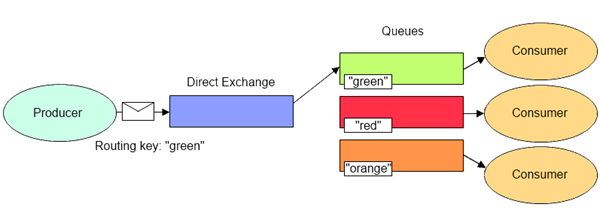
**Exchange Types**

Exchanges control the routing of messages to queues.  Each exchange type defines a specific routing algorithm which the server uses to determine which bound queues a published message should be routed to.

RabbitMQ provides four types of exchanges: Direct, Fanout, Topic, and Headers.

**Direct Exchanges**

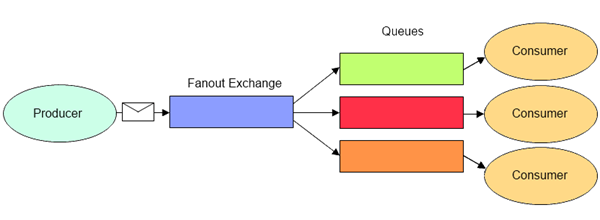
The Direct exchange type routes messages with a routing key equal to the routing key declared by the binding queue.  The following illustrates how the direct exchange type works:

[](http://lostechies.com/derekgreer/files/2012/03/DirectExchange1.png)

The Direct exchange type is useful when you would like to distinguish messages published to the same exchange using a simple string identifier.  This is the type of exchange that was used in our Hello World example.  As discussed in part 3 of our series, every queue is automatically bound to a default exchange using a routing key equal to the queue name.  This default exchange is declared as a Direct exchange.  In our example, the queue named “hello-world-queue” was bound to the default exchange with a routing key of “hello-world-queue”, so publishing a message to the default exchange (identified with an empty string) routed the message to the queue named “hello-world-queue”.

**Fanout Exchanges**

The Fanout exchange type routes messages to all bound queues indiscriminately.  If a routing key is provided, it will simply be ignored.  The following illustrates how the fanout exchange type works:

[](http://lostechies.com/derekgreer/files/2012/03/FanoutExchange2.png)

The Fanout exchange type is useful for facilitating the [publish-subscribe pattern](http://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern).  When using the fanout exchange type, different queues can be declared to handle messages in different ways.  For instance, a message indicating a customer order has been placed might be received by one queue whose consumers fulfill the order, another whose consumers update a read-only history of orders, and yet another whose consumers record the order for reporting purposes.

**Topic Exchanges**

The Topic exchange type routes messages to queues whose routing key matches all, or a portion of a routing key.  With topic exchanges, messages are published with routing keys containing a series of words separated by a dot (e.g. “word1.word2.word3”).  Queues binding to a topic exchange supply a matching pattern for the server to use when routing the message.  Patterns may contain an asterisk (“\*”) to match a word in a specific position of the routing key, or a hash (“#”) to match zero or more words.  For example, a message published with a routing key of “honda.civic.navy” would match queues bound with “honda.civic.navy”, “\*.civic.\*”, “honda.#”, or “#”, but would not match “honda.accord.navy”, “honda.accord.silver”, “\*.accord.\*”, or “ford.#”.  The following illustrates how the fanout exchange type works:

[](http://lostechies.com/derekgreer/files/2012/03/TopicExchange2.png)

The Topic exchange type is useful for directing messages based on multiple categories (e.g. product type and shipping preference ), or for routing messages originating from multiple sources (e.g. logs containing an application name and severity level).

**Headers Exchanges**

The Headers exchange type routes messages based upon a matching of message headers to the expected headers specified by the binding queue.  The headers exchange type is similar to the topic exchange type in that more than one criteria can be specified as a filter, but the headers exchange differs in that its criteria is expressed in the message headers as opposed to the routing key, may occur in any order, and may be specified as matching any or all of the specified headers.  The following illustrates how the headers exchange type works:

[](http://lostechies.com/derekgreer/files/2012/03/HeadersExchange2.png)

The Headers exchange type is useful for directing messages which may contain a subset of known criteria where the order is not established and provides a more convenient way of matching based upon the use of complex types as the matching criteria (i.e. a serialized object).

**Conclusion**

That wraps up our introduction to each of the exchange types.  Next time, we’ll walk through an example which demonstrates declaring a direct exchange explicitly and take a look at the the push API.

**AMQP** ([Advanced Message Queueing Protocol](https://en.wikipedia.org/wiki/Advanced_Message_Queuing_Protocol)) is a network protocol that can deliver messages from one application endpoint to another application endpoint. It does not care about the platform or language of said applications, as long as they support AMQP.

Essentially, one of the application endpoints sends a message, thus being a Producer, through an AMQP broker. In turn the broker will deliver the message to another application endpoint, called the Consumer.

[**RabbitMQ**](https://www.rabbitmq.com/) is an AMQP broker that has support for several programming languages, such as PHP.

The advantage of having a message broker such as RabbitMQ, and AMQP being a network protocol, is that the producer, the broker, and the consumer can live on different physical/virtual servers on different geographic locations.

Also, since networks are unreliable, and applications might fail to process a message completely, AMQP supports message acknowledgements, either automatically or when an application endpoint decides to send them.

RabbitMQ implements the [*AMQP 0-9-1*](https://www.rabbitmq.com/resources/specs/amqp0-9-1.pdf) protocol. At a glance, it follows the following model.

# Transport, Transport, … MSMQ, RabbitMQ, ActiveMQ with NServiceBus

standard

| [February 27, 2014](http://www.planetgeek.ch/2014/02/27/transport-transport-msmq-rabbitmq-activemq-with-nservicebus/) by [Daniel Marbach](http://www.planetgeek.ch/author/danielmarbach/) | [.NET](http://www.planetgeek.ch/category/net/), [Presentation](http://www.planetgeek.ch/category/presentation/), [Software](http://www.planetgeek.ch/category/software/) |

[6 Comments](http://www.planetgeek.ch/2014/02/27/transport-transport-msmq-rabbitmq-activemq-with-nservicebus/#comments)

This is the handout for the presentation I gave at the Basta Conference in Darmstadt. The talk is about different transports available for NServiceBus. Building distributed systems with NServiceBus which only use MSMQ was yesterday. Today we have more possibilities to choose the right transport technology. But how do I decide which transport layer fulfills my functional and non-functional requirements? What is the difference between MSMQ and RabbitMQ? How does a broker based transport differentiate itself from the store and forward transports? Which consistency guarantees do I get from the different transport technologies? In this talk we dive into the pros and cons of the different transports available for NServiceBus, primarily MSMQ, RabbitMQ and ActiveMQ. Why does the transport technology even matter? This reminds me of…

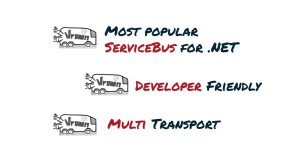
[](http://www.planetgeek.ch/wp-content/uploads/2014/02/Slide2.jpg)

I’m working in a legacy distributed system written in .NET which primarily uses Remote Procedure style communication, in fact it uses a mixture of classic .NET Remoting and Windows Communication Foundation. In order to fulfill new scalability requirements but also to decouple the big ball of mud we introduced NServiceBus into the game. Our server infrastructure was Windows based and therefore we simply started with MSMQ as a transport layer. Because our legacy code base had a hand-written transaction and database layer, we needed a robust transaction management between the queue and the database operations. At first MSMQ seemed to be the right choice, because it supports distributed transactions (two-phase commit protocol) natively which can span the queue and the database. Sometime later we were faced with new business requirements. We needed to integrate with other systems in the corporate network. Those systems were built in Java, running on Linux servers and already used ActiveMQ as a transport layer. We needed to answer a lot of questions: How do we interop with the Java side based upon messaging? Could we somehow bridge between MSMQ and ActiveMQ? … We decided to switch from MSMQ to ActiveMQ because it supported distributed transactions, until we found out that it didn’t… Fast forward later we lost over 20’000 invoice messages in the datacenter.

This war story shows that is crucially important to carefully pick your transport layer based on functional and non-functional requirements. Whether you need to interop between different programming languages or server environments, what kind of consistency guarantees are required, how much throughput you need and more influences the transport layer selection process.

[](http://www.planetgeek.ch/wp-content/uploads/2014/02/Slide3.jpg)

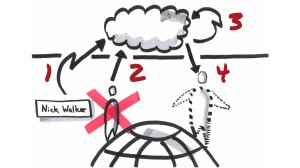
Before we explore the transport level choices which come with NServiceBus I want to briefly cover some basics of NServiceBus and explain the sample project which will guide us through this presentation.

[](http://www.planetgeek.ch/wp-content/uploads/2014/02/Slide4.jpg)

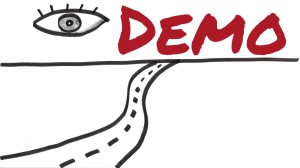
NServiceBus is the most popular ServiceBus for .NET. With over 250,000 downloads and thousands of production deployments, companies of all sizes and across all verticals count on NServiceBus. It supports multiple transports such as MSMQ, RabbitMQ, ActiveMQ, Azure and Sql Server. It is extremely developer friendly and a breeze to start with. You don’t believe me?

[](http://www.planetgeek.ch/wp-content/uploads/2014/02/Slide5.jpg)

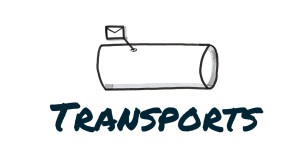
Before I can show you the awesomeness of NServiceBus I want to briefly cover the sample which guides us through this presentation. We will be using a domain from the supernatural comedy film R.I.P.D. The film was released in 2013 and directed by Robert Schwentke based on the comic book Rest In Peace Department by Peter M. Lenkov. The protagonist Ryan Raynolds as Nick Walker is a detective from the Boston Police Department which gets killed during a raid on a warehouse. After ascending to the limbo in the afterlife Nick is taken into the office of Mildred Proctor / director of the Boston divison of the Rest In Peace Department (R.I.P.D). Proctor explains that R.I.P.D. is an agency that recruits deceased police officers to patrol the afterlife and capture “deados”; spirits that failed to cross over and instead stay on Earth as monsters. Nick agrees to join R.I.P.D. and gets sent back to Earth with his new partner Roy Pulsipher, played by Jeff Bridges. The message flow could look like this

[](http://www.planetgeek.ch/wp-content/uploads/2014/02/Slide6.jpg)

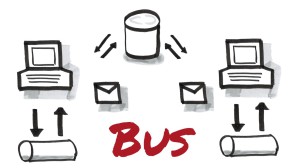
When Nick Walker dies an event called “PoliceOfficerDied” (number 1 in the picture above) is published. This gives the R.I.P.D. time to prepare the registration and recruitement process before Nick ascends to heaven. When the soul ascends to afterlife a command “AscendToHeaven”(number 2 in the picture above) is sent. After ascension the R.I.P.D. eventually recruits Nick Walker. When the recruitement process is finished an event “DeadPoliceOfficerRecruited” (number 3 in the picture above) indicates that. Right after the recruitement process Nick gets sent back to earth to hunt the deados left on earth, which is indicated by the command “HuntDeadosOnEarth” (number 4 in the picture above).

[](http://www.planetgeek.ch/wp-content/uploads/2014/02/Slide7.jpg)

Let us build that crazy sample with NServiceBus.

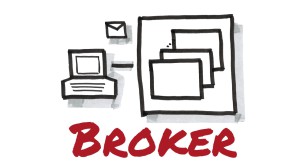
[](http://www.planetgeek.ch/wp-content/uploads/2014/02/Slide8.jpg)

Finally we can start talking about transports. Before we dive into the individual transports I want to briefly cover some general aspects which I think are important to consider when choosing a transport for your project. The first aspect is the **transport style**.

[](http://www.planetgeek.ch/wp-content/uploads/2014/02/Slide9.jpg)

The bus is an architectural style. The bus architectural style is pretty difficult to draw on a paper. Often it is drawn as a centralized line or pipe. This representation is misleading. In a bus architecture there is no physical Ethernet box that things connect to remotely – the bus is physically everywhere. In contrast to the widely used *broker architectural style*, which we’ll discuss next, the bus does not centralize routing, data transformation or orchestration. In order to achieve such a decentralized architecture style we need the store and forward pattern. A publisher first looks into his subscription store (database symbol in the picture above) whether there is any interested subscriber registered. If there is a subscription present, the publisher first stores the message into his local outgoing queue. The messages remains in the outgoing queue as long as the remote machine is not available. As soon as the subscriber machine is available the message gets transferred to the ingoing queue of the subscriber. When the subscriber process is not online, the message remains in the ingoing queue. Finally when the subscriber is online it gets picked from the ingoing queue and processed. Because of the decentralized nature of this architecture style it, is extremely robust and easy to scale but usually harder to manage than a centralized style.

*Note: For simplicity the picture above shows only one subscription database which might imply a centralized architecture style. In reality we normally have multiple independent subscription database per service.*

[](http://www.planetgeek.ch/wp-content/uploads/2014/02/Slide10.jpg)

In contrast to the decentralized bus architecture style the broker architecture style uses a centralized “hub and spoke” or “mediator” design. The initial intend of brokers was to avoid having to change apps. The broker is a physically separate process which all communication is routed through. The broker itself handles fail over, routing and more. It is a single point of failure and therefore must be designed robust and for high-performance. Because all communication is concentrated to a single logical entity the broker enables central management as well as intelligent routing, data transformation and orchestration. Reconfiguration usually only happens on the broker and if well designed, doesn’t require changes to the deployed applications. This could potentially prevent applications from gaining autonomy and often leads to procedural programming at large scale because changes to the broker are not acceptance tested or under source control.

Each architecture style comes with its own pros and cons and have to be carefully discussed with IT operations (hopefully DevOps J ) and weighted against non-functional requirements. But it is also possible to use a combination of both bus and brokers architecture style. The next aspect is **speed**.

[](http://www.planetgeek.ch/wp-content/uploads/2014/02/Slide11.jpg)

*Disclaimer first: I’m not going to give away any speed related numbers like XYTransport can handle X1000’s of messages per second. These numbers are extremely dependent upon the hardware available and how the individual transport is configured.*

Depending upon the system you are going to build the speed (number of messages per second) of a transport can be important. But speed as a general criteria is just too nearsighted. There are a lot of criteria to be considered if you compare the speed of a transport. For example how long does it take from the publisher to the broker, how long does it take until the broker confirmed having written the message to the disk, how long does the consumer process have to consume an individual message, how many external resources are involved in the consumer or publisher process (especially IO access) and many more. I strongly advice to never trust any performance numbers mentioned on the internet. Carefully spike and load test different transports for your business scenarios! The following (incomplete) list of questions can help:

* What IO resources are involved in a typical business scenario
* Do we need to wait for the transport to confirm the disk write (fsync)
* Do we even need to write the message to the disk
* Do we need transactional guarantees or can we risk losing messages
* Can we solve performance issues with vertical or horizontal scaling (better/faster disk, disk caching, more servers…)
* Are the performance requirements really necessary or just nice to have
* Do we need to solve every scenario with messaging (sometimes another technology is more suitable, never forget that!)

Ask yourself those questions for each business scenario. There is no general answer for the whole system if you want maximum possible performance for each scenario. This brings us to the next aspect **durability and consistency**.

[](http://www.planetgeek.ch/wp-content/uploads/2014/02/Slide12.jpg)

This goes hand in hand with the speed criteria. More consistency and durability (for example write confirmation from multiple nodes) has great impact on the speed of a transport. During the evaluation process of a transport you should ask yourself the following (incomplete) list of questions:

* Does the transport support native transactions
* Does the transport support distributed transactions
* How does it handle durability to disk, can it be controlled for each message type
* Is it possible to disable write confirmation
* Can it operate completely in memory

Always try to avoid distributed transactions like XA with JTA under Java/Linux or DTC under Windows (two-phase commit protocol) if possible. Especially when you are working in a Greenfield system. Design the process so that messages or their consumptions are idempotent. I’m not going to cover message idempotency in this presentation. For larger brown field applications it can sometimes be helpful, for example during the refactoring and redesign process, to be able to rely on distributed transactions. The distributed transactions will then span transactions over multiple IO resources such as the transport itself and involved databases (if they support two-phase commit). But distributed transactions have large impact on the processing speed and are cumbersome to manage when they fail.

[](http://www.planetgeek.ch/wp-content/uploads/2014/02/Slide13.jpg)

Scale out is not only important when the system gets bigger and bigger but also for high-availability. For example if you need to update your broker you want the system still be operational. Brokers usually come with a large number of features which allow to scale out or make it high available. For example there are operation modes like broker clustering, master slave, network of brokers and more. Never forget to consider good old load balancers!

[](http://www.planetgeek.ch/wp-content/uploads/2014/02/Slide14.jpg)

Besides all the transport “featureitis” this point is often forgotten. But believe me: You absolutely need good operational tools to detect failures, bulk move and copy messages between queues and monitor your message system.

[](http://www.planetgeek.ch/wp-content/uploads/2014/02/Slide15.jpg)

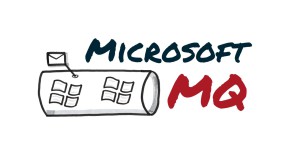
No matter if you do store and forward, broker based transport or both in combination. You often need to integrate the transport into existing authorization and authentication solutions such as ActiveDirectory and LDAP. Depending upon your security requirements you might need to tunnel over SSL. Especially with broker based transports you want to isolate applications running on the same broker infrastructure from each other. By default each application running on the same broker can read and write to all queues on the broker.

[](http://www.planetgeek.ch/wp-content/uploads/2014/02/Slide16.jpg)

When your project starts it is often a homogenous environment. The longer the project runs the more it needs to interop with other programming languages and platforms. Of course it is perfectly valid to restrict the system architecture to a certain operating system. But if the transport supports interoperability between programming languages and operating systems by default, you might save a lot of effort in the future. As always with architecture, everything is a tradeoff!

[](http://www.planetgeek.ch/wp-content/uploads/2014/02/Slide17.jpg)

If you don’t want to suffer from vendor lockin and built upon messaging standards for better interoperability, you have to take AMQP (Advanced Message Queuing Protocol) support into consideration. The AMQP standard is an open standard protocol for messaging originally develop by JP Morgan Chase and iMatix (development started in 2003). It is a wire level protocol like HTTP and totally vendor agnostic. It can be used from client to broker but also from broker to broker communication. It is adopting more and more general acceptance. It has a couple of key concepts like Message Broker, Exchanges (Direct, Fan-out, Topic and Headers), Queues and Bindings. I’m going to briefly cover those key concepts during the RabbitMQ demo.

[](http://www.planetgeek.ch/wp-content/uploads/2014/02/Slide18.jpg)

Microsoft Message Queuing (MSMQ) is developed by Microsoft and available since 1997 on every Windows platform (has to be installed as Windows Component). The latest MSMQ standard was released with Windows 7 and Windows Server 2008 R2 and has the version number 5.0.

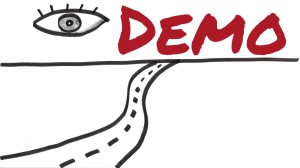
[](http://www.planetgeek.ch/wp-content/uploads/2014/02/Slide19.jpg)

It provides Active Directory integration, does not support AMQP and uses the store and forward model. It supports native transactions as well as distributed transaction with distributed transaction coordinator integration. Messages are written to disk on the local machine. In order to not lose messages in case of disk failure the messages need to be written to highly available storage.

[](http://www.planetgeek.ch/wp-content/uploads/2014/02/Slide20.jpg)

Native Interoperability is quite limited because MSMQ only runs under Windows Operating systems. It is reasonably fast when used with managed code (available in the .NET framework) and even faster when combined with unmanaged code. Scaling and high availability can be achieved with building an MSMQ cluster and/or using the distributor provided by NServiceBus. Administration of MSMQ is fairly easy. It can be installed, managed and configured with Powershell, in the Microsoft Management Console (MMC) or with third party tools such as ServiceInsight from Particular Software, QueueExplorer from Cogin or other advanced monitoring tools such as NewRelic or AppDynamics.

Let’s see it in action.

[](http://www.planetgeek.ch/wp-content/uploads/2014/02/Slide21.jpg)

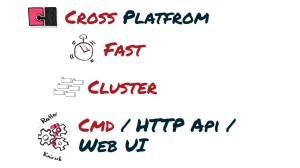
The next transport we’ll cover is RabbitMQ.

[](http://www.planetgeek.ch/wp-content/uploads/2014/02/Slide22.jpg)

RabbitMQ is an open source messaging-oriented middleware written in Erlang. It leverages the Open Telecom Platform framework from Erlang for clustering and failover support. It is developed and supported by Rabbit Technologies Ltd. which started as a joint venture between LShift and CohesiveFT in 2007. In 2010 it was acquired by SpringSource which is a division of VMWare. The currently released major number is 3.0.

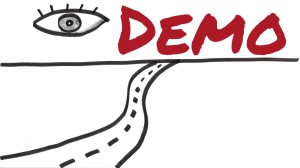
[](http://www.planetgeek.ch/wp-content/uploads/2014/02/Slide23.jpg)

RabbitMQ has the notion of vhosts (virtual hosts) which allows to separate different applications running on the same broker. Authorization and permission is built in into the broker. RabbitMQ supports AMQP specification 0.9.1 out of the box and AMQP 1.0 via a plugin. It is a broker based transport and by default optimized for speed. That means if you don’t tell RabbitMQ to do it otherwise, RabbitMQ will store many information in-memory. When you restart the broker all the information will be lost. Of course this can be tuned to your needs by defining messages, queues and exchanges durable, which will instruct RabbitMQ to write it to its internal database called Mnesia. RabbitMQ supports limited native transactions (for example there is no cross-queue transactions) but it is recommended to use publisher confirms.

[](http://www.planetgeek.ch/wp-content/uploads/2014/02/Slide24.jpg)

RabbitMQ is truly cross-platform. It runs everywhere where Erlang can be hosted. It offers a wide range of client libraries for different programming languages. As previously described it is tuned for performance out of the box. It can be clustered extremely easy based on the Open Telecom Platform framework from Erlang. By default it is administrated with the RabbitMQ command line interface, via the HTTP API or the WebUI.

Let’s see it in action.

[](http://www.planetgeek.ch/wp-content/uploads/2014/02/Slide25.jpg)

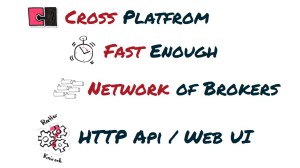
The last transport we’ll cover is ActiveMQ.

[](http://www.planetgeek.ch/wp-content/uploads/2014/02/Slide26.jpg)

ActiveMQ is an open source messaging-oriented middleware written in Java. Development startet around 2003 by a group of open source developers who built Apache Geronimo. The currently released major number is 5.0.

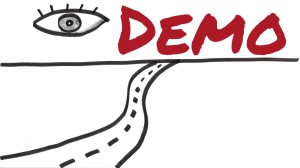
[](http://www.planetgeek.ch/wp-content/uploads/2014/02/Slide27.jpg)

As ActiveMQ is fully developed in Java it leverages the security capabilities of the underlying JVM platform. Namely it fully integrates with Java SE Security model. It is a broker based transport and the broker is AMQP 1.0 compliant. By default messages are saved to the internal database called KahaDB. In the future the default storage will be LevelDB. It supports native transactions. The .NET driver implementation claimed to support distributed transactions which doesn’t hold true as we found out!

[](http://www.planetgeek.ch/wp-content/uploads/2014/02/Slide28.jpg)

Because it is based upon Java it is truly cross platform. It is also fully JMS 1.1 specification compliant which allows to integrate ActiveMQ into every application server in the wild. It is reasonably fast for most applications. It can be “clustered” by buildup a Network of Brokers. It can be management with a WebUI or the provided HTTP API.

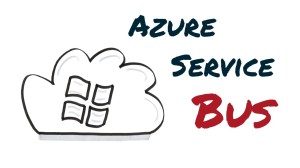
Let’s see it in action.

[](http://www.planetgeek.ch/wp-content/uploads/2014/02/Slide29.jpg)

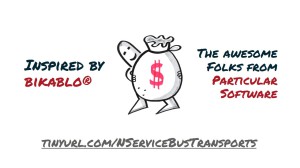
Before we wrap up I want to briefly mention that the NServiceBus ecosystem contains even more transports.

[](http://www.planetgeek.ch/wp-content/uploads/2014/02/Slide30.jpg)

Queues and messages are stored inside SqlServer. The cool thing is that you can use the same Sql Server transactions for the business data and the messages. No need for two-phase commit protocol as long as you go with one database. But the transport also supports multiple databases. This then requires DTC.

[](http://www.planetgeek.ch/wp-content/uploads/2014/02/Slide31.jpg)

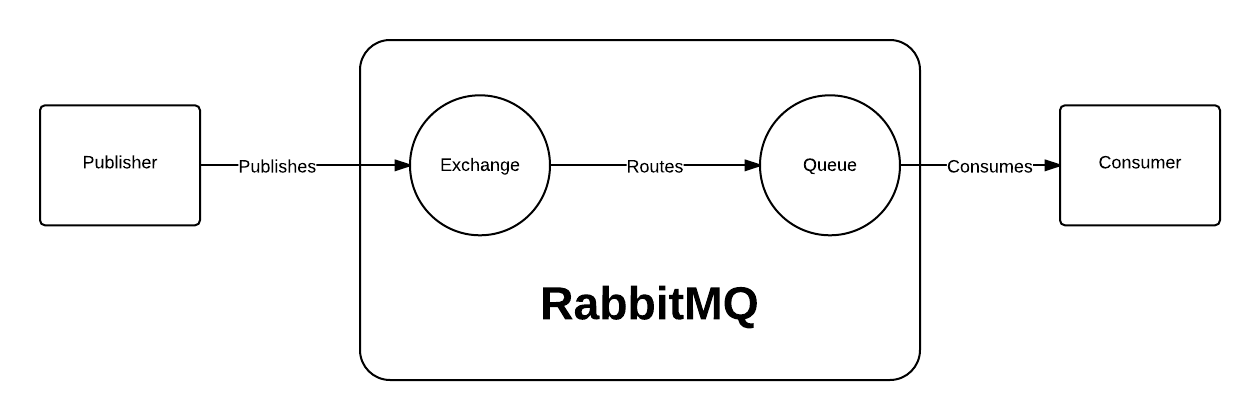
This allows hosting NServiceBus based solutions in the Azure cloud. The interop and reach for this transport is massive because Azure operates on global scale.

[](http://www.planetgeek.ch/wp-content/uploads/2014/02/Slide32.jpg)

All the graphics in this presentation are hand drawn by myself and heavily influenced by the Bikablo® technique. I also thank the awesome folks from Particular Software which are always here for help and support, and this around the clock!

## References

* RabbitMQ  
  <http://www.rabbitmq.com/>
* RabbitMQ in Action – Distributed messaging for everyone by Alvaro Videa and Jason J.W. Williams, Manning  
  <http://manning.com/videla/>
* ActiveMQ  
  <http://activemq.apache.org/>
* ActiveMQ in Action by Bruce Snyder, Dejan Bosanac and Rob Davies, Manning  
  <http://manning.com/snyder/>
* Advanced Message Queuing Protocol  
  <http://www.amqp.org/>
* Building an MSMQ cluster  
  <http://blogs.msdn.com/b/mismail/archive/2007/04/18/building-an-msmq-cluster.aspx>  
  How to Cluster Message Queuing  
  <http://msdn.microsoft.com/en-us/library/dd897482(v=bts.10).aspx>
* Message Queuing  
  <http://technet.microsoft.com/en-us/library/cc732184.aspx>
* NServiceBus  
  [http://www.particular.net](http://www.particular.net/)
* R.I.P.D  
  <http://en.wikipedia.org/wiki/R.I.P.D>.
* Transport.Transport Demo  
  <http://github.com/danielmarbach/Transport.Transport>



## Glossary

| **Concept** | **Definition** | **Icon** |
| --- | --- | --- |
| Producer | Application endpoint that sends messages | Producer icon |
| Consumer | Application endpoint that receives messages | Consumer icon |
| Connection | Handles protocol, errors, authentication, etc… The connection is done using TCP protocol. | – |
| Channel | Connections are multiplexed through channels. Even though all channels share the same tcp connection, communication from one channel is completely independent of another. | – |
| Exchange | Receives messages from producers and pushes them to queues. Depending on the situation, this can be transparent to the developer. | Exchange representation |
| Queue | Buffer that stores messages | Queue icon |
| Message | Piece of arbitrary information conforming to the AMQP format that is sent from the producer to a consumer through the broker. The broker cannot modify the information inside the message. | – |
| Acknowledgements | Notice sent back from the consumer to tell the server that a message has been received and processed, so the server can delete it from the queue. | – |

Another advantage of AMQP 0-9-1 is that the application defines the routing logic instead of a broker administrator. This gives the developer a lot of flexibility, without the need to learn a new programming/scripting/markup language.

You can learn more about AMQP and RabbitMQ at the “[AMQP 0-9-1 Model Explained](https://www.rabbitmq.com/tutorials/amqp-concepts.html)” guide. Although not necessary for this tutorial, I encourage you to read it completely.

