Web Services versus Distributed Objects: A Case Study of Performance and Interface Design

William R. Cook, Janel Barfield
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712
Email: {wcook,janelbarfield}@cs.utexas.edu

***Abstract*— Web services are promoted as a new model for distributed systems, yet many skeptics see them as simply a poor implementation of traditional remote procedure calls (RPC) or distributed objects. Previous comparisons support the skeptics: web services are shown to be significantly slower than RPC, and they lack features like automatic proxies. However, these studies are biased because they are based on an RPC communication style. Web services support a document-oriented style of communication that performs well even in the face of the high latency found in internet or business transactions. We investigate these issues by comparing the design and implementation of a small file server application implemented using RMI and web services. For this application, using the most straightforward implementation in both technologies, web services outperform RMI when accessing multiple/deeply nested files, especially over high-latency channels. However, the default web services interfaces are awkward to use, so we develop a technique for wrapping the web service to make it as easy to use as the distributed object implementation. The same wrappers are then used to implement the document-oriented communication style in RMI, which improves performance but significantly complicates the design. This case study provides a more detailed comparison of the relationship between web services and distributed objects.**

I. Introduction

There is significant debate on the relationship between web services and distributed objects [1], [2], [3]. Concrete case studies can help to clarify and quantify these differences. In this paper, we take a step toward this goal by examining the performance and design of an example application using distributed objects and web services.

Previous approaches to comparing performance of distributed objects and web service have focused on the ability of web services to perform traditional RPC-style interactions: invoking a single method whose arguments are simple primitive values or arrays [4], [5], [6], [7]. Given the overhead of encoding and decoding XML, it is not surprising that web services are an order of magnitude slower than distributed object implementations in CORBA [8], DCOM [9], or RMI [10]. However, these studies are biased because they only measure RPC-style communication. They do not consider the possibilities of document-oriented designs that demonstrate the strengths of web services.

In this paper we do the opposite: consider a scenario that is better suited for web service implementation. The example application is a remote file server based on the HTTP protocol [11]. The question considered is: what if HTTP had been defined using distributed object middleware (RMI/DCOM/CORBA) or as a web service? The resulting designs are implemented and evaluated for performance and usability.

Object-oriented analysis and design [12] of this application produces a clean object-oriented interface to the repository of files. The interface is easily converted to remote invocation over RMI using automatic proxies. There is a tight coupling between clients and servers, since the client must also have access to the interfaces used to program the server. More significant, the design results in a large number of round-trips to the server, causing poor performance as client transactions become more complex – for example, in downloading multiple files or increasing path length.

The web service implementation is defined by creating a service object and java classes describing the request and result XML documents. These classes a translated to a WSDL (Web Service Description Language) file and corresponding server-side wrappers using Apache Axis in the Eclipse Web Tools Project [13], [14], [15]. The client was created similarly by importing the WSDL. The web service approach naturally reduces the number of round-trips.

We evaluate the performance of each implementation in loading batches of files over networks with a range of latencies. The pure-object RMI implementation is faster for small batches of documents and low-latency networks, but performance degrades rapidly with larger batches and higher latency. The web services has a high initial cost but shows little or no change with larger batches. Higher latency creates a greater initial cost, but performance is still independent of batch size. As latency increases, the performance benefits of the document-oriented approach increase significantly. This is relevant when in some real world scenarios, latency may even be minutes, hours, or days, as for disconnected or asynchronous workflow processes.

Unfortunately, the web service client code is awkward to use: the programmer must manipulate request and response structures, rather than directly perform operations on server objects as in the RMI implementation.

In short, the most natural designs for distributed objects are easy to use but scale poorly, while web services have good scaling properties but are awkward to use. To address this problem, we create better client wrappers for the web

```
public interface Container {
 Container sub(String name) throws FileNotFound;
 File get();
 File invoke(HashMap params);
}

public interface File {
  String getRequest();
  long getModified();
  int getLength();
  String getText();
  String getType();
  String getEncoding();
}
```

Fig. 1. Object-oriented file-server interface.

```
Container file;
file = root.sub("base").sub("index.htm");
String s = file.get().getText();
```

Fig. 2. Example client using the object-oriented interface

service, which implement the same interfaces used in the distributed object model, with one additional call and a slight change in semantics. We also show how these wrappers can be reused to convert the RMI solution to use a document-oriented communication style based on mobile code and value objects. The resulting RMI implementation is very fast – but it is also quite complex, does not make significant use of automatic proxies (it is stateless except for a connection to a singleton server object), and is still platform-specific. The resulting implementations use the communication style that is natural to web services, but use wrappers to provide a user-friendly object-oriented facade.

II. File Server Interfaces and Implementations

As a traditional network protocol, HTTP [11] is defined by commands and responses that are formatted as strings of characters. The syntax of the commands and legal responses are defined by a grammar in RFC 2616 [11]. While this is a reasonable approach for specifying network protocols based on TCP, the specification does not provide a high-level programming model.

For a programming model, high-level concepts should be represented as explicit data abstractions with appropriate operations and consistency checking. A programming interface that captures elements of the HTTP protocol would allow clients and servers to traverse a hierarchical namespace to access files and file properties, and invoke actions that create dynamic files. Thus, the object-oriented file server interface is significantly different from the protocol-oriented HTTP interface.

A. Object-Oriented Interface for File Server

Our goal is to design a clean object-oriented interface for a file server that is easy to use for programmers. Since the server presents a hierarchical view of a collection of files, the primary interface is that of a `Container`. This interface is defined in Fig. 1. The user can access a subcontainer with the `sub` method. If the named item does not exist, and error is returned. The `get` method retrieves a file and its attributes. Finally, the `invoke` method performs an action with a specified set of parameters and returns a dynamically created file as a result. When `get` or `invoke` are executed on the server, a `File` object is returned. A file object contains information about the file and a method to return the text of the file. As in HTTP, the file server API does not distinguish between directories and files. A call to `get` on a directory could return an error, or produce a file containing a listing of the contents of the directory.

This interface is similar to many other interfaces for hierarchical structures. For example, the `FileSystemObject` in Microsoft Windows has a similar structure [16]. One difference is that it uses explicit collection objects for folders (subcontainers) and files (documents). The directory APIs for Java, `javax.naming`, are also similar, although much more complex.

A client can easily traverse the hierarchy and access files. An example is given in Fig. 2. This code gets the text of a file named "index.htm" within a "base" folder. It is equivalent to the HTTP operation `GET /base/index.htm`.

B. Distributed Object Implementation: an RMI File Server

The RMI file-server classes implement the `Container` and `File` interfaces shown in Fig. 1. To support remote access, they also implement the `Remote` interface of RMI. When `Container` or `File` objects are returned by a server operations, they are converted to proxies in the client. Operations on proxies are sent back to the server for processing, which may create more proxies. One proxy is created for each container in the file path. When more files are retrieved, more calls to the server take place.

The client code to access a remote server is the same as the code to access the server locally. This also applies to error conditions: if the name passed to the `sub` call is not a valid name in a given container on the server, the error will be reported immediately.

Registering the server is also simple:

```
ContainerImpl server = new ContainerImpl();
Naming.rebind("docServer", server);
```

C. Document-Oriented Interface for File Server

In a document-oriented interface, the operations to be performed on the server are described in a *request* document, and the results from the server are contained in a *response* document. The document-oriented interfaces are defined in Fig. 3. The server exports a single operation to `perform` a request and return a set of responses.

The request object reifies, or makes concrete, a set of calls to a container. There are significant differences between the `Container` interface in Fig. 1 and the `Request` interface: sub-containers are returned by the `sub` method of

```
// server processes requests and returns files
public interface ResourceServer {
  public FileResult[] perform(Request doc);
}

// hierarchical requests can specify multiple
// files in different directories
public interface Request {
  String getName();
  void setName(String name);

  Request[] getItemList();
  void setItemList(Request[] items);

  boolean getResultID();
  void setResultID(int ID);

  Param[] getParams();
  void setParams(Param[] params);
}

// the result ID matches results to requests
public interface FileResult extends File {
  int getStatusCode();
  int getResultID();
}
```

Fig. 3. Request interface for web services.

```
<wsdl:portType name="ResourceServer">
 <wsdl:operation name="invoke"
             parameterOrder="doc">
  <wsdl:input message="impl:invokeRequest"
           name="invokeRequest"/>
  <wsdl:output message="impl:invokeFileResult"
            name="invokeFileResult"/>
 </wsdl:operation>
</wsdl:portType>

<wsdl:message name="invokeRequest">
 <wsdl:part name="doc" type="impl:Request"/>
</wsdl:message>

<wsdl:message name="invokeFileResult">
 <wsdl:part name="invokeReturn"
         type="impl:ArrayOfFileResult"/>
</wsdl:message>
```

Fig. 4. Web service operation defined in WSDL.

`Container`, but sub-requests are collected as an explicit array of `items` in the `Request` interface. The `name` parameter of `sub` is stored as a `name` attribute in each `Request` item. A similar treatment is used for the `params` of `invoke`. Finally, the `get` method, which retrieves a `File` from a container, is translated into a `ResultID` attribute that assigns an integer sequence number to each requested file.

Only one remote method is used in the web services interface to model a document-centric approach in which a complete request is sent to the server as a single document, and a single document is returned by the server that contains all requested results. This `perform` method takes a fully populated `Request` as input, and returns an array of `FileResult` objects containing any files returned from the server. The `FileResult` interface extends `FileResult` to include a `ResultID` that associates each result with its corresponding `Request` object.

```
<complexType name="Request">
 <sequence>
  <element name="itemList"
           type="impl:ArrayOfRequest"/>
  <element name="name" type="xsd:string"/>
  <element name="params"
           type="impl:ArrayOfParam"/>
  <element name="resultID" type="xsd:int"/>
 </sequence>
</complexType>

<complexType name="ArrayOfRequest">
 <complexContent>
  <restriction base="soapenc:Array">
   <attribute ref="soapenc:arrayType"
       wsdl:arrayType="impl:Request[]"/>
  </restriction>
 </complexContent>
</complexType>

<complexType name="FileResult">
 <sequence>
  <element name="text"     type="xsd:string"/>
  <element name="Encoding" type="xsd:string"/>
  <element name="length"   type="xsd:int"/>
  <element name="type"     type="xsd:string"/>
  <element name="modified" type="xsd:long"/>
  <element name="location" type="xsd:string"/>
  <element name="resultID"   type="xsd:int"/>
  <element name="statusCode" type="xsd:int"/>
 </sequence>
</complexType>

<complexType name="ArrayOfFileResult">
 <complexContent>
  <restriction base="soapenc:Array">
   <attribute ref="soapenc:arrayType"
       wsdl:arrayType="impl:FileResult[]"/>
  </restriction>
 </complexContent>
</complexType>
```

Fig. 5. Web service request and response structures.

Error handling is significantly different when using a document-oriented interface. Errors are also reified as properties of result objects. For example, the `FileNotFound` exception thrown by `sub` in the object-oriented interface becomes a *status code* in the `FileResult` interface. The `perform` method itself does not raise the `FileNotFound` exception, since it handles multiple requests, some of which may fail while other succeed.

An example of client access the web service is given in Fig. 6. To invoke the web service, a client creates a `Request` structure explicitly. The requests form a tree structure that identifies which objects to retrieve from the server. Each file to

```
// create a request for base/index.htm
Request base = new Request("base");
Request index = new Request("index.htm");
base.setItemList(new Request[] { base });
index.setResultID(0);

// perform the request
File[] docs = server.perform(base);

// access the text of the result
String s = docs[0].getText();
```

Fig. 6. Example client using the document-oriented interface.

be retrieved is assigned a unique response identifier by calling `setResultID`.

The client interface is awkward to use. Rather than perform operations on file system objects, as in the object-oriented interface, the client must create objects that represent the operations to be performed. The web service interface is more *indirect* conceptually. The client must check the status code explicitly to determine if the request was successful.

D. Web Service Implementation of the File Server

Web services can use a document-oriented communication style to implement the file server. We used Apache Axis includes tools to automatically translate Java interfaces into WSDL files. The WSDL file for Fig. 3 is given in Fig. 4 and Fig. 5. The `nillable="true"` attribute was omitted for readability.

The design of the `Request` interface was constrained to some degree by existing web service tools (Axis). In the web services implementation, a `Request` object can represent either a *container* object or a *file* object. We attempted to distinguish between these representations by using subclassing. It was not possible to subclass the `Request` object to provide this distinction because of limitations of the technology. The Axis tools did not produce the expected WSDL when this approach was taken.

III. Performance

The test driver loads files from a file server using either the web service interface or the RMI interfaces. The independent variables are the communication configuration and the number of files retrieved in a single transaction. Three configurations were used for communication from client to server: both processes on the same machine (localhost), over closely connected local network, and over a wireless network on different subnetworks. The files being retrieved are generated synthetically to avoid disk access. They are also very small, less than 100 bytes. The tests are performed iteratively 50 to 100 times and the results averaged.

The server machine is a Dell Precision 360 running Windows XP and Apache Tomcat version 5.0. The client was a Dell Lattitue D600 laptop running Windows XP. The Sun Java VM 1.5.0 runtime was used in both implementations and for the `rmiregistry` executable the RMI server.

All the files retrieved are two folders down from the root, with paths of the form `/base/section/testi.htm`. The RMI interface requires at least 5 round-trips to access a single file. The costs do not include setting up the connection for RMI or the web service. Accessing files with longer paths would cause more round-trips.

In addition to the Web Services and RMI implementations described in Section II, the result below include data for an RMI Optimized implementation, which is defined below in Section IV-B.

A. Performance Results

The results for the localhost configuration are shown in Fig. 7. The Web Service implementation is significantly slower than the RMI implementation in retrieving one document. It is well-known that here is greater overhead associated with making a Web Service call. This result is analogous to previous studies [4], [5], [6], [7], although the performance difference is less significant because our RMI application requires multiple round trips to access even a single document. Both implementations scale linearly with the number of files being retrieved. However, the slope of the RMI implementation is much greater. This is because *each* file takes approximately 8ms to load, and so five files take 40ms. The Web Service, on the other hand, only performs one round-trip to the server to load any number of documents. Thus the incremental cost of retrieving another document is low. The results show that between 2 and 3 files the web services performance exceeds that of the RMI implementation. When 5 files are loaded, the web service takes half the time of RMI.

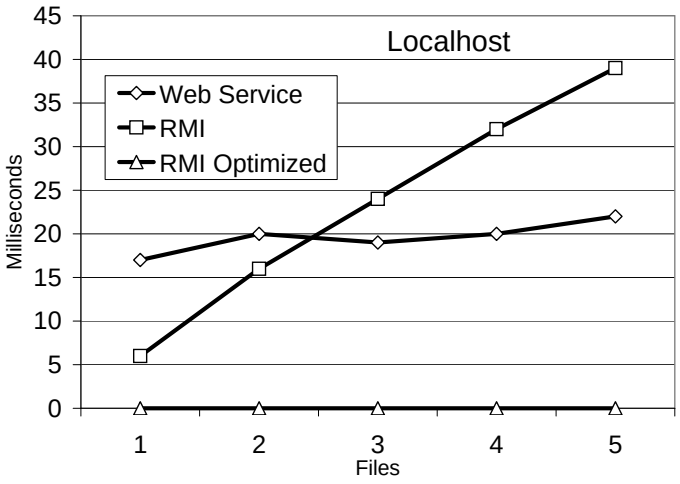

Fig. 7. Performance of Web Service, RMI, and RMI Optimized with client on same machine as server.

The results for a local area network configuration are given in Fig. 8. In this case the latency is approximately 2ms. With this higher level of latency, the web service and RMI implementations are approximately equivalent for retrieving one file. At 5 files, the web services takes only 25% of the time of the RMI implementation.

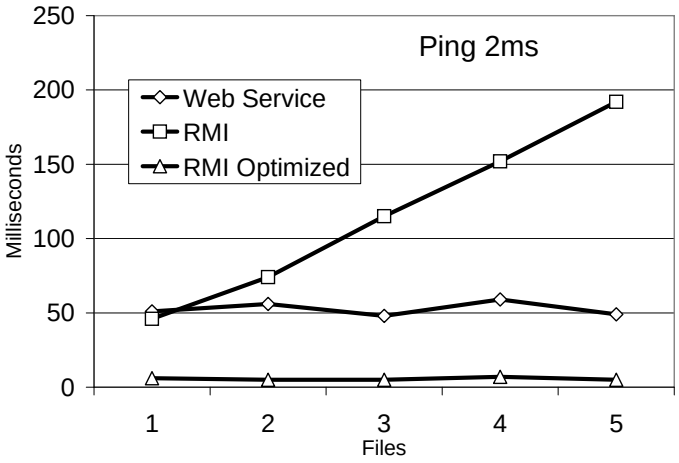

Fig. 8. Performance of Web Service, RMI, and RMI Optimized with client on local area network (average ping time 2ms).

Finally, the results for the higher-latency wireless configuration are given in Fig. 9. In this case the latency is 29ms. The effect of increased latency on the RMI implementation are clearly visible. The web service is faster retrieving any number of documents. For 5 files, the web service takes only 12.5% of the time required by RMI. Web service performance is essentially constant as the number of files increases.

As more and larger files are retrieved both RMI and the web service will also experience bandwidth costs in transferring files from the server to the client. These costs would be fairly equivalent between the two, assuming that appropriate compression schemes are used. However, the basic relationship between latency and bandwidth is unlikely to change: improvements in latency are more difficult than improvements in bandwidth [17]. The key observation is that latency costs dominate when performing multiple operations on remote objects.

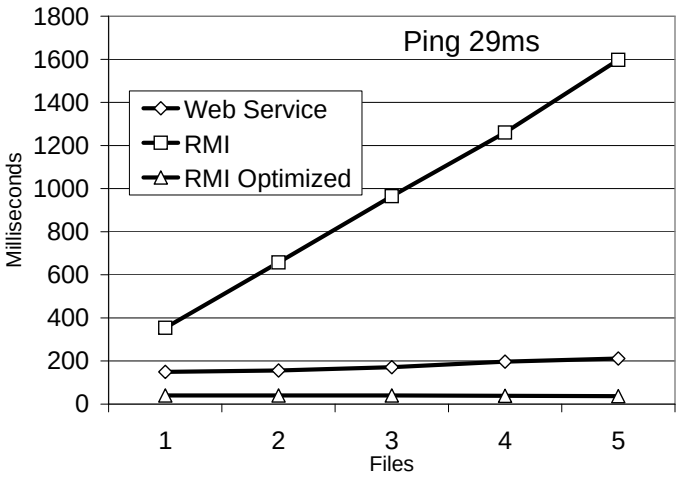

Fig. 9. Performance of Web Service, RMI, and RMI Optimized with client on wireless network (average ping time 29ms).

IV. Improving Usability and Performance

As seen in previous sections, the RMI implementation is easy to use but has significant performance problems in accessing multiple files with long paths. On the other hand, the web service scales well (despite its initial higher cost), yet is difficult to use. In this section we introduce a client-side wrapper to improve the usability of the web service. It turns out to be exactly what is needed to improve the performance of the RMI implementation.

A. An Object-Oriented Wrapper for the Web Service Client

Using the `Request` and `ResourceServer` interfaces directly is awkward. It exposes too many details about the structure of the communication model. To solve this problem, we defined a wrapper classes to assist the client in constructing the request to the server, as shown in Fig. 10. The wrapper automate the creation of request objects and the matching of responses to requests after calling the server's `perform` method. The key point is that `ClientRoot` implements the original `Container` interface in Fig. 1.

`RequestWrapper` provides `sub` and `get` methods that create requests and futures on the client. A *future* is a placeholder value for a response that will be returned from the server in the future. The `sub` method instantiates a new `RequestWrapper` object with the `String` value passed as an argument, adds the new instance to its `items` array, and returns the new `RequestWrapper` object. The `get` method calls the `ClientRoot` to create a new future. It creates a new result ID and initializes the `RequestWrapper` and `ResponseFuture` with this ID.

The key to this interface is that the `File` objects returned by `get` and `invoke` are not populated – they are *futures*, or placeholders that are undefined until after the call to the `perform` method. When the `Request` object is fully constructed by the client and is passed to the `ResourceServer`'s `perform` method, the wrapper fills in the `File` objects using the data returned from the server. The futures also perform error handling, by throwing the `FileNotFound` exception if the status code of the result file indicates an error. The same exception is thrown as in the object-oriented interface, but it is thrown at a different time: instead of being thrown by `sub`, it is thrown when the result is accessed.

The client wrappers are as easy to use as the RMI interfaces. For example, we used the code segment shown in Fig. 11 to retrieve 20 files from a single directory on the server. Note that different clients can implement different wrappers – clients are responsible for creating their own web service interfaces, rather then depending on the server to publish appropriate interfaces.

B. Document-Oriented RMI Implementation

Implementing a document-oriented RMI implementation involves careful design to allow the client to collect multiple logical operations into a single request, which is then sent to the server for processing.

```
class RequestWrapper
  extends Request
  implements Container {

  ClientRoot root;

  Request sub(String name) {
    RequestWrapper[] items =
      (RequestWrapper[]) getItemList();
    // omitted: find named object in items,
    // or creates if it does not exist.
  }

  public Response get() {
    return root.newResponse(this);
  }
}

class ClientRoot
  implements ResourceServer {

  RequestWrapper request;
  ResourceServerBase server;
  int nextID;
  Vector<ResponseFuture> futures =
    new Vector<ResponseFuture>();

  // forward to request
  public RequestWrapper sub(String name) {
    return request.sub(name);
  }

  // perform complete request and decode results
  void perform()
    throws RemoteException, ServiceException
  {
    FileResult[] files = server.perform(request);

    // fill in the futures
    for (int i = 0; i < files.length; i++) {
      FileResult file = files[i];
      ResponseFuture future =
        futures.get(file.getResultID());
      future.setResult(file);
    }
  }

  // create a response future
  ResponseFuture newResponse(Request location) {
    ResponseFuture resp = ResponseFuture();
    futures.add(resp);
    location.setResultID(++nextID);
    return resp;
  }
}
```

Fig. 10. Wrappers for an object-oriented to a document-oriented server

```
int numberOfFiles = 20;
ClientRoot request = new ClientRoot();
File[] docs = new File[20];

for (int i = 0; i < numberOfFiles; i++) {
  String filename = "test" + i + ".htm";
  Container base = request.sub("base");
  docs[i] = base.sub(filename).get();
}
service.perform(request);
print(docs[3].getText()); ...
```

Fig. 11. Using the improved web service interface.

The key technique is the appropriate use of remote and serializable objects to define which objects should be transmitted as proxies, and which should be copied. In summary, the strategy is to create a *smart server stub*, implemented by the same `ClientRoot` used for wrapping the web service, that is copied to the client to manage client-side interaction. This stub contains a proxy that refers back to the server, which implements the `ResourceServer` interface defined in Fig. 3 for processing requests.

The client stub collects client-side operations into a batch, which is then transmitted to the server using the server proxy. The batch is represented by a `Request` object, which is made serializable so that it will be copied to the server. The server returns a set of serializable response objects, which are copied to the client. Since the client logic is the same as for the web service wrapper, we reused the wrapper code.

Clients must use another remote object, the `ContainerRootFactory` to obtain the smart `ClientRoot` stub. Creating the factory is more complicated than simply registering a remote object. The server contains a remote object (the file server) contained in a serializable object (the smart client wrapper) which is referenced by a remote object (the factory):

```
// ResourceServer is Remote
ResourceServer server = new ResourceServer();
// ClientRoot is Serializable, server is proxy
ClientRoot root = new ClientRoot(server);
// ClientFactory is Remote
ClientFactory factory = new ClientFactory(root);
// Bind the factory
Naming.rebind("rmiopt", factory);
```

Since `RequestWrapper` is serializable, care must be taken to avoid serializing the `ClientRoot` referenced by the `root` vabiable: it is set to null before the request is sent. In the web service implementation, only the `Request` part of the `RequestWrapper` is serialized; in RMI it is harder to select what parts of an object need to be serialized.

The steps in the execution of the optimized RMI server are illustrated in Fig. 12. In step 1, the client connects to the factory. In Step 2 the client calls `create` to obtain a local copy of the smart container root. In step 3, the client invokes operations on the container root that create a local request object and the response futures. When the client calls

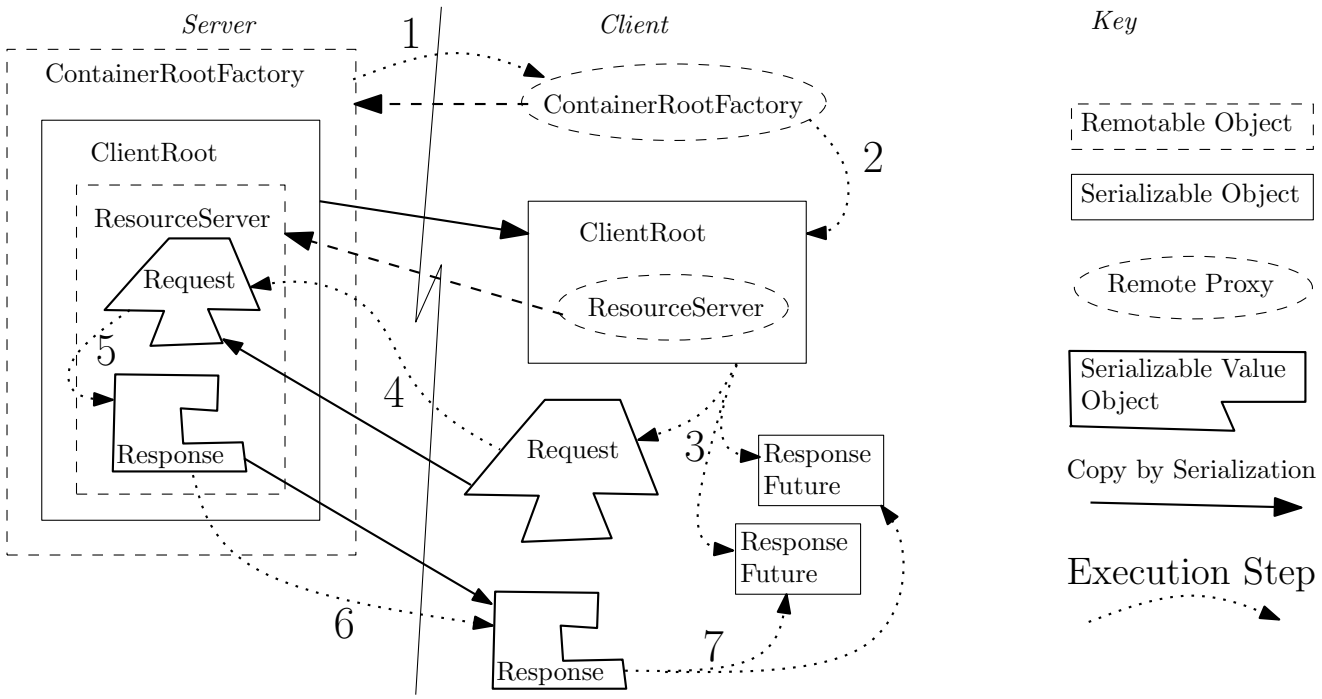

Fig. 12. Steps in the execution of the optimized, wrapped server

`perform`, in step 4, the container root invokes the server proxy with the request, causing the request to be copied to the server. In step 5 the server processes the request and creates the response structure. In step 6, the call to `perform` returns and the response is copied to the client. Finally, in step 7 the response futures are filled in with the actual responses. The response futures reuse the same code defined for the web service, but instead of serializing with XML for a web service, native Java serialization is used. As shown in Figs.7-9, the resulting optimized RMI implementation is very fast and has good scalability. Its main drawbacks are that it is difficult to develop and is still platform-specific.

V. Related Work

There are a number of existing studies that examine the performance of web services and SOAP compared to other distributed object technologies, including Java RMI. In each of these studies, a comparison is made based on the performance of a web service implementation that calls a simple method and an RMI, CORBA, or other implementation that calls the same method. The authors in [7] compare the round trip method invocation times and instantiation times for all simple data types for web services (with RPC), Java RMI, and RMI with tunneling. The study described in [5] compare 20 different middleware implementations, including Java RMI and web services with RPC encoding, using measurement of an empty 'ping' method invocation on the server. Authors of the other works cited [4], [6] performed similar studies with measurements of single method invocations, in which the web service implementation mimicked the RPC-style paradigm of calling the server with multiple simple requests, always using an RPC encoding in the web services implementation.

In all cases mentioned, the authors showed that web services implementations performed slower than other implementations, and always slower than a Java RMI implementation. Some of them pointed out that web services don't provide value in performance, but do provide a convenient way to provide user interface, automatic firewall protection (because they generally use HTTP for transport), mobility of applications, transparent proxies, and thin clients [2]. The authors in [6] acknowledge that a naïve use of web services to exactly model CORBA resulted in an overwhelming degradation in performance, a factor of 400 in this study, but with some tuning this degradation was only a factor of 7.

It is not surprising that web services performed poorly in all of these studies when one considers that they are not intended to be used RPC-style like other distributed object technologies. When considering performance alone, web services provide value when the overhead of parsing XML and SOAP is outweighed by the business logic or computation performed on the server. Web services provide a literal encoding that can be used in a document-centric paradigm rather than an RPC-centric one. In this model, the client sends over multiple requests in a single document to the server, and retrieves multiple pieces of information back from the server in a single XML document. In this study, we use the document-centric nature of web services to show that web services implementations can outperform other traditional implementations when this document-centric approach is used and compared with an RPC-centric approach.

Davis [4] examines the call latency of SOAP implementa-

tions in comparison to Java RMI and CORBA. Call latency is the total elapsed time for a call. When run on the same machine, they show that web services implementations are on average 20 times slower than Java RMI to call an empty procedure without network delay. When run on different machines the RPC systems reflect the small network delay of approximately 0.4 ms in their test setup. Web services experience a delay of approximately 170 ms, which they show to be due to an interaction between acknowledgement packets and the Nagle algorithm used to limit the number of small packets on the internet.

Our design can be viewed as an explicit and generalized form of batched futures [18]. The original definition that will perform a remote operation for non-proxy data is needed, for example when calling `File.getStatusCode`. In our web service wrapper design, the remote operation is explicit; such a call would raise an exception. The explicit approach is more flexible and can be more efficient, because it allows multiple requests for non-proxy data to be performed at the same time.

VI. Conclusion

In this project we challenged the existing studies that compare the performance of web services and traditional distributed object technologies, which have all shown that web services perform poorly compared to traditional technologies [4], [5], [6], [7]. We noted that these previously performed studies constructed web services that exactly model the RPC-style of traditional distributed object implementations like Java RMI or CORBA. In these studies, the service created performs a simple operation, for example, calling a simple method without arguments that returns an integer or string value [5]. We argued that by doing the opposite in this study, using a scenario well suited for a web services implementation and using design strategies that make web services effective, we could show that web services can outperform traditional technologies.

We implemented a web services and Java RMI programming interface to the HTTP get method, providing a way for clients to form requests and retrieve results from a server. In the web services implementation, we pass a request for multiple files in a single document, and receive all resulting files in a single document. In the Java RMI RPC-style implementation, each file retrieved requires at least one call to the server, with the presence of subdirectories resulting in multiple trips.

A client-server program that allows the client to load a variable number of files from the server was created with both web service and RMI interfaces, and test scenarios that retrieved a varying number of files were executed with three different network latency conditions. All tests show the performance benefits and near perfect scaling of the document-centric web service implementation, particularly in the presence of network latency, where it always outperformed the RMI interface.

The performance of the systems can be approximated by a simple model based on latency time L, transmission/encoding time T (defined as message size divided by bandwidth), distributed object overhead D, and web service overhead W. The RMI implementation executes in time $n(L+T+D)$ while the web service executes in $L + nT + W$, where n is the number of operations performed in a batch. The batch size at which web services will perform as well as distributed objects is $\hat{n} = (L + W)/(L + D)$. For low latencies, web services are competitive only when for batches of (approximately) size W/D. Given that W is currently much larger than D, web services are only competitive if they can perform batches of 10 or more primitive RMI operations. Our tests indicate that latencies can be equal or greater than W, in which case web services are competitive with a batch size closer to 2.

We identified that the web service interface is awkward and difficult to use. The RMI implementation is easy to use but has significant performance problems. We suggested a way to improve usability and performance by creating an optimized document-centric RMI implementation. In this implementation, the client constructs the request, a factory sends the request to the server, and the results back to the client, and the client populates the result structure from the server response. This implementation has the best of both worlds, outperforming the other two implementations in all tests, and providing an interface that is easy to use.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant No. 0448128.

References

[1] W. Vogels, "Web services are not distributed objects." *IEEE Internet Computing*, vol. 7, no. 6, pp. 59–66, 2003.

[2] A. Gokhale, B. Kumar, and A. Sahuguet, "Reinventing the wheel? CORBA vs. web services." in *Proceedings of the International World Wide Web Conference*, 2002.

[3] K. P. Birman, "Like it or not, web services are distributed objects," *Commun. ACM*, vol. 47, no. 12, pp. 60–62, 2004.

[4] D. Davis and M. Parashar, "Latency performance of SOAP implementations," *IEEE Cluster Computing and the Grid*, 2002.

[5] 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*, vol. 4, no. 1, p. 724, May 2005, iSBN: 2912590310.

[6] R. Elfwing, U. Paulsson, and L. Lundberg, "Performance of SOAP in web service environment compared to CORBA." in *APSEC*. IEEE Computer Society, 2002, pp. 84–.

[7] M. B. Juric, B. Kezmah, M. Hericko, I. Rozman, and I. Vezocnik, "Java rmi, rmi tunneling and web services comparison and performance analysis," *SIGPLAN Not.*, vol. 39, no. 5, pp. 58–65, 2004.

[8] OMG, *The Common Object Request Broker: Architecture and Specification*. Object Management Group, Framingham, MA, 1998.

[9] "Distributed object component model (DCOM)." [Online]. Available: http://www.microsoft.com/com/tech/DCOM.asp

[10] "Remote method invocation (RMI) in the jdk 1.1 specification." [Online]. Available: http://javasoft.com/products/jdk/1.1/docs/guide/rmi

[11] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext transfer protocol – HTTP/1.1," 1999. [Online]. Available: citeseer.ist.psu.edu/fielding97hypertext.html

[12] G. Booch, *Object oriented analysis and design with applications*. Addison-Wesley, 1994.

[13] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, "Web Services Description Language (WSDL) 1.1," 2001, http://www.w3.org/TR/wsdl.

[14] "Apache Axis," 2002, http://xml.apache.org/axis/.

[15] S. Holzner, *Eclipse*. O'Reilly, 2004.

[16] *Microsoft Developer Network Online Documentation*, http://msdn.microsoft.com.

[17] D. A. Patterson, “Latency lags bandwith,” *Commun. ACM*, vol. 47, no. 10, pp. 71–75, 2004.

[18] P. Bogle and B. Liskov, “Reducing cross domain call overhead using batched futures,” in *OOPSLA ’94: Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications*. New York, NY, USA: ACM Press, 1994, pp. 341–354.