Design and Implementation of an Asynchronous Invocation Framework for Web Services

Uwe Zdun[1], Markus Voelter[2], and Michael Kircher[3]

[1] Department of Information Systems, Vienna University of Economics, Austria
zdun@acm.org
[2] voelter – Ingenieurbro für Softwaretechnologie, Germany
voelter@acm.org
[3] Siemems AG, Corporate Technology, Software and System Architectures, Germany
michael.kircher@siemens.com

Abstract Asynchronous invocations are an important functionality in the context of distributed object frameworks, because in many situations clients should not block 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 we build a framework using patterns for asynchronous invocation of web services. The framework design is based on the asynchrony patterns and other patterns from the same pattern language.

1 Introduction

In this paper we discuss the problem of asynchronous invocation of web services. 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 [7] 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 transfered in XML encoded formats, such as SOAP [4] and WSDL [6].
- 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.).

Advantages of this approach to invoke REMOTE OBJECTS [16] 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 [5] (and XML-RPC [18] 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, are quite complex and thus not very easy to comprehend.

Liabilities of the approach are that the functionality of current web service frameworks is relatively limited compared to other standard middleware. The string-based, human-readable transport formats are bloated compared to more condensed (binary) transport formats. This results in larger messages and a more extensive use of network bandwidth. Also more processing power is consumed because XML consists of (human-readable) strings for identifiers, attributes, and data elements. String parsing is more expensive in terms of processing power than parsing binary data. The HTTP protocol may also cause some overheads because it is not as optimized for distributed object communication as protocols specifically designed for this task.

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. 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. Note that various messaging protocols are integrated with web services, such as the use of Java Messaging Service (JMS) in Axis and WSIF [2], JAXM, or Reliable HTTP (HTTPR) [10]. These protocols provide asynchrony at the transport protocol level. They are more sophisticated than simple asynchronous invocations (e.g. they support reliability of message transfers as well) and use a different communication paradigm than synchronous transport protocols. Under high volume conditions, messaging might incur problems such as a bursty and unpredictable message flow. Messages can be produced far faster than they can be consumed, causing congestion. This condition requires the messages to be throttled with flow control. In this paper, we do not directly deal with messaging protocols, even tough it is possible to use a messaging protocol in the lower layers of our framework design.

Hard-coding different styles of asynchronous invocation into a client application by hand for each use is tedious, error-prone, and results in different styles of invocation. Instead one invocation model should be offered to the developer that supports all invocation variants with a simple and intuitive interface. In this paper, we present an asynchronous invocation framework for Apache Axis. Its design is based on a set of asynchrony patterns [17] to fulfill the specific client-side requirements for integrated asynchronous invocation in the web service context (on top of HTTP). The framework is designed to be easily adapted to other web service frameworks and/or other synchronous (or asynchronous) transport protocols.

The paper is structured as follows: First we give an overview of the goals of an asynchronous invocation framework in the context of web services. Next we present the asynchrony patterns from [17] briefly. Then we discuss the design of an asynchronous invocation framework for Apache Axis and compare its performance with synchronous invocations. Finally, we present some related work and conclude.

2 Goals of an Asynchronous Invocation Framework in the Context of Web Services

There are a number of issues about web services because of the limitation to synchronous invocations only. To avoid hard coding asynchronous invocations in the client code, we provide an object-oriented framework [11] to offer a flexible and reusable software implementation. In particular our framework aims at the following issues:

- *Better Performance of Client Applications*: Asynchronous invocations can lead to better performance of the client application, as we can avoid idle times waiting for a blocking invocation to return. This is specifically important because handling of XML encoding and HTTP is not the fastest variant of remote invocation.
- *Simple and Flexible Invocation Model*: A simple invocation model should be offered to client developers. Asynchronous invocation should not be more complicated to use than synchronous invocation. That is, the developer should not have to deal with issues such as multi-threading, synchronization, or thread pooling.
- *Support for multiple Web Services Implementations and Protocols*: The strength of web services is heterogeneity, thus an asynchronous invocation framework should (potentially) work with different protocols (such as JMS or Secure HTTP) and implementations. If the invocation framework can be built on top of an existing web service framework (that already integrates different protocols), then they are automatically integrated in the invocation framework as well.
- *Avoiding the Use of Messaging Protocols*: Messaging protocols such as JMS or HTTPR can provide asynchrony on the protocol level. To provide for heterogeneity, web services should not depend on a special protocol such as JMS, but all required functionality should be provided for all supported protocols. For instance, if asynchrony is required and HTTP should be used for firewall tunneling, then asynchrony should be provided for HTTP natively.
- *Client as a Reactive Application*: Some clients are reactive applications, such as GUI applications or servers that are clients to other servers. In such reactive clients a blocking invocation is not possible because that would mean to block the reactive event handling as well. A blocking server or GUI is usually not acceptable.

3 Client Asynchrony Patterns

In this section, we present a set of client asynchrony patterns [17] that are part of a larger pattern language for distributed object communication[1] (see also [16]).

[1] The complete pattern language will be published in a book entitled "Remoting Patterns" in Wiley's pattern series in 2004.

A pattern[2] 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 [1]. 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 [1].

The client asynchrony patterns are in particular:

- FIRE AND FORGET: 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. When invoked, the CLIENT PROXY sends the invocation across the network, returning control to the caller immediately. The client does not get any acknowledgment from the REMOTE OBJECT receiving the invocation.
- 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 response arrives back. Sometimes the middle of both extremes is needed. 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.
- POLL OBJECT: There are situations, when an application needs to invoke an operation asynchronously, but still requires to know 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. As a solution POLL OBJECTS 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.
- 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 immediatly to react on it. In the meantime the client executes concurrently. A callback-based interface for remote invocations is provided on the client. Upon an invocation, the client passes a RESULT 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.

[2] We present pattern names in SMALLCAPS font.

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.

Client Asynchrony Pattern	*Result to client*	*Acknowledgment to client*	*Responsiblity for result*
FIRE AND FORGET	no	no	-
SYNC WITH SERVER	no	yes	-
POLL OBJECT	yes	yes	Client is responsible for getting the result
RESULT CALLBACK	yes	yes	Client is informed via a callback

Table 1. Alternatives for applying the patterns

4 Design and Implementation of an Asynchronous Invocation Framework for Apache Axis

In this section, we explain a framework design to implement the client-side asynchrony patterns, explained in the previous section, in a generic and efficient way for a given web service implementations. We use the popular Apache Axis framework for our implementation in Java, though the general framework design can also be used with other web service implementations.

4.1 Client Proxies

Our general design relies on the CLIENT PROXY pattern [16]. 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 [16] (such as WSDL). In our description, we first concentrate on CLIENT PROXIES that build up a remote invocation at runtime. We also discuss how to use the stubs that are automatically generated from WSDL in an asynchronous CLIENT PROXY in Section 4.6.

In our framework, we provide two kinds of CLIENT PROXIES, 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. Thus it is just a wrapper to the ordinary CLIENT PROXY of the Axis framework for convenience. 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 we pass an `AsyncHandler` and `clientACT` as arguments and do not wait for a result (`AsyncHandler` and client invocation handlers are described in the next sections in detail):

```
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) [15]. The ACT pattern is used to let clients identify different results of asynchronous invocations. In contrast to the `clientACT` field, the ACT (in the description in [15]) is passed across the network to the server, and the server returns it to the client together with the result. 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.

4.2 Client Invocation Handlers

In the case of a synchronous invocation, invocation dispatching and subsequent invocation handling do not need to be decoupled. This is because the invoking process (or thread) blocks until the invocation is completely handled. In contrast, asynchrony means that multiple invocations are handled in parallel, and the invoking thread can continue with its work while an invocation is handled. Therefore, invocation dispatching and invocation handling should be decoupled.

Synchronous and asynchronous invocation handling is performed by different kinds of invocation handlers. These, however, require the same information about the invocation, such as endpoint URL and operation name as web service IDs, an argument list, and a return type. Also constructing a `Call` from these information is common for all different kinds of invocation handlers (see Figure 1).

The synchronous invocation handler mainly provides a method `invoke` that synchronously invokes the service constructed with `constructCall`. The invocation returns when the response arrives.

The asynchronous invocation handler (`AsyncInvocationHandler`) implements the `Runnable` interface. This interface indicates that the handler implements a variant

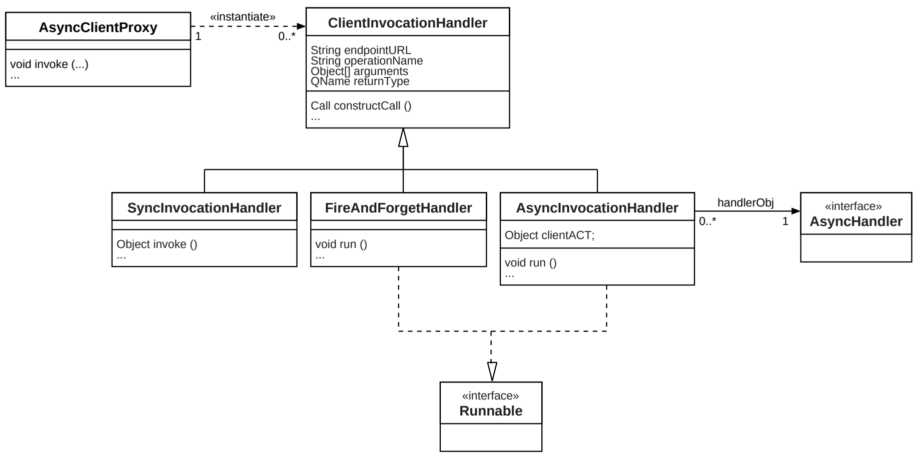

Figure 1. Invocation Handlers

of the COMMAND pattern [8] that can be invoked in the handler's thread of control using a method `run`. The class `AsyncInvocationHandler` associates a handler object to hand the result back to the client thread. It also contains a `clientACT` field that stores the ASYNCHRONOUS INVOCATION TOKEN supplied by the client. Usually, the field is used identify the invocation later in time, when the response has arrived.

The `AsyncInvocationHandler` decides on basis of the kind of handler object which asynchrony pattern should be used, RESULT CALLBACK, POLL OBJECT, or SYNC WITH SERVER (see Section 4.4). The decision is done using Java's `instanceof` primitive.

Finally, FIRE AND FORGET is implemented in its own invocation handler class (see next Section).

4.3 Fire and Forget Invocations

The FIRE AND FORGET pattern is not implemented in the class `AsyncInvocationHandler` (or as a subclass of it) due to a specialty of web services: the WSDL standard [6] that is used for interface description of web services supports so-called one-way operations. These are thus implemented by most web service frameworks that support WSDL. Therefore, we do not implement FIRE AND FORGET with the `AsyncInvocationHandler` class, but use the one-way invocations to support FIRE AND FORGET operations. All invocations dispatched by the `AsyncInvocationHandler` class are request-response invocations.

A FIRE AND FORGET invocation executes in its own thread of control. The FIRE AND FORGET invocation simply constructs the `Call`, performs the invocation, and then the thread terminates.

A FIRE AND FORGET invocation is invoked by a special `invokeFireAndForget` method of the `AsyncClientProxy` class:

```
AsyncClientProxy clientProxy = new AsyncClientProxy();
clientProxy.invokeFireAndForget(endpointURL, operationName,
                                null, rt);
```

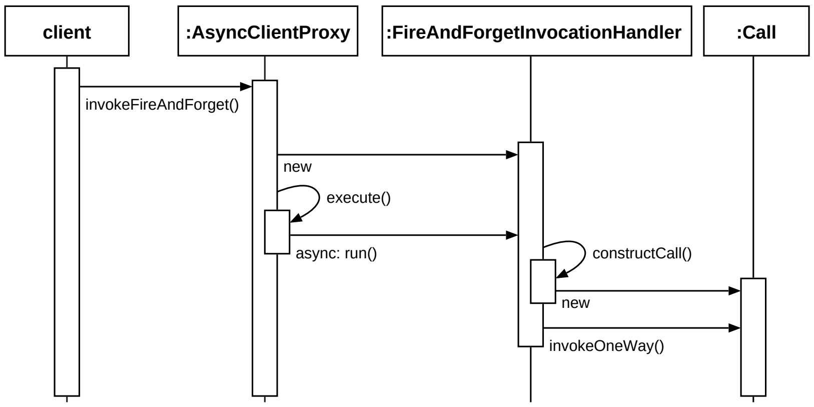

Figure 2. Fire And Forget Dynamics

Figure 2 shows the dynamic invocation behavior of a FIRE AND FORGET invocation.

4.4 Asynchrony Pattern Handlers

To deal with the asynchrony patterns RESULT CALLBACK, POLL OBJECT, or SYNC WITH SERVER the client asynchrony handler types `ResultCallback`, `PollObject`, and `SyncWithServer` are provided. These are instantiated by the client and handed over to the CLIENT PROXY (for instance, in the `invoke` method).

The asynchronous CLIENT PROXY handles the invocation with an `AsyncInvocationHandler`. Each invocation handler runs in its own thread of control and deals with one invocation. A thread pool is used to improve performance and reduce resource consumption (see Section 5.1). The client asynchrony handlers are sinks that are responsible for holding or handling the result for clients.

For an asynchronous invocation, the client simply has to instantiate the required client asynchrony handler (a class implementing one of the following interfaces: `ResultCallback`, `PollObject`, or `SyncWithServer`) and provide it to the CLIENT PROXY's operation `invoke`. This operation is defined as follows:

```
public void invoke(AsyncHandler handler, Object clientACT,
                   String endpointURL, String operationName,
                   Object[] arguments, QName returnType)
  throws InterruptedException {...}
```

The parameter `handler` determines the responsible handler object and type. It can be of any subtype of `AsyncHandler`. `clientACT` is a user-defined identifier for the invocation. The client can use the `clientACT` to correlate a specific result to an invocation. The four last arguments specify the service ID, operation name, and invocation data.

For instance, the client might invoke a POLL OBJECT by first instantiating a corresponding handler and then providing this handler to `invoke`. Subsequently, it polls the POLL OBJECT for the result and works on some other tasks until the result arrives:

```
AsyncClientProxy clientProxy = new AsyncClientProxy();
SimplePollObject p = new SimplePollObject();
clientProxy.invoke(p, null, endpointURL, operationName,
                   null, rt);

while (!p.resultArrived()) {
  // do some other task ...
}
System.out.println("Poll Object Result Arrived = " +
                   p.getResult());
```

Note that the `clientACT` parameter is set to `null` in this example because we can use the object reference in `p` to obtain the correct POLL OBJECT.

The pre-defined client asynchrony handlers and interfaces are depicted in Figure 3.

The client asynchrony handlers that are informed of the results run in the invoking thread. To enable synchronization of the access from different threads (and clients) we apply the MONITOR OBJECT pattern [15], which is supported by Java's `synchronized` language construct. The operations of each client asynchrony handler are synchronized and the access is scheduled.

Figure 4 shows the dynamic invocation behavior of a POLL OBJECT invocation. The dynamics of handling a RESULT CALLBACK are identical, with the exception that a RESULT CALLBACK asynchrony handler is passed to the CLIENT PROXY, and the client does not poll it. A SYNC WITH SERVER uses the SYNC WITH SERVER asynchrony handler and does not obtain the result, but only an acknowledgment.

4.5 Queued Asynchrony Handlers

Sometimes we want to use one instance to handle multiple responses. A simple implementation of such behavior is an asynchrony handler that queues the arriving responses. Such queuing handlers with FIFO (first-in,first-out) behavior are pre-defined in our framework for RESULT CALLBACK, POLL OBJECT, and SYNC WITH SERVER (as depicted in Figure 3).

In the queuing variant the client cannot use the handler object reference to identify the invocation that belongs to the result. Thus generally the `clientACT` field should be used to identify the invocation that belongs to an asynchrony handler. The `clientACT` field is also important for clients, if they need to customize the handler objects. For instance, if a RESULT CALLBACK should forward the callback to an operation of the client

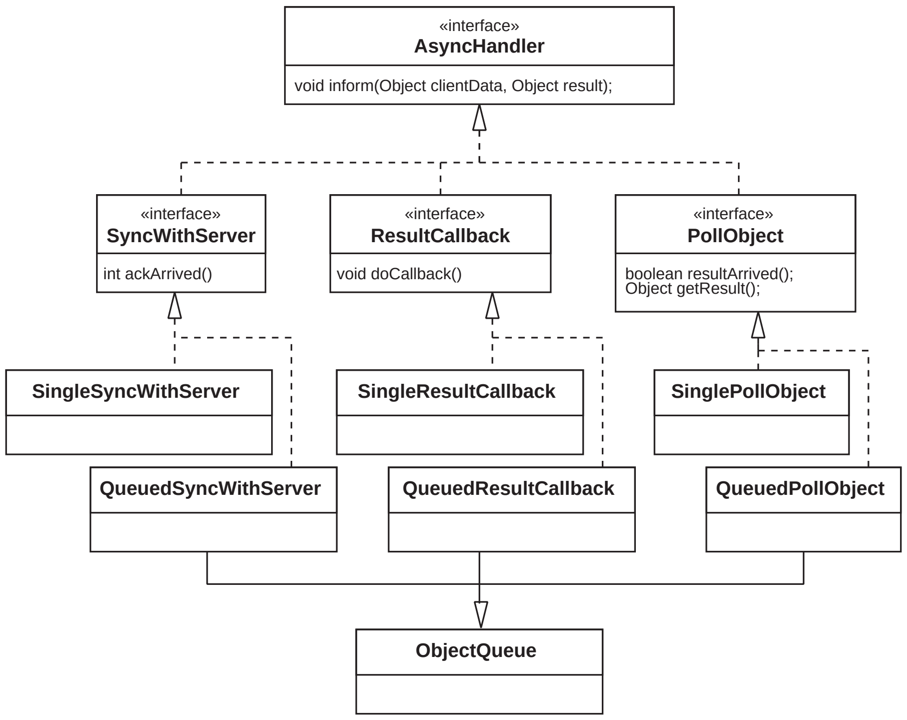

Figure 3. Handlers for Obtaining Asynchronous Results

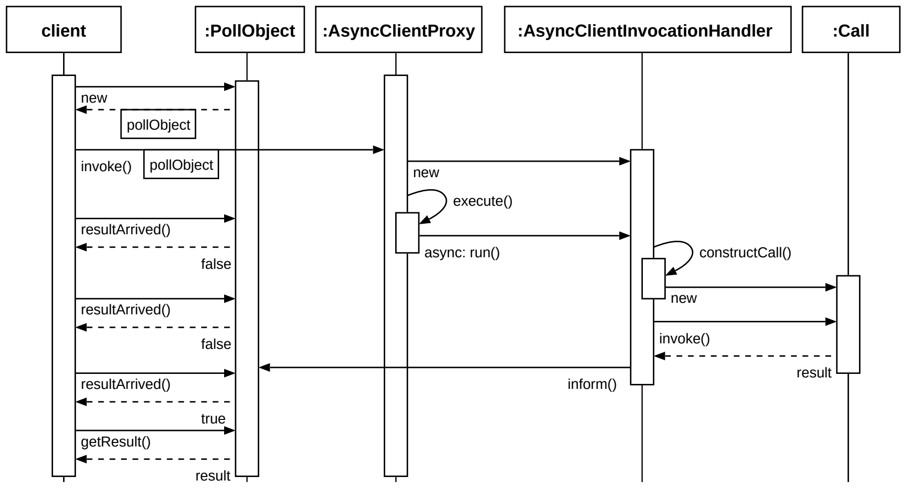

Figure 4. Poll Object Dynamics

object, a reference to the client object is needed. This reference can be passed as part of a client ACT structure, which is then used by the custom asynchrony handler to dispatch the callback to the client.

Consider a RESULT CALLBACK as a second example. A developer might define a RESULT CALLBACK class as an extension of the existing RESULT CALLBACK type `ResultCallbackQueue`:

```
class DateClientQueue extends ResultCallbackQueue {...};
```

Then the client can use this custom type to handle invocations. When we use a queue handler type, we usually want to handle more than one result with the same handler; thus we instantiate a number of invocations in different threads of control:

```
AsyncClientProxy clientProxy = new AsyncClientProxy();
DateClientQueue results = new DateClientQueue(10);
for (int i = 0; i < 10; i++) {
  String id = "callback" + i;
  clientProxy.invoke(results, id, endpointURL, operationName,
                     null, rt);
}
```

In this example the ten invocations are all reported to one queuing RESULT CALLBACK object. This object can either handle the result on its own (e.g. if the client is just a `main` method) or forward the callback to the client object that has invoked it. Of course, if the client is an object that implements the `ResultCallback` interface it can also be itself handed over as a RESULT CALLBACK object.

4.6 Using WSDL Generated Client Stubs in An Asynchronous Client Proxy

WSDL [6] is used as a standard INTERFACE DESCRIPTION [16] language in the context of web services. The main goal of using WSDL is to provide a language to interchange information about web services and transfer these to clients.

Axis provides two models of invocation and both can be used within our asynchronous invocation framework:

– The `Call` interface provided by Axis can be used to construct an invocation at runtime. This interface is used by the `constructCall` operation mentioned earlier.
– When using WSDL, Axis generates a stub class that already constructs the invocation using the `Call` interface. Thus, when this stub is provided by the client, the CLIENT PROXY in our asynchronous invocation framework can directly use the stub and does not need to invoke the `constructCall` operation.

5 Performance Considerations

Providing an asynchronous invocation framework provides a better performance regarding the invocation times because the client can resume its work after dispatching an invocation. Yet, compared to synchronous invocation dispatching, multi-threaded invocations also incur an invocation overhead due to instantiating the threads. This overhead can be minimized with thread pooling discussed in Section 5.1. Next, we compare the performance of asynchronous invocations to synchronous invocations in our framework.

5.1 Thread Pooling

To optimize resource allocation for threading, the threads can be shared in a pool using the POOLING pattern [12]. Clients can acquire the resources from the pool, and release them back into the pool, when they are no longer needed. To increase efficiency, the pool eagerly acquires a pre-defined number of resources after creation. If the demand exceeds the available resources in the pool, it lazily acquires more resources. POOLING thus reduces the overhead of instantiating and destroying threads.

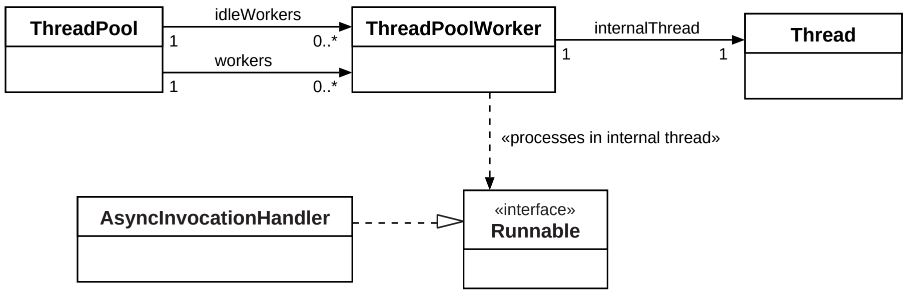

Figure 5. Thread Pooling

We use a generic thread pool with thread pool workers that require the client to provide COMMANDS [8] of the type `Runnable` (see Figure 5). The thread pool acquires a pre-defined number of thread pool workers in its idle workers list. Whenever a thread pool worker is required, it is obtained from the pre-instantiated worker pool, if possible. If there is no worker idle, the thread pool lazily instantiates more workers. After the work is done, the (pre-defined) workers are put back into the pool.

The asynchronous invocation handlers implement the `Runnable` interface and can thus be used with the thread pool. Thus each invocation handler runs in its own thread of control and is automatically pooled.

5.2 Performance Comparison

As a performance comparison we have used a simple web service that just returns the current date as a string.

For each variant we have tested 1, 3, 10, and 20 invocation in a row. The thread pool had a size of 10 pre-initialized workers. All results are measured in milliseconds. We have used the Sun JDK 1.4, Jakarta Tomcat 4.1.18, Xerces 2.3.0, and Axis 1.0. All measurements were performed on an Intel P4, 2.53 GHz, 1 GB RAM running Red Hat Linux. We have measured all performance tests 10 times and used the best results (the average results were quite close to the best results and therefore we omit them here).

The results are summarized in Table 2.

For synchronous invocations we have simply measured the time that all invocations took. We can see that the invocation times increase as the number of invocations increases.

For FIRE AND FORGET and SYNC WITH SERVER we have measured the time until the requests were sent. We can see that the times are much shorter than the synchronous invocations, as expected. Only the 20 invocations case is 2-3ms slower than it could be expected when a linear progression would be assumed. This overhead is approximately the time needed to instantiate 10 thread pool workers.

For POLL OBJECT and RESULT CALLBACK we have measured the times until the invocations are dispatched and the invoking thread can resume its work. These numbers are more or less equal to the times of FIRE AND FORGET and SYNC WITH SERVER. Also we have measured the times until the last response has arrived. We can see that these numbers are similar to the synchronous invocation times yet there is a slight overhead.

Performance Test	Synchronous Invocation	FIRE AND FORGET	SYNC WITH SERVER	POLL OBJECT	RESULT CALLBACK
1 invocation	30ms	1ms	1ms	1ms/39ms	1ms/42ms
3 invocation	68ms	2ms	2ms	2ms/89ms	2ms/69ms
10 invocation	204ms	2ms	2ms	2ms/265ms	2ms/189ms
20 invocation	378ms	5ms	4ms	5ms/409ms	4ms/368ms

Table 2. Performance Comparison

6 Related Work: Other Known Uses of the Patterns

In this section we summarize some known uses of the asynchrony patterns as related work.

There are various messaging protocols that are used to provide asynchrony for web services on the protocol level, including JAXM, JMS, and Reliable HTTP (HTTPR) [10]. In contrast to our approach these messaging protocols do not provide a protocol-independent interface to client-side asynchrony and require developers to use the messaging communication paradigm. Yet these protocol provide a reliable transfer of messages, something that our approach does not deal with. Messaging protocols can be used in the lower layers of our framework.

The Web Services Invocation Framework (WSIF) [2] is a simple Java API for invoking Web services with different protocols and frameworks (similar to the internal invocation API of Axis). It provides an abstraction to circumvent the differences in protocols used for communications, similar to our invocation framework. However, it deals with asynchrony using messaging protocols (HTTPR, JMS, IBM MQSeries Messaging, MS Messaging) only. The approach presented in this paper can also be used on top of with WSIF.

For a long time CORBA [9] supported only synchronous communication and unreliable one-ways operations, which were not really an alternative due to the lack of reliability and potential blocking behavior. Since the CORBA Messaging specification appeared, CORBA supports reliable one-ways. With various policies the one-ways can be made more reliable so that the patterns FIRE AND FORGET as well as SYNC WITH SERVER, offering more reliability, are supported. The RESULT CALLBACK and POLL OBJECT patterns are supported by the Asynchronous Method Invocations (AMI) with their callback and polling model, also defined in the CORBA Messaging specification.

.NET [13] provides an API for asynchronous remote communication. Similar to our approach, client asynchrony does not affect the server side. All the asynchrony is handled by executing code in a separate thread on the client side. POLL OBJECTS are supported by the `IAsyncResult` interface. One can either ask whether the result is already available or block on the POLL OBJECT. RESULT CALLBACKS are also implemented with this interface. An invocation has to provide a reference to a callback operation. .NET uses one-way operations to implement FIRE AND FORGET. SYNC WITH SERVER is not provided out-of-box, but it can be implemented with a similar approach as used in this paper.

Actiweb [14] is a web object system implemented in Tcl. It provides sink objects for all kinds of blocking and non-blocking communication. A client can register a callback for the sink (to implement RESULT CALLBACKS), block on the sink, or use the sink as a POLL OBJECT. FIRE AND FORGET can be implemented by using sink with an empty RESULT CALLBACK. Similarly, SYNC WITH SERVER can be implemented by a RESULT CALLBACK that raises an error if a timeout exceeds and does nothing if the server responds correctly.

7 Conclusion

In this paper we have provided a practical approach to provide asynchronous invocations for web services without using asynchronous messaging protocols. The framework was designed with a set of patterns from a larger pattern language for distributed object frameworks. The functionalities as well as the performance measurements indicate that the goals of the framework (as introduced in Section 2) were reached; in particular:

- A client can significantly faster resume with its work so that the performance penalty of web services can be avoided to a certain degree.
- The invocation API provided by the framework is very simple and can flexibly be extended with custom handlers.
- As the framework is built on top of Axis we automatically can use its heterogeneity regarding transport protocols and back-ends of web services (so-called “service providers”).
- If the client is a reactive server applications, a remote invocation does not block it.

As a drawback, an asynchrony framework on top of a synchronous invocation framework always incurs some overhead in terms of the overall performance of the client application. Further functionalities of messaging protocols, for instance, are not supported. But as messaging protocols can be used internally this is not a severe drawback. Our framework does not introduce any security functionalities (yet) at the invocation layer, and can thus only use security functionalities implemented at lower layers, say, at the transport protocol layer.

References

1. C. Alexander. *The Timeless Way of Building*. Oxford Univ. Press, 1979.
2. Apache Software Foundation. Web services invocation framework (WSIF). http://ws.apache.org/wsif/, 2002.
3. Apache Software Foundation. Apache axis. http://ws.apache.org/axis/, 2003.
4. 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.
5. T. Bray, J. Paoli, and C. Sperberg-McQueen. Extensible markup language (XML) 1.0. http://www.w3.org/TR/1998/REC-xml-19980210, 1998.
6. E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (WSDL) 1.1. http://www.w3.org/TR/wsdl, 2001.
7. R. Fielding, J. Gettys, J. Mogul, H. Frysyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. RFC 2616, 1999.
8. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
9. O. M. Group. Common request broker architecture (corba). http://www.omg.org/corba, 2000.
10. IBM developerWorks. Httpr specification. http://www-106.ibm.com/developerworks/webservices/library/ws-httprspec/, 2002.
11. R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June/July 1988.
12. M. Kircher and P. Jain. Pooling pattern. In *Proceedings of EuroPlop 2002*, Irsee, Germany, July 2002.
13. Mircrosoft. .NET framework. http:///msdn.microsoft.com//netframework, 2003.
14. G. Neumann and U. Zdun. Distributed web application development with active web objects. In *Proceedings of The 2nd International Conference on Internet Computing (IC'2001)*, Las Vegas, Nevada, USA, June 2001.
15. D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Patterns for Concurrent and Distributed Objects*. Pattern-Oriented Software Architecture. J. Wiley and Sons Ltd., 2000.
16. 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. http://wi.wu-wien.ac.at/∼uzdun/publications/vikingPlop02.pdf.
17. M. Voelter, M. Kircher, and U. Zdun. Patterns for asynchronous invocations in distributed object frameworks. submitted, a draft can be found at http://wi.wu-wien.ac.at/∼uzdun/publications/AsynchronyDraft.pdf, 2003.
18. D. Winer. XML-RPC specification. http://www.xmlrpc.com/spec, 1999.