# Patterns for Asynchronous Invocations in Distributed Object Frameworks

*Markus Voelter*
voelter,
Ingenieurbüro für Softwaretechnologie,
Germany,
voelter@acm.org

*Michael Kircher*
Siemens AG,
Corporate Technology,
Software and System Architectures,
Germany,
michael.kircher@siemens.com

*Uwe Zdun*
New Media Lab,
Department of Information Systems,
Vienna University of Economics,
Austria,
zdun@acm.org

*Michael Englbrecht*
AddOn Software GmbH,
Germany,
Michael.Englbrecht@addon.de

NOTE: These patterns will appear in a heavily reworked and updated version in the Remoting Pattern book [VKZ04], to be published by Wiley in 2004.

**The patterns in this paper introduce the four most commonly used techniques for providing client-side asynchrony in distributed object frameworks.** FIRE AND FORGET **describes best-effort delivery semantics for asynchronous operations that have void return types.** SYNC WITH SERVER **notifies the client only in case the delivery of the invocation to the server application fails.** POLL OBJECTS **provide clients with means to query the distributed object framework whether an asynchronous response for the request has arrived yet, and if so, to obtain the return value.** RESULT CALLBACK **actively notifies the requesting client of the returning result.**

## Introduction

OO-RPC middleware typically provides synchronous remote method invocations from clients to server objects. In some scenarios, asynchronous behavior is necessary, though. This collection of patterns introduces the four most commonly used techniques in this context.

FIRE AND FORGET describes best-effort delivery semantics for asynchronous operations that have void return types. SYNC WITH SERVER looks the same from the client's point of view, however it is able to notify the client (by throwing an exception) in case the delivery of the invocation to the SERVER APPLICATION fails. POLL OBJECTS provide clients with means to query the distributed object framework whether an asynchronous reply for the request has arrived yet, and if so, to obtain the return value. Last but not least, RESULT CALLBACK will actively notify the requesting client of the returning result.

Note that these patterns are part of a larger pattern language on remoting middleware (see also [VKZ04]). This is why some of the PATTERN REFERENCES point to patterns not found in this paper.

| | Result to client | Ack to client | Responsibility for result |
|---|---|---|---|
| Fire and Forget | no | no | - |
| Sync with Server | no | yes | - |
| Poll Object | yes | yes | Client is responsible |
| Result Callback | yes | yes | Client is informed via callback |

## Fire and Forget

Your SERVER APPLICATION provides REMOTE OBJECTS with operations that have neither a return value nor report any exceptions.

❊ ❊ ❊

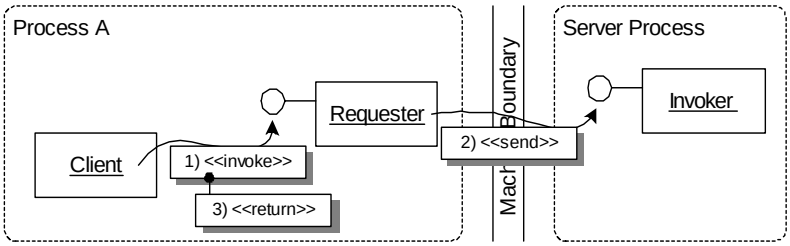
**In many situations, a client application needs to invoke an operation on a REMOTE OBJECT simply to notify the REMOTE OBJECT of an event. The client does not expect any return value. Reliability of the invocation is not critical, as it is just a notification that both client and server do not critically 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.

Note that this scenario is quite typical for distributed implementations of patterns, such as *Model-View-Controller* [BMR+96] or *Observer* [GHJV95], especially if the view or observer is constantly notified and old data is stale data.

Therefore:

**Provide FIRE AND FORGET operations. When invoked, the REQUESTER sends the invocation across the network, returning control to the calling client immediately. The client does not get any acknowledgement from the REMOTE OBJECT receiving the invocation in case of success or failure.**



When the client invokes a FIRE AND FORGET operation, the REQUESTER marshals the parameters and sends them to the server.

❊ ❊ ❊

The implementation of a FIRE AND FORGET operation can be done in multiple ways, specifically:

- The REQUESTER can simply put the bytes on the wire in the caller's thread, assuming the send operation does not block. Here, asynchronous I/O operations, as supported by some operating systems are of great help to avoid blocking.
- Alternatively, the REQUESTER can spawn a new thread, that puts the bytes on the wire independently from the thread that invoked the remote operation. This variant also works when the send operation temporarily blocks. However, this variant has some drawbacks: it works only as long as the application does not get bogged down from the operating system perspective due to huge numbers of such threads, and the existence of such threads does not overwhelm the underlying marshaling, protocol, and transport implementations due to lock contention, etc. Another drawback of concurrent invocations is that an older invocation may bypass a younger one. This can be avoided by using MESSAGE QUEUES.
- As FIRE AND FORGET operations are not considered to be reliably transported, an option is to use unreliable protocols such as UDP for their implementation, which are much cheaper than reliable protocols such as TCP.

The INVOKER on the server side typically differentiates between FIRE AND FORGET operations and synchronous operations, as it is not necessary to send a reply for a FIRE AND FORGET operation. When the remote invocation is performed in a separate thread, a thread pool will be used instead of spawning a new thread for each invocation to avoid a thread creation overhead.

In cases where the distributed object framework does not provide FIRE AND FORGET operations, the application can emulate such behavior by spawning a thread itself and performing the invocation in that newly created thread. But be aware that such an emulation heavily influences

the scalability. In particular, many concurrent requests lead to many concurrent threads, decreasing overall system performance.

The benefit of the FIRE AND FORGET pattern is the asynchrony it provides compared to synchronous invocations. Client and REMOTE OBJECT are decoupled, in the sense that the REMOTE OBJECT executes independently of the client; the client does not block during the invocation. This means the pattern is very helpful in event-driven applications that do not rely a continuous control flow nor on return values. Further, it is important that the applications do not rely on the successful transmission.

However, REMOTING ERRORS during sending the invocation to the remote object or errors that were raised during the execution of the remote invocation 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 counteractions.

## Sync with Server

Your SERVER APPLICATION provides REMOTE OBJECTS with operations that have neither return value nor report any errors, but FIRE AND FORGET is too unreliable.

❄ ❄ ❄

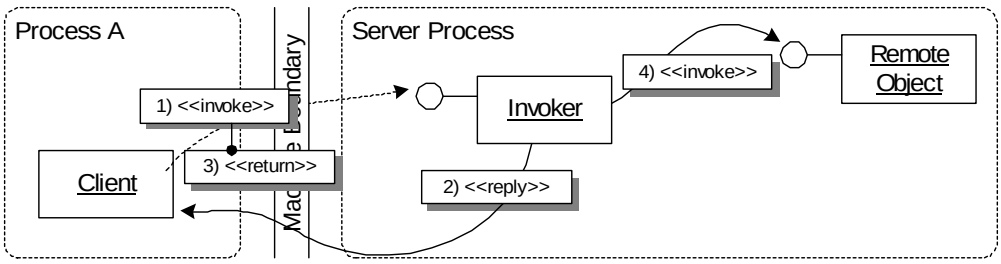
**FIRE AND FORGET is a useful but extreme solution in the sense that it can only be used if the client can really afford to take the risk of not noticing when a remote invocation does not reach the targeted REMOTE OBJECT. The other extreme is a synchronous call where a client is blocked until the remote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

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 delivered as a message to the server. 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:

**Provide SYNC WITH SERVER semantics for remote invocations. 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 (not the execution!) 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.**




A client invokes a remote operation. The REQUESTER puts the bytes of the invocation on the wire, as in FIRE AND FORGET. But it then waits for a reply from the SERVER APPLICATION, that the invocation has been received from by the server.

❋ ❋ ❋

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 REQUESTER about the successful reception.

If the distributed object framework supports SYNC WITH SERVER operations, the INVOKER can send the reply message immediately after reception of the invocation. Otherwise, SYNC WITH SERVER can be emulated by hand-coding SYNC WITH SERVER into the respective operation of the REMOTE OBJECT. The operation spawns a new thread that performs the remote invocation while the initial thread invoking the remote invocation returns immediately resulting in a reply to the client.

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 REQUESTER 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.

## Poll Object

Invocations of REMOTE OBJECTS should execute asynchronously but the client depends on the results for further computations.

❊ ❊ ❊

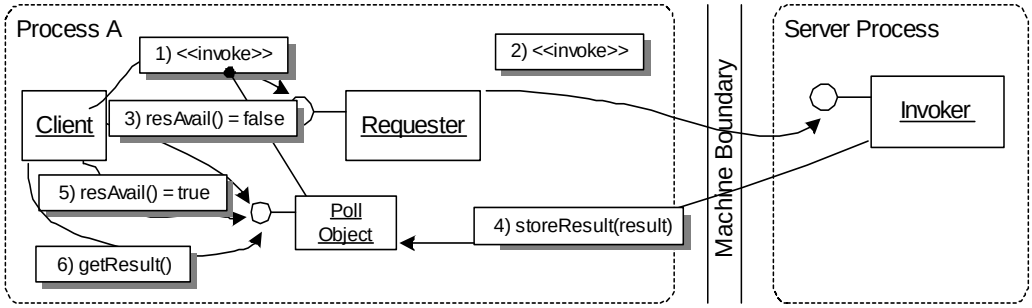
**There are situations, when an application needs to invoke an operation asynchronously, but still requires to know the results of the invocation. The client does not necessarily need the results immediately to continue its execution, and it can decide for itself when to use the returned results.**

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 shall have an 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:

**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 with other tasks asynchronously.**




A client invokes an remote operation on the REQUESTER, which in turn creates a POLL OBJECT to be returned to the client immediately. As long as the remote invocation has not returned, the "result available" method returns false. When the result becomes available, it is memorized in the POLL OBJECT. When it is polled the next time, it returns true, so that the client can fetch the result by calling the "get result" method.

❄ ❄ ❄

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. Besides this client interface, an operation for storing the result as received from the server is needed.

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. The core idea of this pattern follows the concept of *Futures*, but POLL OBJECT extends it to distributed settings and allows to query for the availability of the result non-blocking, whereas with *Futures* this query would block.

POLL OBJECTS typically depend on the interface of the REMOTE OBJECT. To be able to distinguish results of two invocations on the same remote object, either, the two above mentioned methods must be provided for each of the remote object's operations, or a separate POLL OBJECT for each operation must exist.

Use POLL OBJECTS when the time until the result is received is expected to be rather short; however, it should be long enough so that the client

can use the time for other computations. For longer waiting periods, especially if the period cannot be pre-estimated, use a RESULT CALLBACKS, as it is typically hard to cope with long waiting periods.

POLL OBJECTS offers the benefit that the client application does not have to use an event-driven, completely asynchronous programming model, as in the case with RESULT CALLBACK, while it can still make use of asynchrony to some extend. The SERVER APPLICATION can stay unaware to client side poll objects.

From an implementation perspective, the client framework typically starts a separate thread to "listen" for the result and fill it into the POLL OBJECT. In this thread, the client typically sends a synchronous invocation to the SERVER APPLICATION.

Some synchronisation mechanisms between the client thread and the retrieving thread are needed, for instance by applying mutexes or a *Thread-Safe Interface* [SSRB00].

When using POLL OBJECTS the client has to be changed slightly to do the polling. POLL OBJECTS either need to be generic, which typically requires programming language support, or they have to be specific to the REMOTE OBJECT and its interface operations. In the latter case they are typically code generated. More dynamic environments can use runtime means to create the types for POLL OBJECTS.

## Result Callback

Your SERVER APPLICATION provides REMOTE OBJECTS with operations that have return values and/or may return errors. The result of the invocation is handled asynchronously.

❋ ❋ ❋

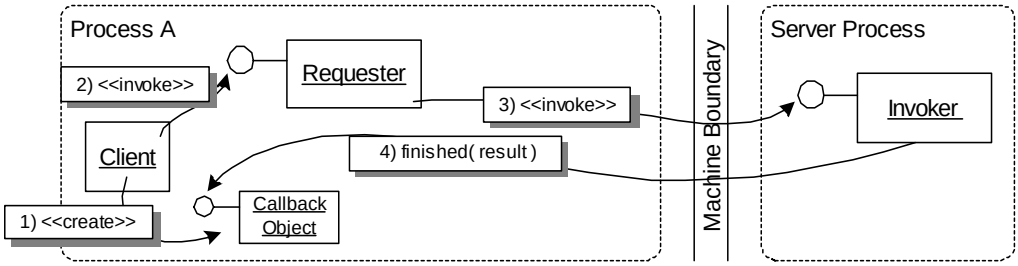
**The client needs to be actively informed about results of asynchronously invoked operations on a REMOTE OBJECT. That is, if the result becomes available to the REQUESTER, the client wants to be informed immediately to react on it. In the meantime the client executes concurrently.**

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 subsequently 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 RESULT CALLBACK object to the REQUESTER. The invocation returns immediately after sending the invocation to the server. Once the result is available, the distributed object framework invokes a predefined operation on the RESULT CALLBACK object, passing it the result of the invocation (this can be triggered by the CLIENT REQUEST HANDLER, for instance).**




The client instantiates a callback object and invokes the operation on the REQUESTER. When the result of the remote invocation returns, it is dispatched by the distributed object framework to the callback object, calling a pre-defined callback method.

❊ ❊ ❊

You can use the same or separate callback objects for each invocation of the same type. Somehow the correct callback object has to be identified. There are different variants, and it depends on the application case which variant works better:

- When you reuse a callback object you obviously need an *Asynchronous Completion Token* [SSRB00] to associate the callback with the original invocation. An *Asynchronous Completion Token* contains information that uniquely identify the callback object and method that is responsible for handling the result message.
- In cases where a callback object is not reused, responses to requests, do not need to be further demultiplexed, which simplifies interaction and no *Asynchronous Completion Token* is necessary.

Using RESULT CALLBACK the client can immediately react on results of asynchronous invocations. As in the case of POLL OBJECT, the SERVER APPLICATION has nothing to do with the specifics of result handling in the client. The use of RESULT CALLBACK requires an event-driven application design, whereas POLL OBJECTS allow to keep a synchronous programming model.

However, the RESULT CALLBACK pattern also incurs the liability that client code, namely the code doing the original asynchronous invocation, and the code associated with the RESULT CALLBACK, is executed in

multiple thread contexts concurrently. The client code therefore needs to be prepared for that, for example when accessing resources. There needs to be a separate thread or process handling asynchronous replies - this also is true for POLL OBJECTS.

The callback itself can either be implemented inside the client only; i.e. the client executes an ordinary synchronous invocation on a REMOTE OBJECT in a separate thread. When the invocation returns, the clients calls back into the provided callback object. Alternatively, a "real" callback from the distributed object framework in the SERVER APPLICATION can be used. This requires that the client makes the callback object available as a REMOTE OBJECT for the server-side distributed object framework to invoke. Note that in both cases the implementation of the target REMOTE OBJECT is not affected.

In network configurations where a firewall exists between client and server, callback invocations using a new connection for the callback are problematic, as they might get blocked by the firewall. There are two options to solve this problem:

- Use bidirectional connections that allow requests to flow in both directions.
- Let the callback object internally poll the REMOTE OBJECT whether the result is available.

As the second option is fairly complex, the first option should be considered in practice.

The major difference between POLL OBJECT and RESULT CALLBACK is that RESULT CALLBACK requires an event-driven design, whereas POLL OBJECT allows to keep an almost synchronous execution model.

## Related Patterns

An REMOTE OBJECT is very similar to an *active object* in [SSRB00]. An *active object* [SSRB00] decouples method invocation from method execution. The same holds true for REMOTE OBJECTS that use any of the above presented patterns for asynchronous communication. 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 follows the same patterns are presented in RESULT CALLBACK and POLL OBJECT.

In the case of a result callback, an *asynchronous completion token* [SSRB00] can be used to allow clients to identify different results of asynchronous invocations to the same REMOTE OBJECT.

## Know Uses Examples

For a long time CORBA [OMG00] 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 [Mic03] provides an API for asynchronous remote communication. Similar to our approach, 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.

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). These messaging protocols do not provide a protocol-independent interface to client-side asynchrony and require developers to use the messaging communication paradigm. The patterns presented in this paper can be used to provided asychrony for the sychnronous web service protocols as well.

## Acknowledgements

Thanks to Angelo Corsaro, our EuroPLoP 2003 shepherd, for his helpful comments, as well as to all the participants of the EuroPLoP writer's workshop.

## References

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*, *John Wiley and Sons*, *1996*.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. *Addison-Wesley*, *1995*.

[Mic03] Microsoft. *.NET framework*. http:// msdn.microsoft.com/netframework, 2003.

[OMG00] Object Management Group (OMG). *Common request broker architecture (CORBA)*. http://www.omg.org/ corba, 2003.

[SSRB00] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Patterns for Concurrent and Distributed Objects. Pattern-Oriented Software Architecture*. John Wiley and Sons, 2000.

[VKZ04] M. Voelter, M. Kircher, and U. Zdun. *Remoting Patterns - Patterns for Enterprise, Realtime and Internet Middleware*, Wiley & Sons, to be published in 2004