Performance Enhancing Proxies for Java™ 2 RMI over Slow Wireless Links*

Stefano Campadello[†], *Heikki Helin*[‡], *Oskari Koskimies*[†], *Kimmo Raatikainen*[†]

[†]Department of Computer Science, University of Helsinki

P.O.Box 26 (Teollisuuskatu 23), FIN-00014 UNIVERSITY OF HELSINKI, Finland

[‡]Sonera Ltd.

P.O.Box 970 (Teollisuuskatu 13), FIN-00051 SONERA, Finland

Email: {Stefano.Campadello, Oskari.Koskimies, Kimmo.Raatikainen}@cs.Helsinki.FI, heikki.j.helin@sonera.com

Abstract

Due to its high protocol overhead, both in data traffic and in round-trips, Java RMI is poorly suited for wireless communication. However, it can be optimized without breaking compatibility with Java RMI specification, and with minimal changes to existing software in the network hosts. This paper analyzes the reasons for the poor performance of Java RMI and outlines a solution based on mediator technology. A prototype has been implemented, which improves performance by a factor of three.

1 Introduction

Remote Method Invocation (RMI) (Sun, 1998) is the object-oriented version of the well-known Remote Procedure Call (Birrel and Nelson, 1984). It is essentially the same concept that allows the programmer to transparently invoke methods on objects that reside on another computer. In this way, the object-oriented paradigm is preserved in distributed computing. There are several implementation architectures for the remote method invocation. The most well-known one is CORBA (OMG, 1995), which has several commercial implementations available.

However, with the success of the Java (Arnold and Gosling, 1996) language, Java RMI is getting more and more attention. Unlike CORBA, Java RMI only works between Java programs. On the other hand, Java RMI is far more flexible than CORBA. Jini Connection Technology (Arnold et al, 1999) and most Agent Platforms use Java RMI as a communication framework.

* This work was supported by Nokia, Sonera Ltd., and the National Technology Agency of Finland (TEKES).

Internet offers an ever-increasing amount of information services. People have become increasingly dependent on access to those services. In fact, some people need them even when they do not have network access. By using wireless networks, people can utilize network information services even when they are on the move. However, current communication services do not work well in a mobile environment, because of the different characteristics of fixed and wireless networks. Applications that were designed for fixed, reliable networks often perform poorly in a mobile environment. The main problems include unreliability, low bandwidth, high latency and high cost of wireless communication when compared to communication in the fixed network. These problems can be solved by using light-weight, mobile-aware protocols and applications in the wireless environment.

Given that Java RMI is gaining popularity, it is only a matter of time before its performance over wireless links becomes important. We have analyzed some performance characteristics of Java RMI over GSM—protocol overhead and round-trips, in particular. GSM (Mouly et al, 1992 and Rahnema, 1993) is pan-European digital cellular telephone network. In addition to voice services, it defines a number of data services that can be used for data transmission between computers.

Our conclusion is that Java RMI works poorly in a slow wireless environment. This is due both to the poor performance of TCP in a wireless environment and to the Java RMI protocol itself. In this paper we propose a mediator-based solution to overcome those problems.

Mediators—known as *performance enhancing proxies* in IETF (Border et al, 1999)—are widely used to improve TCP/IP performance on communication paths including a slow wireless link. Examples include Indirect TCP by Rutgers University (Bakre and Badrinath, 1995 and 1996), Snoop by University of California at Berkeley (Balakrishnan et al, 1995 and 1996), Mowgli by University of Helsinki (Kojo et al, 1994 and 1997), and TACO by University of Copenhagen (Hansen et al, 1998). Mediators are also used to improve application level protocols like HTTP (Liljeberg et al, 1995 and Padmanabhan and Mogul, 1995). In addition, the WAP architecture (WAP, 1999) is also based on mediators.

In a mediator-based solution there are *access nodes* in the fixed network that serve the mobile terminals by providing them with an access point to the fixed network. Each application, which is not mobile aware, requires two mediators:

1. An *agent*, which is located in the mobile terminal, is bound with the terminal part of the application (usually a client).
2. A *proxy*, which is located at the access node, is bound to the fixed network part of the application (usually a server).

All communication between the terminal and the fixed network part of the application goes via the agent and proxy, which collaborate to optimize the communication over the wireless link.

The rest of this paper is organized as follows. In Section 2 we analyze the RMI wire protocol. In Section 3 we give the main points of optimizing RMI in a wireless environment and in Section 4 we outline our solution. In Section 5 we compare normal RMI and our optimized version and give the preliminary results. Finally, Section 6 states the conclusions.

2 Investigating the RMI Implementation

2.1 Java RMI Protocol

Java RMI was designed to simplify the communication between two objects in different virtual machines by allowing transparent calls to methods in remote virtual machines. Figure 1 depicts the protocol stack in the Java RMI communication.

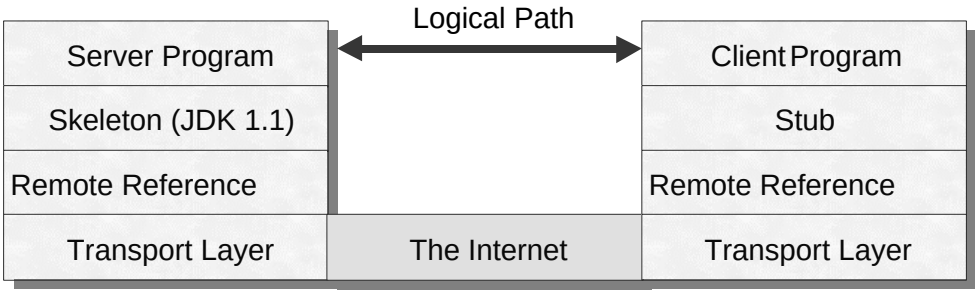

Figure 1: Java RMI Layers

Once a reference of a remote object is obtained, it is possible to call methods of that object in the same way as methods of local objects. Since the remote object resides in a different virtual machine, an RMI Registry is needed to manage remote references. When an RMI server wants to make its local methods available to remote objects, it registers the objects to a local Registry. A remote object connects to the remote Registry, which listens a well-known socket, and obtains a remote reference.

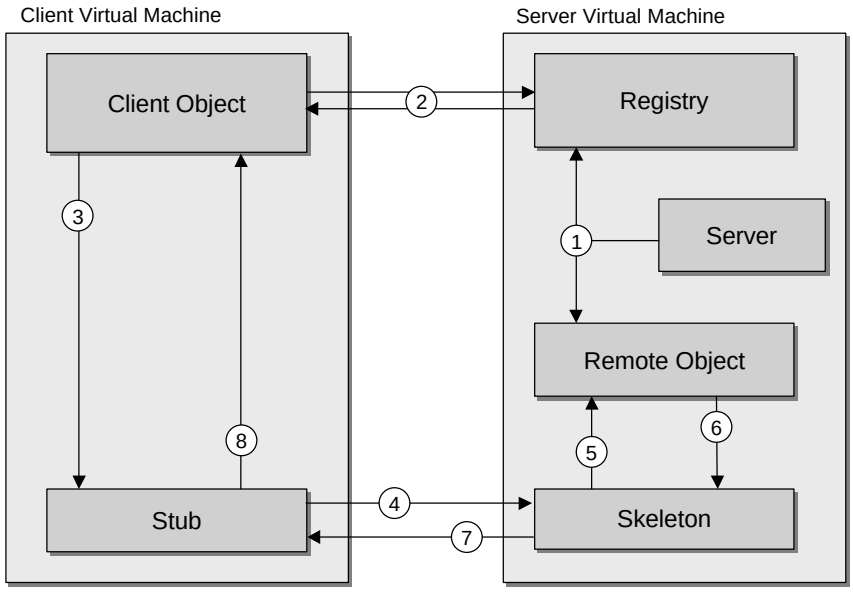

Figure 2: Java RMI Protocol

The general Java RMI architecture (Sun, 1998) is depicted in Figure 2. First a server creates a remote object and registers it to a local Registry (1). The client then connects to the remote Registry (2) and obtains the remote reference. At this point, a stub of the remote object is transferred from the remote virtual machine to the client virtual machine, if the stub is not yet present. When the client (3) invokes a method at a remote object, the method is actually

invoked at the local stub. The stub marshals the parameters and sends a message (4) to the associated skeleton on the server side. The skeleton unmarshals the parameters and invokes the appropriate method (5). The remote object executes the method and passes the return value back to the skeleton (6), which marshals it and sends a message to the associated stub on the client side (7). Finally the stub unmarshals the return value and passes it to the client (8).

2.2 RMI use of TCP Connections

Java RMI is built on top of a Transport Layer, which provides abstract RMI connections. When an RMI connection is opened, the transport layer either opens a new TCP connection, or reuses an existing one if a free one is available. If the reused connection has been idle for more than the time of a round-trip, the transport layer first sends a ping packet to make sure the connection is still working. Once an acknowledgment for the ping packet is received, the new RMI connection is established. If a TCP connection has not been used by any RMI connections for a while, it is closed.

Reusing TCP connections is commendable because it saves resources. However, the implementation causes frequent ping messages. This is problematic with high-round-trip wireless connections.

While Java RMI is not optimal for wireless networks, neither is TCP upon which Java RMI is built. The problems with TCP in a wireless environment are well-known (Kojo et al, 1997 and Cacerer et al, 1995), and the main ones are:

Slow-Start Algorithm – The slow-start congestion control algorithm is employed at the start of each new TCP connection, and whenever the sender believes a packet has been lost.

Retransmission Timers – The low initial round-trip timer value, which is feasible for fast networks, causes unnecessary retransmissions in wireless networks. Also, adjustment of the retransmission timeout is hampered by the round-trip time fluctuation due to link level retransmissions and buffering.

Interference of multiple TCP connections – Data buffered for one TCP connection causes unpredictable delays in round trip times for other connections, which again triggers retransmissions.

Especially in JDK1.1, RMI and TCP conspire to produce bad results. RMI writes header data byte by byte, and because of the slow start algorithm (Jacobson, 1988), TCP has to wait for an acknowledgment once it has sent the first segment containing only one byte. This means that it takes a full round-trip (which can be almost a second with e.g. GSM Data) before the second segment can be sent.

In JDK1.2 (Java™ 2), data is no longer written byte by byte, and performance is much better. However, the protocol itself still enforces many round-trips for a single invocation (due mainly to the ping packets).

1.3 Remote Call Message Sequence

In this section we will show an analysis of a simple RMI call. The method is called "sayHello". As a return value the method returns the String "*Hello World*".

In our test the stub class was already present on the client side, so there was no need to download the stub class. The trace of the call is outlined below (see Figure 3):

1. The first round-trip is between the client and the Registry on the remote side and uses a new TCP connection (*TCP1*). The registry returns an acknowledgment of the RMI protocol and what it believes to be the client's IP address. It should be noted that this first round-trip happens every time a new connection is opened.
2. In the second round-trip the client requests (and obtains) the remote reference of the desired remote class. At this point the TCP connection is logically closed.
3. Opening a second TCP connection *TCP2*, the client connects with the server. Since this is the first RMI call a header and a protocol acknowledgment are exchanged.
4. The client-side Distributed Garbage Collection (DGC) requests from the server a lease of the required remote reference through a *Dirty()* invocation. The TCP2 connection is logically closed.
5. At this point the Client must tell the DGC of the Registry that it obtained a remote reference. The first round-trip between the client and the Registry is a ping: in this way the client verifies that the TCP connection, that was logically closed before, is still alive[1]. Having verified this, the client communicates the Registry that it has received a lease from the server with a DGCAck message.[2]
6. In parallel with the previous point, the client can invoke the remote method on the server. But, since the TCP connection was closed, a ping round-trip takes place. After this, the client invokes the method and obtains the results of the invocation as return values.
7. When the client does not need the remote reference any more, usually when the remote reference is locally unreferenced, it sends a "clean" message to a the server. This exchange is preceded by the usual ping round-trip.

---

[1] Note that a ping does not occur if the TCP connection has been idle for a time less than a ping round-trip.

[2] Before this acknowledgment has been received, the Registry must not release its server reference, since that might cause the server to wrongly conclude that there are no references in use.

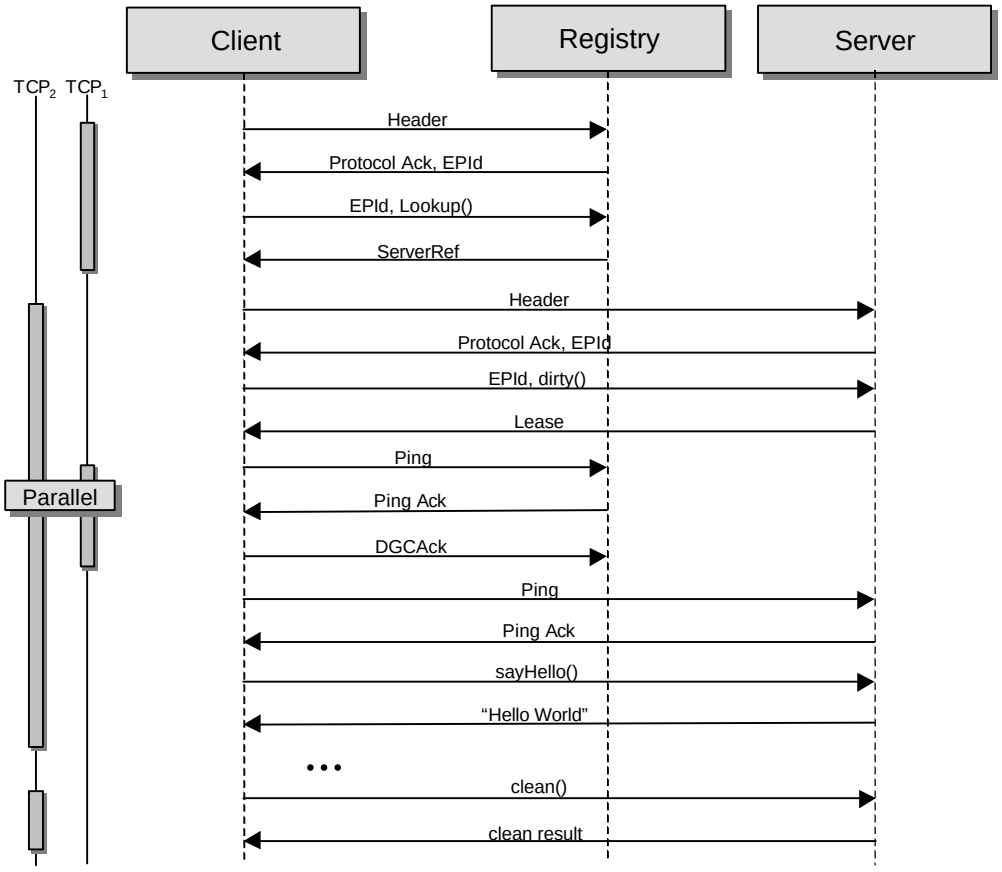

Figure 3: The trace of the "sayHello" remote invocation

Data traffic is summarized in Table 1. In each row is given the amount of data (and percentiles) transferred in each transfer pattern.

	Client to Server and Registry	Server and Registry to Client	Total
Registry Lookup	55 (6%)	276 (42%)	331 (20%)
Invocation Data	41 (4%)	37 (6%)	78 (5%)
DGC Data	831 (85%)	305 (46%)	1136 (69%)
Protocol Overhead	52 (5%)	40 (6%)	92 (6%)
Total	979 (100%)	658 (100%)	1637 (100%)

Table 1: Invocation data traffic (bytes)

On a slow wireless link the amount of data that it is sent over the link is important. In this example the actual invocation takes up only 5% of the total transmitted data while 69% was related to DGC protocol. This means that the channel is primarily used for auxiliary data, making the invocation expensive.

Another important issue is the high number of round-trips. On slow links, like GSM, even a single byte exchange like ping causes delays due to the long latency times involved (a round-trip over GSM is typically around one second). In this example six round-trips were necessary before the invocation was completed, not counting the two round-trips caused by TCP handshaking. However, only two are really needed—one to get the server reference, and another for the actual invocation.

3 Optimizations

Optimization of Java RMI for wireless links means reduction of protocol overhead and the number of round-trips. We have two ways to reach this goal: The first one is to change the implementation itself, i.e. to change JDK system classes. The disadvantage is that modification of both client and server host software is necessary, and that makes this solution unattractive. The second solution, presented here, attempts to preserve the original implementation supporting it with the use of mediators and compression to avoid, where possible, every redundant communication between the client side and the server side[3]. The solution uses a mediator architecture, with an RMI agent in the mobile terminal and an RMI proxy in the fixed network.

The proposed optimizations are outlined below. Note that "locally" means "in the terminal" for software residing on the terminal, and "in the Access Node" for software residing on the fixed network.

Data compression – The serialization protocol used by Java RMI produces a large amount of overhead in the invocation. Since we do not want to modify the source code, the easiest way to reduce this overhead is to compress the data using a generic compression algorithm. In our prototype we use the GZIP standard file format.

Protocol Acknowledgment – The acknowledgments can be handled by local mediators. Only in the (extremely rare) case that the protocol is unknown to the mediator, does the mediator actually forward the protocol header to the server.

Avoiding downloading of stubs – A client needs to obtain a stub from the server side. The stub is downloaded from the network, and, in case the classes needed by the stub are not present in the client side, they are downloaded through a web server. In our solution a generic stub is created on the fly on the client side. No stub download is needed.

Registry lookups – When a client does a lookup for the first time, the remote reference is cached locally, and the original reference in the Registry is marked as cached by the mediator on the other side. Mediators synchronize cached references by notifying their peer when a reference marked as cached changes. To reduce initial cache misses, the reference cache can be initialized with frequently used references at startup.

---

[3] Note that RMI is used symmetrically—a mobile terminal may have both RMI clients and servers. This symmetry must be preserved.

Distributed Garbage Collection – This protocol introduces heavy data overhead and its use in wireless environment is less meaningful, since the link is subject to sudden disconnections that can be handled at transport layer. To avoid its redundancy and high number of round-trips the client and the server can be decoupled. The use of mediators for this decoupling is explained in the next section.

4 Use of Mediators

The role of the RMI mediators is shown in Figure 4. The RMI Agent captures the invocation made by the client. A header preamble can be answered immediately, as stated before. A lookup request is first checked in the local cache, and only if the remote reference is unknown is the request forwarded to server.

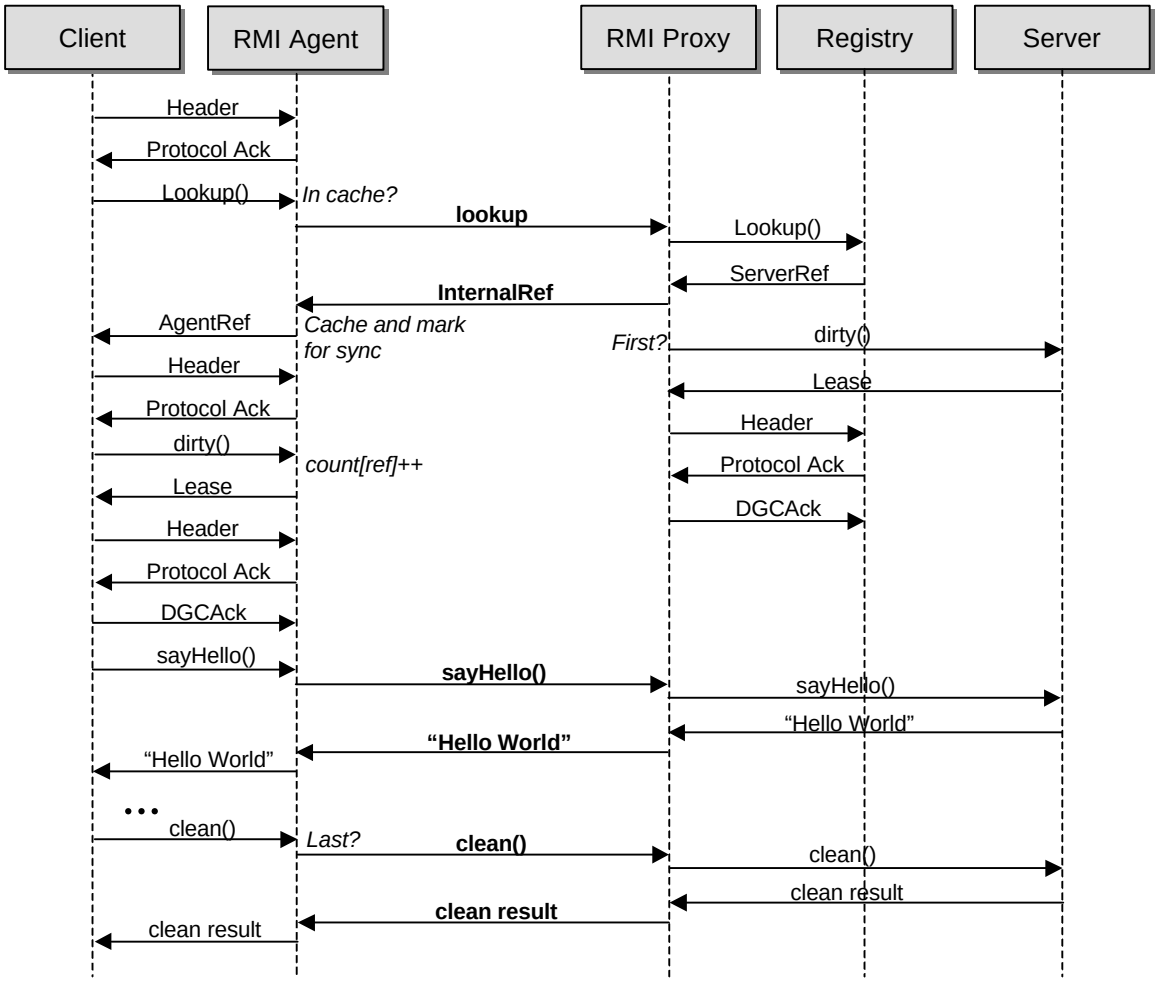

Figure 4: Using mediators to optimize the remote invocation

DGC invocations are optimized by decoupling client and server. The local mediator keeps servers alive by periodically renewing the leases[4]. It will only stop doing so once its peer mediator tells it that no more references to the server exist on the other side. Correspondingly, the mediator on the other side keeps track of clients that have references to this server, and of the time left on their leases. In this way the DGC semantics are loosened to suit the needs of wireless communication, without modifying client or server code. No lease requests (*dirty()* invocations) need to be sent over the wireless link, since all leases are managed locally. In this way the number of round-trips is minimized. The amount of data transferred over the wireless link is also reduced. Of course, clean requests still have to sent to inform the other side that no reference to a server exist anymore. An optimized data representation can be used for these requests, further reducing DGC overhead. The Figures 5 and 6 outline the difference between our RMI optimization and the normal RMI.

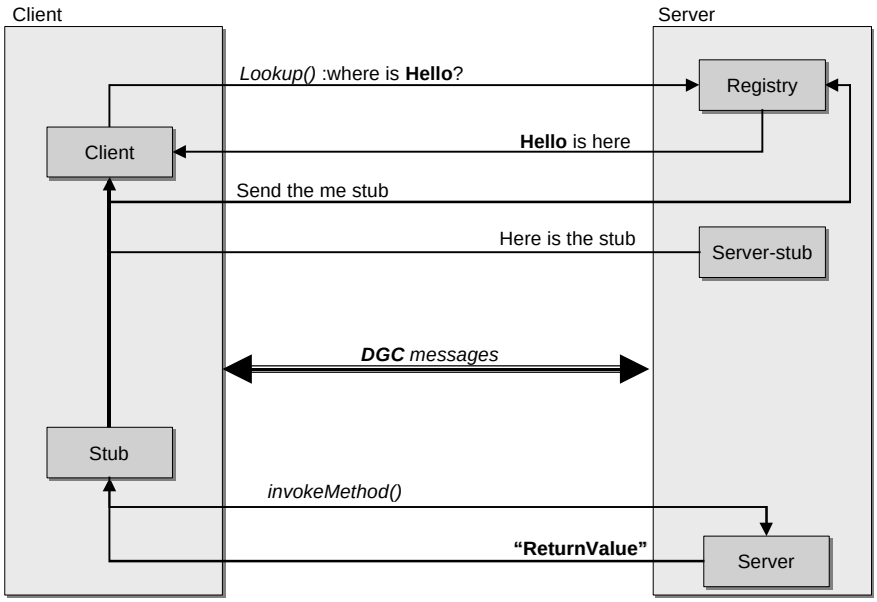

Figure 5: The normal RMI structure

[4] A lease is a time period after which the server will assume that the client has died unless the client renews the lease.

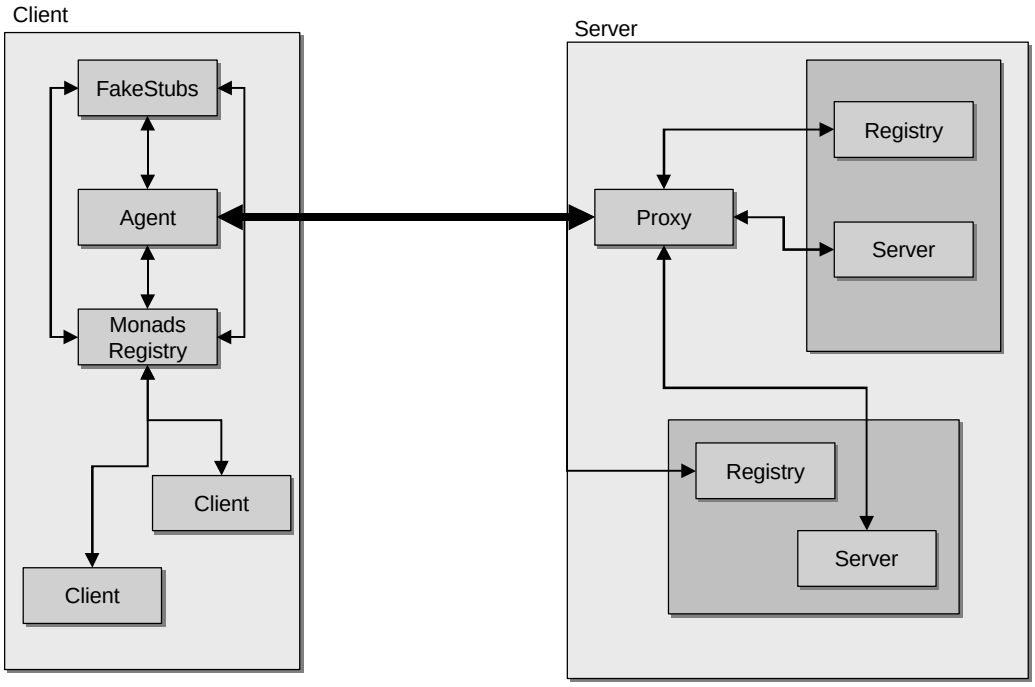

Figure 6 :Optimized RMI Structure

5 Comparison between Normal RMI and Optimized RMI

To give an idea of the numbers involved in Java RMI we built a prototype of an optimized RMI architecture and set up several tests using the HelloWorld example explained in Figure 4. The remote method is invoked through a GSM Data wireless link. The prototype is composed of two mediators exchanging uncompressed data and using a special protocol. Essentially it implements the concepts presented in the previous section, although no compression is used. The results of our tests are shown in Table 2.

	Registry Invocation	Remote Invocation	Total
Java RMI	7.1 sec	1.3 sec	8.4 sec
Optimized RMI	1.7 sec	0.6 sec	2.3 sec
Improvement	417%	216%	365%

Table 2: Comparison between normal RMI and optimized RMI for the HelloWorld case

While the HelloWorld example is simple to analyze and represents a common case, we also wanted to verify our prototype with a larger scenario. The authors are members of the Monads research project (Campadello et al, 2000), which examines the use of adaptive agents in mobile computing. Since agent platforms typically use Java RMI for remote communication, we introduced our optimization to the Monads prototype implementation. The Monads prototype simulates the movement of a user on a city map with varying Quality of Service conditions. The simulation continuously gives location and Quality of Service information to

the system, which then tries to achieve optimal performance through optimization, prediction and adaptation (Misikangas et al, 1999). One feature of the prototype is providing location-sensitive information to the user. For example, a restaurant might provide its menu as an advertisement to passing users. In this case, several RMI servers are distributed on the map, every one sending information when the client is arriving in the proximity of the server. The client acquires the reference and then sends an invocation to obtain the desired information.

We analyzed the performance of the RMI system in both configurations (with and without optimizations) using a GSM data wireless link. Since our optimizations introduce overhead in the client and on the access node side, we measured the time between client request and receiving the reference or the return value of the invocation. The results for this experiment are given in Table 3.

	Registry Invocation	Remote Invocations		
		3300 bytes	7048 bytes	14760 bytes
Java RMI	18.9 sec	8.1 sec	14.8 sec	41.4[5] sec
Optimized RMI	2.9 sec[6]	3.0 sec	4.8 sec	5.0 sec
Optimized RMI, no compression	3.0 sec[6]	7.8 sec	13.7 sec	28.1 sec

Table 3: Comparison between normal RMI and optimized RMI for the Monads prototype

In the first column is indicated the time spent by the client to obtain the remote references. During the demonstration three remote references were requested from three different servers. After this, the client made three different invocations, one to each server: the amount of data transferred from the servers to the client is indicated in the table. The given values are average between several experiments. The hardware and software configurations used in measurements are shown in Table 4.

[5] This particular value appears to be extremely high. We are conducting further experiments to discover the reason of such bad performance when Java RMI transfers a large amount of data through a GSM link.

[6] This value is the time necessary to get the reference from the remote Registry for the first time. Following lookups of the same reference require 0.4 seconds.

	Client	Server
Operating System	Linux 2.2.12	Linux 2.0.36
JDK	1.2.1	1.2.1
Hardware	Intel Pentium II/366	Intel Pentium II/400

Table 4: Hardware and Software used in Measurements

As can be seen, the results are extremely good. Tracing of RMI behavior during the test showed that during the time spent receiving the references, Java RMI opens multiple sockets while our optimized version opens only one. In the GSM environment, where the nominal transfer rate is 9.6 kbps and a roundtrip takes more than 500 ms, that means several seconds of overhead. Furthermore, several times the reference was found in the RMI Agent's cache, so no data exchange through the wireless link was needed. On the other hand, the improvement provided by our implementation during the invocations was due mostly to the compression factor. In fact, while serializing String classes, Java uses two bytes for every character, thus doubling the amount of data transmitted to the wireless link. We investigated the use of our system also without the use of compression, and still the results are encouraging, mostly because normal Java RMI closes idle sockets after a few seconds, and thus needs to re-open them when needed, adding more than a second of roundtrip delay.

In conclusion, although the JDK1.2 implementation of RMI improved performance considerably from the JDK1.1 version, it is still far from optimized for wireless links. With our prototype we decreased the time required for the initial acquisition of the remote reference to one fourth and halved the time required for the remote invocation.

6 Conclusions

Due to its high protocol overhead, both in data traffic and in round-trips, Java RMI is poorly suited for wireless communication. However, it can be optimized without breaking compatibility with Java RMI specification, and with minimal changes to existing software. New software is necessary only at the mobile terminal and at its access point to the fixed network. This is possible by utilizing mediator technology, which is widely exploited in wireless communications. The results are encouraging. We expect further enhancement by the optimization of the Serialization Protocol.

In the future, we are planning to extend our tests to other operating systems and different wireless networks.

7 References

Arnold, K and Gosling, J. *The Java Programming Language*. Addison-Wesley, 1996. See also http://java.sun.com/.

Arnold, K.; O'Sullivan, B.; Scheifler, R.; Waldo, J; Wollrath, A: *The Jini Specification*. Addison-Wesley, 1999. See also http://www.sun.com/jini/

Bakre, A. and Badrinath, B.R.: I-TCP: Indirect TCP for Mobile Hosts, in *Proc. of the 15th International Conference on Distributed Computing Systems* (Vancouver, Canada), pp. 136-143, June 1995.

Bakre, A. and Badrinath, B.R.: Indirect Transport Layer Protocols for Mobile Wireless Environment, in *Mobile Computing*, Kluwer, 1996, pp. 229-252.

Balakrishnan, H.; Seshan, S.; and Katz, R.: Improving Reliable Transport and Handoff Performance in Cellular Wireless Networks, *Wireless Networks* 1, 4 (Dec. 1995), pp. 469 - 481

Balakrishnan, H.; Padmanabhan, V.N.; Seshan, S.; and Katz, R.: A Comparison of Mechanisms for Improving TCP Performance over Wireless Links, in *Proc. ACM SIFCOMM'96* (Palo Alto, Calif.), Aug. 1996, pp. 256-269.

Birrel, A.D. and Nelson B.J.: Implementing Remote Procedure Calls, *ACM Transactions on Computer Systems* 2, 1 (Feb 1984), pp. 39-59.

Border, J.; Kojo, M.; Griner, J.; and Montenegro, G.: Performance Enhancing Proxies, Internet draft draft-ietf-pilc-pep-01.txt, December 3, 1999.

Cacerer, R. and Iftode, L.: Improving the Performance of Reliable Transport Protocols in Mobile Computing Environments, *IEEE J. Selected Areas of Communications*, V 13, N 5, (June 1995), pp. 850-857.

Campadello, S.; Helin, H.; Koskimies, O.; Misikangas, P.; Mäkelä, M.; and Raatikainen, K.: Using Mobile and Intelligent Agents to Support Nomadic User, *in the 6th International Conference on Intelligence in Networks (ICIN2000)*, 17-20 January 2000, Bordeaux, France, pp. 199-204.

Hansen, J.S.; Reich, T.; Andersen, B.; and Jul, E.: Dynamic Adaptation of Network Connections in Mobile Environments, *IEEE Internet Computing* 2, 1 (Jan/Feb 1998), pp. 39-48.

Jacobson, V.: Congestion Avoidance and Control, in *Proc. ACM SIGCOMM'88 Symposium on Communications Architectures and Protocols*, Stanford, California, August 1988, pp. 314-329.

Kojo, M.; Raatikainen, K.; and Alanko, T.: Connecting Mobile Workstations to the Internet over a Digital Cellular Telephone Network, in *Proc. of MOBIDATA Workshop* (Rutgers University, NJ), Nov. 1994. Updated version in Mobile Computing, Kluwer, 1996, pp. 253-270.

Kojo, M.; Raatikainen, K.; Liljeberg, M.; Kiiskinen, J.; and Alanko, T.: An Efficient Transport Service for Slow Wireless Telephone Links, *IEEE Journal on Selected Areas in Communications* 15, 7 (Sep. 1997), pp. 1337-1348.

Liljeberg, M.; Alanko, T.; Kojo, M.; Laamanen, H. and Raatikainen, K. Optimizing World-Wide Web for Weakly-Connected Mobile Workstations: An Indirect Approach, in *Proc. of the 2nd International Workshop on Services in Distributed and Networked Environments (SDNE)*, IEEE Computer Society Press, 1995, pp. 132-139.

Misikangas, P.; Mäkelä, M.; Raatikainen,K: Predicting QoS for Nomadic Applications Using Intelligent Agents, in *Proc. of the IMPACT99*, Seattle (WA) 2-3 December 1999.

Mouly M. and Pautet M.-B.: *The GSM System for Mobile Communications*. Mouly and Pautet, 1992.

Object Management Group: The Common Object Request Broker: Architecture and Specification, 1995. Available electronically from http://www.omg.org/corba2/.

Padmanabhan, V.N. and Mogul, J.C.: Improving HTTP Latency, *Computer Networks and ISDN Systems*, 28, 1&2 (Dec. 1995), pp. 25-35.

Rahnema, M.: Overview of the GSM System and Protocol Architecture, *IEEE Communications Magazine*, vol.31, no.4, pp.92-100, Apr.1993.

Sun Microsystems: Java Remote Method Invocation–Distributed Computing for Java. White Paper, March 1998.

WAP Forum: Wireless Application Environment Overview, version 1.2, November 1999. Available electronically from http://www.wapforum.org/.