

Mobile Code Enabled Web Services

Pu Liu and Michael J. Lewis

Department of Computer Science, State University of New York (SUNY) at Binghamton
{pliu1, mlewis}@binghamton.edu

Abstract

A primary benefit of Web Services is that they provide a uniform implementation-independent mechanism for accessing distributed services. Building and deploying such services do not benefit from the same advantages, however. Different Web Services containers are implemented in different programming languages, with different constraints and requirements placed on the programmer. Moreover, client side programmers must use the Web Service interface specified by the service developer. Therefore, the kinds of applications and uses for a Web Service are unnecessarily restrictive, constrained by the granularity of access defined by the interface and by the characteristics of the service functions. This paper describes an approach that addresses both of these drawbacks by enabling Web Service containers with the ability to accept new mobile code on the fly, and to run it within the containers, providing direct local access to the containers' other services. The code can be specified in a small simple language (a subset of C), and translated and passed to the container in a common XML-based intermediate language called X#. This approach effectively removes the dependence on any single implementation environment. Our prototype implementation for two different containers demonstrates the feasibility of the approach, which represents a first step toward write-once deploy-anywhere Web Services.¹

1. Introduction

Web Services provide a standard means to achieve interoperability between different software applications in a heterogeneous distributed environment [6]. XML and standard Web protocols such as HTML and SOAP

have made Web Services robust, extensible, and independent of platform, language and framework.

To access a Web Service, client programmers discover and obtain a description of that service in the form of a Web Services Description Language (WSDL) description. Typically, a stub generator then creates client-side stubs for accessing the remote service. The stubs provide access to the service by sending requests to the remote server and fetching the results. Because the WSDL file and the SOAP communication protocol are both based on XML, the client is able to access the Web Service ubiquitously.

The interoperability and expressiveness afforded by XML-based standards come at the expense of performance [4, 5]. A typical Web Service invocation and corresponding response require serialization, communication, and de-serialization, in both directions. Clearly, this associated overhead makes remote Web Service access significantly slower than a local method call. If the client code could run in the address space of the Web Service itself, then the serialization and access could be much more comparable to that of a local function call. The overhead then would depend on the one-time cost to send, install, and run the client's code, instead of the time to serialize and transmit the data associated with however many calls the code contains. Potential performance improvements can be achieved when clients make many calls, send lots of data, or both. This is the primary motivation for mobile agents and mobile code [7, 8, 9, 10]

In this paper, we describe an approach that enables Web Services containers to accept new mobile code on the fly. This code is incorporated into the container dynamically, and runs local to the server, rather than in the client's address space. Three distinct advantages can be gained from this approach:

(1) *Moving code to data*: First, as motivated briefly above, a mobile code component can contain multiple calls to the Web Service, thereby conserving bandwidth by reducing the number of remote calls to the service, and replacing them by local calls. Data that needs to move from the server to the client need not be put on

¹ This research is supported by NSF Career Award ACI-0133838 and DOE Grant DE-FG02-02ER25526.

the network. This makes more fine grained Web Service interfaces more appropriate, thereby increasing the situations in which Web Services can be used.

(2) *Callbacks*: Second, clients can send mobile code that accepts a notification message sent from within the container into which the new code is deployed. This allows Web Service programmers to build more fine grain notifications, since they can be caught and filtered by services that run within the same container. Therefore, a notification need not imply a remote message. The mobile code that accepts the notification can decide when and where to generate an “external” notification event back to the original client.

(3) *Dynamic Deployment*: Third, the mobile code can enable the dynamic deployment of new components in a Web Service. In the static deployment strategy, updating or upgrading even a single component of a Web Service can be expensive. The new service must be built, the existing service must be stopped, and the new one must be deployed and run in its place. State information that should remain across the update must be written to stable storage and read back in, or otherwise transferred to the new version of the service. With our new approach, dynamically deployed services can continue running while programmers update them.

However, simply introducing platform dependent or language dependent mobile code techniques into Web Services would undermine Web Services’ basis for success. Any such approach should be independent of language and container environment.

In this paper, we describe our approach to enabling Web Services containers with the ability to accept new mobile code on the fly. Clients write their mobile code in C--, our simplified C language, and then use a tool to convert it into our XML-based intermediate mobile code language, called X#. Our approach removes the dependence on any single implementation environment. A module running within each mobile code enabled container translates the X# mobile code into native code in the language appropriate for that container. The container then compiles and dynamically incorporates the component as a new Web Service. The newly deployed code can then interact with the container directly to fulfill clients’ requests. Because the mobile code is compiled, it runs as fast as if it had been deployed to the container in the standard way. Currently, we have successfully applied our approach to both the Microsoft ASP.NET Web Services container and the Tomcat Java Web Service container.

This approach comes with a cost, and clearly is not appropriate under some circumstances. Our performance study in Section 4 demonstrates the

circumstances under which this approach can be beneficial to Web Service clients, for a simple illustrative example.

The primary contribution of our work is in providing Web Services programmers with the ability to write their code once, in our simple language, and deploy it into a number of different kinds of containers (any with the facility to support X# code). Programmers need not know and understand the implementation characteristics of the server-side container. In this sense, our work represents a first step toward write-once deploy-anywhere Web Services. Client programmers can build applications that run in the address space of the server and avoid the cost of remote transmission on every call; server programmers can deploy new services, or update old services, on the fly, without shutting down existing instances.

Security aspects of our approach are not described in this particular paper. We assume that Web Services can restrict access to functionality by associating potential clients with capabilities or rights to access to particular services, and that this functionality can be used to restrict access to extension services to only trusted clients.

The remainder of this paper is organized as follows. Section 2 compares our work with related approaches. Section 3 presents the design of our new languages (C-- and X#) and our approach to mobile code enabled Web Service containers. Section 4 characterizes performance, and Section 5 summarizes our contributions.

2. Related Work

Web services and mobile agents can be combined in two fundamentally different ways. Mobile agents can simply be utilized within a Web Services architecture, or Web Services can themselves be implemented to act and behave as mobile agents.

In the first category, for example, Maamar et. al. [12] use mobile agents to carry out the composition and execution of Web Services in parallel in a dynamic environment. This helps overcome the drawbacks of composing Web Services in a sequential multi-stage manner. Shonali et. al. [13] use mobile agents to select the most efficient Web Services for the clients in a dynamic environment. In both cases, the Web Services themselves are not changed. Instead, the application simply utilizes them as just another of its components. In this sense, the Web Services are complementing other separate and disjoint mobile code mechanisms, not implementing or directly supporting them.

The work in the second category, including ours, integrates mobile agents and Web Services more closely, leading to “Mobile Web Services.” A Mobile Web Service can migrate between containers, and can interact with other existing Web Services and clients. Its internal functionality helps determine when and where the service migrates.

Fuyuki [14] proposes a general framework for Mobile Web Services. The prototype of this framework utilizes BPEL to specify the interactions with other Web Services or clients, and uses Java to implement internal behavior. The resulting mobile Web Service is dependent on the platform and limited to Java. The authors mention that they are looking for a mechanism to achieve interoperability between mobile code and different containers, which is precisely the functionality that our approach provides.

Maamar [15] uses an agent-based architecture to provide Web Services to a mobile device. To provide clients with seamless access to Web Services, a Web Service is transferred from its host site to a mobile device and executes there. IBM Aglets [8] are used to implement mobile agents for service invocation at resource sites, with Sun Java 2 Platform Micro Edition running on the mobile devices. The ServiceGlobe system [23] also supports Java-based mobile Web Services. Again, these systems cannot be applied to non-Java-based services.

CLI [17], ELF [18], and Java [19] all enable on-the-fly composition of application components. But with Web Services, explicitly specifying the programming language is a severe restriction that violates the foundations of Web Services, whose success relies on language and platform independence, and accessibility.

Our platform and language independent mechanism to dynamically incorporate components could enhance the work described above. Since X# is based on XML, it can be processed by any language that can process XML, which is a basic requirement to implement Web Services in the first place.

Another straightforward way to allow the same code to be incorporated into different Web Services containers is to translate the mobile code in one language into the container’s implementation language. For example, if the mobile code is written in Java and the container in C, a translator could convert the Java mobile code to C. But this one to one source translation is not scalable since n different languages would need $O(n^2)$ translators.

As an improved approach, we can introduce an intermediate language and a corresponding virtual execution system. Components written in different languages would first be compiled into the intermediate

language and then deployed in the virtual execution system. The resulting component written in different languages would only depend on this virtual execution system. Moreover, with this virtual execution system, for n languages in m platforms, we would need only $m+n$ compilers to generate the final executables instead of $m*n$. But the use of virtual execution system introduces extra overhead compared with one-to-one language transforming. In some platforms with limited resources, utilizing such powerful but complex virtual execution systems is not feasible. Our approach introduces an intermediate language without the virtual execution system. This helps achieve both scalability and simplicity.

BPEL [19] possesses some of the beneficial features of X#. BPEL is a pure XML-based language that is used to compose multiple Web Services into an end-to-end business process. Clients submit a script written in BPEL to a BPEL execution engine in which the script is interpreted and executed. The execution engine can publish the BPEL script as a Web Service. BPEL is mainly used to choreograph Web Services. Thus, BPEL is not designed as a full-fledged programming language for mobile code. A BPEL alternative called BPEL4Java adds Java code snippets into BPEL to complement a BPEL script for complicated choreography. But using BPEL4Java as mobile code again depends on Java, which our approach does not.

3. Mobile Code Enabled Web Services

The basis of our approach to enabling mobile code in Web Services is the design and definition of a common intermediate language for the expression and transmission of mobile code. We call this XML-based language X# (pronounced “X sharp”). We do not intend programmers to directly write their code in X#, instead, they will typically use a high level language of their choice, and use a translator to map it into the equivalent X# code. For illustrative purposes, and to handle simple mobile code programs, we have designed the C-- language. C-- is a simple subset of C for which we have built an X# translator. Thus programmers can write mobile code in C-- and translate it directly into X#.

Once the mobile code is in X#, it can be passed to any Web Services container that has been instrumented to accept and support it. We have augmented two such containers, Tomcat and ASP.Net, with this functionality. Each Web Service container runs a special service whose purpose is to accept mobile X# code. The service then translates the code out of X# and into the language appropriate to the container. For

Tomcat, this language is Java, for ASP.Net, it is C#. The container-specific code is then compiled just as any other new Web Service that targets that container would be, and is installed at the container. Therefore, once it has been installed and deployed, the mobile code runs as efficiently as it would if it had been deployed “statically.”

Thus, our approach requires two kinds of translators, one for converting programmer-written mobile code into X#, and one for converting X# back out into the high level language appropriate for the container into which it will run. As shown in Figure 2, our implementation contains three different compilers, one for C-- to X# translation, one for X# to Java (for Tomcat), and one for X# to C# (for ASP.Net). Separate subsections below describe the design of X#, the design of C--, and the translation of X# out into container specific languages.

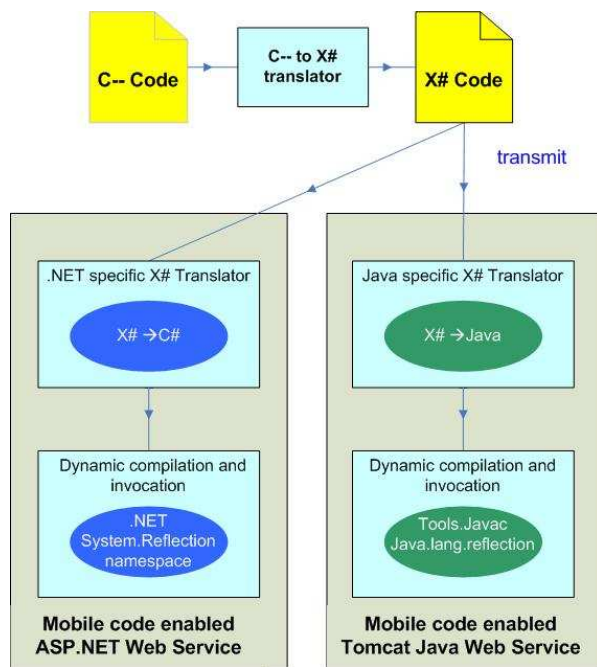


Figure 1: System Architecture: Programmers write code in a high level language, translate it into X#, and transmit it into any of several X#-enabled containers, where it runs for the client.

3.1. The X# Mobile Code Language

In keeping with other Web Service protocols, which are designed for interoperability, portability, and platform independence, X# is based on XML. X# is flexible enough to be generated from many different high level languages, and expressive enough to be

translated back out into many others, including object oriented languages.

X# code takes the form of an Abstract Syntax Tree (AST), which captures the essential structure of the code. The AST omits unnecessary syntactic details, including, for example, semi-colons to terminate statements and commas to separate function arguments. Productions in the grammar are directly represented in ASTs by the structure of the tree. Since XML is designed to tag structured data[22], and XML tools are designed to manipulate those data, XML and XML tools are well suited to implementing an X#-generating compiler. For runtime execution, X# code could be either interpreted node-by-node or converted to other languages for more efficient execution. Our prototype implementations take the latter approach.

X# includes basic object-oriented language constructs. In X#, a program consists of one or more modules, each of which can have one or more Class definitions. A Class includes fields, properties, and methods. X# supports flow control constructs including iteration and branching statements. X# supports static typing and includes the primitive data types and arrays expressible in WSDL.

X# supports two different types of method invocations; X# code can call other methods implemented in the same X# code segment, or methods implemented in other Web Services in the hosting container. This second type of invocation is described by an XML element that indicates the port type and operation in WSDL associated with the function. Such information serves the purpose of an external function definition, and is used by the compiler to convert the X# code to Web Services implementation languages like Java or C#.

Other implementation information necessary for calling external functions includes, for example, the DLL file name (in Windows) or Jar file name (in Java) together with the referenced class name and method name for the operation that needs to be called. The compiler converts the X# calling code to a method call expressed in the appropriate implementation language. X# mobile code only specifies what to do not how to do it. Late binding attaches mobile code to the implementation in the hosting container instead of on the client side.

3.2 The C-- Mobile Web Service Language

Although X# is powerful and expressive, it is not easily written directly by programmers; X# is better utilized—and designed to be—an intermediate language. Clients programmers should write mobile

code in their favorite languages and then use compilers to convert them to X# before transmitting the mobile code to the hosting Web Services container. For simple Web Service extensions and mobile code snippets, we have developed a simple language called C--, to verify the feasibility of our approach.

C-- is a subset of the C language. Its prototype implementation supports primitive types: *int*, *char*, *bool*, *double* and *void*. C-- supports one-dimensional arrays, function calls, variable declarations, function definitions, while loops, if statements, binary operators like +, -, *, /, %, &, and |, and unary operators including ~ and ^.

C-- code allows programmers to invoke a service deployed in a potential container, just as she would write code to call a local function. The only difference is that such functions are declared as “extern”, and a tool generates the signatures of the extern functions by parsing the WSDL file of the Web Service. The C-- code then is translated to X#. The compiler will include all the necessary information in X# for further processing.

3.3 Deploying X# Into Containers

Once mobile code exists in X#, it can be transferred directly to any container that supports the ability to accept and deploy it locally. The first requirement of a container that can do so is that it supports the appropriate Web Service extension interface. This interface contains three primary functions, *installCode()*, *invokeCode()*, and *evaluate()*. The Java definitions of these functions are presented below:

```
public ReturnValue installCode  
(String XsharpCode)  
throws RemoteException;  
public ReturnValue invokeCode  
(String MethodName,  
Object[] params)  
throws RemoteException;  
public ReturnValue evaluate  
(String XsharpCode,  
String entry, Object[] methodParams)  
throws RemoteException;
```

The *installCode()* function deploys the mobile code into the container, *invokeCode()* calls a function in the mobile code using a dynamic invocation interface (DII) like mechanism, and *evaluate()* temporarily installs the code and calls it once, but the code does not persist for other clients to use.

The class *ReturnValue* wraps up the success flag, result, and the necessary information. Its definition is:

```
public class ReturnValue {  
    public boolean success;  
    public Object result;  
    public String info;  
}
```

The *Object* class type in Java is described by “anyType” in WSDL. In the WSDL Schema specification, “anyType” could be any type defined in W3C WSDL schema. When one end sends out the SOAP message on the wire, the actual runtime type names are attached to the message such that the other end can handle it appropriately. Both Java and .NET Web Services tools can handle “anyType” correctly.

The challenge of interface design is how to provide the extensible functionality to Web Services without changing the WSDL. Using static invocation, a stub is created and linked into the client code. Clients use the stub to invoke Web Services as if they were calling a local function. The drawback is that if the WSDL changes, this stub needs to be regenerated, along with the client program.

As an alternative, clients can utilize dynamic proxies or the dynamic invocation interface (DII) to invoke Web Services. When using dynamic proxies, stubs are created at runtime. The dynamic proxy client does not rely on the statically precompiled code. With DII, a client can call a remote procedure even if the signature of the remote procedure or the name of the service is unknown until runtime. These two approaches are useful when clients need to dynamically invoke Web Services. However, the source code for a DII client or a dynamic proxy client is more complicated than the code for the static stub clients.

To allow clients to access the extended Web Services through static stubs, we use Web Services reflection. Similar to reflection in languages like Java and C#, Web Services reflection provides a uniform interface for clients to access the dynamically loaded mobile code. Because the interface is static, clients can use the static stub to access the dynamic mobile code on the hosting Web Services server. The overhead for this approach is that it is the responsibility of the client to explicitly check the types of the parameters and the return value.

Once a container receives new X# mobile code through the *installCode()* (or *evaluate()*) function, it must make that code available within the container. To do so, it first converts X# code to the container specific implementation language. For example, in Tomcat, the

X# code is converted to Java; in .NET, the X# code is converted to C#. During conversion, the compiler needs the reference name and location of the original service implementation components so that the generated code can access the implementation components directly.

3.4 A Simple Example

Consider the following C# code snippet:

```
Extern int add(int, int);
//Extern indicates that add() is a late binding
//function implemented by the hosting Web Service
int i=5;
int j=6;
int sum = add(i,j);
//invoke add() just like a local call
```

The X# code will contain the following information for late binding:

```
<MethodInvoker
  MethodName="add"
  InvokeThrough="Remote">
  <Parameter>
    <VariableReference VariableName="i"/>
  </Parameter>
  <Parameter>
    <VariableReference VariableName="j" />
  </Parameter>
</MethodInvoker>
```

Setting the value of attribute "InvokeThrough" to "Remote" indicates that the X# compiler needs to fulfill this call by calling a service in the remote container. On the server side, the X# compiler must create the "remote" calling code on the fly for the add() function. For example, in our mobile code enabled ASP .Net container, which uses C#, suppose the implementation component DLL is mathService.dll and the reference name is MathService.MathServiceImpl. The resulting C# class is then:

```
public Class SimpleMath: Object {
  private
  MathService.MathServiceImpl Impl1;
  public SimpleMath() {
    Impl1 = new MathService.MathServiceImpl()
  }
  public void mobile_code() {
    ...
    sum = Impl1.add(i,j);
  }
}
```

Here the class MathService.MathServiceImpl instance "_Impl_1" provides the capability for add() to fulfill the caller's request. The generated C# code will be further compiled into a DLL, which will be used in invocations of the mobile code.

After the container successfully installs the mobile code, clients can invoke it through "invokeCode()", published by the hosting Web Service. When a request for that invokeCode() arrives at the container, it will invoke the corresponding method in the DLL or Jar files generated in the previous steps, and return the result back to the clients in the form of a SOAP message. The process is the same as with dynamic invocation through reflection. The evaluate() operation essentially combines the functionality of the installCode() and invokeCode() operations into one step, but does not make the code available through the public interface of the container.

4. Experimental Results

We have implemented a suite of tools and libraries to develop mobile code enabled Web Services in both the Microsoft ASP .Net Web Services container and the Tomcat 5.0 Java Web Service container. For illustrative purposes, and to experimentally evaluate the overhead introduced by mobile code, we implemented a simple mobile code enabled "math" Web Service. We show that the overhead is low, especially considering the potential performance gains derived by the use of mobile code. The sample math service publishes four basic operations: *Add*, *Subtract*, *Multiply*, and *Divide*. All operations take two integers as parameters and return an integer result.

4.1 Experimental Environment

The client runs Windows XP sp2 on a Dell Laptop Inspiron 4150 with Intel Pentium 4 1.6 GHz processor, 512 MB of DDR SDRAM at 266 MHz, and a 30G Ultra ATA/100 hard drive. The client resides on the same subnet, connected by 100 Mbps Ethernet.

The first server runs version 1.1.4322 of the ASP .Net Web Services container in Windows XP sp2. The second server runs the Tomcat Web Service container in Debian Linux version 2.4.24. Both servers run on a Dell Dimension 4500 with Intel Pentium 4 2.26GHz processor, 1GB of DDR SDRAM at 266 MHz, and an 80GB Ultra ATA/100 hard drive.

JWSDP 1.5 is used to generate the WAR file for the math Web Service. The Java client uses Java version "1.4.2" Java™ 2 Runtime Environment, Standard Edition (build 1.4.2-b28) Java HotSpot™ Client VM

(build 1.4.2-b28, mixed mode). The Java server uses Java version “1.4.2_04” Java™ 2 Runtime Environment, Standard Edition (build 1.4.2_04-b04), Java HotSpot™ Client VM (build 1.4.2_04-b04, mixed mode).

4.2 Analysis

First, we derive an equation to determine the circumstances under which using mobile code can gain performance improvement. We define T_{eval} to be the total time to make a single Web Service call. This time is divided into two parts. O_{ws} is the overhead associated with a call, including serialization, de-serialization, and communication. T_{comp} is the average time spent only on computation within the service. So, $T_{eval} = T_{comp} + O_{ws}$

We define O_{mc} to be the average overhead associated with mobile code, including X# code parsing and dynamic code compilation. If the mobile code fulfills a task that requires N Web Service calls and assuming T_{eval} is the average time to finish one eval() Web Service invocation call with this mobile code, we have:

$$T_{eval} = N * T_{comp} + O_{ws} + O_{mc} \quad (1)$$

For Web Services without mobile code, the total time is

$$T_{eval} = N * (T_{comp} + O_{ws}) \quad (2)$$

From (1) and (2), we can conclude that if

$$N > O_{ws} / O_{mc} + 1 \quad (3)$$

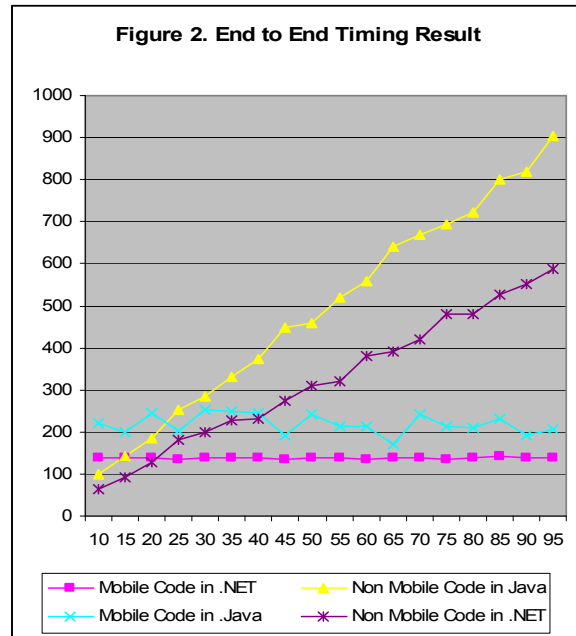
then the overall time of using mobile code is less than that of without using mobile code. Note also that this is independent of T_{comp} . Thus, deciding to send client code to run in the server depends on how many calls to the services can be encapsulated in that code, and the difference between the overhead of making a remote call vs. compiling and installing the service.

Based on the analysis, we used code to calculate the average of all elements in an array to measure mobile code overhead (O_{mc}) and to determine the value of N necessary to make the mobile code approach worthwhile in this case. Again, this is used for illustrative purposes, not because this is necessarily a realistic Web Services example.

If the number of elements in the array is E , then in calculating the average value in the array, we first invoke the Add() operation E times to get the sum of the array. We then invoke the Divide() operation to get the result. In this example, the number of remote calls necessary, if mobile code is not used, is $N=E+1$. And because the time spent on the computation of Add() and Divide() is trivial, the total used time in each case is essentially equal to the overheads.

The end-to-end timing results, comparing the mobile-code approach to the standard non-mobile-code

approach, are summarized in Figure 2. The number of Web Service calls N appears in the X-Axis, varying from 10 to 95. From the figure, we see, when $N > 15$, the mobile code enabled approaches in .NET (For Java, $N > 25$) Web Service containers exhibit better performance than non mobile code enabled approaches. The difference grows with N . However, when $N < 15$ in .NET and $N < 25$ in Java, the gain from mobile code did not compensate for the mobile code processing overhead. This situation happened when the overhead of Web Service calls are small (less than 16 ms). When Web Service call overhead is higher, we would expect



a smaller N .

The average installation times in .NET container is 204 milliseconds and in Java container 181 milliseconds. The average invocation times in .NET container is 12 milliseconds and in Java 16 milliseconds. The average time to invoke one Web Service is 11 milliseconds in the .NET container and 14 milliseconds in the Java container. Although the installation overhead is significant, it is a one-time operation. The overhead of the following invocation times is less than 2 milliseconds compared with the non-mobile-code approach. Considering the gain of flexibility from mobile code, this is acceptable.

5. Summary

We describe an approach to enabling Web Services containers to accept and run mobile code. Programmers express code in a simple subset of C, and their code is translated into a new XML-based mobile code language called X#. X# code can then be deployed

directly to containers that are equipped to accept and install it, through a simple common interface that all such containers export. The code is translated back out into the container-specific implementation language, compiled into an implementation component, and deployed locally, where it has direct access to local Web Service functionality. Our implementation works for both ASP.Net and Tomcat containers, with performance surpassing that of multiple remote calls for a relatively small number of Web Service invocations. Our approach equips Web Services with the beneficial functionality of mobile agents, allowing more fine grain Web Services to be defined and deployed by Web Service developers, since they now can be invoked by locally deployed mobile code, not necessarily just from remote clients.

6. References

- [1] N. Abu-Ghazaleh, M.J. Lewis, and M. Govindaraju, "Differential Serialization for Optimized SOAP Performance," in *Proceedings of HPDC-13: IEEE International Symposium on High Performance Distributed Computing*, Honolulu, Hawaii, pp. 55-64, June 2004.
- [2] N. Abu-Ghazaleh, M.J. Lewis, and M. Govindaraju, "Performance of Dynamically Resizing Message Fields for Differential Serialization of SOAP Messages," in *proceedings of ISWS '04: The International Symposium on Web Services and Applications*, pp. 783-789, June 2004.
- [3] N. Abu-Ghazaleh, M. Govindaraju, and M.J. Lewis, "Optimizing Performance of Web Services with Chunk-Overlaying and Pipelined-Send," in *proceedings of The 2004 International Conference on Internet Computing (IC'04)* pp. 482-485, June 2004.
- [4] M. Govindaraju, A. Slominski, K. Chiu, P. Liu, R. van Engelen, M.J. Lewis, "Toward Characterizing the Performance of SOAP Toolkits". In *proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pp. 365-372, November 8th, 2004, Pittsburgh, USA.
- [5] K. Chiu, M. Govindaraju, and R. Bramley. "Investigating the Limits of SOAP Performance for Scientific Computing," In *Proceedings of HPDC-11*, pages 246-254, Edinburgh, Scotland, July 23-26, 2002.
- [6] W3C. Web Services Architecture. <http://www.w3.org/TR/ws-arch/#id2260892>.
- [7] C.G. Harrison, D.M. Chess, A. Kershenbaum, "Mobile Agents: Are They a Good Idea?", Research Report, IBM Research, 1995.
- [8] D. Lange, M. Oshima, G. Karjoth, and K. Kosaka, "Aglets: Programming Mobile Agents in Java," In *Proc. Worldwide Computing and its Applications (WWCA 97)*, *Lecture Notes in Computer Science*, Vol. 1274, 1997.
- [9] D. Kotz, R. Gray, and D. Rus, "Future Directions for Mobile-Agent Research," *IEEE Distributed Systems Online*, 3(8), Aug. 2002.
- [10] R. Gray, G. Cybenko, et al., "D'Agents: Applications and Performance of a Mobile-Agent System," *Software - Practice and Experience*, 32(6):543-573, May 2002.
- [11] J. W. Stamos and D. K. Gifford. "Remote Evaluation," *ACM Trans. Programming Languages and Systems*, 12(4):537-565, March 1990.
- [12] Z. Maamar, Quan Z. Sheng, and B. Benatallah. "Interleaving web services composition and execution using software agents and delegation." In *AAMAS2003 Workshop on Web Services and Agent-based Engineering*, 2003.
- [13] A. Padovitz, S. Krishnaswamy, S. W. Loke, "Toward Efficient and Smart Selection of Web Service." In *AAMAS2003 Workshop on Web Services and Agent-based Engineering*, 2003.
- [14] F. Ishikawa, N. Yoshioka, Y. Tahara, and S. Honiden, "Towards Synthesis of Web Services and Mobile Agents." In *AAMAS2004 Workshop on Web Services and Agent-based Engineering*, 2004.
- [15] Z. Maamar, Q.Z. Sheng, and B. Benatallah, "On Composite Web Services Provisioning in an Environment of Fixed and Mobile Computing Resources", *Information Technology and Management*, Kluwar Academic, 2004, pp 251-270. Vol 5, No 3-4
- [16] Microsoft, *ECMA and ISO/IEC C# and Common Language Infrastructure Standards*, <http://msdn.microsoft.com/net/ecma/>
- [17] "Tool Interface Standards. ELF: Executable and Linkable Format." <ftp://ftp.intel.com/pub/tis>, 1998.
- [18] Sun Microsystems, "The Java Language Specification," <http://java.sun.com>
- [19] Specification: Business Process Execution Language for Web Services Version 1.1, <http://www-128.ibm.com/developerworks/library/ws-bpel/>
- [20] S.C. Johnson, "Yacc: Yet Another Compiler-Compiler", Bell Laboratories, Murray Hill, New Jersey, July 1978. 24
- [21] T. Parr and R. Quong, "ANTLR: a predicated-LL(k) parser generator", *Software--Practice and Experience*, 25(7), 789--810 (1995).
- [22] Y. Zou, K. Kontogiannis, "Towards A Portable XML-based Source Code Representation", *International Conference on Software Engineering (ICSE) 2001 Workshops of XML Technologies and Software Engineering (XSE)*, Toronto, Ontario, Canada, May 2001 (4 pages).
- [23] M. Keidl, S. Seltzsa, A. Kemper, "Reliable Web Service Execution and Deployment in Dynamic Environments," *Proceedings. Of Lecture Notes in Computer Science* 2819 Springer 2003, ISBN 3-540-20052-5 TES 2003: 104-118.