches is that they do not require the server to be optimized for specific clients. Although the client must be rewritten to create batches explicitly, most aspects of the familiar distributed object style are preserved.

We have implemented explicit batching for Java in Batched Remote Method Invocation (BRMI), a middleware facility that enhances Java RMI. Using BRMI, any RMI client program can be modified to invoke multiple remote methods in programmer-defined batches.

We evaluate BRMI by using it to optimize several third-party applications and report performance improvements, programming effort, and latency effects. Unfortunately we do not know of a publicly available implementation of implicit batching for Java, so our comparison to implicit batching is subjective. The usability and performance studies that we have conducted indicate that explicit batching can be a worthwhile addition to an implementation of distributed objects, as a programming abstraction for managing latency and bandwidth costs.

2 Background

Explicit batches uses the well-known concept of a *future*. Futures were first introduced in Multilisp for creating and synchronizing concurrent tasks [11]. A future is a place-holder that stands in for a value that is not yet available. A simple generic interface for futures can be defined in Java:

```
public interface Future<T> {
  T get() throws Throwable;
}
```

Attempting to get a value before it is defined throws an error. Other exceptions may also be returned, as defined below.

Java RMI is a middleware mechanism for invoking the methods of a remote object located at a different Java Virtual Machine (JVM). From the programmer's perspective, RMI makes local and remote calls almost indistinguishable. To be remote, an object must implement one or more remote interfaces (i.e., extending java.rmi.Remote). The RMI client can call remote object's methods only through a remote interface. A stub implements the remote interface and serves as the remote object's client-side proxy [4]; operations on the stub are forwarded to the remote object.

For the client to obtain an initial stub object, the RMI runtime provides a simple distributed naming service, called RMI Registry. Additional stubs may be created when a local object is passed to the server, or server objects are returned to the client.

From the programmer's perspective, the differences between local and remote calls lie in parameter passing and exceptions. RMI passes reference parameters based on their runtime types: subtypes of java.rmi.Remote are passed by *remote-reference* and subtypes of java.io.Serializable by *copy*. Copying objects can improve performance, but it changes the semantics of a program. Remote calls can throw a RemoteException in response to communication and program-related errors. Program semantics is also affected by variations in identity of remote and serialized objects.

3 Explicit Batches with BRMI

BRMI is an extension of RMI that provides programming abstractions and runtime support for invoking multiple remote methods in a batch. On the client side, BRMI uses a new kind of proxy for objects involved in a batch. Operations on these proxies are recorded and method results returned as futures. When the client calls an explicit flush method, the batch is executed on the server and the results are assigned to the futures. The server is extended to execute batches and return multiple results to the client.

3.1 A Running Example

A running example that will be used throughout the paper is a distributed program that provides a hierarchical view of a collection of remote files. The server side functionality of the system is represented by interfaces File and Directory:

```
public interface File extends Remote {
  String getName() throws RemoteException;
  int getSize() throws RemoteException;
}
public interface Directory extends Remote {
  File getFile(String name) throws RemoteException;
}
```

A client program can use this interface to retrieve any file from a remote file system. For example, the following retrieves file "index.html" from the root directory.

```
Directory root = (Directory)Naming.lookup("url");
File index = root.getFile("index.html");
String name = index.getName();
int size = index.getSize();
print("File " + name + " size: " + size);
```

Not counting the initial lookup of the root reference, it takes 3 remote calls to retrieve file "index.html" and to obtain its name and size.

3.2 Batch Object Interfaces

BRMI introduces *batch* object interfaces, which are similar to RMI remote interfaces with systematic changes to types. Two of the rules for deriving a batch interface from a remote interface are: (1) a non-remote return type T is replaced by a future type Future<T>. (2) all remote interface return and argument types are replaced by corresponding batch interfaces. Primitive types are unchanged; exceptions and arrays are described below. By convention, the name of a batch interface starts with an B. The batch version of the Directory interface is as follows:

```
public interface BFile extends Batch {
  Future<String> getName();
  Future<Integer> getSize();
}
public interface BDirectory extends Batch {
  BFile getFile(String name);
}
```

The example given above can be performed as a batch using BRMI:

```
BDirectory root = BRMI.create(BDirectory.class, Naming.lookup("url") );
BFile index = root.getFile("index.html");
Future<String> name = index.getName();
Future<Integer> size = index.getSize();
root.flush();
print("File " + name.get() + " size: " + size.get());
```

Before using an object in an explicit batch, it must be wrapped in a batch-object proxy. This proxy is created by calling the BRMI factory BRMI.create method. The first parameter is the batch object interface to be used, and the second is the remote object being wrapped. The BRMI.create method returns a batch-object proxy that implements the batch interface.

Calls to the batch interface look similar to calls on a normal interface, except that the results are futures rather than values: getName returns a Future<String> rather than a String. The batch object proxy records the client calls and keeps

track of the futures that are created. Implementation details are given in Section 4.

To execute the batch, the client calls flush, which is defined in the base interface, Batch. The flush method sends the batch to the server. The results returned from the batch execution are then stored into the futures. After calling flush, the client can retrieve the future values using get. Any attempt to get the value of a future before flush results in an error.

Using BRMI decreases the number of network round trips required to invoke multiple remote methods. Any number of remote calls on many remote objects can be combined into a batch. This enables each client to optimize its own pattern of method invocation, without requiring server changes. The performance benefits of BRMI are evaluated in Section 5. BRMI also avoids the creation of remote RMI proxies other than for the root object. Although BRMI requires client program changes, the server code does not need to be changed.

3.3 Exceptions

As with RMI, explicit batches have an effect on exception handling. With explicit batching, a simple method call (T x = o.m()) is split into two parts: first the method is invoked to create a future (Future<T> fx = o.m()) and then after flush the actual value is retrieved (T x = fx.get();). With BRMI, any exception thrown on the server by method m is re-thrown during the second phase, when getting the value of the future. The client must use exception handlers in the second phase when futures are accessed, rather than during the first phase when methods are invoked. For example, the following code extends the running example to include an exception handler when accessing the size of the file.

```
try { // getSize may throw an error
  String size = index.getSize();
  print("File " + name + " size: " + size);
} catch (PermissionError e) {
  print("File " + name + " unknown");
}
```

Below is the full example using BRMI, which illustrates how exception handling is performed after flush, rather than before.

```
BFile index = root.getFile("index.html");
Future<String> fname = index.getName();
Future<Integer> fsize = index.getSize();
root.flush();
string name = fname.get();
try {
  print("File " + name + " size: " + fsize.get());
} catch (PermissionError e) {
  print("File " + name + " unknown");
}
```

Exceptions may also be thrown by the getFile method on the server, which returns a batch interface instead of a future. For example, getFile might throw FileNotFoundException. If getFile throws an exception, then the file name and size are meaningless, because these futures depend upon the result of getFile. In this case, attempting to get the value of the name future will rethrow the FileNotFoundException. In general, the get method of a future rethrows any exception on which the future's value depends.

In some cases very fine-grained exception handling is needed, where it is necessary to identify the exact method call that threw an exception. The base Batch interface includes a method ok to check if the method call was successful: it rethrows any exceptions on which the batch object depends. A remote method that returns **void** has type Future<Void> in its batch interface, so its exceptions can also be checked using the get method.

Note that network and communication errors are raised by flush, since it is the only call that performs remote communication. One downside of this approach is that methods in batch interfaces do not have exception declarations. To solve this problem, the exceptions would have to be declared on the future methods, by using a specific a future type for each combination of exceptions.

The default behavior of a batch is to abort processing when an exception is thrown. In some cases it may be useful to apply a different exception policy, for example to continue execution or restart the batch. For example, when reading files in a directory, an illegal access on one file should not terminate the entire batch. If the server follows a transactional model, then it may be useful to restart a batch after an error.

BRMI uses *exception policies* to specify how exceptions should be handled during the execution of a batch. The exception policy object is passed as an extra argument when a batch is created:

```
MInterface batch = BRMI.create(MInterface.class, ri, new ContinuePolicy());
```

The BRMI implementation currently provides three exception policy classes: AbortPolicy, ContinuePolicy and CustomPolicy. When a method in a batch causes an exception, a provided exception policy determines how this exception affects the execution of the batch. AbortPolicy aborts the execution if any exception is thrown. ContinuePolicy always continues the execution of a batch. CustomPolicy enables the programmer to express a custom exception policy, in which an action is specified for combinations of exceptions and method calls:

```
public enum ExceptionAction {
  Break, Continue, Repeat, Restart;
}
class CustomPolicy ... {
  ...
  public void setDefaultAction(ExceptionAction status) {...}
  public void setAction(String methodName, int index,
                        Exception e, ExceptionAction status) {...}
}
```

Through the setAction method, the programmer can specify what should happen to the execution of a batch when an exception is thrown. Specifically, in response to an exception of a certain type, the execution of a batch can be instructed to *break*, to *continue*, to *repeat* the method that caused the exception, or to *restart* the batch. The exception policies are implemented as **final** classes, as the programmer should not need to create their own exception policy classes. Thus, exception policies are implemented without the need for mobile code and dynamic class loading.

3.4 Operations on Arrays

It is useful to operate on arrays within a batch, but there are several difficulties in achieving this. As an example, consider adding a method allFiles () to return an array of files in a Directory. The batch interface version of this method, according to the rules above, would be Future<BFile[]>. Although the array contains batch objects (interface BFile) they cannot be accessed until after flush when the containing array is returned to the client. In this case proxies must be created for the files, and at least two batches are needed: one to get the array and its size, and one to operate on the items.

To support operations on arrays within a batch, BRMI introduces the concept of a *cursor*. A cursor is created when an array is accessed within a batch operation. Operations performed on the cursor are applied to every element of the array. The following example uses a cursor to retrieve the name and size of every file in a directory.

```
CFile cursor = root.allFiles();
Future<String> name = cursor.getName();
Future<Integer> size = cursor.getSize();
root.flush();
while(cursor.next())
  print( name.get() + ": " + size.get() );
```

A cursor is a special kind of batch interface. Before flush, the cursor stands for an arbitrary element of the array. Operations on the cursor return futures as if the cursor was a file. When flush is called, the server executes the batch and performs the requested operations on every element of the array. Any operation that uses the cursor as a target or argument is repeated for each array element. After flush, the cursor acts as an iterator: each time next is called, the iterator updates the futures to contain the values for the next array element.

In this example, the remote method (i.e., allFiles) is executed on the server, returning an array of Files. Within the same batch, the methods getName and getSize are called on each file in the array. The size of the array is not known in advance. The complete set of results are returned to the client, and used to populate the futures on each iteration of the loop.

To support cursors, the rules for translating remote interfaces to batch interfaces are modified so that an array of remote interfaces, R[], is converted to a cursor type CR. By convention, the name of a cursor interface starts with a C.

The automatically generated cursor type extends both a batch and CursorBase interfaces to contain batch and iteration methods:

```
public interface CursorBase {
  boolean next();
}
public interface CFile extends BFile, CursorBase {};
public interface BDirectory extends Batch {
  BFile getFile(String name);
  CFile allFiles();
}
```

Cursors allow multiple operations to be performed on all elements of an array. The operations can involve multiple remote references; for example, it would be possible to copy all files from one folder to another using cursors. Cursors can also be extended to allow multiple operations to be performed on all elements of a collection object, whose class implements interface java.lang.Iterable.

Despite their power, cursors are limited in applying a batch of operations to *every* element of a collection. It would be useful to allow operations to be applied to a selected subset of a collection. The client could specify the subset by passing a *filter* object, using mobile code, but we wish to avoid mobile code in the design of BRMI. Instead, operations on subsets are implemented by chaining batches together.

3.5 Chained Batches

When a series of client operations cannot be performed in a single batch, BRMI allows multiple batches to be chained together. A *chained batch* is a batch that depends upon the results of a previous batch. The later batches can use the objects created in the previous batch, and also create new objects and futures. As an example, consider the case of deleting a file if it is older than a given date. This requires two batches because the date must be retrieved to the client in order to decide whether to delete the file.

```
BFile mFile = root.getFile("A.txt"); // first batch
Future<Date> date = mFile.getDate();
root.flushAndContinue();   // finish first batch, begin second
if (date.get().before(cutOffDate)) {
  Future<String> name = mFile.getName();
  mFile.delete();
  root.flush();
  print("deleted " + name.get());
}
```

To create a chained batch, the client calls flushAndContine instead of flush. Existing batch calls are flushed and then a new batch is created. The server context of the previous batch is preserved, so that additional calls can be made to any batch interface from the original or chained batch. In the example above,

the first batch selects a file and gets its date. The client then uses the actual date value from the first batch to decide whether to delete the file. If so, then the second batch gets the file's name and deletes the file. After this batch is flushed, the client prints the file name.

Cursors can also be used in chained batches. A cursor is a batch object, so operations on the cursor in a chained batch are logged. Note that the cursor does have two different interpretations: in the batch that *creates* the cursor, operations on the cursor are applied to every element of the batch. After that batch is flushed, the cursor represents individual items from the array, so an operation on the cursor only applies to the current element. As an example, consider a function that deletes all files whose date is before a given cutoff date. This is accomplished in just two batches:

```
CFile cursor = root.allFiles();
Future<Date> date = cursor.getDate();
root.flushAndContinue();
while(cursor.next())
  if (date.get().before(cutOffDate))
    cursor.delete();
root.flush();
```

Batches can be chained as many times as necessary.

4 Implementation

BRMI is implemented as a layer on top of Java RMI, without changes to the Java language or runtime. This section focuses on the implementation issues of BRMI, its underlying techniques, and its integration with Java RMI. Section 5.2 quantifies BRMI performance benefits.

BRMI includes a tool to generate batch and cursor interfaces, as defined in the previous section. The batch interface generation process is transitive: it does not stop until all the transitively-referenced Batch interfaces have been generated. Thus, a Batch interface contains references only to other Batch interfaces, but never to Remote interfaces. The batch interface tool is invoked by using the −batch command line switch to `rmic`. Generating Batch interfaces automatically makes programming with BRMI easier.

The BRMI runtime architecture is logically divided into the client and server modules. Figure 1 compares the RMI and BRMI runtime architectures. It shows how BRMI can invoke multiple RMI methods as a batch. There are three phases in creating and executing an explicit batch: 1) invocation monitoring 2) batch execution and 3) result interpretation.

4.1 Invocation Monitoring (Client BRMI Stub)

The first phase, invocation monitoring, starts when the BRMI stub is created, and ends when flush is called. During this phase, client calls are recorded by

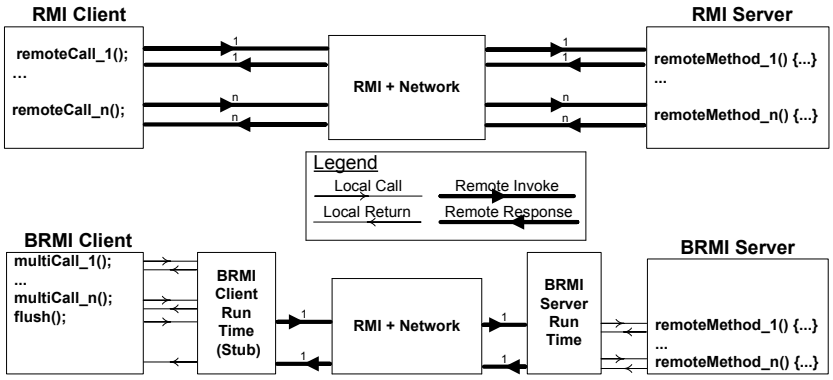

Fig. 1. RMI vs. BRMI Runtime Architecture

the client stub, but not sent to the server. The target and arguments of the call are stored in a *method descriptor* (an object of class InvocationData). Each method call is assigned a sequence number which acts as an identifier for that call. Some special processing is applied to the return values and arguments of the invocation.

If the method returns a Future type, a new future is created and added to the list of futures stored in the batch. A future is identified by the sequence number of method that created it. The future is returned as the result of the method invocation.

For a method returning a batch interface, the stub creates and returns an BRMI stub for this interface. The identification number of the stub is the sequence number of the method that created it. The new stub is returned as the result of the invocation.

Cursors are a special case of batch interface. Invocation monitoring for a cursor is the same, except that there is an ordering constraint: the operations on the cursor must be contiguous, so that operations on stubs from the cursor are not interleaved with operations on stubs that are not derived from the cursor. A cursor also maintains a list of all futures created by stubs derived from the cursor. In effect the cursor is a sub-batch of its containing batch.

For method arguments, the simple case is when the argument is a value (or serializable object). In this case the argument value is simply stored in the method descriptor.

If the argument type is a batch interface, then its value must be a stub that was previously created in the batch. An error is raised if the stub was created within a different batch chain. The stub is stored in the method descriptor as the argument value. When transmitting the InvocationData to the server, only the identifier of the stub is needed.

The same technique is used to identify the target of an invocation. Any method calls on a stub created by the batch will be included in the batch. The BRMI stubs are implemented as dynamic proxies [12].

```
object [] invokeBatch(InvocationData [] batch)
  object [] returnValues;
  RemoteObject[] batchObjects;
  batchObjects[0] = this;
  for each method in batch
    target = batchObjects[method.target_id];
    for each Remote method argument
      remoteArg = batchObjects[remoteArg.id_num];
    end for
    try
      object result = target.method(...);
    catch (exception)
      exceptions[method.id] = exception; // apply exception policy
    if (method.returnType is Remote)
      batchObjects[method.id] = result
    else
      returnValues[method.id] = result;
  end for
  return (returnValues, exceptions);
```

Fig. 2. BRMI server runtime functionality

The end result of invocation monitoring is a list of method descriptors, a list of stubs and a list of futures. The method descriptors are serializable objects that are sent to the server. The stubs are only needed for exception handling and chained batches — only the sequence numbers are sent to the server, so that method arguments can be matched to prior method return values. The futures are used when the results are returned from a batched execution.

4.2 Batch Execution

When the client calls flush, the recorded method calls are sent to the server as a batch by calling a regular RMI method invokeBatch. To make the BRMI functionality available to all RMI remote objects, the invokeBatch method is added to UnicastRemoteObject, a super class extended by RMI application remote classes.

As Figure 2 shows, invokeBatch implements all of the BRMI server runtime functionality. It decodes method descriptors from the InvocationData array, performs method invocations one-by-one, and returns the results back to the BRMI client.

The server plays back the method invocations in the same order in which they were invoked on the client. The remoteObj array keeps track of the objects created during the batch; they correspond one-to-one with the stubs created on the client during invocation monitoring. The non-remote values returned by

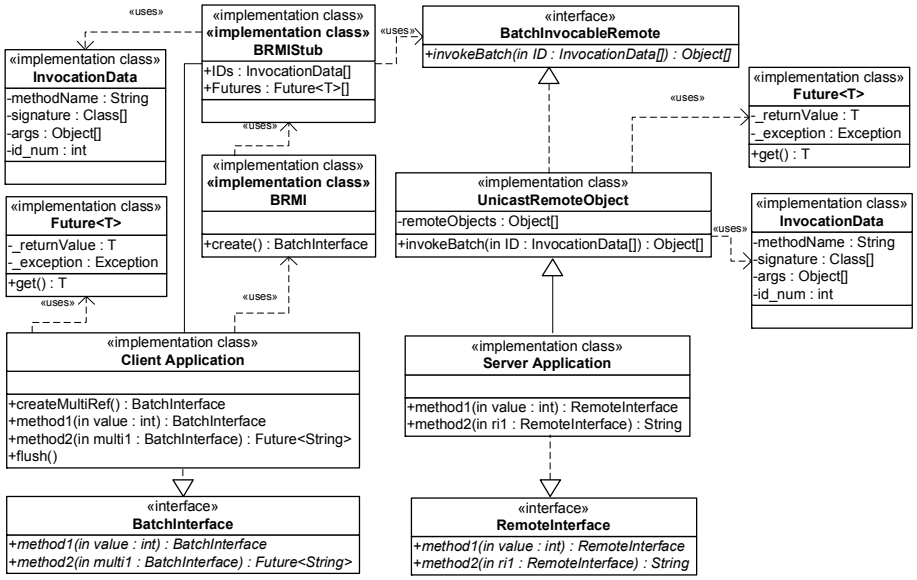

Fig. 3. BRMI UML Class Diagram

methods are stored in the returnValues array, which corresponds one-to-one with the list of futures on the client.

Exception handling and cursors are not fully specified in this pseudo code. Exceptions are trapped and handled as specified by a given exception policy (see Section 3.3 for details). Cursors are implemented by executing a sub-batch of methods for each item in the array. All of the cursor operations are performed at the point when the cursor value is created on the server. As a result, the relative order of playback of cursor operations may differ slightly from the order in which they were recorded, if non-cursor operations are interleaved with cursor operations (the client could signal an error if this happens). The server returns the number of elements in the cursor's array to the client and assigns sequence numbers to them so that they can be referenced in chained batches.

Upon the completion invokeBatch, the array of return values is sent back to the client, along with any exceptions that arose. Note that normal RMI proxies are never returned to the client.

4.3 Result Interpretation

The results from the server are an array of objects and an array of exceptions. The values are assigned to the futures in the client. The exceptions are assigned to the futures and the stubs. If the future has an exception, rather than a value, then this exception is thrown when accessing the content of the future.

For cursors, result interpretation is more complicated. Each time next is called on the cursor, the futures associated with the cursor are assigned values from the

return value array. The number of values in the array is the number of elements in the cursor times the number of futures. Futures normally do not change value after they have been assigned, but in the case of futures created within a cursor, the future values may change on each iteration of the loop.

Figure 3 contains a UML diagram that describes the main classes of a typical BRMI program. For BRMI stubs to call e, UnicastRemoteObject implements interface MultiInvocableRemote, which defines method invokeBatch. As a result, an BRMI stub can call invokeBatch on any of remote interface, implemented by a Remote object that extends UnicastRemoteObject.

4.4 Remote Reference Identity

Explicit batches solve a problem of remote reference identity that can arise in RMI: object identity is not necessarily preserved between remote calls. Consider a Remote object with two remote methods: create and use. The first method creates and returns another Remote object to the client, and the second one uses the returned Remote object as a parameter. This functionality can be represented by the following interface:

```
interface RemoteIdentityI {
  RI create() throws RemoteException;
  void use(RI ri) throws RemoteException;
}
```

The RemoteIdentityObj below provides a minimal implementation:

```
class RemoteIdentityObj implements RemoteIdentityI
{
  RI remoteObj;
  RI create() throws RemoteException {
    remoteObj = new SomeRemoteObject();
    return remoteObj;
  }
  void use(RI arg) throws RemoteException {
    assert arg == remoteObj;
    //will throw an error in RMI!
  }
}
```

The client code could call the remote methods as follows:

```
RemoteIdentityI remoteTest = ...;
RI ri = remoteTest.create();
remoteTest.use( ri );
```

The implementation of method use above will throw an error when the assert statement executes on the server. The top portion of Figure 4 shows why this is the case in Java RMI. While the result of create and the argument of use

represent the same remote object remoteObj, the RMI runtime does not ensure that these references be the same.

With remoteObj pointing to the original Remote object and arg to its stub is not a matter of programming inconvenience only. Any method invocation made through arg will result in an RMI remote call, even though the Remote object and its stub are in the same JVM. Having to go through the RMI runtime to invoke a call in the same JVM imposes a significant performance penalty, as we quantify in the next section.

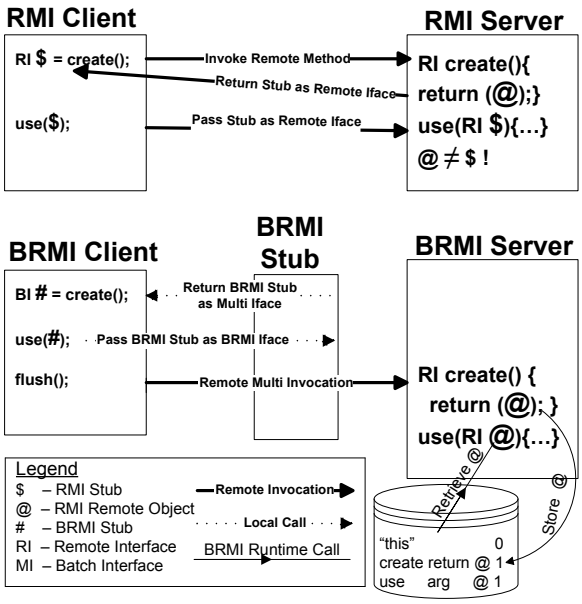

Fig. 4. Handling Remote Reference Identity RMI vs. BRMI

The argument arg points to an RMI stub of remoteObj rather than remoteObj itself is because Java RMI chooses not to preserve object identify when passing remote references across the network. Specifically, a Remote object is marshalled as its stub, but an RMI stub is always marshalled as itself, even if it is passed to the network site of the Remote object to which it is pointing.

The BRMI architecture enables efficient maintenance of remote reference identity, for remote objects created at the server never cross the network boundary. The BRMI server runtime replays locally all the invocations made on the client. In the example above, unlike RMI, which sends remoteObj back to the client, BRMI runtime stores the return value of method create in the local array right on the server. Figure 4 illustrates how the use method takes the identical object that was returned from method create, by loading it from the array maintained by invokeBatch.

This server-side mechanism not only completely avoids the costs of marshalling but also preserves remote reference identity. Indeed, when executing this example in BRMI, the return value of create and the argument of use is the same local reference to remoteObj.

When a remote parameter is Serializable , the behavior of a program can be different when executed locally, remotely using RMI, and in a BRMI batch. In particular, if the same object is passed to multiple methods, and the methods modify the object, then the resulting behavior depends upon when the serializable object is copied. Local execution will keep a single object, which is modified.

RMI makes two copies of the object, one for each method call, and the copies are discarded after being modified. BRMI will use the same copy for two methods executed in a batch, and then discard the copy. One of the main motivations for serializable objects is optimizing remote communications patterns. BRMI reduces the need for serializable objects to support the *Data Transfer* pattern [6]. When a remote parameter is Serializable , the behavior of a program can be different when executed locally, remotely using RMI, and in a BRMI batch. In particular, if the same object is passed to multiple methods, and the methods modify the object, then the resulting behavior depends upon when the serializable object is copied. Local execution will keep a single object, which is modified. RMI makes two copies of the object, one for each method call, and the copies are discarded after being modified. BRMI will use the same copy for two methods executed in a batch, and then discard the copy. One of the main motivations for serializable objects is optimizing remote communications patterns. BRMI reduces the need for serializable objects to support the *Data Transfer Object* pattern [6].

4.5 Concurrency

The recording algorithm of the BRMI stub can record only one batch at a time. Therefore, client threads sharing an BRMI stub cannot use it to create multiple batches. To create multiple concurrent batched invocations on the same remote object, client threads must obtain individual BRMI stubs by calling BRMI.create.

The BRMI server runtime invokes batched methods sequentially in the order they were recorded by the client program. Batching does not affect the synchronization constructs or the creation of new threads as part of the remote methods' application logic.

5 Evaluation

We evaluate both the applicability and the performance of BRMI. We have reengineered three third-party RMI applications to use BRMI and conducted a rigorous performance evaluation of our implementation through a series of micro-benchmarks.

5.1 Applicability

To assess the applicability of BRMI, we introduced explicit batching to three third-party RMI applications. These applications came from publicly available projects and books and represent different application domains. The success of these reengineering case studies confirms that the BRMI programming interface is expressive enough for introducing explicit batching in a variety of RMI programming scenarios.

Remote File Server This RMI application [13] is similar to the running example used throughout the paper: it uses Java RMI to obtain a listing of all files in a directory from a remote file system.

The client code obtains an array of files and prints their name, directory, modification date, and length:

```
...
RemoteFile[] files = remoteFile.listFiles();
for (int i = 0; i < files.length; i++) {
  System.out.println
  (
    files[i].getName()
   +": isDirectory="+files[i].isDirectory()
   +"; lastModified="+new Date(files[i].lastModified())
   +"; length="+files[i].length()
   );
}
```

The code snippet above requires **(1 + 4 * Number-of-Files)** remote calls to obtain a listing of files from a remote directory. BRMI can provide identical functionality in a single remote call as follows:

```
BRemoteFile remoteFiles = BRMI.create(BRemoteFile.class, Naming.lookup("url");

CRemoteFile filesCursor = remoteFiles.listFiles();

Future<String> nameFuture      = filesCursor.getName();
Future<Boolean> isDirectoryFuture = filesCursor.isDirectory();
Future<Long> dateFuture        = filesCursor.lastModified();
Future<Long> lengthFuture      = filesCursor.length();

remoteFile.flush();

while( filesCursor.next()) {
  System.out.println
  (
  nameFuture.get()
   +": isDirectory="+isDirectoryFuture.get()
   +"; lastModified="+new Date(dateFuture.get())
   +"; length="+lengthFuture.get()
   );
}
```

In the code above, BRemoteFile wraps the server RMI remote object. Then it obtains a cursor object filesCursor and uses it to specify methods getName, isDirectory, lastModified, and length, which are to be invoked on each element of the array of file objects returned by method listFiles on the server. It then calls flush, resulting in the invocation of *all* the methods on the server; their

results are transferred to the client in *one* batch, too. Finally, all the method calls within the **while** loop are local. These local calls retrieve the results using the filesCursor and Future objects.

Bank This RMI application [14] models a credit card management system. The server component represents a bank that manages credit card accounts. It exposes two external interfaces to the client CreditManager, for creating and looking up credit card accounts, and CreditCard, for making purchases and keeping track of the remaining balances. As a bootstrapping arrangement, the methods of the RMI interface CreditManager return CreditCard remote references to the client.

```
public interface CreditManager extends java.rmi.Remote {
    public CreditCard findCreditAccount(String Customer) throws RemoteException;
  ...
}

public interface CreditCard extends Remote {
    public float getCreditLine() throws RemoteException;
    public void makePurchase(float amount) throws RemoteException;
}
```

Thus, the client of this credit management system looks up a reference to CreditManager in the RMI registry, calls one of its methods (e.g., findCreditAccount) obtaining a reference to a remote CreditCard object, and uses it to make purchases.

In the original application, the account lookup and each purchase require a separate remote call. The goal of this case study is to explore whether with BRMI the application can accomplish the same functionality with only one remote call. While it is fairly straightforward to use BRMI to combine all the remote calls into a single batch, the account lookup presents a complication. If it throws an exception, the batched execution should be terminated, as it failed to return a valid CreditCard object on which the subsequent purchase methods are performed. The BRMI exception policy mechanism can provide an elegant solution to this problem. Specifically, the following exception policy can be specified when creating an BRMI invocation object:

```
CustomPolicy cp = new CustomPolicy();
cp.setDefaultAction(ExceptionAction.Continue);
cp.setAction(DuplicateAccountException.class, "findCreditAccount",
             0, ExceptionAction.Break);
BCreditManager cmm = BRMI.create(BCreditManager.class, cm, cp);
```

With this exception policy in place, it is safe to specify the rest of the methods to be invoked in the same batch:

```
account = cmm.findCreditAccount("AccountName");
account.makePurchase(123.00);
account.makePurchase(456.00);
```

```
Future<Float> creditLineFuture = account.getCreditLine();
cmm.flush();
float creditLine = creditLineFuture.get();
```

If a thrown exception signals that an account cannot be found, the subsequent methods will not be executed; the exception will be re-thrown when the client calls creditLineFuture.get.

Translator This RMI application [15] implements a simple translation service. The client specifies a translation request using a Word object, and the server replies with the translation represented as another Word object. The application was built to handle one translation request at a time. With BRMI, it is possible to change this application to handle multiple requests without any changes to its server design. Because the actual number of words to be translated is entered at runtime, a dynamic array of Future objects is used:

```
Word[] words = getWords();

Future<Word>[] responseFutures = new Future[words.length];
for (int i = 0; i < responseFutures.length; ++i)
  responseFutures[i] = translatorRoot.translate(words[i]);
translatorRoot.flush();

for (int i = 0; i < responseFutures.length; ++i)
  System.out.println("result " + i + ": " + responseFutures[i].get());
```

This simple example demonstrates how the BRMI API makes it possible for the programmer to express the size and composition of batches at runtime.

5.2 Performance Numbers

To assess the advantages of explicit batching, we compared the performance of RMI and BRMI versions of three micro benchmarks and a macro benchmark. All the experiments were run in JDK 1.6 (build 1.6.0-b10) in two configurations:

1. Two identical Windows XP Professional workstations, with Dual Core 3GHz processors and 2 GB of RAM, connected with a dedicated 1Gbps, 1ms latency network.
2. Two identical Windows XP Professional Toshiba Satellite laptops, with Dual Core 1.66 GHz Processor and 1GB of RAM, connected with a 54.0 Mbps, 252ms latency wireless network.

To ensure accuracy, all the benchmarks were repeated between 5000 to 10000 times with the results averaged.

5.3 Micro benchmarks

The three micro benchmarks that we ran consisted of a no-op method, linked list traversal, and a simple remote simulation.

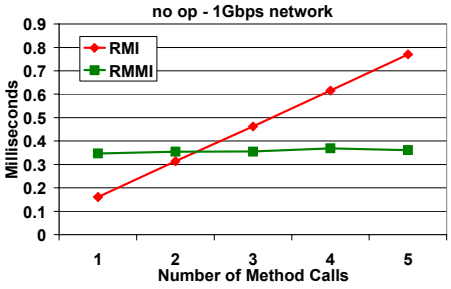

Fig. 5. No-op Benchmark (LAN)

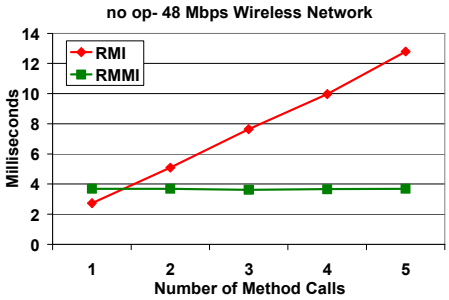

Fig. 6. No-op Benchmark (Wireless)

no-op As a no-op benchmark, we used a do-nothing remote method that takes no parameters and returns **void**. This benchmark, thus, isolates the middleware processing overhead. In BRMI, we used a single batch irrespective of the number of method calls.

The performance graphs in Figures 5 and 6 indicate that in both networking configurations, as the number of calls increases, the time taken grows linearly in RMI and stays almost constant in BRMI. The execution time of a remote call can roughly be split into network transmission and processing. Since the processing amount is about the same for both RMI and BRMI (BRMI does some extra processing), the BRMI version's advantage is due solely to reduced overall latency.

As expected, RMI outperforms BRMI when the batch size is smaller than two due to the overhead of the BRMI runtime.

Linked List Traversal In this benchmark, we measure the time it takes a client to traverse and retrieve an n^{th} nodes value of a linked list implemented as a remote object with the following Remote and Batch interfaces:

```
interface RemoteList extends Remote
{
   RemoteList next();
   int getValue();
}
```

```
interface BRemoteList extends Batch
{
   BRemoteList next();
   Future<Integer> getValue();
}
```

Despite being an inefficient approach to retrieve an n^{th} nodes value of a linked list, this micro benchmark demonstrates an important piece of functionality: traversing a variable chain of references.

The performance graphs in Figures 7 and 8 show trends similar to that of the no-op benchmark in both configurations: whereas RMI numbers grow linearly, BRMI ones stay close to constant. One unexpected result is that in both configuration, BRMI outperforms RMI even when traversing only one node (i.e., no batching is possible). In the BRMI version, method next returns a MRemoteList

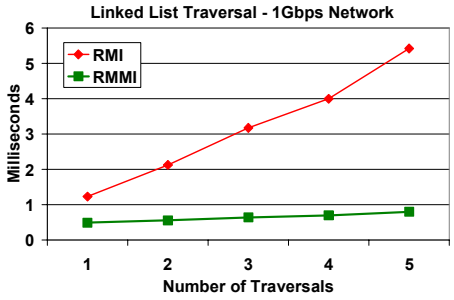

Fig. 7. Traversing a Linked List (LAN)

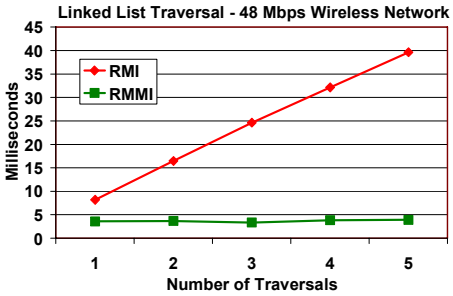

Fig. 8. Traversing a Linked List (Wireless)

reference. This completely avoids any network transfer, as explained in Section 4.4: the actual Remote return value is simply stored on the server for future invocations.

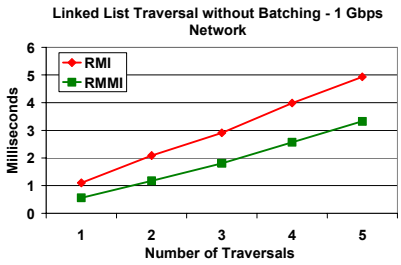

Fig. 9. Linked List Traversal (Batches of Size 1)

To test the extent of this advantage in terms of performance improvement, we changed the BRMI version of the benchmark to call flush after each method invocation, resulting in sending a batch of size one to the server. Figure 9 shows the results.

Surprisingly, even without batching, BRMI consistently outperforms RMI, even though the BRMI execution time now grows linearly. This shows that eliminating the overheads of marshaling and unmarshaling a Remote object, inherent in transferring it over the network, offers performance improvements.

Remote Simulation In this micro benchmark, we measure the time taken by performing a simulation running on a remote machine. The simulation functionality is implemented and managed by two remote object: Simulation and Balancer. This is not an unrealistic scenario in computer simulations with advanced systems requirements such as load balancing. The purpose of this benchmark is to assess the benefits of preserving remote reference identity in BRMI. The MSimulation interface is listed below:

```
interface BSimulation extends Batch {
  BBalancer createBalancer();
  Future<Integer> performSimulationStep(int reps,
                                        BBalancer balancer);
  Future<Double> getSimulationResults();
}
```

The Simulation object first creates and returns to the client a Balancer object; the client parameterizes it with the balancing policy to be followed. performSimulationStep method then takes the Balancer object as a parameter to be able to call its only balance method.

Figures 10 and 11 show the results for this benchmark. The values of x-axes move in the increments of five. To isolate the benefits provided by preserving remote reference identity, flush is called after every invocation of performSimulationStep, sending a batch with a single method to the server. Thus, the performance improvements are due solely to BRMI maintaining the identity of the balancer's Remote interface. As the following pseudocode demonstrates, on the server, the call to balance is a regular local call in the BRMI version, whereas it is a remote call in the RMI version.

```
int performSimulationStep(int reps, Balancer b) {
  from 1 to reps
  b.balance(); //local call in BRMI; remote call in BRMI
}
```

The performance improvements in the BRMI version remain consistent even for high numbers of simulation steps (40 in this case).

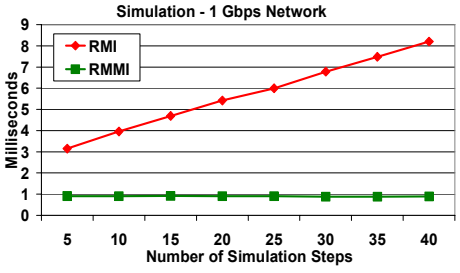

Fig. 10. Remote Simulation (LAN)

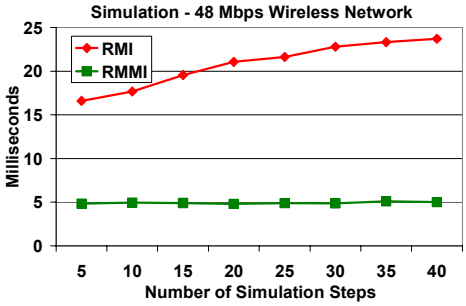

Fig. 11. Remote Simulation (Wireless)

5.4 Remote File Server Macro Benchmark

This macro benchmark details the performance characteristics of the Remote File Server used as the running example throughout the paper.

The server first creates a directory with 10 files of a specified size. It then loads all the files from disk into main memory, to avoid disk access tainting the results. We measured the total time it takes to request and transfer n files from the server, with the total size of all the files equal to 100KB. Figures 12 and 13 shows that BRMI achieves several orders of magnitude performance increase over RMI for most input parameters (i.e., number of files to retrieve from server),

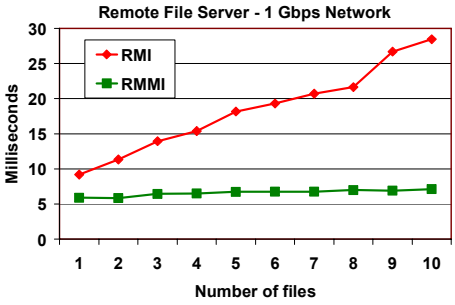

Fig. 12. File Server (LAN)

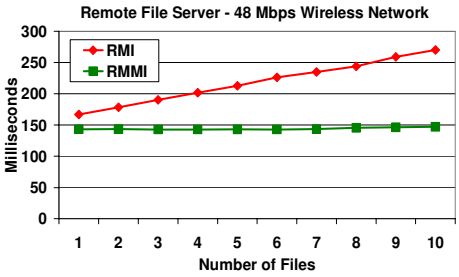

Fig. 13. File Server (Wireless)

with the advantages present in both configurations. This BRMI version benefits from both batching and preserving remote reference identity.

6 Related Work

Explicit Batching. Software design patterns [6] for *Remote Facade* and *Data Transfer Object* (also called Value Objects [16]) can be used to optimize remote communication by performing block transfers of data between client and server. A *data transfer object* is a Serializable class that contains properties retrieved from or sent to a remote object. However, a system must often be rearchitected to support value objects. In addition, different kinds of value objects may be needed by different clients. Rather than hard-coding a batching decision into the implementation of a value object, the programmer can use BRMI and its convenient API to combine any set of remote methods into a batch. BRMI constructs an appropriate value object on the fly, automatically, as needed by a particular situation. BRMI also generalizes the concept of a data transfer object to support transfer of properties from arbitrary collections of objects, and also invocation of arbitrary methods, rather than just accessors and getters.

The DRMI system [17] aggregates RMI calls following an approach similar to BRMI. DRMI also uses special interfaces to record and delay the invocation of remote calls. An interesting feature of DRMI is the ability to pass a Future that resulted from one call as an argument to a later call. This design choice basically precludes Serializable arguments from being passed to batched methods. In addition to simple call aggreation, BRMI supports array cursors, custom exception handling, and chained batches. BRMI also allows passing a result of a batched call to a later call, but only for remote results, thereby allowing passing arbitrary Serializable arguments to batched methods.

Detmold and Oudshoorn [8] present analytic performance models for RPC and its optimizations including batched futures. It also proposes a new optimization construct termed *a responsibility*. Their analytic models could be extended

to model the performance properties of the new optimization constructs of BRMI such as array cursors and chained batches.

Explicit batches are sometimes defined directly in a communication protocol. One example is the compound procedure in Network File System (NFS) version 4 Protocol [18]. It reduces latency by combining multiple NFS operations into a single RPC request. The compound procedure in NFS is not a general-purpose mechanism; the calls are independent of each other, except for a hard-coded current filehandle that can be set and used by operations in the batch. There is also a single built-in exception policy. Web Services are often based on transfer of documents, which can be viewed as batches of remote calls [19, 20].

Cook and Barfield showed how a set of hand-written wrappers can provide a mapping between object interfaces and batched calls expressed as a web service document. BRMI automates the process of creating the wrappers, and generalizes the technique to support exception policies and remote cursors. As a result, BRMI scales as well as an optimized web service, while providing the raw performance benefits of RPC [21]. Web services choreography [22] defines how Web services interact with each other at the message level. BRMI can be seen as a choreography facility for distributed objects.

Mobile Code. Mobile object systems such as Emerald [23] reduce latency by replacing multiple remote method calls with moving an entire object so that the calls could be performed locally. Ambassadors is a communication technique that uses object mobility [24] to minimize the aggregate latency of multiple interdependent remote methods. DJ [25] adds explicit programming constructs for direct type-safe code distribution, improving both performance and safety.

Mobile objects generally require sophisticated runtime support not only for moving objects and classes between different sites, but also for dealing with security issues. A Java application can essentially disable the use of mobile code by not allowing dynamic class loading. Because BRMI does not require any changes to the server, it can minimize the aggregate latency of remote communication without using mobile code.

Implicit Batching. *Batched futures* have been introduced to reduce the aggregate latency of multiple remote methods [7]. If remote methods are restructured to return futures, they can be batched. The invocation of the batch can be delayed until a value of any of the batched futures is used in an operation that needs its value. Our approach uses the batched future construct to implement explicit batching for distributed objects.

Yeung and Kelly [9] use byte-code transformations to delay remote methods calls and create batches at runtime. A static analysis determines when batches must be flushed. Small changes in the program, for example introducing an assignment to a local variable, or an exception handler, can cause a batch to be flushed. From a performance viewpoint, the programmer cannot be sure what batches will be performed. In addition, it is not clear how the array cursors of BRMI could fit into an implicit batching model. Chained batches, however, are

possible, as the implicit mechanism will combine any possible set of operations into a batch.

Future RMI [10, 26] uses asynchronous communication to speed up the execution of RMI in the Grid environment when one remote method is invoked on the result of another. Remote results of a batch are not transferred over the network, being stored on the server to be used for subsequent method invocations.

7 Conclusion

This paper presents explicit batching for distributed objects, a general mechanism for clients to optimize access to distributed objects. Clients must be rewritten to use explicit batched interfaces. However, the resulting program still uses the familiar object-oriented style and has the added benefit of making distributed communication very explicit. Explicit batches support multiple operations on any number of objects, including operations on the results of previous method calls in the batch. Additional features include custom exception handling, bulk operations on every element of a collection, and chained batches to support operations that cannot be performed in a single batch.

The server infrastructure supports explicit batches for all clients, so that servers do not need to be optimized for specific clients. Traditionally, servers must be optimized to support Data Transfer Objects that are specific to client access patterns. Explicit batches perform the necessary builk data transfers on the fly, as specified by the batch. These transfers both send data to the server in bulk, and return results in bulk.

We implemented explicit batches in Batched Remote Method Invocation (BRMI), an extension to Java RMI. A tool automatically creates batched interfaces from normal remote interfaces. It it interesting to note that BRMI avoids use of object proxies, except for the root of a batch. The design also avoids all use of mobile code. In this sense, explicit batching is more similar to web services than traditional distributed object systems.

We evaluated the applicability and peformance of BRMI. Several third-party applications were converted to use BRMI without difficulty. Benchmarks of these applications and several micro-benchmarks demonstrate that BRMI outperforms RMI and has significantly better scalability when clients make multiple remote calls.

References

1. The Object Management Group (OMG): The Common Object Request Broker: Architecture and Specification. (1997)
2. Brown, N., Kindel, C.: Distributed Component Object Model Protocol–DCOM/1.0 (1998) Redmond, WA, 1996.
3. Sun Microsystems: Java Remote Method Invocation Specification. (1997)
4. Shapiro, M.: Structure and Encapsulation in Distributed Systems: The Proxy Principle. In: Proceedings of the 6^{th} International Conference on Distributed Computing Systems. (1986) 198–204

5. Patterson, D.A.: Latency lags bandwith. Commun. ACM **47** (2004) 71–75
6. Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)
7. Bogle, P., Liskov, B.: Reducing cross domain call overhead using batched futures. ACM SIGPLAN Notices **29** (1994) 341–354
8. Detmold, H., Oudshoorn, M.: Communication Constructs for High Performance Distributed Computing. In: Proceedings of the 19^{th} Australasian Computer Science Conference. (1996) 252–261
9. Cheung Yeung, K., Kelly, P.: Optimising Java RMI Programs by Communication Restructuring. In: ACM/IFIP/USENIX International Middleware Conference, Springer (2003)
10. Alt, M., Gorlatch, S.: Future-based RMI: Optimizing compositions of remote method calls on the grid. Proc. of the Euro-Par **2790** (2003) 427–430
11. Halstead, R.: MULTILISP: a language for concurrent symbolic computation. ACM Transactions on Programming Languages and Systems **7** (1985) 501–538
12. Sun Microsystems: Dynamic proxy classes specification. (1999)
13. Pitt, E., McNiff, K.: Java.Rmi: The Remote Method Invocation Guide. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA (2001)
14. Heller, P.: Java 1.1 developer's handbook. SYBEX San Francisco, CA (1997)
15. Grosso, W.: Learning command objects and RMI. `http://www.onjava.com/pub/a/onjava/2001/10/17/rmi.html` (2001)
16. Alur, D., Crupi, J., Malks, D.: Core J2EE Patterns: Best Practices and Design Strategies. Prentice Hall PTR (2003)
17. Marques, E.: A study on the optimisation of Java RMI programs. Master's thesis, Imperial College of Science Technology and Medicine, University of London (1998)
18. Shepler, S., Callaghan, B., Robinson, D., Thurlow, R., Beame, C., Eisler, M., Noveck, D.: Network File System (NFS) version 4 Protocol (2003)
19. Vogels, W.: Web services are not distributed objects. Internet Computing, IEEE **7** (2003) 59–66
20. Cook, W., Barfield, J.: Web Services versus Distributed Objects: A Case Study of Performance and Interface Design. In: Proceedings of the IEEE International Conference on Web Services (ICWS'06)-Volume 00, IEEE Computer Society Washington, DC, USA (2006) 419–426
21. Demarey, C., Harbonnier, G., Rouvoy, R., Merle, P.: Benchmarking the Round-Trip Latency of Various Java-Based Middleware Platforms. Studia Informatica Universalis Regular Issue **4** (2005) 7–24
22. Peltz, C.: Web services orchestration and choreography. Computer **36** (2003) 46–52
23. Black, A.P., Hutchinson, N.C., Jul, E., Levy, H.M.: The development of the Emerald programming language. In: HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages. (2007) 11–1–11–51
24. Detmold, H., Hollfelder, M., Oudshoorn, M.: Ambassadors: structured object mobility in worldwide distributed systems. In: Proc. of ICDCS'99. (1999) 442–449
25. Ahern, A., Yoshida, N.: Formalising Java RMI with explicit code mobility. In: Proc. of OOPSLA '05, New York, NY, USA, ACM (2005) 403–422
26. Alt, M., Gorlatch, S.: Adapting Java RMI for grid computing. Future Generation Computer Systems **21** (2005) 699–707