The MooDS protocol: a J2ME object-oriented communication protocol

Romain Pellerin
CNAM-CEDRIC, 292, rue St Martin, 75141 Paris Cedex 03, Tel: +33 1 58 80 85 13 France
GET-INT, 9 rue Charles Fourier, 91011 Evry Cedex, Tel: +33 1 60 76 45 61 France
romain.pellerin@cnam.fr

ABSTRACT

This paper describes *MooDS*, an object-oriented communication protocol dedicated to *Java 2 Micro Edition* (J2ME) based mobile phones. This protocol is used in *GASP*, an open source middleware enabling J2ME multiplayer gaming interactions. This paper enlightens the difficulty in developing mutiplayer games in heavily constrained environments, like the J2ME CLDC platform: small application memory footprint, lack of object serialization support, network protocol limitations, small bandwith and lack of client IP addressing support within cellular phone networks. This paper details how MooDS takes care of these constraints by providing an efficient object serialization protocol in order to reduce the amount of transmitted data and to increase the communication speed. Moreover, MooDS is compared with the available J2ME SOAP based protocol implementations, kSOAP2 and JSR172. This paper shows that MooDS obtains the best results in terms of encoding length, transmission delay, memory allocation and application code size, thus supplying a proper gaming interaction support, in particular for time critical multiplayer games.

Categories and Subject Descriptors

C.2.2 [**Network Protocols**]: Application; D.2.3 [**Coding Tools and Techniques**]: Object-oriented programming

General Terms

J2ME communication protocol

Keywords

Object-oriented, serialization, communication protocol, multiplayer games, J2ME mobile phone, MooDS, GASP, OMA

1. INTRODUCTION

The GASP[9][29], *GAming Services Platform*, middleware is a common project of CNAM-CEDRIC and GET-INT laboratories. Its objective is to provide an open source Java middleware for the development of multiplayer games for *Java 2 Micro Edition*[11] (J2ME) phones based on *Connected Limited Device Configuration*[4] (CLDC) profiles, such as *Mobile Information Device Profile*[16] (MIDP) and *NTTDocomo Java*[6] (Doja). The middleware implements the *Open Mobile Alliance Games Services*[18] (OMA GS) specifications v1.0. The client-server communications in GASP is handled by a specific protocol named MooDS, for *Mobile optimized object Description and Serialization*. The constraints of J2ME CLDC profiles and current cellular phone networks generally impose a particular design for multiplayer games, compared to PC-based multiplayer games. Furthermore, when dealing with object-oriented message communications in J2ME multiplayer games, additional constraints arise: small application memory footprint, lack of serialization support, limited network bandwidth, pay-per-byte policy of cellular operator. MooDS covers up these constraints by providing an efficient object serialization protocol to support time critical multiplayer games requiring low transmission delay and bandwidth. Besides, it aims at sympliflying the development process of the multiplayer game communication part offering through generated codes: server skeleton, message communication model management and persistency services. This paper presents first a brief overview of communications in GASP middleware. Section 3 enlightens the J2ME and network object serialization constraints in the context of object-oriented communications. Section 4 gives an overview of the existing object-oriented communication protocols available for J2ME devices. Section 5 presents the MooDS protocol, especially its serialization mechanism and binary encoding structure. It also describes the game and persistency services offered by the protocol to simplify the development of multiplayer games. In Section 6, MooDS is compared with the available J2ME *Simple Object Access Protocol*[21] (SOAP) implementations, *kSOAP2*[15] and *JSR172*[14]. The last section concludes and gives an overview of the future works on MooDS.

2. COMMUNICATIONS IN GASP MIDDLEWARE

GASP middleware enables the creation of J2ME multiplayer games running over GPRS (2G), EDGE (2.5G) and UMTS (3G) cellular networks. The HTTP protocol is the basis of J2ME client-server communications[29][27]. This is due to the Socket communication mechanism restrictions imposed by cellular operators to control their network. Another constraint is the lack of mobile phone IP addressing on 2G and 2.5G networks, which prevents server to client communications. The direct consequences for communications

model are: i) the game events must be stacked on the server side, ii) clients must do periodic requests to get the stacked server side events. For these reasons, the GASP middleware server side is implemented over *Apache Tomcat*[22] servlet container to handle HTTP client requests. As depicted in figure 1, GASP offers communication interfaces for game client and server logics, named respectively *GASPClient* and *GASPServer*. These interfaces, based on OMA GS specifications, must be extended by game logics to run on GASP and provide methods to use game services, such as login, lobby, game joining and in-game communications. The OMA GS in-game communications are object-oriented and thus does require object serialization to send data over the network. However, object serialization for J2ME CLDC profiles is not supported.

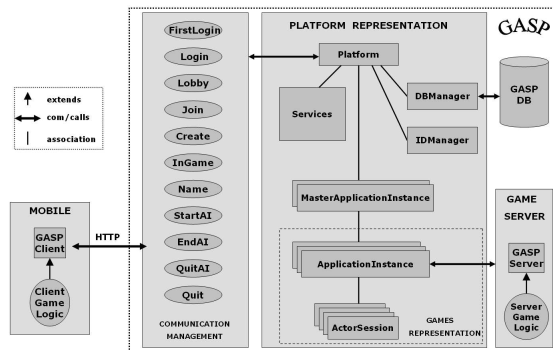


Figure 1: GASP online architecture

3. CONSTRAINTS OF OBJECT-ORIENTED COMMUNICATIONS IN J2ME MULTIPLAYER GAMES

The lack of mobile phone IP addressing support implies a particular design for client-server communications in J2ME multiplayer games. However, additional constraints have to be considered in order to establish object-oriented message communications: J2ME API restrictions, cellular network capabilities and multiplayer game needs.

In J2ME CLDC profiles, Java APIs have been reduced and do not provide support for object serialization, which hinders object-oriented communications. In Doja profile, iApplications are limited to 30 kilobytes in the 1.5 version and 100 kilobytes in the 2.5 release. For MIDP profile based devices it depends on the mobile phone's memory. Memory footprint restrictions have an heavy impact on object-oriented message communications: in a straightforward implementation, each message type will correspond to a Java class, requiring at least 800 bytes (size of an empty class). Thus the more message types used, the less memory footprint is available to the game engine.

The network bandwidth of cellular phones networks is quite low, about 20 kilobytes per seconds in 2G, 59 kilobytes per seconds in 2.5G and about 384 kilobytes per seconds in 3G. Consequently, an object-oriented communications approach must provide an efficient object serialization, particularly on 2/2.5G networks, to increase communication speed. In addition, a lightweight serialization is required to limit the amount of transmitted data as many cellular phone operators apply a billing system based on a pay-per-byte policy.

Finally, latency and bandwith are the two major issues to make compelling multiplayer games[31], particularly in time critical multiplayer games, such as *First Person Shooters* (FPS), *Real Time Strategy* (RTS) and action-based *Massively Multiplayer Online Role Playing Game* (MMORPG) games. The optimization of communications as a lightweight message oriented protocol is mandatory to decrease communication delay and to limit the amount of transmitted data.

4. EXISTING J2ME OBJECT-ORIENTED COMMUNICATION PROTOCOLS

On current J2ME MIDP phones, client-server communications can be established using SOAP protocol. Two implementations are available: the Sun implementation called JSR172 and the kSOAP2 implementation used in various *Web Services*[23] projects.

4.1 The kSOAP2 API

Due to the lack of web services and XML support in J2ME in 2001, the kSOAP project has been launched to propose an API for J2ME phones, enabling Web Services access using SOAP. kSOAP was born in the context of *Enhydra*[7], a project working on open source Java/XML applications, Web Services and workflow servers. In 2003, kSOAP2, a complete redesign of the first version of kSOAP, has been released as an independent project. kSOAP2 offers XML data parsing, data object serialization and remote Web Services invocation. It has a Java-oriented approach to describe messages that can be transfered through SOAP requests. In addition, it handles all primitive Java types and supports the *Vector* Java class to manage object list. Finally, it enables user defined serializable objects and also allows their registration to kSOAP2 class map. If a developer wants to use the kSOAP2 API, which takes up about 41 kilobytes, he must embed it in the package of the wireless application.

4.2 The Sun JSR172 API

The JSR172 API has been released by Sun in 2004 coming as an optional API of J2ME MIDP v2.0. It is a subset of the *Java API for XML-based RPC*[13] (JAX-RPC) 1.1, a *Java 2 Standard Edition*[12] (J2SE) API. In fact, in order to ensure a low memory footprint, the functionalities of the API have been reduced. It only offers the *document/literal* SOAP encoding style, the literal representation of a RPC call. It handles the XSD primitive types (same as Java primitive types), as well as complex types and arrays of primitive/complex types. However, it does not support some XSD tag like *xsd:choice* tag. The JSR172 serialization approach is the same as JAX-RPC, the user needs: to describe its Web Services application in the *Web Services Definition Language*[24] (WSDL) file, to use a stub/skeleton generator and to generate the marshalling/unmarshalling classes corresponding to the described types. Then he must embed these generated classes in the wireless application package to enable data object serialization during SOAP calls.

4.3 SOAP protocol limitations

SOAP is a XML based communication protocol, in other words a text based protocol, which encodes message data structure descriptions. As a result, such a protocol requests a lot of communication bandwith, increasing transmission

delays. In the case of time critical applications, such as real time games, this would lower game responsiveness. On a financial point of view, the player communication bill is also affected due to pay-per-byte policy. An object-oriented communication protocol with a lightweight serialization is much more appropriate than a XML based protocol to create compelling J2ME based multiplayer games. This is demonstrated in section 6 where MooDS, kSOAP2 and JSR172 are compared in terms of encoding length, marshalling/unmarshalling delay, memory allocation and application code size to confirm this statement.

5. THE MOODS PROTOCOL

MooDS (Mobile optimized objects Description and Serialization) takes care of the J2ME multiplayer games constraints by minimizing the binarization of message objects. In this section we will see how MooDS provides a lightweight object serialization.

5.1 Principle

In order to use MooDS, the developer has to describe the messaging data structures chosen for his multiplayer game using MooDS description syntax (detailed in section 5.2). Then these descriptions are parsed by the MooDS generator to obtain a class, which enables data serialization (detailed in section 5.3). The approach is to send game message objects by values, thus authorizing only Java primitive type fields: *boolean*, *byte*, *short*, *int*, *long*, *String* and *array*. During game execution, message objects to be sent are encoded over the network using the static encoder/decoder class. When an object is sent, the receiver decodes the binary stream and gets an object copy of the original message object. Figure 2 depicts the MooDS message object serialization mechanism.

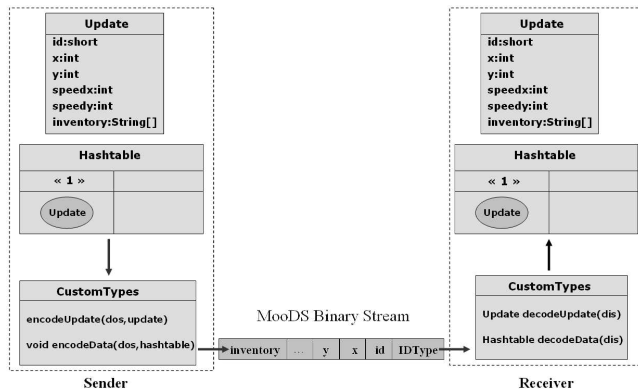
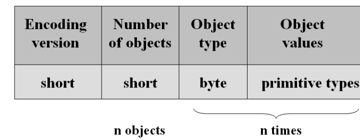


Figure 2: MooDS serialization mechanism

Figure 3 shows the MooDS binary encoding structure composed of: a *short* indicating the version of MooDS encoding, a *short* matching the number of objects contained in the binary stream and, for each encoded object, a specific *byte* corresponding to the object type followed by the object values (Java primitive types).

To summarize, there are three distinct steps in the MooDS approach: firstly data messages are specified in a description file, secondly the stub/skeleton code is generated upon compilation of the description file and finally the generated classes are embedded in the wireless application package. This approach is very close to the one used in Web Services serialization, presented in the previous section.



Encoding version	Number of objects	Object type	Object values
short	short	byte	primitive types

Figure 3: MooDS binary encoding structure

5.2 Message object type description

MooDS uses a basic *XML Schema*[25] syntax to describe message types to be used. Figure 4 presents an example showing the description of two different message types: *NewPlayer* containing information about newly connected players and *Update* which indicates the player's avatar position, speed and inventory. We can see in this example that XSD primitive types correspond to Java primitive types. The only mandatory XSD element is the *root* tag. This tag includes a choice sequence for each message type tag, in our example *newPlayer* and *update*.

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<xsd:schema xmlns:xsd='http://www.w3.org/2001/XMLSchema'
  xmlns:gmr='http://www.example.com/XML_MOODS'
  elementFormDefault='qualified'
  targetNamespace='http://www.example.com/XML_MOODS'>

  <xsd:element name='root' type='gmr:Root'/>
  <xsd:element name='newPlayer' type='gmr:NewPlayer'/>
  <xsd:element name='update' type='gmr:Update'/>

  <xsd:complexType name='Root'>
    <xsd:sequence>
      <xsd:choice>
        <xsd:element ref='gmr:newPlayer'/>
        <xsd:element ref='gmr:update'/>
      </xsd:choice>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name='NewPlayer'>
    <xsd:attribute name='id' type='xsd:short'/>
    <xsd:attribute name='username' type='xsd:string'/>
  </xsd:complexType>

  <xsd:complexType name='Update'>
    <xsd:attribute name='id' type='xsd:short'/>
    <xsd:attribute name='x' type='xsd:int'/>
    <xsd:attribute name='y' type='xsd:int'/>
    <xsd:attribute name='speedX' type='xsd:int'/>
    <xsd:attribute name='speedY' type='xsd:int'/>
    <xsd:attribute name='inventory' type='xsd:string[]'/>
  </xsd:complexType>
</xsd:schema>
```

Figure 4: MooDS message description

5.3 Object serialization mechanism

Once the message type descriptions are specified, the MooDS generator parses it and generates an encoding/decoding class, named *CustomTypes*, which has two root methods, *encodeData* and *decodeData*. The *encodeData* method takes as input the output data stream, as well as a hashtable containing several objects. Upon invocation, it calls specific methods to encode each described types. In our example, the process

will generate the methods *encodeNewPlayer* and *encodeUpdate*. These methods encode each object fields on the data output stream using Java API dedicated methods, such as the *writeByte* method. In the same way, the *decodeData* method takes as input the input data stream and returns a hashtable of decoded objects. This hashtable is obtained via the *decodeNewPlayer* or *decodeUpdate* methods, which decode the data stream. In addition, the MooDS Generator generates the message type classes, which in the example are *NewPlayer.java* and *Update.java*.

5.4 Message driven code generation

In GASP middleware, MooDS is used as a message driven code generator. To simplify the multiplayer game development, MooDS uses XSD type annotations to generate: the server game logic skeleton, the communication code to handle specific communication model for each message type and the services to access persistent object data. In addition, it proposes several generation modes to reduce the memory footprint of message classes.

The message annotation approach consists in adding context information to messages types. These information depend on the communication model and the persistency mechanism to be used. Concerning the communication model, the message types are annoted with keywords indicating whether the communications shall be of unicast, multicast or broadcast nature. *CustomTypes* class as well as the game server logic skeleton class will be generated according to the type of communication specified. Figure 5 presents this annotation built with *gasp:options* and *gasp:communication* tags to specify that the *Update* message type is a broadcast message.

Another annotation, *sql:requests*, provides services to store or retrieve data from a distant server. It enables basic SQL requests to a distant HTTP persistency server: *sql:select*, *sql:update*, *sql:insert* and *sql:delete* annotation tags are available. The MooDS generator parses type descriptions and generates two related classes: a servlet that handles persistency requests and a class, named *SQLStorageManager*, that executes the SQL requests, specified in the annotations, on a *MySQL*[17] database. Figure 5 depicts the annotation which specifies that a player inventory should be retrieved from the *players* table and to be set to the *Update* inventory field. The generated persistency service can be used also as a standalone service.

When using numerous message types, MooDS provides a compilation mode to compact all types into a unique message type class to reduce the communication code size. This unique class includes all fields of described types and an identifier field to distinguish the message type being instantiated. *KouizMarket*[29], a Doja game prototype running on GASP, uses about ten message types, which represents about 9 kilobytes of memory footprint. Using this compilation mode, the application code size has been reduced by 8 kilobytes (25 % of the total application code size in Doja 1.5).

However, for gaming applications requiring a lot functionalities, even the use of one unique class encapsulating all message types is not sufficient to stay within the application memory limit imposed by MIDP or Doja profiles. To circumvent this problem, MooDS proposes a compilation mode which makes use of Java hashtables to represent the message

```
<xsd:complexType name='Update'>
  <xsd:annotation>
    <xsd:appinfo>
      <gasp:options>
        <gasp:communicationModel>
          broadcast
        </gasp:communicationModel>
      </gasp:options>
      <sql:requests>
        <sql:select>
          select inventory as 'Update.inventory'
          from players where id=[Update.id];
        </sql:select>
        <sql:update/> <sql:insert/> <sql:delete/>
      <sql:requests>
    </xsd:appinfo>
  </xsd:annotation>
  <xsd:attribute name="id" type="xsd:short"/>
  ...
</xsd:complexType>
```

Figure 5: MooDS message annotations

type data structure. For example, in this compilation mode a *NewPlayer* object is represented by a *Hashtable* containing 3 entry keys: *type* indicating the object type, followed by *id* and *username* keys corresponding to the original object fields. Nevertheless, on the server side, object message type representation can still be used.

6. MOODS VS J2ME SOAP APIS

This section presents a comparision of MooDS, kSOAP2 and JSR172 performances by measuring: data encoding length, client memory allocation, total marshalling/unmarshalling execution delay from the first data object instantiation to the final server response unmarshalling, and obfuscated client application code size.

6.1 Test methodology

The tests are a combination of two test suites. The first test suite is designed to demonstrate the performance of each protocol in basic conditions for transferring primitive types and arrays of primitive types: one integer, an array of 2000 integers, one string of 300 bytes and an array of 200 strings. Each test server handles the client request and returns the handled data to the latter. Table 1 presents the data length of each primitive type messages contained in the client request and the server response.

	int	int[2000]	string	string[200]
Total bytes	8	16000	600	120000

Table 1: Primitive message data length

The second test suite is designed to study the protocol performances in multiplayer gaming conditions with representative message subsets of existing PC games: a message subset of an open source FPS game named *Cube*[5], a message subset of *Civilization:CallToPower2*[3] (CTP2) RTS game and a message subset of an open source action-based MMORPG named *PlaneShift*[19]. These message subsets will also enable the assessment of each protocol, regarding their capacity to represent usual multiplayer game message types. The chosen message subsets for the tests are depicted in figures 6, 7 and 8.

```
message Position{
  int x, y, z;
  int yaw, pitch, roll; //Euler angles
  int velocityX, velocityY, velocityZ;
}

message Shoot{
  int guntype;
  int shooterX, shooterY, shooterZ;
  int ennemyX, ennemyY, ennemyZ;
}

message Damage{
  int target, damage, sequence;
}
```

Figure 6: Cube message subset

```
message TradeOffer{
  long id, fromCityId, toCityId;
  long offerResource, askingResource;
  byte owner, offerType, askingType;
}

message Army{
  long id, removeCause;
  byte owner, numUnits;
  long[ ] unitIds;
  short x, y;
}

message DiplomacyProposal{
  long id, owner;
  byte senderId, receiverId;
  short priority;
  ProposalData proposalData;
}

message ProposalData{
  byte firstProposalType, secondProposalType, tone;
  Argument firstArgument, secondArgument;
}

message Argument{
  byte playerId;
  long cityId, armyId, agreementId;
  long advanceType, unitType, pollution;
  long gold, percent;
}
```

Figure 7: Civilization:CTP2 message subset

In the Cube test, the client sends a *Position* and a *Shoot*, the server returns a *Damage*. In the CTP2 test, the client sends an *Army* and a *DiplomacyProposal*, the server returns a *TradeOffer*. In the PlaneShift test, the client sends a *CraftingInfo* and an *Inventory*, the server returns a *ReadBookText* and a *WeatherMessage*. Client and server implementations handle non ordered lists of the various messages appearing in the tested subsets. The table 2 summarizes the Cube, CTP2 and PlaneShift message subset data length contained in the client request and the server response.

Subset	Cube	CTP2	PlaneShift
Total bytes	76	264	261

Table 2: Muliplayer game message data length

6.2 Experimental environment

These tests require various softwares: the *Sun Wireless Toolkit* emulator (WTK) in order to simulate a J2ME MIDP phone execution environment and its embeded memory monitor, the Apache Tomcat servlet container to execute the servlet needed during the MooDS test and the *Apache Axis2*[1] Web Services server running over Tomcat needed to handle SOAP request during the kSOAP2 and JSR172 tests. In addition, the *Sun Java Application Server*[10] is used to offer a concurrent solution to support Web Services. *Ethereal*[8] is used to analyse the network. The *ProGuard*[20] code obfuscator is used to optimize the final wireless package size of each test client.
Two PCs running Windows XP are used. One is in charge of each test client execution using WTK and its embedded memory monitor. The other runs the servlet container, the Web Services server and the network analyzer.
Test clients are executed on WTK for several reasons: to avoid application deployment time on real phone, to profit from all its tweak options like network/memory monitor or HTTP version and optional APIs selections, to avoid cellular network latency in order to clearly test the isolated performance of the protocols. Testing on real phones could be an extension to this work.

```
message ReadBookText{
  int clientnum, containerId, slotId;
  int parentContainerId;
  string name, text;
  byte canWrite;
  long visibility;
}

message CraftingInfo{
  int clientnum, count, itemsNum;
  int requiredWorkItem;
  byte requiredEquipment;
  string[ ] items;
}

message WeatherMessage{
  byte type;
  int time, downfallDrops, downfallFade;
  int fogDensity, fogFade;
  int r, g, b;
}

message Inventory{
  byte command;
  int totalItems, totalEmptiedSlots, maxWeight;
  Item[ ] items;
}

message Item{
  string name, iconImage;
  int container, slot, weight, size;
  byte purifyStatus;
}
```

Figure 8: PlaneShift message subset

6.3 Experimentation feedback

6.3.1 MTU and Nagle algorithm impact on delay

The analysis of first primitive test results shows that long messages have lower transmission delays than the short ones! These unexpected results are due to the *Nagle* TCP algorithm, which causes a delayed ACK, on server side, when the message data length is under the TCP MSS (MTU - IP/TCP headers) value. This issue is known as the *Short-Initial-Segment* problem[28]. During tests, this delayed ACK incurred an additional delay of 180 milliseconds. In Socket communication, this problem can be solved using the *TCP-NODELAY* option. However, despite the fact that HTTP is implemented over Socket, this option is not available from

J2ME HTTP connection API. Apache Tomcat offers an option to disable *Nagle* but it does not seem to work. To circumvent this drawback, the only solution with HTTP communication is to adjust the server MTU according to the length of the minimal transferable message subset.
Figure 9 shows the MTU impact on MooDS delay during primitive type test suite when decreasing its value. On Ethernet LAN network, the MTU value is by default equal to 1500 bytes. As we can see here, with a MTU value of 500 bytes the integer as well as the string data messages have higher transmission delays than their array data message counterparts. This, as mentioned before, is due to a server acknowlegment which is delayed by approximately 180 milliseconds. With a MSS value smaller than the data length, this delay is avoided. In fact, with a default MTU value the string data message (307 bytes) is sent in 200 milliseconds, whereas with a MSS value of 300 bytes, it is sent in 20 milliseconds. However, we can see that the integer data message is also sent in 20 milliseconds with a MSS value of 122 bytes, regardless of the message data length (7 bytes). This result is due to the HTTP headers length (123 bytes), which causes packet fragmentations. These fragmentations circumvent the delayed ACK of the server.

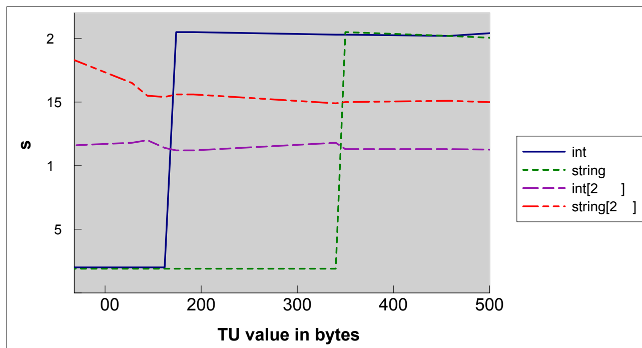


Figure 9: MTU impact on delay

Consequently, to determine the best value for the server MTU, the formula shown in figure 10 can be applied. For these tests, the formula determines a MTU value of 162 bytes. In fact, as the integer test is the smallest transferred message subset (9 bytes) and the HTTP headers represents 123 bytes, condition *(i)* is applied. However if the MTU value is decreased too much, a performance loss is noticed when long messages are used. This is due to the fact that these messages are fragmented during transmission. This is illustrated in figure 9 by the string array data message. In fact, as from an MTU of 162 bytes, decreasing the MTU by one byte entails an increase in the transmission delay by 0.27 milliseconds.

$lmsg$: length of smallest transferable message subset
$lhead$: minimal length of http headers

$$MSS = \begin{cases} lhead - 1 & if\ lhead > lmsg \quad (i) \\ lmsg & otherwise \quad (ii) \end{cases}$$

$$MTU = MSS + 40$$

Figure 10: Optimal MTU value formula

6.3.2 kSOAP2 vs JSR172

The kSOAP2 API is relatively simple to use, with a comprehensive serialization process. Currently, there are a lot of papers about kSOAP1 but there is lesser litterature about kSOAP2.
Dealing with the JSR172 API is more difficult because it requires a solid knowledge about the Web Services domain, particularly the WSDL language. In fact, the domain suffers of a lack of comprehensive documentation about how to describe correctly the Web Services features of JSR172 based wireless application. For example, it is very tedious to find the appropriate encoding style to use, as well as the XSD tags that have been restricted, such as the useful *xsd:choice* tag (unordered sequence of tags). Moreover, there exists some problems with list structure handling during unmarshalling of the server response with the JSR172 version embedded in WTK 2.2. The second beta version of WTK 2.5 fixes the problem.

6.3.3 Axis vs J2EE web service server

A lot of tricky bugs arised during the development of SOAP based test client using the Axis Web Services server. For instance, during evaluations, the JSR172 client was frequently crashing unexpectedly. After some painstaking debuggings the investigations revealed that the crash orignated from the parsing of an empty SOAP header tag generated by Axis in its response. The empty header generation has been fixed by modifying the source code of the Axis SOAP encoding generation API, *Axiom*. Finally, through this implementation experience, some interoperability obstacles regarding kSOAP2 and the Axis server have also been revealed. For instance, the Axis Server does not support SOAP array encoding, which is heavily used by kSOAP2 to encode list-like structures. For all these reasons, the Sun Java Application Server has been used in these tests to support server application codes.

6.4 Test results

Figure 11 shows the results in terms of encoding for each protocols. Regarding the primitive type test results, MooDS is about 1.3 times longer than the raw data length values shown in table 1, whereas JSR172 and kSOAP2 are respectively about 29.7 times and 27 times longer. In fact, MooDS provides a better encoding mechanism, which enables a more compact representation of big arrays. In the context of multiplayer gaming, MooDS is about 1.2 times longer than raw data length values shown in table 2, whereas JSR172 and kSOAP2 are respectively about 12.2 times and 14 times longer. These huge encoding length differences between MooDS and SOAP based protocols enlighten the benefit of data binarization in the context of heavily constrained embedded environments. The difference between JSR172 and kSOAP2 in primitive type tests is dependent on the SOAP encoding, generated to handle lists. In fact in kSOAP2, list-like structures are represented as SOAP arrays, whereas in JSR172, list-structures are represented by *xsd:sequence* tags, containing both primitive and/or complex type elements.

Figure 12 shows the average execution delay of the marshalling and unmarshalling for each protocol. In primitive type tests, MooDS has the fastest marshalling/unmarshalling rates, with a delay of about 20 milliseconds for normal-sized messages (ranging from 8 to 600 bytes) and

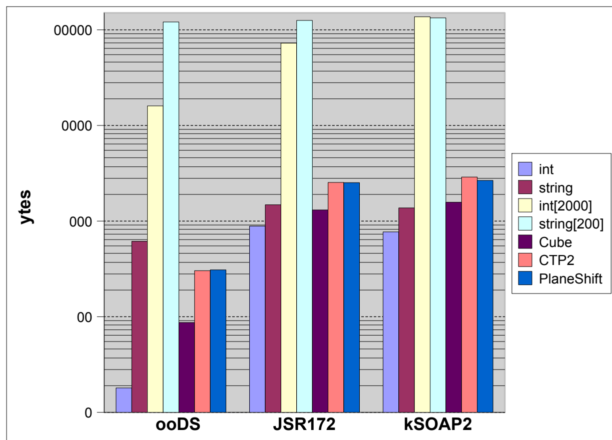


Figure 11: Request data encoding length

a delay of about 130 milliseconds for big array messages (greater than 16000 bytes). The delay for normal-sized messages is uniform, regardless of the number of bytes exchanged. This shows that the MooDS encoding mechanism has an insignificant impact on transmission delays. We can see in this case that JSR172 and kSOAP2 are respectively 2 times and 4.2 times slower than MooDS. When considering big array messages, JSR172 is 6 times slower than MooDS whereas kSOAP2 is 23 times slower. The difference in delay between MooDS and JSR172 can be explained by the serialization/deserialization delay of the SOAP encoding. However, it is important to underline that the large difference between JSR172 and kSOAP2 delays results from the SOAP array marshalling/unmarshalling. As it is demonstrated in the experimentation feedback section, it is necessary to correctly adjust the server MTU in order to obtain the proper delays. Here it is set to 162 bytes.

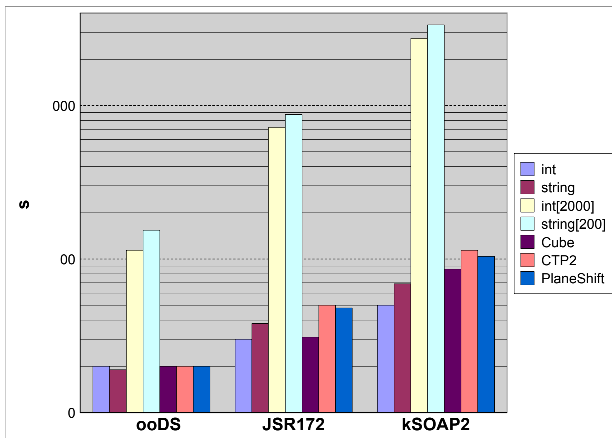


Figure 12: Marshalling/unmarshalling delay

Figure 13 illustrates the results in terms of memory allocation under multiplayer gaming conditions. MooDS uses less memory than the SOAP based protocols, 73 kilobytes of used memory, against 83 kilobytes for the JSR172 and 129 kilobytes for kSOAP2. These differences can also be explained by the memory footprint of XML data representation during the creation of SOAP messages. Here, we can notice that JSR172 has a much more efficient management of XML data representation than kSOAP2.

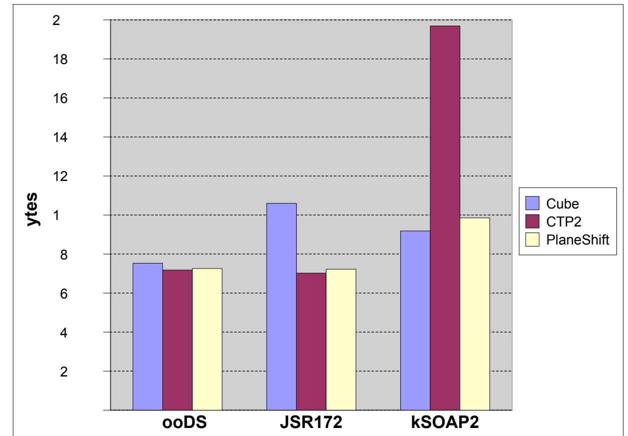


Figure 13: Memory allocation

Finally, figure 14 shows the code size of each test client package under multiplayer gaming conditions, obfuscated with Proguard. We can see also that MooDS outperforms the two SOAP based protocols, with only a 6 kilobytes code size, against a 9.6 kilobytes and a 35 kilobytes code size for the JSR172 and kSOAP2 respectively. The differences between MooDS test client and the JSR172 test client illustrates the code optimization of the MooDS stub/skeleton class *CustomTypes* in comparison to the JSR172's one. kSOAP2 is penalized by the embedding of the kSOAP2 API into the wireless application package. On the other hand, the JSR172 API is directly embedded in the phone MIDP 2.0 APIs with no impact on the code size of the application.

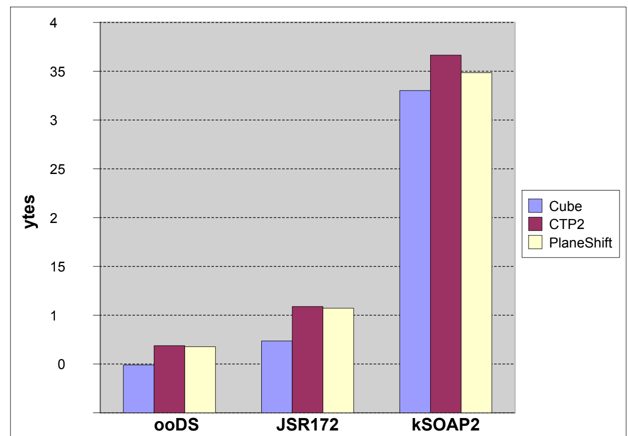


Figure 14: Client code size obfuscated

6.5 Protocol comparison conclusions

The comparison results demonstrate that the MooDS protocol is a serious alternative to SOAP based protocols for handling communication in heavily constrained embedded environments, such as the J2ME CLDC platform. MooDS obtains the best results in terms of encoding length, memory allocation, communication delay and client code size.
Numerous works reveal that Web Services lack of perfor-

mance under heavy load. This results from the SOAP message XML encoding, implying heavy parsing process at both client and server side with huge encoding length, execution delay and memory allocation (especially in deserialization of SOAP arrays)[26]. The *Fast Web Services* is a Sun initiative started in 2003 to enhance SOAP performance by replacing XML encoding of SOAP messages by an *ASN.1* binary encoding[30], but currently there is no implementation available. As a perspective for MooDS, we can imagine of using the MooDS binary encoding to replace the SOAP encoding mechanism in web services by generating a stub/skeleton class from the WSDL parsing.
As a result, all tested subsets of usual multiplayer game messages have been successfully described in each protocol. Finally, MooDS is compatible with all versions of MIDP and Doja profiles. In opposition, the JSR172 is only compatible with MIDP 2.0, whereas kSOAP2 is compatible with all MIDP versions but not with Doja.

7. CONCLUSIONS

This paper presents MooDS, an object-oriented communication protocol designed for compelling J2ME multiplayer games. MooDS takes care of the specific constraints imposed by J2ME devices over 2/2.5/3G cellular phone networks. Beyond multiplayer games, MooDS protocol can be used in general apurpose multi-user connected applications. MooDS is the communication protocol used in the GASP project, both in online (HTTP) and Bluetooth®[2] versions of the middleware. It is available as an Ant task from the GASP website[9].
This paper compares the performances of MooDS to those of SOAP based protocols, JSR172 and kSOAP2 APIs, in both basic and multiplayer gaming conditions. The comparison results assert that MooDS delivers much better performances in terms of encoding length, memory allocation, marshalling/unmarshalling delays and final application code size. Additionnally, this paper underlines the importance of server MTU value, which has a direct impact on communication delays over HTTP, and presents a formula to determine its optimal value.
The OMA GS works on specifications for the next generation of mobile-server interactions. According to recent drafts, OMA GS wants to introduce the SOAP protocol as the root communication protocol for their future recommended architecture. Taking in account the lack of performance on part of SOAP based protocols, this choice may entail highly delayed communication in the context of resource constrained environments like the J2ME platform, particularly to support time critical multiplayer games. MooDS can be considered as a serious alternative to handle the communication part of OMA GS architecture.
In the near future, MooDS will support some forms of language heterogeneity by enabling multiplayer gaming interactions between various platforms. Finally, as stated before, MooDS can be considered as a solution to enable fast Web services access for J2ME mobile applications by supporting WSDL parsing to generate efficient stub and skeletons. In this case, a solution is to integrate MooDS as an interface on the server-side to enable interaction between J2ME clients and the usual web services.

8. REFERENCES

All URLs last visited in April 2007.

[1] Axis2 server, http://ws.apache.org/axis2/.
[2] Bluetooth technology, http://www.bluetooth.com.
[3] Civilization:CallToPower2, http://apolyton.net/civ2/.
[4] CLDC, http://java.sun.com/cldc/.
[5] Cube, http://cubeengine.com.
[6] Doja profile, http://www.doja-developer.net.
[7] Enhydra project, http://www.enhydra.org.
[8] Ethereal network analyzer, http://www.ethereal.com.
[9] GASP project, http://gasp.objectweb.org.
[10] J2EE server, http://java.sun.com/j2ee/.
[11] J2ME platform, http://java.sun.com/j2me/.
[12] J2SE, http://java.sun.com/j2se/.
[13] JAX-RPC, http://java.sun.com/webservices/jaxrpc/.
[14] JSR172, http://jcp.org/jsr/detail/172.jsp.
[15] kSOAP2, http://ksoap2.sourceforge.net.
[16] MIDP profile, http://java.sun.com/products/midp/.
[17] MySQL, http://www.mysql.com.
[18] Open Mobile Alliance Gaming Services, http://www.openmobilealliance.org/tech/wg_committees/gs.html.
[19] PlaneShift, http://www.planeshift.it.
[20] ProGuard, http://proguard.sourceforge.net.
[21] SOAP, http://www.w3.org/tr/soap/.
[22] Tomcat server, http://tomcat.apache.org.
[23] Web Services, http://www.w3.org/2002/ws/.
[24] WSDL, http://www.w3.org/tr/wsdl.
[25] XML Schema, http://www.w3.org/xml/schema/.
[26] K. Chiu, M. Govindaraju, and R. Bramley. Investigating the limits of soap performance for scientific computing. In *11th IEEE International Symposium on High Performance Distributed Computing (HPDC-11 '02)*, pages 246–255, July 2002.
[27] D. Fox. Wireless gaming using the java micro edition. In *Game Programming Gems 3 - Charles River Media*, pages 573–581, 2002.
[28] J. C. Mogul and G. Minshall. Rethinking the tcp nagle algorithm. *ACM SIGCOMM Computer Communication Review*, 31:6–20, January 2001.
[29] R. Pellerin, F. Delpiano, F. Duclos, E. Gressier-Soudan, and M. Simatic. GASP: an open source gaming service middleware dedicated to multiplayer games for j2me based mobile phones. In *7th Int. Conference on Computer Games CGAMES'05 Proceedings*, pages 75–82, November 2005.
[30] P. Sandoz, S. Pericas-Geertsen, K. Kawaguchi, M. Hadley, and E. Pelegri-Llopart. Fast web services, http://java.sun.com/developer/technicalarticles/webservices/fastws/index.html, 2003.
[31] J. Smed, T. Kaukoranta, and H. Hakonen. Aspect of networking in multiplayer computer games. In *The International Conference on Application and Development of Computer Games in the 21stCentury proceedings*, pages 75–82, November 2001.