On the Performance of SOAP in a Non-Trivial Peer-to-Peer Experiment

Tom Van Cutsem, Stijn Mostinckx,
Wolfgang De Meuter, Jessie Dedecker*, Theo D'Hondt
{tvcutsem,smostinc,wdmeuter,jededeck,tjdhondt}@vub.ac.be

Programming Technology Lab
Department of Computer Science
Vrije Universiteit Brussel
Pleinlaan 2 - 1050 Brussels - Belgium
Fax: +32 2 629 35 25

Abstract. This paper reports on the experiences we gained while trying to build an interpreter for a new programming language aimed at developing strong mobile software. The interpreter is actually a distributed virtual machine that can be used in a peer-to-peer setting on a heterogeneous platform. In our quest for an experimental implementation, simplicity and portability led us to using a combination of Java and SOAP technologies. The paper reports on the problems we encountered in this experiment and shows that SOAP is inadequate in peer-to-peer communication that cannot afford fat servers to run on all nodes.

1 Introduction

The topic of our research is the design of programming languages that simplify the construction of strongly mobile applications. This fits in what has been called "Ambient Intelligence" (AmI) by the European Council's IST Advisory Group (ISTAG, 2003). The vision of AmI is that soon individuals will be surrounded by a dynamically configured processor cloud running smoothly integrated applications. In this context, we conduct experiments in language design in order to distil language concepts that will facilitate writing mobile applications. The languages we design are typically dynamically typed, reflective and have built-in provisions for distribution and mobility.

The current scion of our language family is called Pic% (pronounced Pic-oh-oh). It is an object-oriented mobile extension of a language family called Pico (D'Hondt, 1996). Pico has been extended in various experimental ways, ranging from distributed agents (Van Belle and D'Hondt, 2000) to objects and delegation (De Meuter et al.,2003). The experiment described here is a unification of these two. Hence, the scion discussed here features:

* Research Assistant of the Fund for Scientific Research Flanders, Belgium (F.W.O.)

- **Minimality**: Ordinary calculus syntax is used and the concept space is restricted to basic values, functions and tables (D'Hondt, 1996): Pico can be seen as an attempt to recover as many concepts of Scheme (Abelson and Sussman, 1985) as possible given a conventional infix syntax restriction.
- **Strong reflection**: All entities, including parse trees are first-class. Moreover, they can be easily inspected *and* changed by the programmer.
- **First-class computational state**: Computations are first-class entities, similar to Scheme **continuations**. This enables *Strong Code Mobility*, such that *running* programs can transparently migrate to another machine[1]. In Pic%, programs can grab their own computational state, transmit it over a wire, and, upon arrival, resume the computation.
- **Prototypes**: Pic% features prototypical objects (De Meuter et al., 2003) which are created 'ex nihilo' (without classes!) or by cloning existing objects. The absence of classes is very adequate in a language aimed at strong mobility since no (transitive closures) of classes have to be transmitted.

We adhere the vision that language design is an iterative experimental activity. In an experimental implementation, test applications can be written in the language. This allows one to detect unforeseen interactions in its concept space. Very often, these are reflected by similar interactions between its implementation components, such as the VM core and the distribution layer in our case. Based on these experiments, the language and its implementation are further polished and another iteration cycle can be entered. In our case, we wanted to experiment in a *realistic* processor cloud constellation with PDA's, PC's, mobile phones, etc. The limited computational power of some of these forced us reconcile *efficiency* with *portability* as good as possible. Thus, our evaluator had to be constructed using portable lightweight technology, which led us to Java[2] and to SOAP (World Wide Web Consortium, 2003) for the networking and mobility of our *experimental* distributed evaluator. The latter choice was made because SOAP (Simple Object Access Protocol) is claimed to be a lightweight object exchange protocol. Furthermore, SOAP is claimed to be deployable in all kinds of network topologies (client-server, n-tier or peer-to-peer) (Snell et al., 2001). Unfortunately this turned out to be untrue. The point of this paper lies in sharing our experience with the SOAP technology and in arguing why we found SOAP to be unsuitable in order to successfully finish our project. We explain why SOAP is not suitable in a peer-to-peer setup with small devices without huge amounts of memory and without fast processors.

In the following section, we explain a bit more about the mobile Pico version in order to give the reader a good feeling of the flexibility we were after. In section 3 we give an overview of the (sometimes hyped) arguments that led us to the technological platform with which we tried to implement our interpreter. In section 4 we give a thorough overview and analysis of the problems we encountered using this technology. Finally, section 5 concludes.

[1] Note that this is much more expressive than *weak mobility*, which is about moving "dead" code (cfr. Java Applets).

[2] We often violated the"rules of good object-oriented practice" for efficiency reasons.

2 Context

As explained our research is about new language concepts for programming distributed and mobile systems in the context of AmI. We believe that current distributed programming languages and middleware solutions (such as Java RMI and CORBA) are too static to match the dynamicity encountered in open distributed environments. Therefore we designed a highly dynamic mobile object-oriented extension of Pico as an alternative (De Meuter et al., 2003). It is not our intent to present its features in detail, but because they influenced our design decisions we briefly explain them in following subsections to give the reader a basic feeling.

2.1 Remote Object Lookup

The first important aspect of our model is how objects find other objects in the network. Given our setting (a heterogeneous, pervasive peer-to-peer environment) we want a system to lookup objects as declaratively as possible. To this extent, our remote object lookup is a distributed generalisation of the lobby concept introduced in Self (Ungar and Smith, 1987). A lobby is an object that denotes a set of processes. Each process can register itself in a lobby and can request other members of a lobby. This is the way a process can lookup other processes in our model. An object is made accessible to remote processes by publishing it under a given name:

```
anObject.publish("alias");
```

To retrieve the remote reference to the service object from `processA` we can write:

```
remoteObject : remoteProcess.alias;
```

2.2 Message Passing

Once we have referenced a remote object we may want to send messages to it. Message sending between local and remote objects happens transparently and is handled by a variation of *wait-by-necessity* (Caromel, 1993). When a message is sent to a remote object, then it is invoked asynchronously (the sender does not block until the message has been performed). The return value of such a remote method invocation is an *awaited object*. When the remote object has received the method call and has computed the result the awaited object *becomes* the real object. The sending process will only wait when the awaited object is sent a message and has not yet *become* the real result. Below is an example of the remote message passing semantics:

```
result = remoteService.perform(aRequest); // does not wait
remoteObj.processResult(result); // does not wait
result.operation() // waits till 'result' is a true object
```

The first expression asynchronously sends the `perform` message to the `remoteService` object and the variable `result` now contains an awaited object. The second expression uses the reference to the `result` object as a parameter. The third expression is a message that is sent to the `result` object. It is only in the last expression that the process will block until the awaited object has become the result that has been computed by the `remoteService` object.

2.3 Mobility

Issues such as partial failures and efficiency can be anticipated using object mobility. In our language all objects understand the **move** message by default. A move method can move any object graph up to a certain cut-off point to any location. This way it is possible to both pull an object to your process as well as to push it to another process. Strong mobility comes for free in our language, because in Pic% the computational state of a process is first-class and also represented as an object (which understands the **move** method). Below is an example of object migration:

```
anObject.move(destination, pruningExpression);
```

The second parameter is a pruning expression that determines what part of the object graph should migrate. Objects that are pruned away are replaced by remote references to the objects that stayed behind on the source process.

3 Architecture and Implementation

Now that we have given a short overview of the distribution and mobility features of Pic%, we can turn our attention to the experimental implementation we built; the main topic of this paper. As already said, Pic% was implemented in Java and the distribution and mobility layer of the interpreter was conceived using SOAP. SOAP stands for Simple Object Access Protocol. It is presented as a lightweight protocol intended for exchanging structured information in a decentralized, distributed environment (World Wide Web Consortium, 2003). The protocol is independent from its protocol binding. HTTP is frequently used as protocol binding, but others such as SMTP can be used as well. The content of a SOAP message is written in XML (World Wide Web Consortium, 2000). A SOAP message contains one main information element, the *envelope* which is divided into several information subelements. The most important subelements are header, body and fault information elements. The *header* specifies the execution directives (such as transaction information) of the message. The SOAP *body* element is optional and contains application specific element information items. A SOAP *fault* element provides a structured way to report various errors that ocurred while processing a message.

One of the design goals of SOAP was the ability to encapsulate and exchange remote procedure calls (RPC). This resulted in SOAP-RPC, a set of rules that specify how a remote procedure call must be embedded in a SOAP envelope.

Our implementation uses SOAP-RPC with HTTP as protocol binding to communicate between the different interpreters of our language. More information on the structure of SOAP messages in the context of our application is given later in this section.

The Apache Software Group developed a library called Apache-SOAP (The Apache Software Foundation, 2001), which is considered to be a modern implementation of the SOAP specification. As said above, we used HTTP as protocol binding, which requires a special kind of HTTP server that supports Web Services (often called an application server). In our experiments, we used the Tomcat (The Apache Software Foundation, 2003) application server . This server is then used by the Apache-SOAP library as the communication layer for the implementation of the Web Service. In our case, the Web Service is a wrapped Pic% interpreter whose methods are remotely invocable. Hence, one Pic% interpreter can "talk SOAP-RPC" (over an HTTP binding) to a Web Service that encapsulates another Pic% interpreter. Figure 1 identifies these different large components that are involved in the communication of two Pic% interpreters.

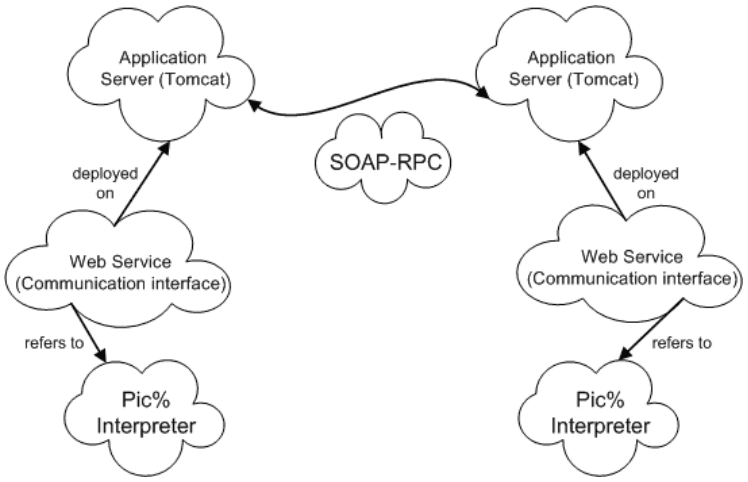

Fig. 1. Distribution Components involving SOAP

The following subsections give a more in-depth discussion of the implementation and the concrete setup of the experiment.

3.1 Implementation Choices

We wanted our language to be a medium for writing experimental, yet *real world* applications. By this, we mean that experimental programs written in Pic% should immediately work in the context of PDAs and mobile phones as well as on traditional PC's. Hence our choice for Java since a virtual machine currently exists for all these platforms.

Another important issue was which networking mechanism to choose from. Although we were no experts in the field of networked objects, we documented ourselves on several existing techniques that were available given the Java restriction. Using documentation, such as a.o. (Snell et al., 2001), we made our decision in favor of SOAP based on the following other arguments:

- *Text-based protocol:* Unlike most of the other available technologies (such as Java RMI and CORBA) SOAP is text-based. This is as a matter of fact an advantage as many of the binary protocols risk getting blocked by firewalls that shield the several *administrative domains* in which most of the internet is currently divided (Cardelli and Gordon, 1998).
- *Independent of implementation language:* The current implementation is done in Java, as it is at this point the most promising platform available for the platforms we target. Nevertheless we thought it would be an extra benefit if we could later on write interpreters in other languages. SOAP will allow us to communicate with interpreters written in other programming languages. This was an important argument not to choose for Java RMI.
- *Independent of transport layer:* Nearly all current SOAP applications use the HTTP protocol. Nevertheless, this is not compulsory so that we can change this to transport protocols that are more oriented towards wireless protocols like Bluetooth or WiFi in the very near future.
- *Standardized communication support:* SOAP enables remote procedure calls which are a well known concept, understood by different packages for all types of languages (World Wide Web Consortium, 2003).
- *Extensible medium:* Since we use XML to present our argument types, we have a degree of flexibility which is harder to achieve using other technologies which generate precompiled stubs.
- *Simple:* As the name SOAP (the 'S' of SOAP stands for simple) and documentation suggests (Snell et al., 2001), the usage of SOAP is claimed to be simple.

In brief, we can say that the portability constraint led us to Java and that the lightweight and simplicity constraints led us to SOAP. In the following sections, we will explain how the Pic% language features explained in section 2 were implemented using this technology.

3.2 Representing Processes

As explained in section 2 a Pic% process can be registered in a lobby, so that it becomes accessible for other Pic% processes. In Java such a Pic% process is represented as an instance of `ProcessServer`. The public interface of such a `ProcessServer` consists of the set of methods making up the Web Service encapsulating a Pic% process. Hence, a `ProcessServer` class implements all methods necessary for interprocess communication. The `ProcessServer` Web Service is an object which is "deployed" on the application server through a so-called deployment descriptor. Such deployment descriptors are XML files containing

several configuration parameters for the service such as a unique identifier, the object's class, the set of invocable methods and so on.

As for Apache-SOAP, it was designed in such a way that its Web Services are indistinguishable from 'normal' Java classes at the source code level. This means that one does not need to write a single line of code to promote a Java object to a Web Service. All necessary information must be given in a deployment descriptor and Apache-SOAP will take it from there. When a web server receives an HTTP `POST` request carrying a SOAP call, it will automatically deserialize arguments into Java objects, call our ordinary Java method and respond by serializing the return value of the method.

3.3 Remote Object Lookup

When the programmer accesses the public fields of a process object (like in section 2.1 when we execute `remoteProcess.alias` to lookup the published object), we construct a SOAP call to the underlying remote `ProcessServer` object. The requesting process spawns a new Java Thread which is going to perform the SOAP-RPC call. The return value is an awaited object (Caromel, 1993) that will eventually become a proxy to the requested object. This reflects the implementation of what was described in section 2.2.

Shown below is the actual message sent by the call Thread when `processB` is requesting a public object called `alias` residing on `processA`. Default SOAP and XML namespaces are replaced by an ellipsis because the messages otherwise become too verbose.

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="..."
                   xmlns:xsi="..."
                   xmlns:xsd="...">
   <SOAP-ENV:Body>

     <ns1:getPublicObject xmlns:ns1="processA"
                          SOAP-ENV:encodingStyle="...">
      <name xsi:type="xsd:string">alias</name>
      <sender xmlns:ns2="..."
              xsi:type="ns2:ProcessId">
        <processId xsi:type="xsd:string">processB</processId>
        <processURL xsi:type="ns2:java.net.URL">
          <value xsi:type="xsd:string">URL to B</value>
        </processURL>
      </sender>
     </ns1:getPublicObject>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The SOAP call contains `name` and `sender` attributes, which contain respectively the alias of the remote object we want to get a reference to and the re-

questing process identification. This process is identified by a pair `(name, url)` that uniquely identifies a process. The SOAP message below is the content of the HTTP response:

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="..."
                   xmlns:xsi="..."
                   xmlns:xsd="...">
  <SOAP-ENV:Body>
    <ns1:getPublicObjectResponse xmlns:ns1="processA"
                                 SOAP-ENV:encodingStyle="...">
      <return xmlns:ns2="urn:picoo.vub.ac.be"
              xsi:type="ns2:RemoteObject">
        <!-- serialized version of a RemoteObject -->
      </return>
    </ns1:getPublicObjectResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The method returns successfully and sends the public object to the requesting process. Even though the serialized representation of the proxy is omitted, we have printed the XML messages here in order to illustrate the amount of data that is actually sent over the network.

3.4 Message Passing

Recall that our programming model specifies messages to remote objects to be sent asynchronously, but always immediately return an awaited value which still might have to *become* its actual value. Nevertheless, in the implementation we resorted to a well-established encoding of synchronous SOAP-RPC calls. As illustrated in the above code, for the remote object lookup this problem was solved using a call thread. We cannot apply this scheme to implement message passing, however, because messages sent to remote objects are not handled by the remote process immediately. Instead, the remote process *schedules* such messages in a queue and handles them whenever computing resources are available. We therefore model asynchronicity of remote messages explicitly by using a *callback* mechanism. As an example, consider the evaluation of `remoteObj.m(x)` from within `processA`, where `remoteObj` represents an object on a remote process, say `processB`. The evaluator on `processA` will first perform a SOAP-RPC call to `processB` to initiate the computation of the message send, supplying a return address as an extra argument.

`processA` maintains a mapping of these return addresses onto `AwaitedObject`s that still have to *become* their real value. Thus, the return address is an identifier for some awaited object which needs to be "replaced" by the function's return value when it is computed. Given this information, `processB` can schedule the request in its queue. When the request is eventually evaluated, the return value of the function needs to be sent back explicitly to the caller.

In short, message passing is made asynchronous by using two synchronous SOAP-RPC calls, one to signal a remote method invocation, the other to return the result.

4 Experiment Results

Throughout the development of the Pic% interpreter, we encountered several weaknesses relating both to the concepts underlying SOAP as well as to the specific libraries we used. This section discusses these problems. In our argument we clearly make the distinction between problems inherent to SOAP and those one might encounter when applying the model using contemporary technology. The latter problems are also important since the software we used is a reflection of the technological state of the art of the field. A developer choosing to use SOAP can be confronted with its technical issues just as likely as its conceptual issues.

4.1 The Client-Server model

The first problem basically boils down to the fact that SOAP is claimed to be applicable in any distribution topology including client-server applications and peer-to-peer. This turns out not to be the case.

Conceptual problems Usually, SOAP provides "service objects" that perform a potentially complex operation, for relatively simple clients. This can indeed be observed by looking at most programs that currently use SOAP. A prototypical example of the usage of SOAP in internet environments is a web service offering the latest stock exchange information. This means there is one *heavyweight* service object, containing a database of stock quotes of a given stock exchange, updated on a regular basis. *Lightweight* clients can then query for the stock quote of a company, e.g. using a company's ticker tape symbol. Other concrete examples include Amazon's [3] SOAP interface for retrieving product information and Google's [4] SOAP interface for retrieving search results. In all these cases the topology is basically client-server with heavyweight servers and lightweight clients. This is not a coincidence. Its (essentially service-based) design, clearly reveals that SOAP was basically designed to offer 'fat' services that perform tasks for 'thin' clients rather than for a truly interoperating set of collaborative entities. Such a setup, which should clearly be our aim given our context of communicating interpreters, requires an architecture that SOAP apparently cannot offer, despite its promising claims. After all, in our setting, SOAP forces every Pic% interpreter to be a (heavy) web service, since each interpreter must be able to act both as a server and as a client. This is too heavyweight a setup for our purposes. The technology is inherently too complex in order to run on small devices with limited computing power.

[3] `http://www.amazon.com/webservices`
[4] `http://www.google.com/apis`

Technological problems In the Apache-SOAP library, the most natural way to write a distributed application is by using HTTP as protocol-binding. This automatically implies a client-server architecture and that each embedded device needs to run an application server. Another possibility is to use SMTP as protocol-binding which would worsen the situation because we would need to run both an SMTP and a POP server.

The cost of the innate client-server architecture is further aggravated by the fact that Java objects which act as a SOAP service must be **deployed** on an application server, like Apache Tomcat. The application server runs a Web Service, which unfolds the HTTP requests and passes them on to the SOAP layer. Hence, before communication between two computer nodes can occur we have to pass through three layers:

1. the application that needs to communicate (a Pic% interpreter) has to pass communication through the web service.
2. the web service on its turn has to interact with the application server.
3. the application server has to interact with the low level communication layers, such as TCP/IP.

This need for layers immediately presents performance problems of using SOAP in the context of lightweight mobile machines, which do not always have the computing power to run an application server with deployed web services listening for incoming HTTP requests.

4.2 XML serialization

The second problem basically boils down to the fact that XML is not very well suited for encoding object graphs to be transported over a network.

Conceptual problems Choosing SOAP as a communication medium between components implies using XML to encode the data being communicated. Using XML has the obvious advantage of ensuring portability across platforms and languages. One could use a proprietary protocol and wrap this data in an XML message, but this would nullify the reasons for using XML or even SOAP in the first place. Converting objects to another representation for the purposes of storing them on disk or sending them over a wire is called **serialization** (also called *pickling* or *marshalling*). Using XML for this in our context implies that we have to be able to *represent* any first-class value of our programming language in XML, as any such value can be transported over a wire to another interpreter We thus faced the problem of having to serialize any Pic% value (such as a number, a function, an array of strings, the runtime stack, ...) to XML. This is more problematic than one would imagine at first sight.

First, Pic% objects, like most objects in other object-oriented languages, basically consist of data fields and methods, more generally known as slots. Since objects can contain other objects in their slots, objects and other values are connected with each other in a graph-like manner (a so-called *object graph*). Of

course, this object graph may contain cycles, meaning there are objects pointing (directly or indirectly) to themselves. A serialization algorithm should be able to cope with such cases. Unfortunately, XML is designed to describe **tree** structures. **Graph** structures, however, are more complex and require the use of "pointers" to avoid the duplication of graph nodes. Serializing such object graph results in pretty complex, but – more importantly – in very large XML files. We have experienced more than a factor 10 when going from a pointer representation to an XML representation.

But the situation is even worse. One must make sure that, when reconstructing an object graph from the XML representation on the receiver side (called **deserializing** an object), object identity is maintained. As an example, consider an object o having two slots, one containing some value $v1$, and the other containing a value $v2$. If $v1 == v2$ before o is sent over the network, then it should hold that $v1' == v2'$, given that $v1'$ and $v2'$ are the reconstructed versions of $v1$ and $v2$. In general, any two objects pointing to the same object before serialization should also point to the same object after deserialization. This requires a complex encoding of 'pointers' in the XML files sent around. But apart from the complexity it also poses some serious conceptual problems. An object that reaches a process in two different ways should still be equal to itself. This requires an encoding of pointers in XML that is 'globally consistent' over different machines. Assuring this global consistency actually means implementing a distributed memory management system in XML!

Technological problems The Apache SOAP library was very minimal in its support for serializing Java objects to XML. Therefore, we had to write our own serialization algorithm capable of safely (i. e. avoiding aforementioned pitfalls) transporting any Pic% object graph across the network.

Apache-SOAP provides standard serializers to map primitive Java types and Arrays, Maps, Dates etc. into XML using standard SOAP encoding (see World Wide Web Consortium (2003), section 5). It also provides the necessary deserializers to transform the XML back into the proper Java objects. However, when (de)serializing arbitrary Java classes, things get more difficult. Apache-SOAP provides a generic *BeanSerializer* capable of (de)serializing arbitrary Javabeans. This serializer was impractical for us to use for a number of reasons:

- Our implementation classes do not adhere to the Bean model in that we do not allow (for security reasons) all instance variables of our classes to be accessed or changed by accessor or mutator methods, and that we do not want to provide no-args constructors for them. Writing accessors for every instance variable would break encapsulation of Pic% objects. In a mobile context this kind of security breaching is totally unacceptable.
- Since we have knowledge of the structure of the classes that we serialize, a dedicated (de)serializer would outperform the generic Bean serializer.
- Some Pic% objects require special serialization to preserve object identity. Other objects can be singletons, requiring the deserializer to return just the existing singleton instance instead of creating a new one.

Because of all these technical problems, we ended up writing a serializer capable of (de)serializing specific Pic% interpreter objects. Dedicated (de)serializers were written for Pic% objects that required special (de)serialization needs. This serializer maintains object identity and handles circular structures. Although not impossible, all this code put extra burden on the machinery it is supposed to run. This was no longer adequate in the context of the lightweight devices we are targeting.

4.3 XML and Typing

Another issue in the transformation between object graphs and XML documents in a statically typed language like Java are **typing** problems. Unfortunately, Apache SOAP bases its serialization on *static* types as specified in method signatures. Of course, in an object-oriented setup that uses inheritance, serialization should logically be performed on *dynamic* types since one wants the 'real' object to be serialized and not only the part indicated by the static (abstract) type it is assigned at that particular moment. When Apache SOAP wants to serialize an object, it retrieves an associated serializer for such objects based on the static type of the variable in which it resides. We therefore explicitly had to override framework methods to circumvent this strategy and to retrieve serializers based on the dynamic type. This is necessary since e.g. many methods in our language implementation operate on abstract classes. One cannot write serializers for such abstract classes. Moreover, subclasses may require special serialization behaviour which cannot be expressed at the level of the superclass.

4.4 Performance Issues

Last, but not least, as already suggested a few times, SOAP (and XML) suffers from some serious performance problems.

Conceptual problems Using SOAP in our setting gives rise to two performance bottlenecks. The first is the limited amount of bandwidth usually available for communication between small mobile devices. To circumvent this drawback, small-size messages and data representation are essential. But, obviously, XML is not a very compact data description language. As already indicated, we observed a factor 10 difference between a tree and its XML representation.

Another drawback of SOAP is that the use of XML leads to speed performance penalties both due to the construction of XML documents and due to parsing them back to object graphs upon reception. In the first part, a lot of verbose information has to be written to the XML document which would not be included in a binary serialization. Concerning the second part, parsing XML is a costly operation. Indeed, apart from the program logic that actually deserializes the flattened object graph, there is also a lot of parsing code active that is merely about XML parsing. Actually, XML puts an extra 'parsing indirection' between the object graph representation and its flattened representation. The point here is that SOAP's use of XML implies generality and thus a larger overhead in

parsing. This imposes a significant performance bottleneck on SOAP message reception. Thus, binary protocols outperform XML not only in size but also in speed. Lightweight devices like cellular phones will probably not be able to cope with such costly operations when large object-graphs are transmitted.

Technological problems Apart from these conceptual problems presented abstractly, we can shed some light on the concrete difficulties we encountered in our implementation. In the following, we show that the size and overhead associated with the XML representation of Pic% objects is *really* substantial.

When transmitting Pic% values, every node in the XML structure requires:

- an **xsi:type** element denoting the **dynamic** Java type of the serialized object, qualified by an XML namespace denoting the encoding style used.
- a **picooId** element uniquely identifies a given XML part. It acts as an address to which one can refer later on in the XML document in the case of multiple references. This was already explained in section 4.2.

The following XML excerpt shows part(!) of a SOAP body, representing a serialized version of the Pic% function `f():void`. This is about the simplest function we can write in Pic% as it is a function without arguments that always returns the `void` value, Pic%'s null-value. The generated XML is incredibly verbose:

```
...
<argument xsi:type="ns2:edu.vub.picoo.grammar.AGFunction">
    <_picooId_ id="1"/>
    <name_ xsi:type="ns2:edu.vub.picoo.grammar.AGText">
        <_picooId_ id="2"/>
        <text xsi:type="xsd:string">f</text>
    </name>
    <parameters xsi:type="ns2:edu.vub.picoo.grammar.AGTable">
        <_picooId_ id="3"/>
        <table xmlns:ns3="http://schemas.xmlsoap.org/soap/encoding/"
               xsi:type="ns3:Array"
               ns3:arrayType="ns2:edu.vub.picoo.grammar.PicoValue[0]">
        </table>
    </parameters>
    <body xsi:type="ns2:edu.vub.picoo.grammar.AGVoid">
        <_picooId_ id="4"/>
    </body>
</argument>
...
```

Although we have not performed any scientifically founded measurements, one has to admit that the amount of data actually transmitted is tremendous compared to the simplicity of the Pic% function. As good as all experiments we conducted seem to indicate that, in general, an XML representation is at least 10 times bigger than the corresponding binary representation.

5 Conclusion

The long term goal of our research is to design small and conceptually clean programming language features that are dedicated to the construction of distributed and strong mobile systems in the context of Ambient Intelligence. Having established an initial design of a language, our experimental vision on language design demanded us to construct an experimental interpreter for it as soon as possible. Even though we were willing to make some performance sacrifices as a trade-off for portability and simplicity, we wanted our implementation to be really usable on small mobile devices. Although we do not claim to be experts in middleware technology, the available (commercial) literature led us to using Java and SOAP. Unfortunately, SOAP did not prove to be suitable in this context due to its inherent client-server architecture which requires deploying program classes on a separate web server, and due to the inherent weaknesses of XML when it comes to performance and expressivity. In the context of our restrictions we cannot help but conclude that SOAP is not only a simple but also a simplistic object access protocol.

References

Abelson, H. and Sussman, G. J. (1985). *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA.

Cardelli, L. and Gordon, A. D. (1998). Mobile ambients. In *Foundations of Software Science and Computation Structures: First International Conference, FOSSACS '98*. Springer-Verlag, Berlin Germany.

Caromel, D. (1993). Toward a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102.

De Meuter, W., D'Hondt, T., and Dedecker, J. (2003). Intersecting classes and prototypes. In *Proceedings of PSI-Conference*. Springer-Verlag.

D'Hondt, T. (1996). The pico programming language project. `http://pico.vub.ac.be`.

ISTAG (2003). Ambient intelligence: from vision to reality. Draft report.

Snell, J., Tidwell, D., and Kulchenko, P. (2001). *Programming Web Services with SOAP*. O'Reilly.

The Apache Software Foundation (1999-2003). The tomcat 5 servlet/jsp container. `http://jakarta.apache.org/tomcat/tomcat-5.0-doc/index.html`.

The Apache Software Foundation (2001). Apache soap v2.3.1 documentation. `http://ws.apache.org/soap/docs`.

Ungar, D. and Smith, R. B. (1987). Self: The power of simplicity. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 227–242. ACM Press.

Van Belle, W. and D'Hondt, T. (2000). Agent mobility and reification of computational state, an experiment in migration, in: Infrastructure for agents, multi-agent system, and scalable multi-agent systems. *Springer Verlag Lecture Notes in Artificial Intelligence nr. 1887*.

World Wide Web Consortium (2000). Extensible markup language (xml) 1.0 (second edition). `http://www.w3.org/TR/REC-xml`.

World Wide Web Consortium (2003). Simple object access protocol (soap) 1.2 w3c note. `http://www.w3.org/TR/SOAP/`.