Java RMI, RMI Tunneling and Web Services Comparison and Performance Analysis

Matjaz B. Juric[*], Bostjan Kezmah[*], Marjan Hericko[*], Ivan Rozman[*], Ivan Vezocnik[**]
[*]University of Maribor, FERI, Institute of Informatics, Smetanova 17, SI-2000 Maribor
[**]Razvojni center IRC Celje, Ulica XIV. divizije 14, SI-3000 Celje
E-mail: matjaz.juric@uni-mb.si

Abstract

This article compares different approaches for developing Java distributed applications which have to communicate through firewalls and proxies, including RMI over open ports, HTTP-to-port, HTTP-to-CGI, HTTP-to-servlet tunneling and web services. A functional comparison of approaches has been done, as well as a detailed performance analysis with overhead analysis and identification of optimizations. Therefore the paper contributes to the overall understanding of different approaches for developing Java distributed applications in circumstances, where the communication through firewalls and/or proxies is inevitable. The paper also contributes to the understanding of performance related issues.

Keywords: RMI, tunneling, web services, SOAP, performance

1. Introduction

RMI (Remote Method Invocation) has been a part of Java SDK since version 1.1 and provides an API for developing distributed applications in Java. RMI uses the JRMP (Java Remote Method Protocol), which uses TCP/IP for communication. RMI client requires an open port to communicate with RMI server. When there is a firewall between the client and the server, RMI communication is blocked by default. This can be a major problem, particularly with applications, which require connectivity and interoperability over different LANs, each secured by its own firewall. Such applications are more and more common with the adoption of e-business applications and integration between enterprises. On the other hand the security restrictions are getting stricter because of the increased threats. In most cases this prevents us from using the most straight-forward solution – opening the ports which RMI needs on the firewalls.

In this paper we will identify and evaluate alternative solutions available for Java distributed applications communicating over firewalls and proxies. These alternative solutions can be classified in two groups:

- RMI tunneling techniques, which include HTTP-to-port, HTTP-to-CGI and HTTP-to-servlet tunneling, and
- Using Web Services instead of Java RMI.

The major advantage of RMI tunneling techniques is that there we can use the same RMI API as before. Existing RMI applications can therefore be used without code modifications. This is however related with relatively complex configuration of tunneling. The transition to web services on the other hand requires the use of a new API – the JAX-RPC (Java API for XML based Remote Procedure Call), which is a part of the JWSDP (Java Web Services Development Pack). Web services enable the communication over the XML-based SOAP (Simple Object Access Protocol) protocol and thus enable interoperability with other web services, which do not have to be developed in Java. Web services on the other hand introduce a few restrictions, compared to RMI, such as no support for object references, absence of distributed garbage collection, etc. Because they use XML-based protocol (SOAP) for communication we would also expect that the performance of web services will be lower than the performance of RMI. Because performance is often important for distributed applications in this paper we will analyze the performance of the identified alternative solutions.

The review of related work has shown that the comparison and performance analysis of RMI tunneling techniques and/or web services has not been done yet. There are a few articles on RMI performance [1], [2], some of them from our earlier research [3], [4], [5], [6], and [7], where we have compared the performance of RMI and CORBA. Our earlier articles have been based on a set of benchmarks for distributed object models, proposed in [8] and [9]. We use an updated version of these benchmarks in this paper too. There are also several papers in which the performance of different CORBA implementations with the C++ programming language is compared [10], [11], and [12]. The authors in [13] and [14] compare the CORBA performance for the purposes of real-time systems. At the time of writing this article there was no web services performance analysis related to Java, nor was there a comparison of RMI and web services performance. There was also no systematic analysis of RMI tunneling techniques and their comparison to web services. Therefore we believe that this article contributes to the overall understanding of Java distributed computing.

The article is organized as follows: first, we analyze the use of RMI over firewalls and proxies and identify different tunneling scenarios, including HTTP-to-port, HTTP-to-CGI and HTTP-to-servlet. Then we evaluate the transition to web services and compare it to RMI. Next we make a detailed performance comparison of RMI, RMI tunneling approaches and web services. Finally we make an overhead analysis and identify possible optimizations.

2. RMI, Firewalls and Fall-back Scenarios

RMI enables the development of distributed Java applications in which the methods of remote Java objects can be invoked from other Java virtual machines, possibly on remote hosts. The RMI system consists of three layers: the stub/skeleton layer, the remote reference layer and the transport layer [15]. The stub/skeleton layer provides support for client-side stubs and server-side skeletons, the remote reference layer for reference and invocation behavior and the transport layer for connection setup and management as well as remote object tracking. The RMI transport layer usually opens direct sockets to remote object hosts. Firewalls do not allow this and block direct socket connections. Usually firewalls block all traffic to a set of source and/or destination ports. If a client chooses to use a port for connection to a specific service, blocking that port means blocking access to that service for the client. Firewalls also introduce problems related to call-back scenarios. In a call-back a RMI server reference is passed as a parameter of a remote method invocation. This allows the server to call-back the original caller (client). Firewalls block such communication, therefore callback is not possible. RMI however offers a multiplexing protocol, which provides a model where two endpoints can each open multiple full duplex connections in an environment where only one of the endpoints is able to open such a connection [15].

Firewalls generally consist of two parts: a transport firewall and one or more application firewalls. Transport firewall is configured to permit communications originating from trusted application firewalls. Application firewalls are usually referred to as proxies. Proxies are applications running on a firewall, which provide communication services to clients and outside services. They act as a mediator between the client and the server. When a client connects to a server through a proxy it appears like the proxy is making the connection. This sophisticated application allows a number of filtering possibilities, making them flexible and also complex. They are also not transparent to the user and therefore require configuration on the client side.

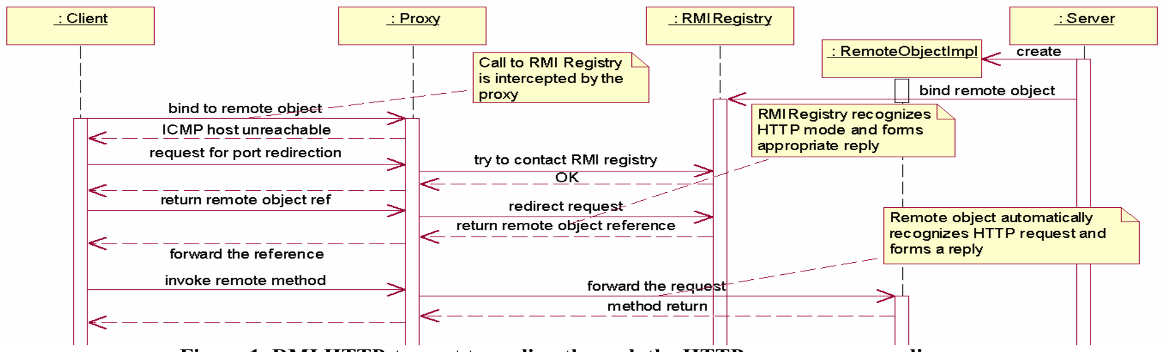
Most firewalls and proxy configurations will deny any access to ports other than a selection of well-known ones. These are used to communicate with web servers and other Internet services. HTTP requests are usually sent to port 80 while secure HTTPS (e.g. SSL – Secure Sockets Layer or TLS – Transport Layer Security) requests are sent to port 443. Installing a firewall on the client side would have two implications on distributed application using Java RMI. First, it would block access to the RMI Registry running on port 1099. Second it would block direct access to the RMI server if it was not running on port 80 or 443. There are two alternatives to solve the problem: use of RMI tunneling, which includes HTTP-to-port, HTTP-to-CGI and HTTP-to-servlet tunneling; or use of SOAP protocol (web services), which usually uses HTTP for transport. We will evaluate both alternatives, first focusing on the RMI tunneling.

2.1. RMI Over Open Ports

The most straightforward solution to make Java RMI work through the firewall is to open two ports. If there is no web server running behind the firewall, there are probably already two free, open ports on the firewall: 80 and 443. Forcing the RMI server and RMI Registry to run on those two ports would make them accessible to the outside network. This approach is not without downsides. Opening ports on the firewalls is rarely an option because it compromises the security of the network considerably. A conscious administrator would not even agree to temporarily open new ports because they tend to become permanently open and after a few years the firewall may come to resemble Swiss cheese [18]. Using ports 80 and 443 is also rarely an option, because most networks have a web server. Even if these ports are free, the RMI clients will have to know the port numbers of the RMI server and the RMI Registry, because they are not running on the standard port numbers. This solution is therefore rarely an option.

2.2. HTTP-to-port Tunneling

In a situation where there is an HTTP proxy between the client and the server we can use HTTP-to-port tunneling. The client will send a specially formatted HTTP POST request to the proxy. The proxy will read that request and make direct connections to corresponding RMI registry and RMI remote object ports on the server. Then it will accept a response from the server and return it to the client. The server and the client are thus communicating indirectly, through the proxy. The sequence diagram is shown in Figure 1. The performance penalty is mainly related to the marshalling and unmarshalling of requests and their forwarding and also heavily depends on the proxy performance, as our performance measurements have shown. The functionality of marshalling and unmarshalling RMI requests in HTTP post requests is part of the Java RMI.



: Client
: Proxy
: RMI Registry
: Server
: RemoteObjectImpl
create
Call to RMI Registry is intercepted by the proxy
bind remote object
bind to remote object
RMI Registry recognizes HTTP mode and forms appropriate reply
ICMP host unreachable
request for port redirection
try to contact RMI registry
OK
return remote object ref
redirect request
return remote object reference
Remote object automatically recognizes HTTP request and forms a reply
forward the reference
invoke remote method
forward the request
method return

Figure 1. RMI HTTP-to-port tunneling through the HTTP proxy sequence diagram

There are two ways to implement HTTP-to-port tunneling. First, we can use automatic fall-back functionality provided by the Java RMI. The problem is that this approach will only work if the proxy rejects instead of drops packets. When rejecting, it should also return one of the following error codes: `NET_UNREACHABLE`, `HOST_UNREACHABLE`, `PORT_UNREACHABLE`, `NET_PROHIBITED`, or `HOST_PROHIBITED`. This makes Java raise one of the following exceptions: `java.net.NoRouteToHostException` or `java.net.UnknownHostException`. The standard system properties `http.proxyHost` and `http.proxyPort` must also be set to the hostname and the port of the HTTP proxy.

Secondly, we can force RMI to use HTTP-to-port tunneling. This is particularly useful in cases where the proxy server cannot be configured accordingly to return requested errors because of technical or security reasons. This can be done by setting the `RMISocketFactory` to `RMIHttpToPortSocketFactory`. The major disadvantage of both approaches is that inward callback calls and multiplexed connections are not supported [16]. Caching data on the proxy could also result in integrity problems.

2.3. HTTP-to-CGI/Servlet Tunneling

With the firewall on the client side, on the server side, or on both sides HTTP-to-port tunneling is insufficient because the RMI server cannot be reached directly if it is running on a protected port. An additional mechanism for port redirection is needed. This can be done using a special CGI script or a servlet capable of redirecting a request to a designated port. The one that is part of the Java SDK uses the `sun.rmi.transport.proxy.CGIHandler` to forward requests. It has to be available at `http://server:80/cgi-bin/java-rmi.cgi`.

As in the previous case, we have two options. The first one is to use automatic fall-back to HTTP-to-CGI/servlet tunneling, as shown on the sequence diagram on Figure 2. The system property `http.proxyHost` has to be set to activate the automatic fall-back mechanism; however the address of the web server has to be provided instead of the HTTP proxy. Again, the firewall should return proper error code as in the HTTP-to-port tunneling scenario. The second is to force the HTTP-to-CGI/servlet tunneling by setting the `RMISocketFactory` to `RMIHttpToCGISocketFactory`.

The drawbacks are similar to HTTP-to-port tunneling with additional performance penalty of port redirections. Using a servlet instead of the CGI script results in increased performance because in contrast to CGI servlet does not require a new process for every invocation. Therefore the server load is lower too. This approach works with Java-compliant web servers.

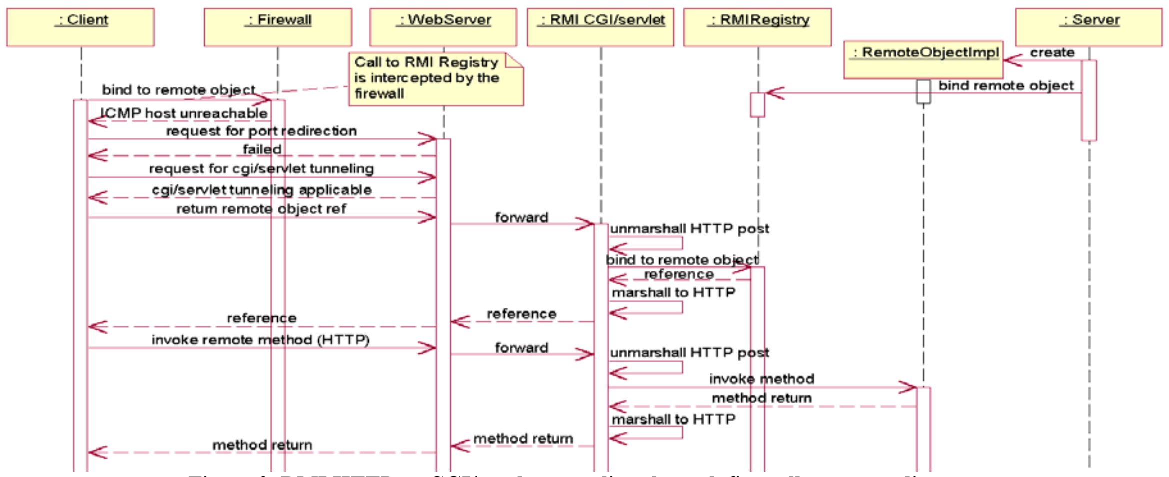


Figure 2. RMI HTTP-to-CGI/servlet tunneling through firewall sequence diagram

2.4. Decision Table

The selection of the most appropriate RMI tunneling approach depends on several criteria, such as presence of the firewall, presence of the proxy, ability to return correct error cods, availability of open ports, ability to open ports and availability of Java web server, as shown in Table 1.

Firewall	yes	yes	no	no	yes	yes	yes	yes
HTTP proxy	no	no	yes	yes	n/i	n/i	n/i	n/i
Returns correct error code	n/i	n/i	yes	no	yes	no	yes	no
Two free open ports available	yes	no	n/a	n/a	no	no	no	no
Can open additional ports	no	yes	n/a	n/a	no	no	no	no
Java web server available	n/i	n/i	n/i	n/i	no	no	yes	yes
Recommended approach	*RMI over open ports*	*RMI over open ports*	*automatic HTTP-to-port tunneling*	*forced HTTP-to-port tunneling*	*automatic HTTP-to-CGI tunneling*	*forced HTTP-to-CGI tunneling*	*automatic HTTP-to-servlet tunneling*	*forced HTTP-to-servlet tunneling*

Table 1: RMI tunneling decision table

3. Transition to Web Services

Web services are a serious alternative to RMI for distributed applications. They use the XML-based SOAP (Simple Object Access Protocol) for communication between remote services. SOAP builds on top of existing Internet protocols, such as HTTP, FTP or SMTP. Therefore SOAP can transverse firewalls seamlessly, because firewalls will treat SOAP messages similarly as other HTTP (or FTP, SMTP) messages, respectively to the transport protocol used for SOAP messages.

SOAP is not bound to a particular programming language or software platform, therefore the transition to SOAP enables interoperability with services developed on other platforms (such as Microsoft .NET). Web services are defined with the WSDL (Web Services Description Language), which enables language-independent description of types, messages, port types, operations, and ports used in a web service. Instead of the RMI Registry web services use the UDDI (Universal Description, Discovery and Integration) registry [17].

For implementing web services in Java developers can use the JWSDP (Java Web Services Developer Pack), which provides a set of APIs related to XML and web services. The most important is the JAX-RPC (Java API for XML based Remote Procedure Call). Implementing a web service using JAX-RPC is similar to implementing a RMI remote object. Implementation consists of an interface extending the `java.rmi.Remote` interface and a class implementing the interface. The way a web service is exposed to remote clients however differs considerably from RMI. Web services are typically executed within the web container as servlets. The choice is to transform web service classes directly to servlets or use a servlet handler which forwards the requests to appropriate classes. Java Web Services Developer Pack (JWSDP) uses the latter technique.

In addition to understanding JAX-RPC, web server container internals, web services technologies and platform specific tools a developer will also need to understand at least the following Java APIs:

- Java API for XML Processing (JAXP), which provides a parser and support for XML Schemas, DTD (Document Type Definition), XSLT (Extensible Style-sheet Language for Transformations), DOM (Document Object Model) and SAX (Simple API for XML);
- Java Architecture for XML Binding (JAXB), which provides support for XML serialization and mapping for object model to XML Schemas and vice versa;
- Java API for XML Registries (JAXR), which provides an API for dealing with UDDI registries.

Web services overcome the problems of firewalls; however they do not provide the whole functionality offered by RMI. The functionality not offered by web services includes:

- support for object references,
- support for distributed garbage collection,
- dynamic class loading,
- remote object activation,
- RMI security (web services provide WS-Security instead).

4. Performance Analysis and Comparison

4.1. Test Method

In the second part of this paper we present quantitative results of performance comparison of RMI, RMI tunneling alternatives and web services. Performance can be one of the more important criteria factors in the selection of the most appropriate technology solution for Java distributed applications, which have to support communication through firewalls. We have compared plain RMI, HTTP-to-port, HTTP-to-CGI/servlet and web services scenarios. For the HTTP-to-CGI/servlet we have selected a servlet, which is a more appropriate solution than a CGI script in terms of performance, as identified earlier in this paper. We expect that RMI tunneling performance will be significantly slower than plain RMI, with HTTP-to-port tunneling being faster than HTTP-to-servlet. Because RMI tunneling and web services use HTTP as the underlying protocol, we would also expect that the performance of RMI tunneling and web services will be comparable.

We have measured the instantiation time and the method invocation time using the subset of the performance assessment framework [8]. We wanted to gain results that will make it possible to understand how the alternative solutions perform and why there are differences in performance. An important goal was to make the results comparable. For the purposes of this article we have measured the round trip method invocation times and the instantiation times. Round trip method invocation time is the time that elapses between the initiation of a method invocation by the client until the results are returned to the client. Because the methods used for performance evaluation did not do any processing, the round trip time expresses the overhead of remote method invocation. It is important to understand how different data types influence the result; therefore we have measured round trip times for eight data types: integer, short integer, long integer, float, double, boolean, byte, and string. We have achieved the comparability of the results between RMI and web services with identical implementations that differ only in necessary details regarding obtaining the initial references. Further, we have used a consistent mapping between Java and web services (XML Schema) data types, as shown in Table 2. For the web services implementation we have used SOAP RPC/encoded data representation. We have accomplished the measurements on identical equipment in an identical environment.

To test the performance we have used two identically configured computers, one acting as a client and one as a server. The computers had Intel Pentium 4 processors running at 2.4 GHz (FSB 133 MHz), with Microsoft Windows 2000 operating system. Both computers were installed with the same settings, running only essential services. We have used the Java 2 Platform Standard Edition version 1.4.2_03_b02 and Java Web Services Developer Pack version 1.3. Web server was the Apache Tomcat v5.0 and the HTTP Proxy was the open source Privoxy version 3.0.2, because they also run on other platforms, which enables a future performance comparison between platforms. The computers were connected to a 100 Mb/s switched network, free of other traffic, and restarted before each test to ensure the same starting conditions.

To measure the performance, we have defined the `IPerformanceTester` interface with the corresponding implementation classes, shown on Listing 1. Every scenario was analyzed in a series of steps:

- First 5000 instantiations were made to initialize the server component. Initialization was needed to allow all communicating components to start fully exploiting built-in optimization features, such as pooling and caching. The number of initializations was determined with the calculation of the coefficient of variation.
- For each test 1000 invocations were measured to gain the necessary resolution.
- Each test was repeated 10 times and the average time, variation, and the coefficient of variation have been calculated.

```
public interface IPerformanceTester extends Remote {
    int getInt() throws RemoteException;
    short getShort() throws RemoteException;
    long getLong() throws RemoteException;
    float getFloat() throws RemoteException;
    double getDouble() throws RemoteException;
    boolean getBoolean() throws RemoteException;
    byte getByte() throws RemoteException;
    String getString() throws RemoteException;
}
```

Listing 1. Java RMI and web services remote interface for performance tests

Java RMI data type	XML Schema data type
int	xsd:int
short	xsd:short
long	xsd:long
float	xsd:float
double	xsd:double
boolean	xsd:boolean
byte	xsd:byte
String	xsd:string

Table 2: Data types mappings between Java RMI and XML Schema (web services)

4.2. Performance Results

4.2.1 Local performance

First we measured local performance for all scenarios, where all tests were run on the same computer to avoid network overhead. The results are shown in Table 3. Instantiation was considered separately (shown as instantiation). We can see that the times for basic data types (`int`, `short`, `long`, `float`, `double`, `boolean`, and `byte`) do not differ considerably, therefore we have calculated the geometric average. This is because the data type payload represents only a small portion of the whole JRMP or SOAP message payload exchanged between the client and the server.

Time in ms	***RMI***	***HTTP-to-port***	***HTTP-to-servlet***	***Web services***
Instantiation	0,686	19,070	19,483	0,616
Simple types average:	0,157	5,052	7,663	2,336
- int	0,157	5,044	7,659	2,330
- short	0,156	5,041	7,689	2,356
- long	0,156	5,050	7,622	2,319
- float	0,157	5,052	7,680	2,375
- double	0,161	5,066	7,689	2,342
- boolean	0,155	5,059	7,670	2,316
-byte	0,155	5,050	7,631	2,313
String	0,172	5,105	7,658	2,353

Table 3: Local performance results

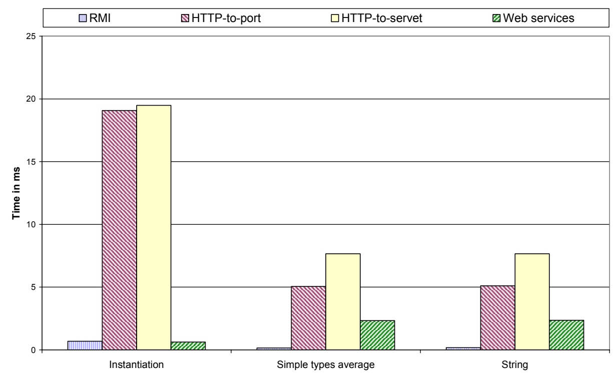


Figure 3. Performance results without network overhead

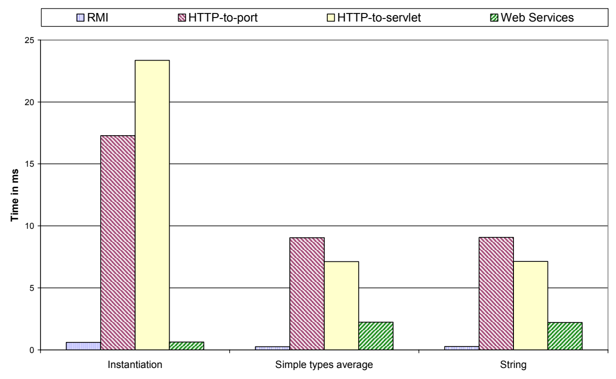


Figure 4: Performance results with network overhead

From Figure 3 we can see, that RMI offers results, which are an order of magnitude better than the other alternatives. Web services are the second fastest alternative. By the basic data types, they are on average ~15 times slower than RMI, by string, web services are ~13.6 times slower. It is interesting, that by the instantiation, web services are ~10% faster than

RMI. The RMI tunneling scenarios are even slower. HTTP-to-port tunneling is ~32 times slower on average basic data types and ~30 times slower on string than RMI and still ~2 times slower than web services on basic data types and string. HTTP-to-servlet is almost ~49 times slower on basic data types and ~44 times on string than RMI and ~3.2 times than web services. By the instantiation all RMI HTTP tunneling alternatives are ~28 times slower than RMI and ~31 times slower than web services.

According to the measurements the best alternative to RMI are web services and the worst HTTP-to-servlet tunneling. Compared to our previous work [5], RMI performance improved significantly from an average of 1.54 ms for simple data type local RMI calls to 0.16 ms in this research for the same data types. This is ~9.6 times faster. Web services performance today is comparable to RMI performance, as measured in [5].

4.2.2 Networked performance

To include network overhead we have repeated the tests on two network-connected computers. The server-side functionality (web server and service implementation) was installed on one computer and the client-side functionality on the other computer. From Table 4 and Figure 4 we can see that in the remote network scenario RMI still performs much faster than the other alternatives. The average factor comparison however shows a slightly different picture than in local scenario. HTTP-to-servlet tunneling is now considerably faster than HTTP-to-port. The reason can be found in the internals of proxy software and the fact that HTTP-to-port works almost 80% slower when used over a network, because local optimizations cannot be applied anymore.

The best alternative to RMI is still web services. When used over the network they are ~8.8 times slower for average basic data types and ~8.5 times slower for string than RMI compared to ~15 and ~13.6 for local scenario. This is even more important in real-world scenarios because most clients will invoke services over the network. HTTP-to-servlet, the second fastest alternative, is ~28 times slower than RMI for all data types and ~3.2 times slower than web services. HTTP-to-port is ~36 times slower than RMI and ~4 times slower than web services. This is shown in Figure 5.

Time in ms	***RMI***		***HTTP-to-port***		***HTTP-to-servlet***		***Web Services***	
Factor to local	Remote	*Factor*	Remote	*Factor*	Remote	*Factor*	Remote	*Factor*
Instantiation	0,600	*0,87*	17,283	*0,91*	23,352	*1,20*	0,620	*1,01*
Simple types average	0,251	*1,60*	9,051	*1,79*	7,117	*0,93*	2,224	*0,95*
- int	0,254	*1,62*	9,112	*1,81*	7,305	*0,95*	2,189	*0,94*
- short	0,248	*1,59*	9,050	*1,80*	7,053	*0,92*	2,225	*0,94*
- long	0,250	*1,60*	9,039	*1,79*	7,048	*0,92*	2,224	*0,96*
- float	0,252	*1,61*	9,042	*1,79*	7,067	*0,92*	2,247	*0,95*
- double	0,252	*1,57*	9,042	*1,78*	7,125	*0,93*	2,253	*0,96*
- boolean	0,248	*1,61*	9,042	*1,79*	7,108	*0,93*	2,222	*0,96*
-byte	0,250	*1,62*	9,033	*1,79*	7,114	*0,93*	2,208	*0,95*
String	0,261	*1,51*	9,078	*1,78*	7,138	*0,93*	2,206	*0,94*

Table 4: Remote performance results

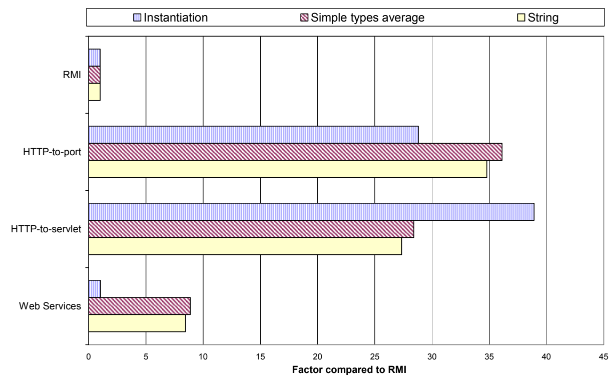


Figure 5. Relative factor of slowdown, compared to RMI

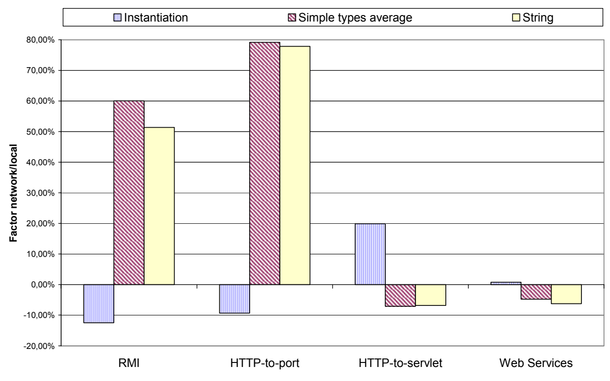


Figure 6: Network vs. local performance comparison

Performance impact of a network connection adds an average of 59% in the case of RMI and 79% in the case of HTTP-to-port tunneling. HTTP-to-servlet and web services performance improved for 7% and 5% respectively. This was due to load distribution. When running on a single computer the CPU utilization reached maximum. Gain from load distribution was higher than the degradation from the network connection resulting in some improvement. In the network scenario the instantiation times for RMI and web services were almost identical. We can observe instantiation time improvement for RMI, while web services instantiation is almost identical in local and remote scenario. In the HTTP-to-port scenario the instantiation time also improved by ~10%, while in the HTTP-to-servlet scenario the instantiation times were ~20% slower. Comparing the results to our previous work [5], where average times for RMI increased for ~37% in the network scenario, an increase of ~59% is a step back. This is due to the fact that processor performance increased faster than network throughput.

A detailed analysis of the networked vs. local results shows considerable performance degradation in the network scenario for alternatives that are not web server based, which is shown in Figure 6. The reason can be found in the local optimizations, performed by RMI and proxy, which allowed comparably better times in the local scenario. The times for web server based solutions improved slightly in the network scenario because of load distribution, as we noted previously.

4.3. Overhead Analysis

To be able to understand the reasons for the differences in performance between the described scenarios, we have done an overhead analysis. We have used the Borland OptimizeIt Enterprise Suite 6 profiler and the PerfAnal tool [19]. We have profiled the test scenarios and identified the methods and packages in which the majority of time was spent. Then we have calculated the percentage of time and compared the methods between the scenarios. The results are shown in Table 5.

Package / Method (in %)	*RMI*	*HTTP-to-port*	*HTTP-to-servlet*	*Web Services*
java.net.socketOutputStream.socketWrite	29,23	37,76	69,84	16,38
sun.rmi.transport	24,16	<0,10	<0,10	<0,10
com.sun.xml.rpc.sp	<0,10	<0,10	<0,10	14,37
com.sun.xml.rpc.encoding.soap	<0,10	<0,10	<0,10	12,79
java.net.PlainSocketImpl.doConnect	<0,10	17,85	0,68	<0,10
sun.net.www	<0,10	14,63	7,10	4,73
java.io.ObjectInputStream	5,05	<0,10	<0,10	<0,10
sun.rmi.server	3,82	<0,10	<0,10	<0,10
java.io.ObjectOutputStream	5,63	<0,10	1,10	<0,10
java.lang.SecurityManager	<0,10	2,36	<0,10	<0,10
java.net.PlainSocketImpl	<0,10	2,66	0,98	<0,10
sun.nio.cs	<0,10	1,07	<0,10	1,19

Table 5: Overhead analysis with relative times spent in packages/methods for different scenarios

From the results we can see that web services spend considerably more time for XML serialization and deserialization, which is used for creating SOAP messages (`com.sun.xml.rpc` package). RMI on the other hand uses binary serialization (`java.io.ObjectInputStream/ObjectOutputStream`), which is more effective in terms of performance, as shown in [20]. XML serialization/deserialization requires more marshaling/demarshaling costs because XML parsing includes a lot of string comparison and matching, and data type casting. RMI is also more effective in terms of the size of network messages. The binary JRMP representation is ~29 times smaller than the SOAP representation, which uses XML. The tests which we have performed in this research were limited to primitive data types, but we can infer that if the payload, exchanged between the client and the server would be larger in size, the advantage of RMI would be even more evident.

For both RMI tunneling alternatives, HTTP-to-port and HTTP-to-servlet there are two reasons for slower performance: a lot of additional communication, related to fallback and routing/forwarding of requests, as shown in Figures 1 and 2; and the additional marshaling and demarshaling costs related to creation of HTTP POST requests and the encoding/decoding of binary JRMP messages. In our overhead analysis this is expressed in more than 30% of time spent in `java.net` and `sun.net` packages for HTTP-to-port tunneling and in the disproportional high share of `java.net.SocketOutputStream` for the HTTP-to-servlet scenario.

A specific overhead factor is also the instantiation, which is important particularly in scenarios where clients communicate with many server remote objects. RMI and RMI tunneling alternatives use `java.rmi.Naming.lookup` method to obtain references to a remote objects. The additional overhead of RMI tunneling alternatives is related to the `sun.rmi.RegistryImpl_Stub.lookup` method in which HTTP-to-port and HTTP-to-servlet scenarios spent ~1.4 times the time as spent in the normal RMI lookup. For web services lookup and invoke are combined. That is why `com.sun.xml.rpc.client.http.HttpClientTransport.invoke` spent 62% of the time making a connection in `com.sun.xml.rpc.client.http.HttpClientTransport.connectForResponse` and the rest to actually read the response in `com.sun.xml.rpc.client.http.HttpClientTransport.readResponse`.

The optimizations should therefore focus on the performance bottlenecks as identified in this research:

- Marshaling/demarshaling overhead can be minimized with the use of fast, delayered and flexible marshaling algorithms. Instead of the layered marshaling/demarshaling a perfect hashing and active demarshaling can be implemented for best performance.
- SOAP overhead related to marshaling can be further reduced with the parser optimizations, use of pre-generated methods for obtaining the state of the objects and in the reduction of generated SOAP messages.
- Stub/skeleton layer overhead optimization can be achieved with generation of optimized stubs and skeletons, which use static method calls instead of reflection.
- Instantiation overhead can be minimized with the introduction of instance and connection pooling and management algorithms together with pre-instantiation and reuse of physical entities, which reduces local garbage collection overhead.
- Implementation related optimizations, such as omitting the unnecessary data copying, optimized range checking, minimization of local method invocations, reducing the overhead of frequently called methods with the optimization for the common case, replacement of large methods with efficient small special purpose methods, avoiding the repeated

computation of invariant values and storing redundant data. Also, elimination of the run-time checking for the debugging code brings some improvements.

5. Conclusions

In this article we have compared technology alternatives for developing distributed Java applications, which have to communicate through firewall and proxy secured networks. These alternatives can be classified in two groups: (1) Using RMI tunneling techniques, including HTTP-to-port, HTTP-to-CGI and HTTP-to-servlet tunneling; and (2) using Web Services instead of Java RMI. The comparison of RMI tunneling alternatives has shown that the transition to RMI tunneling is related with administrative tasks, including the deployment and configuration of corresponding tunneling components and settings. We have compared different RMI tunneling approaches and provided a decision table to help make the most appropriate selection based on parameters such as firewall and proxy configuration, availability and ability to open ports, and availability of Java web server. We have also evaluated the transition to web services, which is related with significant training and development efforts as the technologies used by RMI and web services differ considerably. We have also identified the functionality not offered by web services.

Often an important parameter for distributed applications is performance; therefore we have analyzed and compared the performance of RMI, RMI tunneling and web services. We have identified that RMI offers an order of magnitude better performance than other alternatives, being at least 8.5 times faster than the second alternative – web services. HTTP-to-port and HTTP-to-servlet tunneling are even slower. In networked scenario HTTP-to-servlet is more than 3 times slower and HTTP-to-port more than 4 slower times than web services. We have seen that RMI tunneling alternatives offer poor performance compared to RMI therefore web services should be used for performance sensitive distributed applications if ports for RMI communication cannot be opened. To identify the bottlenecks we have made an overhead analysis and identified the following groups for optimizations: marshaling/demarshaling optimizations, stub/skeleton optimizations, optimizations related to instantiation, SOAP-related optimizations, and implementation related optimizations.

References

[1] Kenji Kono, Takashi Masuda, Efficient RMI: Dynamic Specialization of Object Serialization, The 20th International Conference on Distributed Computing Systems (ICDCS 2000), April 10 - 13, 2000, Taipei, Taiwan

[2] Christian Nester, Michael Philippsen, Bernhard Haumacher, A more efficient RMI for Java, Java Grande Conference, Proceedings of the ACM 1999 conference on Java Grande, San Francisco, 1999

[3] Juric B. Matjaz, Rozman Ivan, Nash Simon, Java 2 Distributed Object Middleware Performance Analysis and Optimization, ACM SIGPLAN Notices, 2000, vol. 35, no. 8, pg. 31-40

[4] Juric B. Matjaz, Performance Comparison of CORBA and RMI, Information and Software Technology Journal, Elsevier, 2000, vol. 42, no. 13, pg. 915-933

[5] Hericko Marjan, Juric B. Matjaz, Zivkovic Ales, Rozman Ivan, Java and Distributed Object Models: An Analysis, ACM SIGPLAN Notices, December 1998

[6] Juric B. Matjaz, Zivkovic Ales, Rozman Ivan, Are Distributed Objects Fast Enough, *Mora Java Gems,* Cambridge University Press, March 2000

[7] Juric B. Matjaz, Rozman Ivan, Java 2 RMI and IDL Comparison, Java Report, February 2000, Vol. 5, No. 2, pg. 36-48

[8] Juric B. Matjaz, Domajnko Tomaz, Zivkovic Ales, Hericko Marjan, Brumen Bostjan, Welzer Tatjana, Rozman Ivan, Performance Assessment Framework for Distributed Object Architectures, Conference proceedings of ADBIS'99, LNCS series of Springer Verlag, September 1999

[9] Juric B. Matjaz, The Efficiency of Distributed Object Models, ACM OOPSLA'99, November 1999

[10] Aniruddha S. Gokhale, Douglas C. Schmidt, The Performance of the CORBA Dynamic Invocation Interface and Dynamic Skeleton Interface over High-Speed ATM Networks, IEEE GLOBECOM '96 conference, November 18-22nd, 1996, London, England

[11] Aniruddha S. Gokhale, Douglas C. Schmidt, Measuring and Optimizing CORBA Latency and Scalability Over High-Speed Networks, IEEE Transactions on Computers, Vol. 47, No. 4, April 1998

[12] Plasil F., Tuma P., Buble A.: CORBA Benchmarking, Tech. Report No. 98/7, Dep. of SW Engineering, Charles University, Prague

[13] Christopher D. Gill, David L. Levine, Douglas C. Schmidt, The Design and Performance of a Real-Time CORBA Scheduling Service, August 1998

[14] Ashish Singhai, Aamod Sane, Roy Campbell, Reflective ORBs: Supporting Robust, Time-critical Distribution, ECOOP'97 Workshop Proceedings, 1997

[15] Sun Microsystems, Inc., Java Remote Method Invocation Specification, Sun Microsystems, Revision 1.8, 2002

[16] The RMI Proxy White Paper. http://www.rmiproxy.com/doc/WhitePaper.pdf

[17] Juric M. B., Basha S. J., Leander R., Nagappan R.: Professional J2EE EAI. Wrox Press, Birmingham, 2001

[18] Garfinkel S., Firewall Follies, Technology Review, Vol. 33, September 2002

[19] Nathan Meyers, Java Programming on Linux, Macmillan Computer Publishing, 1999

[20] Hericko Marjan, Juric B. Matjaz, Ivan Rozman, Simon Beloglavec, Ales Zivkovic, Object Serialization Analysis and Comparison in Java and .NET, ACM SIGPLAN Notices, August 2003, vol. 38, no 8