# **Choosing a Message Exchange Pattern**

* 03/29/2017
* 7 minutes to read
  + [+5](https://github.com/dotnet/docs/blob/master/docs/framework/wcf/extending/choosing-a-message-exchange-pattern.md" \o "10 Contributors)

The first step in writing a custom transport is to decide which message exchange patterns (or MEPs) are required for the channel you are developing. This topic describes the options available and discusses the various requirements. This is the first task in the channel development task list described in [Developing Channels](https://docs.microsoft.com/en-us/dotnet/framework/wcf/extending/developing-channels).

## **Six Message Exchange Patterns**

There are three MEPs to choose from:

Datagram ([IInputChannel](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.channels.iinputchannel) and [IOutputChannel](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.channels.ioutputchannel))

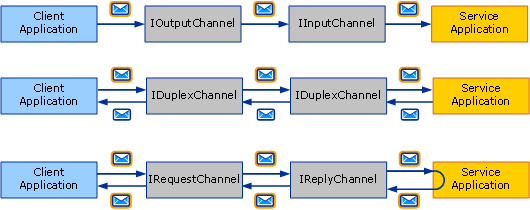
When using a datagram MEP, a client sends a message using a fire and forget exchange. A fire and forget exchange is one that requires out-of-band confirmation of successful delivery. The message might be lost in transit and never reach the service. If the send operation completes successfully at the client end, it does not guarantee that the remote endpoint has received the message. The datagram is a fundamental building block for messaging, as you can build your own protocols on top of it—including reliable protocols and secure protocols. Client datagram channels implement the [IOutputChannel](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.channels.ioutputchannel) interface and service datagram channels implement the [IInputChannel](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.channels.iinputchannel) interface.

Request-Response ([IRequestChannel](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.channels.irequestchannel) and [IReplyChannel](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.channels.ireplychannel))

In this MEP, a message is sent, and a reply is received. The pattern consists of request-response pairs. Examples of request-response calls are remote procedure calls (RPC) and browser GET requests. This pattern is also known as half-duplex. In this MEP, client channels implement [IRequestChannel](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.channels.irequestchannel) and service channels implement [IReplyChannel](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.channels.ireplychannel).

Duplex ([IDuplexChannel](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.channels.iduplexchannel))

The duplex MEP allows an arbitrary number of messages to be sent by a client and received in any order. The duplex MEP is like a phone conversation, where each word being spoken is a message. Because both sides can send and receive in this MEP, the interface implemented by the client and service channels is [IDuplexChannel](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.channels.iduplexchannel).

  
The three basic message exchange patterns. Top to bottom: datagram, request-response, and duplex.

Each of these MEPs can also support sessions. A session (and implementation of [System.ServiceModel.Channels.ISessionChannel<TSession>](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.channels.isessionchannel-1) of type [System.ServiceModel.Channels.ISession](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.channels.isession)) correlates all messages sent and received on a channel. The request-response pattern is a stand-alone two-message session, as the request and reply are correlated. In contrast, the request-response pattern that supports sessions implies that all request/response pairs on that channel are correlated with each other. This gives you a total of six MEPs to choose from:

Datagram

Request-response

Duplex

Datagram with sessions

Request-response with sessions

Duplex with sessions

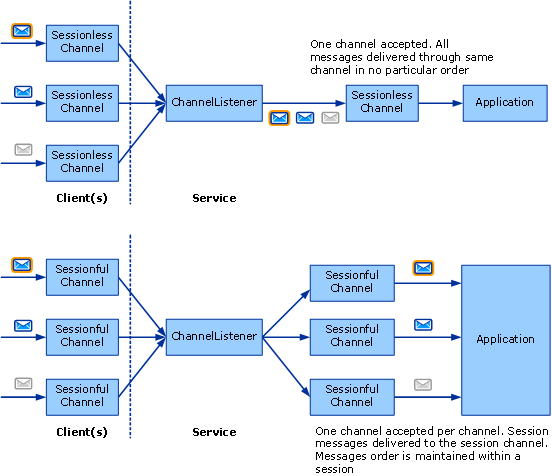
**Note**

For the UDP transport, the only MEP that is supported is datagram, because UDP is inherently a fire and forget protocol.

## **Sessions and Sessionful Channels**

In the networking world, there are connection-oriented protocols (for example, TCP) and connection-less protocols (for example, UDP). WCF uses the term session to mean a connection-like logical abstraction. Sessionful WCF protocols are similar to connection-oriented network protocols and sessionless WCF protocols are similar to connection-less network protocols.

In the channel object model, each logical session manifests as an instance of a sessionful channel. Therefore every new session created by the client, and accepted on the service, corresponds to a new sessionful channel on each side. The following diagram shows, on the top, the structure of sessionless channels, and on the bottom, the structure of sessionful channels.



A client creates a new sessionful channel and sends a message. On the service side, the channel listener receives this message and detects that it belongs to a new session so it creates a new sessionful channel and hands it to the application (in response to the application calling AcceptChannel on the channel listener). The application then receives this message and all subsequent messages sent in the same session through the same sessionful channel.

Another client (or the same client) creates a new sessionful and sends a message. The channel listener detects this message is in a new session and creates a new sessionful channel and the process repeats.

Without sessions, there is no correlation between channels and sessions. Therefore a channel listener creates only one channel through which all received messages are delivered to the application. There is also no message ordering because there is no session within which to maintain message order. The top portion of the preceding graphic illustrates a sessionless message exchange.

## **Starting and Terminating Sessions**

Sessions are started on the client by simply creating a new sessionful channel. They are started on the service when the service receives a message that was sent in a new session. Likewise, sessions are terminated by closing or aborting a sessionful channel.

The exception to this is [IDuplexSessionChannel](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.channels.iduplexsessionchannel) which is used for both sending and receiving messages in a duplex, sessionful communication pattern. It is possible that one side will want to stop sending messages but continue to receive messages therefore when using [IDuplexSessionChannel](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.channels.iduplexsessionchannel) there is a mechanism that lets you close the output session indicating you will not send any more messages but keep the input session opened allowing you to continue to receive messages.

In general, sessions are closed on the outgoing side and not on the incoming side. That is, sessionful output channels can be closed, thereby cleanly terminating the session. Closing a sessionful output channel causes the corresponding sessionful input channel to return null to the application calling [IInputChannel.Receive](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.channels.iinputchannel.receive) on the [IDuplexSessionChannel](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.channels.iduplexsessionchannel).

However sessionful input channels should not be closed unless [IInputChannel.Receive](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.channels.iinputchannel.receive) on the [IDuplexSessionChannel](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.channels.iduplexsessionchannel) returns null, indicating that the session is already closed. If [IInputChannel.Receive](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.channels.iinputchannel.receive) on the [IDuplexSessionChannel](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.channels.iduplexsessionchannel) has not returned null, closing a sessionful input channel may throw an exception because it may receive unexpected messages while closing. If a receiver wishes to terminate a session before the sender does, it should call [Abort](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.icommunicationobject.abort) on the input channel, which abruptly terminates the session.

## **Writing Sessionful Channels**

As a sessionful channel author, there are a few things your channel must do to provide sessions. On the send side, your channel needs to:

For each new channel, create a new session and associate it with a new session id which is a unique string. Or obtain a new session from the sessionful channel below you in the stack.

For each message sent using this channel, if your channel created the session (as opposed to obtaining it from the layer below you), you need to associate the message with the session. For protocol channels, this is typically done by adding a SOAP header. For transport channels, this is typically done by creating a new transport connection or adding session information to the framing protocol.

For each message sent using this channel, you need to provide the delivery guarantees mentioned above. If you are relying on the channel below you to provide the session, that channel will also provide the delivery guarantees. If you’re providing the session yourself, you need to implement those guarantees as part of your protocol. In general, if you are writing a protocol channel that assumes WCF on both sides you may require the TCP transport or the Reliable Messaging channel and rely on either one to provide a session.

When [ICommunicationObject.Close](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.icommunicationobject.close) is called on your channel, perform the necessary work to close the session using either the specified timeout or the default one. This can be as simple as calling [Close](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.icommunicationobject.close) on the channel below you (if you just obtained the session from it) or sending a special SOAP message or closing a transport connection.

When [Abort](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.icommunicationobject.abort) is called on your channel, terminate the session abruptly without performing I/O. This may mean doing nothing or may involve aborting a network connection or some other resource.

On the receive side, your channel needs to:

For each incoming message, the channel listener must detect the session it belongs to. If this is the first message in the session, the channel listener must create a new channel and return it from the call to [IChannelListener<TChannel>.AcceptChannel](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.channels.ichannellistener-1.acceptchannel). Otherwise the channel listener must find the existing channel that corresponds to the session and deliver the message through that channel.

If your channel is providing the session (along with the required delivery guarantees) the receive side may be required to perform some actions such as re-order messages or send acknowledgements.

When [Close](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.icommunicationobject.close) is called on your channel, perform the necessary work to close the session either the specified timeout or the default one. This could result in exceptions if the channel receives a message while waiting for the close timeout to expire. That’s because the channel will be in the Closing state when it receives a message so it would throw.

When [Abort](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.icommunicationobject.abort) is called on your channel, terminate the session abruptly without performing I/O. Again, this may mean doing nothing or may involve aborting a network connection or some other resource.

<https://docs.microsoft.com/en-us/dotnet/framework/wcf/extending/choosing-a-message-exchange-pattern>