SMILE-JS, a SIP-based Middleware for J2ME Devices

Giovanni Bartolomeo
University of Rome "Tor Vergata"
Rome, Italy
+39 06 72597450
giovanni.bartolomeo@uniroma2.it

Stefano Salsano
University of Rome "Tor Vergata"
Rome, Italy
+39 06 72597770
stefano.salsano@uniroma2.it

Andrea Polidoro
University of Rome "Tor Vergata"
Rome, Italy
+39 06 72597769
andrea.polidoro@uniroma2.it

ABSTRACT
In this paper we report our two years experience in designing and implementing a new middleware solution for distributed mobile applications exploiting the Session Initiation Protocol (SIP) and the JavaScript Object Notation (JSON). The proposed solution has been designed to port the Simple Middleware Independent LayEr (SMILE) framework to mobile devices running the limited version of Java 2 Micro Edition (J2ME CLDC). It provides J2ME developers with the very same abstraction layer offered by the SMILE API under the J2SE environment, allowing seamless interoperability between SMILE peers running on desktop computers/servers and peers running on mobile devices. The solution will be denoted as SMILE-JS, where JS stands for JSON over SIP. We first describe the SMILE framework, explaining its APIs for communication, addressing, lifecycle management and service discovery. Afterwards we explain how truly peer to peer communication among mobile devices has been achieved using SIP, and which additions we implemented to turn the open source MjSIP framework into the first SIP-based middleware for J2ME CLDC enabled devices.

Categories and Subject Descriptors
C.2.4 [**Computer-Communication Networks**]: Distributed Systems – *Distributed applications*

General Terms
Design, Experimentation

Keywords
Mobile Middleware, J2ME, SIP, JSON, SMILE

1. INTRODUCTION
The Simple Middleware Independent LayEr (SMILE) was originally intended as a lightweight framework allowing to decouple the functional model of a distributed Java application from the underlying middleware platform used to run the application itself. Its original motivation could be found in the need for reducing time and costs in porting existing software to different middleware platforms. According to [1] the SMILE framework can be assimilated to an "abstract platform", i.e. a *collection of characteristics assumed in the construction of an application at some point of the design process*. However, what SMILE adds is the ability to seamlessly execute the designed application on a real middleware platform, through its so called "bindings". In facts, SMILE definition of interfaces is inspired by the Service Oriented Architecture (SOA) and uses WSDL as interface description language (in [6] we give more details about this choice). Borrowing concepts and the terminology from the Web Service Definition Language (WSDL) [3], a "binding" represents a link between the abstract definition of an application in terms of SMILE API and a concrete technology to rely on for the execution of the application itself. However, whereas traditional WSDL bindings are limited to web technologies, in the past we developed SMILE bindings for the most known middleware platforms (such as OSGi, CORBA, JXTA and JADE). All of them present similarities which have been used to define the SMILE abstraction layer.

An early proof of concepts for SMILE was achieved in late 2005, in the context of the IST-Simplicity project, where SMILE API were used to port the project demonstrator (see [7]) to two well known middleware platforms (JXTA and JADE). With the beginning of the IST Simple Mobile Service (IST-SMS) project [13] in middle 2006, we started exploring the porting of SMILE API to mobile devices, having the J2ME CLDC platform as target. Our first design goal was to achieve complete interoperability: peers running on desktop/server should have been allowed to seamlessly talk with peers running on mobiles, and vice-versa. Also, we wanted to keep the very same abstraction layer, i.e. having exactly the very same API for mobile version and desktop/server version. A second design goal was to cope with the typical networking conditions of mobile devices, which show intermittent connectivity and NAT issues (mobile devices often get private IP addresses behind a NAT). Incidentally we note that NAT traversal issues does not only apply to mobile devices and mobile middleware, but to any middleware solution that wants to support universal connectivity of nodes irrespective of their access network. Our third and fourth design goals were respectively to have support for automatic generation of code stubs and for automatic serialization of data structures also in the mobile (J2ME CLDC) environment. The fifth design goal was to keep our technology backward compatible with traditional SOA/Web Services approach.

We investigated some legacy middleware solutions for mobile device, including JXME [9], an implementation of JXTA for Java 2 Micro Edition (J2ME), and the agent oriented middleware

This work was supported in part by the EC in the context of IST project Simple Mobile Services – IST-SMS (IST 2006-034620).

JADE in its version for mobile devices (LEAP, [10]). Both of the aforementioned solutions are based on a connection oriented protocol (HTTP or TCP) establishing a persistent connection with a proxy server acting on behalf of the client. In JXME, for example, a "relay" in the network takes care of almost all JXTA functionalities, participating in the JXTA network on behalf of one or more J2ME devices, freeing mobile devices from parsing verbose XML messages, caching advertisements and, most important, accept and handle incoming connections. JXME "peers" uses a simplified protocol, and exchange HTTP-based binary messages with the JXTA relay. A similar solution is adopted in JADE LEAP for software agents running on J2ME devices. In conclusion, despite these solutions claimed to be "peer-to-peer", they actually shift middleware functionalities from mobile devices to network servers able to maintain the state of several clients at the expense of a powerful but costly central network infrastructure. Being not happy with existing solutions, we decided to implement our own middleware solution, making challenging choices to cope with specific constraints imposed by mobile devices.

In June 2007, a first prototype of SMILE-JS for mobile device was released and a demo application (an indoor tracking system running on mobile devices) was presented at WWRF 18 in Helsinki, Finland. The experience gained allowed us to refine several details in the internals of our middleware solution which was then definitively adopted in the IST-SMS project demonstrator, presented in the first Workshop on "User Generated Services" held in Madrid, Spain, June 2008.

This paper describes our two years experience in designing and implementing SMILE-JS, the first SIP-based middleware for J2ME CLDC enabled devices, exploiting the open source MjSIP SIP stack [11]. In particular, section 2 provides details of SMILE operations and APIs, section 3 deals with transport of messages in SMILE-JS, section 4 discusses serialization of messages in J2ME. SMILE and SMILE-JS are developed under an open source license and can be downloaded from [12].

2. SMILE APIs

The details of SMILE operations and APIs provided in this section show how we fulfilled the first design goal, i.e. to achieve complete interoperability between peers running on desktop/ servers and peers running on mobile devices. What is described hereafter applies without changes both to the J2SE and to the J2ME CLDC implementation of SMILE-JS.

2.1 Communication Primitives

In SMILE, communications between peers rely on "operations". There are four kinds of operations, classified according to the originator and the executor of the operation itself; the WSDL[1] terminology refers to these latter entities as client and server, and defines the first pair of operations as originated by the client:

- A "OneWay" operation consists in a message originated by the client toward a server. The message is sent asynchronously, no "session" is created between the client and the server.
- A "RequestResponse" operation consists in a request message from the client to the server, followed by a reply message ("response"). Alternatively, a "fault message" may come to the client if something has gone wrong at the server side. There is a logical correlation between the request and the response: the execution environment creates a "session" and provides the corresponding response or fault in an unambiguous way as reply to a given request.

The second pair of operations is complementary to the first one, the originating being the server as follow:

- A "Notification" operation is made of a message sent asynchronously by the server to a client, no "session" is created between the client and the server.
- A "SolicitResponse" operation consists in a solicitation message sent from the server to the client, followed by a reply message in the opposite direction. Solicitations may originate faults as well. The same "session" concept explained for "RequestResponse" applies.

Being based on a peer to peer paradigm, SMILE peers uses the same primitives to generate and to receive "oneway" messages and "notifications". These primitives are, respectively, the *send* method and the *onReceived* callback. The *send* method is not blocking, therefore it realizes an "asynchronous" interaction pattern. Instead, to implement "RequestResponse" and "SolicitResponse" operations, the blocking *doRequest* method is used. This method accepts the request message to be sent as a parameter and returns the corresponding response or raises an exception if a fault is received. SMILE takes care of correlation between request and response, as explained below:

- Each message has a serial number that is automatically generated whenever the message is instantiated.
- Each response message has a *refSerial* field keeping the serial number of the request it answers.
- A boolean query method *refersTo* tells if a reply refers to a given sent request.
- A internal method *expectMessage* allows to register the reply number to expect.
- After sending the request, the sender peer performs message cleanup and then wait for a given a timeout. This blocks the flow execution and makes the call synchronous.
- Incoming messages that do not refer to the outstanding one are ignored and discarded.
- Once the right message arrived, the *expectingMessage* state is cleared, and the *doRequest* method returns the response message to the application.
- Timeout expiration throws an exception and clears the *expectingMessage* state as well.

Thus, "requests" are synchronous, and implicitly confirmed as soon as a response message is received. Remote Procedure Call (RPC) is implemented using the request/response pattern as described in section 2.5.

A second use case for the request/response pattern is receiving confirmation of message delivery. As with asynchronous messages there is no guarantee for the sender that the issued message comes to the recipient, whenever a confirmation is

[1] At the time of writing, SMILE supports only the two basic patterns defined in WSDL (unreliable oneway and synchronous twoway messages). Future extensions will probably support more asynchronous modes of operation.

needed, it is possible to use the response message of a request to get confirmation of message delivery.

2.2 SMILE-JS Peers, Processes, Identifiers

An application built on an abstraction layer like SMILE will be "abstract" as well. In order to make it actually work, the application needs a "binding" to a concrete runtime. A binding is intended to bind each abstract SMILE peer to at least one running "process". A process (later on sometimes referred itself as "binding") is an implementation of SMILE abstract primitives in a middleware specific platform. Each process has its own address, which is an instance of a class inheriting from SMILE ProcessID class. In this sub-section we only focus on the JSON/SIP binding (SMILE-JS).

In SMILE-JS, processes are mapped into SIP "user agents", which are identified through their "Address of Record" (AoR), in form of an URI like `sip:alice@iptel.uroma2.it`. This URI is used in each SIP message exchanged between agents to identify the sender and the recipient. Like email addresses, traditionally SIP AoR are assigned by a SIP provider to SIP users to enable them to receive and initiate communication sessions. A one-to-one mapping between processes and SIP agents would result in an inefficient use of resources, requiring a new SIP AoR to be assigned to each new SMILE process created in the local execution environment. A many to one mapping is more efficient. To fit this goal, we defined two new fields to be used, additionally, in the "From" and "To" headers of SIP messages in order to distinguish among different SMILE processes exploiting the same SIP agent.

- *pType*, containing information about the "kind" of service implemented by the process. This field could be assimilated to the concept of class name in an Object Oriented programming language.
- *pName*, unique "address" of a specific process, can be used to distinguish between more instance of the same "kind" of service described by *pType*. It could be assimilated to the concept of instance identifier in an Object Oriented programming language.

An example of "From" header used in messages generated by our JSON/SIP binding could be the following:

```
From:<sip:alice@iptel.uroma2.it;pType=
org.istsms.mem.MemPeer,pName=0a87f4>
```

The object taking care of delivering SIP messages to the right process is called Dispatcher (Figure 1).

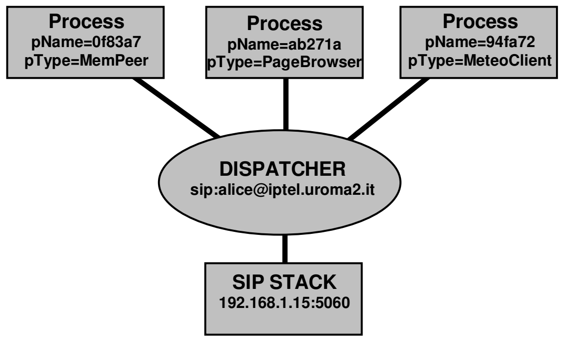

Figure 1. A Dispatcher serving several SMILE processes using the same SIP agent.

The dispatcher is assigned an AoR and acts as a SIP agent for the SIP stack, thus receiving any SIP message sent to its SIP AoR. Whenever a message arrives, the dispatcher extracts the specific process identifier from the *pName* and *pType* fields contained into the "To" header and forwards the message to the corresponding process. Viceversa, whenever a process wants to send a message to another one, it specifies its own identification parameters (sender's *pName* and *pType*), the recipient's identifications parameters (recipient's SIP AoR and recipient's *pName* and *pType*) and delivers the outgoing message to the Dispatcher. The Dispatcher prepares a corresponding SIP message and sends it to the network through the SIP Stack.

2.3 SMILE API for Lifecycle Management

Other than responding to events originated whenever messages are received, each SMILE peer may execute proactively a business logic. Each peer implements three callback methods which are inherited by the *ProcessLifecycle* interface and are called in sequence, in a thread different than the one used to notify peers about incoming messages. These callbacks are hereafter described:

- *setup*, called as soon as the peer starts, is usually implemented to perform start up operations.
- *doBusiness*, invoked as soon as *setup* returns, is exploited by the programmer to implement the application's main loop. As described in 2.4, an expiration is associated to each service description published by the peer, thus an example of implementation for this callback is the periodic re-publication of service descriptions. Other example may include periodic operations specific to the peer's logic, like polling a resource to obtain a fresh information or periodically notify other peers subscribed to a service the peer provides.
- Whenever the *doBusiness* method ends, the *takedown* callback is called. Typical usage for *takedown* includes release of resources and un-publishing of service descriptios (if any).

2.4 SMILE API for Publishing and Discovering Services

The service discovery API is related to the functional features provided by each peer. These features are called "services". Services provided by SMILE peers are described using a *Descriptor* object which contains details related to the service type and the operations the service does support. The field identifying the service type is mandatory, whereas the list of supported operations may be empty. Each peer is provided with the following set of primitives:

- *publish*, used to publish a service description; the publication is limited in time by an expiration time which could be specified whenever this method is called. A maximum expiration time can be defined by the platform administrator.
- *search* is used to find peers providing a given service; This method returns an array of *ProcessID* objects identifying all the peers which have published a matching service description. In order to match against a list of *Descriptor* objects, a service *DescriptorFilter* object is used. The *DescriptorFilter* object specifies the following fields: service type (mandatory), supported operations, maximum number of services to be returned, "matching ProcessID". The optional "matching ProcessID" field may contain a regular expression which is

evaluated against the identifiers of the peers providing one or more services matching the specified type and the specified operations.

- *delete*, used to remove a published *Descriptor* object.

Normally one single service publication is made at the beginning of the peer lifecycle, repeated periodically at regular time intervals to overcome expiration, and deleted once the peer ends. More complex patterns are possible, e.g. to cope with temporary service unavailability or publication of new services made dynamically available during the peer's lifecycle. The programmer should take care that no more valid service publications are removed, in order to prevent client requests to fail.

2.5 RPC in SMILE

Remote Procedure Calls (RPC) is implemented in SMILE using the *doRequest* primitive. The example hereafter discussed is related to a "Music Lovers Service" and shows two parts of the code, corresponding to the client stub code, Figure 2 and to the skeleton implemented on server side, Figure 3. It exploits the request/response operation "getTopArtists".

```
public com.ftrd.om.ws.dto.xsd.User[]
  getTopArtists(int limit){
    GetTopArtistsRequest req=
      new GetTopArtistsRequest();
    req.setLimit(limit);
    req.setOperation(GT_TOP_ARTSTS);
    try {
      GetTopArtistsResponse res=
        (GetTopArtistsResponse)
        doRequest(req,provider);
      return res.getArtists();
    } catch (Fault e) {
        log("Fault: "+e);
    } catch (InvalidReceiverException e) {
        log("Receiver is offline");
    }
    return null;
  }
}
```

Figure 2. Stub code running in the client.

```
public Message onRequestReceived(ProcessID
sender,Message request){
  if(running){
    if
(request.getOperation().equals(GT_TOP_ARTSTS)
&& request instanceof GetTopArtistsRequest) {
        GetTopArtistsRequest req=
          (GetTopArtistsRequest)request;
        GetTopArtistsResponse res =
          new GetTopArtistsResponse();
        res.setArtists(
          getTopArtists(req.getLimit())
        );
        return res;
    } // else if ...
  }
}
com.ftrd.om.ws.dto.xsd.User[]
getTopArtists(int limit) {
        //server code here...
```

Figure 3. Skeleton code in the server.

As soon as the client's method *getTopArtists* is invoked, the stub sends a corresponding request message to the service provider (which may have been discovered at startup using the *search* method described in section 2.4). The *getTopArtists* method blocks until a response is received, a fault is generated or the request's timeout expires. The server implements the onRequestReceived callback in a way that whenever a "GetTopArtistsRequest" message is received *and* the message is related to a "getTopArtists" operation, the corresponding skeleton method *getTopArtists* is called. The returned value is transported inside the related response message which is eventually returned to the client.

3. SIP transport for SMILE-JS:

It is well known that NATs and firewalls do not allow peer to peer communication among mobile devices: a peer behind a NAT is usually not reachable from the outside world. In past years, SIP [5] has emerged as a signalling protocol to establish calls and multimedia sessions between user agents on end user equipments, even behind NATs and firewalls. With 3GPP mandating the use of SIP for the future evolution of 3G networks, SIP will be largely supported in operator's network in next years. Thus, to fulfil our second design goal, (i.e. to overcome "natted" network limitations *and* to cope with frequent network disconnections) we decided to use SIP messages transported over UDP datagrams and to resort to a known NAT traversal solution for SIP, based on the so called "Session Border Controller" (SBC) element. This approach allows to run "full" SMILE peers keeping all of their functionalities even on devices which do not own a public IP address[2].

The SIP infrastructure elements and the SIP stack for both mobile devices and server side are based on the open source MjSip project [11]. The overall architecture for the JSON/SIP binding of SMILE over SIP is shown in Figure 4: the SIP infrastructure is composed of a *Registrar and Proxy server,* which maintains the mapping between SIP user agent identifiers (SIP addresses) and their IP addresses, and routes SIP messages to recipients, and by a SBC able to route incoming SIP messages to peers behind NATs. Although very simple, this infrastructure allows the exchange of SIP messages among mobile devices and between mobile devices and server side elements.

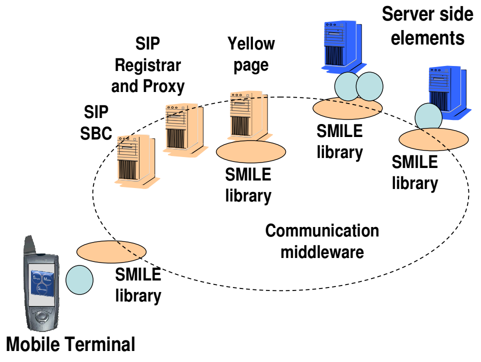

Figure 4. SMILE and SIP elements.

[2] We also report that, in principle, other SIP based solution for NAT traversal might be used as well.

Figure 4 also shows the "Yellow Pages" discovery proxy, which allows SMILE peers to register/deregister their service descriptions and to look for services offered by other peers, using the API described in section 2.4.

3.1 A Support for Large SIP Messages

Unfortunately, SIP messages keep one drawback. In fact, given its nature of application-level "signalling protocol", independent from the underlying transport, the original SIP specifications do not mandate any form of reliable delivery for messages of large size. Whenever SIP is transported using a connection oriented protocol such as TCP, fragmentation is handled by the transport level, which ensures reliability. This solution, however, is less suitable for mobile devices which experience frequent disconnections, being more appropriate, in this case, transporting SIP messages in UDP datagrams. This way, unfortunately, SIP messages are subject to IP message fragmentation, whose behaviour may vary from network to network, depending on the Maximum Transmission Unit (MTU) allowed.

To alleviate this problem, the JSON/SIP binding implements a simple, sliding windows based fragmentation/defragmentation mechanism. Application level messages are fragmented and transported in SIP messages over UDP datagrams (Figure 5). Confirmation of reception of a single fragment is implicitly given by the SIP response, sent back by the SIP receiver agent as soon as a SIP message arrives. If a fragment is not acknowledged, then it is retransmitted using the exponential backoffs strategy described in [5]. Purpose of this application level fragmentation mechanism is neither to replace IP fragmentation, nor to reinvent TCP flow and congestion control algorithms. Rather, by giving control on the maximum size of each message payload, this solution should be only intended as an enabler to send relatively large amount of data over conventional SIP messages. More details can be found in section 3.2.

```
MESSAGE sip:stefano@stefano:5070 SIP/2.0
[some headers removed for clarity's sake]
Max-Forwards: 68
To:
<sip:stefano@sipdev.netgroup.uniroma2.it;pNam
e=c5da66;pType=MemPeer>
From:
<sip:yellowpages1.0@sipdev.netgroup.uniroma2.
it>;tag=681452288432
Content-Length: 225
Content-Type: application/text
{"Req":"false","__class":"it.uniroma2.smile.s
ipbinding.sipmessage.MessageEnvelope","FNo":"
0","Frag":"{\"Name\":\"it.uniroma2.smile.core
.Message\",\"Fault\":\"false\",\"__class\":\"
it.unir","SNo":"10135","RefS":"1","FTot":"4"}
```

Figure 5. A SIP message containing a fragment to be delivered to a SMILE peer.

3.2 Binary Objects over SIP

SMILE-JS supports the transport of binary object within messages, but at the expense of performances. Since JSON is a text format, binary objects are text-coded before being sent on the wire. The chosen encoding is the widely known MIME Base64 encoding, which increases the original data length of about 30%. In our tests, we transmitted JPEG images of about 20 Kb, turned into 25 Kb text streams. Using a maximum payload less than 1500 Kb per SIP message, about 20 SIP messages were needed to delivery this binary object, using the above described fragmentation mechanism. This took up to 5.5 seconds on an UMTS network (tests performed on working days, in the morning).

We concluded that this simple mechanism is suitable to transfer relatively small/medium size binary files (e.g. icons, thumbnails, small images) whereas transmission of larger binary objects should preferably rely either directly on optimized TCP based transport protocols, or on more sophisticated SIP based solutions, similar to those described in IETF MSRP protocol [2].

4. Seamless Serialization for J2ME

A serialization mechanism is needed to transform the internal representation of an object into a stream of bytes that can be transported on the wire, interpreted and reconverted to a copy of the original object at destination. The specific serialization format could be binary or text based. In Java 2 Standard Edition, a binary serialization mechanism is built in and text based serialization (e.g. XML serialization) is provided in form of API. On the contrary, J2ME-CLDC does not support any automatic serialization, thus the application developer has to implement her own serialization mechanism for each application. In order to free developers from this tedious task, which reduces interoperability and limits the development of distributed applications for mobile devices, we integrated a general, seamless serialization mechanism into SMILE-JS, thus fulfilling our fourth goal.

Even if, in principle, any serialization format could have been used, for its compactness (compared to XML) and human readability (contrary to binary formats), we have chosen to relay on the JavaScript Object Notation (JSON) [8]. JSON is a simple text format, based on a subset of the JavaScript Programming Language. Other than primitive values, it supports only two basic structures: a collection of key/value pairs and an ordered list of values. These data structures are implemented in almost all modern programming languages, this makes it easier to achieve portability for application written in different programming languages. In particular, Java implementations of JSON typically map the two basic JSON structures into, respectively, hash tables and vectors, providing objects named JSONObject and JSONArray, which are java specific runtime representation of a JSON stream.

Despite it is possible to use JSON API directly into Java applications, the challenge for our JSON/SIP binding has been to provide a general mechanism to allow any Java bean to be seamlessly serialized into and de-serialized from JSON streams. Some existing tools allow such a translation, but, unfortunately, they mostly rely on class introspection, not available on J2ME. Thus we decided to implement our own translator for SMILE-JS. As a first step, we defined how arbitrary Java beans and arrays should have been mapped into corresponding JSONObjects and JSONArrays. Java objects which are beans are mapped to JSONObjects. Each object's field accessible through a public getter method is serialized into a corresponding field into the target JSONObject. Figure 6 illustrates a thus produced JSON stream. The field's name is taken as key whereas the actual value depends on the specific type returned by the getter method, as hereafter described.

- There may be four cases: a primitive value, a Java array, another Java bean or a null.
- Primitive values and wrapper classes are mapped into strings.
- Arrays are turned into JSONArrays and serialization is recursive: for each element in the array, a corresponding entry in the JSONArray is created, and the procedure described in this paragraph is recursively applied to the element, according to its actual type.
- Java beans are recursively serialized.
- Serialization of null fields does not produce any entry.
- Serialization of empty array produces an empty JSONArray.
- To cope with inheritance, annotation is used in the produced JSONObject: whenever an actual parameter in a field is of a type inherited from the type declared in the corresponding formal parameter, a special entry is added in the corresponding JSONObject to remember the actual parameter's type and allow proper deserialization.

```
{   "Req":"false",
    "Fault":"false",
"__class":"it.uniroma2.smile.sipbinding.sipme
ssage.MessageEnvelope",
    "FNo":"0",
    "Frag":{
         "ContactList":{
          "Contacts":[
"sip:andrea@iptel.uniroma2.it;pName=a6f087;pT
ype=MemPeer",
"sip:stefano@iptel.uniroma2.it;pName=c5da66;p
Type=MemPeer"]},
    }
"__class":"sms.contactlist.message.ContactLis
tResponseMessage",
    "SNo":"1047",
    "RefS":"1",
    "FTot":"1"
}
```

Figure 6. The bean *ContactListResponseMessage* serialized as a JSON stream.

Using the above rules we implemented a serialization/ deserialization library for J2SE platforms, using introspection. A straightforward way to port this approach to J2ME platform is to replace the class introspection mechanism provided by J2SE with pieces of code enumerating the different beans to serialize/deserialize. This does not change the internals of the serialization mechanism, it just substitutes introspection with enumeration. The drawback is that for any class that has to be serialized a corresponding code enumerating every public class' field should be written. This task however can be easily done automatically importing the classes to be serialized into a J2SE environment and using a tool which introspects them and automatically produces the enumeration code needed for serialization in J2ME. We actually implemented such a tool, and called it "JavaBean2JSON StubGenerator". This tool is an open source software included in the SMILE-JS distribution [12].

5. CONCLUSIONS

In this paper we described some features of SMILE-JS, a new truly peer-to-peer middleware solution for distributed mobile applications running on J2ME CLDC devices. We have discussed how SMILE-JS meets a set of design goals including identical APIs and protocols in J2SE and J2ME implementation, seamless NAT traversal, seamless serialization of messages in J2ME. Due to space constraints, we do not describe our approach to automatic code generation from SMILE interface descriptions. We just mention that we make available a syntax for SMILE interface description simpler than WSDL, more human readable and allowing inline Javadoc-like comments. A corresponding model-to-code transformation tool converting interface description into SMILE code has been released. We do not even discuss the interoperability between SMILE and SOAP based Web Services; we only mention that gateways to map SMILE messages into SOAP messages and vice versa have been implemented [6].

Finally we would like to mention that recently a SMILE-JS implementation for IMS has been released [4] and that SMILE-JS is currently under beta testing in an ongoing trial at University of Rome Tor Vergata, involving about 100 participants [1].

6. ACKNOWLEDGMENTS

Authors would like to thank MiNEMA - Middleware for Network Eccentric and Mobile Applications - for funding the presentation of this work.

7. REFERENCES

The links below are last accessed on the 2nd of October 2008.

[1] J. P. Almeida, R. Dijkman, M. van Sinderen, L. F. Pires, "On the Notion of Abstract Platform in MDA Development", IEEE International Enterprise Distributed Object Computing Conference (Monterey, California, USA, 2004)

[2] B. Campbell, R. Mahy, C. Jennings (eds.), The Message Session Relay Protocol (MSRP), IETF RFC 4975

[3] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana, Web Services Description Language (WSDL) 1.1, W3C Note 15 March 2001, http://www.w3.org/TR/wsdl

[4] A. Polidoro, S. Salsano, G. Bartolomeo: "Simple Mobile Services for IMS", IEEE Conference and Exhibition on Next Generation Mobile Application, Service and Technologies (Cardiff, Wales, United Kingdom, 2008)

[5] J. Rosenberg, H. Schulzrinne et Al., SIP: Session Initiation Protocol, IETF RFC 3261

[6] S. Salsano, G. Bartolomeo, R. Glaschick, "SMILE (Simple Middleware Independent LayEr) and SMILE-JS (JSON over SIP binding) documentation", available at http://netgroup. uniroma2.it/twiki/bin/view.cgi/SMS/TechnicalReports

[7] S. Salsano, G. Bartolomeo, C. Trubiani, N. Blefari Melazzi: "SMILE, a Simple Middleware Independent LayEr for distributed mobile applications", IEEE Wireless Communications and Networking Conference (Las Vegas, USA, 2008).

[8] JavaScript Object Notation (JSON), http://www.json.org

[9] JXTA-JXME Project, https://jxta-jxme.dev.java.net/

[10] LEAP libraries for JADE, http://jade.tilab.com/

[11] MjSIP Java SIP stack, http://mjsip.org/

[12] SMILE Home Page http://netgroup.uniroma2.it/smile

[13] Simple Mobile Service project http://www.ist-sms.org/

[14] Simple Mobile Service project, trial platform, http://netgroup.uniroma2.it/SmsPlatform