Remote Batch Invocation for Compositional Object Services

Ali Ibrahim[2], Yang Jiao[1], Eli Tilevich[1], and William R. Cook[2]

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

Abstract. Because Remote Procedure Calls do not compose efficiently, designers of distributed object systems use Data Transfer and Remote Façade patterns to create large-granularity interfaces, hard-coded for particular client use cases. As an alternative to RPC-based distributed objects, this paper presents *Remote Batch Invocation* (RBI), language support for explicit client-defined batches. A Remote Batch statement combines remote and local execution: all the remote code is executed in a single round-trip to the server, where all data sent to the server and results from the batch are communicated in bulk. RBI supports remote blocks, iteration and conditionals, and local handling of remote exceptions. RBI is efficient even for fine-grained interfaces, eliminating the need for hand-optimized server interfaces. We demonstrate RBI with an extension to Java, using RMI internally as the transport layer. RBI supports large-granularity, stateless server interactions, characteristic of service-oriented computing.

1 Introduction

The Remote Procedure Call (RPC) has long been the foundation of language-level approaches to distributed computing. The idea is simple: replace local calls with stubs that transfer the procedure call to a remote machine for execution. RPC has been generalized for objects to create distributed object systems, including Common Object Request Broker Architecture (CORBA) [22], the Distributed Component Object Model (DCOM) [8], or Java Remote Method Invocation (RMI) [29]. Stubs are defined on a local object that acts as a proxy for a remote object. One advantage of this approach is that it does not require language changes, but can be implemented using libraries and stub generator tools.

Standard object-oriented designs, which focus on flexibility and extensibility through the use of fine-grained methods, getters and setters, and small objects, do not perform well when distributed remotely. Every method call on a remote proxy is a round trip to the server. To achieve suitable performance, remote objects must be designed according to a different set of principles[3]. Data Transfer Objects and Remote Façades are used to optimize data transfer and combine operations to reduce the number of round trips [18]. One effect of this approach is that servers and protocols are hard-coded to support specific client invocation patterns. If a client changes significantly, then the entire system, including the server and its interfaces, must be redesigned.

[2] This work was supported by the National Science Foundation under Grant CCF-0448128.

[3] Approaches using asynchronous messaging are discussed in related work

This paper presents *Remote Batch Invocation* (RBI), a new approach to distributed object computing. Remote Batch Invocation allows multiple calls on remote objects to be invoked in a batch, while automatically transferring arguments and return values in bulk. The following example uses a Remote Batch in Java to delete low-rated albums from a personal online music database.

```
int minimum = 5;
Service musicService = new Service("MusicCloud", Music.class);
batch (Music favoriteMusic : musicService) {
  for (Album album : favoriteMusic.getAlbums())
    if (album.rating() < minimum) {
      System.out.println("Playing: " + album.getTitle());
      try {
        album.play();
      } catch (Exception e) {
        System.out.println("error: " + e.getMessage());
}}}
```

The batch mixes local and remote computation. In this case, all the computation is remote except the two calls to `System.out`. The semantics of Java is modified within the batch to first perform all remote operations, then perform all local operations. Thus the typical ordering between local and remote statements is not necessarily preserved. For example, all of the albums are played before any of the names are printed. All loops and conditionals are executed twice: once on the server and then again on the client. Exceptions on the server terminate the batch by default, and raise the error in the analogous execution point on the client.

A remote batch transfers all data between client and server in bulk. In this case, just the `minimum` rating is sent to the server. The server returns a list of all titles of played albums. But it also returns a boolean for each album indicating whether it was played. In general, any number of primitive or serializable values can be transfered to and from the server. Remote Batch Invocation creates appropriate Data Transfer Objects and Remote Façades on the fly, involving any number of objects and methods. Standard Java objects can be published as a batch service by adding a single line of code. The semantics of the batch statement require that only a single remote invocation is made in the lexical block. This strong performance model is important, because the cost of remote invocations may be several orders of magnitude higher than local invocations.

We demonstrate Remote Batch Invocation with an extension to Java. A source-to-source translator converts the **`batch`** statement to plain Java which uses Batch Execution Service and Translation (BEST), our middleware library for batched execution using Java RMI. Remote Batch Invocation is not tied to RMI, but could also be implemented using other middleware transport, for example web services or mobile objects. A server can publish a remote service by making a single library call.

The performance benefits of batching operations are well-known, especially in high-latency environments. We evaluate our language extension by comparing it with other approaches to batching such as implicit batching, mobile code, and the Remote Façade pattern.

In summary, Remote Batch Invocation is a new approach to distributed objects that supports service-orientation rather than remote procedure calls and proxies. The fundamental insight is that remote execution need not work at the level of procedure calls, but can instead operate at the level of blocks, with bulk transfer of data entering and

leaving the block. Unlike traditional distributed objects that maintain server side state, Remote Batch Invocation has a stateless execution model that is characteristic of service oriented computing [20, 17].

2 Remote Batch Invocation

Remote Batch Invocation allows clients to combine remote operations into a single remote invocation. We will illustrate the features of Remote Batch Invocation by example. The basis of our examples is a sample remote service described by Fowler in *Patterns in Enterprise Application Architecture* [18]. This simple remote music service is comprised of three classes: Album, Artist, and Track as shown in Figure 1. The `Album` interface also provides the `play` method which returns the lyrics on the album and plays the album on a sound system.

```
interface Album {
  String getTitle();
  void setTitle(String title);
  Artist getArtist();
  void setArtist();
  Track[] getTracks();
  void addTrack(Track t);
  void removeTrack(Track t);
  String play();
}
```

A natural remote interface to these three classes is shown below:

```
interface Music {
  Album createAlbum(String id, String title);
  Album getAlbum(String id);
  Artist addArtist(String id, String name);
  Artist getArtist(String id);
  Track createTrack(String title);
}
```

Using the `Music` interface, a client can create and find artists and albums as well as create tracks. A client may update object fields using the appropriate setters. We will use this interface for our Remote Batch Invocation examples.

Unfortunately, this natural interface is too fine-grained in a system where individual method calls are expensive. Using the Remote Façade and Data Transfer patterns, Fowler wraps the Music interface:

```
interface FowlerMusic {
  String play(String id);
  AlbumDTO getAlbum(String id);
  void createAlbum(String id, AlbumDTO dto);
  void updateAlbum(String id, AlbumDTO dto);
  void addArtistNamed(String id, String name);
  void addArtist(String id, ArtistDTO dto);
  ArtistDTO getArtist(String id);
}
```

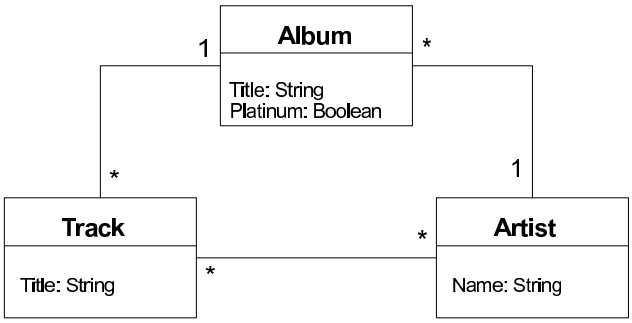

Fig. 1: Fowler Album Class Diagram

`FowlerMusic` is a Remote Façade for the `Music` interface. For example, the `FowlerMusic.play` method is simply calling the `Music.getAlbum` method followed by the `Album.play` method. The `AlbumDTO`, `ArtistDTO`, and `TrackDTO` are data transfer objects (DTO) that transfer information in bulk to and from the remote server. Fowler also defines `AlbumAssembler`, which maps between DTOs and objects residing on the server.

```
class AlbumAssembler {
  public AlbumDTO writeAlbum(Album subject) {
    AlbumDTO result = new ALbumDTO();
    result.setTitle(subject.getTitle());
    result.setArtist(subject.getArtist().getName());
    writeTracks(result, subject);
  }
  void writeTracks(AlbumDTO result, Album subject) { ... }
  void writePerformers(TrackDTO result, Track subject) { ... }
  public void createAlbum(String id, AlbumDTO source) {
    Artist artist = Registry.findArtistNamed(source.getArtist());
    if (artist == null) throw new RuntimeException(...);
    Album album = new Album(source.getTitle(), artist);
    createTracks(source.getTracks(), album);
    Registry.addAlbum(id, album);
  }
  void createTracks(TrackDTO[] tracks, Album album) { ... }
  void createPerformers(Track newTrack, String[] performers) { ... }
}
```

Although `AlbumAssembler` encapsulates the logic of mapping between DTO and model objects, it is not generic, containing a hard-coded decision about the DTO content. In the book, Fowler decides to have the Album DTO provide all the information about a single album.

The next sub-sections give examples of using Remote Batch Invocation for batch data retrieval, batch data transfer, loops, branching, and exceptions.

2.1 Batch Data Retrieval

A simple client may want to print the title and name of the artist for an album. With the fine-grained `Music` interface, the client must execute four remote calls: a call to find the album, a call to get the title of the album, a call to get the artist for the album, and a call to get the name of the artist for the album.

Using Remote Batch Invocation, the client can use the `Music` interface while still executing a single remote call. The input to the remote batch is the id of the album "1". The output of the remote batch is the title of the album and the name of the artist of the album. A remote batch can combine an arbitrary number of method calls as long as they are invoked on objects transitively reachable from the root object of the batch, in this case `music`.

```
batch (Music music : musicService) {
  final Album album = music.getAlbum("1");
  System.out.println("Title: " + album.getTitle());
  System.out.println("Artist: " + album.getArtist().getName());
}
```

The same client using the remote façade `FowlerMusic` executes a single remote method `getAlbum` which returns `AlbumDTO`. For this client, the DTO is an over-approximation of the data needed; a Remote Façade optimized for this client would need another DTO for albums that only provides the title and artist name.

```
AlbumDTO album = music.getAlbum("1");
System.out.println("Title: " + album.getTitle());
System.out.println("Artist: " + album.getArtistName());
```

For other clients, the DTO may be an under-approximation of the data needed. For example, this client prints the title of two different albums.

```
batch (Music music : musicService) {
  final Album album = music.getAlbum("1");
  System.out.println("Title: " + album.getTitle());
  final Album album = music.getAlbum("2");
  System.out.println("Title: " + album.getTitle());
}
```

`FowlerMusic` does not contain a method that matches this client pattern. Consequently, the same client using `FowlerMusic` must make an additional remote call compared to using Remote Batch Invocation. Alternatively, the `FowlerMusic` interface can be changed to include a method that takes two album IDs as input and returns a new DTO containing two fields representing the titles of the input albums. This highlights one of the disadvantages of the Remote Façade pattern; it creates a non-functional dependency between the server interface and the client call patterns.

2.2 Batch Data Transfers

Remote Batch Invocation also allows clients to transparently transfer data in bulk to the server. The following code creates `Album`, `Artist`, and `Track` objects and wires them together. The input to the remote batch is all the information about the album, artist, and track to be created and there is no output. The actual construction of the objects and method calls occur entirely on the server.

```
batch (Music music : musicService) {
  final Album album = music.createAlbum("2", "First Album");
  final Artist artist = music.addArtist("2", "John Smith");
  album.setArtist(artist);
  final Track track = music.createTrack("First track");
  track.addPerformer(artist);
  album.addTrack(track);
}
```

A client using `FowlerMusic` can also create the objects using a single remote invocation using the appropriate DTOs.

```
AlbumDTO album = new AlbumDTO("First Album");
AlbumDTO artist = new ArtistDTO("2", "John Smith");
album.setArtist(artist);
TrackDTO track = new TrackDTO("First Track");
track.addPerformer(artist);
album.addTrack(track);
music.createAlbum("2", album);
```

A drawback to using data transfer objects for creating and updating objects, is that DTO is under-specifying some of the semantics of the operation. In particular, the DTO does not tell the server whether the artist object is an artist object which should be created or if it already exists. This is a well-known problem in data mapping and commonly arises in distributed systems. A common approach and the one taken by Fowler in his book, is to specify a convention to either always create objects, always use existing objects, or create an object if it does not already exist. Another approach is to enrich the DTO with *status* fields for each normal field that specify the right semantics. Sometimes this status field is encoded into the field, for example, by using **`null`** as a special value. A related problem is updating objects if the client only has a partial description of the object. The client must be able to update the subset of fields which are known, but not the fields which are unknown.

The remote batch is more explicit in that specifies that the `artist` is a new `Artist` object. If the client wanted to reference an existing artist the code would be rewritten as follows:

```
batch (Music music : musicService) {
  final Album album = music.createAlbum("2", "First Album");
  final Artist artist = music.getArtist("2");
  album.setArtist(artist);
  final Track track = music.createTrack("First track");
  track.addPerformer(artist);
  album.addTrack(track);
}
```

2.3 Loops

So far, we have shown that Remote Batch Invocation supports straightline code. However, it is common for a client to need more complex logic involving branching and loops. Remote Batch Invocation allows for remoting of the enhanced **`for`** loop introduced in Java 1.5 if the collection can be evaluated remotely. If data from the iterations

is needed locally, the remote batch constructs a data transfer object with an array of the data needed and transparently maps it on the client. Below is a simple example which shows how explicit batching can operate over arrays. The input to the remote batch is simply the id of the album and output is the title of all of the tracks, the name of all of the performers on the tracks, and the lyrics returned by the `play` method.

```
batch (Music music : musicService) {
  final Album album = music.getAlbum("1");
  System.out.println("Tracks: ");
  for (Track t : album.getTracks()) {
    System.out.print(t.getTitle());
    System.out.println(',');
    System.out.print("Performed by: ");
    for (Artist a : t.getPerformers()) {
      System.out.print(a.getName());
      System.out.print(' ');
    }
    System.out.print('\n');
  }
  System.out.println("Song: " + album.play());
}
```

The `FowlerMusic.getAlbum` method in Remote Façade nearly provides all the functionality required by this client; however, it does not include a call to the `Album.play` method.

2.4 Branching

Conditional statements, including **`if`** and **`else`**, are remoted if their condition is a remote operation. Below is a simple example that shows such a remoted conditional statement also containing the primitive operator `&&`.

```
batch(Music music : musicService) {
  final Album album = registry.getAlbum("1");
  if (album.getName().startsWith("A")
      || album.getName().startsWith("B")) {
    album.play();
    System.out.print("Title starts with A or B: " + album.getTitle());
  } else {
    System.out.print("Title does not start with A or B: "
       + album.getArtist().getName());
  }}
```

RBI supports boolean and numeric primitive operators, both unary and binary. Conditional code can also be included as part of operations on collections. In that case, the conditions are reevaluated on each iteration over a collection. The following example adds albums composed by Yo-Yo Ma to the favorites collection.

```
for (Artist a : t.getPerformers()) {
  if (a.getName().equals("Yo-Yo Ma")) {
    favorites.addArtist(a);
  }}
```

2.5 Exceptions

Remote Batch Invocation separates exceptions caused by failures in communication from logical exceptions that arise when executing the statements in the batch. The **batch** statement itself can raise network exceptions, which must be handled by the surrounding context. If there are no network errors, then exceptions raised by statements in the batch can be handled in the client.

Within a **batch**, a remote operation can raise an exception on the server that will terminate the batch. The thrown exception will be raised in the corresponding execution point on the client. The client must use exception handlers as in regular Java code. In addition, the execution of a remote batch may result in a `RemoteException` that can be handled by wrapping an entire **batch** block with a **try/catch** block.

For example, the following code extends an earlier example to include an exception handler when trying to play an album, and another handler that deals with network and communication errors raised at any point of executing the batch.

```
try {
  batch (Music favoriteMusic : musicService) {
  ...
    try {
      album.play();
    } catch (PermissionError pe) {
      System.out.println("No permission to play album"
        + album.getTitle());
    }
  } //end batch
} catch (RemoteException re) {
  System.out.println("Error communicating batch.");
}
```

The default behavior of a batch is to abort processing when an exception is thrown. As future work, we would like to be able to apply a different exception policy, for example to continue execution or restart the batch. Batches also provide a natural unit of atomic execution. In many cases it is desirable for the entire batch to succeed or fail, so that incomplete operations are never allowed. One way to achieve this is to use transactional memory on the server [7].

Even so, it is possible for the batch to succeed on the server but for a communication error to prevent the client from completing the batch. A standard two-phase commit could be used to ensure that both the server and client parts of the batch have executed to completion. These topics are beyond the scope of our current research, but we do not see any obstacles to combining RBI with distributed transactions.

2.6 Service Implementation

Implementing a Remote Batch Invocation service is much simpler than implementing a server using traditional distributed object middleware, including RMI or CORBA. There is no need to create method stubs. Instead, the server simply registers a root object with a single call after creating the server implementation object.

```
Music musicServer = new MusicImpl(...);
rbi.Server server = new rbi.Server("MusicCloud", musicServer);
```

The client connects to this service by using the same name and interface.

```
rbi.Service musicService =
  new rbi.Service("MusicCloud", Music.class);
```

As in most distributed systems, interface mismatches between client and server are detected at runtime. Standard Java interfaces define the service contract.

2.7 Service-Oriented Interaction

Remote Batch Invocation supports a service-oriented style of interaction, so it does not support object proxies. This is not a problem for many client/server interactions, which can be naturally accomplished in a single round-trip. These interactions have the following pattern:

client $\overset{input}{\longrightarrow}$ server* $\overset{results}{\longrightarrow}$ client

The client sends any number of inputs to the server, which performs multiple actions and returns any number of results to the client. There may be cases; however, when a server computation depends upon client input *and* previously defined server objects.

This situation is easily handled in distributed object systems like CORBA and RMI, since each server operation is controlled by the client and it can use proxies to refer to the intermediate server results needed in the last step.

This interaction pattern requires some other solution in a stateless service-oriented system. The simplest approach is to have the second server batch reload or recreate the server objects that were defined in the first batch. The server may also provide public identifiers for its objects. The first *results* can include a server object identifier, which is used in the second batch to relocate the necessary server object. These patterns have been studied extensively in the context of service-oriented computing [20, 17].

2.8 Allowed Remote Operations

Any Java code may appear inside the batch block; however, the compiler enforces some data flow restrictions described in Section 3. Many Java constructs such as constructor calls, casts, **`while`** loops, and assignments cannot be remoted; they are always executed on the client. Future work may relax some of these restrictions. If remote assignments were allowed, then it would be possible to aggregate (e.g. sum or average) over collections remotely. General loops could also be remoted without significant changes to the model.

Exceptions are a special case. The remote batch cannot catch exceptions remotely, but it does propagate them to the client in the original location of the remote operation that produced the exception. In this way, the client can catch exceptions raised remotely and handle them locally.

Keeping the remoteable constructs simple and as universal as possible increases the viability of using RBI against remote interfaces written in other languages.

3 Semantics

Our Java implementation of Remote Batch Invocation uses the following syntax:

batch (*Type Identifier* : *Expression*) *Block*

The *Identifier* specifies the name of the root remote object. The *Expression* specifies the service which will provide the root remote object. The *Block* specifies both remote and local operations. A remote operation is an expression or statement executed on the server. All remote operations inside the batch block are executed in sequence followed by the local operations in sequence. A single remote call is made which contains all of the remote operations. This is the key property as it provides a strong performance model to the programmer albeit lexically scoped. Exceptions in a remote operation are re-thrown in the local operation sequence at the original location of the remote operation. If the remote operations fail due to a network error, then an exception is thrown before any of the local operations execute. Operations inside the batch block are reordered and it is possible that the block executes differently as a batch than it normally would. The compiler does try to identify some of these cases and warn the programmer, however, it is up to the programmer to be aware of the different Java semantics inside the batch block.

Each expression in the batch is marked as *local* or *remote*. Local expressions are further subdivided into *static locals* and *non-static locals*. Remote expressions execute on the server, possibly with input from static local expressions. Local expressions execute on the client, possibly with output from remote expressions. Static local expressions are literals and variable expressions defined outside of the batch and not assigned within the batch before their use. All other local expressions are non-static.

The compiler determines the location of an expression statically. A component of this analysis is a forward flow-sensitive data-flow analysis that maps variables to locations. Locations are ordered as a small lattice where *static local* $<$ *remote* $<$ *non-static local*. The $\uplus$ operator adds or changes a mapping for a variable. The $pred$ function returns the predecessors of a statement node in the control flow graph. For simplicity, we will assume in this paper that all assignments are statements; however, in Java they are actually expressions. The data flow analysis is defined in Figure 2.

The **batch** variable is remote. Variables only assigned outside the batch are static locals. Variables declared final and initialized with remote expressions are remote. All other variables inside a batch block are non-static locals. Assignments may change the mapping of a variable up the lattice of locations. For this analysis, the only case where this happens is a variable mapped as a static local may be remapped as a non-static local. It cannot happen for variables mapped as remote, because final variables cannot be reassigned.

Figure 3 defines the $location$ function which maps expressions to locations. To determine the location of a variable expression, the analysis looks up the variable name in the result of the data flow analysis flowing into the statement containing the variable expression. The mutual definition of $location$ and gen introduces a cyclic dependency which is resolved by taking the fix point of the two functions starting with the bottom value of our location lattice (static locals). The location of a primitive operation is the join of the locations of the operands. The location of an instance method call expression

$$
\begin{aligned}
n, m &\in Statement \\
e &\in Expression \\
inBatch(e) &= \begin{cases} true & \text{e is an expression inside a batch statement} \\ false & \text{otherwise} \end{cases} \\
varBatch(e) &= \begin{cases} v & \text{e is an expression inside a batch statement of the form } \mathsf{batch}(T\ v : e) \\ undefined & \text{otherwise} \end{cases} \\
s \uplus \mathsf{nil} &= s \\
s \uplus [v \mapsto l] &= \begin{cases} s \cup [v \mapsto l] & [v \mapsto \_] \notin s \\ (s - [v \mapsto k]) \cup [v \mapsto l] & [v \mapsto k] \in s \end{cases} \\
in[n] &= \bigcup_{m \in pred(n)} out[m] \\
out[n] &= in[n] \uplus gen(n) \\
gen(n) &= \begin{cases} [v \mapsto \text{remote}] & n = [\![\mathsf{batch}(T\ v : e)]\!] \\ [v \mapsto \text{static local}]) & n = [\![v = e]\!] \wedge !\ inBatch(n) \\ [v \mapsto \text{non-static local}] & n = [\![v = e]\!] \ \wedge \ varBatch(n) \neq v_b \\ [v \mapsto location(e)] & n = [\![\mathsf{final}\ v = e]\!] \ \wedge \ varBatch(n) \neq v_b \\ \mathsf{nil} & otherwise \end{cases}
\end{aligned}
$$

Fig. 2: Analysis of Java to identify local and remote variables

$$
\begin{aligned}
location([\![v]\!]) &= in[Stmt(v)](v) \\
location([\![e_1\ \mathsf{op}\ e_2]\!]) &= location(e_1) \sqcup location(e_2) \\
location([\![o.m(\bar{e})]\!]) &= location(o) \\
location([\![\_]\!]) &= \begin{cases} \text{non-static local} & inBatch(\_) \\ \text{static local} & !\ inBatch(\_) \end{cases}
\end{aligned}
$$

Fig. 3: Location of Java expressions

is the location of the target of the method call. All other expressions inside or outside the batch statement are non-static local or static local respectively.

One important thing to note in the rules is that general assignment is not supported in the remote batch. Therefore, variables are only remote if they correspond to the **`batch`** variable or if they are **`final`** and assigned remote expressions. Java 1.5 **`for`** statements are executed remotely if their collection is a remote expression. A remote **`for`** loop is replayed locally to support local expressions or statements inside the loop. Similarly, conditional statements are executed remotely if their condition is a remote expression. A remote conditional is replayed locally to support local expressions or statements inside the **`if`** statement.

Data is passed by value from the client to the server and from the server to the client. For example, the remote identity function returns a copy of the local argument. This implies that all input and output values of the batch must be serializable and specifically in Java implement the Serializable interface. Remote values not used locally are not subject to this restriction. Remote expressions do have identity as long as they are part of computations on the server, and similarly local expressions have the normal notion of identity in Java.

The compiler rejects all programs in which the remote operations cannot be legally moved above the local operations. For example, parameter expressions in remote method calls cannot contain local variables defined within the batch. The compiler also rejects some programs in which moving the remote operations above the local operations might result in non-intuitive behavior. For example, parameter expressions in remote method calls should not have their value changed in the local operations. The following are considered illegal expressions by the compiler.

- Method invocations on remote values that have a parameter which is a non-static local expression or is not serializable.
- Expressions with remote locations inside of an **`if`** block where the condition is a local expression.
- Expressions with remote locations inside of a loop construct where the condition is local.
- Nested batch statements.

One design goal was to ensure that programmers could easily understand the semantics of the **`batch`** construct. To that end, our analysis uses a very simple local data flow analysis and is lexically scoped. This may allow non-intuitive programs to be accepted by the compiler, because they change the state of static local expressions via different threads, heap aliasing, or local method calls [19]. The following example shows a case where the compiler accepts a program that behaves non-intuitively from the point of view of the programmer.

```
StringBuilder sb = new StringBuilder();
sb.append("My Album");
batch(Music music : musicService) {
  m(sb);
  music.createAlbum("1", sb);
}
...
void m(StringBuilder sb) { sb.append(": Blues"); }
```

The programmer might expect that the remote method call `createAlbum` will be passed the string `"My Album: Blues"`, but in a remote batch it will be passed the string `"My Album"`, because the remote method call will occur first. Unfortunately Java reflection, virtual methods, and dynamic class loading all complicate whole program analysis. Our local lexical analysis trades off catching some non-intuitive behavior to gain simplicity, practicality, and locality.

4 Implementation

Support for Remote Batch Invocation in Java is implemented as a source to source translator which takes code containing remote batch constructs and translates them into regular Java code. The output of the source to source translator uses a script recording API that sends the remote operations as a single batch to the remote server. In the current implementation, the script recorder uses the transport layer and the service discovery mechanism of Java RMI. The support system for RBI is called BEST, which is an acronym for Batch Execution Service and Translation. BEST is implemented as a layer on top of Java RMI, without changes to the Java language or runtime. First, we discuss the translation of the batch syntax. Then, we focus on the implementation issues of BEST, its underlying techniques, and its integration with Java RMI. Section 5 quantifies BEST performance benefits.

4.1 Language Translation

The source to source translator is implemented as an extension to JastAddJ [16]. JastAddJ is a Java compiler based on JastAdd and written as a circular attribute grammar. JastAdd provides several useful features. As a circular attribute grammar, many static analyses can be expressed naturally and fixed point computations are handled by the JastAdd engine. In addition, JastAdd provides many aspect-oriented features which allow composition of different analyses and language features in a a modular fashion. The data flow analysis is implemented on top of a control flow graph module written by the authors of JastAddJ for Java 1.4. We modified the their module slightly to add support for the new **`batch`** construct and to support Java 1.5. For each expression, the translator computes its location as described in Section 3.

The translator traverses the program abstract syntax tree (AST) downwards starting from the root AST node. Outside of a batch, the translator does not change the Java code. Inside a batch, the translator always produces two code strings, one for the remote operations and one for the local operations. Once the entire batch is translated, some boilerplate code to setup the batch is generated first, then the remote operations are inserted, then a call to execute the batch is generated, and finally the local operations are inserted. While translating code in a batch, the translator has two different modes of operation. Initially the translator is in local mode. Expressions in local mode produce no remote operations and produce themselves as local operations. Most statements behave similarly except for remote loops and remote conditionals which produce both remote and local operations. Once the translator reaches an expression whose location is remote, it binds that remote value to a temporary variable as a remote operation and enters remote mode for that expression. The translator also adds a local operation which invokes the `get` method on the temporary variable. In remote mode, the translator can safely assume all sub-expressions are remote operations.

```
Service musicService = new Service("MusicCloud", Music.class);
batch(Music music : musicService) {
  final Album album = music.getAlbum("1");
  if (album.getTitle().startsWith("A")) {
    System.out.println("Tracks:");
    for (Track t : album.getTracks()) {
      System.out.print(' ');
      System.out.print(t.getTitle());
    }
  } else {
    System.out.print("Title does not start with A: "
      + album.getArtist().getName());
}}
```

Fig. 4: RBI source code

```
// Remote part
Service service$ = musicService;
{ Batch batch$ = service$.getRoot();
Handle album$73751 = batch$.doInvoke(batch$,"getAlbum",
  new Class[] {String.class}, new Object[] {"1"});
Handle var$0 = batch$.doInvoke(
  batch$.doInvoke(album$73751,"getTitle", null, null),
  "startsWith", new Class[] {String.class}, new Object[] {"A"});
batch$.rIf(var$0);
cursor.Cursor t$86036$Cursor = batch$.createCursor(
         batch$.doInvoke(album$73751,"getTracks", null, null));
Handle var$1 = t$86036$Cursor.doInvoke(
                       t$86036$Cursor,"getTitle", null, null);
batch$.rElse();
Handle var$2 =
  batch$.doInvoke(batch$.doInvoke(album$73751,"getArtist", null, null),
    "getName", null, null);
batch$.rEnd();
batch$.flush();
// Local part
if((Boolean)var$0.get()){
  System.out.println("Tracks:");
  while (t$86036$Cursor.next()) {
    System.out.print(' ');
    System.out.print((String)var$1.get());
  }
} else {
  System.out.print("Title does not start with A: "
    + (String)var$2.get());
}}
```

Fig. 5: Translation of Figure 4

Figure 4 shows a RBI program which uses many of the supported features. Figure 5 shows the translation into Java code which uses BEST. An interesting part of the translation is how conditionals and loops require both remote and local operations.

4.2 BEST Client Interface

The main client interface of BEST is defined in Figure 6.

```
public interface Batch {
  public Handle doInvoke(Object obj, String method,
                  Class[] types, Object[] args);
  public Cursor createCursor(Handle value);
  public Handle unary(Ops op, Handle val1);
  public Handle binary(Ops op, Handle val1, Handle val2);
  public Handle constant(Object o);
  public Handle rIf(Handle condition);
  public Handle rElse();
  public Handle rEnd();
  public void flush();
}
```

Fig. 6: Interface to the BEST batch execution runtime

A `Batch` is a client object that represents a collection of statements. Method `flush` delineates the boundary of a batch. When `flush` is called, all the recorded statements are sent to the server in bulk, executed there, and the relevant results are returned back together. Each recorded statement returns a `Handle` which is a placeholder for a remote object, existing or created on the server. A `Handle` has two different semantics before and after `flush` is called. Before `flush`, a `Handle` serves as a placeholder for a result which has not yet been obtained. After `flush`, a `Handle` object holds a result of a remote operation that can be retrieved.

The `Batch` interface describes a script recording service. To add a method to be invoked remotely, the API provides the method `doInvoke`. The parameters of this method loosely mirror that of `Method.invoke` in the Java Reflection API. The method's parameters are deliberately weakly-typed to enable greater flexibility. This design choice fits well the BEST programming model, in which all the calls to the script recording API are automatically generated by the source-to-source translator, thereby ensuring that the resulting code is type safe.

The `Batch` interface also provides methods to express conditional remote control flow and operators. These methods are used to express conditions and operations used in a **`batch`** block. The translator maps Java conditional and primitive operators into regular methods (e.g., `rIf`, `rElse`, `binary`) that are recorded for remote execution.

The `makeCursor` method takes a `Handle` parameter and returns a `Cursor`, which represents an iteration context for the collection of objects existing on the server. The assumption for calling `makeCursor` is that its `Handle` parameter represents an `Iterable` object such as a `java.util.Collection` or an array.

The `Cursor` interface is implemented as follows:

```
public interface Cursor extends Batch {
  public boolean next();
  public void setPosition(int position)
                  throws IllegalArgumentException;
  public int getPosition();
}
```

Remote operations recorded on a `Cursor` interface will be replayed on each element of an `Iterable` collection on the server. After `flush`, the `Cursor` can be iterated to retrieve the results of remote operations for every element.

The end result of recording operations using the `Batch` interface is a list of method descriptors, which are serializable objects sent to the server. Each recorded operation is assigned a sequence number which acts as an identifier for that call. The sequence numbers are sent to the server, so that method arguments can be matched to prior method return values.

4.3 Batch Execution

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

The BEST server runtime decodes method descriptors, invokes batched methods one-by-one and returns the results back to the client. To implement conditional statements such as **`if`** and **`else`**, the BEST server interprets the operations by evaluating the specified conditional statements and changing the control flow of a batch based on their results. Similar strategy is applied to executing unary and binary operations. While at the script recording time on the client the operands are represented by handles, their actual values are obtained during the execution of a batch on the server. Then the interpretation simply operates on the actual values as was specified by the script.

Cursor operations are interpreted analogously to regular operations, with the exception that each recorded operation is executed on each element of an `Iterable` server object with the results stored in a table. The rows in the table correspond to the different variables associated with a cursor and the columns correspond to each iteration of the cursor.

4.4 Result Interpretation

For each non-cursor client `Handle`, the server returns a value, exception, or nothing. The server returns no value for a client `Handle` associated with an unexecuted remote operation. At most one `Handle` is assigned an exception, because the the remote batch is terminated by the first exception. If a `Handle` has an exception, rather than a value, then this exception is thrown when accessing its content.

For cursors, result interpretation is more complicated. Each time `next` is called on a `Cursor`, the `Handle` objects associated with it 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 `Handle`'s. `Handle`'s normally do not change value after they have been assigned, with the exception when they are created within a cursor–the `Handle` values may change on each iteration of the loop.

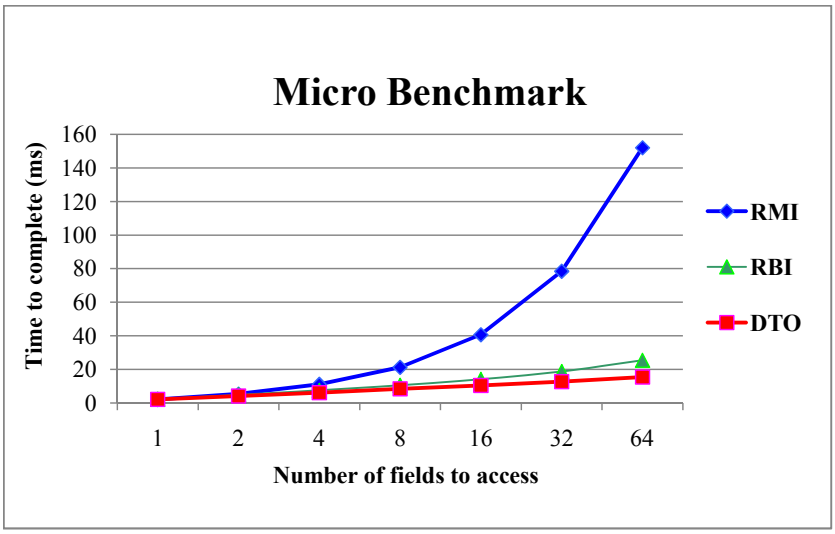

Fig. 7: Performance Comparison between RMI, RBI, and DTO versions

5 Performance

In essence, *Batch Remote Invocation* is a language level mechanism that optimizes remote communication by leveraging the improved bandwidth characteristics of modern networks [23], especially in high-latency environments. Although the performance benefits of batching remote operations are well-known and have been the target of several research efforts [6, 21, 9], the purpose of evaluating the performance of RBI is to ensure that the overhead of its runtime, BEST, does not impose an unreasonable performance overhead. The following benchmark uses data objects with different numbers of `String` fields: 1, 2, 4, 8, 16, 32, and 64. The benchmark emulates a common usage scenario, in which the client retrieves the object from the server and updates its fields. This scenario was implemented and measured using three different communication styles: plain RMI, a hand-coded DTO, and RBI. Figure 7 shows the performance numbers for each version.

All the experiments were run in the Windows XP version of JDK 1.6.0_13 (build 1.6.0_13-b03), with the server running Dual Core 3GHz processors, 2 GB of RAM, and the client running Dual Core 2.4GHz Processors, 2GB of RAM, connected via a LAN with a 1Gbps, 1ms latency network. The results represent the average of running each benchmark 1000 times after first running it 2000 times to warm the JVM. Warming the JVM ensured that the measured programs had been dynamically compiled before measurements.

As expected, the RMI version is the slowest, with its slope growing linearly at a fixed rate, as the number of fields increases. The DTO and RBI versions exhibit comparable performance, with DTO being faster by a small constant factor. These results are predictable, as the execution time is dominated by the number of remote calls performed by each version of the benchmark, and in most networking environments the latency of a remote call is several orders of magnitude larger than that of a local call.

The specific number of remote calls performed by each version of the benchmark is as follows. If f is the number of fields, the RMI version performs $2 * f$ remote calls (to get and set every field); the DTO version performs only 2 calls (i.e., getting and setting all fields in bulk); and finally, the RBI version performs exactly 1 remote call.

Even though the RBI version performs only one remote call, whereas the DTO version two, RBI is still slower due to the overhead imposed by its client and server runtime. To provide flexibility, BEST uses Java language features that are known to have a negative effect on performance, including reflection to locate and invoke methods as well as multiple `Object` arrays to pass parameters. In addition, the current implementation of BEST has not been fine-tuned for performance. Finally, the BEST overhead would be amortized more significantly in a higher-latency network environment. Compared to the hard-coded interface of DTO, RBI makes it possible to create a flexible DTO on the fly with the accompanying performance benefits due to the reduced network communication enabled by its service-oriented execution model.

6 Related Work

6.1 RPC Critique

Even though Remote Procedure Call (RPC) [32] has been one of the most prevalent communication abstracts for building distributed systems, its shortcoming and limitations have been continuously criticized [30, 36, 26]. Recently some experts even express the sentiment that RPC has had an overwhelmingly harmful influence on distributed systems development and wish that a different communication abstraction had become dominant instead [34]. A frequently mentioned alternative for RPC is asynchronous messaging and events, including publish-subscribe abstractions [12].

Despite all the criticisms of RPC and its object-oriented counterparts, exposing distributed functionality through a familiar procedure call paradigm has unquestionable convenience advantages. Remote Batch Invocation is an attempt to address some of the limitations of RPC, while retaining its advantages, without introducing the complications of asynchronous processing imposed by message- and event-based abstractions.

Among the main criticisms of RPC is its attempt to eliminate the distinction between the local and remote computing models, with respect to latency, memory access, concurrency, and partial failure [36]. By combining multiple operations into a single batch, RBI reduces latency. By executing all remote operations on the server in bulk, RBI maintains the local memory access model for method parameters. As future work, a transactional execution model can be combined with RBI to achieve an all-or-nothing execution property. And while batch invocations in RBI are synchronous, the resulting execution model is explicit, giving the programmer a clear execution and performance model.

6.2 Explicit Batching

Software design patterns [18] for *Remote Façade* and *Data Transfer Object* (also called Value Objects [3]) can be used to optimize remote communication. A *Remote Façade* allows a service to support specific client call patterns using a single remote invocation. Different Remote Façades may be needed for different clients. Remote Batch Invocation

provides a custom Remote Façade for each client as long as the client call pattern is supported as a single batch. A *Data Transfer Object* is a `Serializable` class that provides block transfer of data between client and server. As with the Remote Façade, different kinds of Data Transfer Objects may be needed by different clients. Remote Batch Invocation constructs an appropriate value object on the fly, automatically, as needed by a particular situation. Remote Batch Invocation also generalizes the concept of a data transfer object to support transfer of data from arbitrary collections of objects.

The DRMI system [21] aggregates RMI calls as a middleware library much like BEST. DRMI uses special interfaces to record and delay the invocation of remote calls. DRMI only supports simple call aggregation and simple branching, while Remote Batch Invocation and BEST also support cursors, primitive operations, and exception handling. Like BEST, DRMI requires that the programmer partition the remote and local operations themselves. This often forces the programmer to replicate loops and conditionals manually, whereas Remote Batch Invocation offers a more flexible style of programming and relies on the source to source translator to partition the program into remote and local operations.

Detmold and Oudshoorn [15] 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 Remote Batch Invocation such as cursors and branching.

Sometimes a communication protocol defines batches directly, as is in the compound procedure in Network File System (NFS) version 4 Protocol [27], 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 [35, 11].

Cook and Barfield [11] showed how a set of hand-written wrappers can provide a mapping between object interfaces and batched calls expressed as a web service document. Remote Batch Invocation automates the process of creating the wrappers and generalizes the technique to support branching, cursors, and exception handling. As a result, Remote Batch Invocation scales as well as an optimized web service, while providing the raw performance benefits of RPC [13]. Web services choreography [24] defines how Web services interact with each other at the message level. Remote Batch Invocation can be seen as a choreography facility for distributed objects.

6.3 Mobile Code

Mobile object systems such as Emerald [5] reduce latency by moving active objects, rather than making multiple remote calls. JavaParty [25] migrates objects to adapt the distribution layout of an application to enhance locality. Ambassadors is a communication technique that uses object mobility [14] 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. An RBI server is fairly simple to implement. Clients only gain access to interfaces that are reachable from the service root.

Even in an environment that supports mobile code, there are advantages to Remote Batch Invocation. This can be understood by considering a translation from RBI to mobile code. A `batch` statement could be implemented using mobile code by writing two mobile classes, one that is sent from the client to the server to execute the remote operations, and another that is sent from the server back to the client to transport the results in bulk to the client. The first class would contain member variables to store all the local data sent to the server, and a method body to execute on the server. At the start of this method an instance of the second class is created and populated with data created by the remote method. At the end of the method the result object is sent back to the client. A custom pair of classes is needed for each `batch` statement in the program. While mobile code is more flexible and powerful than RBI, it can also be more work to use this power to implement common communication patterns.

6.4 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. There are several different client invocation patterns that cannot be batched in this model. For example, unrelated remote method calls will not be batched together.

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.

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.

In all of these implicit batching techniques, it is not clear how to support loops, branches, and exceptions as in Remote Batch Invocation. In addition, 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. This means the performance is very sensitive to the ordering of remote and local operations. On the other hand, Remote Batch Invocation automatically tries to reorder remote and local operations to maintain a single batch, while checking that the reordering makes sense.

6.5 Automatic Partitioning

Remote Batch Invocation can be seen as a language level abstraction for automatic application partitioning, a semi-automatic approach for deriving a distributed application from a centralized one.

One line of research has explored coarse grained program partitioning. The programmer, by means of a GUI or a configuration file, designates different parts of a centralized application, typically at a class or object granularity, to run on different network nodes. The resulting distribution specification then parameterizes a compiler-based tool

that automatically rewrites the centralized application for distributed execution. To introduce distribution, a partitioning tool may need to both change the structure of the application (e.g., to introduce a proxy indirection) and add middleware functionality (e.g., to replace local calls with remote ones). In the Java world, recent automatic partitioning tools include Addistant [31], Pangaea [28], and one of the co-author's J-Orchestra [33]. Addistant and J-Orchestra partition programs at a class granularity; Pangaea can partition at the individual object level. J-Orchestra addresses the challenges of partitioning programs safely in the presence of unmodifiable code that comes as part of their runtime systems.

Automatic program partitioning has also been applied at finer granularaties. Swift [10] partitions Java programs into a web application backend and Javascript at the Java statement level. Constraints on the locations of statements is inferred from information flow policies and the placement of statements is optimized to minimize round-trips with respect to those constraints. Similarly, RBI infers the location of statements and expressions from a forward data-flow analysis. Some of the co-authors have previously developed Query Extraction [37]; a system for extracting database queries from Java code traversing persistent object structures. Query Extraction performs a very similar analysis to RBI to extract the code operating over persistent data and converts that code's loops and conditions to *join* and *where* clauses in database queries.

6.6 Asynchronous Remote Invocation

Another approach to optimizing distributed communication is dispatching remote calls asynchronously. One example is ProActive [4]. An asynchronous remote call in ProActive returns a future; a placeholder for to be computed results. When a client tries to resolve the future's actual value, the client blocks until the result is available.

Although asynchronous remote invocations can optimize many patterns in client-server communication, they offer no performance improvements for chains of remote calls (i.e., `o.m1().m2()`). Compared to asynchronous invocation, the RBI programming model does not involve futures and can combine chains of remote calls into a batch, thus improving their performance.

Although the current version of RBI does not take advantage of concurrent processing, in the future the script recorder could also convey dependencies between batched operations to the server, which can be used to safely introduce concurrency into the batch execution on the server.

7 Conclusion

Most of the related work discussed in Section 6 improve distributed programming using libraries and compiler optimizations. On the other hand, *Remote Batch Invocation (RBI)* addresses distributed programming with a language extension. We argue that the benefits of RBI over existing library and compiler approaches may overcome the natural inertia to changing a programming language. The benefits of RBI include:

– RBI provides a strong performance model. One server round-trip is executed for each lexical batch block.

– RBI allows multiple remote operations to be combined in a *batch* which is executed in a single round-trip to a remote server. A batch supports both control and data flow dependencies between remote operations. As a consequence, the remote server may provide a flexible fine-grained interface.
– RBI allows the programmer to mix remote and local operations naturally. The compiler separates the remote operations and takes care of transferring multiple inputs to the remote server and interpreting the multiple outputs.

RBI was implemented as a Java extension using a source to source translator and the BEST runtime middleware library. In the future, we will look at incorporating transactions and advanced failure handling approaches into RBI.

The performance of RBI was evaluated by comparing plain RMI and hand-coded DTO designs. Predictably, RBI significantly outperforms RMI and is only marginally slower than hand-optimized DTO implementations. Since RBI provides greater flexibility and control to the programmer, the small overhead imposed by its runtime is compensated by the added usability and expressiveness. RBI is also attractive compared with implicit batching because it can combine a larger set of remote operations.

RBI combines the convenience and flexiblity of fine-grained interfaces with the performance advantages of coarser-grained interfaces. In addition, the RBI stateless execution model aligns well with the increasingly prevalent service-oriented architectures, a rapidly-emerging industry standard.

Availability:

The implementation and examples discussed in the paper can be downloaded from:
`http://research.cs.vt.edu/vtspaces/best`

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. E. Brevnov, Y. Dolgov, B. Kuznetsov, D. Yershov, V. Shakin, D.-Y. Chen, V. Menon, and S. Srinivas. Practical experiences with java software transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 287–288, New York, NY, USA, 2008. ACM.
8. N. Brown and C. Kindel. Distributed Component Object Model Protocol–DCOM/1.0, 1998. Redmond, WA, 1996.

9. K. Cheung Yeung and P. Kelly. Optimising Java RMI Programs by Communication Restructuring. In *ACM Middleware Conference*. Springer, 2003.
10. S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web application via automatic partitioning. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 31–44, New York, NY, USA, 2007. ACM.
11. 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.
12. C. Damm, P. Eugster, and R. Guerraoui. Linguistic support for distributed programming abstractions. In *Distributed Computing Systems. Proceedings. 24th International Conference on*, pages 244–251, 2004.
13. 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.
14. H. Detmold, M. Hollfelder, and M. Oudshoorn. Ambassadors: structured object mobility in worldwide distributed systems. In *Proc. of ICDCS'99*, pages 442–449, 1999.
15. 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.
16. T. Ekman and G. Hedin. The JastAdd Extensible Java Compiler. *SIGPLAN Not.*, 42(10):1–18, 2007.
17. T. Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall, Upper Saddle River, NJ, USA, 2005.
18. M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
19. R. Gabriel. Is worse really better? *Journal of Object-Oriented Programming (JOOP)*, 5(4):501–538, 1992.
20. D. Krafzig, K. Banke, and D. Slama. *Enterprise SOA : service-oriented architecture best practices*. Prentice Hall, 2005.
21. 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.
22. The Object Management Group (OMG). *The Common Object Request Broker: Architecture and Specification*, 1997.
23. D. A. Patterson. Latency lags bandwith. *Commun. ACM*, 47(10):71–75, 2004.
24. C. Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, 2003.
25. M. Philippsen and M. Zenger. JavaParty– transparent remote objects in Java. *Concurrency Practice and Experience*, 9(11):1225–1242, 1997.
26. U. Saif and D. Greaves. Communication primitives for ubiquitous systems or RPC consideredharmful. In *Distributed Computing Systems Workshop, 2001 International Conference on*, pages 240–245, 2001.
27. S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. Network File System (NFS) version 4 Protocol, 2003.
28. A. Spiegel. *Automatic Distribution of Object Oriented Programs*. PhD thesis, FU Berlin, FB Mathematik und Informatik, 2002.
29. Sun Microsystems. *Java Remote Method Invocation Specification*, 1997.
30. A. S. Tanenbaum and R. v. Renesse. A critique of the remote procedure call paradigm. In *EUTECO 88*, pages 775–783. North-Holland, 1988.
31. M. Tatsubori, T. Sasaki, S. Chiba, and K. Itano. A Bytecode Translator for Distributed Execution of "Legacy" Java Software. In *European Conference on Object-Oriented Programming (ECOOP)*, 2001.
32. B. Tay and A. Ananda. A survey of remote procedure calls. *Operating Systems Review*, 24(3):68–79, 1990.

33. E. Tilevich and Y. Smaragdakis. J-Orchestra: Enhancing Java programs with distribution capabilities. *ACM Transactions on Software Engineering and Methodology*. in press.
34. S. Vinoski. RPC Under Fire. *IEEE INTERNET COMPUTING*, pages 93–95, 2005.
35. W. Vogels. Web services are not distributed objects. *Internet Computing, IEEE*, 7(6):59–66, 2003.
36. J. Waldo, A. Wollrath, G. Wyant, and S. Kendall. A Note on Distributed Computing. Technical report, Sun Microsystems, Inc. Mountain View, CA, USA, 1994.
37. B. Wiedermann, A. Ibrahim, and W. R. Cook. Interprocedural query extraction for transparent persistence. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 19–36, New York, NY, USA, 2008. ACM.