Extending Web Services Semantics to Support
Asynchronous Invocations and Continuation

Giancarlo Tretola
University of Sannio
Department of Engineering
Benevento, 82100 Italy
tretola@unisannio.it

Eugenio Zimeo
University of Sannio
Research Centre on Software Technology
Benevento, 82100 Italy
zimeo@unisannio.it

Abstract

Asynchronous invocation and continuation are common patterns in some middleware infrastructures for object-based distributed computing. Their benefits are particularly significant in distributed environments characterized by high communication latencies and coarse-grained operations. Therefore, Web Services could strongly benefit from the adoption of these patterns to (1) overlap communication with computation, (2) reduce the high number of interactions typically needed to handle stateless services by migrating the state of a service as parameters of service operations, (3) intercept at run-time data dependencies among consecutive services in a composition not visible from service descriptions. Unfortunately, current semantics of Web Services do not directly support the patterns, but some specifications (i.e. WS-Addressing) can simplify their implementation. In the paper we present the patterns, their benefits, and a module that implements a flexible schema useful to perform asynchronous invocations in several contexts. This way, modelling composed services can benefit from abstractions whilst more sophisticated low-level interactions among services are automatically handled at run-time.

1. Introduction

Web Services are today an accepted and common technology to develop distributed applications both in inter-enterprise and open environments. However, the technology is not sufficiently mature to handle complex interactions, so introducing some limitations at modelling and developing times. As an example, consider a Web Service that allows delivering a service or a product, we may describe it with a single synchronous operation. This operation is blocking for the client if it needs the result of the operation or an acknowledgement. Alternatively we can use an asynchronous approach choosing to define two operations [5]. The first allows for requiring the functionality, while the second may be used to give the operation result. This is possible using several patterns of interaction: polling or callback for example [8]. In both cases, the state between the two method calls has to be stored and the answer should be associated to the correct requester. Therefore, even though asynchronous invocation is feasible with current web services technology, it put a lot of burden on the programmer that has to define web services interfaces taking into account non-functional mechanisms during the programming of applications. Asynchronous invocations have many advantages. 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 high communication latencies in 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 homogeneously consider interactions with humans and machines in order to handle them in the same way at control level.

We believe that a step towards extended semantics for web services is necessary to exploit the technology in a flexible way and for different applications. Such extension could allow managing service interaction and composition using a new interaction pattern in a transparent way. To this end, we propose an architectural pattern that, hiding the invocations details, allows for dynamic and asynchronous calls,
keeping simple the implementation of both services and clients. The pattern introduces asynchronous calls in which the return value is modelled as a placeholder of the future service result to be computed, according to the semantics introduced in concurrent programming [11].

Our proposal is particularly focused on extending synchronous invocations with automatic continuation and anticipation. To achieve continuation and anticipation, we propose an extension of Web Service asynchronous invocations that enable the placeholder to be propagated to subsequent services, as parameter, in order to anticipate activation and so allowing the continuation of operations that do not depend on invoked activities results. Such functionality could be very valuable for Web Service based applications. To this end, considering composed services, modelled as a workflow process, we have described and tested a new workflow description pattern for workflow processes: the Early Start Pattern. It can be used to ease the modelling and, at same time, to improve performance in sequential activities enactment, using a supporting middleware [12]. The central concept is the activity anticipation that allows for exploiting fine-grained concurrency at run time.

The Early Start Pattern has relevant consequences in other fields related to Web Services. It is useful in Workflow Management [20], for example, considering the interaction with functionality offered as Web Services. In the context of Grid computing, where applications are seen as workflows that handle Grid resources [10][26], the Early Start pattern could be used to increase resource exploiting as described by a previous work of the authors [13].

The implementation of the Early Start pattern is bond to the possibility of asynchronous invocation using placeholders for results and the ability of subsequent activities to accept the placeholder as a parameter. We have identified four requirements for Early Start pattern:

- asynchronous invocation with symbolic placeholder for the result;
- forwarding of placeholder to subsequent activities;
- stalling of activity that attempt to use the result not computed yet;
- updating of the result transparently to all activities that make use of it.

In our previous experimentation, we used ProActive [22], a Java middleware that satisfies all the requirements using Active Objects and Future Objects. We used ProActive to develop components for distributed objects based application. In order to transfer to Web Services this workflow description pattern the first step to perform is the introduction of asynchronous invocation returning future objects in Web Services interactions. We have already performed the implementation and evaluation of a module for dynamic and asynchronous invocation of Web Services [14]. That work is the starting point of the current development which is involved in the exploiting of future objects to achieve continuation and anticipation with Web Services.

The remaining part of the paper is organized as follows. Section 2 describes related work on asynchronous Web Services invocation and discusses the interaction patterns that could be used for asynchronous invocation of Web Services. Section 3 proposes the ideal reference architecture for asynchronous interaction with Web Services. Section 4 describes the design and implementation of a library that concretizes the ideal architecture. Section 5 shows the performed test and their results. Section 6 concludes the work outlining possible employments and open issues.

2. Related works

Asynchronous 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 [5]. 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 idea of asynchronous invocation is present in the literature and it is recognized that is a valuable mechanism.

Considering Web Services, it is possible to analyze the interaction behaviour starting from the operation classification [8]. Each operation is associated to composing messages. There are two possible associations, as depicted in Figure 1.

![Figure 1. WS operation classification](image)

The synchronous operation contemplates a request message followed by a response message, this interaction is initiated by the client (request/response) or by the server (solicit response). In the asynchronous interaction, the operation is associated to only one message sent by the client with the one way or by the server with notification. Four patterns for
asynchronous interactions are known for distributed objects interacting with an RPC style [8].

Fire and Forget consists in a pure asynchronous one way 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, with a notification message. 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 polling, i.e. a non blocking query about result availability. If the result is ready, it may be obtained by the client with a synchronous interaction, 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 an interface defined by the provider of the service, 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.

Several Web Services improving attempts are aimed to interaction and implementation issues. Several works are involved with bringing asynchronous service invocation in the Web Services world [4][6][7]. Other activities have proposed the introduction of Object-Oriented techniques in Web Services modelling [1].

In [4], the authors describe the results of their studies about Web Services asynchronous invocation. 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 [28].

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 involve also human participants and human interactions. Both of them benefit of asynchronous interactions. The followed approach uses WS-Addressing [17] coupled with a call-back-based technique.

In [7], the authors tackle the problems that arise when asynchronous invocations are performed in complex applications composed with Web Services. They have defined a Document Flow Model (DFM), a message-based workflow modelling of asynchronous interactions among Web Services in workflow processes.

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.

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 components we have realized are in charge to dynamically perform the invocation that could be synchronous or asynchronous [14].

3. Modelling asynchronous invocation and continuation

Starting from the developed invocation component [14], we have analyzed the requirements to introduce anticipation and continuation in Web Services modelling and execution. The requirements we tried to satisfy were about using the symbolic placeholder as parameters for other services. Moreover it is necessary to monitor the future object, intercepting access to the placeholder and stalling the accessing application whereas the result is not available. Finally it is necessary to update the computed result to the placeholder in order to allow completion of the
computation. In Figure 2 it is depicted the reference scenario of the asynchronous invocation pattern we have defined. The Invoker calls an operation on WS A using asynchronous invocation. The result is a placeholder that is stored in the Repository. Invoker, then, uses the placeholder as parameter for invoking WS B (forwarding the placeholder as actual parameter).

When A completes the functionality execution, the result is sent to the Repository (updating). When WS B tries to access to the placeholder value, it asks for the value to the Repository. If it is available, the repository replies it to B; otherwise the request servicing is suspended until WS A performs its updating, effectively stalling WS B execution.

Figure 2. Collaboration diagram of the pattern

Figure 3 depicts the conceptual architecture of the placeholder. The PlaceholderType interface defines the fundamental type: a placeholder or future to the result that may be returned instead of the real result. The operations defined for the placeholder type are two. isFuture is a non blocking function that informs whether the object is still a future or if it contains the computed result. Non-blocking means that the invocation always returns a result also if the placeholder is not updated, so avoiding the stall of the caller. The method obtainValue is a blocking method that returns the result of the computation when it is available otherwise the caller is placed in a busy waiting state. The PlaceHolderType is parameterized on the type T of the result to manage. The AbstractPlaceHolder is an abstract class that uses the Strategy and Template Method design patterns. It contains the template implementation of the operation defined in the PlaceHolderType, defined as leaf method that may not be overridden. This method specifies the algorithm for accessing to the future value using the hook methods extractValue and assignValue, defined abstract. These methods are the hot-spots that enable the insertion of the customized code to manage return values. The other fundamental type defined is ReferenceType, which is used for associating to the placeholder the information about the repository for its storage and access.

Figure 3. Result management

Figure 4. Abstract Repository

The role of Repository, shown in Figure 4, is to contain the placeholder used in a Web Service interaction, answering to the request for the value that receiver of the placeholder may formulate. The repository stores the placeholder generated as return for asynchronous call and, when receives the computed result, updates the value in the placeholder. The updated value is available for each requestor that made request before or that will make request after. Requests stated before the updating have the outcome of stalling the requestor.

The defined architecture could be implemented in different ways and using different languages. Interactions with Web services are made by SOAP. If a Web Service involved in a complex interaction presents an implementation compliant to the defined architecture, it is possible to make asynchronous invocations and perform continuation.

Continuation is useful for the called WS that could perform other computation while waiting for the result of the operation. In this case, we can give a simple solution to the scenario depicted in the introduction about the web services that delivers a service or a product. The modelling solution is simple to implement in that it presents an interface similar to the synchronous services, but benefits from asynchronous features offered by the supporting middleware that uses our components. The service, in fact, may be modelled with only one method that returns the outcome of the operation. In this way correlation between caller and called is performed by the result future object and managed in the supporting middleware. It is how
session information was encapsulated in the placeholder. Furthermore, another possible employment is the realization of composed services or workflow processes that make use of anticipation, using Early Start Pattern [12] for example.

4. Design and implementation

To test and evaluate the proposed architecture we have designed and developed a Java implementation of the interaction pattern. depicts the design of the reference architecture implemented in Java language.

![Figure 5. Java Design](image)

There is the introduction of a type used in transmission, designed as a Java Bean, for serialization and de-serialization using Axis libraries [19]. The design was performed introducing methods that are compliant with getter and setter patterns for attributes. The placeholder type has been not changed and also the abstract class AbstractPlaceHolder presents the same design. The Repository, implemented also in Java and deployed as a Web Services, could be deployed everywhere on the network. The services invocation and the interaction between services and repository were implemented using JAX-WS 2.0 [18].

During implementation, we solved several issues and in particular de-serialization of parameterized classes using Axis. The parameterized placeholder that contains a null object could not be de-serialized correctly, so we have added another layer in the architecture. In that layer we implemented classes that wrap primitive types, like Integer, String and so on. In this way we have already the object to use and may perform the serialization and de-serialization without problems.

Considering updating of futures with the computed result, it is possible to consider two peculiar features: responsible of updating and time of updating. When the responsible is the requestor, the strategy is classified forward based, while if the responsible is the called object it is labelled home based. If result is updated to all that received the future it is called eager. If it is updated only to whom making a request for the value it is labelled lazy. We have developed two solutions to address the two features. In the first one when the real result is computed, it is returned to the caller that has the responsibility to update the repository, in a way similar to an eager forward based strategy [11]. After that step, however, the updating is like a lazy home based updating strategies where the responsible (home) is considered to be the repository itself. When the real result is computed it is returned to the caller that has the responsibility to update the repository. The other implementation makes use of the WS-Addressing [17] to specify the resulting data destination. Specifically we use the replyTo field of the WS-Addressing to specify, when the invocation is made, which is the service to whom send the computed result, in our case the repository. In such a way it is possible to obtain an eager home based strategy from called service to the repository. After the first updating step of the future the updating strategy is always lazy home based, like the previous case.

Other issue we had to resolve was about implementation of the repository. To save time we did not implement an interaction to create the placeholder on the repository. This operation is performed only when there is the first request for a result value. The repository check in its table for the received placeholder unique ID, and if there is not a placeholder with the ID provided by the requestor, then a new placeholder is created, associated with that ID, and the thread servicing the request is put in a waiting state, causing also the stalling of the requestor. When the placeholder is updated with the computed result, every pending request is served. To be able to serve future requests, the repository hold the placeholder for a period of time that may be specified at the creation of the repository itself, basing on domain specific application duration.

A temporized thread is in charge of the placeholder deletion when the allowed period of time is expired. In the actual implementation, furthermore, the timer is restarted every time that a new request for the value is performed, to avoid premature deletion.

5. Testing and evaluation

To test the performance of the proposed solution we have considered three different configurations mapped on the deployment architecture shown in Figure 6. The Invoker component is responsible of calling operations on two Web Services, grouped in a sequence pattern.
In our test we executed the sequence of two Web Services using four different techniques. In all the execution we used Axis and JAX-WS, and to have comparable application, we employed our Dynamic Invoker Component [14]. Table 1 defines the names of the four tests used.

<table>
<thead>
<tr>
<th>Test type</th>
<th>Technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sync</td>
<td>Synchronous and sequential</td>
</tr>
<tr>
<td>Async</td>
<td>Asynchronous invocation, client side based</td>
</tr>
<tr>
<td>Placeholder</td>
<td>Asynchronous invocation, automatic continuation and anticipation</td>
</tr>
<tr>
<td>Placeholder+WSA</td>
<td>Asynchronous invocation, automatic continuation and anticipation, WS Addressing</td>
</tr>
</tbody>
</table>

The first execution, labelled Sync, was performed using classic synchronous invocation and sequential enacting, and gives the reference time. Then we used the Dynamic Invoker to model the asynchronous invocation based on client-side implementation. The third and fourth tests were realized using our components to implement automatic continuation. The implementation used in the third case was the one without WS-Addressing, so the invoker is responsible of updating to the repository. In the fourth one we used WS-Addressing and the updating of the repository is done directly by the called service. To complete the interaction, we used different repository deployments to manage the placeholder returned from the first service of the sequence. In the first run of tests, we deployed the repository on the same computer of the invoker. In the second case, the invoker was deployed on the same computer that hosted service WS1. In the third case we placed the invoker on a fourth host, which hosted only the repository.

The tests were aimed to determine the entity of the performance improvement in the two extreme cases of variation. The performance improvement is bond to the dependence point in the second activity, i.e. the first operation that requires the resulting of the preceding activity, as we shown in preceding works [12][13]. The ideal worst case is when the first operation of the subsequent activity requires the result of the preceding one. The ideal best case is when the depending point is the last activity. To quantitatively assess the solution we used as performance index the inverse of the execution time. To make a measurement of expected performance we have considered two services with the same duration $T_1$ and $T_2$, so the total execution time of the sequence is greater than $T_1 + T_2$, considering the overhead of the invocation enactment and network delay. Considering services with $T_1 = T_2 = T$, the expected performance are bounded by two asymptotic values. The lower asymptote is the performance of the sequential execution ($1 / (T_1 + T_2)$). The execution time, in fact, is equal to the total time, that is the time $2T$ plus the sum of all other involved times. The higher asymptote is the performance of the two services executed concurrently. This time is greater than $T$, considering invocation and network delay.

The testing phase was executed as follow. Each test type was executed five times for each deployment case in the worst case configuration and then repeated for other five times in the best case situation. Each time we executed the runs, we considered the following values for $T$: 1 second, 5 seconds, 10 seconds and 20 seconds. The measured value was the total execution time by the client side point of view. For each value of $T$, the average time was considered. Recalling that we considered the performance as the inverse of the execution time, we calculated the speed up as the ratio of the performance of one test type versus the performance of synchronous invocation, considered as the reference. So the obtained value was used to estimate the speed up of the second, third and fourth test type versus the synchronous reference time, for the best and the worst cases, as indicated in the following table:

<table>
<thead>
<tr>
<th>Speed Up Computed Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1 Async Performance/Sync Performance</td>
</tr>
<tr>
<td>S2 Placeholder Perf./ Sync Performance</td>
</tr>
<tr>
<td>S3 (Placeholder + WSA) Perf./ Sync Perf.</td>
</tr>
</tbody>
</table>

The obtained results are presented in the following graphs subdivided for the three deployment cases and for the worst case and the best case $N$.
Figures 9 and 10 show the results of the second configuration, in which the repository was deployed with the first service. Again in this case, the WS Addressing solution presents a slight margin.

Figures 11 and 12 show the result when the repository was deployed on a third host, which is different from both the one used for invoker and service 1. From a quantitative point of view, it is possible to state that the asynchronous invocation, when used alone could not ensure performance improvement. In our tests simply asynchronous invocation is similar, in performance, to synchronous invocation.

Moreover synchronous invocation is quite simple to implement with respect to asynchronous invocation, which requires additional modelling and developing effort. Using our Dynamic Invoker [14], however, the complexity for realizing an asynchronous invocation is very simple.

Asynchronous invocation coupled with futures and automatic continuation is more interesting. The performances observed are quite good, ranging from a minimum of 160% to a maximum of 197% of speed up in the best case. Considering the worst case we may do an important observation: performance improvement is never negative compared to the classic sequence enactment. It is important to note also the fact that WS addressing could ensure an improvement in performance that varied from 10% to 3%, depending on the placeholder implementation that does not employ WS Addressing.

6. Conclusion

In this work we presented asynchronous invocation and continuation of Web Services. We have defined an abstract interaction pattern that identifies the principal components and interaction necessary to support asynchrony and autonomic behaviours.
Then we designed and developed two implementations of our conceptual pattern and we conducted several kinds of testing to assess performances. From a qualitative point of view we showed that asynchronous invocations and futures perform always better that synchronous invocation.

It is worth to note that, asynchronous invocation and automatic continuation could be implemented also without our components. But the correlation between a call and its result as well as the intra-call state management must be supervised at the application levels, and then they cause an increment in modelling complexity and development difficulty.

Our components, instead, may ensure a margin in performance while modelling stay simple and clear. So it could be possible to extend the semantics of Web Services introducing non-functional attributes that describe the type of implementation, its ability to be invoked asynchronously or not. Another non-functional annotation could be the possibility to receive a placeholder as parameter to enable data synchronization. All this information then could be used by automatic process composition system to automatically define the process description. We are aimed to further investigate performance, considering cases in which services are long-time running and also situations that present services with different durations. Finally, more sophisticated techniques for updating and managing placeholders in the repository could be investigated.

7. References

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