Pattern-Based Design of an Asynchronous Invocation Framework for Web Services

Uwe Zdun
Department of Information Systems, Vienna University of Economics, Austria
E-mail: zdun@acm.org

Markus Voelter
voelter - Ingenieurbüro für Softwaretechnologie, Germany
E-mail: voelter@acm.org

Michael Kircher
Siemems AG, Corporate Technology, Software and System Architectures, Germany
E-mail: michael.kircher@siemens.com

Abstract: Asynchronous invocations are needed in the context of distributed object frameworks to prevent clients from blocking during remote invocations. Popular Web Service frameworks 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. We present a number of patterns for asynchronous invocations and explain how these patterns can be used to build asynchronous invocation facilities for Web Service frameworks. We exemplify this approach by explaining the design and implementation of an asynchronous invocation framework for Apache Axis.

Keywords: Web Services, remote objects, asynchronous invocations, patterns.

Biographical notes:

Uwe Zdun is assistant professor in the department of information system at the Vienna University of Economics and Business Administration. He received his PhD from the University of Essen in 2002, where he has also worked as research assistant from 1999 to 2002. His research interests include software patterns, scripting, object-orientation, software architecture, and web engineering. He is (co-)author of the object-oriented scripting language Extended Object Tcl (XOTcl), the web object system ActiWeb, and many other software systems.

Markus Völter works as an independent consultant for software technology and engineering. He focuses on the architecture of large, distributed systems, as well as model-driven software development. Markus is the author of several magazine articles and patterns, a regular speaker at conferences and co-author of Wiley's "Server Component Patterns" book. He can be reached at voelter@acm.org or www.voelter.de.

Michael Kircher is currently working as Senior Software Engineer at Siemens AG Corporate Technology in Munich, Germany. His main fields of interest include distributed

object computing, software architectures, design patterns, Extreme Programming, and management of knowledge workers in innovative environments. During the last years he published at numerous conferences, like OOPSLA, EuroPLoP, PLoP, Middleware, XP, on topics such as patterns, open-source, software architectures for distributed systems, and extreme programming.

INTRODUCTION

This paper discusses asynchronous invocations in the context of Web Services. Web Services provide a standardized means of service-based, language indepdendent and platform independent interoperation between different, distributed software applications. The use of Web Services on the World Wide Web is expanding rapidly as the need for application integration and interoperability grows [5].

Although there are many different kinds of distributed object frameworks that refer to the term Web Services, a Web Service can be described by a set of technical characteristics, including:

- The HTTP protocol family [9] is used as the basic communication protocol.
- Data, invocations, and results are transferred in XML encoded formats, such as SOAP [6] and WSDL [8].
- Remotely offered services are invoked with a simple, stateless request/response scheme, and thus Web Services are more often message-oriented than they are RPC-oriented.
- Many Web Service frameworks are not limited to HTTP as transport protocol.
- The services are often implemented with different back-end providers (for instance, a Java class, an EJB component, a legacy system, etc.) and a model for integration of these back-ends is provided by the Web Service framework.

Advantages of this approach to invoke remote objects [21], or other kinds of service implementations such as procedures, are that Web Services provide a means for interoperability in a heterogeneous environment. Basing the information exchange only on stateless message exchanges leads to a loose coupling of clients and servers. Web Services are also relatively easy to use and understand due to simple APIs, and XML content is human-readable. The goals of Web Services go beyond those of classical middleware frameworks, such as CORBA, DCOM, or RMI: they aim at standardized support for higher-level tasks such as service and process flow orchestration, enterprise application integration (EAI), and providing a "middleware for middleware" [20].

Current Web Service frameworks have some liabilities associated. In the spirit of the original design ideas of XML [7] (and XML-RPC [24] as the predecessor of today's standard Web Service message format SOAP), XML encoding was expected to enable simplicity and readability 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 read and understand (by humans). In many cases, stateless communication as imposed by HTTP and SOAP causes some overheads because it may result in repeated transmission of the same data (for instance, for authentication or identifying the current session). Cai et al. provide detailed benchmarks comparing different encoding mechanism for Web Services [4]. This study leads to the following results: XML as a (string-based) transport format is bloated compared to more condensed (binary) transport formats. This results in larger messages, as well as a more extensive use of network bandwidth. This problem can be avoided by compressing XML data, but compression leads to an additional performance overhead. XML consists of strings for identifiers, attributes, and data elements. String parsing is more expensive in terms of processing power than parsing binary data.

Many Web Service frameworks, such as Apache Axis [3], only allow for synchronous invocations (for synchronous protocols such as HTTP). That is, the client process (or thread) blocks until the response arrives. For client application 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 is, 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, a client should not depend on the response time of a Web Service. As most Web Service frameworks are not designed for asynchronous communication we need to provide the asynchronous behaviour on top of the synchronous invocation layer.

Note that there are various efforts to integrate messaging protocols in Web Services, such as the use of Java Messaging Service (JMS) in Axis and WSIF [2], JAXM, or Reliable HTTP (HTTPR) [13]. These protocols provide asynchrony on the protocol level, whereas the approach proposed in this paper provides asynchronous invocations at the invocation layer or on top of an existing distributed object framework. The messaging approaches are more sophisticated than simple asynchronous invocations. For instance they support reliability of message transfers as well. But they use a different communication paradigm than synchronous 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. Such issues require the messages to be throttled by flow control. In this paper, we do not deal with messaging protocols, even tough it is possible to use a messaging protocol in the lower layers of our framework design. Yet the synchronous programming model would stay and clients cannot take advantage of concurrent execution – or, alternatively, client developers need to be aware of the asynchronous protocol and use a different invocation style for asynchronous invocations.

There are many different styles of asynchronous invocations. For instance, the client might not be interested in the invocation result, or it might be informed via a callback, or it might actively obtain the result when it has finished some subsequent tasks. Hard-coding different styles of asynchronous invocation into a client application by hand for each use is tedious and error-prone. Instead, the invocation model should be offered to the developer that supports all invocation variants with a simple and intuitive interface.

In this paper, we present patterns for implementing asynchronous invocations [22]. These are part of a larger pattern language for building distributed object frameworks [21,23]. To provide the context, we also describe the asynchrony and concurrency patterns from POSA2 [19] on which the implementations of these patterns rely. We apply the patterns to an asynchronous invocation framework for Apache Axis[1]. Using the patterns, different variants of synchronous and asynchronous invocations can be chosen to fulfill the specific client-side requirements for synchronous or asynchronous invocation in the Web Service context (here: on top of HTTP). The framework is designed to be easily adapted to other Web Service frameworks and/or other synchronous or asynchronous communication 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 introduce the aforementioned patterns from [22,21,23,19] 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.

GOALS OF AN ASYNCHRONOUS INVOCATION FRAMEWORK IN THE CONTEXT OF WEB SERVICES

[1] The framework, described in this paper, can freely be downloaded from: `http://saiws.sourceforge.net`

There are a number of issues about Web Services because of the limitation to synchronous invocations only. To avoid the work-around of hard coding asynchronous invocations in the client code, we provide an object-oriented framework [14] that can be reused as an extension for existing Web Service frameworks. The framework design is based on a number of software design patterns. Before we explain these patterns in the next section, let us summarize the goals of our asynchronous invocation framework:

- *Better Performance of Client Applications:* Asynchronous invocations typically lead to better performance of client applications, as idle times waiting for a blocking invocation to return are avoided.
- *Simple and Flexible Invocation Model:* The invocation model must be simple to use by developers. Asynchronous invocations should not be more complicated to use than synchronous invocations. That is, the client developer should not have to deal with issues such as multi-threading, synchronization, or thread pooling. There are different kinds of invocations, including synchronous invocations and various ways to provide asynchronous invocations. All these kinds of invocation should be offered with an integrated invocation model that is easy to comprehend.
- *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. An invocation framework that builds on top of an existing Web Service framework automatically integrates the different protocols provided by that Web Service framework.
- *Avoiding the Use of Messaging Protocols:* Messaging protocols such as JMS or HTTPR can provide asynchrony on the protocol level. But they use a different communication paradigm than synchronous invocations and may cause problems such as a bursty message flow or congestion of the message consumer. 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, then asynchrony should be provided for HTTP natively.
- *Client as a Reactive Application:* Some clients are reactive applications, such as GUI applications or server applications that are clients to another 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.

Asynchronous invocation support can be found in many distributed object frameworks. However, when designing an asynchronous invocation framework for a given synchronous Web Service infrastructure, we have to deal with the particularities of that infrastructure. In the case of Web Services this means in particular that we can rely on the operation types defined by WSDL [8] (synchronous handling of In-Out, In-only, Out-only, and Out-In operations), as these are provided by most Web Service infrastructures. A framework design should not limit the heterogeneity of services, infrastructures, and protocols, as this is one of the main goals of the Web Services approach. As a minimum requirement, the HTTP protocol family should be supported with its typical properties such as stateless interaction. Web Service frameworks usually provide invocation handling with requesters [21] or client proxies [21] that are automatically generated from WSDL interface descriptions [21], as well as runtime construction of invocations. An asynchronous invocation framework should support both invocation schemes.

PATTERNS FOR ASYNCHRONOUS INVOCATIONS

In this section, we present a number of patterns for asynchronous invocations. These patterns are part of a larger pattern language for distributed object communication (see [22,23,21]). We also present some patterns from [19] that are typically used for implementing asynchronous and concurrent systems.

A pattern[2] is a proven 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].

Invocation Asynchrony Patterns

There are four patterns that represent basic alternatives for implementing asynchronous invocations. All four patterns are alternatives to synchronous invocations. We use two other patterns from [23] to explain the asynchronous invocation patterns: REQUESTER and REMOTE OBJECT.

A REMOTE OBJECT is a distributed object (here: the Web Service), offered by a server application, that should be reached by the client remotely. Note that a REMOTE OBJECT describes the remote interface, not the actual implementation – thus the service implementation can well comprise a set of procedures or be a wrapper to a legacy system.

The pattern REQUESTER describes how to build up a remote invocation on client side and hand it over to the transport layer of the distributed object framework. Note that clients often do not access the REQUESTER implementation directly, but use a (generated) CLIENT PROXY instead. The CLIENT PROXY offers the interface of the REMOTE OBJECT in the client process and uses the REQUESTER internally to build up the remote invocation.

The four patterns for asynchronous invocations are in particular:

- FIRE AND FORGET: In many situations, a client application needs to invoke an operation of 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, for instance, might be resent in regular intervals. When invoked, the REQUESTER 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 invocation where a client is blocked until the remote operation 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 REQUESTER, 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 of a REMOTE OBJECT. That is, if the result becomes available to the REQUESTER, the client wants to be informed immediately to react on it. In the meantime the client executes concurrently. A callback-based interface for remote invocations is provided within the client. Upon an invocation, the client passes a RESULT CALLBACK object to the REQUESTER. The invocation returns immediately after sending the invocation to the server. Once the result is available, the REQUESTER calls a predefined operation on the callback object, passing it the result of the invocation.

In [23] we present another pattern, MESSAGE QUEUE, that is often used together with the asynchronous invocation

[2] We present pattern names in SMALLCAPS font.

alternatives, introduced above. A MESSAGE QUEUE deals with (temporal) problems of the networked environment, such as network latency, network unreliability, or server crashes. It can be added between the network and invocation layers, both on client and server side. When the client sends a request, it is not transmitted immediately, but first put into a MESSAGE QUEUE. The request is kept in this queue until it can be reliably transmitted to the server side, and an acknowledgement is received. When a new request arrives at the server side, it is added to a MESSAGE QUEUE as well, and an acknowledgement is sent back to the client. Either immediately, some time later, the server process consumes the message and invokes the REMOTE OBJECT. The result has to be transmitted back to the client. The reply is sent back using the same scheme. First, it is put into a MESSAGE QUEUE on server side, then it is transmitted and put into the client's MESSAGE QUEUE, and eventually it is consumed by the client process.

<<Figure 1 to be inserted here>>

Invoking operations asynchronously, using one of the patterns FIRE AND FORGET, SYNC WITH SERVER, POLL OBJECT, or RESULT CALLBACK, allows to avoid blocking clients. However, using these patterns, the client has to care for dealing with problems of (temporal) unreliability of a networked environment. That is, if the server cannot be reached, the client needs to deal with the resulting REMOTING ERROR, for instance, by resending the invocation or raising an exception. MESSAGE QUEUES, in contrast, can be used to decouple and temporarily store multiple requests or replies handled either synchronously or with one of the patterns POLL OBJECT or RESULT CALLBACK.

MESSAGE QUEUE is the basic pattern for implementing messaging systems. Today's messaging systems and protocols, however, deal with multiple other issues than implemented by simple MESSAGE QUEUES, such as guranteed delivery, message channels, or message expirations. See [12] for patterns describing these aspects of messaging systems.

Figure 1 presents an overview of the invocation asynchrony patterns. Table 1 illustrates the alternatives for applying the patterns. It distinguishes whether there is a result sent to the client or not, and whether the client gets an acknowledgment or not. If there is a result sent to the client, it may be the client's burden to obtain the result or it is informed using a callback.

<<Table 1 to be inserted here>>

Asynchrony and Concurrency Patterns

There a number of other synchronization, asynchrony, and concurrency patterns, described in [19], that are important to be understood in order to implement the invocation asynchrony patterns, explained above. We describe those patterns briefly in this section.

When accessed asynchronously, a REMOTE OBJECT is very similar to an ACTIVE OBJECT [19]. An ACTIVE OBJECT decouples method invocation from method execution. The same holds true for REMOTE OBJECTS that use one of the above presented patterns for asynchronous invocations. However, when REMOTE OBJECTS are invoked synchronously the invocation and execution of a method is not decoupled, even though they run in separate threads of control.

ACTIVE OBJECTS typically create *Future* objects that clients use to retrieve the result from the method execution. The implementation of such future objects can be used within a RESULT CALLBACK and POLL OBJECT.

In the case of a RESULT CALLBACK, an ASYNCHRONOUS COMPLETION TOKEN [19] can be used to allow clients to identify different results of asynchronous invocations to the same REMOTE OBJECT.

MONITOR OBJECT [19] is an alternative to ACTIVE OBJECT for synchronizing and scheduling concurrently invoked remote operations. A MONITOR OBJECT ensures that only one operation runs within an object at a time by queuing operation executions on a mutex – this is a key difference to ACTIVE OBJECT where requests are queued. It applies one lock per monitor object to synchronize access to all operations.

The pattern HALF-SYNC/HALF-ASYNC decouples asynchronous and synchronous processing by defining an asynchronous and a synchronous service processing layer. A queue between these layers maps asynchronous invocations to synchronous execution.

A HALF-SYNC/HALF-ASYNC architecture has a few drawbacks, in particular, the data passing overhead (for dynamic memory allocation, synchronization operations, etc.) and the latency are unnecessarily high. The pattern LEADER/FOLLOWERS [19] solves these problems. It also allocates a number of threads, however, it does not separate the architecture into a synchronous and an asynchronous layer. Instead the threads take turns being the leader. The leader receives an event, and handles it (or tries to handle it). Before event handling, the leader promotes one of the threads in the pool (the "followers") to become the new leader, which then receives events. After handling the event, the thread is put back into the pool.

HALF-SYNC/HALF-ASYNC and/or LEADER/FOLLOWERS can be used to manage network connections and threading efficiently. POOLING [19] can be used to thus minimize the initialization overhead of threads or connections. The threads of a thread pool can be managed by LEADER/FOLLOWERS.

DESIGN AND IMPLEMENTATION OF AN ASYNCHRONOUS INVOCATION FRAMEWORK FOR APACHE AXIS

In this section, we explain a framework design implementing the invocation 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.

Requesters

Our general design relies on the REQUESTER pattern [23]. A REQUESTER is provided as a local object within the client process that can construct invocations to arbitrary REMOTE OBJECTS in a generic way. A REQUESTER dynamically constructs an invocation and hides networking details. In most distributed object frameworks, CLIENT PROXIES [23] are provided as well: CLIENT PROXIES offer the REMOTE OBJECT'S interface. They are most often generated from an INTERFACE DESCRIPTION [23], such as WSDL in the case of Web Services. In our description, we first concentrate on a REQUESTER that builds up remote invocations at runtime. We also discuss how to use the CLIENT PROXIES that are automatically generated from WSDL (see below).

In our framework we provide two kinds of REQUESTERS, one for synchronous invocations and one for asynchronous invocations. Both use the same invocation scheme. The synchronous REQUESTER blocks the invocation until the response returns. Thus it is just a wrapper to the ordinary REQUESTER of the Axis framework, provided for convenience. A client can invoke a synchronous REQUESTER by instantiating it and waiting for the result:

```
SyncRequester sr = new SyncRequester();
String result = (String)
  sr.invoke(endpointURL, operationName,
            null, rt);
```

This REQUESTER simply instantiates a handler for dealing with the invocation, and after the response has arrived, it returns control to the client.

The asynchronous REQUESTER 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 handler, corresponding to the kind of invocation, is instantiated in its own thread of control. The general structure of asynchronous invocations is quite similar to synchronous invocations. 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 = ...;
AsyncRequester ar = new asyncRequester();
ar.invoke(ah, clientACT, endpointURL,
          operationName, null, rt);
// ... resume work
```

Note that the `clientACT` field is used here as a pure client-side implementation of an ASYNCHRONOUS COMPLETION TOKEN (ACT) [19]. 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 [19] 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 behaviour. 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.

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, operation name, an argument list, and a return type. Also constructing an invocation using a REQUESTER (in Axis this is done using the type `Call`) from these information is common for all different kinds of invocation handlers (see Figure 2).

<<Figure 2 to be inserted here>>

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 of the COMMAND pattern [10] 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 COMPLETION 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 below). The decision is done using Java's `instanceof` primitive.

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

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 [8] 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 invocation using Axis' class `Call`, performs the invocation, and then the thread terminates.

A special `invokeFireAndForget` method of the `AsyncRequester` class is used for invoking FIRE AND FORGET operations:

```
AsyncRequester ar = new AsyncRequester();
ar.invokeFireAndForget(endpointURL,
                       operationName,
                       null, rt);
```

<<Figure 3 to be inserted here>>

Figure 3 shows the dynamic invocation behaviour of a FIRE AND FORGET invocation.

Asynchrony Pattern Handlers

To deal with the asynchrony patterns RESULT CALLBACK, POLL OBJECT, and 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 REQUESTER, for instance, in the `invoke` method.

The asynchronous REQUESTER 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 below). 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. An client asynchrony handler is a class implementing one of the following interfaces: `ResultCallback`, `PollObject`, and `SyncWithServer`. The client provides the asynchrony handler to the REQUESTER'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` parameter to correlate a specific result to an invocation. The four last parameters 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:

```
AsyncRequester requester = new AsyncRequester();
PollObject p = (PollObject)
                  new SimplePollObject();
requester.invoke(p, null, endpointURL,
                 operationName, null, rt);
while (!p.resultArrived()) {
  // do some other task ...
}
System.out.println("Polled 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 4.

<<Figure 4 to be inserted here>>

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 [19], which is supported by Java's `synchronized` language construct. The operations of each client asynchrony handler are synchronized and the access is scheduled.

Figure 5 shows the dynamic invocation behaviour 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 REQUESTER, 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.

<<Figure 5 to be inserted here>>

Queued Asynchrony Handlers

MESSAGE QUEUES can be used at the transport level, for instance, for building a simple messaging system. As mentioned before, MESSAGE QUEUES can be used together with RESULT CALLBACK and POLL OBJECT for result queuing. To a certain extent, MESSAGE QUEUES can also be used in context of SYNC WITH SERVER for queuing the acknowledgments. However, MESSAGE QUEUES at the transport layer will not work for SYNC WITH SERVER, as long as somebody between client and the transport layer, such as the REQUESTER, blocks until the acknowledgement is received. If MESSAGE QUEUES are not supported at the transport level, we can introduce queues at the invocation layer to support acknowledgment queuing for SYNC WITH SERVER or result queuing for RESULT CALLBACK or POLL OBJECT (as exemplified below).

Consider we want to use one instance to handle multiple responses. A simple implementation of such behaviour is an asynchrony handler that uses a queue for the arriving responses. Such queuing handlers with FIFO (first-in, first-out) behaviour are pre-defined in our framework for RESULT CALLBACK, POLL OBJECT, and SYNC WITH SERVER. In the SYNC WITH SERVER variant the acknowledgements are stored in the queue, otherwise the results. Figure 4 already depicts these queuing variants.

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

As a second example, consider a developer defines 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:

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

In this example the ten invocations are all reported to one and the same queuing RESULT CALLBACK object. This object can either handle the result on its own (for instance 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.

Using WSDL Generated Asynchronous Client Proxies

WSDL [8] is used as a standard INTERFACE DESCRIPTION [23] 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:

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

PERFORMANCE CONSIDERATIONS

The asynchronous invocation framework provides a better client performance than synchronous invocations 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 (which is discussed first in this Section). Next, we compare the performance of asynchronous invocations to synchronous invocations in our framework.

Thread Pooling

To optimize resource allocation for threading, the threads can be shared in a pool using the POOLING pattern [15]. This optimization can be combined both with a HALF-SYNC/HALF-ASYNC [19] or a LEADER/FOLLOWERS [19] concurrency architecture. As explained above, LEADER/FOLLOWERS is used for sharing a resource between several threads.

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.

We use a generic thread pool with thread pool workers that require the client to provide COMMANDS [10] of the type `Runnable` (see discussion above). 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.

Figure 6 shows the thread pool design.

<<Figure 6 to be inserted here>>

Performance Comparison

For 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.

<<Table 2 to be inserted here>>

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.

RELATED WORK AND 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) [13]. 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 protocols 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 does deal with asynchrony using messaging protocols (HTTPR, JMS, IBM MQSeries Messaging, MS Messaging). The approach presented in this paper can potentially be used on top of with WSIF – only implementing a few ADAPTERS [10] is necessary.

A similar situation can be found in the Mind Electric's GLUE [17], another popular Java implementation of Web Services, that supports HTTP, HTTP over SSL, and JMS as transport protocols. JMS has to be used to support client asynchrony.

For a long time CORBA [11] supported only synchronous communication and unreliable one-ways operations, which were not really an alternative due to the lack of reliability and potential blocking behaviour. 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 [16] 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 [18] 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 sinks 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.

CONCLUSION

In this paper we have provided a pure client side approach to provide asynchronous invocations for Web Services without necessarily using asynchronous messaging protocols. The framework was designed from a set of patterns of 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 at the beginning of this paper, were reached; in particular:

- A client can significantly faster resume with its work. That is, the client does not depend on the dispatching and processing time of the remote invocation.
- The invocation framework is very simple and can flexibly be extended with custom handlers.
- Other communication protocols than HTTP and back-ends of Web Services (so-called "service providers"), supported by Axis, can be used with our framework. The framework design does not rely on Axis though. By writing ADAPTERS for our framework, other Web Service frameworks can be used with our framework as well.
- 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. We have quantified this overhead in the performance measurement section. As we have provided a pure client side implementation, functionalities that rely on server framework support, such as synchronous acknowledgments for SYNC WITH SERVER can only be realized at the application layer; that is, with support in the REMOTE OBJECT implementation. Further functionalities of messaging protocols, such as guaranteed delivery, message channels, or message expirations, are not supported. But as messaging protocols can be used internally this is not a severe drawback.

REFERENCES

1 C. Alexander (1979) *The Timeless Way of Buildin*g. Oxford Univ. Press.

2 Apache Software Foundation (2002) *Web services invocation framework (WSIF).* http://ws.apache.org/wsif/

3 Apache Software Foundation (2003) *Apache Axis*. http://ws.apache.org/axis/

4 M. Cai, S. Ghandeharizadeh, R. Schmidt, S. Song (2002) *A Comparison of Alternative Encoding Mechanisms for Web Services*. In Proceedings of the 13th International Conference on Database and Expert Systems Applications (DEXA), Aix en Provence, France.

5 D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, D. Orchard (2003) *Web Services Architecture,* W3C Working Draft 8 August 2003, http://www.w3.org/TR/2003/WD-ws-arch-20030808/

6 D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer (2000) *Simple object access protocol (SOAP) 1.1.* http://www.w3.org/TR/SOAP/

7 T. Bray, J. Paoli, and C. Sperberg-McQueen (1998) *Extensible markup language (XML) 1.0.* http://www.w3.org/TR/1998/REC-xml-19980210

8 E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana (2001) *Web services description language (WSDL) 1.1*. http://www.w3.org/TR/wsdl

9 R. Fielding, J. Gettys, J. Mogul, H. Frysyk, L. Masinter, P. Leach, and T. Berners-Lee (1999) *Hypertext transfer protocol – HTTP/1.1*. RFC 2616

10 E. Gamma, R. Helm, R. Johnson, and J. Vlissides (1994) *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

11 Object Management Group (2000) *Common request broker architecture (CORBA).* http://www.omg.org/corba

12 G. Hohpe, B. Woolf (2003) *Enterprise Integration Patterns*, Addison-Wesley.

13 IBM developerWorks (2002) *HTTPR specification*. http://www-106.ibm.com/developerworks/webservices/library/ws-httprspec/

14 R. E. Johnson and B. Foote (1988) *Designing reusable classes*. Journal of Object-Oriented Programming, Vol. 1, No. 2, pp. 22–35.

15 M. Kircher and P. Jain (2004). *Pattern-Oriented Software Architecture-Patterns for Resource Management,* To be published by *J. Wiley and Sons.*

16 Microsoft (2003) *.NET framework*. http:///msdn.microsoft.com//netframework

17 The Mind Electric (2003). GLUE. http://www.themindelectric.com/glue/

18 G. Neumann and U. Zdun. (2001) *Distributed web application development with active web objects.* In Proceedings of The 2nd International Conference on Internet Computing (IC'2001), Las Vegas, Nevada, USA.

19 D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann (2000) *Patterns for Concurrent and Distributed Objects. Pattern-Oriented Software Architecture.* J. Wiley and Sons Ltd.

20 S. Vinoski (2003) *IEEE Internet Computing. Toward Integration Column: Integration With Web Services.* Nov-Dec 2003.

21 M. Voelter, M. Kircher, and U. Zdun (2002) *Object-oriented remoting: A pattern language.* In Proceedings of The First Nordic Conference on Pattern Languages of Programs (VikingPLoP 2002), Denmark.

22 M. Voelter, M. Kircher, and U. Zdun (2003) *Patterns for asynchronous invocations in distributed object frameworks.* In Proceedings of EuroPlop 2003, Irsee, Germany.

23 M. Voelter, M. Kircher, and U. Zdun (2004) *Remoting Patterns*. To be published by J. Wiley and Sons Ltd. in Wiley's pattern series in 2004.

24 D. Winer (1999) *XML-RPC specification.* http://www.xmlrpc.com/spec

TABLES

Pattern name	Acknowledgement to client	Result to client	Responsibility for result
FIRE AND FORGET	no	no	-
SYNC WITH SERVER	yes	no	-
POLL OBJECT	yes	yes	The client is responsible for getting the result.
RESULT CALLBACK	yes	yes	The client is informed via callback.
MESSAGE QUEUE	yes	yes	The server actively sends back the result. The client can receive it synchronously (by blocking on a message queue) or asynchronously using one of the other asynchrony patterns.

Table 1: Summary of the Invocation Asynchrony Patterns

Performance Test	Synchronous	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

FIGURES

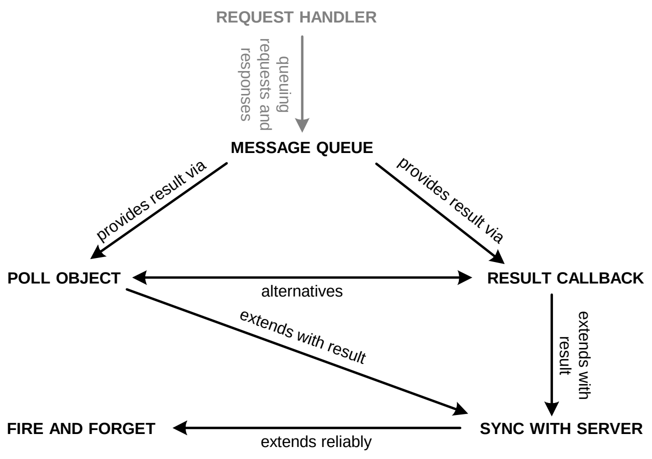

Figure 1: Overview Invocation Asynchrony Patterns

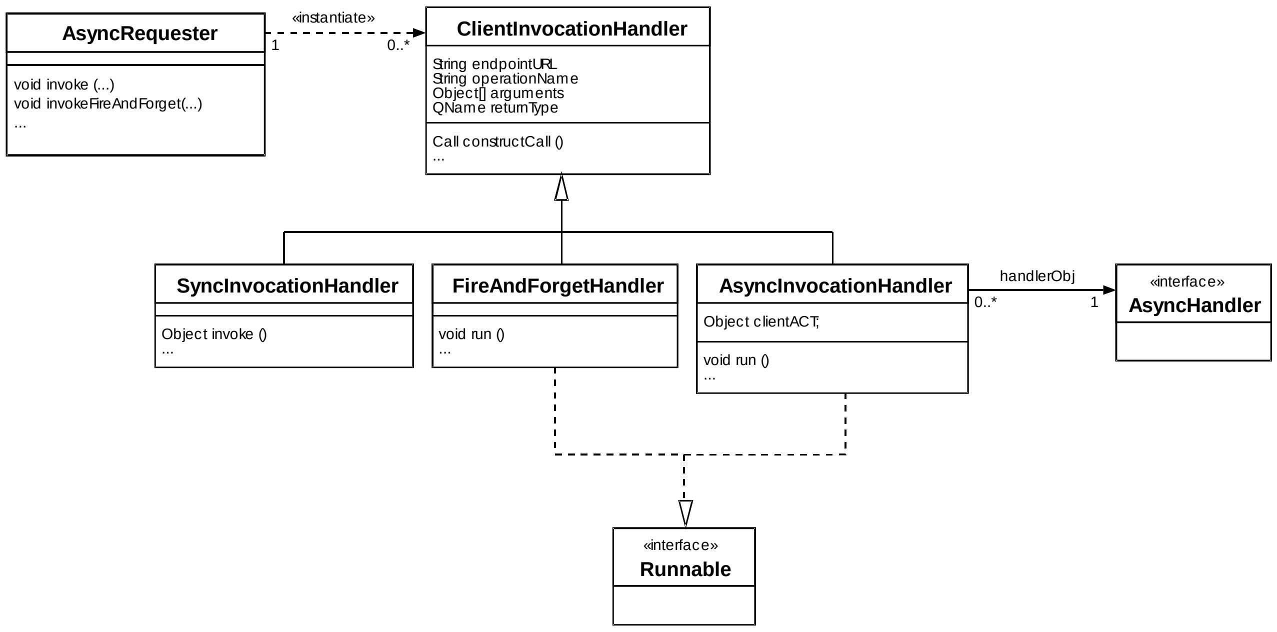

Figure 2: Invocation Handlers

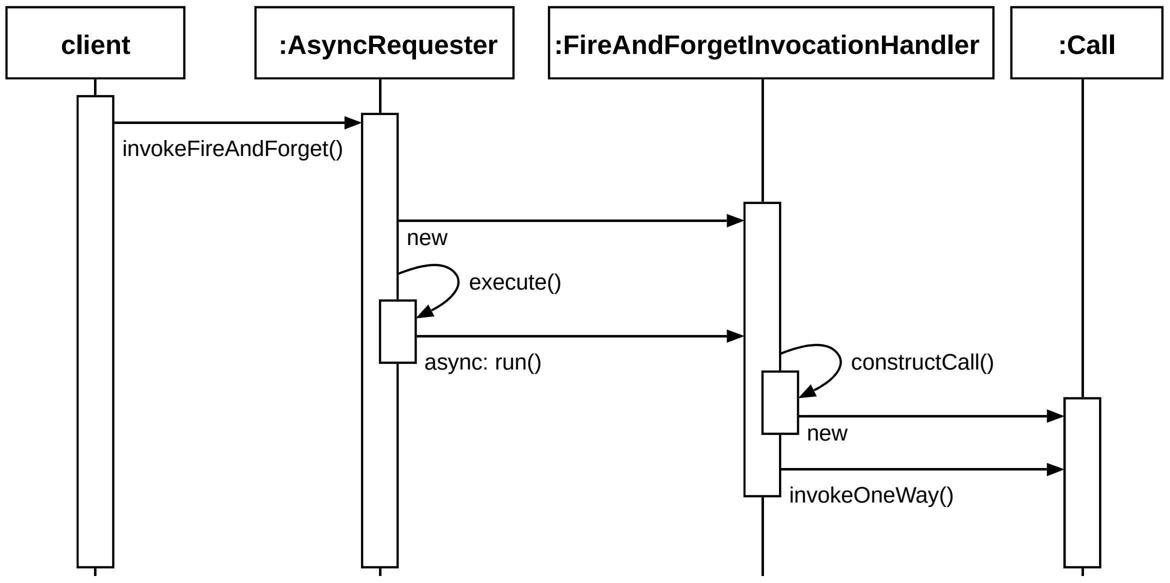

Figure 3: Fire And Forget Dynamics

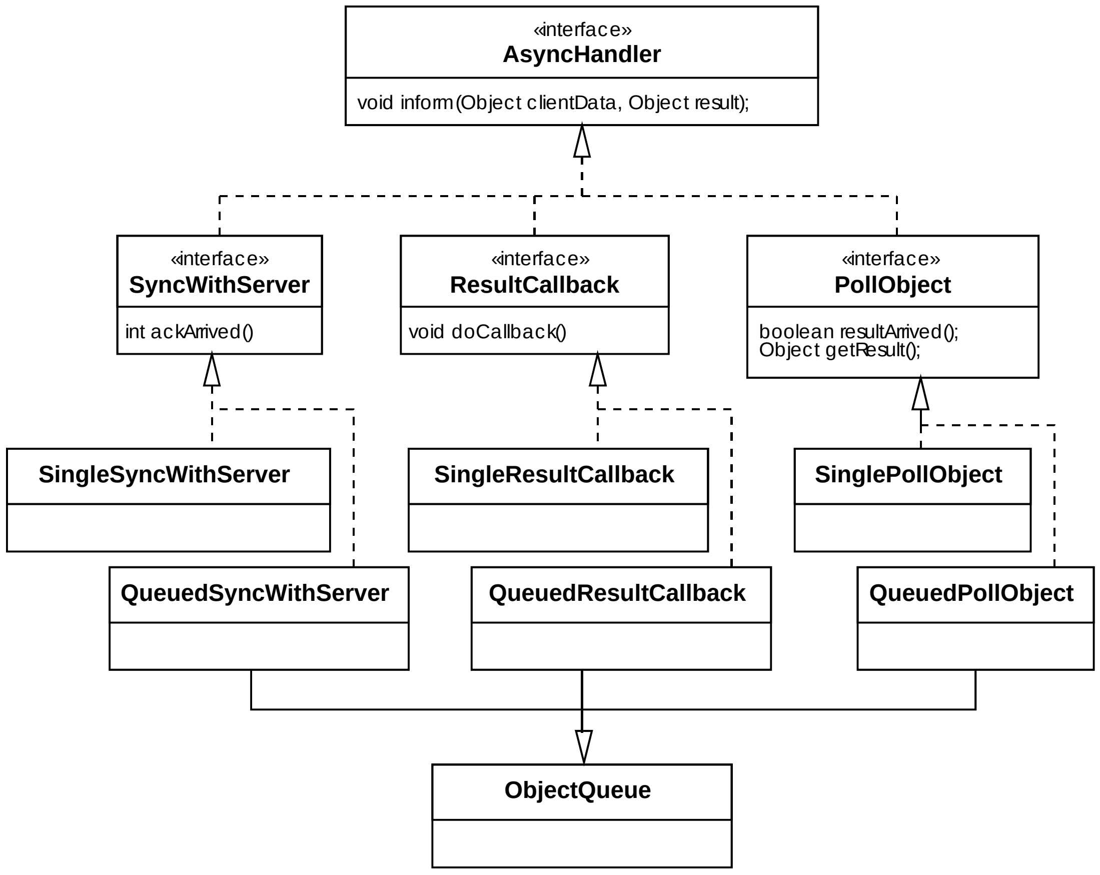

Figure 4. Handlers for Obtaining Asynchronous Results

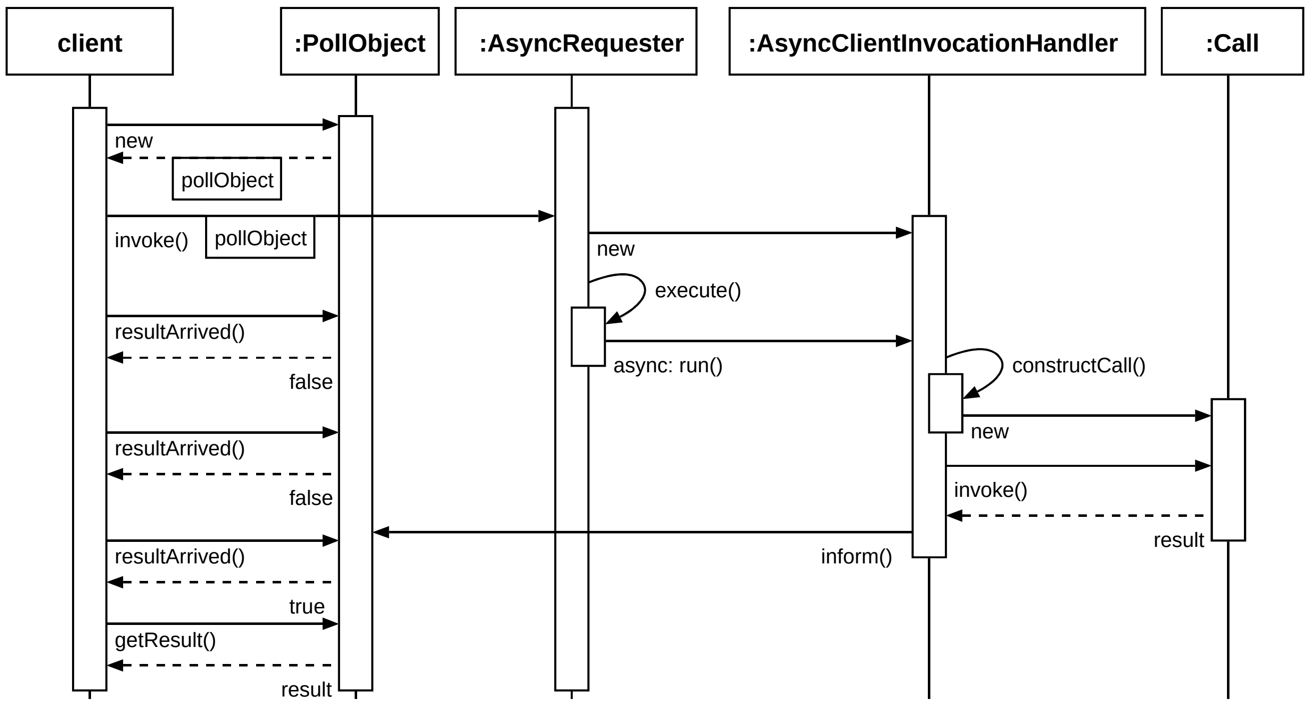

Figure 5. Poll Object Dynamics

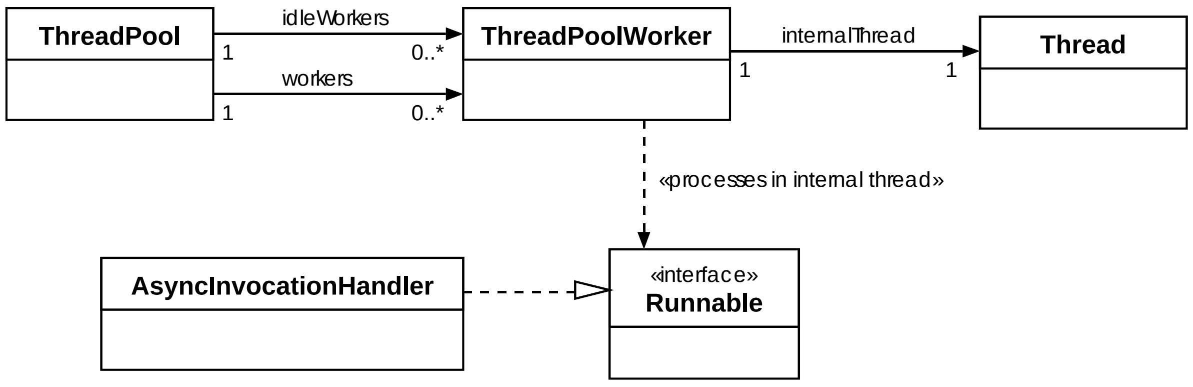

Figure 6. Thread Pooling