From Web Service Composition to Megaprogramming

Cesare Pautasso, Gustavo Alonso

Department of Computer Science
Swiss Federal Institute of Technology (ETHZ)
ETH Zentrum, 8092 Zürich, Switzerland
{pautasso,alonso}@inf.ethz.ch

Abstract. With the emergence of Web service technologies, it has become possible to use high level megaprogramming models and visual tools to easily build distributed systems using Web services as reusable components. However, when attempting to apply the Web service composition paradigm in practical settings, some limitations become apparent. First of all, all kinds of existing "legacy" components must be wrapped as Web services, incurring in additional development, maintenance, and unnecessary runtime overheads. Second, current implementations of Web service protocols guarantee interoperability at high runtime costs, which justifies the composition of only coarse-grained Web services. To address these limitations and support the composition of also fine-grained services, in this paper we generalize the notion of service by introducing an open service meta-model. This offers freedom of choice between different types of services, which also include, but are not limited to, Web services. As a consequence, we argue that service composition – defined at the level of service interfaces – should be orthogonal from the mechanisms and the protocols which are used to access the actual service implementations.

1 Introduction

Megaprogramming [23] was originally introduced to describe the large scale composition of megamodules, capturing the functionality of services provided by large, independent organizations. Megaprogramming prescribed a clear separation of the description of the externally accessible data structures and operations of a megamodule from the mechanisms used to interact with it. It also emphasized the importance of *mediation* between incompatibile megamodule descriptions.

Some of the existing languages for Web service composition (e.g. [5,11]) do not yet completely fulfill the megaprogramming paradigm because the services to be composed are all assumed to be of a single type: Web services. Clearly,

Part of this work is supported by grants from the *Hasler Foundation* (DISC Project No. 1820) and the *Swiss Federal Office for Education and Science* (ADAPT, BBW Project No. 02.0254 / EU IST-2001-37126).

when facing software integration problems at an Internet-wide scale, Web services seem to be the most appropriate tool [8]. However, for many other kinds of service integration scenarios, it would be an unnecessary restriction to assume that all services that are to be composed must all be Web service compliant. In fact, there are many existing, well established service access protocols (e.g., RMI, CORBA, JMS, HTTP) that should not necessarily be considered as out of date, when compared to Web services [1]. Furthermore, the *mediation* between incompatibile services turns out to be a very important requirement for successful integration projects. Thus, unless such “mediation services” themselves are encapsulated behind a Web service interface, it is not possible to efficiently address this important issue with current Web service composition languages [6].

In this paper, we show how we applied megaprogramming concepts to generalize Web service composition in the context of the JOpera project [15]. Web services can be considered as one *kind* of service, which is very useful, e.g., as it offers syntactical interoperability with remote services in a platform independent way [21,22]. However, these benefits come at a price of a very high access overhead. This is justified for invoking coarse-grained services, for which the internal execution time dominates the overall invocation time. For other kinds of services, i.e., fine-grained services, which perform a small computation, or for local services, which are published within the same organization doing the service integration, it may be reasonable to employ other kinds of access mechanisms and protocols. This way, it is possible to choose the most appropriate service type in terms of the effort required to integrate it with others with the possibility of minimizing the corresponding invocation overhead. As it would be impossible to provide out-of-the-box support for all possible kinds of services, JOpera’s service meta-model and the corresponding architecture can be extended to describe and interact with an open set of heterogeneous service access mechanisms.

This paper is structured as follows. In Section 2 we discuss related work in the context of Web service composition. In Section 3 we introduce JOpera’s open service meta-model, followed in Section 4 by some examples on how to apply it to describe three (very) different kinds of services: Web services, Java snippets and legacy UNIX applications. In Section 5 we describe the relevant aspects of JOpera’s architecture implementing the service meta-model. To give an indication of the difference between the cost of invoking coarse-grained Web services and fine-grained Java snippets we have included an overhead comparison in Section 6. In Section 7 we draw some conclusions.

2 Related Work

The need for supporting a variety of service access protocols is also recognized in the Web services community. To this end, the WSDL interface description standard supports an open-ended set of bindings. Therefore, a Web service, whose interface must be described using WSDL, does not necessarily need to be invoked using the relatively slow SOAP protocol if the client understands other (non standard) protocols which may offer better performance.

Currently, however, alternative protocols are not yet widely supported and as long as they are not standardized, using them would defeat the main point of the Web service vision, where everything should be standardized in order to achieve widespread interoperability [2].

Along these lines, the Web Services Invocation Framework (WSIF [9]) should be mentioned, as it provides this kind of access *transparency*. It allows to dynamically build clients to Web services described in WSDL, independent of the actual access mechanisms (e.g., SOAP) involved. As we will discuss in this paper, our service meta-model goes beyond that since it is not limited to services described with WSDL. Instead, it can be also applied to other interface description languages.

Moreover, in order to bridge the gap between the existing component heterogeneity and the uniform Web services standards, wrappers and interface adapters are still required to make the "legacy" types of components and protocols fit with the new standards. This approach introduces unnecessary execution overhead and shifts development and maintenance costs from the infrastructure to the end user [14]. Thus, we believe it is less expensive to build *once* a generic adapter to integrate a certain type of components into JOpera, instead of having to setup a different Web service wrapper for each of the service of that particular type that have to be integrated within a composite service.

Recently, to address the limitations of coarse-grained Web service composition, IBM and BEA systems proposed to extend the BPEL4WS [11] language with support for including Java snippets [10]. Although the need for such an extension was well argued, it remains unclear why, as opposed to Java, a .NET compliant language should not be chosen instead. Thus, a service composition technology which was originally tied to platform neutral Web services, becomes tangled into portability issues [20].

This problem originates from the confusion between the description of the composition and the description of the components. In our approach, we have chosen to keep a clear separation between the two. Thus, our visual service composition language [17] doesn't have to be modified to support new kinds of services, such as Java snippets, as this extension only affects the service meta-model.

3 An open Service Meta-Model

Before describing in detail the properties of some of the service types currently supported by JOpera, we introduce JOpera's open service meta-model. This way, we both motivate its flexibility and extensibility and summarize the information required to model and to access each type of service.

As shown in Figure 1, the interface of a service is defined in terms of a set of user-defined input and output parameters. This is the only information which is used in JOpera to define how the services are composed when drawing the data flow graph linking the parameters of different service interfaces [17]. Thus, a service interface constitutes the minimal unit of composition. As a first

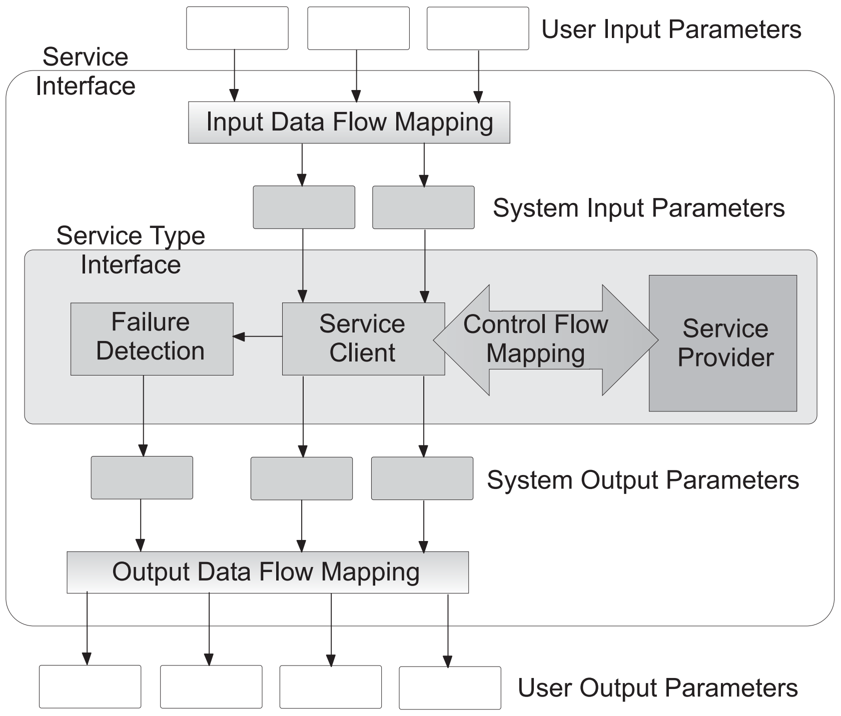

Fig. 1. Relationships between the various entities of the JOpera service meta-model

approximation the mechanisms involved in the invocation of a specific type of service are kept completely transparent when modeling how to compose different service interfaces.

However, in order to support the actual invocation of a service, it is necessary to model additional information describing how to invoke its functionality and how to structure the data exchanged with it. Such information is abstracted into a *service type*. More precisely, when adding a new service type to JOpera's model it is necessary to define its interface (in terms of system parameters); design how to interact with it in terms of control and data flow; and devise a failure detection strategy.

Furthermore, the same service interface can be associated with multiple service types. This way, it becomes possible to choose between alternative service access mechanisms. On the one hand the service invocation can be dynamically adapted to the actual system configuration, whereby the most optimal mechanism is chosen depending on the current environment. On the other hand, if the invocation fails using one mechanism, another path can be attempted to access an alternative service provider.

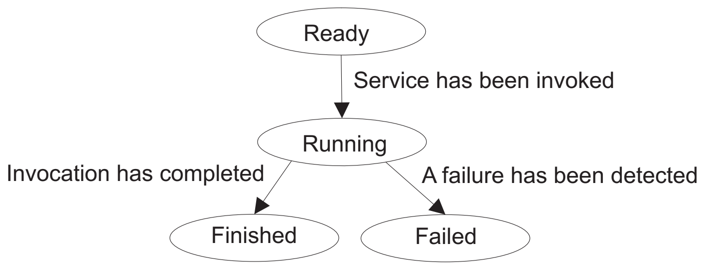

Fig. 2. Simple model of a service invocation

3.1 System Parameters

First of all, the interface of each service type is defined as a set of input ([i]) and output ([o]) parameters. These are called system parameters, to distinguish them from the user parameters, which are associated with the interface of the service. It is worth noting that user parameters depend on the specific application and therefore have nothing to do with the system parameters, which instead model the information required to access a particular type of service.

The input system parameters control the service invocation, as they identify the service and describe the information required to interact with the corresponding service provider. Their values are set at design time, when registering a new service with JOpera's component library.

The output system parameters model the raw results of the invocation as well as related metadata (e.g., status, performance profiling or debugging information). Their values are set after the invocation has completed and can be used to determine its outcome.

3.2 Control flow

The transfer of control during one service invocation may involve different interaction patterns between the client and the service provider.

In the simplest case, the service is invoked synchronously, i.e., the client blocks until the results of the invocation are available. This case captures typical procedure-like invocations, e.g., a call to a local method, a remote procedure call, an HTTP request/response round.

However, other protocols involve the asynchronous (or event-based) interaction between client and service provider, based on the exchange of a pair of messages representing the starting of the invocation and the notification that it has completed. Following this protocol, the client does not block after sending the request to the service provider, although the invocation only completes after the client is notified with a response. Depending on the available mechanisms, the client may periodically poll the service provider for a response, or a notification message is pushed back from the service provider.

More sophisticated interactions with a service provider may involve the ability to abort, suspend and resume an ongoing invocation [19]. Likewise, it may be possible to retrieve partial results even before the whole invocation has completed [18].

In order to ensure the transparency of these different interaction patterns, we introduce a simple model of a single transfer of control between client and service provider in Figure 2. Using this model, a control flow mapping can be easily designed for the aforementioned synchronous and asynchronous cases. If necessary, the *Running* state can be extended to support other forms of interaction.

3.3 Data flow

From the point of view of transferring control, the interaction with different service types is not so difficult to model, as this amounts to describing the invocation of the service and the corresponding notification that the service's invocation has completed.

In our experience, a more difficult challenge lies in modeling the data to be exchanged with the service and in how to map JOpera's parameter based representation of its interface to the service's internal one. For some service types this can be relatively simple, at least from a syntactical perspective, where standards (e.g., SOAP) define how to format the input data and how to interpret the output data. In other cases, e.g., when integrating legacy UNIX applications, the problem is much more difficult and there is no general solution, i.e., the adhoc development of wrappers may be required.

In order to provide the necessary flexibility to integrate several different service types, in JOpera we follow a two step approach to address the problem of mapping user-level data parameters to the actual structure of the data understood by the service type.

The mapping between user (application) parameters and system (service type) parameters is specified once, when a new service component is registered with JOpera. This mapping can be derived automatically, e.g., by reading the WSDL description of a Web service.

The data flow mappings depicted in Figure 1 can be formally represented as a composition of two mappings (m_i, m_o) which are applied to fit the input and output parameters of a certain service call C to the given interface S. More precisely, the interface of a service contains a set of user-defined input ($[I]$) and output ($[O]$) parameters:

$$[O] = S([I])$$

Furthermore, a set of predefined service types C_t are available. These define the interface representation of the corresponding access mechanisms and invocation protocols in terms of input ($[i]$) and output ($[o]$) system parameters:

$$[o] = C_t([i])$$

In order to bind a service interface to an implementation of a given service type, it is necessary to provide the corresponding input and output mappings:

$$[i] = m_i([I])$$

$$[O] = m_o([o])$$

At runtime, these mappings are composed with the invocation of service of a given type as follows:

$$[O] = m_o(C_t(m_i(I))$$

Following such mapping, before a service can be invoked at runtime, the user input parameters are translated to its system input parameters. The main mechanism to model and perform this mapping (m_i) consists of using parameter placeholders, which identify one user input parameter and are replaced with its content when the mapping is evaluated. These placeholders follow the simple convention of including the name of a parameter between `%` characters [13].

The service is then invoked and the results are placed in the system output parameters corresponding to its type. The reverse mapping m_o from the system output parameters to the user-defined output parameters is applied. As opposed to the input mapping, where a relatively large number of user parameters are assigned to a small number of system parameters, in this case it is more complex to take the content of a few parameters, e.g., the output of a program or a Web page, and model how to extract the application dependent information. For data having a relatively well defined syntax, e.g., XML, it is possible to follow the convention of encoding parameter names as tags and insert their values between those tags [21].

In general, *ad-hoc wrappers* can be plugged into JOpera with the purpose of scraping the values of the output parameters from the arbitrarily formatted data produced by the service. Conversely, it is also possible to avoid breaking up the results of the invocation into output parameters and treat the result (e.g., in form of XML documents or other encodings) as a whole.

3.4 Failure detection

Not only do service invocations finish; sometimes they fail. Depending on the type of service, failure detection may be based on different assumptions. For each type of service, it is important to devise a well-defined failure detection strategy, which determines the outcome of a service invocation. In case of failed invocations, a description of the problem involved can be stored in the corresponding system output parameters.

Furthermore, depending on the type of failure, different low-level error handling policies may be implemented. For example, the service invocation may be retried, if this option is supported by the underlying protocol. Thus, only unrecoverable failures occurring during the interaction with a particular service provider remain to be handled at the level of the service composition. In this case, exception handling constructs can be used to specify whether alternative (or compensating) services should be be invoked instead.

Service Type		**Input and Output Data**		**Failure**
WWW services				
Web Service	(`SOAP`)	SOAP	SOAP	SOAP Fault
Web Server	(`HTTP`)	CGI/URL	HTML	HTTP Error
Local services				
UNIX Application	(`UNIX`)	CmdLine, Stdin	Stdout	ExitCode, StdError
Java services				
Java Program	(`JVM`)	CmdLine, Stdin	Stdout	ExitCode, StdError
Java Snippet	(`JAVA`)	Local Variables		Exception
Java Remote Method	(`RMI`)	Method Parameters		Exception
Database services				
Database Query	(`SQL`)	Parameters	XML	JDBC Error
XML services				
X-Path Query	(`XPATH`)	XML	XML	X-Path Processor Error
Style Sheet Transformation	(`XSL`)	Parameters	XML	XSLT Processor Error
System services				
JOpera Echo	(`ECHO`)	XML	XML	XML Parser Error
JOpera Process	(`OPERA`)	Implicit Parameters and Failures		
Cluster/Grid computing services				
BioOpera [4]	(`PEC`)	CmdLine	Stdout	ExitCode, StdError
Grid services [7]	(`GLOBUS`)	SOAP	SOAP	SOAP Fault
Business process modeling services				
Workflow task	(`WF`)	Text	Text	User Error

Table 1. Summary of the service types currently supported by JOpera

4 Examples

In this section we show how to apply our service meta-model to abstract the common features of different kinds of services. These represent three extreme cases: standard compliant Web services, fine-grained Java scripts and legacy UNIX applications.

Additionally, the current version of JOpera includes supports for many other kinds of services, modeling a Java remote method invocation (RMI), a job submitted to a batch scheduling system of a cluster of computers, an SQL query to be sent to a database, the asynchronous exchange of messages through a queuing system, a human activity, and an XSL style sheet transformation to be applied to some XML data packet [16]. In Table 1 we summarize the main properties of some of the service types to which we have applied JOpera's service meta-model.

4.1 Web Services

This first type of services models the latest form of standard compliant Web services, whose interface and location are described in a WSDL document [22] and which are remotely accessible through the SOAP protocol [21]. Web services offer the benefit of standard-based interoperability between heterogeneous programming languages and platforms. With this technology, the effort of building systems composed out of services distributed across the Internet is greatly reduced, at the price of a relative high runtime overhead due to the nature of the protocols involved. Thanks to these standards, it is possible to automatically import the service's WSDL description into JOpera's component library and use it to generate the corresponding service declarations automatically.

System Parameters The invocation Web service is described by the following system input parameters: `WSDL`, with the URL used to locate the description of the service; `service`, `operation`, `port`, with the names of the WSDL elements used to identify the actual service, operation and port to be invoked; `soapin`, which contains the complete envelope of the SOAP request message to be sent when invoking the service. This includes both the header and the body of the SOAP request message. The response (or fault) message returned by a Web service is stored in the `soapout` system output parameter.

Data flow The values of the user-provided input parameters are inserted in the SOAP request message using the previously described placeholder mechanism. In most cases, each input parameter corresponds to a SOAP message block. If necessary, JOpera escapes the content of the parameters so that it conforms to the required SOAP/XML encoding. The output parameters are filled by parsing the SOAP response message.

Failures The invocation of a Web service may fail for several reasons: its WSDL description may be invalid; no response message from the service has been received after a certain timeout has expired; the service has responded with a soap fault message.

4.2 Java Snippets

This service type models the most efficient way of invoking Java code. By design, such code (or snippet) is embedded by the compiler into the code generated for a process. Thus, it can be invoked with minimal overhead. It can be very beneficial to use this kind of service to perform small computations [10]. Java snippets can be applied to perform data conversions, transforming the data in transit between incompatible services. Also, it gives a convenient syntax for the evaluation of complex conditional expressions. If the same computation would have to be invoked using a different mechanism (e.g., Web services), the overhead of the protocols involved would make it impractical to do so.

System Parameters For Java snippets, there is only one system input parameter (`script`) which contains the Java code itself. If an error occurs, the `exception` system output parameter contains the message of the Java exception.

Data flow There is a one to one correspondence between user defined parameters and the Java variables that can be implicitly used in the script. JOpera's compiler automatically declares Java variables for each input and output parameters. After the snippet has completed, the values assigned to the Java variables are copied into the corresponding output parameters.

Failures JOpera detects a failure if a Java exception is raised and it is not caught during the execution of the script.

4.3 UNIX Applications

Another type of services, quite different from remote Web services, are commands to be executed in a shell of the local operating system. A shell command is typically used to provide a generic mechanism of invoking entire "legacy" applications. As long as these applications do not provide an explicit API, the command line may be the only viable mechanism to allow JOpera to interact with such applications and control their execution. In other words, this type of service is used to access the services provided by essentially any executable program, which is started by typing a command line at the prompt of the operating system shell.

System Parameters As it is reflected by its system parameters (`command`, `stdin`, `stdout`, `stderr`), JOpera employs both the command line itself and pipe-based interprocess communication mechanisms in order to exchange data with the external program. Furthermore, the `retval` system output parameter contains the program exit code.

Data The values of the user input parameters are transferred to the external program both using its `command` line and can also be copied onto its `stdin` system input parameter. If necessary, the `stdout` parameter can be parsed by a user-provided plugin to extract relevant information to be assigned to the user defined parameter.

Failures JOpera interprets the value of the `retval` system parameter, which contains the exit code of the process as it is returned by the operating system,

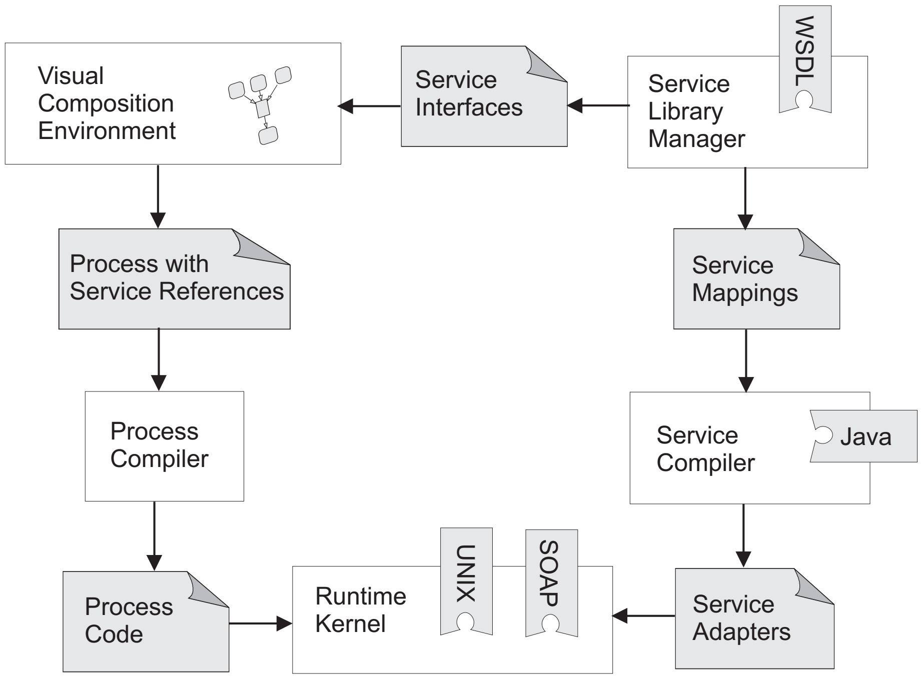

Fig. 3. JOpera plugin based architecture

to distinguish between a successful execution (0) and a failed execution (non-0). In both cases, it also stores the program's standard error into the `stderr` parameter so that the user can gather useful debugging information.

5 Architecture

In order to support an open and heterogeneous set of service invocation mechanisms, JOpera's architecture uses plugins to extend the system's behavior at three different stages: service definition, service compilation and service invocation.

As shown in Figure 3, the Service Library Manager uses service import plugins to automatically import services described using other meta-models (e.g., WSDL). Using the Visual Composition Environment, the developer may browse through the service library and select the service interfaces to be composed into processes [17]. During process compilation, all of the data flow mappings, which are part of the services referenced by a process are compiled into service adapters[1]. By default, the service compiler produces an efficient executable

[1] Although it is always possible to merge the code of the process with the service adapter code at compile-time, this would fix the binding between service interface

Service Type	Description
`JAVA`	Java Snippet
`UNIX`	UNIX Application
`SOAP/A11`	Local Web Service using Axis 1.1 [3].
`SOAP/A12`	Local Web Service using Axis 1.2α.
`SOAP/WS`	Remote Web Service using Axis 1.1.

Table 2. Service Invocation Mechanisms to be compared

representation of the data flow mappings of a service. However, the service compiler can be extended with plug-ins corresponding to a specific type of service. For example, in case of Java snippets, the Java code entered as part of the aforementioned `script` parameter is injected into the resulting service adapter code, surrounded by the variable declarations corresponding to the user-defined parameters.

At run-time, the service invocation proceeds as depicted in Figure 1. The runtime kernel uses the compiled service adapters to perform the input and output data flow mappings, while the service is invoked through a kernel plugin. Such plugin uses the mechanisms and protocols specific to a certain service type (e.g., UNIX, SOAP) to interact with the service provider and perform the service invocation. Considering the service meta-model presented in Section 3, these plug-ins define the control flow mapping and the failure detection strategy for a given type of services and exchange information with the service adapters through system parameters. The kernel plugins are loaded on-demand, so that the system can be dynamically extended to deal with new types of services.

When adding support for a new type of service, a kernel plug-in is required. A compiler plugin is only necessary if the service adapter should perform some special processing before or after the invocation. A service import plugin can be added if it is possible to automatically generate JOpera service definitions from other interface description languages.

6 Overhead

Performance is one of the arguments behind the idea of providing support for invoking services of different service types. In order to give an indication of the overhead involved, we compare the time required by JOpera to invoke a remote Web service across the Internet with the time JOpera takes to perform a local Java method call, and – quantitatively – determine the cost (or the benefit) of preferring services of a certain type over another.

and invocation adapter. Thus, in order to support late binding, the code of the process only contains references to services, which are resolved at the latest possible time.

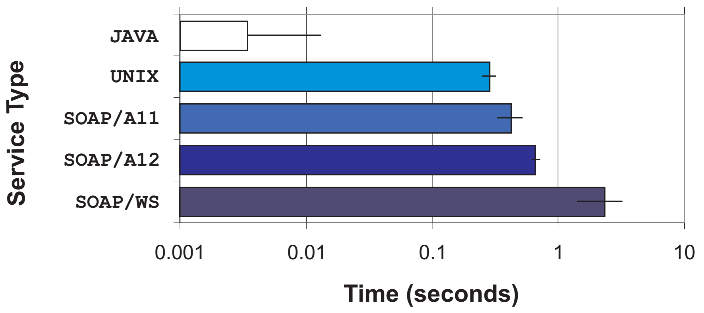

Fig. 4. Service Invocation Overhead for different service types

As listed in Table 2, in this performance comparison we use services of various types and several implementations of the corresponding kernel plugins.

More precisely, in this experiment we compare different access mechanism to the same "Temperature Conversion Service". We chose this service due to its trivial implementation, so that the execution cost is negligible when compared to the overhead of invoking it. Another reason to choose this service is that we found a remote implementation on the Internet at [12]. With it, it becomes possible to present an interesting comparison between the invocation overhead of local and remote Web services.

As shown in Figure 4, the most important result of this simple experiment is that the average service invocation overhead varies about three orders of magnitude (from about 1 millisecond to 2.31 seconds) depending on the service type.

The invocation of the Java snippet (`JAVA`) service offers an invocation overhead of significantly less than $1/100^{th}$ of a second, as the implementation of the service is located within the same Java virtual machine where the JOpera kernel is running.

Invoking the `UNIX` application requires to spawn a child process through the local operating system, and this requires more time: about 0.28 seconds.

The average Web service invocation time is 0.42 seconds in case of a Web service deployed on the local area network, called using Axis version 1.1 (`SOAP/A11`). This time grows to 0.66 seconds using the latest version of Axis 1.2α (`SOAP/A12`). In case of the invocation of remote Web service with Axis 1.1 (`SOAP/WS`), the delay and jitter of the wide area network need to be discounted. This effect can be recognized both in the higher (2.31 seconds) average response time and in the very high standard deviation (0.9 seconds).

As expected, Web services are the most expensive service type in terms of the overhead involved. Given the current state of flux of the relevant standards and available implementations, the performance of the service invocation may be significantly affected by the choice of which libraries are used. Additionally, the location of the Web service also affects the overhead, as the cost of invoking the remote Web service shows.

Since this additional cost is due to the distributed nature of the service interaction, it should not be blamed on the Web services protocols, which – instead – are one of the few technologies currently enabling such type of distributed interaction. Nevertheless, such overhead should be paid only when necessary, i.e., to invoke remote services, while more efficient mechanisms should (and can) be used to access local services.

7 Conclusion

The main contribution of this paper lies in the idea that service composition should be orthogonal with respect to the types of components involved. By introducing a clear separation between the service *composition language* and the *service meta-model*, we are able to isolate the description of how to compose the services from how to invoke them. This approach is similar to megaprogramming [23], as it gives several conceptual and practical advantages. First of all, it is not necessary to extend the composition language if a new kind of service access mechanism has to be included, as this affects only the component model. Likewise, if it is possible to redefine the access mechanism (e.g., synchronous vs. asynchronous) to be employed without modifying the corresponding service interface, such modifications are completely transparent as far as the description of the composition is concerned. Such flexibility also leads to the possibility of doing optimizations since it becomes possible to choose the most efficient mechanisms and protocols to access both fine-grained and coarse-grained services. We are currently investigating several policies to autonomously select the optimal mechanism. This is much more difficult to accomplish if the services to be composed are restricted to only one type.

Finally, we believe that the possibility of choosing (wisely) between the use of Web Services or other kinds of services can be of great value, as the most appropriate type of service in terms of performance, security, reliability and convenience of use can be chosen.

References

1. G. Alonso. Myths around Web services. *Bulletin of the IEEE Technical Committee on Data Engineering*, 25(4):3–9, December 2002.
2. G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web services: Concepts, Architectures and Applications*. Springer, November 2003.
3. Apache Software Foundation. *AXIS version 1.1*. `http://xml.apache.org/axis`.
4. W. Bausch, C. Pautasso, R. Schaeppi, and G. Alonso. BioOpera: Cluster-aware computing. In *Proceedings of the 2002 IEEE International Conference on Cluster Computing (CLUSTER 2002)*, pages 99–106, Chicago, IL, USA, 2002.
5. BPMI. *BPML: Business Process Modeling Language 1.0.* Business Process Management Initiative, Match 2001. `http://www.bpmi.org`.
6. C. Bussler. Semantic Web services: Reflections on Web Service Mediation and Composition. In *Proceedings of the Fourth International Conference on Web Information Systems Engineering (WISE 2003)*, pages 253–260, Roma, Italy, December 2003.

7. I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Technical report, Service Infrastructure Workgroup, Global Grid Forum, 2002. http://www.globus.org/research/papers/ogsa.pdf.
8. K. Gottschalk, S. Graham, H. Kreger, and J. Snell. Introduction to Web services architecture. *IBM Systems Journal*, 41(2):170–177, 2002.
9. IBM and Apache Foundation. *Web Service Invocation Framework (WSIF)*, 2003. `http://ws.apache.org/wsif/`.
10. IBM and BEA Systems. *BPELJ: BPEL for Java technology*, March 2004. `http://www-106.ibm.com/developerworks/webservices/library/ws-bpelj/`.
11. IBM, Microsoft, and BEA Systems. *Business Process Execution Language for Web services (BPEL4WS) 1.0*, August 2002. `http://www.ibm.com/developerworks/library/ws-bpel`.
12. C. Jensen. *Temperature Conversion Service*. `http://developerdays.com/cgi-bin/tempconverter.exe/wsdl/ITempConverter`.
13. F. Leymann and D. Roller. Business Process Management With FlowMark. In *Proceedings of the 39th IEEE Computer Society International Conference (CompCon '94)*, pages 230–234, February 1994.
14. J. Oberleitner and S. Dustdar. Constructing Web services out of Generic Component Compositions. In *Proceedings of the International Conference on Web services (ICWS-Europe 2003)*, pages 37–48, Erfurt, Germany, 2003.
15. C. Pautasso. JOpera: Process Support for Web services. `http://www.iks.ethz.ch/jopera/download`.
16. C. Pautasso. *A Flexible System for Visual Service Composition*. PhD thesis, Diss. ETH Nr. 15608, July 2004.
17. C. Pautasso and G. Alonso. Visual Composition of Web Services. In *Proceedings of the 2003 IEEE International Symposium on Human-Centric Computing Languages and Environments (HCC 2003)*, pages 92–99, Auckland, New Zealand, 2003.
18. N. Sample, D. Beringer, and G. Wiederhold. A Comprehensive Model for Arbitrary Result Extraction. In *Proceedings of the 2002 ACM symposium on Applied computing (SAC 2002)*, pages 314–321, Madrid, Spain, 2002.
19. H. Schuster, S. Jablonski, P. Heinl, and C. Bussler. A General Framework for the Execution of Heterogeneous Programs in Workflow Management Systems. In *Proceedings of the 1st IFCIS International Conference on Cooperative Information Systems (CoopIS'96)*, pages 104–113, Los Alamitos, CA, 1996. IEEE Computer Society Press.
20. H. Smith. *Enough is enough in the field of BPM: We don't need BPELJ: BPML semantics are just fine*, April 2004. `http://www.bpm3.com/bpelj/BPELJ-Enough-Is-Enough.pdf`.
21. W3C. *Simple Object Access Protocol (SOAP) 1.1*, 2000. `http://www.w3.org/TR/SOAP`.
22. W3C. *Web services Definition Language (WSDL) 1.1*, 2001. `http://www.w3.org/TR/wsdl`.
23. G. Wiederhold, P. Wegner, and S. Ceri. Towards Megaprogramming: A Paradigm for Component-Based Programming. *Communications of the ACM*, 35(11):89–99, 1992.