Client-Side Implementation of Dynamic Asynchronous Invocations for Web Services

Giancarlo Tretola[1], EugenioZimeo[2]

[1]University of Sannio
Dept. of Engineering
Benevento,82100 Italy
tretola@unisannio.it

[2]University of Sannio
Research Centre on Software Technology
Benevento,82100 Italy
zimeo@unisannio.it

Abstract

Web Services are becoming more and more fundamental building blocks of Web-based distributed applications and a core technology for Grid systems. Due to their flexibility, Web Services easily combine, in a common and coherent framework, ubiquitous computing with heterogeneous applications composed of different kinds of resources and, typically distributed in many organizations. We expect that this technology will follow the same evolution paths that have characterized other technologies so far, with some specificity due to the openness and size of the application context. In this connection, optimizations tied to invocations and workflows are assuming a primary role in Web Services research. The synchronous request/reply nature of the most diffused underling protocol (HTTP) introduces several restrictions in many application scenarios. On the other hand, asynchronous interactions are allowed by using message oriented middleware platforms, like JMS, which are typically harder to handle than object- and process-oriented middleware. In this paper, we propose a first implementation of a module that allows for dynamic Web Services invocations, which, on the basis of meta-data added to WSDL, is able to select the most appropriate invocation technique for calling a Web Services operation.

1. Introduction

Nowadays, Web Services are achieving a growing maturity in the field of distributed applications development. There are many domains that are involved with Web Services composition. Workflow Management [18] products, for example, are able to interact with functionality offered as Web Services. Grid Computing is showing an increasing interest in Web Services for enhancing and simplifying access to heterogeneous and dispersed computing resources [9]. Business processes performed using Web Services composition and orchestration is a soundly possibility in several field of distributed application development [7] and they are increasingly used in B2B and B2C applications.

In recent years, in fact, the Web has become the primary environment for operating B2B and B2C heterogeneous applications using Service Oriented Computing (SOC) [13]. The SOC paradigm is increasingly appreciated and we are going towards a future in which organizations interact by means of interoperable Web Services. The vision of a software market based on providing and requiring components offered as Web Services is more and more real. An interesting and promising context in which Web Services are assuming a significant role is automotive, with particular regard to e-procurement, collaborative engineering, and supply chain management. We have matured a specific experience on these topics in the context of the LOCOSP Project [22], which aims to define a distributed platform for the logistics of knowledge in collaborative engineering. In the LOCOSP Platform Web Services wrap engineering activities in the product development process and use Grid technologies (WSRF [14] and related standards) to deliver and retrieve data (CAD artefacts) to and from external suppliers.

Due to this growing maturity it is important starting to consider possible weakness to fix in the model or, at least, take into account efficiency improvements. Several research activities involve Web Services at different abstraction levels. At high level, an open discussion exits on the approach to use for Web Services modelling and development. An important debate is about considering Web Services as distributed objects or not [2][3]. Some authors affirm that Web Services are different from distributed objects and must be treated with a dedicated approach. Others advocate that Web Services are only particular cases of distributed Objects and could benefit of similar treatment. In our opinion it is not important how to

1-4244-0910-1/07/$20.00 © 2007 IEEE.

consider Web Services from an abstract level but it is really important to have tools and methodology that are already used and tested in other environments, like the object oriented one. These approaches could allow an improvement in the way services are consumed. In fact a lot of attention is also appointed to interaction and implementation issues. Several works are involved with bringing asynchronous service invocation in the Web Services world [4][5][7]. Other activities have proposed the introduction of Object-Oriented techniques in Web Services modelling [1].

In this work, we will report the first results of our ongoing research that aims at extending Web Services from a semantic point of view in order to support more advanced features. Our vision is to consider an Object Oriented approach to Web Services consuming, introducing more abstract interface to interact with them as they were instance of remote objects. This approach, in our opinion, could ease the client side application building and management. Moreover, it is our conviction that using the Object Oriented approach in distributed computing is an efficient and more sure path to introduce concurrent programming concepts [10]. The introduction of such principles allows for further improvements in other fields like workflow management, for example. In our works, we have shown how it is possible to improve performance using fine-grained concurrency and asynchronous invocation in workflow enactment [11] and in Grid scheduling [12].

The first aspect we consider is related to asynchronous invocations. As stated in [5], an application that invokes a service asynchronously could continue its execution without needing to wait for a result and could perform other operations, stopping only when the result is needed to continue computing. Moreover, asynchronous interactions avoid the necessity for managing a session in the communication with service but require persistence of the call state information. This mechanism is useful for many applications in a distributed and heterogeneous execution environment: (1) to overlap computation with communication in order to tolerate the high latencies that characterize wide-area distributed systems; (2) to anticipate the scheduling and the execution of activities that do not completely depend on the result of an invocation; (3) to easily support interactions for long-running transactions; (4) to homogenously consider interactions with humans and machines in order to handle them in the same way at control level.

The remaining part of the paper is organized as follows. Section 2 describes related work on asynchronous Web Services invocation. Section 3 analyzes the interaction patterns that could be used for asynchronous invocation of Web Services. Section 4 describes the architecture of the dynamic invocation component we have designed, implemented and tested. Section 5 concludes the paper and describes the current activities we are involved with to improve the component and to add other functionalities.

2. Related Work

In [4], the authors describe the results of their studies about Web Services asynchronous invocation by presenting a description of several approaches that could be used to implement correlation between requests and responses and proposing an exhaustive semantic description model to achieve the result. Moreover, a classification and a description of the most used asynchronous interaction patterns is presented. The work discusses several important aspects of asynchronous interactions. The implementation effort has been aimed to realize the described patterns in an experimental environment, like Acer Business Portal, in PattiChiari Web site [27], and MetalC project [1]. Differently from them, we intend to realize an autonomous and dynamic invoker that eases interaction with Web Services where asynchronous invocation is one possible interaction scheme supported.

In [6], the author presents an analysis of enterprise applications, in a SOA environment, which could benefit of the asynchronous invocation given the fact that business processes involves human participants and human interactions. Both of them benefit of asynchronous interactions. The approach followed uses WS-Addressing [15] coupled with a call-back-based approach. The method proposed calls for a client side Web Service that implements a call-back interface, i. e. an interface used by the server to notify operation completion an to deliver the result. Although, the approach is interesting, it requires service modification and is not transparent.

In [7], the authors tackle the problems that arise when asynchronous invocations are performed in complex applications composed with Web Services. The authors are concerned with problems related to activities failure in long running processes and management of running process reconfiguration. They have defined a Document Flow Model (DFM), a message-based workflow modelling of asynchronous interactions among Web Services in workflow processes. The authors are now working on the realization of simulation tools for asynchronous invocation to test their ideas.

Another important aspect in Web Services modelling is about the relationship with distributed objects paradigm and methodologies [2][3]. The discussion is about considering or not Web Services as distributed objects and the comparison between the relative performances.

In [1] the authors present a performance comparison using document oriented applications. Another point that

authors underline is that the Web Service client code is uneasy to use. The client application is involved to manipulate request and response, rather than directly perform operations on server objects as in the RMI implementation, for example. The authors propose to implement a document centric RMI implementation, which benefits of the advantages of both approaches. The work is very interesting and proposes a new point of view of the problems arising in distributed computing realized with services composition. However, we stick with Web Service approach and believe that interoperability and security issues are better managed in respect to distributed objects. There is, however, the need to intervene for performance improvement.

Summarizing, the idea of asynchronous invocation is present in the literature and it is recognized that is a valuable mechanism. As we anticipated, our interest is to introduce concurrent programming [10] in Web Services. Moreover we are interested in developing a concrete component in Java language, to couple with other projects in which we are involved.

3. Asynchronous invocation patterns

The aynchronous invocation of Web Services (also known as deferred synchronous invocation when a result is returned) could be described as a call in which the consumer must not wait for the result from the provider counterpart. The caller, or consumer, may continue the execution and can receive the result when it is ready. So the result is requested from the server if it is really needed and just in the moment it is needed. The first problem that arises is related to the connection of one response to the right request. A soundly solution is the use of a transaction ID associated to the request that is attached to the response to obtain the right coupling. Another important point to consider is the possibility for the caller to query the called, or provider, about result availability.

There are four patterns that describe asynchronous interactions in the distributed objects field adopting an RPC style.

Fire and Forget consists in a pure asynchronous request message sent from the client to the server, without any result restitution. In this case, the client does not wait for the service completion of the functionality and continue its execution.

The *Sync with Server* describes an interaction similar to the preceding one but with the difference that the client must wait until the server confirms the reception of the request. When the acknowledgement is received, the client and server could continue the execution concurrently.

The *Polling Object* pattern is used when the invocation is asynchronous but the client will need the result to complete its computation. Then, the client does not need the result immediately and so can continue to run without stopping. In this case, the client receives an object on which it is possible to perform a polling, i.e. a query about result availability. If the result is ready, it is provided to the client, otherwise it is placed in waiting state until the result will be ready.

The *Result Callback* asks for asynchronous invocation of the server functionality and the result is returned by the server with the invocation of an appropriate functionality of the client object: the call-back handler. Such handler must be provided by the client, implementing a defined interface, and passed to the server when the asynchronous invocation is done. When the server completes the execution, it uses the call-back handler to asynchronously send the result to the client.

In the context of Web Services, the transport layer that supports the interaction plays an important role. Some protocols already support asynchronous messaging (HTTPR, JMS, MS Messaging) and are well fitted for asynchronous invocations. Other protocols are inherently synchronous and so require sessions and correlations (HTTP, HTTPS, RMI, SMTP).

The selection of the protocol is often bound to the operative environment and network infrastructure used for communication. Another open question is about the RPC implementation and the message based interaction. The former is preferred by developers because they are more used to act in terms of method or procedure invocations, and problems related to the remote components are hidden by the middleware.

The message based approach is less familiar to developers but has several advantages. It does not use a client-server description, and so the participants in message exchange could be seen as peers.

The message exchange could be time-independent while RPC requires an active connection between the participants. RPC is intrinsically point-to-point, while message could be replicated and delivered to many receivers.

In our approach, we chose to decouple the interaction of the consumer with the provider through an intermediary. This component allows for the client to use RPC style call of Web Services functionality. Then, the component is in charge to dynamically perform the invocation that could be synchronous or asynchronous.

4. Dynamic invocation module architecture

The WSDynamicInvoker module could be used as independent component to support the development of a client for Web Services consuming or it could be used as a component of a system performing dynamic invocation using different interaction patterns. The aim is to simplify client-side invocation of any Web Service. The invocation

mechanism could be synchronous or asynchronous, depending on the client necessity and the service implementation. The interaction is dynamic, in respect to Web Services binding, because the client must only provide the chosen service, the method to invoke and the parameters, in similar manner to a method invocation on an object instance. The module is able to autonomously contact the service, retrieve the WSDL, prepare and send the request message. Furthermore the module is able to (1) receive the response message, (2) perform its analysis and (3) provide to the client the data contained.

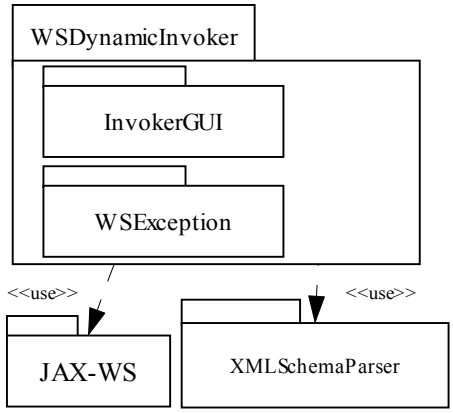

Figure 1. Package Diagram

The principal objective is to allow the invocation of any type of Web Service independently from the operating system. The server-side environment could be Tomcat/Axis, Microsoft IIS, J2EE, etc. In regards to invocation style, it is possible to use RPC or Document style. Furthermore using SOAP, it is possible to use encoded or literal style for the invocation. Combining the styles, four basic combinations are possible, plus one used only in Microsoft Windows environments: RPC/Encoded, RPC/Literal, Document/Encoded (rarely used and not compliant to WS-I), Document Literal, Document Literal Wrapped (introduced to improve usability in Windows environments).

The functionality offered to clients are: synchronous invocation, asynchronous invocation with polling object, asynchronous invocation with call-back, and finally a so called auto invocation that let the module to choose the best technique to use. In the last case, the dynamic invoker takes the decision basing on the complexity of the functionality involved in the invocation. The idea is that provider could describe the computational complexity of the service, based on its implementation. By knowing such information, the requestor could choose the invocation pattern that fits in well. The complexity of a service could be described with an additional optional tag defined as a WSDL extension: the *complexity* tag. Such tag could be introduced in the *operation* part of the WSDL description giving metadata about its computational complexity. This tag could assume a value belonging to an enumeration, which should provide the possible complexity description of an operation. Each value of the enumeration could group a complexity class, for example linear, logarithmic, exponential, factorial and so on. One labelled an operation with the complexity tag and if the problem size is known, the invoker could be able to estimate the duration of the operation. Basing on this information, the invoker could choose the more apt way to invoke the service. Currently, the policy that guides the selection assigns synchronous invocation to operations with low complexity and asynchronous with call-back handler to operations with high complexity. It is our intention to provide an extensible hot spot to define specialized policy for invocation style selection.

The module for dynamic invocations is organized in three packages: the *WsDynamicInvoker*, the JAX-WS 2.0 and the XML Schema Parser. The package diagram, depicted in figure 1, shows that WSDynamicInvoker contains also two sub-packages for Exception handling and for a graphical testing application.

The package WSDynamicInvoker is in charge to interact with the client application, to manage the result of asynchronous invocations and to serialize and de-serialize the request and response messages.

The package's classes Invoker and the Caller are the classes responsible of invoking Web Services. They use an instance of the WSDLAnalyzer class to perform the analysis of the WSDL of the service to invoke. The related class diagram is shown in figure 2.

The creation of SOAP request message is performed using an instance of SOAPMessageBuilder class. The invocation to the Web service is performed by the Caller object that is in charge to send the request message. If the invocation is asynchronous, a Future Object [10][16][21] is returned, which in this context acts like a placeholder for the result and also as a pollable object for testing result availability.

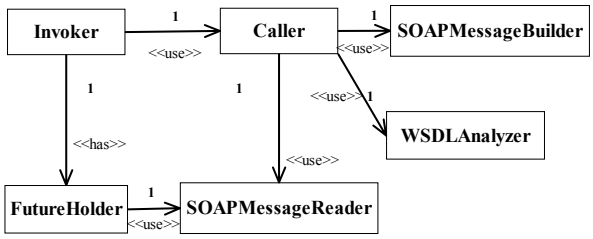

Figure 2. Class Diagram of WS Dynamic Invoker

The WSDLAnalyzer has the role to interact with the WSDL of the Web Services and retrieve from it the information on the service to invoke and the operation to require. Furthermore, it has to understand the SOAP types

of the parameters to use in the interaction. The SOAPMessageBuilder uses this info to create the request message to be sent and for serializing the complex types. The SOAPMessageReader receives the response from the Web Service and extract the operation result, performing the required deserialization operations. The FutureHolder class, inspired to the homonymous object implemented in ProActive middleware [20] and available as interface to be implemented in Java [21], models the result of the invocation either synchronous or asynchronous and has the following methods:

- getValue(): provides the result of the operation as an Object. It is a blocking operation in the sense that if the result is still not available, the accessing thread is suspended (wait by necessity [20],[21])
- isDone(): allows to test if the result is available and is a non blocking operation.

The package XMLSchemaParser is used to perform analysis of WSDL, particularly complex types that could be used as operation parameters. The class SchemaParser has the responsibility to analyze the types in WSDL and, if necessary, to retrieve the external resource imported.

The last package is the library JAX-WS 2.0 [16] that is used to perform the invocation of services both synchronously and asynchronously. In WSDL four interaction patterns are possible: *one-way*, *notification*, *request-response* and *solicit-response*.

The first and the second ones could be imagined as procedure without a return value; the difference is that the one-way is initiated by the client, while the notification is performed by the server. The other two interactions are modelled on a couple of messages: a request and an answer message. Also in this case, the only difference is that the request-response interaction is initiated by the client while the solicit-response is initiated by the server. Therefore, two general interactions can be considered independently from the caller: one-way and request-response. The first category of interaction is obviously asynchronous, but could be used only if the required operation does not return a result to the caller; it is performed in JAX-WS with the method *invokeOneWay*. The second category could be performed in both ways, synchronously and asynchronously using JAX-WS.

The synchronous invocation is performed with the method invoke() and uses the request-response interaction (or solicit-response); in this case the invoking threads is blocked. The asynchronous invocation (or solicit-response) can be done with the method invokeAsync(), which has two overloaded versions. The first returns a pollable object, i.e. that could be polled asking for result, which extends the Future interface of Java. The second uses a handler, implementing the interface *asyncHandler*, which is passed to the JAX-WS and works as a call-back for receiving the notification when the result is computed. To better understand the invoker architecture, a look to the dynamic behaviour at run-time is useful.

A primary difference between synchronous and asynchronous interactions exists. Only one thread is responsible, in the former case, of the invocation process, serialization and deserialization of the parameters and the result of every message exchanged with the invoked Web Service. In the asynchronous invocation there are three running threads. The first one interacts with the client application, receives the methods name and the arguments, serializes the parameters, provides the Future object to the client. The second thread is responsible of the real invocation activity interacting with the service. The last thread is responsible of future object management and updates its value, de-serializing the response message.

The interaction with the client of the module is performed creating an object of the Invoker class, that receives the parameters necessary to perform the invocation: the end point of the service, the operation to invoke, a vector of objects that contains the parameters and an integer value used to choose the type of invocation to perform. The possible values of the last parameter allows for choosing between synchronous invocation, asynchronous invocation with polling, asynchronous invocation with call-back and automatic and is specifiable with symbolic constants. The automatic mode allows the module to choose autonomously the nature of the invocation, basing on services description meta-data contained in the *complexity* tag, if it is available, otherwise it executes the invocation with a call-back handler.

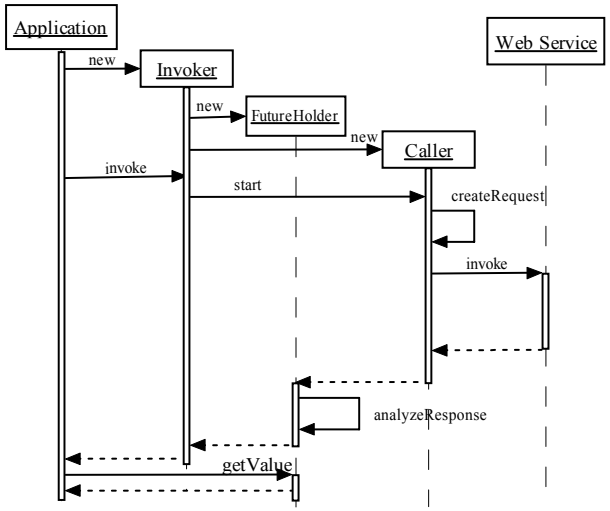

Figure 3. Synchronous invocation

Figure 3 shows the sequence diagram for the synchronous invocation. The first thing to note is that also synchronous invocation receives a future object as result, but the control is returned to the invoking application only

when the value is computed and the returned future already contains the result of the operation.

The interaction is performed with the following steps. The application creates the invoker object passing the parameters. The constructor of the Invoker class creates an instance of FutureHolder and one of Caller. The Caller instance performs the creation of the SOAP request message and invokes the Web Service synchronously. When the result is computed it is returned with a response message to the Caller. The Caller object updates the Future Holder object that de-serializes the result and signals to the Invoker that the execution is completed. The invoker, then, returns the control to the application that could act on the Future Holder, received as result, invoking the getValue() method and obtaining the result of the operation.

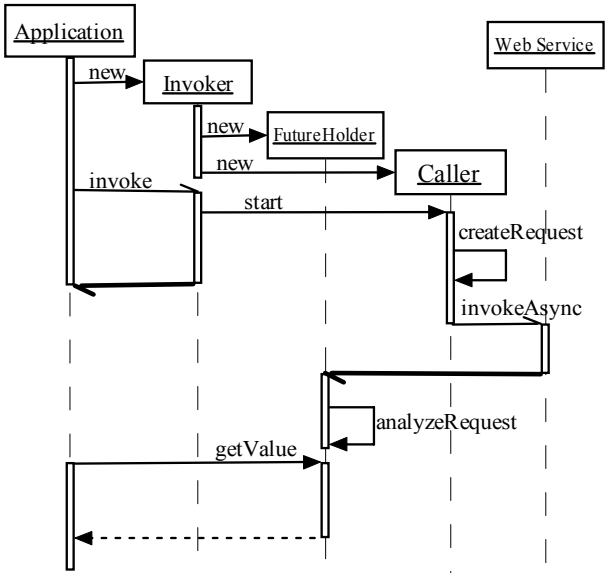

Figure 4. Non-blocking Asynchronous Invocation

Figure 4 shows the sequence diagram for the asynchronous invocation. The interaction is the same either using the call-back handler or using the polling object. The only difference is that in the case a call-back handler is employed, the FutureHolder is used also in that role. The sequence of interaction is quite similar to the synchronous case. The differences are that the Invoker object returns a FutureObject to the application that can continue its execution without blocking.

The reference to the FutureObject resolves the problem of the correlation between request and the corresponding result. The invocation of the Web Service is performed using the asynchronous method provided by JAX-WS. When the result is computed, it is returned to the Caller that updates the Future, in the case of pollable object invocation.

In the other case, the call-back handler, which is the FutureHolder itself, is invoked and used to store the value that is available for the application. If the request for value, with getValue() method, is performed by the client after the Future updating, the invoking application could continue the execution without waiting.

Figure 5, on the other hand, depicts the situation in which the invoking application tries to act on the FutureHolder before the value is updated. In this case, the accessing thread is blocked in the getValue() method and can not continue its execution. The thread is awakened when the result of the operation is provided by the Web Services to the FutureHolder. From the application point of view, the use of the pollable object is similar to the asynchronous invocation with callback, but the client application is responsible to test whether the value is ready, with the isDone() method, to avoid blocking with the invocation of the getValue() method.

The autonomous management of the invocation can be done deploying services using the proposed optional WSDL extension that gives semantic information about the service implementation: the complexity flag for the operation attribute of Web Service. This optional flag could be used by the provider to define the computational complexity of the operation and allows the dynamic invoker module to choose the style of invocation.

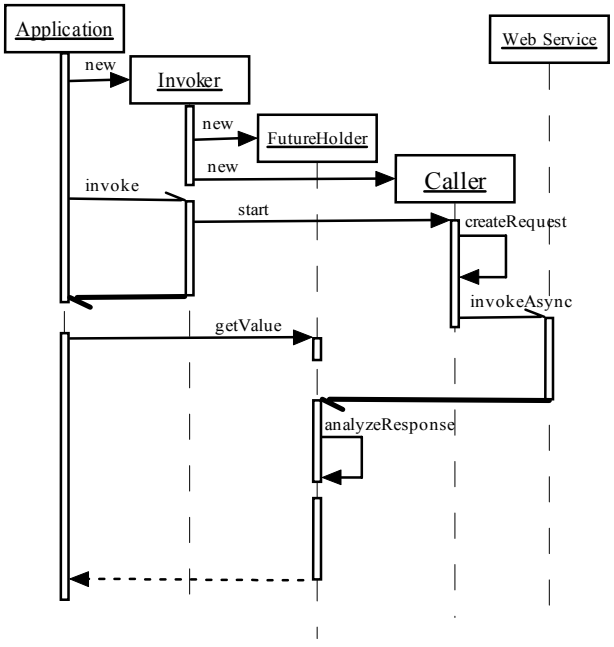

Figure 5. Blocking Asynchronous Invocation

Figure 6 shows the client application provided with the Dynamic Invoker, useful to test the different ways of interaction. We have used the graphical client to invoke

Web Services developed by ourselves and provided by other organizations on the Internet.

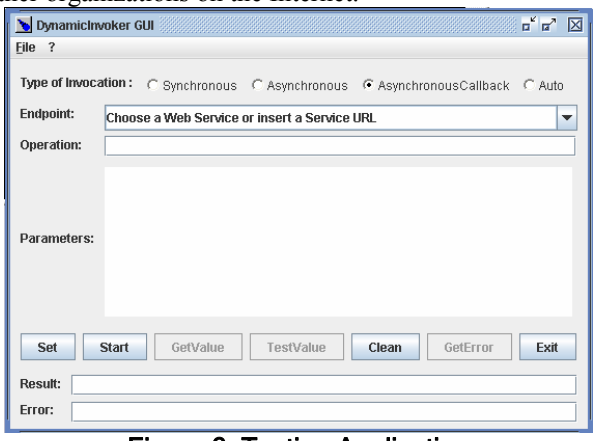

Figure 6. Testing Application

From the user point of view, the result of the invocation is always a future object that wraps the real result. The access to the result is obtained always with the method getValue, but it is more correct to test always the availability of the result using the method isDone. This is particularly true if the invocation style is chosen autonomously by the dynamic invoker.

4.1. Workflow and Grid computing improvements

The asynchronous invocation could be used in workflow execution to obtain performance optimization, by mean of activities anticipation, as we have discussed in our previous work [11].

To obtain activities anticipation it is possible to use asynchronous invocation techniques. The asynchronous invocation could be a mechanism that, as we have shown in [11], allows the process enactor, the Workflow Engine [23], to not suspend process execution waiting for activities completion.

Our contribution was the definition of a Workflow Pattern, the Early Start Pattern, which can ease the modelling effort and, at same time, allows for performance improvements. The experimental work was conducted using the ProActive Middleware to develop services able to be invoked asynchronously. The key point is the exploitation of fine-grained concurrency at run-time to overlap sequential activities execution, completing process in a minor total time and improving performances.

Our goal is to integrate the same concepts implemented for obtaining fine-grain concurrency in Web Services environments and, consequently, in modern Grid systems [24] [25].

Web Services are a useful way to model Grid Resources and to obtain better interoperability and platform independence [26]. As we have shown in a previous work [12], execution of a workflow process in Grid is a problem able to be separated in two main parts: scheduling of activities and mapping to appropriate resources. The Early Start Pattern could be advantageously introduced in Grid systems to perform optimistic scheduling. Such kind of scheduling may be very useful in performance improvements thanks to the overlapping of dispatching, scheduling and other managing activities of grid systems with the computational activities. The overlapping of Grid activities contribute to improve performance also because it increases the resource allocated at same time. This is a very useful way to shorten the total execution time as shown in figure 7.

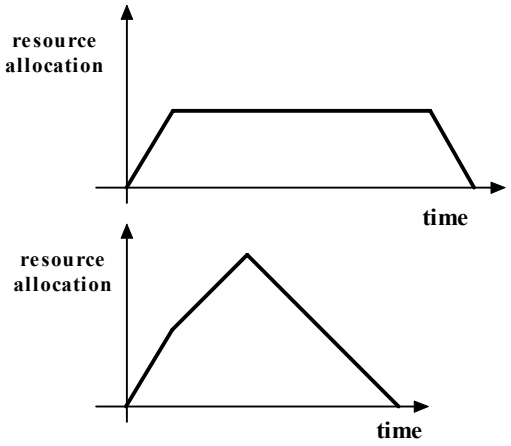

Figure 7. Resource allocation with conservative sequential and optimistic anticipated executions

Another important point that emerges from both our preceding works is the possibility to transform the control flow into dataflow synchronization for some parts of a process.

Using the early start pattern it is possible to obtain such advantages in a simple way keeping the modelling easy. Both our works have shown results obtained from execution of services modelled as active objects provided by ProActive. Our future objective is to extend such results to Web Services implementing asynchronous invocation and automatic continuation in a middleware for services invocation. The DynamicInvoker described in this work represents the first step in such direction.

5. Conclusion and future works

In this paper, we discussed the problem of asynchronous invocation. We tackled the problem of designing a Dynamic component that is able to exploit the features offered by the JAX-WS library. The module is able to perform asynchronous invocation of Web Services

resolving the problem of correlation between the invocation and the result. Moreover the module is also able to perform the invocation dynamically, acting as a proxy for the invoking application. We presented also a proposal to label Web Service operation with an additional tag to indicate computational complexity to allow for automatic selection of the invocation style.

The implemented module, furthermore, is more than a solution for asynchronous invocation. It opens a new development style in the field of Web Services. In our current activity, we are working on an extension of the module to realize a middleware that is able to perform asynchronous invocations and also automatic continuation as in ProActive [20]. We are testing the automatic continuation using future objects to invoke other services. To obtain future updating, we are using WS-Addressing [15] in order to implement updating server-side strategies, which do not require further client interventions. The objective is the realization of a new methodology in Web Services consuming that supports an object-oriented approach to distributed application development based on Web Services composition. One of the advantages that could be obtained is the possibility to use the fine-grained concurrency [11][12], that we have tested with RMI and ProActive, to execute composed Web Services. It is also possible to adopt Web Services to wrap workflow client applications so allowing asynchronous interactions towards human participants in workflow processes.

Furthermore, in our current activity, we are conducting performance measurements to give a quantitative evaluation of the performance gain obtainable with asynchronous invocation of Web Services, since we have already obtained significant improvement of performance in a workflow management system whose resources are modelled as active objects.

Acknowledgements

We thank Fabio Muollo for his contribution during the development phase of DynamicInvoker. The work described in this paper is framed within the LOCOSP Project funded by Italian Ministry of Research and Education (MIUR) [22].

References

[1] W. R. Cook, J. Barfield, “Web Services versus Distributed Objects: A Case Study of Performance and Interface Design”, in Proceedings of the IEEE Intl. Conf. on Web Services (ICWS) pp. 419-426, 2006.

[2] W. Vogels, “Web services are not distributed objects.” IEEE Internet Computing, vol. 7, no. 6, pp. 59–66, 2003.

[3] K. P. Birman, “Like it or not, web services are distributed objects,” Communication of ACM, vol. 47, no. 12, pp. 60–62, 2004.

[4] M. Brambilla, S. Ceri, M. Passamani, A. Riccio, “Managing Asynchronous Web Services Interactions”, in Proceedings of the IEEE Intl. Conf. on Web Services (ICWS), 2004, 80-87.

[5] H. Adams, Asynchronous operations and Web Services. IBM, http://www-128.ibm.com/developerworks/. webservices/library/ws-asynch1.html

[6] R. R. Kodali, “Asynchronous Web Services Using WS-Addressing”, SOA Web Service Journal, 2006, http://webservices.sys-con.com/read/183956.htm

[7] J. Yang, C. Cîrstea, P. Henderson, “An Operational Semantics for DFM, a Formal Notation for Modelling Asynchronous Web Services Coordination”, QSIC 2005, pp. 446 - 451.

[8] C. Peltz, “Web services orchestration and choreography”, Computer, vol. 36, 2003, pp. 46-52.

[9] Jia Yu and Rajkumar Buyya, “A Taxonomy of Workflow Management Systems for Grid Computing”, Volume 3, Numbers 3-4, pp. 171-200, Springer Science+Business Media B.V., New York, USA, Sept. 2005.

[10] Denis Caromel, “Towards a Method of Object-Oriented Concurrent Programming”, Communication of ACM volume 36, number 9, 90-102, 1993.

[11] G. Tretola, E. Zimeo, “Workflow Fine-grained Concurrency with Automatic Continuations”, in Proceedings of the IEEE IPDPS 06, 20th International Parallel and Distributed Processing Symposium, Rhodes Island, Greece, April 25-29, 2006.

[12] G. Tretola, E. Zimeo, “Activity Pre-Scheduling in Grid Workflow”, to appear in Proceedings of the 15th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP), February 7-9, 2007.

[13] “Web Services Architecture”, http://www.w3.org, W3CWorking Group Note 11 February 2004.

[14] WSRF, http://www.globus.org/wsrf/.

[15] WS-Addressing, http://www.w3.org/Submission/ws-addressing/.

[16] JAX-WS 2.0, https://jax-ws.dev.java.net/.

[17] Axis, http://ws.apache.org/axis/.

[18] Workflow Management Coalition, “The Workflow Reference Model”, Document Number WfMC TC-1003, www.wfmc.org.

[19] XMLSchema 1.1, http://www.w3.org/XML/Schema.

[20] Proactive, www-sop.inria.fr/oasis/ProActive/.

[21] Java language, http://java.sun.com.

[22] LOCOSP Project, http://plone.rcost.unisannio.it/locosp.

[23] Workflow Management Coalition, http://www.wfmc.org.

[24] Jia Yu and Rajkumar Buyya, “A Taxonomy of Workflow Management Systems for Grid Computing”, Journal of Grid Computing, Springer Press, New York, USA, 2005.

[25] The Grid Workflow Forum, www.gridworkflow.org.

[26] Globus Project. www.globus.org.

[27] PattiChiari. http://www.pattichiari.it/.

[28] MetalC site. http://www.metalc.it/.