Explicit Batching for Distributed Objects

Eli Tilevich[1] William R. Cook[2] Yang Jiao[1]

[1] Computer Science Department, Virginia Tech
{tilevich,jiaoyang}@cs.vt.edu

[2] Department of Computer Sciences, The University of Texas at Austin
wcook@cs.utexas.edu

Abstract

Although distributed object systems, for example 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, do not work as well as hand optimization. This paper presents Batched Remote Method Invocation (BRMI), a language-level technique for clients to specify explicit batches of operations on remote objects. We have implemented BRMI for Java as an extension of RMI, with support for batches with array cursors, custom exception handling, conditionals and loops. BRMI allows common design patterns, including Data Transfer Objects and Remote Object Facade, to be constructed on the fly by clients without special server support. The performance benefits of batching operations are well known; our evaluation focuses on the usability of explicit batches, but we also confirm that BRMI outperforms RMI and scales significantly better when clients make multiple remote calls. The applicability of BRMI is demonstrated by rewriting several RMI client applications to use BRMI.

1 Introduction

When using existing distributed object systems, including Common Object Request Broker Architecture (CORBA) [16], the Distributed Component Object Model (DCOM) [7], or Java Remote Method Invocation (RMI) [21], care must be taken to design efficient interfaces for remote operations. Design patterns for optimization, including Remote Facade and Data Transfer Object, assume some knowledge of typical client interaction patterns [13]. In a more dynamic or open environment, when client interaction scenarios are unknown or change over time, a more flexible approach to optimizing communication is needed.

One way to do this is *implicit batching*, in which the system automatically combines communications into batches [6, 8]. When one call returns a proxy that is used in a subsequent call, the calls can sometimes be combined into a batch without affecting program semantics. Implicit batches do not optimize multiple calls that return primitive values, so they do not support implicit Data Transfer objects. As a result, implicit batches are often less efficient and more unpredictable than hand-optimized remote interfaces. More flexible optimization is possible using mobile code, but this does not work easily across multiple platforms and introduces potential for security problems.

This paper presents *explicit batching*, a client API and middleware extension for flexible management of communication in distributed object systems. Explicit batches allow multiple calls on remote objects to be invoked in a batch, while automatically transferring arguments and return values in bulk, in effect creating appropriate composite Data Transfer Objects and Remote Facades on the fly. Explicit batches allow clients to specify their own communication patterns, using generic batching capabilities provided by the server. The infrastructure provides some of the benefits of mobile code, while using a platform-independent representation for batches. Although the client must be written 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.

The performance benefits of batching operations are well-known, especially in high-latency environments. 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.

This paper makes the following contributions:

- A general language-level technique that enables clients to specify explicit batches of operations on remote objects; batches can include array cursors, custom exception handling, conditionals and loops.
- BRMI, a practical implementation of explicit batching for Java, including a client API and middleware extension.
- A thorough evaluation of BRMI that assesses its performance improvements, programming effort and latency effects.

2 Background

Java RMI is a middleware mechanism for invoking methods on 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 a 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 [19]; 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.

BRMI also uses the well-known concept of a *future*. 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:

```
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.

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:

```
interface File extends Remote {
  String getName() throws RemoteException;
  int getSize() throws RemoteException;
}
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 a B. The batch version of the Directory interface is as follows:

```
interface BFile extends Batch {
  Future<String> getName();
  Future<Integer> getSize();
}
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. Section 5 evaluates the performance benefits of BRMI. BRMI also avoids the creation of remote RMI proxies other than for the root object. BRMI requires only client program changes, with the server code remaining the same.

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. 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 { // getSize may throw an error
   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 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:

```
BInterface batch = BRMI.create(BInterface.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:

```
enum ExceptionAction {
   Break, Continue, Repeat, Restart;
}
```

```
class CustomPolicy ... {
  ...
  void setDefaultAction(ExceptionAction status) {...}
  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. These are special cursor-dependent futures, which take on multiple values after the flush. 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 all cursor-dependent futures to contain the values for the next array element.

In this example, the remote method (i.e., allFiles) is executed on the server, creating an array of Files. The size of the array is not known in advance. Within the same batch, the methods getName and getSize are called on each file in the array. 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:

```
interface CursorBase { boolean next(); }
interface CFile extends BFile, CursorBase {};
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.

3.5 Conditionals and Loops

To enable batches with a non-linear control flow, BRMI supports remote conditionals and loops. Without having to interrupt its execution, a batch can branch or repeat some of its operations based on the result of a batched method returning a Future<Boolean>. Because the syntax of Java is not extensible, the BRMI design for conditionals and loops introduces new constructs sparingly, having them as much as possible, look and feel as their local Java counterparts. BRMI defines its conditional and looping operations in interface ConditionalsAndLoops:

```
interface ConditionalsAndLoops {
  Future<Boolean> rIf(Future<Boolean> condition);
  void rElse();
  void rWhile(Future<Boolean> condition);
  void rEnd();
}
```

By extending this interface, all batch and cursor interfaces inherit the capacity to express a non-linear control flow for remote batches. Thus, a simple **if**/**else** logic can be implemented in a batch as follows:

```
BR br = ...; // some batch interface
br.rIf (br.test ()); // start if
  br.foo ();
br.rElse (); // start else
  br.bar ();
br.rEnd(); // end if
br.flush ();
```

This code snippet expresses functionality equivalent to that of a regular Java conditional statement with a few syntactic differences. The control flow constructs of **if** and **else** become methods rIf and rElse, respectively, and method rEnd terminates the conditional branch. In lieu of the curly brace characters in BRMI, rEnd terminates all statement blocks starting with either rIf or rWhile.

As a specific example of using conditionals in a batch, consider a scenario that requires deleting a remote file, if it was modified before a given date. rIf can express this logic similarly to an **if** statement in an imperative programming language:

```
BR bFile = br.getFile (" A.txt ");
Future<Boolean> res =
br. rIf  (bFile.isExpired(cutOffDate));
   bFile.delete ();
br.rEnd();
br.flush ();
print (" rIf  was " + res.get ());
```

Note that rIf returns Future<Boolean>, which is simply the value of its parameter, allowing the client to determine which remote branch was taken during the execution of a batch.

When rIf is called on a BRMI cursor, it concisely expresses a filtering operation to be applied on each item of a remote collection. Consider extending the above example of removing an expired file to a collection of remote files:

```
CRemoteFile cr = br.allFiles ();  // cursor
cr. rIf (cr.isExpired(cutOffDate);
   cr.delete ();
cr.rEnd();
 ...
br.flush ();
```

This example first retrieves a cursor for a collection of remote files and then deletes all files that have expired, with all the remote operations executed in a single batch.

Conditionals and loops increase the expressive power of BRMI by allowing a non-linear control flow in a batch, whereas their presence hinders implicit batching optimizations. For example, an **if** statement using the result of a remote method would require breaking an implicit batch, as to execute an **if**, its **boolean** operand must be first evaluated. By providing the remote equivalents of the Java conditional and looping constructs, BRMI can batch longer sequences of remote methods.

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. 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 (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 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 a 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 are recorded contiguously, even if operations on stubs from the cursor are 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 [22].

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, conditionals and loops — 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.

The BRMI server runtime decodes method descriptors, invokes batched methods 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 methods are stored in the returnValues array, which corresponds one-to-one with the list of futures on the client.

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 non-cursor operations will be executed after the cursor operations. 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 conditionals and loops.

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.

5 Evaluation

We evaluate both the applicability and the performance of BRMI. We have reengineered two 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 two third-party RMI applications that came from publicly available projects and books and represent different domains.

5.1.1 Remote File Server

This RMI application [18] 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, and can be implemented in BRMI 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.

Compared to implicit batching, the use of the cursor abstraction here makes it possible to obtain a listing of an entire directory in a single remote call. To optimize a regular RMI program to that level of efficiency, implicit batching would need compiler optimization techniques that could synthesize the specialized semantics of cursor objects (i.e., applying a set of operations on each element of a to-be-created remote array). We are not aware of complier optimizations powerful enough to transform a target program in such a significant way fully automatically.

5.1.2 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 BRMI can provide the same functionality in a single remote call. Although 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 a 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.

Without the ability to express a custom exception policy, implicit batching would have no choice but to emulate the execution semantics of the original RMI program. To do this safety will require *two* implicit batches, so that the client could properly handle exceptions potentially thrown by findCreditAccount.

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 48.0 Mbps, 252ms latency wireless network. To ensure accuracy, all the benchmarks were repeated between 5000 to 10000 times with the results averaged.

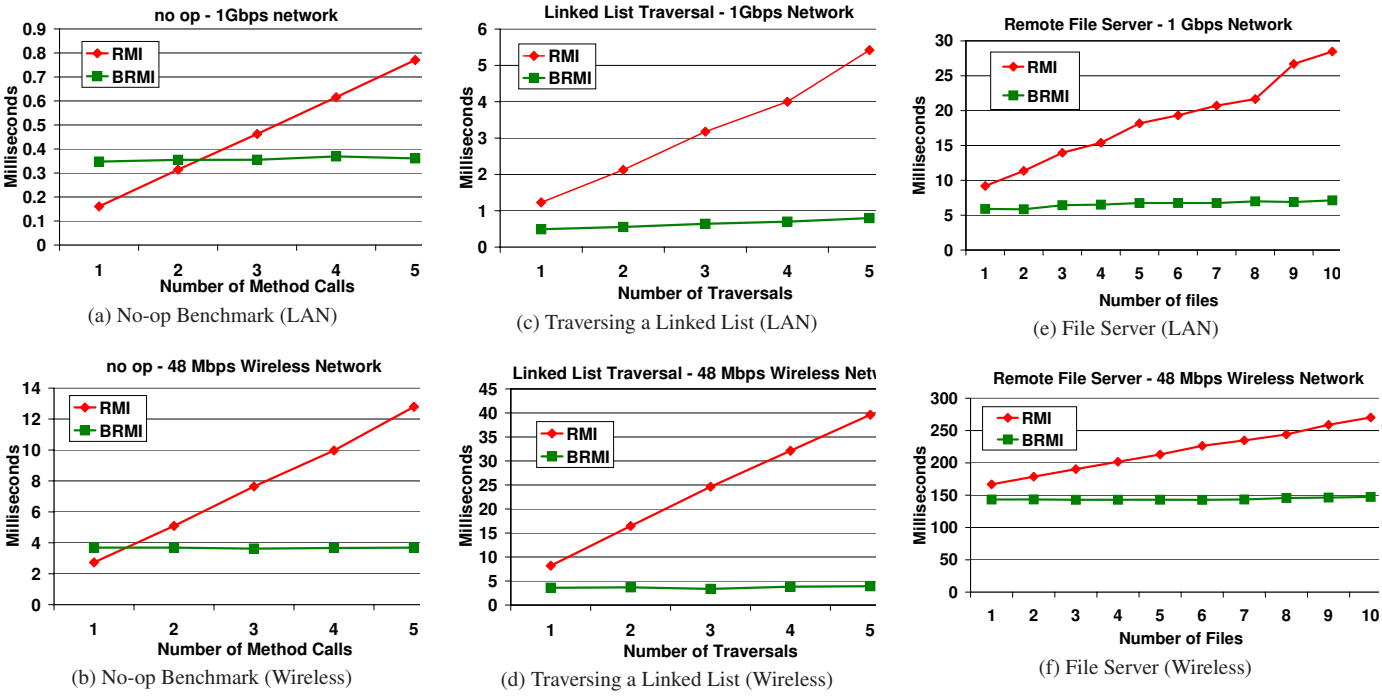
5.3 Micro benchmarks

5.3.1 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 1a and 1b 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.



(a) No-op Benchmark (LAN)

(c) Traversing a Linked List (LAN)

(e) File Server (LAN)

(b) No-op Benchmark (Wireless)

(d) Traversing a Linked List (Wireless)

(f) File Server (Wireless)

Figure 1: Performance of BRMI and RMI for different applications and network conditions

5.3.2 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 1c and 1d 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 BRemoteList reference to a client BRMI stub, thus completely avoiding any network transfer.

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. As per Figure 2, even without batching, BRMI consistently outperforms RMI, but with the BRMI execution time growing linearly. The BRMI version does not suffer the overheads of marshaling a RemoteList object, used as a return value.

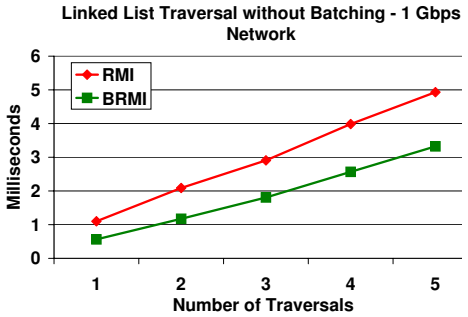


Figure 2: Linked List Traversal (Batches Size 1)

5.4 Remote File Server Macro Benchmark

This benchmark evaluates the performance of the Remote File Server, 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 taken by transferring n files from the server, with the total size of all the files equal to 100KB. Figures 1e and 1f shows that BRMI achieves several orders of magnitude performance increase over RMI for most in-

put parameters (i.e., number of files to retrieve from server), with the advantages present in both configurations.

6 Related Work

Explicit Batching. Software design patterns [13] for *Remote Facade* and *Data Transfer Object* (also called Value Objects [3]) 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 invocations of arbitrary methods, rather than just accessors and getters.

The DRMI system [15] 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 aggregation, BRMI supports array cursors, custom exception handling, conditionals, and loops. 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 [12] present analytic performance models for RPC and its optimizations including batched futures as well as 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, conditionals, and loops.

Sometimes a communication protocol defines batches directly, as is in the compound procedure in Network File System (NFS) version 4 Protocol [20], which combines 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 [23, 9].

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 [10]. Web services choreography [17] 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 [5] 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 [11] to minimize the aggregate latency of multiple inter-dependent remote methods. DJ [1] 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* reduce the aggregate latency of multiple remote methods [6]. 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. We extend batched futures to include support for cursors, conditionals, and exception policies.

Yeung and Kelly [8] 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. The implicit mechanism will combine any possible set of operations into a batch. However, it is not clear how the array cursors, conditionals, and loops of BRMI could fit into an implicit batching model.

Future RMI [2] communicates asynchronously to speed up RMI in Grid environments, when one remote method is invoked on the result of another. Remote results of a batch are not transferred over the network, remaining on the server to be used for subsequent method invocations.

Asynchronous RMI. Another approach to improving the performance of RMI is asynchronous dispatch of remote calls, as is in ProActive [4]. Its asynchronous remote calls also return futures, and the client waits for a result by blocking on a future. Asynchronous RMI provides no performance benefits for a chain of calls o.m1().m2() to remote objects. BRMI can effectively batch such a chain of remote calls, thereby reducing their aggregate latency.

7 Conclusion

This paper presents explicit batching for distributed objects, a general mechanism for clients to optimize access to distributed objects. The server infrastructure supports explicit batches for all clients, so that servers do not need to be optimized for specific clients. This is useful in settings where client needs cannot be predicted, or change over time.

Clients must be rewritten to use explicit batch interfaces, but 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, conditionals and loops to support operations that cannot be performed in a single batch.

We implemented explicit batches in Batched Remote Method Invocation (BRMI), an extension to Java RMI. A tool automatically creates batch interfaces from normal remote interfaces. It it interesting to note that BRMI avoids use of traditional object proxies, except for the root of a batch. The design also avoids all use of mobile code. Future work will explore language support for batches and integration of batches with transactions.

We evaluated the applicability and performance of BRMI by converting several third-party RMI applications to use BRMI. Benchmarks of these applications and several micro-benchmarks demonstrate that BRMI outperforms RMI and scales significantly better when clients make multiple remote calls.

Availability: BRMI can be downloaded from `http://research.cs.vt.edu/vtspaces/brmi`

References

[1] A. Ahern and N. Yoshida. Formalising Java RMI with explicit code mobility. In *Proc. of OOPSLA '05*, pages 403–422, New York, NY, USA, 2005. ACM.

[2] M. Alt and S. Gorlatch. Adapting Java RMI for grid computing. *Future Generation Computer Systems*, 21(5):699–707, 2005.

[3] D. Alur, J. Crupi, and D. Malks. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall PTR, 2003.

[4] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming, Deploying, Composing, for the Grid. Springer-Verlag, January 2006.

[5] A. P. Black, N. C. Hutchinson, E. Jul, and H. M. Levy. The development of the Emerald programming language. In *HOPL III*, pages 11–1–11–51, 2007.

[6] P. Bogle and B. Liskov. Reducing cross domain call overhead using batched futures. *ACM SIGPLAN Notices*, 29(10):341–354, 1994.

[7] N. Brown and C. Kindel. Distributed Component Object Model Protocol–DCOM/1.0, 1998. Redmond, WA, 1996.

[8] K. Cheung Yeung and P. Kelly. Optimising Java RMI Programs by Communication Restructuring. In *ACM Middleware Conference*. Springer, 2003.

[9] W. Cook and J. Barfield. Web Services versus Distributed Objects: A Case Study of Performance and Interface Design. In *the IEEE International Conference on Web Services (ICWS'06)*, pages 419–426, 2006.

[10] C. Demarey, G. Harbonnier, R. Rouvoy, and P. Merle. Benchmarking the Round-Trip Latency of Various Java-Based Middleware Platforms. *Studia Informatica Universalis Regular Issue*, 4(1):7–24, 2005.

[11] H. Detmold, M. Hollfelder, and M. Oudshoorn. Ambassadors: structured object mobility in worldwide distributed systems. In *Proc. of ICDCS'99*, pages 442–449, 1999.

[12] H. Detmold and M. Oudshoorn. Communication Constructs for High Performance Distributed Computing. In *Proceedings of the* 19^{th} *Australasian Computer Science Conference*, pages 252–261, 1996.

[13] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[14] P. Heller. *Java 1.1 developer's handbook*. SYBEX San Francisco, CA, 1997.

[15] E. Marques. A study on the optimisation of Java RMI programs. Master's thesis, Imperial College of Science Technology and Medicine, University of London, 1998.

[16] The Object Management Group (OMG). *The Common Object Request Broker: Architecture and Specification*, 1997.

[17] C. Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, 2003.

[18] E. Pitt and K. McNiff. *Java.RMI: The Remote Method Invocation Guide*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2001.

[19] M. Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principle. In *Proceedings of ICDCS'86*, pages 198–204, 1986.

[20] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. Network File System (NFS) version 4 Protocol, 2003.

[21] Sun Microsystems. *Java Remote Method Invocation Specification*, 1997.

[22] Sun Microsystems. *Dynamic proxy classes specification*, 1999.

[23] W. Vogels. Web services are not distributed objects. *Internet Computing, IEEE*, 7(6):59–66, 2003.