Design and Implementation of an Encrypted Mobile Objects Protocol (EMOP) for J2ME, J2SE and J2EE Applications

Sehlabaka Qhobosheane[a], **Mokakatlela Mokakatlela**[a], **Makhamisa Senekane**[b]

[a] Department of Mathematics and Computer Science, National University of Lesotho, Maseru, Lesotho
[b] Department of Electrical and Electronic Engineering, University of Cape Town, Western Cape, South Africa

Summary
The current trend in telecommunications is the movement towards mobility of devices, and as more mobile devices are being invented together with their associated applications, growing interest is on defining secure ways of communication for such applications. This paper describes the Encrypted Mobile Objects Protocol (EMOP), an object-oriented communications protocol designed to allow communication of objects between Java 2 Micro Edition (J2ME), Java 2 Standard Edition (J2SE), and Java 2 Enterprise Edition (J2EE) applications. The report discusses the theory behind the protocol, from the Mobile optimized object Description and Serialization (MooDS) protocol - also an object-oriented protocol dedicated to J2ME based phones, to the Secure Sockets Layer (SSL) protocol, and how EMOP was build from them. Moreover, the paper compares EMOP with MooDS in two J2SE applications. The results show that for objects less than 50 000 in number, MooDS transmission is an order of magnitude faster than that of EMOP because of extra encoding done by EMOP on objects for security purposes, but EMOP obtains best results in terms of reducing the application code size to sizes unattainable by MooDS, and consequently reducing the time-to-market of applications.

Key words:
Encryption, Mobile object, Communication Protocol, Object serialization.

1. Introduction

Most of the protocols used in mobile-device networks, such as the Mobile optimized object Description and Serialization (MooDS) protocol, an object-oriented protocol dedicated to Java 2 Micro Edition based devices, do not enforce security mechanisms on the transmitted data. The common approach for most of these protocols is to simply send the data over the Hyper Text Transfer Protocol (HTTP), but this only secures the transmitted data at the network level and avoids addressing the bigger issue of securing contents [17]. Also, MooDS does not allow the interaction of mobile objects between other Java technologies such as the Java 2 Standard and Enterprise Editions. The Encrypted Mobile Objects Protocol then comes into play to address the limitations of the aforementioned protocols for mobile device networks, and provide a step closer to a next generation network protocol of tomorrow.

The Encrypted Mobile Objects Protocol was engineered to address the following: manage communication of objects between J2ME, J2SE and J2EE applications; minimizing the binarization of message objects during communication between the sender and the receiver; address the lack of object serialization support in J2ME MIDP profiles; provide encryption and decryption of message objects; and encoding and decoding of message objects. All the above objectives were met except for the first objective which is only partially fulfilled. EMOP currently allows communication of J2SE and J2EE objects but does not support J2ME objects because of the SSL API (Application Programming Interface) used in EMOP which is not supported by the restrictive J2ME environment. Suggestions on attaining this objective are presented in the *recommendations and future work* subsection of the *conclusion*. This paper is organized as follows. Section 2 provides a basic explanation of EMOP's predecessor, the Mobile optimized object Description and Serialization (MooDS), explaining its purpose and functionality, as well as its shortcomings. The section then ends by discussing the Secure Sockets Layer protocol (SSL). Section 3 is about the entire development of the EMOP protocol, from the methodology followed to how it was implemented. Section 4 follows thereafter and provides a performance comparison between EMOP and MooDS in two example applications. Section 5 concludes the paper with general observations on EMOP and MooDS, ending with recommendations for future work on EMOP.

2. Overview of MooDS and SSL

2.1. MooDS

MooDS (Mobile Optimized Objects Description and Serialization) is an Object-Oriented Communication protocol dedicated to Java 2 Micro Edition (J2ME) multiplayer games [16]. It is used in GASP, an open

Manuscript received May 5, 2011
Manuscript revised May 20, 2011

source middleware enabling J2ME multiplayer gaming interactions. MooDS takes care of the specific constraints imposed by J2ME devices over second generation (2G), 2.5G, or the third generation (3G) cellular phone networks.

MooDS uses a basic XML Schema syntax to describe message types to be used [16]. Thus, in order to use MooDS, the developer has to describe the messaging data structures chosen for his/her multiplayer game using MooDS description syntax. In other words, the message ought to be in XML schema format.

There are three distinct steps in the MooDS approach: [16]

- Firstly data messages are specified in a description file (XML format).
- The stub/skeleton code (java classes) is generated upon compilation of the description file.
- The generated code (java classes) is embedded in the wireless application package. Message objects to be sent are encoded over the network using the static encoder/decoder methods. The receiver decodes the binary streams and gets an object copy of the original message object.

2.1.1. Strengths of MooDS

- Minimizing the binarization of message objects during communication between the sender and the receiver.
- Addressing the lack of object serialization support in J2ME Mobile Information Device profiles (J2ME MIDP).

2.1.2. Weaknesses of MooDS

- Limited security, hence there is no data encryption or decryption.
- Oriented to games (GASP Platform), thus MooDS can only work in a J2ME environment.
- MooDS does not allow the interaction of mobile objects between other Java technologies such as the Java 2 Standard and Enterprise Editions.

2.2. Secure Sockets Layer (SSL)

The SSL Handshake Protocol was developed by Netscape Communications Corporation to provide security and privacy over the Internet [17]. It is an important piece of the overall puzzle of system security, providing the much needed network security. Other protocols also exist but none has achieved the same level of adoption. It is also an excellent example of using basic cryptography and Public Key Infrastructure (PKI) to meet higher level system security needs [8]. Equally important, SSL supports server and client authentication, and it is application independent, allowing protocols like HTTP (Hyper Text Transfer Protocol), FTP (File Transfer Protocol), and Telnet to be layered on top of it transparently. The SSL protocol is able to negotiate encryption keys as well as authenticate the server before data is exchanged by the higher-level application. In a nutshell, SSL protocol maintains security and integrity of the transmission channel by using encryption, authentication and message authentication codes. SSL is widely adopted and has become the de-facto mechanism to secure the exchange of sensitive information over the Internet [1, 8, 9].

2.2.1. Strengths of SSL

- Application independence: The SSL protocol is application independent, allowing many protocols like HTTP (Hyper Text Transfer Protocol), FTP (File Transfer Protocol), and Telnet to be layered on top of it [17].
- Encryption Keys: The SSL protocol is able to negotiate encryption keys as well as authenticate the server before data is exchanged by the higher-level application. It uses both the public and secret key encryption algorithms for authentication and data integrity [5].

2.2.1. Weaknesses of SSL

- Security depends on key generation: The randomness used in the process of generating a key decides the strength of the resulting key [2]. Creating truly random numbers on a deterministic device such as a computer is impossible. In order to get some strong source of randomness, access to hardware sources is required. For example, strong sources of randomness in a computer may include thermal noise and some radioactive decay source.
- Limited security in J2ME: Creating good random numbers in a constrained environment such as a cellular phone is truly a challenge, hence why the traditional SSL implementation is not supported in J2ME devices.

3. System Design and Implementation

3.1. Protocol Description

EMOP is intended to manage communication of objects between J2ME, J2SE and J2EE applications. The motivation behind the development of the protocol is the same as the development of the open source MooDS protocol, which is; minimizing the binarization of message objects during communication between the sender and the

receiver, and the lack of object serialization support in J2ME MIDP profiles. However, EMOP is intended to have additional features of security by encryption and decryption of message objects, and to allow cross platform communication of java objects. In order to use EMOP the same principle as in MooDS is followed, the developer has to describe the messaging data structures in XML Schema and then the descriptions are parsed by the EMOP generator in conjunction with Java Architecture for XML Binding (JAXB) to obtain a serialization class version of the data structure. The message objects are sent by value, thus authorizing only Java primitive type fields (Boolean, byte, short, int, long, String and array).

3.2. Methodology

The eXtreme Programming (XP) agile software development methodology was followed when implementing EMOP, because of XP's focal values of:

i. *Individuals and interactions* over processes and tools.
ii. *Working software* over comprehensive documentation.
iii. *Customer collaboration* over contract negotiation.
iv. *Responding to change* over following a plan.

Following XP, EMOP's development went through the lifecycle as depicted in figure 1.

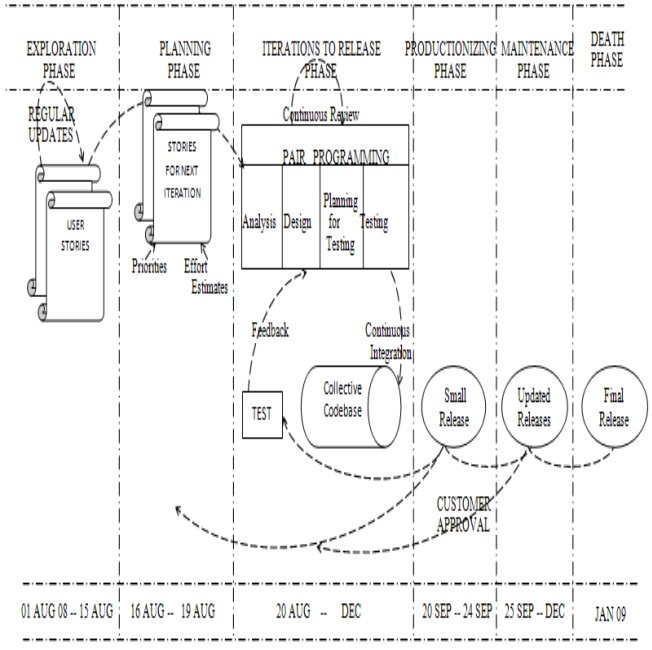

Figure 1: EMOP's Lifecycle

3.3. System Design

3.3.1 Architectural Design

Since EMOP is a network protocol and its central feature is security, the architectural design that was adopted when designing it is the layered architectural approach. In the layered model, a system is organized into layers, each of which provides a set of services [18]. As can be seen from Figure 2, security is implemented in the innermost layer (the SSL layer) while the EMOP layer provides the parsing of user messaging data-structures into a serialization class version, and the basic network functions (socket communications) are handled in the TCP/IP layer. The layered approach was also followed because it supports the incremental development of systems inherent in XP, and thus, it can be viewed as an agile architecture. Besides the complexity and difficulty of structuring systems in the layered manner, another disadvantage of the layered approach is the issue of performance. If there are many layers, a service request from a top layer will have to be interpreted several times in different layers before being processed, thus reducing the system's performance. However, since security was of highest priority and EMOP was designed not to have many layers, it was decided to adopt the layered model and compromise a bit on performance. Statistical details of how exactly performance was compromised are presented in section 4 on *EMOP versus MooDS*.

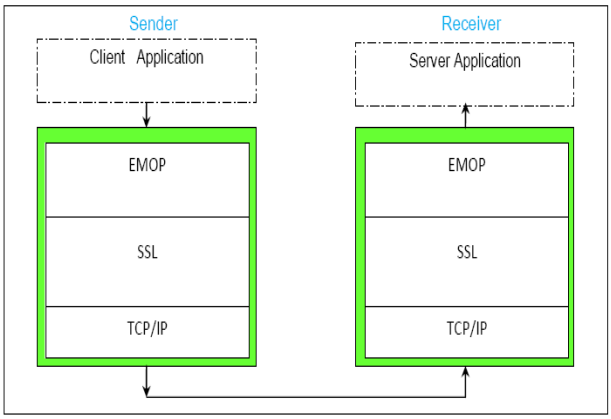

Figure 2: Architectural Design

After the design of an overall system organization, the subsystems were then decomposed into modules as discussed in the following subsection on module interactions.

3.3.2. Module Interactions

In the object-oriented approach, Sommerville (Software Engineering 8e, 2007) describes modules as objects with private states and defined operations on those states, and may be implemented as sequential components or as processes. Figure 3 follows this description and depicts major classes of EMOP as modules which interact in the manner as illustrated.

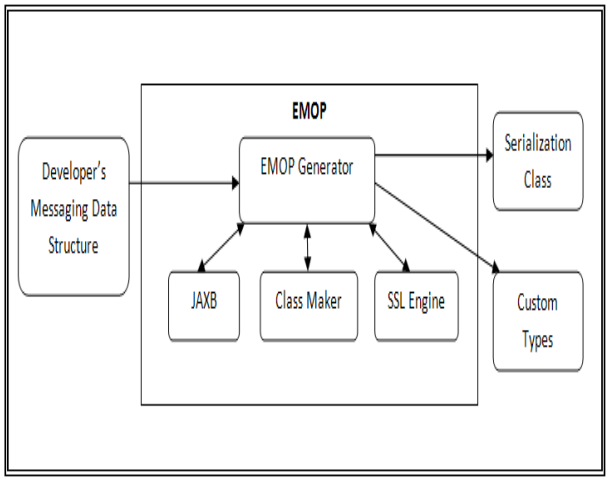

Figure 3: Module Interactions

The EMOP Generator is the entire logic of EMOP and it transforms the developer's messaging data structure into a serialization class version and a CustomTypes class which defines the communication methods of the serialization objects, in accordance with Java Architecture for XML Binding (JAXB), the Class Maker, and the SSL engine for security. JAXB decodes the XML schema of the data structure into java objects (with associated attributes) and together with the Class Maker module, through the coordination of EMOP Generator, produce the Java serialization and CustomTypes classes.

3.4. Implementation

3.4.1. Programming Platform

Being an open-source project, EMOP was developed on a Fedora Core 8 Linux box, and was also tested on Red Hat Enterprise and on Windows XP. The programming was done using the Eclipse Ganymede Integrated Development Environment (IDE) because of its usability and familiarity to the EMOP team, and because of its support for the ANT build tool. ANT was chosen as the preferred build tool over Make because of its simplicity and XML format which make it portable and also easy to integrate with Java. On the other hand, makefiles are basically a list of shell commands, and as a result, it is next to impossible to write portable makefiles. It is also very difficult to integrate makefiles with Java as compared to integrating build.xml with Java. JDK 6 updates 6 to 12 are the Java Virtual Machines (JVM's) that EMOP was compiled under.

3.4.2. Class Interactions

From the protocol description in section 3.1, it can be seen that, like MooDS, the overall essence of EMOP is to parse the developer's messaging data structures coded in XML schema and to generate serialization class versions of those data structures, but with embedded security, and between different Java technologies. Thus, instead of re-inventing the wheel, EMOP developers stood on the shoulders of giants like Romain Pellerin and Lim Chanty (MooDS developers) and transformed MooDS into EMOP according to the design considerations in section 3.3.1. MooDS source code (latest version then, v2.0.1) was downloaded from *http://download.forge.objectweb.org/gasp/MooDSv201.zip* and imported as a Java project into Eclipse IDE. It was then converted into EMOP with java class interactions as depicted in Figure 4.

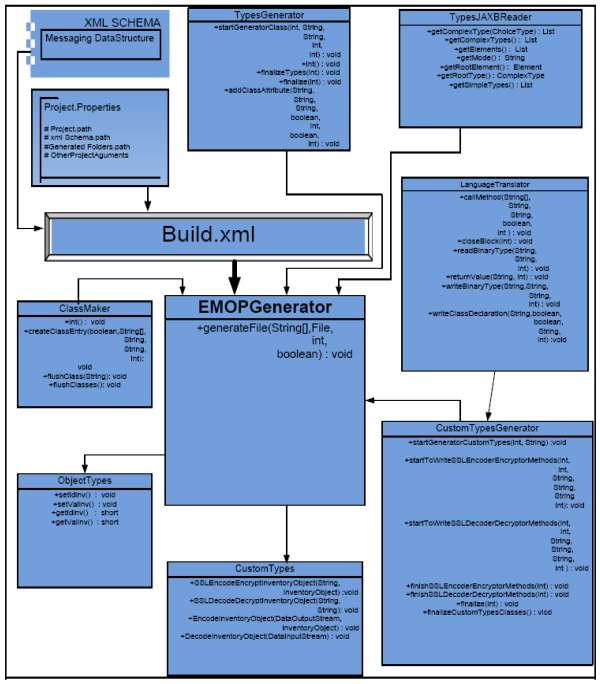

Figure 4: Class Interactions

Project.properties is a file containing all the initialization information of the project, such as the location of EMOP home directory, the location of the messaging data

structures' XML schema file, JDK directory, and SSL parameters. This information acts as input to the build.xml ANT file.

```
# Home path
home.path = C:/Documents and Settings/CS5402/workspace/EMOP(Student-06-11-08)
# JDK Home
jdk.home = C:/Program Files/Java/jdk1.6.0_10
# Developer's data structures in XML Schema (saved in home.path/resources/schemas)
user.xml.schema = student.xsd
# Folder name that will contain the generated tools. This name is also the package name for the Java generated classes
generated.folder.name = Emop_Generated_Types
# War and jar file name
deploy.filename = project_archive
#-----------------------------------------------------#
#---------- SSL Protocol arguments----------#
#-----------------------------------------------------#
# 1. Your name and lastname
developer.name = emop
# 2. Your Organizational Unit name
org.unit = macs
# 3. Your Organization name
org.name = nul
# 4. Your City or Location
developer.city = Roma
# 5. In which State or Province?
developer.state = Maseru
# 6. In which Country?
developer.country = Lesotho
# 7. KeyPass Password
developer.keyPass = explorer
# 8. StorePass Password
developer.storePass = benetoni
# 9. StoreType
developer.storeType = JCEKS
# 10. Key Algorithm
developer.keyAlg = RSA
# 11. Key Store Name
developer.keyStore = server.ks
# 12. Certificate Name
developer.certName = temp$.cer
# 13. Client Key Store Name
developer.clientkeyStore = client.ts
```

Figure 5: Project.properties file

Build.xml is an ANT file consisting of all targets necessary to build, compile, run, and manage the project. It takes input from the project.properties file and the developer's schema, and it then runs the EMOP engine (Generator.java) with the necessary arguments. It also prepares for security by generating relevant keys from the project.properties' SSL information using the keytool utility

The EMOP engine, Generator.java, is the project's main class. It is the one which controls the entire logic of EMOP protocol. It parses the developer's XML schema in accordance with JAXB thus creating a secure serialization class version of the data structure through the LanguageTranslator, ClassMaker and CustomTypesGenerator classes. Two types of classes are generated, the object class specific to the messaging data structure, e.g. for a book data structure, a book object with private attributes such as author, publisher, etc, and the public setter and getter methods which act as interfaces to the object attributes. The second type of classes is the client and server CustomTypes classes which contain methods for communicating the object classes. The root methods of the CustomTypes classes are the sslEncodeEncryptObject and the sslDecodeDecryptObject methods (whereby Object stands for the name of the data structure being sent). The sslEncodeEncryptObject method takes as arguments, a hash table of objects to be communicated, the IP Address of the server as a string value, and an integer port number of the specific server service being used. It then encodes the hashtable objects and transmits them to the receiver/server through the Secure Sockets Layer (thus applying encryption and authentication). On the receiver side, the sslDecodeDecryptObject method takes as inputs, the name of the server key store, the keystore password, and the port number of the server service being processed, and it then receives the sent secure bytes, authenticates (through SSL and the keystore name and password) and decrypts them, and finally decodes the binary streams into the specific objects sent. It returns a hashtable of the objects to its calling environment.

4. EMOP versus MooDS

To complete the EMOP engineering process, the performance and capabilities of EMOP were compared to those of the mother protocol, MooDS, through a couple of example applications. In the first example, the messaging data structure being transformed was that of a book object.

4.1. Book Messaging Data Structure

In this simplified analysis example, a client application communicates book objects to the server and a record of total bytes sent, together with the time taken by each protocol, is made. A book object consists of the title, author, publisher, ISBN, and totalPages attributes, all of which are string values except for the totalPages attribute which is an integer. The client and server applications were on two different workstations but in the same subnet so as to test the efficiencies of both protocols under real network conditions. Firstly, for each protocol, one book

object was sent over the network five times and the results were recorded. The average of the recorded results (i.e. total bytes sent, and time taken) was then taken as the final results obtained for each protocol. The same procedure was repeated for 200, 2000 and 50 000 book objects, and their results were also recorded in a similar manner. The fivetimes approach was followed so as to eliminate network effects which might hamper the results. The Java technology that was thus being worked on is the J2SE (Standard Edition). Figures 6 and 7 illustrate the recorded results.

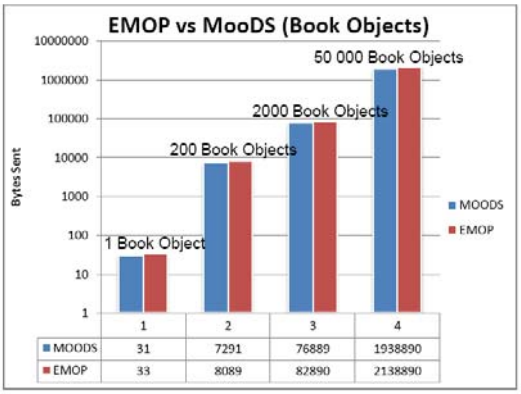

Figure 6: Book Objects – Bytes Sent

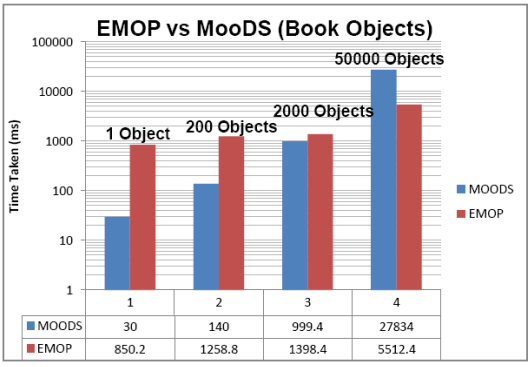

Figure 7: Book Objects – Time Taken

Book Object Discussion

In the bytes sent figure, Figure 6, it can be seen that, for the same number of book objects, EMOP sends relatively more bytes than MooDS. This is due to the fact that EMOP adds an extra layer of security (the Secure Sockets layer) to the objects thus increasing the number of bytes for each object. MooDS, on the other hand, does not do this, hence the reduced number of bytes for objects using MooDS. Technically, this increase in bytes by objects communicating in EMOP means that EMOP does more encoding to the objects than MooDS in order to provide for the desired security. In like manner, this result leads to a prediction that EMOP should take a bit more time to communicate the objects than should MooDS and this is precisely what happens as depicted in figure 7. There is, however, another interesting observation from figure 9, which is that as more objects are communicated, EMOP tends to outperform MooDS in the amount of time each takes for the transmission.

4.2. Student Messaging Data Structure

In this example, a client application now communicates student objects to the server and as in section 4.1 above, a record of total bytes sent, together with the time taken by each protocol, was made. A student object consists of the name, surname, faculty, department, and program attributes, all of which are string values. The same setting and procedure as in the book object example was followed and figures 8 and 9 illustrate the recorded results.

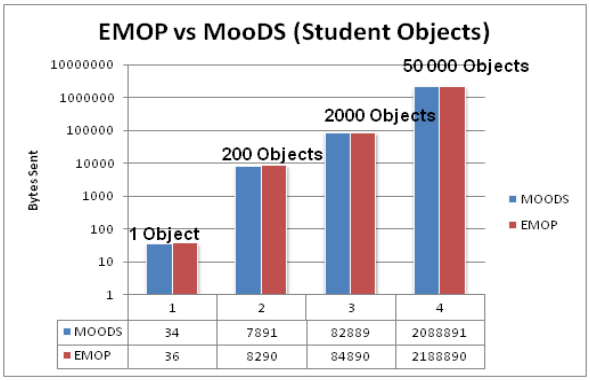

Figure 8: Student Object – Bytes Sent

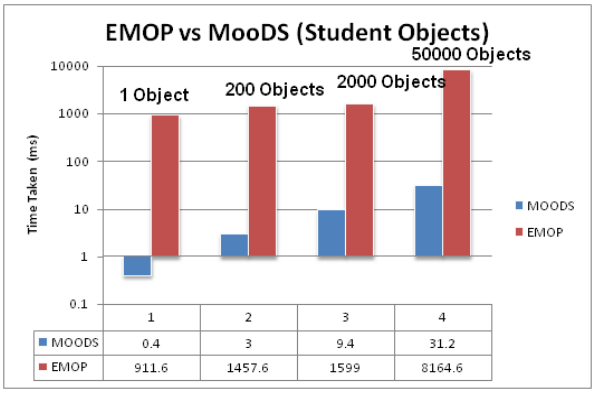

Figure 9: Student Object – Time Taken

Student Object Discussion

In the bytes sent figure, the same observations that were made in the book objects example also apply to the student objects example, namely, the fact that EMOP adds more encoding to the objects than MooDS, due to the added security feature. The second observation is even more interesting as it refines the discussion made in the book example. Instead of EMOP outperforming MooDS when transmitting many (50 000) objects as was expected from the book example discussions, EMOP's performance seems rather not to be influenced much by the number of objects sent as MooDS clearly is. The observation here is that in the graph of objects sent versus time taken, MooDS has a far steeper slope than EMOP thus making EMOP a preferred protocol especially when very many objects are to be communicated.

4.3. Client-Code Size

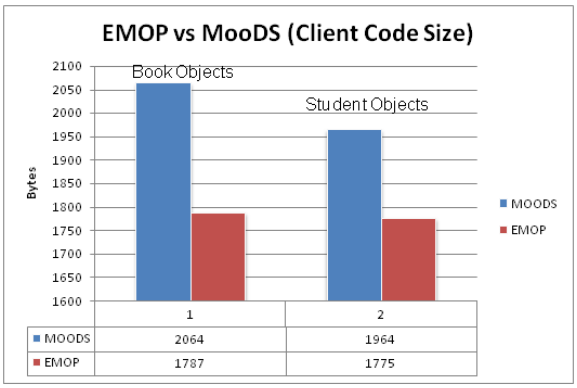

Figure 10: Client Code Size

Another crucial comparison between EMOP and MooDS is that of the client-code size. Both protocols were designed to simplify developers' lives by reducing the time-to-market of the developers' applications since they require the developer to only provide a description of the messaging structure he/she wishes to communicate and leave the rest to the protocols, which are produced after a click of a button. The developer need only know basic XML and the protocols generate the somewhat involved custom types in Java. However, the developer also has to write the client and server applications which will communicate the data, and EMOP has outperformed MooDS in this regard since it requires developers to write less client (and server) code for the applications. This is due to the fact that EMOP includes almost all of the communication information in the generation of the custom types while MooDS does not (thus leaving the developer with the tedious task of socket programming). The sizes of the client applications for the book and student objects, for both protocols, were recorded and the results are illustrated in figure 10.

5. Conclusion

5.1. Observations

- Objects transmitted by EMOP become generally larger in size than when encoded with MooDS. This is because EMOP has an added layer of security which adds SSL parameters to the objects, thus increasing the total number of bytes for the transmitted objects.
- The increase in size of objects transmitted through EMOP makes EMOP perform less than MooDS in terms of total time taken for objects transmission, especially when few objects are being transmitted simultaneously.
- Also, EMOP has been designed to reduce the size of the client code to sizes unattainable by MooDS by including socket programming details into EMOP and relieving the developer of that hassle. This invariably reduces the time-to-market of the developer's application, and introduces another approach to rapid application development.

5.2. Recommendations and Future Work

The work presented in this study on EMOP is not complete. The SSL API (Application Programming Interface) used is that of the Standard Edition (J2SE), and it functions equally well for the Enterprise Edition (J2EE) but poses problems for the Micro Edition because of J2ME's restrictive environment. The next step in EMOP enhancement would be to replace the SSL API with one compatible with J2ME but without impacting on the security currently afforded by EMOP. Fortunately, EMOP has been designed in such a way that corrective maintenance and scalability are easy to achieve, thus modifying a few lines of code and replacing the API should suffice. The following task would then be to enhance EMOP's security by adjusting it to support mutual (client and server) authentication with central authority signed certificates, unlike now where it supports sever authentication with self-signed certificate. The probably most important work that can also be done on EMOP, after the above recommendations, would be to enforce real-time constraints on EMOP thus transforming it into a Real-Time Encrypted Mobile Objects Protocol (RT-EMOP). This would, undoubtedly, require the development platform to be a real-time operating system (RTOS), and would require much emphasis and analysis on network traffic algorithms for improved timing responses. The choice of programming language used would also be of crucial importance since high level languages like Java currently have no standard ways of accessing the system hardware, and most do not allow for

detailed run-time space and processor analysis as compared to low level languages like C. Java, however, has a real-time version; the J2ME, but unfortunately J2ME has a disadvantage on features a system can have due to its restrictive nature (of limited resources), and is mostly intended for hand-held devices.

Acknowledgments

The authors would like to thank their mentor and supervisor, Mr. Motlatsi Seotsanyana, for his expert advice and guidance on how best to conduct a software development project. Many thanks also extend to Romain Pellerin and Lim Chanty, the MooDS development team, on whose shoulders the authors had to stand for EMOP to be realized. Finally, the authors are indebted to the Department of Mathematics and Computer Science (MACS) at the National University of Lesotho for providing them with all resources necessary for the successful completion of this work.

References

[1] S. J. Rees Adel Aneiba, Mobile agents technology and mobility, 2004.
[2] Developer Works, J2me: Step by step. IBM, IBM, p36.
[3] Sabastian Fischmeister, Building secure mobile agents: The supervisor-worker framework. Information Systems, Technische Universitat Wien, February 2000.
[4] Elliotte Rusty Harold, Xml 1.1 bible, 3rd ed. Wiley, 2004.
[5] Vebjrn Moen Kent Inge Simonsen and Kjell Jrgen Hole, Attack on sun's midp reference implementation of ssl. 2005, p12.
[6] Jack Koftikian, Simple object access protocol(SOAP). Information and communication technology, Technische Universitat Hamburg-Hamburg, November 2001.
[7] James F. Kusose and keith W. Ross, Computer networking: a top-down approach featuring the internet. Wiley Publishing, Inc, 2000.
[8] RSA Laboratories, Laboratories' frequently asked questions about today's cryptography, version 4.1. May 2000.
[9] Danny B. Lange, Mobile objects and mobile agents: The future of distributed computing?, 1998, p12.
[10] David Lowery, Utilization of web services to improve communication of operational information, Master's thesis. Naval Postgraduate School Monterey, September 2004.
[11] B. McLaughlin, Java and xml data binding. O'Reilly May 2002.
[12] Brett McLaughlin, Java and xml, 2nd ed. O'Reilly, September 2001.
[13] Sun Microsystems, The java ee5 tutorial. Sun Microsystems, Inc, September 2007.
[14] Parixit Dilip Parekh, Java applet based database management interface. Computer science, The Florida State University College of Arts and Science, 2006.
[15] Jussi Ronkainen Pekka Abrahamsson, Outi Salo and Juhani Warsta, Agile software development methods, review and analysis. VTT, 2002.
[16] Romain Pellerin, The moods protocol :a j2me object-oriented communication protocol. 2007, p8.
[17] Eric Rescorla, Ssl and tls: designing and building secure systems, ch. 6. Eddison- Wesley Professional, October 2000, p528.
[18] Ian Sommerville, Software engineering 8,eigth ed. Eddison-Wesley, 2006.
[19] Brain Suda, Soap web services. Computer science, University of Edinburgh, School of Informatics, 2003.
[20] Dreamtech Software Team, Cracking the code, wireless programming with j2me. Hungry Minds, Inc., 2002.
[21] Kim Topley, J2me in a nutshell, a desktop quick reference. O'Reilly, March 2002.
[22] Tom Karygiannis Wayne Jansen, Nist special publication 800-19 mobile agent security. National Institute of Standards and Technology (NIST) Special Publications (800 Series) (1999), no. 800-19, p42.
[23] Madeleine Wright, A detailed investigation of interoperability for web services, Master's thesis. Rhodes University, December 2005.

Sehlabaka Qhobosheane received the B.Eng. degree in Computer Systems and Networks Engineering from National University of Lesotho in 2009. He is now completing his thesis in MSc.Eng Biomedical Electronics at Stellenbosch University and is expected to graduate in December 2011. He currently works as a Software Engineer at iThemba LABS Medical Radiation in Cape Town South Africa.

Mokakatlela Mokakatlela received the B.Eng. degree in Computer Systems and Networks Engineering from National University of Lesotho in 2009. He now works as a Software Engineer at Computer Business Solutions (CBS) in Maseru Lesotho.

Makhamisa Senekane received the B.Eng. degree in Electronics and MSc.Eng degree in Electrical Engineering from National University of Lesotho and University of Cape Town in 2007 and 2011 respectively. He now plans on pursuing a PhD in Quantum Cryptography at the University of Kwazulu Natal in South Africa.