5th SASTech
Iran | Mashhad
May 12 - 17 | 2011
5th Symposium on Advances in Science & Technology

A Comprehensive Assessment and Comparison of Asynchronous Invocation Patterns and Frameworks

Amir Moazeni

Shahab Danesh Institute of Higher Education

a.moazeni@shahabdanesh.ac.ir

Abstract

Asynchronous invocations are an important functionality in the context of distributed object frameworks, because in many situations clients should not be blocked during remote invocations. There should be a loose coupling between clients and remote services. Popular web service frameworks, such as Apache Axis, offer only synchronous invocations (over HTTP). An alternative are messaging protocols but these implement a different communication paradigm. When client asynchrony is not supported, client developers have to build asynchronous invocations on top of the synchronous invocation facility. But this is tedious, error-prone, and might result in different remote invocation styles used within the same application. In this paper the challenges of asynchronous invocation of web services is discussed. Besides, we have investigated and compared a set of client asynchrony patterns and frameworks.

Key words: web service, asynchronous invocation, framework, pattern

1. Introduction

In this paper the problem of asynchronous invocation of web services is discussed. Although there are many kinds of distributed object frameworks that are called web services, a web service can be described by a set of technical characteristics, including:

– The HTTP protocol [6] is used as the basic transport protocol. That means, remotely offered services are invoked with a stateless request/response scheme.

– Data, invocations, and results are transferred in XML encoded formats, such as SOAP [5] and WSDL [7].

– Many web service frameworks are extensible with other transport protocols than HTTP.

– The services are often implemented with different back-end providers (for instance, a Java class, an EJB component, a legacy system, etc.) [8].

Advantages of this approach to invoke remote objects [12] are that web services provide a means for interoperability in a heterogeneous environment. They are also relatively easy to use and understand due to simple APIs, and XML content is human-readable.

Further, firewalls can be tunneled by using the HTTP protocol. In the spirit of the original design ideas of XML [6] (and XML-RPC [14] as the predecessor of today's standard web service message format SOAP) XML encoding should also enable simplicity and understandability as a central advantage. However, today's XML-based formats used in

web service frameworks, such as XML Namespaces, XML Schema, SOAP, and WSDL [7], are quite complex and thus not very easy to comprehend.
Many web service frameworks, such as Apache Axis [3], only allow for synchronous Invocations (for synchronous transport protocols such as HTTP). That means the client process (or thread) blocks until the response arrives. For client applications that have higher performance or scalability requirements the sole use of blocking communication is usually a problem because latency and jitter makes invocations unpredictable [1].
In such cases we require the client to handle the invocation asynchronously. That means, the client process should resume its work while the invocation is handled. Also the intended loose coupling of web services is something that suggests asynchronous invocations, that is, the client should not depend on the processing times of the web service.
There are four asynchronous invocation patterns. Each pattern has a specific usage. This collection of patterns introduces the four most commonly used techniques in the context of asynchronous invocation. In second section of this paper the patterns would be evaluated.
Based on these patterns, several frameworks are implemented. In third section three of mentioned frameworks are assessed.
In fourth section the frameworks have been compared and the pros and cons of those have been verified. Fifth section concludes this paper.

2. Asynchronous Invocation Patterns

In this section, we present a set of client asynchrony patterns [13] that are part of a larger pattern language for distributed object communication.
A pattern is a proved solution to a problem in a context, resolving a set of forces. Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution [2]. A pattern language is a collection of patterns that solve the prevalent problems in a particular domain and context, and, as a language of patterns, it specifically focuses on the pattern relationships in this domain and context. As an element of language, a pattern is an instruction, which can be used, over and over again, to resolve the given system of forces, wherever the context makes it relevant [11].
The client asynchrony patterns are in particular:

2.1. FIRE AND FORGET

Your server application provides remote objects with operations that have neither return value nor report any errors [13].
In many situations, a client application needs to invoke an operation on a remote object simply to notify the remote object of an event. The client does not expect any return value. Reliability of the invocation is not critical, as it is just a notification that both client and server do not rely on.
Consider a simple logging service implemented as remote object. Clients use it to record log messages. But recording of log messages must not influence the execution of the client. For example, an invocation of the logging service must not block. Loss of single log messages is acceptable.
Therefore, in FIRE AND FORGET invocation, the client proxy sends the invocation across the network, returning control to the caller immediately. The client does not get any acknowledgement from the remote object receiving the invocation.

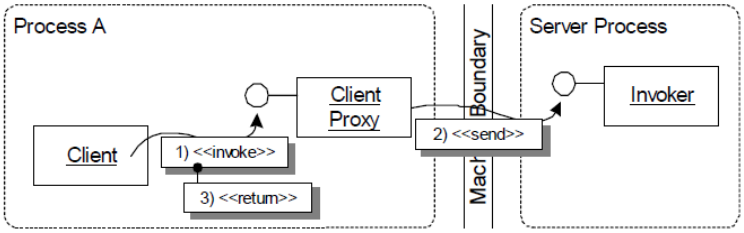

Fig.1: FIRE AND FORGET Pattern [13]

However, errors in transmission to or execution of the remote object cannot be reported back to the client. The client is unaware whether the invocation ever got executed successfully by the remote object. Therefore, FIRE AND FORGET usually has only "best effort" semantics. The correctness of the application must not depend on the "reliability" of a FIRE AND FORGET operation invocation. To cope with this uncertainty, especially in situations, where the client expects some kind of action, clients typically use time-outs to trigger counter-actions.

2.2. SYNC WITH SERVER

FIRE AND FORGET is a useful but extreme solution in the sense that it can only be used if the client can really afford to take the risk of not noticing when a remote invocation does not reach the targeted REMOTE OBJECT. The other extreme is a synchronous call where a client is blocked until the remote method has executed successfully and the result arrives back. Sometimes the middle of both extremes is needed [13].

Consider a system that stores images in a database. Before the images are actually stored in the database, they are filtered, for example by a Fourier transformation that may take rather long. The client is not interested in the result of the transformation but only in a notification that it is triggered. Thus the client does not need to block and wait for the result; it can continue executing as soon as the invocation has reached the REMOTE OBJECT.

In this scenario, the client only has to ensure that the invocation containing the image is transmitted successfully. However, from that point onwards it is the responsibility of the server application to make sure the image is processed correctly and then stored safely in the database.

Therefore, in SYNC WITH SERVER invocation, the client sends the invocation, as in FIRE AND FORGET, but waits for a reply from the server application informing it about the successful reception, and only the reception, of the invocation. After the reply is received by the client proxy, it returns control to the client and execution continues. The server application independently executes the invocation.

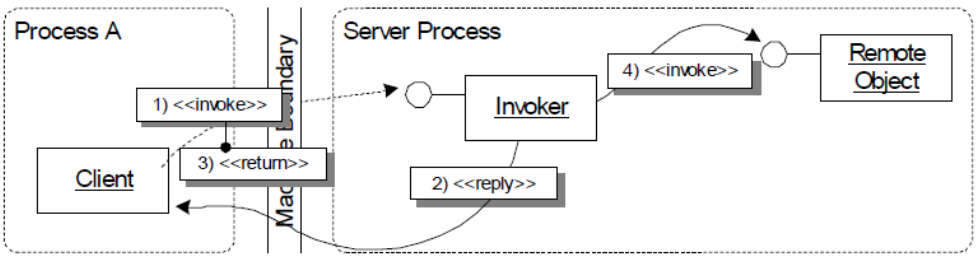

Fig.2: SYNC WITH SERVER Pattern [13]

Note that, as in FIRE AND FORGET, no return value or out parameters of the remote operation can be carried back to the client. The reply sent by the server application is only to inform the client proxy about the successful reception.

Compared to FIRE AND FORGET, SYNC WITH SERVER operations ensure successful transmission and thus make remote invocations more reliable. However, the SYNC WITH SERVER pattern also incurs additional latency - the client has to wait until the reply from the server application arrives. Eventually, it also has to retransmit the invocation.
Note that the client proxy can inform the client of system errors, such as a failed transmission of the invocation. However, it cannot inform clients about application errors during the execution of the remote invocation in the remote object because this happens asynchronously.

2.3. POLL OBJECTS

There are situations, when an application needs to invoke an operation asynchronously, but still requires knowing the results of the invocation. The client does not necessarily need the results immediately to continue its execution, and it can decide for itself when to use the returned results [13].
Consider a client that needs to prepare a complex XML document to be stored in a relational database that is accessed through a remote object. The document will have a unique ID, which is generated by the database system. Typically, a client would request an ID from the database, wait for the result, create the rest of the XML document, and then forward the complete document to the remote object for storage in the database. A more efficient implementation is to first request the ID from the database. Without waiting for the ID, the client can prepare the XML document, receive the result of the ID query, put it into the document, and then forward the whole document to the remote object for storage.
In general, a client application should be able to make use of even short periods of latency, instead of blocking idle until a result arrives.
Therefore, in POLL OBJECTS pattern, as part of the distributed object framework, provide POLL OBJECTS that receive the result of remote invocations on behalf of the client. The client subsequently uses the POLL OBJECT to query the result. It can either just query ("poll"), whether the result is available, or it can block on the POLL OBJECT until the result becomes available. As long as the result is not available on the POLL OBJECT, the client can continue asynchronously with other tasks.

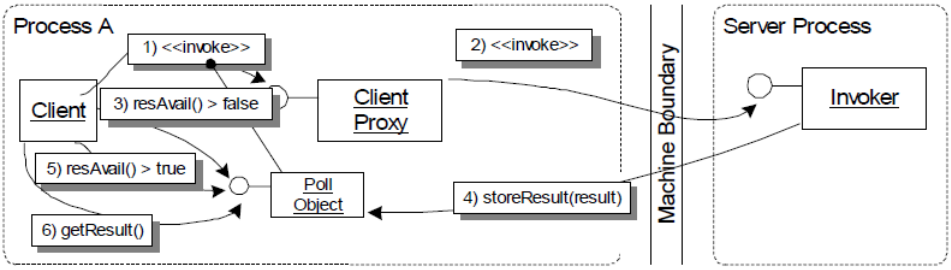

Fig.3: POLL OBJECTS Pattern [13]

The POLL OBJECT has to provide at least two operations: one to check if the result is available; the other to actually return the result to the calling client. Most POLL OBJECT implementations also provide a blocking operation that allows clients to wait for the availability of the result, once they decide to do so.

2.4. Result Callback

The client needs to be actively informed about results of asynchronously invoked operations on a remote object. That is, if the result becomes available to the client proxy,

the client wants to be informed immediately to react on it. In the meantime the client executes concurrently [13].
Consider an image processing example. A client posts images to a remote object specifying how the images should be processed. Once the remote object has finished processing the image, it is available for download and subsequent displayed on the client. The result of the processing operation is the URL where the image can be downloaded once it is available. A typical client will have several images to process at the same time, and the processing will take different periods of time for each image – depending on size and calculations to be done.
In such situations a client does not want to wait until an image has been processed before it submits the next one. However, the client is still interested in the result of the operation to be able to download the result.
Therefore, provide a callback-based interface for remote invocations on the client. Upon an invocation, the client passes a callback object to the client proxy. The invocation returns immediately after sending the invocation to the server. Once the result is available, the client proxy calls a predefined operation on the callback object, passing it the result of the invocation.

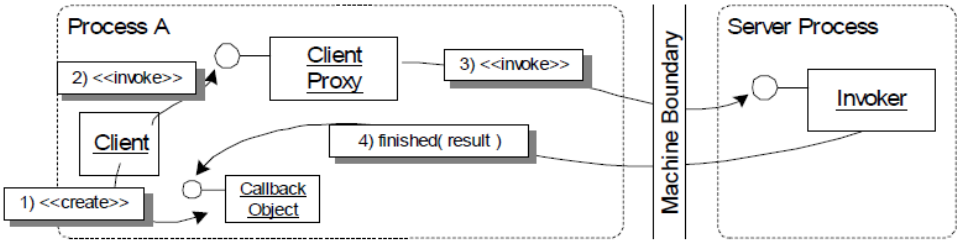

Fig.4: RESULT CALLBACK Pattern [13]

The major difference between poll object and RESULT CALLBACK is that RESULT CALLBACK requires an event driven design, whereas poll object allows keeping an almost synchronous execution model.

3. Asynchronous Invocation Frameworks

In this section, we explain some frameworks designed to implement the client-side asynchrony patterns, explained in the previous section.

3.1. ZDun Frameworks

ZDun's general design relies on the client proxy pattern [12]. A client proxy is provided as a local object within the client process that offers the remote object's interface and hides networking details. Client proxies can dynamically construct an invocation, or alternatively they can use an interface description [12] (such as WSDL).
In this framework, two kinds of client proxies are provided [14], one for synchronous invocations and one for asynchronous invocations. Both use the same invocation scheme. The synchronous client proxy blocks the invocation until the response returns.
A client can invoke a synchronous client proxy by instantiating it and waiting for the result:

```
SyncClientProxy scp = new SyncClientProxy();
String result = (String) scp.invoke(endpointURL, operationName, null, rt);
```

This client proxy simply instantiate a handler for dealing with the invocation, and after it has returned, it returns to the client.
The asynchronous client proxy is used in a similar way. It offers invocation methods that implement the four client asynchrony patterns discussed in the previous section. For this goal a client invocation handlers, corresponding to the kind of invocation, is instantiated in its own thread of control. The general structure of asynchronous invocation is quite similar to synchronous invocation. The only difference is that an AsyncHandler and clientACT are passed as arguments and do not wait for a result:

```
AsyncHandler ah = ...;
Object clientACT = ...;
AsyncClientProxy ascp = new asyncClientProxy();
ascp.invoke(ah, clientACT, endpointURL, operationName, null, rt);
```

Note that the clientACT field is used here as a pure client-side implementation of an asynchronous completion token (ACT). The ACT pattern is used to let clients identify different results of asynchronous invocations. In contrast to the clientACT field, the ACT is passed across the network to the server, and the server returns it to the client together with the result [14]. We do not need to send the clientACT across the network here because in each thread of control we use synchronous invocations and use multi-threading to provide asynchronous behavior. We thus can identify results by the invocation handler that has received it (or, more precisely, on basis of its socket connection). This handler stores the associated clientACT field.

3.2. ARMI Framework

RMI (Remote Method Invocation) is an interface specified by Sun Microsystems for the purpose of native java-client and Java-server communication. Although, RMI is simple to use, it is not desirable in many applications due to its synchronous nature. ARMI (Asynchronous RMI) is a mechanism which is built on top of RMI and allows concurrent execution of local and remote computations [10].
In RMI, when a remote method is invoked, the client is suspended while the computation is carried out in a remote address space. When the computation has completed, the return value, if any, is immediately variable to the caller for use. In the presence of an asynchronous method call, it is neither possible to predict when the result will be available nor when the caller will require it. These constraints necessitate another mechanism for returning values back to the client during an asynchronous method invocation.
ARMI uses a mailbox. The client must register a mailbox with a remote object before any asynchronous RMIs can occur. The client is the owner of that mailbox. When the client calls an asynchronous remote-method, the method returns a receipt generated by the mailbox. When the method has finished its computation, ARMI inserts the return value into mailbox, using the receipt as a key. The client can then query the mailbox at a later time to retrieve the value associated with the receipt. Figure1 shows an overview of ARMI mechanism. Along with the client, the server and the mailbox, figure1 also indicates a stub, a skeleton and a registry.

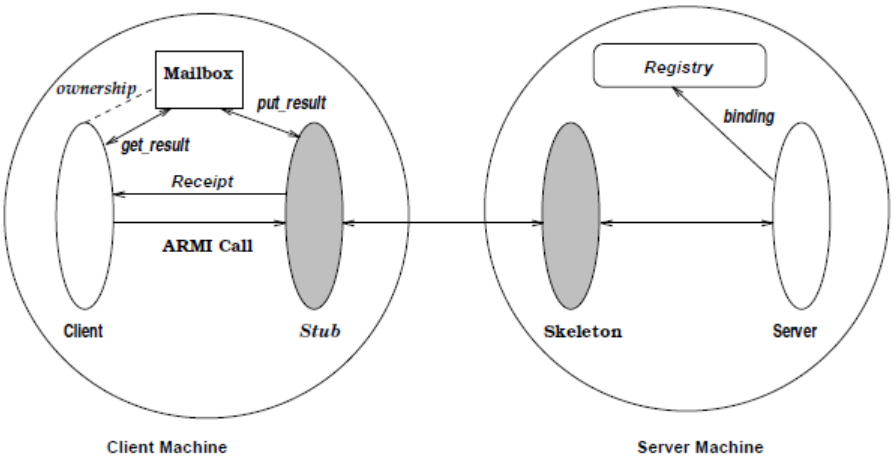

Fig.5: ARMI Architecture [10]

The mailbox provides the ability for the client to check if certain values are available, to retrieve a value associated with a specific receipt, to retrieve the next value inserted into the mailbox and to wait until a specific value is available.

3.3. EDAMI Framework

Asynchronous remote invocations enhance performance of distributed code, by allowing concurrent execution of the distributed components. Dependency of the statements which are being executed after a remote asynchronous invocation to the values affected by the invocation may be a barrier against concurrent execution of the caller and the invocation. To resolve the difficulty, statement reordering algorithms may be applied to increase the distance between the call statement and the very first positions where the results of the call are required [9]. A major difficulty in applying reordering algorithms is necessity to predict the execution time of the program code statically. These algorithms are static and do not consider the runtime behavior of the code to be reordered.

In order to achieve maximum concurrency in the execution of distributed code, execution of the statements which are dependent on any value affected by an asynchronous call statement is delayed by inserting them, at run time, into a separate thread which could be executed when the results of the remote call are required [9]. This approach is offered in a framework introduced by Zdun [14]. An important problem is that in Zdun framework asynchronous method invocation are restricted to web service technology. Also, in Zdun framework the caller has to wait for the results of an asynchronous call by applying a busy waiting method. In EDAMI framework notification of completion of asynchronous calls is sent to the callers through events raised by the invocation. In addition, presenting a layered architecture for asynchronous remote invocations has made it possible to apply any middleware supporting remote calls, within this framework.

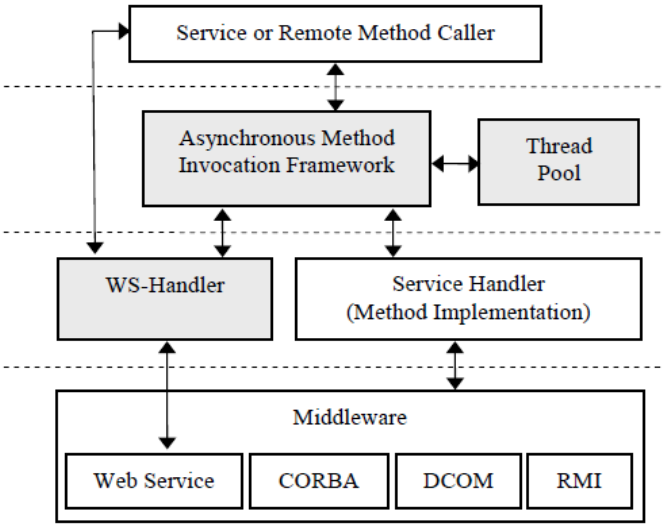

Fig.6: EDAMI Architecture [9]

4. Comparison result

Table 1 illustrates the alternatives for applying the patterns. It distinguishes whether there is a result sent to the client or not, whether the client gets an acknowledgment or not, and, if there is a result sent to the client, it may be the clients burden to obtain the result or it is informed via a callback.

	Result to client	**Ack to client**	**Responsibility for result**
FIRE AND FORGET	No	No	No
SYNC WITH SERVER	No	Yes	No
POLL OBJECTS	Yes	Yes	Client is responsible
RESULT CALLBACK	Yes	Yes	Client is informed via callback

Table1: Comparing asynchronous invocation patterns

The rest of this section is dedicated to comparison of the mentioned frameworks in previous sections.

ARMI framework: This framework's design is based on Java RMI. Two main drawbacks of ARMI are: ARMI only supports one of the asynchronous invocation patterns (Poll Objects) and it is not applicable in *www* for web service invocations.

ZDun frameworks: This framework supports four asynchronous invocation patterns. It is implemented based on Apache Axis and provides different structures for different invocation patterns. Indeed the programmer cannot use this framework uniformly.

EDAMI framework: dependency between program codes and return values is an important point which has not been considered in previous frameworks. In EDAMI framework the concurrency is improved by putting the dependent codes in a thread and executing the threads independently. Middleware-independency is another important point which is considered in this framework. This framework provides RESULT CALLBACK pattern and other patterns are not supported in this framework.

5. Conclusions and future work

In this paper we evaluated the four asynchronous invocation patterns namely Fire and Forget, Sync with Server, POLL OBJECTS and RESULT CALLBACK and usages of each pattern are mentioned. In the rest of paper we evaluated and compared the asynchronous invocation frameworks: ARMI, ZDun and EDAMI. ZDun framework supports the four asynchronous patterns. An important drawback of this framework is its complexity leading to difficulty in the framework's usage. The programmer should decide that which pattern is suitable in the specific situation, then uses the appropriate class for creating the asynchronous call. For further work it is recommended to purpose a framework which recognizes the optimum pattern automatically from program code and create the call.

References

[1] H.Adams, *Asynchronous operations and Web services*, IBM, Jun 2002
[2] C. Alexander (1979) *The Timeless Way of Buildin*g. Oxford Univ. Press.
[3] Apache Software Foundation. Web services invocation framework (WSIF). http:// ws.apache.org/wsif/, 2002.
[4] *Apache Software Foundation. Apache axis*. http://ws.apache.org/axis/, 2003.
[5] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer. *Simple object access protocol (SOAP) 1.1*. http://www.w3.org/TR/SOAP/, 2000.
[6] T. Bray, J. Paoli, and C. Sperberg-McQueen. *Extensible markup language (XML) 1.0*. http://www.w3.org/TR/1998/REC-xml-19980210, 1998.
[7] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. *Web services description language (WSDL) 1.1*. http://www.w3.org/TR/wsdl, 2001.
[8] R.Clarok, *Exploring the Dark Side of SOAs*, ftponline.com, WebLogicPro, Dec 2004
[9] Soheil Toodeh Fallah, Ehsan Zaeri Moghaddam, and Saeed Parsa, *"An Event-Driven Pattern for Asynchronous Invocations In Distributed Systems"* , International Journal of Computer Science and Network Security, VOL.7 No.4, April 2007
[10] Rajeev R.Raje, Joseph I.William, Michael Boyles , *"An Asynchronous Method Invocation (ARMI) Mechanism For Java"*, 2003
[11] S.Srinath, A.Ranabahu, *Axis2-Future of Web Services*, Jax Magazine, Jun 2005
[12] M. Voelter, M. Kircher, and U. Zdun. *Object-oriented remoting: A pattern language.* In Proceeding of The First Nordic Conference on Pattern Languages of Programs (VikingPLoP 2002), Denmark, Sep 2002.
[13] Markus Voelter, Michael Kircher, Uwe Zdun, Michael Englbrecht , *"Patterns For Asynchronous Invocations In Distributed Object Framework"* , In proceedings of EuroPlop, Germany , 2003
[14] ZDun. Uwe, et al, *"Pattern-Based Design of an Asynchronous Invocation Framework from Web Services"*, International Journal of Web Service Research, Volume 1, No. 3, 2004