# NServiceBus Document

## Overview

Designed for collaboration between business-oriented services, NServiceBus is **not** a replacement for RPC technologies like WCF.

Successful SOA and DDD projects use a mix of approaches and technologies - not just NServiceBus for communications.

Here you'll find the similarities and differences between NServiceBus and its Microsoft counterparts.

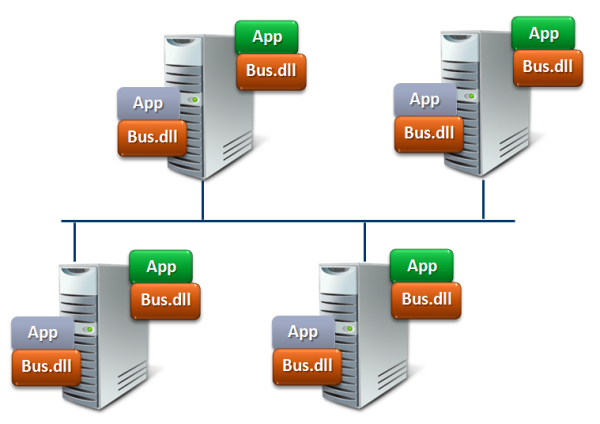
### Closer to WCF than to BizTalk



When many people hear the term "service bus" they picture a central box which all communication goes through, like BizTalk. That's actually a description of the Broker architectural style, not the Bus architectural style. A bus isn't necessarily a physical entity. In that respect, NServiceBus is more similar to WCF than it is to BizTalk.

There is no physical WCF one can point to in the network topology. WCF is part of the infrastructure that is run in-process with a given application's code. NServiceBus is the same.

Just like you can write your own host process and activate WCF explicitly within it, you can do the same thing with NServiceBus. The bus in NServiceBus is something virtual - the collection of framework objects running in the various applicative processes. You can think of it as a kind of peer-to-peer mesh that runs alongside your code as illustrated in the following diagram:



While there are additional stand-alone processes that come with NServiceBus (distributor, timeout manager, proxy, and gateway), they don't change this deployment model.

### Similarities - the next generation

The similarities between the platforms continues into their next versions. The coming Windows Server "Dublin" technologies found in .net 4.0 bring a standard host for WCF/WF applications, message forwarding services, and scaling out stateful workflow applications. While version 1.9 of NServiceBus already provided scaling out long-running business processes, version 2.0 comes with a standard host also with the ability to run as a windows service, console, and forms application. In 1.9, the NServiceBus Proxy supported request/response message forwarding and in 2.0 will be able to forward publish/subscribe messaging as well.

### What's the difference?

The principles which make NServiceBus as robust as it is are decades old. Proven to hold up through countless technological shifts, the queued messaging on which NServiceBus is based is more than just an implementation choice, it's a primary architectural concept. There's no such thing as a blocking call in NServiceBus.

As a general purpose communications technology, WCF does not enforce the queued messaging paradigm - NServiceBus does, and the architectural implications are profound.

When developing systems according to the traditional RPC techniques that WCF supports, it simple and straight-forward to get something working. That's when the problems start. Scalability and fault-tolerance are inherently hindered by RPC principles. At that point, it is close to impossible to solve these problems and even throwing more hardware at it has little effect. While WCF doesn't force developers down this path, it doesn't prevent them from doing so either. NServiceBus directs you away from these problems right from the beginning.

### Scalability with One-Way Messaging

<http://vimeo.com/6222577>

In this presentation, Udi Dahan explains the relationship between reliability, availability, and scalability and why architects should first focus on reliability.

After all, a highly available and scalable service that produces unreliable results isn't very valuable.

Throughout the presentation, the value of queued messaging is underlined and the way it handles various failure scenarios is discussed.

Although the recording missed the first 5-10 minutes, the core message has not been lost.

### Adoption and climbing the learning curve

While it does take some getting used to, code written using NServiceBus is quite a bit simpler and shorter than before, not to mention much easier to unit test. An architect in the financial services domain had this to say:

"It took a few weeks to grok the concepts involved in messaging but our devs needed only a week to implement a pub sub solution which is testament to how straightforward NServiceBus makes the coding. We have just started our NServiceBus journey but already are excited about what it has to offer."  
-- Charlie Barker

## Architectural Principles

Autonomy and Loose Coupling - at design time and at run time  
are not something that any technology can give you.

Service-Oriented Architecture and Event-Driven Architecture together  
provide the basis for identifying where NServiceBus should be used.

Strategic Domain-Driven Design helps bridge the Business / IT divide  
and drives the choice of business events published using NServiceBus.

### How NServiceBus aligns with SOA

<http://vimeo.com/5022174>

In this presentation, Udi Dahan explains about the disadvantages of classical web services thinking that places services at a layer below the user interface and above the business logic.

In its place, he describes an approach which cuts across all layers of an application, outlining the inherent lines of loose and tight coupling.

Finally, Udi shows how these vertical services collaborate together using events in order to bring about flexible and high performance business processes.

### Drilling down into the details

One of the problems with the distributed systems being built today is that they are fragile. As one part of the system slows down, that tends to ripple out and cripple the entire system. One of the primary design goals of NServiceBus was to eliminate that, guiding developers to writing code that would be robust in production environments. That robustness also needs to prevent data loss under various failure conditions.

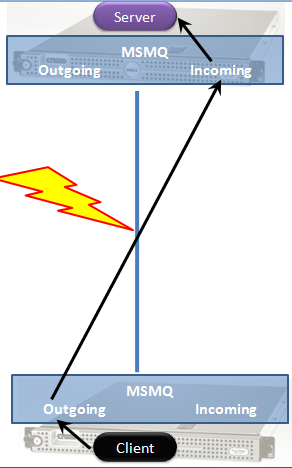
In order to make use of NServiceBus effectively, you need to understand the distributed systems architecture it is designed to support. In other words, if you design your system according to the principles laid out below, NServiceBus will make your life ***a lot*** easier. On the other hand, if you do not follow these principles, NServiceBus will probably make them harder.

The extensibility features in NServiceBus will enable you tweak its behavior to suit your specific needs, yet those will be documented separately.

The communications pattern that enables robustness is one-way messaging - also known as "fire and forget". That will be discussed in more detail shortly.

Since the amount of time it can take to communicate with another machine across the network is both unknown and unbounded, communications are based on a Store-and-Forward model as shown in the following diagram:

### Store & Forward Messaging

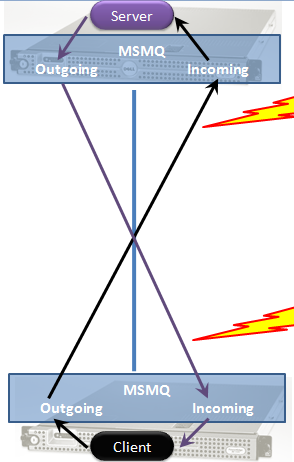


In this model, when the client process calls an API to send a message to the server process, the API returns control to the calling thread before the message is sent.

At that point, the transfer of the message across the network becomes the responsibility of the messaging technology. There may be various kinds of communications interference, the server machine may simply be down, or a firewall may be slowing down the transfer. Also, even though the message may have reached the target machine, the target process may currently be down.

While all of this is going on, the client process is oblivious. Critical resources like threads (and it's allocated memory) are not being held waiting for the call to complete. This prevents the client process from losing stability as a result of having many threads and all their memory used up waiting for a response from the other machine or process.

### Request / Response & One Way Messaging

The common pattern of Request/Response, which is more accurately described as Synchronous Remote Procedure Call, is handled differently when using one way messaging. Instead of letting the stack of the calling thread manage the state of the communications interaction, that is done explicitly. From a network perspective, request/response is just two one-way interactions as shown in the next figure:

This communication is especially critical for servers as client behind problematic network connections now have little effect on the server's stability.

If a client crashes between the time that it sent the request until the server sends a response, the server will not have resources tied up waiting minutes and minutes until the connection times out.

When used in concert with Durable Messaging, system-wide robustness increases even more.

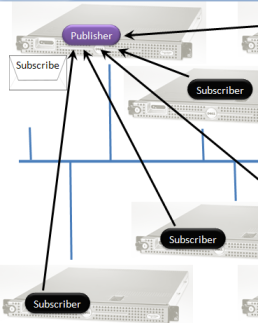
Durable messaging differs from regular store-and-forward messaging in that the messages are persisted to disk locally before attempting to be sent.

What this means is that if after the calling thread has had control returned to it the process crashes, the message sent will not have been lost. In server to server scenarios, where a server can complete a local transaction but might crash a second later, one-way durable messaging makes it easier to create an overall robust system even in the face of unreliable building blocks.

A different communication style involves one-to-many communication.

### Publish / Subscribe

In this style, the sender of the message often does not know about the specifics of those that wish to receive the message. This additional loose coupling comes at the cost of subscribers explicitly opting-in to receiving messages as shown in the following diagram:



#### Subscriptions

Subscribers need to know about which endpoint is responsible for a given message. This information is usually made available as part of the contract, specifying to which endpoint a subscriber should send its request. As a part of the subscription message, a subscriber passes its "return address", the endpoint at which it wants to receive messages.

Keep in mind that the publisher may choose to store the information about which subscriber is interested in which message in a highly available manner. This would allow multiple processes on multiple machines to publish messages to all subscribers, regardless if one had received the subscription message or not.

Subscribers don't necessarily have to subscribe themselves. Through the use of the Return Address pattern, one central configuration station could send multiple messages to each publisher specifying which subscriber endpoints to subscribe to which message.

Another option that can be used is for multiple physical subscribers to make themselves appear as one single logical subscriber. This makes it possible to load balance the handling of messages between multiple physical subscribers without any explicit coordination on either the part of the publisher or the part of any one subscriber. All that is needed is for all subscribers to specify the same return address in the subscription message.

#### Publishing

Publishing a message involves having the message arrive at all endpoints which had previously subscribed to that type of message.

Messages which are published often represent events - things that have happened, for instance Order Cancelled, Product Out of Stock, and Shipping Delayed. Sometimes, the cause of an event is the handling of a previous command message, for instance Cancel Order. A publisher is not required to publish a message as a part of handling a command message although it is the simplest solution.

Since many command messages can be received in a short period of time, publishing a message to all subscribers for every command message multiplies the incoming load and, as such, is a less than optimal solution. A better solution would have the publisher roll up all the changes that had occurred in a given period of time into a single published message. The appropriate period of time is dependent on the Service Level Agreement of the publisher - its commitment to the freshness of the data.

For instance, in the financial domain the publishing period may be 10 ms while in the business to consumer e-commerce domain a minute may be acceptable.

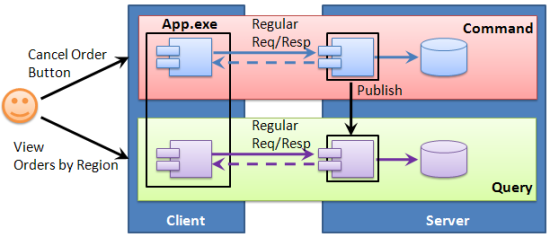
Another advantage of publishing messages on a timer is that that activity can be offloaded from the endpoint/server processing command messages effectively scaling out over more servers.

### Command Query Separation

Many systems provide users with the ability to search, filter, and sort data. While one-way messaging and publish/subscribe are core components of the implementation of these features, the way they are combined is not at all like regular client-server request/response.

In regular client-server development, the server is responsible for providing the client with all CRUD (create, read, update, and delete) capabilities. However, when users look at data they do not often require it to be up to date to the second (given that they often look at the same screen for several seconds to minutes at a time). As such, retrieving data from the same table as that being used for highly consistent transaction processing creates contention resulting in poor performance for all CRUD actions under higher load.

A solution that avoids this problem separates commands and queries at the system-level, even above that of client and server. In this solution there are two "services" that span both client and server - one in charge of commands (create, update, delete), the other in charge of queries (read). These services communicate only via messages - one cannot access the database of the other, as shown in the following diagram:



The command service publishes messages about changes to data, to which the query service subscribes. When the query service receives such notifications, it saves the data in its own data store which may well have a different schema (optimized for queries like a star schema). The query service may also keep all data in memory if the data is small enough.

## Transactions & Message Processing

As a part of the NServiceBus "Fault-Tolerant by Default" design transactions are managed automatically be the infrastructure so that developers don't have to remember the configuration of all threading and state management elements.

### Clients and Servers

While it is desirable for server code to process messages transactionally, it often isn't required for clients, particularly desktop applications.

This is one of the differences between the AsA\_Client and AsA\_Server settings of the [generic host](http://nservicebus.com/GenericHost.aspx) in NServiceBus.

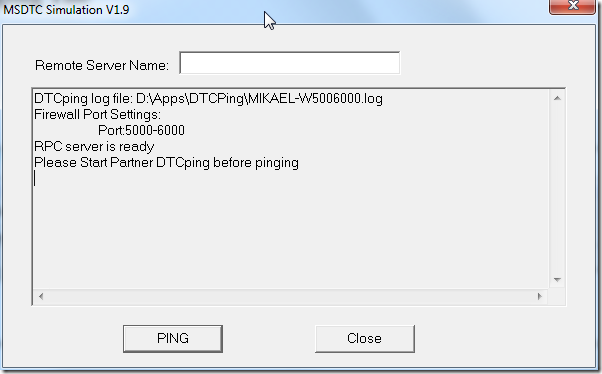
### Specifying transactions in code

If you aren't using the generic host, you can specify whether the current endpoint should process messages transactionally or not by setting the ".IsTransactional(true)" after ".MsmqTransport()"

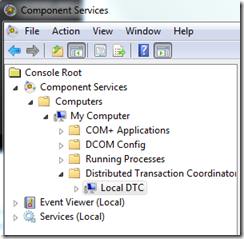
If you want to override the System.Transactions default timeout of 10 minutes, follow the steps described in [this blog post](http://blogs.msdn.com/ajit/archive/2008/06/18/override-the-system-transactions-default-timeout-of-10-minutes-in-the-code.aspx).

### Distributed Transaction Coordinator

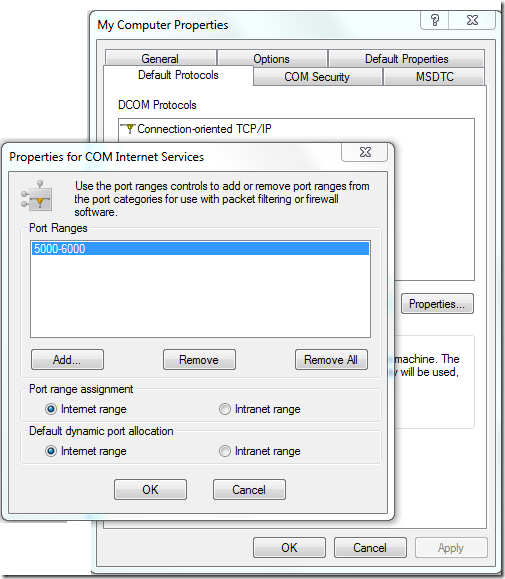
On Windows, there is an OS-level service called the DTC which manages transactions that need to span multiple resources - like queues and databases. This service isn't always configured correctly and may require some trouble-shooting. First of all, you'll need to download a tool called [DTCPing](http://www.microsoft.com/downloads/details.aspx?displaylang=en&FamilyID=5e325025-4dcd-4658-a549-1d549ac17644). This tool will help you find out if one machine is able to access a remote machine over the DTC. Here's what that tool looks like:

[](http://blog.zoolutions.se/image.axd?picture=image.png)

If you get an error referring to the RPC Endpoint Mapper, go to the command prompt and run "dcomcnfg". After you start dcomcnfg you should see the Component Services screen below.

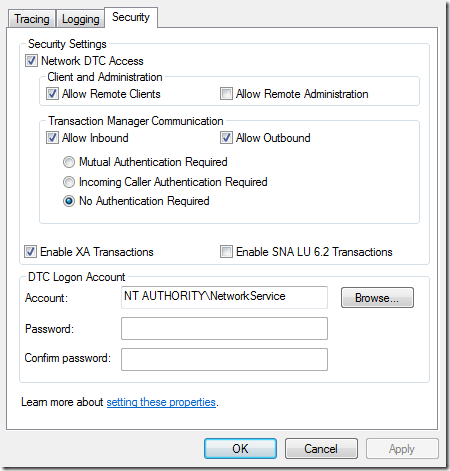
[](http://blog.zoolutions.se/image.axd?picture=dtc_dcomcnfg_1.png)

From here, what needs to be done is open some ports. To do this, right-click "My Computer" and go to the "Default Protocols" tab. From there, select "Connection-oriented TCP/IP" and click the "Properties" button. In the "Properties for COM Internet Services" dialog that opens, check that the Port Ranges includes "5000-6000" as shown in the following image:

[](http://blog.zoolutions.se/image.axd?picture=dtc_dcomcnfg_2.png)

If the list of Port Ranges is empty, click the "Add..." button and enter "5000-6000" in the dialog box that opens up. After doing so, your screen should look like the image above. You can probably make do with less than 1000 open ports, but it depends on the number of machines you are looking to connect to each other over the DTC.

After clicking OK and returning to the Component Services screen, navigate to the "Local DTC" node under the Distributed Transaction Coordinator folder, right click, and select "Properties". In the dialog that opens, select the Security tab as shown below:

[](http://blog.zoolutions.se/image.axd?picture=dtc_dcomcnfg_3.png)

Ensure that the properties you see are the same as the above.

After finishing all the steps above, restart the computer.

If DTCPing isn't working after all of the above, check that the needed ports are open in the firewall. Consider removing the DTC exceptions in the firewall and add them again to make sure.

If DTCPing gives you a message about finding the name but not reaching it, the first thing to do is a simple ping by running “ping computer\_name” in the command prompt. If you discover that the machine cannot be reached by ping, it could be that you have a DNS problem that may require your Network Administrator's help.

Make sure you perform all of the above steps not just on the servers that connect to the database, but also on the database servers as well.

Finally, check the TCP ports in use on the servers making sure that each has a different port configured as the communication is bi-directional. At this point, you should be able to run transactional NServiceBus endpoints.

Content in this section derived from [Mikael Henriksson's](http://blog.zoolutions.se/post/2010/04/01/Conquering-NServiceBus-part-5-e28093-Troubleshooting-DTC.aspx) blog.

### The Message Processing Loop

Messages are processed in NServiceBus as follows:

1. The queue is peeked to see if there's a message.
2. If so, a distributed transaction is started.
3. The queue is contacted again to receive a message. This is because multiple threads may have peeked the same message. The queue makes sure only one thread will actually get a given message.
4. If the thread is able to get it, NServiceBus tries to deserialize the message. If this fails, the message is moved to the configured error queue and the transaction commits.
5. After a successful deserialization, NServiceBus invokes all infrastructure and applicative message modules and handlers. An exception in this step will cause the transaction to rollback and the message to return to the input queue.
   * This will happen the "MaxRetries" configurable number of times
   * After that, the message will be moved to the configured error queue

In this manner, even under all kinds of failure conditions like the application server restarting in the middle of a message, or a database deadlock, messages are not lost.

In the common case, the automatic retry mechanism is able to recover from most temporary problems. When that isn't possible, the message ends up in the error queue.

### Resolving More Permanent Errors

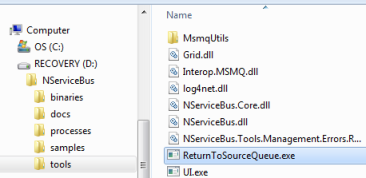
There are multiple situations in which more permanent errors affect systems. Despite their diversity, the NServiceBus solution is the same. Before describing it, let's drill into some of these situations.

1. The database is down
2. An external or internal web service is down
3. The system was upgraded accidentally breaking backwards-compatibility

In all of the above, administrative action is needed - from things as simple as bringing up a database or web service again, to more complex actions like reverting to the previous version of the system.

What we see here is that there's nothing necessarily wrong with the message itself. It might contain valuable information that shouldn't get lost under these conditions. Therefore, after the administrator finishes resolving the issue, they should return the message to the queue it came from. Luckily, NServiceBus comes with a tool that does exactly that.

### ReturnToSourceQueue.exe



You can find this tool in the "Tools" directory found in your NServiceBus download.

Administrators provide the name of the error queue they'd like to use, and either specify a specific message ID to return to its source queue, or 'all' to return all messages in the given error queue, each to its respective source queue.