Wireless Java RMI

Stefano Campadello, Oskari Koskimies,
Kimmo Raatikainen
Department of Computer Science, University of Helsinki
P.O. Box 26 (Teollisuuskatu 23)
FIN-00014 UNIVERSITY OF HELSINKI
Finland
[First name].[Last name]@cs.Helsinki.FI

Heikki Helin
Sonera Ltd.
P.O. Box 970 (Teollisuuskatu 13)
FIN-00051 SONERA
Finland
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 communication over slow wireless links. However, its performance can be enhanced without breaking compatibility with Java RMI specification, and with minimal changes to existing software and network hosts. This paper analyzes the reasons for the poor performance of Java RMI, outlines a solution based on mediator technology and evaluates the performance of a prototype implementing the solution.

1. Introduction

Remote Method Invocation (RMI) [20] is the object-oriented version of the well-known Remote Procedure Call [7]. RMI 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 [23], which has several commercial implementations available. However, with the success of the Java language, Java RMI is earning more and more attention. In contrast to CORBA that is programming language independent, Java RMI only works between Java programs. On the other hand, Java RMI is far more flexible than CORBA. Jini Connection Technology [2] and most Agent Platforms use Java RMI as a communication framework. The importance of Java RMI also increases due to the announcements of many mobile terminal device manufacturers that their equipment will support Java in the future.

Current and forthcoming wireless data services—GSM Data Service, GPRS, UMTS, wireless LAN—enrich the options for communications. Unfortunately, the current Internet-based solutions are not able to fulfill all the needs of nomadic users [21]. The environment of mobile computing is very different compared to today's environment of traditional distributed systems in many respects. Bandwidth, latency, delay, error rate and interference, among other things, may change dramatically as a nomadic end-user moves from one location to another—from an environment to another: from a wired LAN via a wireless LAN to a GPRS/UMTS network for example.

Today the Internet offers a communication backbone to an ever-increasing amount of information services. In fact some persons (not necessarily addicted but primarily due to their profession) have become increasingly dependent on access to those services. By using wireless networks, those persons 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 applications and middleware products, on which the applications run, were designed for fixed, reliable networks. The problems are due to manifold reasons; including sudden loss of connectivity, low bandwidth, high latency, and high cost of wireless communication. Many of the problems can be solved by using light-weight, mobile-aware protocols and middleware in the wireless environment.

Given that Java RMI is gaining popularity, it is only a matter of time before the performance of Java RMI over wireless links becomes important. We have analyzed and measured performance characteristics of Java RMI over GSM Data Service and High Speed Data Service [10]. We examined two platforms—Linux and Windows NT—and several variations of a commercially available GSM HSCSD. The results are not encouraging: Java RMI works poorly in slow wireless environments. This is due to the well-known poor performance of TCP in a wireless environment but also due 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 [8]—are widely used to improve TCP/IP performance on communication paths including a slow wireless link. Examples include Indirect TCP by Rutgers University [3], Snoop by the University of California at Berkeley [5], Mowgli by the University of Helsinki [16], and TACO by the University of Copenhagen [12]. Mediators are also used to improve application level protocols like HTTP [17, 28]. In addition, the WAP architecture [30] 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, that is not mobile aware, requires two mediators: An *agent*, which is located in the mobile terminal, is bound with the terminal part of the application (usually a client), and 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 give background information on how Java RMI works, and why it works poorly in slow wireless environments. In Section 3 we present our approach to optimizing Java RMI, using a solution based on mediator technology. In Section 4 we evaluate the performance of our approach. We briefly summarize related work in Section 5. Finally, Section 6 states the conclusions.

2. Background and Motivation

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. 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 to a well-known socket, and obtains a remote reference.

Java RMI is built on top of a *transport layer*, which provides abstract RMI connections built on top of TCP 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.

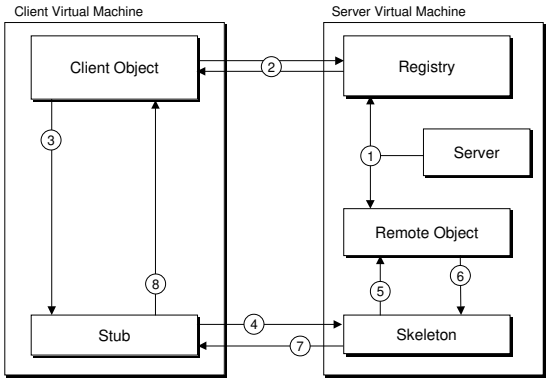

Figure 1. Java RMI Protocol

The general Java RMI architecture is depicted in Figure 1. 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 Problems

2.2.1. RMI Use of TCP Connections

The reuse of TCP connections in the transport layer 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 [16, 9]. 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 [14], TCP has to wait

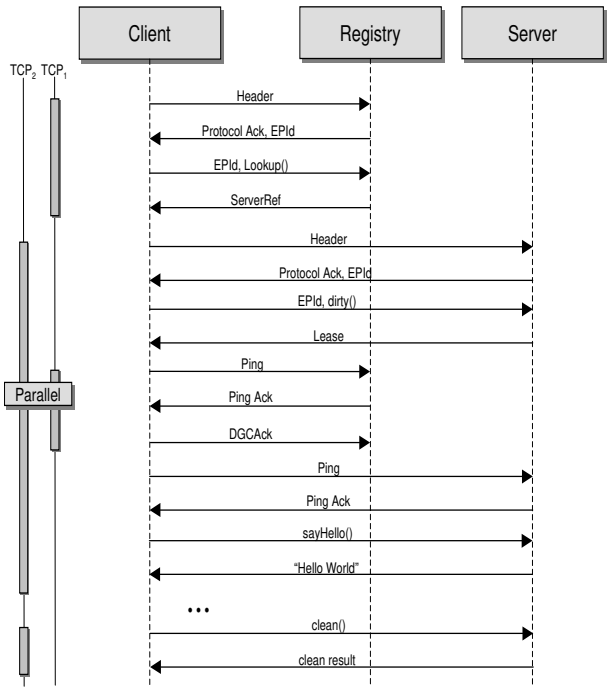

Figure 2. The trace of the "sayHello" remote invocation

for an acknowledgment once it has sent the first segment containing only one byte. This means that it takes a full round-trip before the second segment can be sent.

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

2.2.2. Analysis of an RMI Call

In this section we will show an analysis of a simple RMI call. The method is called "sayHello". As 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 2):

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 (EPId). 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 RMI connection is closed, logically closing the TCP1 connection.

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 RMI connection is closed, logically closing the TCP2 connection.

5. At this point, the Client must tell the DGC of the Registry that it obtained a remote reference, so it opens a new RMI connection. The first round-trip between the client and the Registry is a ping: in this way the client verifies that the TCP connection, which was logically closed before, is still alive[1]. Having verified this, the client communicates to 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 RMI 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 value.

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 the server. This exchange is preceded by the usual ping round-trip.

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

On a slow wireless link the amount of data that 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 the 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

[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 sent, the Registry must not release its server reference, since that might cause the server to wrongly conclude that there are no references in use.

Table 1. Invocation data traffic (bytes)

	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%)

really needed - one to get the server reference, and another for the actual invocation.

3. The Monads Approach

3.1. Optimizations

Optimization of Java RMI for wireless links means a 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.

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 are handled by the RMI proxy, and do not cross the wireless link.

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.

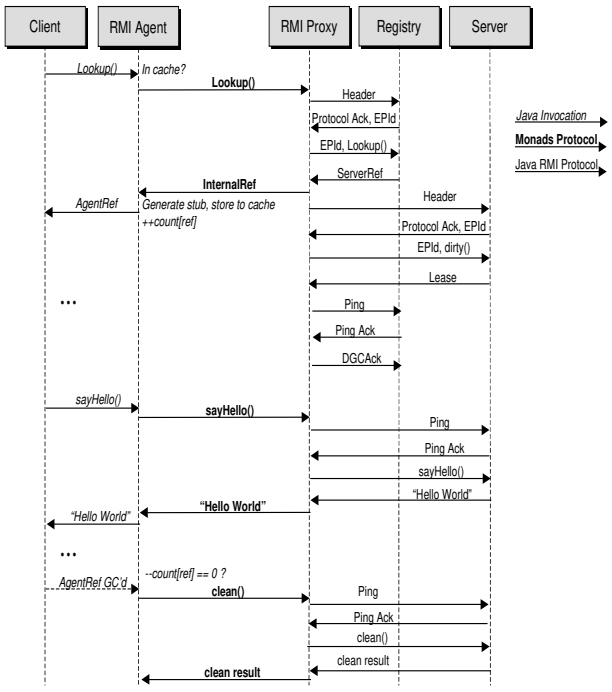

Figure 3. Using mediators to optimize the remote invocation

Distributed Garbage Collection – This protocol introduces heavy data overhead and its use in a wireless environment is less meaningful, since the link is subject to sudden disconnections that can be handled at the 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.

3.2. Use of Mediators

The role of the RMI mediators is shown in Figure 3. The RMI Agent captures the invocation made by the client. A lookup request is first checked in the local cache, and only if the remote reference is unknown is the request forwarded to the server.

DGC invocations are optimized by decoupling client and server. The RMI proxy keeps servers alive by periodically renewing the leases. It will only stop doing so once the RMI agent tells it that no more references to the server exist on the other side. In this way the DGC semantics are loosened to suit the needs of wireless communication, without

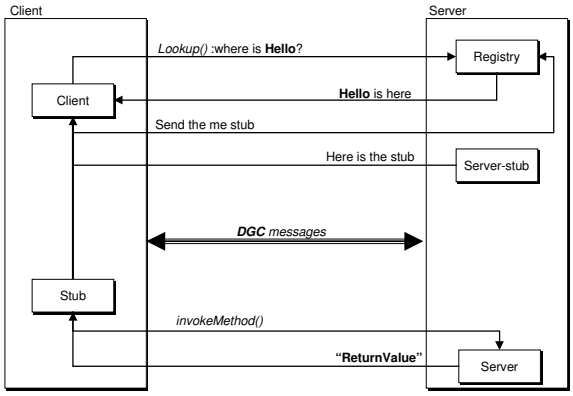

Figure 4. The normal RMI structure

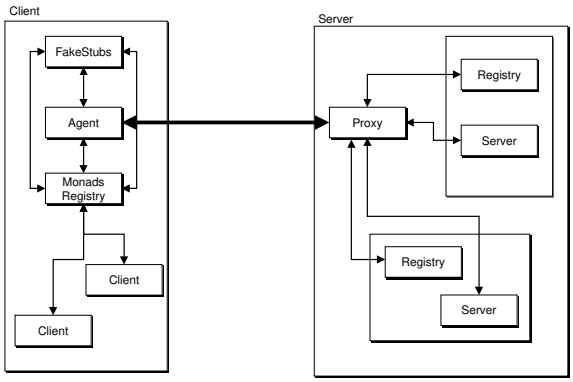

Figure 5. Optimized RMI structure

modifying client or server code. No lease requests (*dirty()* invocations) need to be sent over the wireless link, since all leases are managed on the fixed network side. 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 be sent to inform the other side that no references to a server exist anylonger. An optimized data representation can be used for these requests, further reducing DGC overhead. The Figures 4 and 5 outline the difference between our RMI optimization and the normal RMI.

4. Performance Evaluation

The performance of our optimized RMI implementation was measured in field trials. The objective was to study how our system behaves in different circumstances and to compare its performance with regular RMI.

4.1. Test Arrangements

In the measurements, we used a configuration specified in Table 2. The mobile node was connected to the Access Node using GSM HSCSD data service. Technically

Table 2. Hardware used in the measurements

Mobile Terminal	Toshiba Portégé 7020CT (Intel Pentium II/366MHz; 128Mb Main Memory)
Access Node	Compaq DeskPro EN6350 (Intel Pentium II/400MHz; 128Mb Main Memory)
GSM Phone	Nokia CardPhone 2.0 (Prototype)
Access Node Modem	Multitech MT2834ZDXI (28.8Kbps)

the HSCSD consists of two parts: multislot capability and the modified channel-coding scheme. The former provides the use of several parallel time slots per user where normal GSM can use only one. At the beginning of the HSCSD service, the maximum number of time slots is mostly limited to 3+1 (asymmetrical connection, three time slots for downlink) and 2+2 (symmetrical connection). The modified channel-coding scheme provides the user with a data rate of 14.4 kbps instead of the original maximum 9.6 kbps. In order to achieve higher bit rates, a more efficient puncturing method is used. This, on the other hand, decreases the radio interface error-correction performance. Therefore, the 14.4 kbps channel coding cannot be used when there is a lot of noise and interference affecting the quality of the radio signal. Depending on connection type and the capabilities of the network infrastructure, the maximum user data rate can be 28.8 kbps using a modem connection, 38.4 kbps using an ISDN V.110 protocol, or 57.6 kbps using an ISDN V.120 protocol.

We performed the measurements using a modem connection, thus having 28.8 kbps as maximum user data rate. The following combinations of channel coding (CC) and time slots were used: 1+1 (96CC; original GSM), 1+1 (144CC), 2+2 (96CC), 2+2 (144CC), and 3+1 (96CC). The measurements were conducted in a normal office environment with good HSCSD radio link conditions.

The software package used in the test was the KaRMI benchmark suite provided by the Institute for Program Structures and Data Organization of the University of Karlsruhe [22][4]. The test cases used are shown in Table 4. Each test was repeated 20 times in every configuration.

Since the implementation of the Java Virtual Machine and the underlying TCP/IP implementation are different in different operating systems, we conduct all the experiments in a Windows environment and in a Linux environment, as described in Table 3.

4.2. Summary of Performance Results

4.2.1. Lookup results

In the lookup case, we measured the time to get the reference to a remote object. In our optimized implementation,

[4]Software available at http://wwwipd.ira.uka.de/~hauma/EfficientRMI/

Table 3. Test Environment

	Client	Server
Windows		
OS	Windows 98	Windows NT (SP 6)
JDK	Sun JDK 1.2.2	Sun JDK 1.2.2
Linux		
OS	Linux 2.2.14	Linux 2.2.14
JDK	Sun JDK 1.2.2	Sun JDK 1.2.2

Table 4. Test cases set

Name	Parameters type	Return value
Void	null	null
ReturnPing	Byte[500]	Byte[500]
PingImage	Byte[5998]	null
PingText	Byte[9689]	null
ReturnText	Byte[9689]	Byte[9689]

we did not measure the case when the reference is found in the RMI Agent's reference cache; in this case, the reference is found in a few milliseconds depending on the speed of the underlying hardware and Java implementation.

Tables 5 and 6 show the results of lookup tests in Windows and Linux respectively. As expected, our implementation is significantly faster due to the reduction of unnecessary round-trips (see Figure 6). Using the original GSM (1+1, 96CC) our implementation is more than four times faster than the normal RMI in a Windows environment, and more than five times faster in Linux. An interesting result is that in HSCSD data service using the 14.4 kbps channel-coding scheme, the round-trip time is slightly higher than using the original 9.6 kbps channel coding. Since fewer round-trips are needed in our implementation, we gain more when 14.4 kbps is used as shown in Figure 7. The actual throughput does not significantly affect lookup results, since there is only a small number of bytes to send both in our implementation and in the original Java RMI.

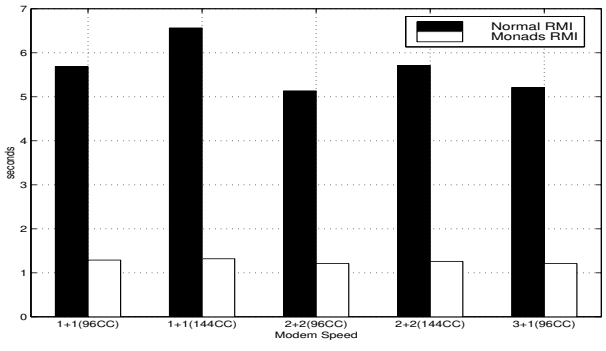

Figure 6. Summary of the lookup test in a Windows environment

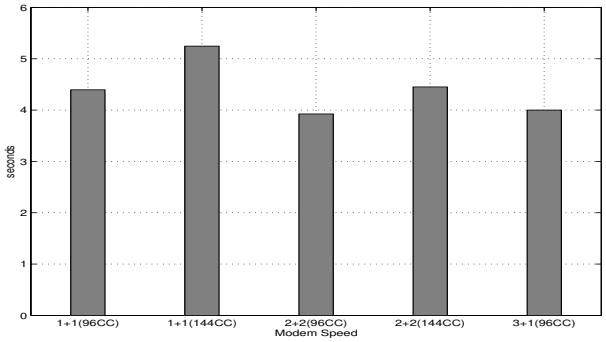

Figure 7. The difference between original RMI lookup and optimized lookup

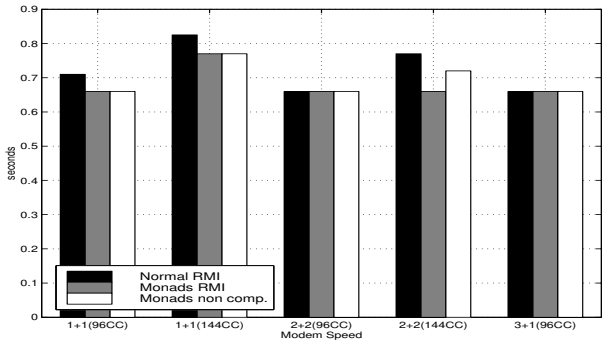

Figure 8. Summary of the void ping() test in a Windows environment

4.2.2. Invocation results

In the Invocation case, we performed an extensive study using different kernels found in the KaRMI package. Table 4 summarizes the test cases we used.

First, to evaluate basic overhead caused by RMI, we used a simple remote method; with no parameters nor return value. The results are given in Tables 7 and 8 for Windows and Linux respectively. As there is only a few bytes to transfer in both directions, the difference between our implementation and original RMI is insignificant. In our implementation there is some additional processing overhead, as all invocations go through the RMI Agent. However, with the slow wireless communication path being the bottleneck, this is not a problem. In Figure 8, the results of the Windows environment are illustrated.

Next we evaluate the case where the remote object takes an array of bytes (500) as an argument, and as a return value, returns the same array back. Tables 7 and 8 summarizes the results of this test case in Windows and Linux respectively. In this case, there is now more data to send, and therefore using compression affects the results significantly. This is

mainly due to the content of the array; in the KaRMI package all arrays are always initialized with zeros.

In order to evaluate our implementation with more realistic data, we used a case where the client sends a file to the server as a parameter of a remote method, and another case, where the server also returns the same file as a return value. As the content of the file affects the results of our implementation significantly when compression is used, we selected two files with very different content; the text file used was a HTML page with size 9689 bytes and the image file used was a GIF image with size 5998 bytes. The general purpose compression algorithm we are using is not able to compress GIF images. As a result, we are not gaining anything from using compression with very random data, but on the other hand, the additional overhead needed in this case is not significant. When we are transferring text data for example, the compression works well, and our implementation is significantly faster than the original RMI. The summaries of image and text transfer results are shown in Tables 7 and 8. The results are summarized in Figure 9.

5. Related Work

Optimizing Java and Java RMI performance have been quite popular research topics; see for example [1]. The work, however, has concentrated on high-speed networks—to the best of our knowledge there are no published results yet on Java RMI performance in slow wireless networks (cellular networks). However, some work has been carried out for CORBA in slow wireless networks.

For the high-speed networks UKA serialization and KaRMI, developed at the University of Karlsruhe [22], provide a more efficient RMI for Java. The Manta project (Fast Parallel Java) in the Vrije Universisteit, Amsterdam [19, 29] has developed an efficient remote method invocation based on a transparent extension of Java for distributed environments. The Manta RMI is a part of the Manta environment and it cannot be used separately. At Indiana University there is a group that has conducted an interoperability and performance study of remote method invocation [6]. Another performance evaluation study has been carried out by the HORB project in Japan [13].

Wireless access and terminal mobility issues in CORBA has been studied in the EC/ACTS projects DOLMEN [18] and OnTheMove [27] as well as in the Alice project at Trinity College Dublin [11]. Somewhat related work has also been carried out in the Rover project at MIT [15]. In addition, a mobile RPC proposal, called M-RPC, has been done by Rutgers University [4].

The DOLMEN work led to OMG Telecom Domain Task Force activities: An RFI [24] was issued in June 1998, a white paper [25] in November 1998, and an RFP [26] in May 1999. The OMG specification is expected to be completed this year. It may have some implications to the Java RMI since future RMI is expected to use IIOP.

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.

In the future we will extend our implementation to support both personal and terminal mobility. These new features will greatly improve the usability of Java RMI in a mobile environment. The use of mediator technology will support the implementation of these features.

Acknowledgements

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

References

[1] ACM 1998 Workshop on Java for High-Performance Network Computing. Available electronically from `http://www.cs.ucsb.edu/conferences/java98/program.html`.

[2] K. Arnold, O. Sullivan, R. Scheifler, J. Waldo, and A. Wollrath. *The Jini Specification*. Addison-Wesley, 1999.

[3] A. Bakre and B. Badrinath. Indirect Transport Layer Protocols for Mobile Wireless Environment. *Mobile Computing*, pages 229–252, 1996.

[4] A. Bakre and B. R. Badrinath. M-RPC: A Remote Procedure Call Service for Mobile Clients. In *Proc. of the ACM MobiCom'95*, pages 97–110, Berkeley, Calif., Nov. 1995.

[5] H. Balakrishnan, S. Seshan, and R. Katz. Improving Reliable Transport and Handoff Performance in Cellular Wireless Networks. *ACM Wireless Networks*, 1(3):469–482, Dec. 1995.

[6] F. Berg, S. Diwan, J. Villacis, J. Balasubramanian, E. Akman, and D. Gannon. Java RMI Performance and Object Model Interoperability: Experiments with Java/HPC++. In *Proc. of the ACM 1998 Workshop on Java for High-Performance Network Computing*, Stanford, Palo Alto, Calif., Feb. 1998.

[7] A. Birrel and B. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, Feb. 1984.

[8] J. Border, M. Kojo, J. Griner, and G. Montenegro. Performance Enhancing Proxies. Internet draft draft-ietf-pilc-pep-01.txt, Dec. 1999.

Table 5. Comparison between normal RMI and optimized RMI for the *Windows Lookup* case

	Original RMI				Monads			
	Min	Max	Median	Average	Min	Max	Median	Average
1+1 (96CC)	5540	6530	5685	5693	1260	4840	1290	1528
1+1 (144CC)	1000	8900	6565	6667	1260	1430	1320	1346
2+2 (96CC)	1543	6040	5135	5118	1150	1380	1210	1234
2+2 (144CC)	1071	8240	5710	5682	1160	2140	1260	1313
3+1 (96CC)	5100	8300	5210	5315	1100	2310	1210	2310

Table 6. Comparison between normal RMI and optimized RMI for the *Linux lookup* case

	Original RMI				Monads			
	Min	Max	Median	Average	Min	Max	Median	Average
1+1 (96CC)	6304	6478	6390	6388	1116	2346	1136	1200
1+1 (144CC)	6490	8332	6684	7080	1107	1622	1119	1154
2+2 (96CC)	5118	5745	5216	5263	958	1225	979	995
2+2 (144CC)	5382	8364	6214	6378	994	5878	1068	1625
3+1 (96CC)	5308	9547	5963	6144	961	1139	980	1139

[9] R. Cáceres and L. Iftode. Improving the Performance of Reliable Transport Protocols in Mobile Computing Environments. *IEEE Journal on Selected Areas in Communications*, 13(5):850–857, June 1995.

[10] GSM Technical Specification, GSM 03.34. High Speed Circuit Switched Data (HSCSD), Stage 2, May 1999. Version 5.2.0.

[11] M. Haahr, R. Cunningham, and V. Cahill. Supporting CORBA Applications in a Mobile Environment. In *Proc. of the ACM MobiCom '99 Conference*, pages 36–47, Seattle, Wash., Aug. 1999.

[12] J. Hansen, T. Reich, B. Andersen, and E. Jul. Dynamic Adaptation of Network Connections in Mobile Environments. *IEEE Internet Computing*, 2(1):39–48, Jan. 1998.

[13] S. Hirano, Y. Yasu, and H. Igarashi. Performance Evaluation of Popular Distributed Object Technologies for Java. In *Proc. of the ACM 1998 Workshop on Java for High-Performance Network Computing*, Stanford, Palo Alto, Calif., Feb. 1998.

[14] V. Jacobson. Congestion Avoidance and Control. In *Proc. ACM SIGCOMM '88 Symposium on Communications Architectures and Protocols*, pages 314–329, Stanford, California, Aug. 1988.

[15] A. D. Joseph, J. A. Tauber, and M. F. Kaashoek. Mobile Computing with the Rover Toolkit. *IEEE Transactions on Computers (Special Issue on Mobile Computing)*, 46(3), Mar. 1997.

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

[17] M. Liljeberg, T. Alanko, M. Kojo, H. Laamanen, and K. Raatikainen. 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)*, pages 132–139. IEEE Computer Society Press, 1995.

[18] M. Liljeberg, K. Raatikainen, M. Evans, S. Furnell, N. Maumon, E. Veltkamp, B. Wind, and S. Trigila. Using CORBA to Support Terminal Mobility. In *Proc. of TINA'97 Conference*, pages 56–67. IEEE Computer Society Press, 1998.

[19] J. Maassen, R. van Nieuwpoort, R. Veldema, H. E. Bal, and A. Plaat. An Efficient Implementation of Java's Remote Method Invocation. In *Proc. 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 173–182, Atlanta, GA, May 1999.

[20] S. Microsystems. Java Remote Method Invocation – Distributed Computing for Java. White Paper, 1998.

[21] G. Montenegro, S. Dawkins, M. Kojo, V. Magret, and N. Vaidya. Long Thin Networks. RFC 2757, Jan. 2000.

[22] C. Nester, M. Philippsen, and B. Haumacher. A More Efficient RMI for Java. In *Proc. of ACM 1999 Java Grande Conference*, pages 152–157, San Francisco, Calif., June 1999.

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

[24] Object Management Group. Telecom DTF. RFI on Wireless Access and Mobility in CORBA. OMG document telecom/98-06-06, June 1998.

[25] Object Management Group. Telecom DTF. White Paper on Wireless Access and Mobility in CORBA. OMG document telecom/98-11-09, Nov. 1998.

[26] Object Management Group. Telecom DTF. RFP on Wireless Access and Terminal Mobility in CORBA. OMG document telecom/99-05-05, May 1999.

[27] OnTheMove Web Site. Available electronically from `http://www.sics.se/~onthemove/`.

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

[29] R. van Nieuwpoort, J. Maassen, H. E. Bal, T. Kielmann, and R. Veldema. Wide-area parallel computing in Java. In *Proc. of ACM 1999 Java Grande Conference*, pages 8–14, San Francisco, Calif., June 1999.

[30] WAP Forum. Wireless Application Environment Overview, version 1.2, Nov. 1999. Available electronically from `http://www.wapforum.org/`.

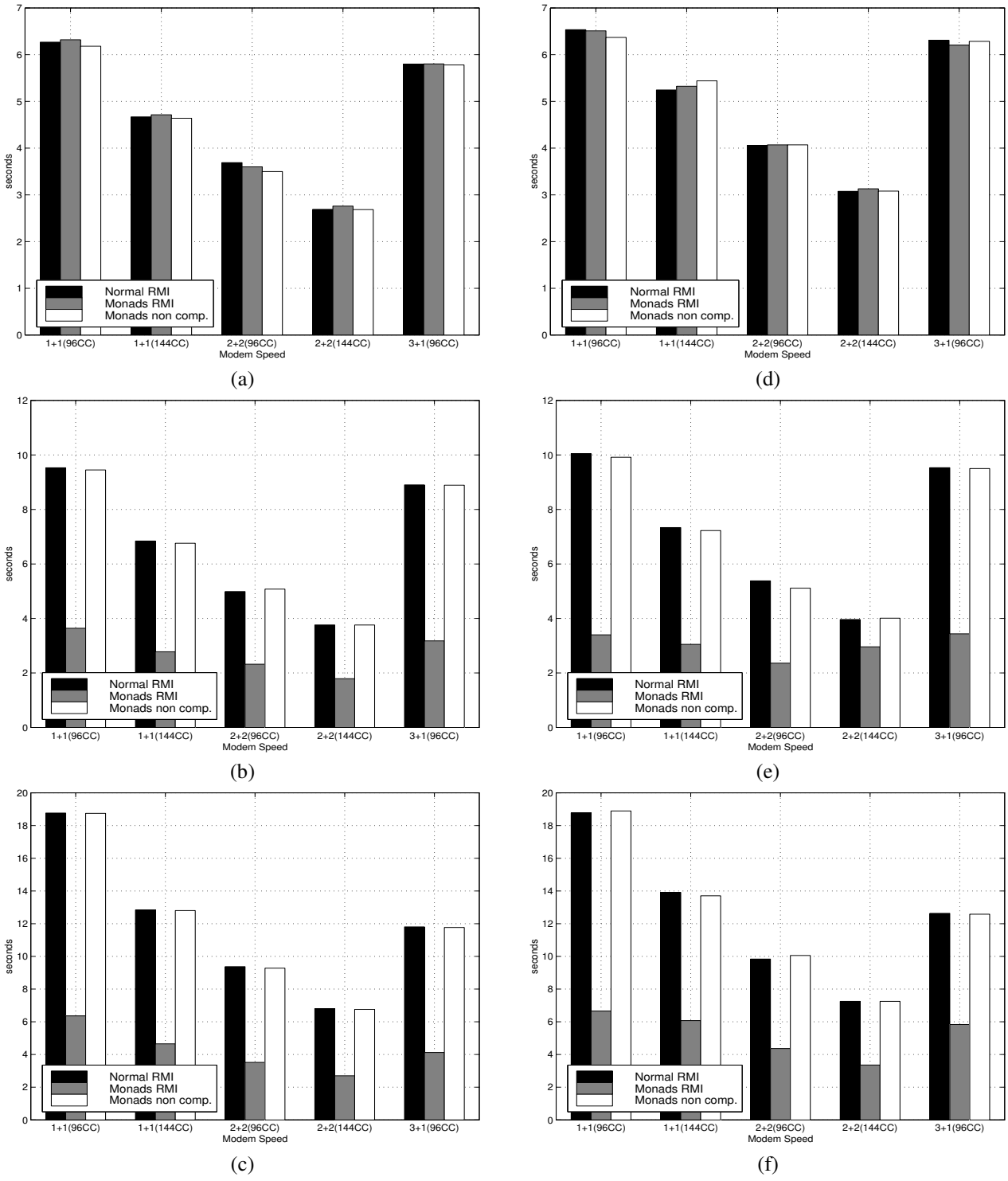

Figure 9. Figure (a) shows the invocation times of uplink image transfer using different link speeds in Linux environment. Figure (b) shows the invocation times of uplink text transfer while Figure (c) shows the invocation times of two-way text transfer in Linux environment. Figures (d),(e), and (f) show the corresponding times in the Windows environment

Table 7. Comparison between normal RMI and optimized RMI in the Windows environment

	Original RMI				Monads Compressed				Monads Uncompressed			
	Min	Max	Median	Average	Min	Max	Median	Average	Min	Max	Median	Average
	Comparison between normal RMI and optimized RMI for the *Windows Void* case											
1+1 (96CC)	650	1380	710	719	650	720	660	662	600	880	660	651
1+1 (144CC)	760	1980	825	867	710	2040	770	835	710	2030	770	826
2+2 (144CC)	710	1980	770	821	650	720	660	681	710	770	720	722
3+1 (96CC)	650	1370	660	697	600	660	660	654	650	660	660	659
	Comparison between normal RMI and optimized RMI for the *Windows ReturnPing* case											
1+1 (96CC)	1640	1710	1650	1672	710	820	770	749	1530	3400	1540	1689
1+1 (144CC)	1480	4060	1540	1649	820	1480	855	881	1420	5540	1455	1667
2+2 (144CC)	1100	1590	1150	1193	710	1210	770	804	1090	3790	1100	1301
3+1 (96CC)	1370	1540	1430	1437	650	2580	685	782	1310	3460	1320	1529
	Comparison between normal RMI and optimized RMI for the *Windows PingImage* case											
1+1 (96CC)	6260	13900	6535	7382	6150	15980	6510	7475	6150	16090	6370	7352
1+1 (144CC)	5210	33940	5245	8436	5220	11210	5325	5989	5160	8680	5440	6031
2+2 (144CC)	3020	4510	3075	3238	3070	4720	3130	3202	3020	6810	3080	3477
3+1 (96CC)	6200	16750	6310	7780	6150	12580	6205	6541	6150	10380	6285	6755
	Comparison between normal RMI and optimized RMI for the *Windows PingText* case											
1+1 (96CC)	9450	17300	10050	10488	3350	6590	3400	3576	9390	19230	9915	10840
1+1 (144CC)	7250	10270	7335	7563	2960	6420	3050	3298	7140	8240	7225	7296
2+2 (144CC)	3950	4330	3960	3990	2690	5380	2960	3125	3950	7960	4010	4399
3+1 (96CC)	9440	12360	9530	9897	3350	6260	3435	4248	9440	11640	9500	9625
	Comparison between normal RMI and optimized RMI for the *Windows ReturnText* case											
1+1 (96CC)	18340	19220	18780	18691	6090	10930	6150	6657	18290	27080	18895	19811
1+1 (144CC)	13680	26920	13920	15027	5000	10990	5110	6072	13620	15050	13705	13841
2+2 (144CC)	7190	9940	7250	7396	2910	6750	3075	3359	7200	10540	7250	7558
3+1 (96CC)	12570	13290	12630	12682	4450	7580	5820	5836	12520	15220	12580	12715

Table 8. Comparison between normal RMI and optimized RMI in the Linux environment

	Original RMI				Monads Compressed				Monads Uncompressed			
	Min	Max	Median	Average	Min	Max	Median	Average	Min	Max	Median	Average
	Comparison between normal RMI and optimized RMI for the *Linux Void* case											
1+1 (96CC)	749	1543	779	809	699	720	719	716	699	719	719	717
1+1 (144CC)	799	1663	799	844	759	2469	759	846	759	800	792	780
2+2 (144CC)	710	1953	719	781	669	1899	719	772	679	720	719	714
3+1 (96CC)	679	2744	694	891	639	1070	659	679	639	2016	649	847
	Comparison between normal RMI and optimized RMI for the *Linux ReturnPing* case											
1+1 (96CC)	1829	1879	1839	1846	800	868	800	803	1740	1837	1760	1756
1+1 (144CC)	1430	1507	1439	1446	790	898	800	805	1390	2210	1440	1466
2+2 (144CC)	1078	1119	1080	1089	720	1330	760	786	1110	1177	1120	1125
3+1 (96CC)	1319	1379	1350	1352	700	2560	720	939	1300	3430	1320	1495
	Comparison between normal RMI and optimized RMI for the *Linux PingImage* case											
1+1 (96CC)	6239	6340	6269	6276	6220	12660	6320	7126	6180	6241	6180	6191
1+1 (144CC)	4629	5829	4669	4826	4590	12510	4714	5690	4600	4791	4639	4652
2+2 (144CC)	2659	3189	2689	2742	2680	7200	2760	3170	2680	3200	2685	2771
3+1 (96CC)	5779	6859	5800	5890	5740	8430	5804	5938	5739	9100	5780	6056
	Comparison between normal RMI and optimized RMI for the *Linux PingText* case											
1+1 (96CC)	9479	10109	9529	9599	3460	5570	3480	3641	9420	9483	9450	9451
1+1 (144CC)	6789	7479	6840	6894	2720	3560	2720	2778	6750	7170	6760	6838
2+2 (144CC)	3712	5490	3764	3938	1719	3000	1720	1790	3750	3920	3760	3771
3+1 (96CC)	8879	9809	8900	8963	3160	3209	3180	3178	8840	10300	8890	9203
	Comparison between normal RMI and optimized RMI for the *Linux ReturnText* case											
1+1 (96CC)	18740	18858	18765	18774	6331	6970	6360	6391	18719	19880	18749	18854
1+1 (144CC)	12829	13450	12840	12920	4600	10820	4655	5595	12750	16830	12800	13079
2+2 (144CC)	6691	7131	6805	6816	2671	3920	2700	2762	6711	7190	6760	6806
3+1 (96CC)	11759	12463	11804	11870	4100	4188	4122	4126	11729	15151	11766	12190