Journal of Computing and Information Technology - CIT 14, 2006, 3, 255–264
doi:10.2498/cit.2006.03.08

Utilizing Object Compression for Better J2ME Remote Method Invocation in 2.5G Networks

Jalal Kawash, Ahmad El-Halabi and Ghassan Samara

Department of Computer Science, American University of Sharjah, UAE

This paper introduces two new Java 2 Platform Micro Edition (J2ME) Remote Method Invocation (RMI) packages. These packages make use of serialized object compression and encryption in order to respectively minimize the transmission time and to establish secure channels. The currently used J2ME RMI package does not provide either of these features. Our packages substantially outperform the existing Java package in the total time needed to compress, transmit, and decompress the object for General Packet Radio Service (GPRS) networks, often called 2.5G networks, even under adverse conditions. The results show that the extra time incurred to compress and decompress serialized objects is small compared to the time required to transmit the object without compression in GPRS networks. Existing RMI code for J2ME can be obliviously used with our new packages.

Keywords: RMI, compression, encryption, J2ME, wireless, GPRS.

1. Introduction

There is a constant increase in the proliferation of wireless handheld devices and in their computing capabilities. Wireless devices have limitations with processor power, memory space, and bandwidth. In order to cope with some of these limitations and to address the needs of a rapidly growing wireless technology market, Sun Microsystems introduced Java 2 Micro Edition (J2ME) platform. About half a billion users currently carry handheld phones that can run J2ME and there are 150 mobile operators supporting Java [1]. Except for mobile gaming, this huge device population is not being taken advantage of.

Java Remote Method Invocation (RMI) allows programmers to rapidly create Java-to-Java distributed applications, without the need to reason about the complex message passing details. J2ME supports an RMI Optional Package (RMI OP) on some device families, and Java RMI on wireless devices offers a promising mechanism to address the need to build successful mobile and Peer-to-peer (P2P) applications. Many reasons lead us for this conjecture. These include Java's wide adoption, development convenience, crossplatform compatibility on a wide range of supporting devices, and an appealing security model.

Java objects are passed by value to remote methods through *serialization*, which is a mechanism of converting a set of objects into a linear stream of bytes. Because objects may contain references to other objects, serializing Java objects converts them into linked structures and it also inflates their original size. Transmitting such large object graphs can consume a large chunk of network bandwidth and CPU time. In spite of low bandwidth and large transmission error rates in wireless networks, RMI OP does not try to compress serialized objects before transmitting them. Furthermore, RMI OP does not encrypt these objects before transmission. The programmers are left with the job of ensuring both efficient and secure transmission of Java serialized objects.

In this paper, we describe RMI – Enhanced Optional Package (RMI EOP), a drop-in replacement for the RMI OP package that incorporates automatic compression and optional encryption for Java objects. Old code written using RMI OP is oblivious to the RMI EOP replacement. The paper describes two versions of RMI EOP that differ in the compression algorithms they use. RMI EOP-GZip makes use of GNU Zip

to compress data objects and RMI EOP-PPM makes use of prediction by partial matching. Our experimental results show that, in spite of the extra time incurred by compressing and decompressing objects, RMI EOP-GZip transmits objects faster than RMI OP by a factor of 60% to 74% in GPRS networks. Similarly, RMI EOP-PPM is 27% to 54% faster than RMI OP also using GPRS.

Sun's original Java RMI implementation was slow and many enhancements and drop-in replacements have been suggested. Krishnaswamy et al. [2] proposed a better implementation of RMI that uses UPD instead of TCP and relies on caching. Thiruvathual et al. [3] and Philipsen et al. [4] suggested alternative object serialization mechanisms. Since serialization is the major performance inhibitor in RMI, Berg and Ploychronopoulos [5] implemented part of Java's serialization protocol in native code to make it faster. Other similar studies to improve RMI efficiency include Nester et al.'s [6] and Massen et al.'s [7]. The latter discussed a new implementation of RMI which uses static instead of just-in-time compiling. This allows the utilization of compile-time information to get better object sterilizations. The implementation also utilizes faster protocol and communication. Their experimental results were obtained on multiprocessor machines with 16 or 32 processors. However, Nester et al. [6] provide RMI protocols that are faster than the standard Sun implementation, even without resorting to native code. Kurzyniec et al. [8] tested different RMI protocol implementations. All of these studies are concerned with "full-size" Java Virtual Machines (JVMs) and do not apply to J2ME environments.

However, there is some (although very limited) related work in J2ME environments. P. C. Wei et al. [9] reported on RMI support and optimization for Bluetooth environments. They developed a cost model for access patterns in RMI communication. This study is limited to Bluetooth. Other related studies include memory management [10], garbage collection [11], and energy savings [12] for limited-memory environments. Optimizing Java-based cryptography applications to Kilobyte Virtual Machines is discussed by Matsuoka et al. [13].

To the best of our knowledge, this is the first study that is concerned with developing a more efficient replacement of J2ME OP using serialized object compression. Our work also includes encryption for secure channel mechanisms.

Paper roadmap: Section 2 provides background information. Section 3 contains a detailed description of both versions of RMI EOP and Section 4 presents the experimental evaluations and results. Section 5 concludes the paper.

2. Background

2.1. J2ME

The J2ME [14] architecture (Figure 1) involves *configurations*, *profiles*, and *optional packages*, which can be used by developers to construct a complete Java runtime environment that closely fits the requirements of a particular range of devices. Each combination is optimized for the memory, processing power, and I/O capabilities of a class of mobile devices. The configurations and profiles are used to customize the J2ME runtime environment.

The basic runtime environment is defined as a configuration, combining a virtual machine and a collection of core classes. These core classes can run on a target family of devices, sharing similar characteristics. A configuration defines minimum requirements for such a family of devices. These requirements are: memory size, virtual machine, language support, and core runtime libraries.

J2ME supports two configurations: the Connected Device Configuration (CDC), version 1.0 known as JSR36 (and recently version 1.1

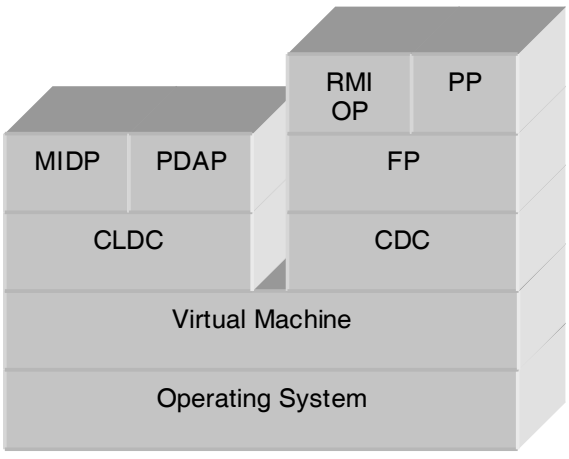

Fig. 1. J2ME architecture.

known as JSR 218), and the Connection Limited Devices Configuration (CLDC), version 1.0 known as JSR30 (and recently version 1.1 known as JSR 139). CLDC is targeted to resource constrained mobile devices. The virtual machines Kuaui Virtual Machine (KVM) and (recently) HotSpot are respectively used in CLDC 1.0 and 1.1. RMI is not supported in CLDC. CDC is targeted to higher-end devices with more capabilities. Compact Virtual Machine (CVM) is used in CLDC 1.0 and the recent CLDC 1.1 uses the HotSpot implementation. In J2ME, RMI is only [14] supported in CDC.

Configurations are complemented by profiles in order to provide support for application development and execution. Profiles provide Application Programmer Interfaces (APIs), for devices that share some capabilities. CDC has many profiles and packages, including the JSR46 Foundation Profile (FP), JSR129 Basis Profile (PBP), JSR62 Personal Profile (PP) and RMI OP. The RMI OP reference implementation can be built with implementations of CDC/FP-based profiles and PBP. RMI OP requires a minimum RAM of 512 KB.

2.2. Data Compression

Data compression is the process of encoding data so that its storage space or transmission time is minimized. There are two types of data compression algorithms, depending on how they preserve data. *Lossy data compression* does not require the compressed data to be exactly the same as the decompressed data, but should be close enough in order to be useful in some way [15]. Obviously, lossy compression algorithms are not suitable for compressing the programs including Java objects, which require *lossless data compression*. This category prohibits any difference between decompressed and compressed data [15].

Our RMI enhancement makes use of two lossless tools, which are Prediction by Partial Matching (PPM) [16, 17] and GNU Zip (GZip) [18].

PPM is an adaptive statistical data compression technique based on context modeling and prediction. PPM was originally proposed by Cleary and Witten [16]. In its finite context form, PPM(k) predicts the $k+1^{st}$ symbol in an input stream, based on the preceding k symbols in the stream. k is called the model order and PPM makes use of PPM(k) for values from 0 up to a predefined maximum value, typically 16. For each model (for each k), PPM keeps track of the characters that have occurred after every subsequence of length k, as well as the frequencies of these occurrences. PPM generates prediction probabilities for the next characters based on these measures. The probability distributions are different for different model orders and PPM combines these distributions into one distribution. That is, PPM-level n makes use of PPM(k), for $0 \leq k \leq n$, to generate what is called escape probabilities. It first uses PPM(n) to calculate the prediction probabilities. If a new character is encountered when using PPM(k), for which model order k cannot be used to encode it, PPM switches to PPM($k-1$). This process is repeated until the character is not new in that model order. In the worst case, this can be repeated until model order 0 is reached.

There has been many improvements to PPM [19, 20]; in particular PPM* [19] uses similar ideas, but with unbounded contexts.

GZip is an open-source replacement for the *UNIX compress* program and was created by Jeanloup Gailly and Mark Adler. GZip gives a better compression rate, the algorithms it uses are non-patented. GZip is based on the deflate algorithm [18]. The deflate algorithm considers the input stream to be a sequence blocks of arbitrary sizes. This algorithm compresses each block separately, using a combination of LZ77 and Huffman coding. Huffman coding creates trees that allow us to encode data based on its frequency in the input stream. The more frequently a data item occurs, the smaller its encoding should be. The deflate algorithm creates a Huffman tree for each block, independently from the trees corresponding to other blocks. However, LZ77 may use references to an input string duplicated in a previous block. Hence, each block with the deflate algorithm consists of Huffman code trees representing the compressed data and the compressed data itself. The compressed data consists of literal (unduplicated) data and references to duplicated data in previous blocks.

3. RMI Enhanced Optional Packages

Our protocol for RMI EOP consists of a revision of how RMI OP performs the *writing* of the serialized object to the "data block output stream" and consequently a revision of the *reading* oper-

ation from the stream. This requires the agreement between the remote object's skeleton and the client's stub on a new header format, implemented by our *create header* operation. We present two versions of the protocol, the first relies on PPM for compression and is called EOP-PPM and the second uses GZip and is called EOP-GZip.

3.1. Reading and Writing

The writing operation is outlined in Figure 2, for both versions. For small size (less than 32 bytes) serialized objects, compression is omitted, unless encryption inflates the object serialization to a size larger than 32 bytes. For larger objects, EOP-PPM makes use of PPM-level 8 and PPM-level 16, depending on the size of the object and EOP-GZip always uses GZip for any object of size more than 32 bytes. Regardless of the compression tool used, an appropriate packet header is generated. The read operation is merely a reversed process for the write operation.

3.2. Operation Modes

In each package there are 4 operation modes reflecting the use of compression or encryption, both, one without the other, or neither. In the case where neither is used, RMI EOP simply operates as RMI OP.

3.3. Packet Header

The Create Header procedure is responsible for generating compatible packet headers between the remote object's skeleton and the client's stub. The header size varies with the object size. RMI EOP uses 3 header sizes, which can be determined by reading the first byte (called the *indicator*). For example, if the indicator includes a value less than 40Hex, the header is a 1-byte header. The headers are as follows:

1-byte header: four bits indicate the size of the object and an additional bit indicates if the object is encrypted or not.

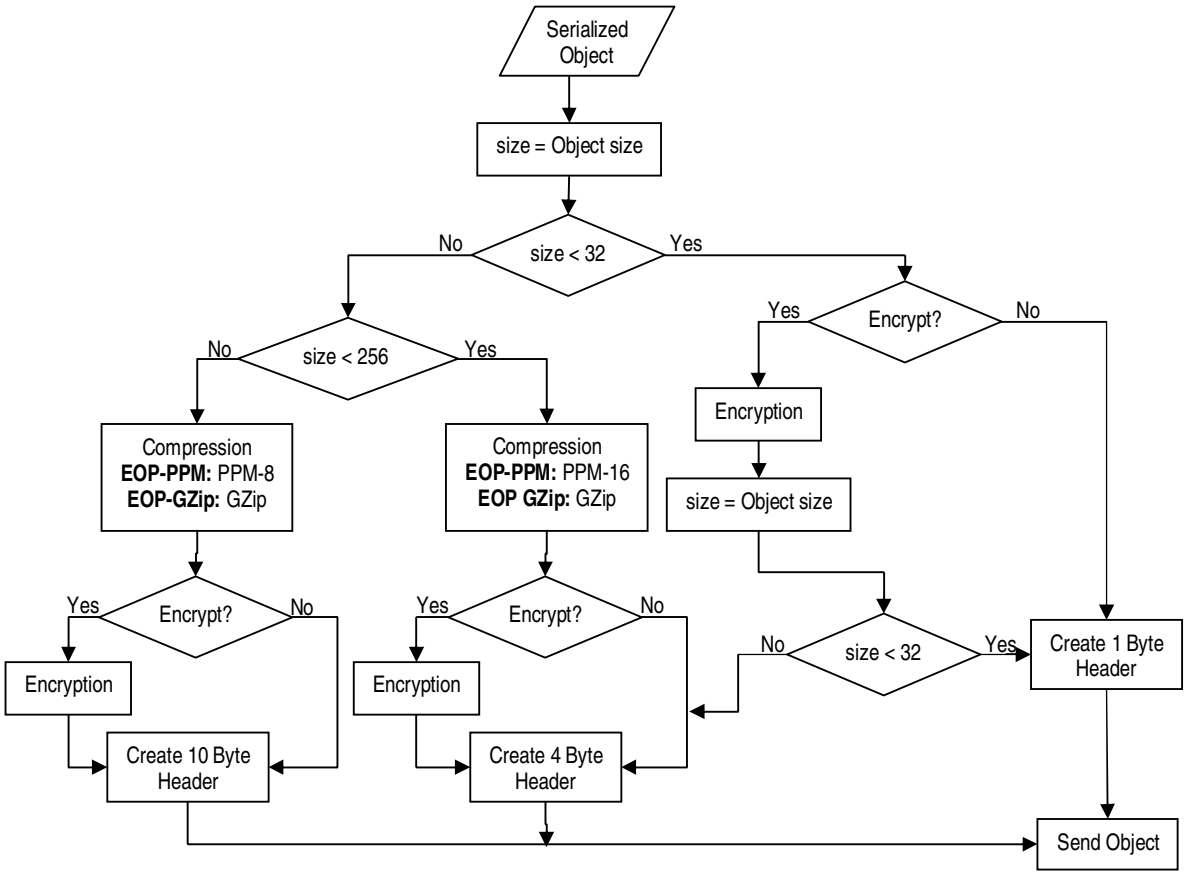

Fig. 2. A summary of the RMI EOP writing operation (both versions).

4-byte header: The first byte is the indicator byte; the second byte is the operation mode; the third is the original object size (prior to compression and decryption); and the last is for the size after compression and encryption.

10-byte header: This header adds (to the 4-byte header) 3 bytes for the original size of the object and 3 bytes for the resulting size.

3.4. Encryption

Secure RMI is established by using symmetric key cryptography. The symmetric key is valid for one RMI session and public key cryptography is used in order to exchange the session key.

Secure channels in RMI EOP are implemented using the Bouncy Castle Crypto package 1.3 [21]. This package is organized so that it contains a light-weight API suitable for use in any environment including the newly released J2ME and doesn't require a high computation power.

The Bouncy Castle Crypto package supports a handful of encryption algorithms. It includes symmetric and asymmetric cryptosystems, in addition to message digest algorithms. RMI EOP makes use of the Data Encryption Standard (DES) [22] to establish a secure channel and the Rivest-Shamir-Adleman (RSA) algorithm [23] to communicate the shared key.

3.5. Memory Requirements

On limited memory devices, PPM can be prohibitively expensive in terms of RAM usage. RMI EOP requires a minimum of 1MB of RAM for fast mode and 30MB for medium mode. Typical CDC devices are equipped with 32 to 128MB of RAM. For storing objects, RMI EOP can optionally make use of chunking. This is accomplished through the use of a 100KB buffer to store the compressed and encrypted object. When the buffer is filled, its contents are transmitted and a new chunk can be prepared in the buffer.

3.6. Using RMI EOP

The changes to existing Java RMI OP code are minimal. In both implementations (EOP-GZip and EOP-PPM) the programmer is required to make remote object types subclasses of `comUnicastRemoteObject`, instead of `UnicastRemoteObject`. A new overloaded constructor is provided in `comUnicastRemoteObject`, `super(port, mode)`, which allows the programmer to choose from the four operation `modes`. All remaining code is oblivious to changes.

4. Experimental Evaluation

4.1. Java Object Benchmark

The benchmark introduced by Nester et al. [6] is not suitable for our purpose, since we are not testing different RMI protocols, and we are not aware of any other RMI benchmarks. In fact, both RMI OP and RMI EOP work using the same underlying RMI protocol and will both generate exactly the same communication patterns and number of messages. Since RMI EOP uses compression and encryption, we are obliged to verify how RMI EOP performs relative to RMI OP on different Java objects. So, our benchmarking is primarily concerned with using different object contents and sizes.

Test Cases

Our benchmark contains objects that contain Java types (all primitive types plus some objects). These are: `boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, `double`, `String`, `Vector`, and `HashTable`. The *base* class contains arrays of all of these types and methods that manipulate each of the existing types. All test cases are generated as "subsets" of the base class. All arrays of all members of one class have the same size, *ArraySize*, which is varied between experiments. ArraySize is one (but not the only) indicator or of the object size. The content types of the class also influence the size of its instance objects. Variations on this base class are constructed by systematically removing a member of the largest type in size. We generated 550 different test case classes, with different contents and sizes. Example test classes are given in Table 1. Class 1 in Table 1 is the base class and contains a balance of variable and method members.

Array type / Class	1	2	3	4
boolean	✓	✓		✓
byte	✓	✓		
char	✓	✓		✓
short	✓	✓		✓
int	✓	✓	✓	
long	✓	✓	✓	
float	✓	✓	✓	
double	✓	✓	✓	✓
String	✓		✓	✓
Vector	✓		✓	
HashTable	✓		✓	✓

Table 1. Examples of the contents of test cases.

4.2. Parameters and Assumptions

RMI EOP incurs an additional cost for compressing the object by the sender and then decompressing it by the receiver. The time required to both compress and decompress an object is called *processing* time. So the total time required to send and receive an object between the calling object and the remote object in RMI EOP consists of processing time in addition to the *transmission* time. We assume that both ends in RMI communication (sender and receiver) are CDC devices (P2P environment). That is, the time taken to compress and decompress an object is substantially larger than the case when one of the parties is a full-size server machine. Even with this assumption, we demonstrate that RMI EOP is appealing.

In RMI OP, only the transmission time needs to be measured. The set up time is similar for both packages and is ignored. When secure channels are used, the processing time for both packages includes the time required for encryption and decryption. Our experiments were conducted based on GPRS wireless network technologies. The practical peak performance of GPRS is 53.6 Kilobit/second [22].

4.3. Development and Testing Environment

RMI EOP was developed using Nokia's Series 80 Developer Platform 2.0 SDK for Symbian OS – For Personal Profile, Version 1.0 [25]. The running time measurements reported in the next subsection are valid for devices with clock speed of 400 MHz and 128MB RAM [26].

4.4. Experimental Results

Even if GPRS operates at its peak, RMI EOP (both versions) show a total time savings over RMI OP. EOP-GZip outperforms both RMI OP and RMI EOP-PPM. Figure 3 depicts a summary of the results, ignoring re-transmissions, and without encryption. In Figure 3, the x-axis represents *ArraySize* and the y-axis represents the total time (processing and transmission).

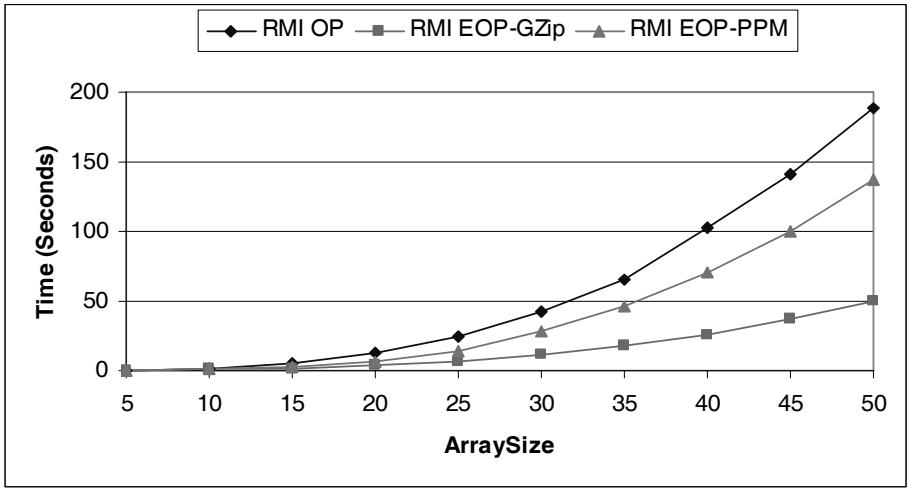

Fig. 3. Total time for the three J2ME RMI packages.

ArraySize	10	20	30	40	50
Original Object Size (Bytes)	12835	88445	286855	686065	1263275
Compressed Object Size (Bytes)	4665.99	24498.7	743391	171733	328958
RMI EOP-GZip Processing (Sec)	0.0067	0.043	0.132	0.314	0.570
RMI EOP-GZip Transmission (Sec)	0.696	3.656	11.095	25.631	49.098
RMI EOP-GZip Total (Sec)	0.703	3.699	11.227	25.945	49.619
RMI OP (Sec)	1.916	13.201	42.814	102.398	188.549
RMI EOP-GZip Time Saving	63%	71%	73%	74%	73%

Table 2. RMI EOP-GZip detailed results for selected values of ArraySize.

ArraySize	10	20	30	40	50
Original Object Size (Bytes)	12835	88445	286855	686065	1263275
Compressed Object Size (Bytes)	4198.8	20475.6	60868.6	140270	268334.6
RMI EOP-PPM Processing (Sec)	0.248	3.388	19.18	49.52	96.98
RMI EOP-PPM Transmission (Sec)	0.626	3.056	9.085	20.936	40.050
RMI EOP-PPM Total (Sec)	0.874	6.444	28.265	70.456	137.03
RMI OP (Sec)	1.916	13.201	42.814	102.398	188.549
RMI EOP-PPM Time Saving	54%	51%	33%	31%	27%

Table 3. RMI EOP-PPM detailed results for selected values of ArraySize.

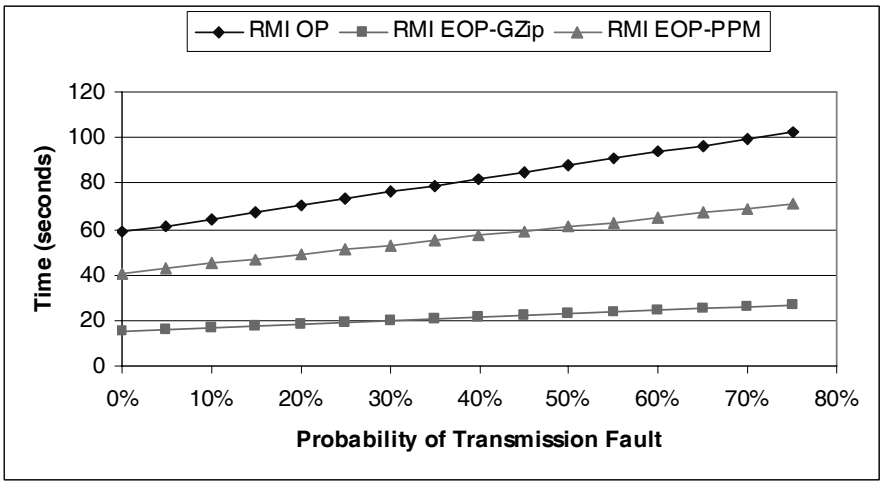

Fig. 4. Effect of unreliable channels, sampled over all test cases.

The figure shows that the total time in EOP-GZip is between 25% and 54% of the RMI OP time. EOP-GZip total time is between 34% and 80% of EOP-PPM GZip. Tables 2 and 3 give more detailed results.

These numbers are even better for RMI EOP in the cases of faulty channels and secure channels. We only show the results for the base test class, which contains all of the Java types mentioned earlier.

Faulty Channels Effect

When channels are faulty (and this is typical in wireless networks), a retransmission of the serialized object will be required. Figure 4 shows the effect of faulty transmissions on RMI EOP and RMI OP. The figure is a projection of Figure 3, averaging all cases of ArraySize. The x-axis represents the probability of fault occurrence up to 75%. It is evident from this sample that the faulty channel effect on RMI EOP-GZip

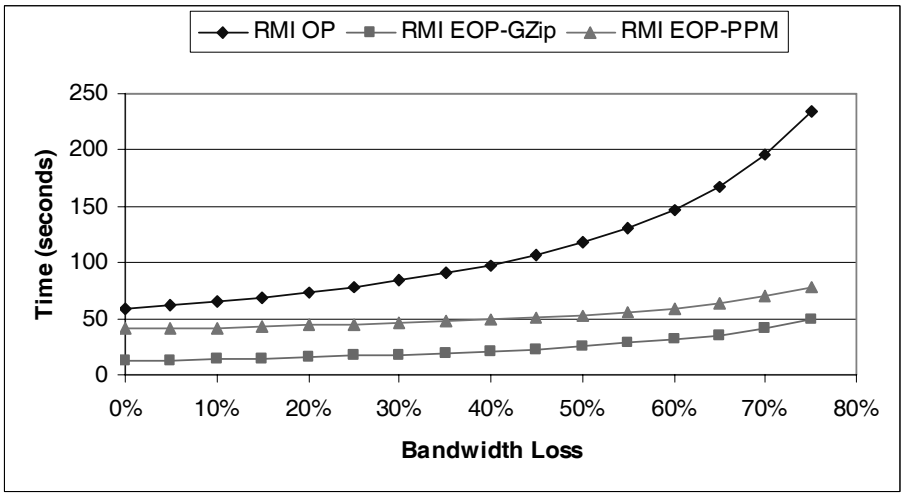

Fig. 5. Effect of throughput loss, sampled over all test cases.

is minimal. However, the high impact on RMI OP is clear. There is also a moderate impact on RMI EOP-PPM.

Bandwidth Loss Effect

When GPRS does not operate at its peak throughput, both versions of RMI EOP become even more attractive to use, as is illustrated in Figure 5. This figure is a projection of the measurements given in Figure 3. The x-axis represents the loss in network throughput, up to a maximum throughput of 25% of the peak performance.

Secure Channels Effect

Encrypting serialized objects inflates their size and, therefore, makes their transmission time higher. Hence, the use of compression is even more urging when using secure channels. Figure 6 summarizes experiments with RMI EOP using secure channels. It shows that the benefits of using RMI EOP over RMI OP are magnified when objects of the base class are encrypted.

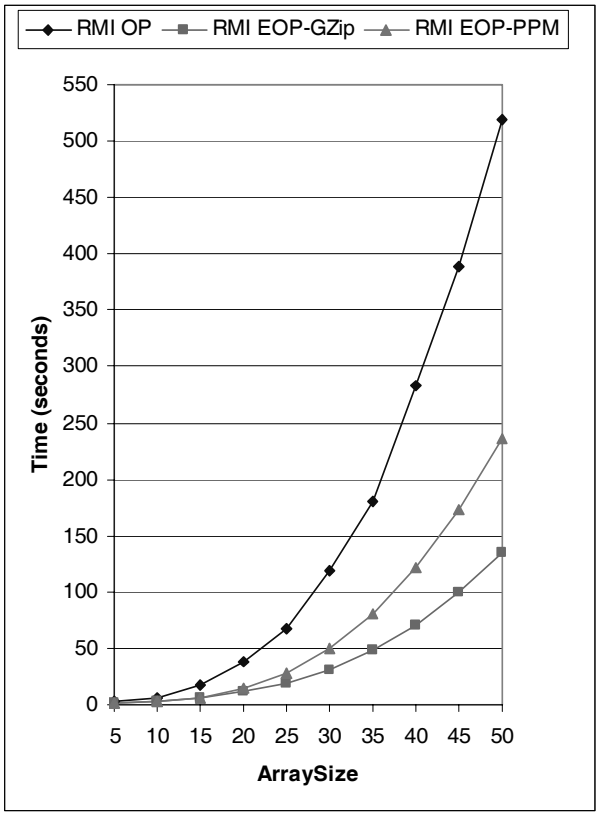

Fig. 6. Effect of secure channels for different values of ArraySize, using selected test cases.

5. Conclusions

We have introduced a new J2ME RMI package, which makes use of object compression in order to minimize the transmission time. The package, called RMI EOP, also supports secure channels. The currently existing RMI package (RMI OP) for wireless devices does not provide either of these crucial features.

In GPRS networks, the total time needed in RMI EOP to compress, transmit, and decompress the object is substantially lower than the time required to transmit an uncompressed object using RMI OP. This conclusion is valid even under extreme conditions that favor RMI OP. These conditions are: (1) peak GPRS performance,

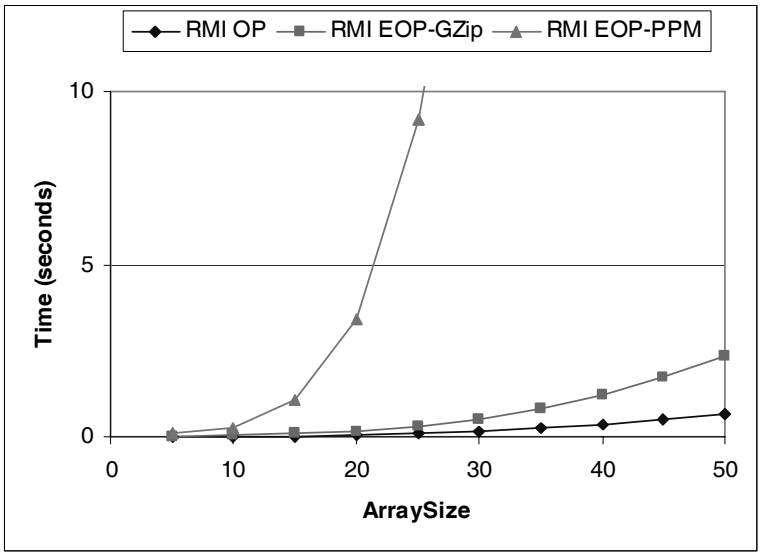

Fig. 7. Projection to 3G technologies with 1.5 Mb/s transmission rate.

(2) non-faulty channels, (3) insecure channels, and (4) both RMI communication ends are CDC devices. When these conditions are relaxed, the results are even more appealing.

RMI EOP makes use of GZip and PPM for object compression. PPM is time and memory intensive. RMI EOP supports three operation modes, one of which (slow) is only suitable for full-size server machines. In the other two modes (fast and medium), RMI EOP requires between 1MB and 30MB of RAM, respectively. This makes RMI EOP in its medium mode unsuitable for memory-constrained devices (Many handheld devices have a maximum of 32MB RAM). Some of our preliminary results on the use of LZMA show that it always outperforms PPM, in time and space.

Our Java object benchmark does not include objects that contain multimedia members, such as images. In such a case, we conjecture that PPM will outperform GZip. Therefore, finding ways to improve the running time of RMI EOP-PPM is crucial.

While this paper showed that RMI EOP outperforms RMI OP in GPRS networks, these conclusions may no longer be valid for 3G networks. For instance, the Universal Mobile Telecommunications System (UMTS) [27] may allow transfer rates close to 2 Mb/s. A projection of the results given in Section 4 to this new rate shows that, contrary to GPRS, RMI OP will outperform RMI EOP in 3G networks. Figure 7 projects the expectations for transmission rates of 1.5 Mb/s., and maintains the compression/decompression figures of Section 4. However, we believe that with 3G technologies, also faster processors and better compression techniques would be developed.

6. Acknowledgments

We are grateful for the anonymous referees (especially, referee #2) for their thorough and careful reading of the manuscript. Their suggestions were very valuable in preparing the final manuscript.

Preliminary results pertaining to the package RMI EOP-PPM were reported in "J. Kawash, G. Samara and A. El-Halabi. More Efficient Java RMI for GPRS Devices. In *New Trends in Computer Networks* (T. Tuğcu, M.U. Çağlayan, F. Alagöz, and E. Gelenbe, Ed.) (2005) pp. 144–153. Imperial College Press, London, UK".

References

[1] Mobile Monday Web Site, April 2005. `http://www.mobilemonday.com`.

[2] V. KRISHNASWAMY, D. WALTHER, S. BOHLA, E. BOMMAIAH, G. RILEY, B. TOPOL AND M. AHAMAD, Efficient Implementation of Java RMI. *Proc. Of the 24 USENIX Conference on Object-Oriented Technologies and Systems*, 1998.

[3] G. K. THIRUVATHUKAL, L. S. THOMAS, AND A. T. KORCZYNSKI, Reflective remote method invocation. *Concurrency: Practice and Experience*, 10(11–13):911–926, 1998.

[4] M. PHILIPPSEN, B. HAUMACHER AND C. NESTER, More Efficient Serialization and RMI for Java. *Concurrency: Practice and Experience*, (12)7:495–518, 2000.

[5] F. BERG AND C. D. PLOYCHRONOPOULOS, Java Virtual Machine Support for Object Serialization. *Proc. of the ACM 2001 Java Grande Conference*, pp. 173–180.

[6] C. NESTER, M. PHILIPPSEN AND B. HAUMACHER, A More Efficient RMI for Java. *Proc. Of the ACM 1999 Java Grande Conference*, pp. 153–159.

[7] J. MAASSEN, R. V. NIEWPOORT, R. VEILDEMA, H. BAL, T. KIELMANN, C. JACOBS AND R. HOFMAN, Efficient Java RMI for Parallel Programming. ACM *Transactions on Programming Languages Systems*, 23(6):747–775, 2001.

[8] D. KURZYNIEC, T. WRZOSEK, V. SUNDERAM AND A. SLOMINISKI, Experiments with Multi-Protocol RMI in Java. *Proc. of the ACM 2002 Java Grande Conference*, pp. 233.

[9] P. C. WEI, C. H. CHEN, C. W. CHEN AND J. K. LEE, Java RMI over Bluetooth. *Proc. of the ACM 2002 Java Grande Conference*, pp. 237.

[10] G. CHEN, M. KANDEMIR, N. VIJAYKRISHNAN, M. J. IRWIN AND B. MATHISKE, Heap Compression for Memory-Constrained Java Environments. *Proc. of the 18th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 282–301, 2003.

[11] G. CHEN, R. SHETTY, M. KANDEMIR AND N. VIJAYKRISHNAN, M. J. IRWIN AND M. WOLCZKO, Tuning Garbage Collection in an Embedded Environment. *Proc. of the 18th Int'l Symposium on High-Performance Computer Architecture*, 2002.

[12] G. CHEN, M. KANDEMIR AND N. VIJAYKRISHNAN AND W. WOLF, Energy Savings through Compression in Embedded Java Environments. *Proc. of the* 10th *International Symposium on Hardware/Software Codesign*, 2002.

[13] Y. MATSUOKA AND P. SCHAUMONT, K. TIRI AND I. VERBAUWHEDE, Java Cryptography on KVM and its Performance and Security Optimization Using HW/SW Co-design Techniques, *Proc. of International Conference on Compliers, Architecture, and Synthesis for Embedded Systems*, pp. 303–311, 2004.

[14] Java 2 Platform Micro Edition. `http://java.sun.com/j2me/index.jsp`.

[15] Data Compression, Retrieved October 1, 2004 from: `http://www.bambooweb.com/articles/l/o/Lossless_data_compression.html`.

[16] J. G. CLEARY AND I. H. WITTEN, Data Compression Using Adaptive Coding and Partial String Matching. *IEEE Transactions on Communications*, 32(4):396–402, 1984.

[17] `http://www.colloquial.com/ArithmeticCoding/`

[18] J. GAILLY, The gzip homepage. Retreived October 1, 2004 from: `http://www.gzip.org`.

[19] J. G. CLEARY, W. J. TEAHNAN AND I. H. WITTEN, Unbounded Length Contexts for PPM. *Computer Journal*, 40(2/3):67–75, 1997.

[20] D. SHAKRIN, PPM: One Step to Practicality. *Proc. of the IEEE Data Compression Conference*, pp. 2020211, 2002.

[21] Bouncy Castle Crypto Package. `http://www.bouncycastle.org`.

[22] N. M. DESHPANDE AND J. GILBERT, GPRS – How Does it Work and How Good Is It? *Intel Developer UPDATE Magazine*, October 2002. Available at `http://www.colloquial.com/ArithmeticCoding/`

[23] R. M. DAVIS, The Data Encryption Standard in Perspective. *Computer Security and the Data Encryption Standard*, National Bureau of Standards Special publication 500–27, 1978.

[24] R. RIVEST, A. SHAMIR AND L. ADLEMAN, A Method for Obtaining Digital Signatures and Public-key Cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[25] Forum Nokia, Series 80 Developer Platform 2.0 SDK for Symbian OS – For Personal Profile User's Guide, July 6, 2004.

[26] i-mate Pocket PC. `http://www.carrierdevices.com.au/products.php?id=3`.

[27] G. ELLIOTT AND N. PHILIPS, *Mobile Commerce and Wireless Computing Systems*, Addison Wesley, 2004.

Received: June, 2005
Revised: January, 2006
Accepted: February 2006

Contact address:
Jalal Kawash
Department of Computer Science
American University of Sharjah
P.O.Box 26666
Sharjah, UAE
e-mail: `jkawash@aus.edu`

JALAL KAWASH received his Ph.D. from The University of Calgary, Canada in 2000. He is currently an assistant professor of Computer Science at the American University of Sharjah, UAE and an adjunct assistant professor at The University of Calgary, Canada. His research interests are in distributed systems and algorithms, mobile collaboration, and computing education.

AHMAD EL-HALABI is currently an application support executive at Datel Systems and Software in Dubai Internet City. He graduated from the American University of Sharjah in 2004. His interests are in database design, mobile applications, and performance analysis.

GHASSAN SAMARA is currently a software engineer at Agilience ME FZ-L.C.C. in Dubai Internet City. He graduated from the American University of Sharjah in 2004. He is interested in developing Web-based and mobile applications and in performance analysis.