Qualitative and Quantitative Analysis and Comparison of Java Distributed Architectures

Ivan Rozman, Matjaz B. Juric, Izidor Golob, Marjan Hericko

Address:

University of Maribor

FERI, Institute of Informatics

Smetanova 17

SI-2000 Maribor

Europe

Corresponding author:

Matjaz B. Juric

University of Maribor

FERI, Institute of Informatics

Smetanova 17

SI-2000 Maribor

Europe

E-mail: matjaz.juric@uni-mb.si

URL: http://lisa.uni-mb.si/~juric/

Phone: +386 2 235 5113

Fax: +386 2 235 5134

Qualitative and Quantitative Analysis and Comparison of Java Distributed Architectures

Summary

In this article we have done a qualitative and quantitative comparison of common approaches used to develop distributed solutions in Java: RMI and Web services for regular unsecured communication, RMI-SSL and WS-Security for secure communication and authentication, and HTTP-to-port and HTTP-to-CGI/servlet tunneling for RMI communication through firewalls and proxies. We have done a functional comparison that helps with the selection of the most appropriate approach. We have also done a detailed performance analysis with the identification of major bottlenecks, identification of design and implementation guidelines for distributed applications, and specification of optimizations for distributed middleware. The paper contributes to the understanding of different approaches for developing Java distributed applications, provides detailed performance analysis, presents design and implementation guidelines, and identifies the major performance overheads.

Keywords: RMI, Web services, Performance, Java

Introduction

The development of distributed applications on Java platform provides several choices regarding the distributed architectures. RMI is the Java native distributed object architecture. For communication it uses the binary JRMP (Java Remote Method Protocol), which is based on TCP/IP. RMI communication is by default blocked by firewalls. This can be problematic for distributed applications, which require interoperability over LANs secured by firewalls. Security restrictions in most cases do not allow opening the ports on the firewalls that are required by RMI. RMI tunneling techniques (HTTP-to-port and HTTP-to-CGI/servlet) enable RMI connectivity through firewalls and proxies. By default RMI and RMI tunneling communication is unsecured. For secure communication RMI-SSL can be used, which uses SSL (Secure Socket Layer) and provides communication channel encryption and authentication. RMI-SSL also enables connectivity through firewalls.

The other important distributed architecture is web services, which differs considerably from RMI. Web services use the XML-based SOAP (Simple Object Access Protocol) for communication and are not limited to the Java platform. They

enable interoperability with other platforms, such as Microsoft .NET. Java provides good support for developing web services through APIs such as JAX-RPC (Java API for XML-based Remote Procedure Call). Web services usually use standard Internet protocols for communication therefore they are not blocked by firewalls. By default web services use unsecured communication protocols. Secure communication (and authentication) is achieved with WS-Security, which provides message-level security.

Selection of the most appropriate approach is important as it influences the architecture and the performance of the applications. In this paper, we have done a functional comparison of RMI, RMI-SSL, RMI tunneling (HTTP-to-port and HTTP-to-CGI/servlet), web services, and WS-Security. We have identified the major differences to ease the selection. An important decision criterion can be performance, particularly because of the differences in performance between the approaches. This is why we have done a detailed performance analysis and comparison, identified the major sources of overhead and proposed several optimizations. Based on the analysis and comparison we have presented design and implementation guidelines for distributed applications which affect the performance.

The review of related work has shown that a comparison and performance analysis of RMI, RMI-SSL, RMI tunneling, web services, and WS-Security has not been done yet. In References [1] and [2] the authors present optimized versions of RMI and provide measurements that show the improvements, but do not compare the performance to other distributed architectures and do not preserve compatibility with original RMI. In Reference [3] the authors compare the performance of native sockets, Java sockets and RMI on Windows, Solaris, and Linux. They measure latency and throughput and show that RMI performs slower than sockets. They use only a limited set of data types for performance measurements and do not provide optimizations. In Reference [4] authors measure and compare latency of RMI, CORBA, and SOAP. The authors measure performance for void, integer, and string, which differs from the measurement method in our research. They also do not compare secure versions (RMI-SSL, WS-Security) or RMI tunneling. There are also some papers (References [5], [6], and [7]) in which the performance of different CORBA implementations with C++ programming language is compared and optimized. The authors in References [8] and [9] compare the CORBA performance for the purposes of real-time systems. In our earlier research we have proposed a set of benchmarks for distributed object architectures [10]. We use an updated version of these benchmarks in this paper too. We have done performance comparison of CORBA and RMI [11, 12], and presented performance optimizations of RMI-IIOP [13]. We have analyzed and compared regular (unsecured) RMI, RMI tunneling, and web services [14], and compared the performance of regular and secured RMI and web services [15]. At the time of writing there was no comprehensive Java-related analysis of web services, WS-Security, RMI, RMI-SSL, and RMI tunneling. Therefore we believe that this article contributes to the overall understanding of these approaches.

The article is organized as follows: In section two a functional comparison of web services, RMI, and RMI tunneling is given. In section three the performance analysis method is presented. In section four local and remote performance analysis and comparison of RMI, RMI-SSL, RMI tunneling, web services, and WS-Security is presented. In section five the analysis and interpretation of the results is presented with the design and implementation guidelines for distributed applications and optimizations for distributed middleware. Concluding remarks are given in the last section.

Comparison of Web Services, RMI, and RMI Tunneling

Web Services and RMI

Web services enable interoperability between different platforms and programming languages. The information flow in web services is based on XML formatted payloads [16] that act as input, output, and/or fault messages. The combination of these messages determines the operation types, which can be request/response or one-way (but also solicit response and notification) [17]. 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. For communication web services use SOAP (Simple Object Access Protocol) [16] and support synchronous and asynchronous, one-way and two-way interactions that can be of RPC (Remote Procedure Call) or document style. RPC usually uses SOAP encoding (RPC/encoded) while document uses literal encoding (document/literal). SOAP encoding follows the encoding rules as defined in the SOAP specification [16] while literal uses XML Schema definitions. For communication SOAP uses an underlying protocol, such as HTTP, FTP, etc. and is by default unsecured. WS-Security provides message-level security and supports authentication and communication security. Web services are the technology foundation for SOA (Service Oriented Architecture). SOA enables flexibility and interoperability of applications and is independent of underlying technologies. SOA is based on the services and their composition with the objective to provide agile and flexible support for realization of business processes [18].

RMI is Java-native distributed object architecture and follows the object paradigm. A distributed object exposes a remote interface through which the clients can invoke methods. RMI supports synchronous request/response method invocation and stateful objects. The three layers of RMI are the stub/skeleton layer, the remote reference layer, and the transport layer [19]. They provide support for client-side stubs and server-side skeletons, for reference and invocation behavior and for connection setup and management as well as remote object tracking, respectively. For communication RMI uses binary protocol that can be either proprietary JRMP (Java Remote Method Protocol) or CORBA-based IIOP (Internet Inter-ORB

Protocol). For secure communication and authentication RMI uses SSL (Secure Socket Layer) or TLS (Transport Layer Security) [20].

The major difference between RMI and web services is related to the definition of information flow between the distributed object/service and the client. RMI is distributed object architecture and the interaction between a distributed object and the client is based on the remote interface definition. In web services the information flow is based on XML formatted payloads. For naming RMI uses JNDI (Java Naming and Directory Interface) or RMI registry, while web services use UDDI (Universal Description, Discovery and Integration). Table 1 shows a functional comparison of RMI and web services. Web services lack some features found in RMI:

- Support for object references and stateful objects,
- Dynamic class downloading,
- Support for distributed garbage collection,
- Remote object activation.

	Web Services	**RMI**
Type of operations supported	Synchronous and asynchronous	Synchronous
Support for one-way operations	Yes	No
Support for object/service state	Stateless	Stateless and stateful
Automatic code propagation	No	Yes
Distributed memory management	No	Yes (with JRMP)
Wire protocol type	XML	Binary
Object/service description	WSDL	Java interface
Object/services directory	UDDI	JNDI/Registry
Binding to instances in run-time	Yes	Reflection
Remote instance creation	No	Yes
Security	WS-Security	SSL/TLS

Table 1: Functional comparison of RMI and Web Services

The implementation of RMI distributed objects and web services using Java is comparable. Using the JAX-RPC (Java API for XML based Remote Procedure Call) API for web services and standard RMI remote package, the implementation code of RMI distributed objects and web services consists of an interface extending the `java.rmi.Remote` interface and a class implementing the interface. RMI distributed objects are deployed as processes while with web services we have the choice between transforming web service classes directly to servlets or to use a servlet handler which forwards the requests to appropriate classes.

RMI Tunneling and Firewalls

The RMI transport layer opens direct sockets to remote object hosts, which are blocked by firewalls. Firewalls block all traffic to a set of source and/or destination ports [22] meaning that services which use blocked ports are not accessible for the clients. Usually the only open ports are the well-known ports, such as port 80 for HTTP and 443 for HTTPS. Firewalls can also be installed on the client side, which results in blocking access to the RMI registry running on port 1099 and to the RMI server. Firewalls also block call-backs. In a call-back a RMI server reference is passed as a parameter of a remote method invocation. To overcome this problem RMI 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 [19]. To enable RMI connectivity through firewalls there are several alternatives:

- Use of RMI over open ports.
- Use of RMI tunneling, which includes HTTP-to-port and HTTP-to-CGI/servlet tunneling.
- Use of RMI-SSL, which provides communication channel security and authentication (point-to-point security).
- Use of web services or WS-Security. The SOAP protocol, which usually uses HTTP for transport, is not blocked by firewalls.

Using RMI over open ports is possible only if ports 80 and 443 are not used, which is the case if there is no web server running behind the firewall. We can force RMI server and RMI registry to use these two ports, making them accessible from outside. In addition to the rare situation when there is no web server this solution has the disadvantage of using non-standard port numbers that the clients have to be aware of. Opening additional ports on firewalls compromises the network security [23] therefore it is often not an option.

HTTP-to-port tunneling can be used when there is a HTTP proxy between RMI client and RMI server. RMI client has to use a special HTTP POST request to the proxy. The proxy will establish direct connections to RMI registry and RMI server, invoke the operations, and return the results to the original RMI client. The proxy is thus acting as a mediator (Figure 1) that marshals, unmarshals, and forward the requests. This influences the performance. HTTP-to-port tunneling can be activated automatically by proxy, which has to reject instead of drops packets and return one of the following error codes: `NET_UNREACHABLE`, `HOST_UNREACHABLE`, `PORT_UNREACHABLE`, `NET_PROHIBITED`, or `HOST_PROHIBITED`. The standard system properties `http.proxyHost` and `http.proxyPort` must also be set to the hostname and the port of the HTTP proxy. HTTP-to-port tunneling can also be forced, which is useful in scenarios where HTTP proxy drops packets. To achieve it the `RMISocketFactory` property has to be set to `RMIHttpToPortSocketFactory`. The major disadvantage of HTTP-to-port tunneling is that inward callback calls and multiplexed connections are not supported [24].

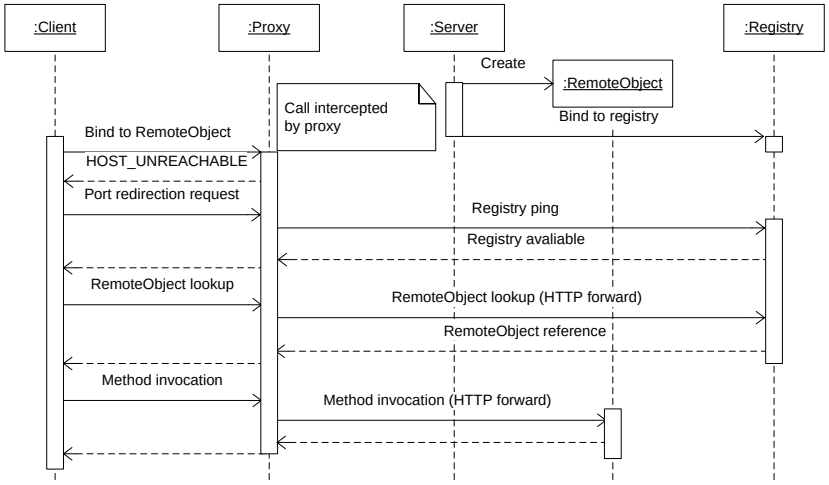

Figure 1. HTTP-to-port sequence diagram for RMI

HTTP-to-port is inadequate if there is a firewall on the client, server, or on both sides. In such case additional port redirection is needed that can be achieved using a special CGI script or servlet capable of redirecting requests to designated ports. We can use automatic fallback to HTTP-to-CGI/servlet tunneling, as shown on the sequence diagram on Figure 2. To activate it system property `http.proxyHost` has to be set to the address of web server and the firewall should return proper error code. We can also force the HTTP-to-CGI/servlet tunneling by setting the `RMISocketFactory` to `RMIHttpToCGISocketFactory`. The drawback is the additional performance penalty for 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 and places lower load on the server. This approach works with Java-compliant web servers.

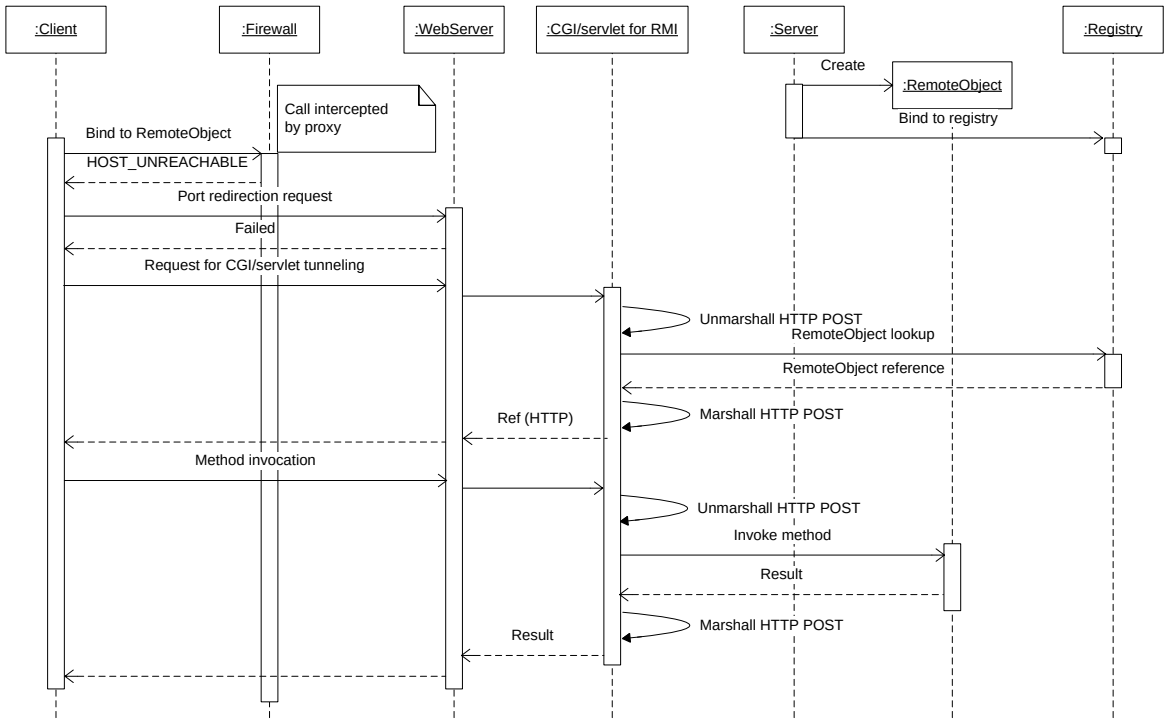

Figure 2. HTTP-to-CGI/servlet sequence diagram for RMI

Comparison Table

In Table 2 we present a comparison table for selecting the most appropriate distributed architecture based on the interoperability requirements, security requirements, firewall, HTTP proxy, and Java web server presence, and their ability to return correct error codes, availability of open ports, and ability to open additional ports.

	Interoperability with other platforms required	Secure communication required	Firewall	HTTP proxy	Java web server available	Returns correct error code	Two free open ports available	Can open additional ports
Web services	yes	no	no	yes	---	---	---	---
Web services	yes	no	yes	---	---	---	---	---
WS-Security	yes	yes	no	yes	---	---	---	---
WS-Security	yes	yes	yes	---	---	---	---	---
RMI over open ports	no	no	yes	no	---	---	yes	no
RMI over open ports	no	no	yes	no	---	---	no	yes
RMI-SSL	no	yes	yes	---	---	---	---	---
RMI-SSL	no	yes	no	yes	---	---	---	---
Forced HTTP-to-servlet tunneling	no	no	yes	---	yes	no	no	no
Forced HTTP-to-port tunneling	no	no	no	yes	---	no	n/a	n/a
Forced HTTP-to-CGI tunneling	no	no	yes	---	no	no	no	no
Automatic HTTP-to-servlet tunneling	no	no	yes	---	yes	yes	no	no
Automatic HTTP-to-port tunneling	no	no	no	yes	---	yes	n/a	n/a
Automatic HTTP-to-CGI tunneling	no	no	yes	---	no	yes	no	no

Table 2: Comparison table of distributed architectures

Performance Analysis and Comparison Method

In many cases, the performance will influence selection of the distributed architecture. In this part of the paper we have done a quantitative performance analysis and comparison of RMI, RMI-SSL, RMI-tunneling, web services, and WS-Security. For RMI-tunneling we have compared HTTP-to-port and HTTP-to-servlet tunneling. To measure the performance we have used an updated subset of the performance assessment framework [10] through which we have been able to

compare the results of different distributed architectures and understand why there are differences in performance. 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. We have measured round trip times as the average values of eight data types: integer, short integer, long integer, float, double, boolean, byte, and string. We have achieved the comparability of the results 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 [19]. For the web services implementation we have used document/literal and RPC/encoded data representation.

To measure the performance each invocation has been repeated 1000 times to gain the necessary resolution. Each measurement has been repeated 12 times, the maximal and minimal times have been discarded. The average time of 10 repetitions has been calculated. Variation and coefficient of variation have been calculated too. For RMI-SSL and WS-Security we have used server and client-side key store with authentication on both sides. We have used X.509 certificates, RSA asymmetric encryption algorithm for digital signature, and Triple-DES symmetric encryption algorithm for securing the communication. We have accomplished the measurements on identical equipment in controlled environment. We have used two identically configured computers running Microsoft Windows 2000 SP4, 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 512 MB RAM. Computers have been installed with the same settings, running only essential services. They 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 perform the measurements we have used the Java 2 Platform Standard Edition version 1.4.2_05 and Java Web Services Developer Pack version 1.4. Web server was the Apache Tomcat v5.0 and the HTTP Proxy was the open source Privoxy version 3.0.2 (both products are also available for other platforms). We have used the following three JCE (Java Cryptography Extensions) security providers:

- The Bouncy Castle Crypto version 1.24 (http://www.bouncycastle.org)
- Wedgetail JCSI Provider version 2.3 (http://www.wedgetail.com/jcsi/provider/)
- OpenSource HBCI Toolkit for Java version 0.0.6 (http://www.jhbci.de)

Performance Analysis and Comparison Results

Local Performance Results

First, we have run all tests on a single computer on which the client and server part have been deployed to avoid the network overhead. From Figure 3 we can see that RMI and RMI-SSL provide performance which is an order of magnitude better than web services. Web services perform ~6% better when using document/literal representation. Web services are also considerably faster than RMI tunneling. WS-Security offers by far the lowest performance. RMI-SSL is ~47% slower than RMI using simple types and strings and ~20% slower by instantiation. Web services are ~15 times slower than RMI using basic data types and ~13.6 times slower using strings. They are however ~10% faster by the instantiation. HTTP-to-port and HTTP-to-servlet are an order of magnitude slower than RMI. HTTP-to-port is ~32 times slower using basic data types and ~29.6 times slower using strings. It is also ~28 times slower by instantiation than RMI. HTTP-to-servlet is even slower in local scenario. It is ~49 times slower than RMI using basic data types and ~44 times slower using strings. The instantiation is comparable to HTTP-to-port and is ~28.4 times slower than RMI.

Both RMI tunneling alternatives are considerably slower than web services (Figure 4). HTTP-to-port is ~2 times slower using basic types and strings and ~31 times slower by instantiation. HTTP-to-servlet is ~3.2 times slower using basic types and strings and ~31.6 times slower by instantiation. WS-Security achieved by far the slowest times and has been ~1000 times slower than RMI (~1100 on basic types, ~957 on strings) but ~25% faster by instantiation. Compared to web services WS-Security has been ~109 times slower on basic types, ~104 times on strings, the instantiation times have been almost identical. To investigate the reasons for such slowdown we have repeated the measurements using three different JCE security providers: Bouncy Castle Crypto version 1.24, Wedgetail JCSI Provider version 2.3, and OpenSource HBCI Toolkit for Java version 0.0.6. The performance differences between the JCE providers have been minimal and not responsible for the slowdown. JCSI and HBCI have provided almost identical performance, while Bouncy Castle has been slower by ~15% using simple types and strings and ~4% slower by instantiation.

The best alternative for using RMI through firewalls is RMI-SSL. The second option is web services which are considerably faster than HTTP-to-port or HTTP-to-servlet tunneling. Web services with document/literal representation perform marginally better than those with RPC/encoded representation. WS-Security as supported by the products used in this test shows poor performance and is only of limited use in real-world scenarios. It should be used only where security has the highest priority over performance.

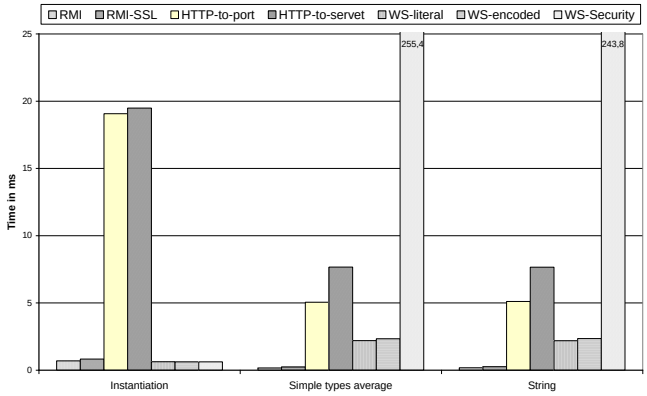

Figure 3: Performance results without network overhead

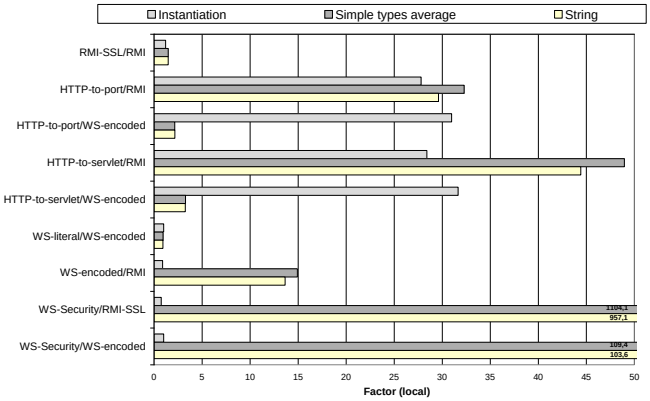

Figure 4: Local performance factors

Networked Performance Results

We have repeated the measurements on two network-connected computers. The measurements in this scenario include network overhead. From Figure 5 we can see that in the remote network scenario RMI and RMI-SSL still perform much faster than the other alternatives. HTTP-to-servlet tunneling however is considerably faster than HTTP-to-port. HTTP-to-port works almost 80% slower when used over a network, because local optimizations cannot be applied.

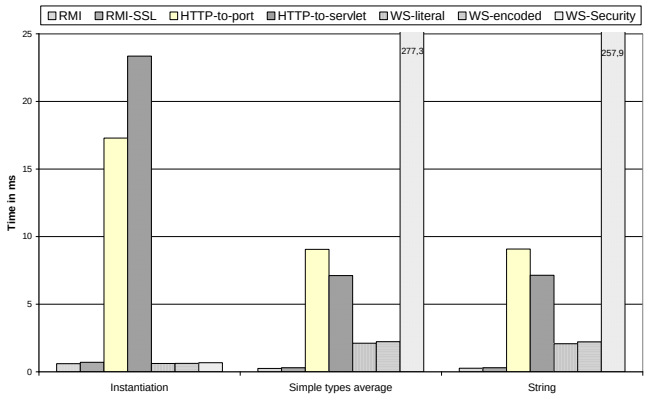

Figure 5: Performance results with network overhead

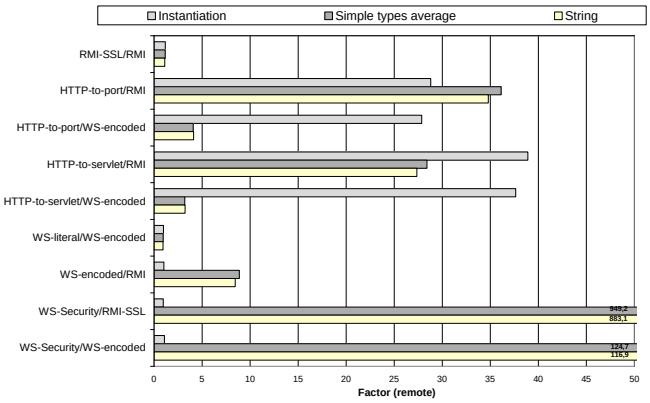

Figure 6: Remote performance factors

Figure 6 shows that RMI-SSL in remote scenario is only ~17% slower for basic data types, ~16% for strings, and ~16% by instantiation than RMI. This makes RMI-SSL the best alternative for using RMI over firewalls. The second fastest is web services with document/literal encoding being slightly faster than RPC/encoded. When used over the network, web services are ~8.8 times slower for average basic data types and ~8.5 times slower for strings 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 third 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. The huge

difference between WS-Security and web services still exists and is even larger than in local scenario. In remote scenario WS-Security is ~124 times slower using simple types, and ~116 times slower using strings. By the instantiation times WS-Security is only ~3% slower. WS-Security, compared to RMI-SSL, is ~900 times slower. We have again measured the difference between the three JCE providers, but have figured out that they performed almost identical.

The remote network scenario has considerable impact on performance of RMI, RMI-SSL, and HTTP-to-port tunneling (Figure 7). For RMI it adds on average ~55%, for RMI-SSL ~20%, and for HTTP-to-port ~78%. WS-Security is in remote scenario also slower than in local but only for ~8%. HTTP-to-servlet tunneling and web services show a performance improvement when used remote. The performance improved for ~7% and ~5% respectively. The reason for such results are local optimizations and load distribution. The scenarios without a web server show degradation in remote scenarios. These scenarios have performed local optimizations. Web server based scenarios have not and their CPU utilization on a single computer has been high. Therefore the additional network overhead is lower than the improvement of using two CPUs. By the WS-Security however the large message size which is transferred over the network is responsible for a slight slowdown in remote scenario. Comparing the results to our previous work [11] where average times for RMI increased ~37% in the network scenario compared to local, an increase of ~55% as we have measured it now is a step back. This is because processor performance increased faster than network throughput.

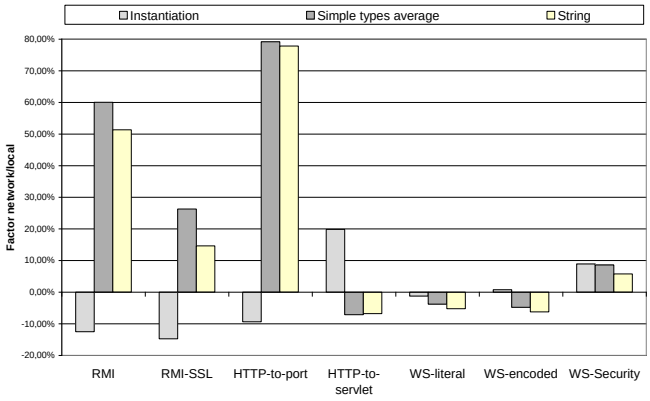

Figure 7: Comparison of remote vs. local performance results

Discussion

The measurements show that the distributed architectures we have compared differ significantly in performance. For using RMI over firewalls the best alternative is RMI-SSL. In addition to the ability to transverse firewalls seamlessly without the need to open additional ports RMI-SSL also provides authentication and communication security. It is an order of magnitude faster than HTTP-to-port and HTTP-to-servlet tunneling that also provides connectivity through firewalls, but not security and authentication. Web services perform better than HTTP-to-port and HTTP-to-servlet tunneling because

they avoid the message forwarding and repacking. Web services require less administrative work on the web server to set up. RMI tunneling alternatives limit the capability of RMI as they prevent callbacks. Therefore web services are usually a better choice compared to the tunneling alternatives. Web services also provide interoperability with other platforms. Secure web services (WS-Security) on the other hand have shown the worst performance in our tests, but we believe that that the performance will improve over time. Until then WS-Security should be used only where security has the highest priority over performance and other approaches such as RMI-SSL cannot be used.

The analysis of the performance results has shown that the performance differences between RMI, RMI tunneling, and web services are mainly related to the message creation, marshalling/demarshalling, dispatching, and demultiplexing. RMI and RMI-SSL use binary messages and the JRMP binary protocol (Java binary serialization is used underneath for call marshalling and returning the data). RMI tunneling uses MIME encoding to pack JRMP messages into HTTP and vice versa (in addition to binary serialization). Web services and WS-Security use SOAP protocol and provide the choice of literal or encoded data representation with the added metadata (start and end tag names, used to denote elements; attributes containing the element XML Schema types in encoded representation; and XML-related headers). Web services make use of XML serialization. In our previous research related to serialization we have shown that XML serialization is an order of magnitude less efficient than binary serialization [25]. This research confirms these results. Additionally it shows that literal encoding produces marginally smaller SOAP messages because xsi:type attribute is not used by each element; rather the type information is specified in the corresponding WSDL types section. Literal representation also shows slightly better performance in processing of messages. RMI JRMP messages are considerably smaller in size compared to SOAP messages. When transferred over HTTP additional header information is added, which further increases the SOAP message size.

In WS-Security scenarios the Triple-DES encryption algorithm has influenced message size because textual encoding is used. Generally Triple-DES does not increase the message size, as has been the case with RMI-SSL where the size of the messages in our tests has been almost identical to the unsecured RMI. In RMI-SSL and WS-Security scenarios we have observed initial overhead because of the authentication (related to verification of digital signatures) and the overhead in message processing (related to encryption/decryption). WS-Security message-level security requires that the binary security token, token reference, signature cipher data, and encrypted chipper data are included with each message [26]. WS-Security also time-stamps each message. To digitally sign data WS-Security uses XML Digital Signature [27]; for encryption it uses XML Encryption [28]. The encrypted data is base64-encoded to retain the textual format of SOAP messages which additionally increases the size of messages. WS-Security messages in our tests have been on average ~6.9 times larger than

unsecured SOAP messages and have not differed considerably between different data types. While WS-Security message-level security provides additional flexibility such as the possibility to encrypt parts of the message with different keys (this is currently not supported in JWSDP – Java Web Services Development Pack) it also places a large performance overhead compared to RMI-SSL.

Design and Implementation Guidelines

Based on the qualitative and quantitative analysis and comparison in this research and on previous research we have identified following general design and implementation guidelines related to performance of distributed solutions:

- Design and implementation of remote methods/operations and proper selection of synchronous/asynchronous and one-way and two-way operations
- Number of remote invocations and the amount of data transferred
- Control of serialization and code deployment in advance
- Use of distributed facades and collocation of objects/services
- Local execution of input validation logic
- Proper design of callbacks
- Use of caching and proper state and thread management

We have also identified the following web services specific design and implementation guidelines:

- Use of state, message correlation and idempotent operations in web services
- Efficient processing of XML content and efficient processing of web services operations
- Use of proper SOAP data representation
- Use of simplified XML schemas
- Limited use of WS-Security

Distributed application performance can be improved if the number of remote invocations is minimized, which can be achieved using coarse-grained interfaces. Operations of such interfaces increase the parameter size, therefore we should use complex types (web services) and value objects (RMI). Based on the fact that XML parsing is an important overhead factor we should design complex XML types with short tag names and use unaligned XML. Additional slight performance improvements can be achieved using document/literal instead of RPC/encoded web services. In our measurements document/literal web services produce slightly smaller SOAP messages and show slightly faster processing. Attention should be paid to the XML parsing techniques where SAX often shows performance benefits compared to DOM [30]. The

selection of the JAXP engine implementation also influences performance [30]. If we have several requests/responses sent to the same service we should use persistent connections.

When using value objects with RMI and RMI-SSL they should be designed with only the necessary set of attributes and relations. Particularly relations heavily influence the serialization performance as by default the whole graph of related objects gets serialized [29]. This behavior can be controlled by marking attributes as transient. Such attributes will not be included in serialization, which is particularly useful for derived attributes. Another approach to control serialization is to write custom methods for reading and writing the state of the objects, which can for complex objects be more efficient than default serialization. Instead of using dynamic class downloading to transfer the object behavior (code) to the remote computer, preinstalling the necessary classes further improves the performance.

If distributed objects/services are deployed on the same server we can move the coordination logic from the client to the server and further reduce the number of remote invocations. This can be achieved with the façade pattern [31], which contains the control logic and invokes related objects/services locally. To gain the benefit of local invocation we can relay on the optimizations provided by the distributed architectures; or we can design objects/services with local interfaces. Our measurements have shown that only RMI and RMI-SSL provide local optimizations, which are less efficient than local interfaces. Web services in our tests have not provided any optimizations. Relying on optimizations is thus in most cases not a good choice. Distributed facades should therefore be carefully designed and the decisions about objects/services with local interfaces should be taken relatively early in the design phase.

When using web services over HTTP requests can arrive out of order, can be lost, or can arrive more than once. Therefore web services operations should be designed as idempotent. Message correlation should be minimized or even avoided. RMI distributed objects and web services differ in the state management. Web services are stateless services, while RMI objects can be stateless or stateful. Stateless objects/services require that unique ids are exchanged by each operation invocation to identify clients. Compact ids improve the performance. Storing the state on the server should be designed and implemented carefully as inefficient state management will result in scalability bottlenecks on the server [11]. Common pitfalls include use of stateful operations, use of state management based on cookies, and inappropriate storing of state information.

Selection between synchronous and asynchronous, and one-way and two-way operations influences the performance. Web services support one-way and two-way, synchronous and asynchronous operations. RMI supports two-way synchronous operations only; support for asynchronous and one-way invocation can be achieved using threads. One-way operations can

improve the performance slightly but this is a tradeoff with reliability and is hardly a choice. Selection between synchronous and asynchronous operations should depend on the type of processing and requirements related to the response. Operations should be modeled as asynchronous if they are (1) long-lasting or (2) response to the caller is not required. Often asynchronous operations are combined into callbacks through which results are submitted back to the caller. RMI tunneling through firewalls or proxies limits or even prevents callbacks, which should be considered in the design phase. Callbacks in web services require that clients expose ports (endpoint references), which requires either a web server or a HTTP stack on the client.

Caching can improve the performance of distributed applications, but requires additional design and implementation effort. Caching should be considered when accessing remote objects/services and is usually achieved through stubs. The type and amount of data being cached should be selected carefully. Instead of using pregenerated stubs we can develop smart stubs which will be able to cache remote invocations and thus speed up the performance. Implementation is particularly simple with read-only operations. We have described how to develop smart stubs in Reference [29]. Caching is related to another important performance factor, input validation. If input validation is done on the middle-tier this requires several remote method invocations, which reduce the responsiveness of the user interface. To move the input validation to the client, RMI value objects or XML complex types can be used. Value objects can contain input validation methods and prefetched and/or cached data form the persistence tier. In the later approach, the necessary synchronization mechanisms have to be introduced. XML complex types used with web services can contain prefetched and/or cached data form the persistence tier but require that input validation code is deployed to the client-side in advance.

Distributed objects/services may be serving multiple simultaneous clients. In Java RMI and JWSDP simultaneous clients are served in multiple threads [10]. Therefore it is absolutely necessary to design and implement services thread-safe [32]. This includes synchronization on access to shared resources. In Java we can make use of synchronized blocks and volatile attributes, but should use these constructs carefully [33]. We should synchronize only the necessary parts of the code and prevent deadlock scenarios. We should select efficient thread management, such as leader/follower thread pooling [6].

Distributed Middleware Optimizations

Distributed middleware influences the performance of distributed applications. To identify the middleware packages and methods with highest execution time we have profiled the test scenarios. We have calculated the relative share of the methods/packages and made a comparison. The results are shown in Table 3. We have used Borland OptimizeIt Enterprise Suite 6 profiler and Enerjy Performance Profiler.

Package / Method (in %)	*RMI*	*RMI-SSL*	*HTTP-to-port*	*HTTP-to-servlet*	*WS-literal*	*WS-encoded*	*WS-Security*
java.net.socketOutputStream.socketWrite	28,77	31,41	36,96	67,24	15,98	17,41	38,76
sun.rmi.transport	23,41	23,65	<0,10	<0,10	0,00	0,00	0,00
com.sun.xml.rpc.sp	0,00	0,00	0,00	0,00	14,17	15,02	21,23
com.sun.xml.rpc.encoding	0,00	0,00	0,00	0,00	12,33	12,22	15,10
java.net.PlainSocketImpl.doConnect	<0,10	<0,10	18,15	0,78	<0,10	<0,10	<0,10
sun.net.www	<0,10	<0,10	14,82	6,74	4,91	5,21	8,16
java.io.ObjectInputStream	5,71	6,43	<0,10	<0,10	<0,10	<0,10	<0,10
sun.rmi.server	4,02	4,15	<0,10	<0,10	0,00	0,00	0,00
java.io.ObjectOutputStream	6,10	6,05	<0,10	0,97	<0,10	<0,10	<0,10
java.lang.SecurityManager	<0,10	1,52	2,11	<0,10	0,00	0,00	0,00
java.net.PlainSocketImpl	<0,10	<0,10	2,58	1,12	<0,10	<0,10	<0,10
sun.nio.cs	<0,10	<0,10	0,94	<0,10	1,14	1,12	0,85
sun.security.pkcs	0,00	2,33	0,00	0,00	0,00	0,00	1,34

Table 3: Comparison of relative time consumption of packages/methods

From Table 3 we can see that in all scenarios a considerable amount of time has been spent in message creation, marshalling and demarshalling. RMI, which uses binary serialization (java.io.Object*Stream), shows less overhead than web services, which use XML serialization (com.sun.xml.rpc). Web services using document/literal representation spend slightly less time in processing of SOAP messages than RPC/encoded. RMI tunneling has the disadvantage that it first uses binary serialization to produce JRMP messages and then those messages are encapsulated into HTTP requests, which results in significant overhead. Binary serialization produces smaller messages and requires fewer resources for their parsing and demarshalling than XML serialization. The latter deficits are due to more string parsing, copying, comparisons, and data type casts. An interesting observation from our measurements is that differences in message size are not directly related to differences in speed. Web services have been ~9 times slower than RMI, although they use ~4.3 times larger messages than RMI. RMI tunneling has shown considerably slower performance than web services. There are two reasons for that: the major reason is the additional cost of marshalling and demarshalling related to creation of HTTP POST requests with encoded JRMP messages. The second reason is the additional communication related to fallback and routing/forwarding of requests. In our code profiling for HTTP-to-port this is expressed as ~57% of time spent in java.net and ~15% of time spent in sun.net packages. For HTTP-to-servlet ~69% of time is spent in java.net package and ~7% in sun.net.

Instantiation is usually not a major overhead factor because applications usually obtain connections only at the beginning. Distributed middleware in our measurements shows deficits at instantiation. The overhead of RMI-SSL compared to RMI is

related to the authentication, done in sun.security package. RMI tunneling shows ~40% slower instantiation than RMI, which is related to additional communication (as identified above). Web services perform instantiation and operation invocation combined. Therefore we can observe considerable amount of time spent in com.sun.xml.rpc package. The additional overhead of WS-Security is explained by message authentication done by every operation invocation and response, which places considerable additional overhead compared to RMI-SSL.

The distributed middleware implementations used in this research show additional bottlenecks and implementation techniques which are not as efficient as possible. Therefore we have identified the major optimizations which can improve the performance of middleware and thus influence the performance of distributed applications on Java platform:

- Optimizations in JWSDP related to WS-Security include optimized acquisition of keys, local in-memory storage of the keys for the session duration, result buffering, and memory related optimizations particularly the reuse of objects (improved memory management without the excessive use of garbage collector).
- Use of fast, delayered and flexible marshaling algorithms reduces the marshaling/demarshaling overhead of SOAP messages. Instead of the layered marshaling/demarshaling a perfect hashing and active demarshaling can be implemented for best performance. These techniques can also be applied to RMI.
- Parser optimizations, use of pre-generated methods for obtaining the state of the objects and reduction of generated SOAP messages can reduce the SOAP overhead related to marshaling.
- Static method calls instead of reflection can provide optimizations of stubs and skeletons.
- Introduction of instance and connection pooling, and management algorithms together with pre-instantiation and reuse of physical entities can improve the instantiation overhead and reduce local garbage collection overhead.
- 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, and elimination of the run-time checking for the debugging code anr the proposed implementation related optimizations.

Conclusions

The qualitative and quantitative analysis of Java distributed architectures done in this paper has included three groups: Java native distributed architecture RMI and secure RMI-SSL, RMI tunneling techniques HTTP-to-port and HTTP-to-CGI/servlet, and web services and WS-Security. The comparison has revealed significant differences between the three groups and switching between them usually required code and design changes, in case of RMI tunneling also administrative

related to deployment and configuration of corresponding tunneling components. Web services and RMI differ considerably in functionality. Web services support synchronous and asynchronous operations, provide interoperability with other platforms and environments, and are better suitable for dynamic service binding and communication through firewalls. RMI supports synchronous remote method invocations only, supports stateful objects, object references, dynamic class downloading, distributed garbage collection, and remote object activation. RMI-SSL offers point-to-point security using SSL/TLS, WS-Security offers message-level security. WS-Security has also been a relatively new technology at the time of writing. We have compared these approaches and provided a comparison table to help make the most appropriate selection.

The performance analysis has revealed significant performance differences. RMI has shown the best performance in all tests. Second alternative has been RMI-SSL, which has been on average ~15% slower than RMI. Web services, the third alternative, have been on average ~8.5 times slower than RMI. Document/literal web services have performed ~6% faster than RPC/encoded. HTTP-to-port and HTTP-to-servlet tunneling have been slower than web services. In networked scenario HTTP-to-servlet has been more than 3 times slower and HTTP-to-port more than 4 slower times than web services. RMI tunneling alternatives offer poor performance compared to RMI therefore RMI-SSL or web services should be used for performance sensitive distributed applications over firewalls. WS-Security has shown poor performance being on average more than 100 times slower than web services. The results have shown that the support for WS-Security in JWSDP version 1.4 is unoptimized and not suitable for production use. The reasons for slower performance of web services and RMI tunneling compared to RMI are related to the message sizes transferred over the network, processing overhead of the messages, and implementation related overheads. We have identified general and web services specific design and implementation guidelines which influence the performance of distributed applications. To identify the sources of overhead in distributed middleware we have profiled the code and identified the following groups for optimizations: optimizations related to WS-Security, marshaling/demarshaling optimizations, SOAP-related optimizations, stub/skeleton optimizations, optimizations related to instantiation, and implementation related optimizations.

References

1. Kono, K, Masuda T. Efficient RMI: Dynamic Specialization of Object Serialization. *The 20th International Conference on Distributed Computing Systems (ICDCS 2000),* April 2000. IEEE Computer Society Press, 2000; 308-316
2. Nester, C, Philippsen, M, Haumacher, B. A more efficient RMI for Java. *Proceedings of the ACM 1999 Conference on Java Grande*, 1999. ACM Press: New York, 1999; 152-159
3. Zeadally, S, Zahang L, Zhu, Z, Lu, J. Network application programming interfaces (APIs) performance on commodity operating systems. *Information and Software Technology Journal*, 2004; 46: 397-402

4. Davis, D, and Parashar, M. Latency Performance of SOAP Implementations. *IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2002, IEEE Computer Society Press, 2002; 407-412
5. Gokhale, S A, Schmidt, D C. The Performance of the CORBA Dynamic Invocation Interface and Dynamic Skeleton Interface over High-Speed ATM Networks. *IEEE GLOBECOM '96 Conference*, November, 1996. IEEE Computer Society Press, 1996; 401-409
6. Gokhale, S A, Schmidt, D C. Measuring and Optimizing CORBA Latency and Scalability Over High-Speed Networks. *IEEE Transactions on Computers*, 1998; 47: 391-413
7. Plasil, F, Tuma, P, Buble A. CORBA Benchmarking. *Tech. Report No. 98/7*. Dep. of SW Engineering, Charles University, Prague, 1998
8. Gill, D C, Levine, D, Schmidt, D C. The Design and Performance of a Real-Time CORBA Scheduling Service. *Real-Time Systems*, 2001; 20: 117-154
9. Singhai, A, Sane, A, Campbell, R. Reflective ORBs: Supporting Robust, Time-critical Distribution. *ECOOP'97 Workshop Proceedings*, 1997. LNCS Springer Verlag, 1997: 149-155
10. Juric, B M, Domajnko, T, Zivkovic, A, Hericko, M, Brumen, B, Welzer, T, Rozman, I. Performance Assessment Framework for Distributed Object Architectures. *Conference proceedings of ADBIS'99*, September 1999. LNCS Springer Verlag, 1999: 349-366
11. Juric, B M, Herciko, M, Rozman I. Performance Comparison of CORBA and RMI. *Information and Software Technology Journal*, 2000; 42: 915-933
12. Juric, B M, Zivkovic, A, Rozman, I. Are Distributed Objects Fast Enough. *Mora Java Gems.* Cambridge University Press, 2000; 391-403
13. Juric, B M, Rozman, I, Nash, S. Java 2 Distributed Object Middleware Performance Analysis and Optimization. *ACM SIGPLAN Notices*, 2000; 35/8: 31-40
14. Juric, B M, Kezmah, B, Hericko, M, Rozman, I, Vezocnik, I. Java RMI, RMI tunneling and Web services comparison and performance analysis. *ACM SIGPLAN Notices*, 2004; 39/5: 58-65
15. Juric, B M, Rozman, I, Brumen, B, Colnaric, M, Hericko, M. Comparison of Performance of Web services, WS-Security, RMI, and RMI-SSL. *Journal of Systems and Software*, accepted for publication, available on-line, 2005
16. World Wide Consortium. *Simple Object Access Protocol, SOAP*. W3C Recommendation, 2003. http://www.w3.org/TR/soap/
17. World Wide Consortium. *Web Services Description Language, WSDL*. W3C Note, 2001. http://www.w3.org/TR/wsdl
18. Erl, T., *Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services*, Prentice Hall, 2004
19. Sun Microsystems Inc. *Java Remote Method Invocation Specification*, Revision 1.8, Sun Microsystems, 2002
20. Sun Microsystems Inc. *Introduction to SSL*. http://docs.sun.com/source/816-6156-10/contents.htm
21. Juric, M B, Basha, S J, Leander, R, Nagappan, R. *Professional J2EE EAI*, Wrox Press, Birmingham, 2001
22. Comer, D E. *Internetworking with TCP/IP Vol.1: Principles, Protocols, and Architecture*, 4th Edition, Prentice Hall, 2000
23. Garfinkel, S. Firewall Follies. *Technology Review*, 2002; 33: 22-26
24. Pitt, E, Belford, N. *The RMI Proxy White Paper*, July 2002. http://www.rmiproxy.com/doc/WhitePaper.pdf
25. Hericko, M, Juric, B, M, Rozman, I, Beloglavec, S, Zivkovic, A. Object Serialization Analysis and Comparison in Java and .NET. *ACM SIGPLAN Notices*, 2003; 38/8: 44-54
26. *Web Services Security (WS-Security)*, 2002. http://www-106.ibm.com/developerworks/webservices/library/ws-secure/

27. World Wide Consortium. *XML Signature Syntax and Processing*. W3C Recommendation, 2002. http://www.w3.org/TR/xmldsig-core/
28. World Wide Consortium. *XML Encryption Syntax and Processing*. W3C Recommendation, 2002. http://www.w3.org/TR/xmlenc-core/
29. Adatia, R, Juric, M B, et al. *Professional EJB*, Wrox Press, Birmingham, 2001
30. *XML Parsing Benchmark.* http://www.devsphere.com/xml/benchmark/index.html
31. Gamma, E, Helm, R, Johnson, R, Vlissides, R. *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, 1995
32. Comer, D E, Stevens, D L, Evangelista M. *Internetworking with TCP/IP, Vol. III: Client-Server Programming and Applications, Linux/Posix Sockets Version*, Prentice Hall, 2000
33. Wellings, A. *Concurrent and Real-Time Programming in Java*, John Wiley & Sons, 2004