|  |  |
| --- | --- |
| **mslogo-1.jpg** | **PluralsightColorLogoLgTrans.png** |

A Developer’s Guide to the Microsoft® .NET Service Bus

Making Software + Services a Reality on the Windows Platform

Aaron Skonnard, Pluralsight

December 2008

Contents

[Abstract 3](#_Toc216846638)

[An Overview of the .NET Service Bus 3](#_Toc216846639)

[The Enterprise Service Bus Pattern 4](#_Toc216846640)

[Moving towards an Internet Service Bus 5](#_Toc216846641)

[Introducing the .NET Service Bus 7](#_Toc216846642)

[Your First .NET Service Bus Solution 8](#_Toc216846643)

[Naming and Discovery 14](#_Toc216846644)

[Naming System 15](#_Toc216846645)

[Service Registry 17](#_Toc216846646)

[Messaging 20](#_Toc216846647)

[The Relay Service 20](#_Toc216846648)

[Listeners and Naming 21](#_Toc216846649)

[Ports Required by the Relay Service 21](#_Toc216846650)

[WCF Relay Bindings 22](#_Toc216846651)

[NetOnewayRelayBinding 24](#_Toc216846652)

[NetEventRelayBinding 28](#_Toc216846653)

[NetTcpRelayBinding 31](#_Toc216846654)

[HTTP Relay Bindings 34](#_Toc216846655)

[Access Control 39](#_Toc216846656)

[Authentication and Authorization 39](#_Toc216846657)

[WCF Programming Model for Access Control 41](#_Toc216846658)

[Web-Style (REST) Authentication 42](#_Toc216846659)

[Guidance 44](#_Toc216846660)

[Summary 45](#_Toc216846661)

[Additional Resources 45](#_Toc216846662)

[Microsoft® .NET Services Whitepaper Series 45](#_Toc216846663)

[.NET Service Bus Resources 46](#_Toc216846664)

[About the Author 46](#_Toc216846665)

[Acknowledgements 46](#_Toc216846666)

# Abstract

As applications begin moving towards the cloud computing platform offered by Windows® Azure, they will need the ability to bridge their existing on-premise software assets any new assets running in the cloud, a strategy commonly referred to as *software + services* today. This type of integration can be accomplished through standard protocols assuming bidirectional connectivity is possible. While it’s usually simple to initiate communication from on-premise applications to services running in the cloud, the reverse is more difficult because on-premise software is usually running behind multiple firewalls or NAT devices. The Microsoft® .NET Service Bus addresses this problem space by making it easy to create secure bidirectional communication channels between on-premise software and services in the cloud.

# An Overview of the .NET Service Bus

Microsoft® .NET Services[[1]](#footnote-2) are a set of highly scalable developer-oriented services running in Microsoft data centers as part of the Azure™ Services Platform. Microsoft .NET Services provides developers with common building blocks and infrastructure services for cloud-based and cloud-aware applications. Much like you rely on the .NET Framework for common building blocks when developing on-premise software, you will rely on Microsoft® .NET Services for the common building blocks in your cloud applications.

The Microsoft® .NET Service Bus is one of the core service offerings found within Microsoft® .NET Services. Today it’s complemented by two other services: the Microsoft® .NET Access Control Service and the Microsoft® .NET Workflow Service. The .NET Service Bus relies on the Access Control Service for controlling access to your solutions through a claims-based security model. The .NET Workflow Service allows you to define cloud-based workflows that model service interactions through the .NET Service Bus. Together these services provide a valuable development fabric required by most cloud applications, thereby simplifying cloud development by allowing you to focus more directly on business needs.[[2]](#footnote-3)

This whitepaper focuses on how you can use the .NET Service Bus to overcome some of the common Internet connectivity challenges inherent in today’s world. First, we’ll discuss the motivation for the .NET Service Bus, the architectural pattern it embodies, and the key aspects of the architecture you’ll need to understand. Then, after taking a quick lap around your first .NET Service Bus application, we’ll dive deeper into each area of the .NET Service architecture, explore the different messaging options it offers, and discuss how it integrates with the .NET Access Control Service. And finally, towards the end of the paper we’ll provide some guidance on how to best use the .NET Service Bus in your applications.

## The Enterprise Service Bus Pattern

There’s a reason Microsoft decided to call what we’re talking about here the .NET Service Bus. The “Service Bus” moniker was chosen to emphasize certain characteristics of its underlying architecture. These architectural characteristics are actually quite common today through the various product incarnations of the *Enterprise Service Bus* (ESB) architectural pattern. The ESB pattern defines a model for integrating enterprise applications through a common messaging fabric or “bus” (see Figure 1).

The ESB pattern typically calls for a federated identity and access control mechanism, a common service naming mechanism, a discoverable service registry, and a common messaging fabric that provides a variety of communication options. This model introduces a level of indirection between the various services found on the bus, and between composite applications and the services they consume. The ESB pattern helps broker differences across services in terms of identity management, naming conventions, message formats, and communication protocols. Once a service gets on the bus, anything else on the bus can connect to it even if it wouldn't normally be able to communicate with the service directly.

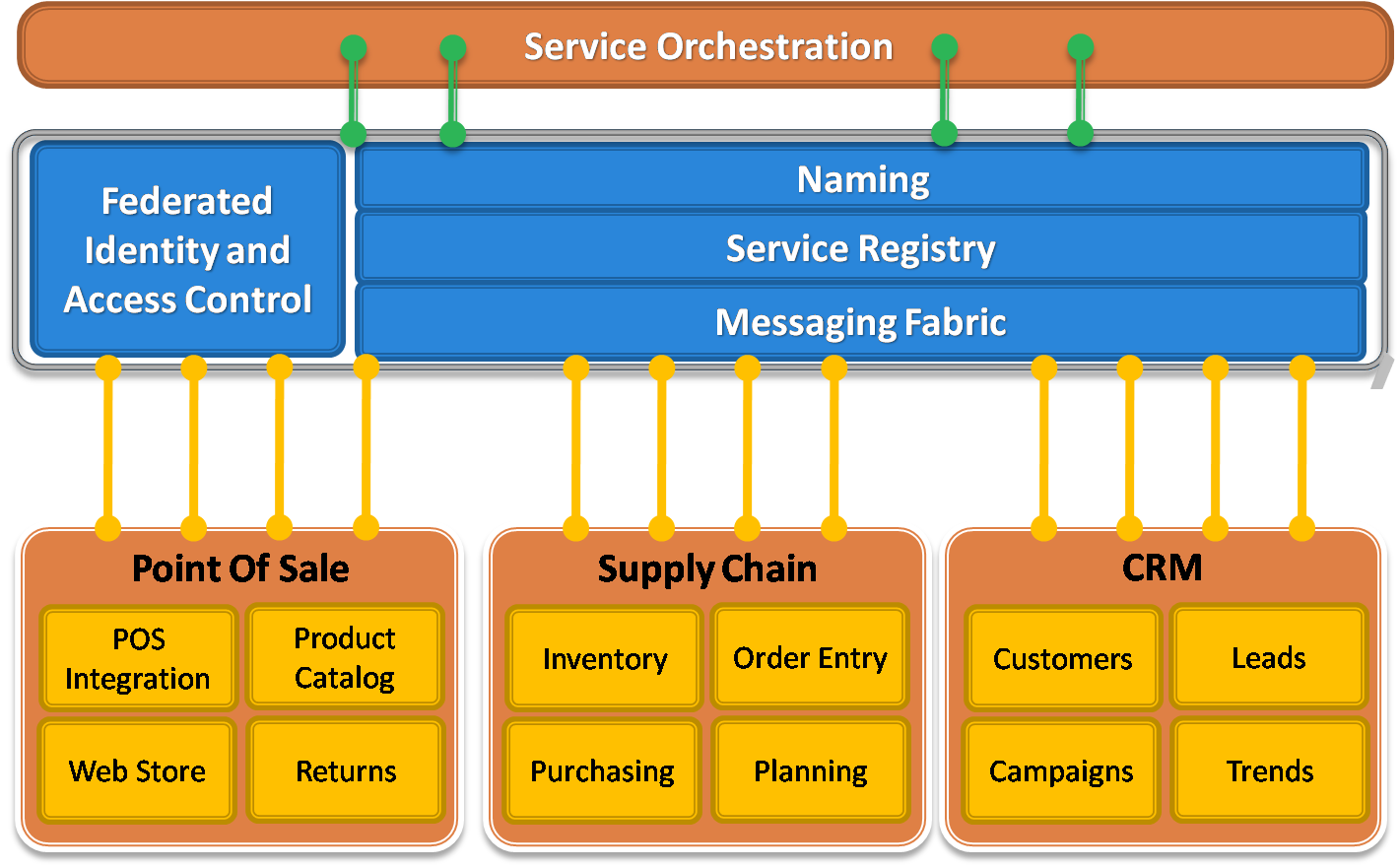


Figure : The Enterprise Service Bus Pattern

The ESB pattern has grown in popularity over the years because it simplifies the management of multiple service connections. One way it does this is by enabling *publish/subscribe* architectures, which provides for even looser-coupling throughout an enterprise. With a publish/subscribe architecture, consumers no longer have to be directly connected to services anymore – everything just needs a connection to the bus – and nodes can be added or removed at will over time. This type of architecture also enables event distribution. You can have multiple subscribers to a single message type, and you can also have multiple publishers of that same message type, giving you N-to-N multicast capabilities.

Most ESB environments are enhanced with a service orchestration layer that provides a process flow engine for orchestrating the messaging interactions that make up a business process or “workflow”.

There are a variety of different products and technologies that can be used to implement an ESB including things like Active Directory, UDDI, BizTalk Server, MSMQ and WCF, and there are many equivalent offerings available on other platforms as well. In most ESB environments, the messaging fabric is simply rooted in Ethernet and TCP/IP. This raw messaging foundation is then enhanced by a variety of server products and messaging technologies that provide different messaging semantics.

You might use MSMQ for durable asynchronous messaging semantics or BizTalk Server when you want to employ a hub-and-spoke integration model. You could also use peer-to-peer messaging techniques through WCF, or a more RESTful model built primarily in terms of HTTP communication. The ESB messaging fabric can support all of these messaging options and abstract their differences away.

The ESB pattern has proven valuable for managing on-premise applications, however, as more software assets begin to move towards the cloud, a new set of challenges arise that must be addressed.

## Moving towards an Internet Service Bus

Microsoft has been actively working on making the ESB pattern a reality at Internet scope. When you consider what this would entail, you’ll realize that the same ESB architectural components are going to be necessary, specifically identity and access control, naming, a service registry, and a common messaging fabric. The primary difference is one of scope: in this case the various ESB components must be designed to operate in the cloud, at Internet scope, in a highly scalable and federated manner. This is what Microsoft has referred to in the past as an *Internet Service Bus* implementation (see Figure 2).[[3]](#footnote-4)

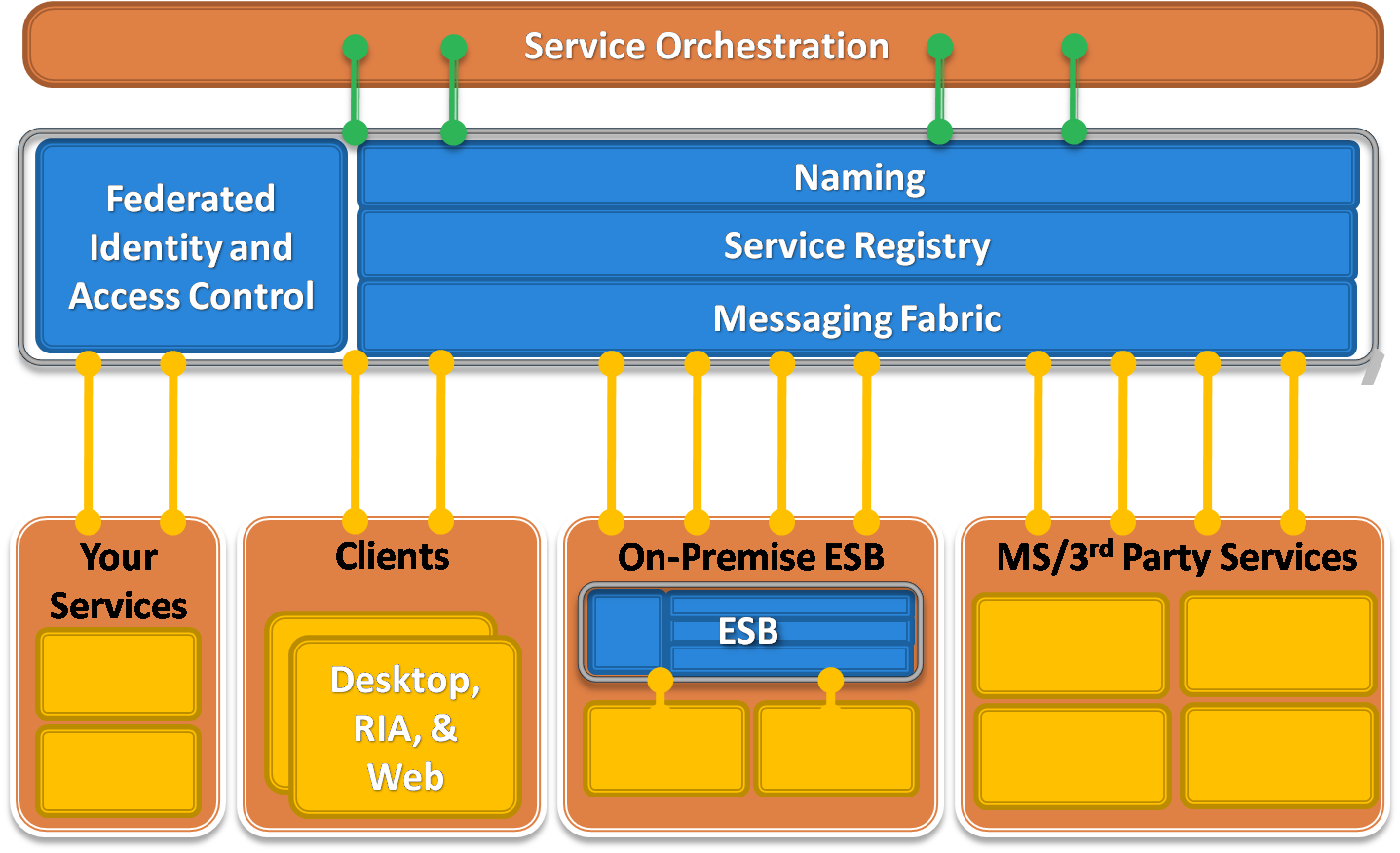


Figure : The Internet Service Bus

An Internet Service Bus would make it possible to integrate your on-premise ESB with your services running in the cloud, with a variety of 3rd party services provided by Microsoft or other vendors (such as those offered within the Azure Service Platform), and with a variety of desktop, RIA[[4]](#footnote-5), and Web applications that may be running in satellite locations outside of the enterprise firewall. In order to make this possible, the implementation needs to provide federated solutions based on open Internet standards and a robust messaging fabric capable of bidirectional communication at Internet scope.

Tackling bidirectional communication at Internet scope is not trivial due to some of today’s realities. The first problem is the shortage of IPv4 addresses. For all practical purposes, we’ve run out. It’s difficult to acquire a public IPv4 address anymore. Instead, most Internet providers, corporate networks, and wireless networks use dynamic IP address allocation and network address translation (NAT) techniques. Such IP addresses are private to those networks and are not publicly addressable from the outside.

Another challenge is related to security. In most enterprise environments, on-premise software is almost completely shielded from the outside world by layers and layers of firewalls and other protective network devices. This is necessary because of the widespread security threads inherent in the Internet.[[5]](#footnote-6) Most network environments allow a variety of outbound ports through their firewalls but highly constrain the number of allowed inbound ports.[[6]](#footnote-7) Ports 80 (HTTP) and 443 (HTTPS) are often the only sanctioned inbound ports, presenting a big challenge for different types of bidirectional communication.

Imagine a situation where a sales person is traveling and she’s using your application on a wireless network in random hotel somewhere in the world (see Figure 3). In scenarios like this, it can be very difficult to establish communication with machines sitting behind these various different network layers.

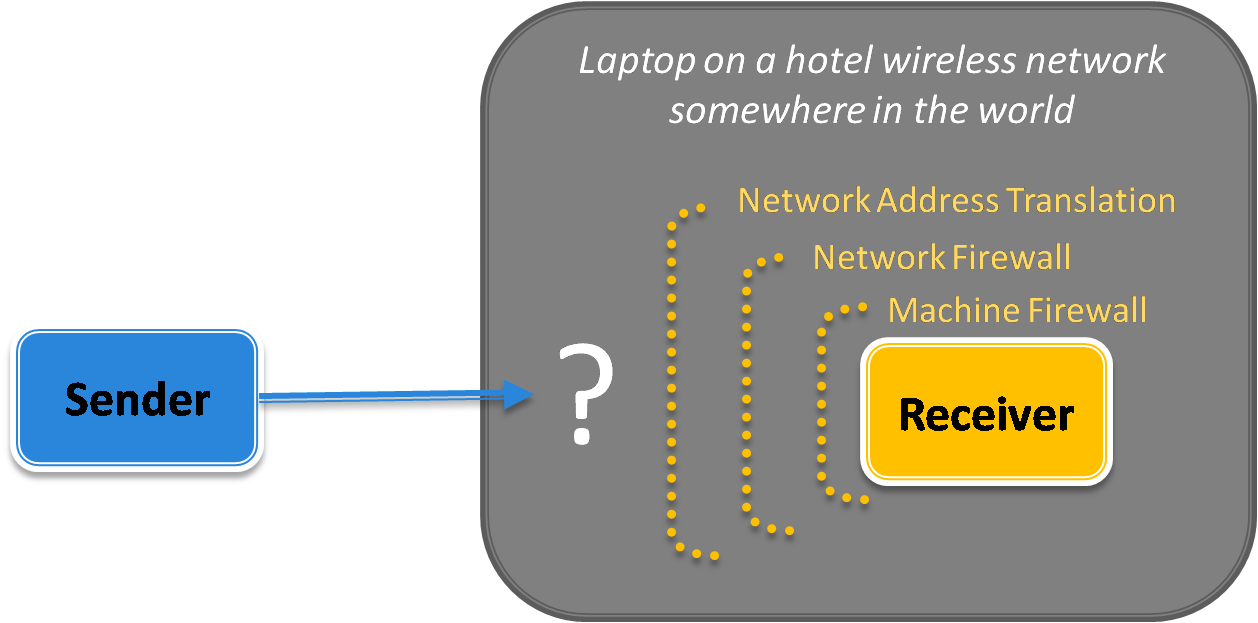


Figure : Internet Connectivity Challenges

Companies often deal with these connectivity challenges by opening inbound firewall ports (much to their system administrator’s dismay) or by using different workarounds like dynamic DNS, NAT port mappings, or UPnP, all of which are brittle, difficult to manage, and susceptible to security threats. As more and more applications are requiring this type of bidirectional communication, we’re experiencing a growing tension here, especially since the mentioned workarounds are often completely impractical.

Despite these challenges, some of today’s most popular Internet applications are inherently bidirectional. Consider things like instant messaging, online multiplayer games, and peer-to-peer file sharing applications that use protocols like BitTorrent, which accounts for a large percentage of all Internet traffic today. These applications have written the low-level networking logic to traverse firewalls and NAT devices, and to create direct peer-to-peer connections when possible. They often accomplish this through a centralized *relay service* that provides the connectivity logic (see Figure 4).

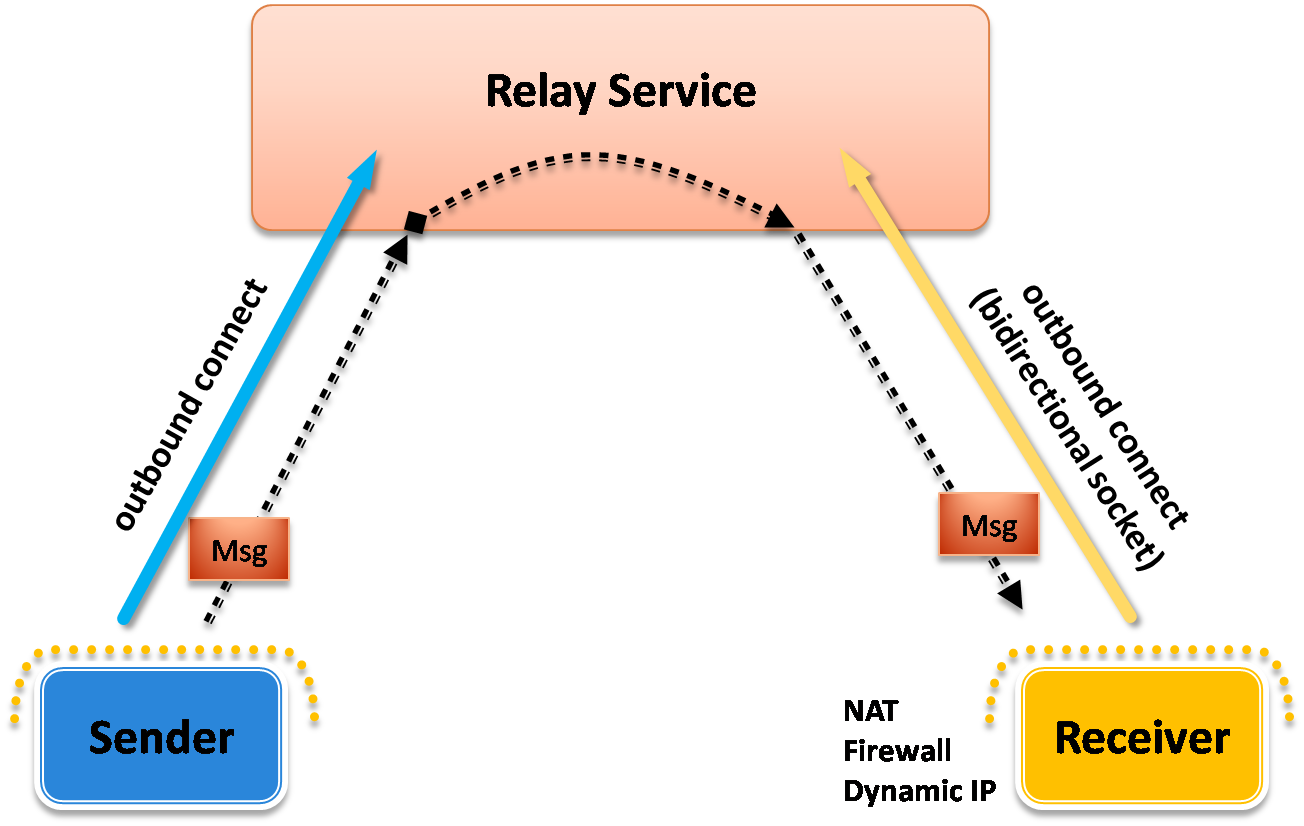


Figure : A Relay Service

The receiver connects to the relay service through an outbound port and creates a bidirectional socket for communication. The sender can then communicate with the receiver by sending messages to the relay service, at which point, the relay service “relays” the messages to the receiver through the bidirectional socket already in place. In this case, the receiver doesn’t need to have any inbound ports open on the firewall to make this work. This is how most instant messaging applications work today.

Ultimately, a “service bus” that successfully operates at Internet scope must provide a messaging fabric capable of dealing with these connectivity challenges through a relay service in the cloud.

## Introducing the .NET Service Bus

The Microsoft® .NET Service Bus is a concrete implementation of the service bus pattern designed to operate at Internet scope within highly-scalable Microsoft data centers. The .NET Service Bus provides a federated identity and access control mechanism (via the .NET Access Control Service), a federated naming system, a dynamic service registry, and a robust messaging fabric capable of overcoming the connectivity challenges described in the previous section. In addition, you can use the .NET Workflow Service in conjunction with the .NET Service Bus for the service orchestration layer illustrated in Figure 2.

A central component of the .NET Service Bus messaging fabric is a centralized (but highly load-balanced) relay service that supports a variety of different transport protocols and Web services standards, including SOAP, WS-\*, and even REST. The relay service provides a variety of different relay connectivity options and can even help negotiate direct peer-to-peer connections when possible.

The .NET Service Bus was designed for all developers, regardless of platform, but was highly optimized for .NET developers using Windows Communication Foundation (WCF), both in terms of performance and usability. The .NET Service Bus provides full access to its relay service through SOAP and REST interfaces, as do the .NET Access Control Service and the .NET Workflow Service. This makes it possible for any SOAP or REST-capable programming environment to integrate with these Azure services.

On the Microsoft .NET Services site (see Additional Resources) you can download several SDK’s for integrating with these services that target different programming environments. Today there’s a Java SDK, a Ruby SDK, and of course a Microsoft® .NET Services SDK. If you’re a .NET developer comfortable with WCF, the .NET Services SDK will make working with the .NET Service Bus feel like its second nature. The .NET Services SDK installs a set of new WCF “relay” bindings (and corresponding channel components) that know how to integrate with the .NET Service Bus behind the scenes. This makes programming the .NET Service Bus feel just like working with traditional WCF clients and services.

## Your First .NET Service Bus Solution

Before you can begin working with the .NET Service Bus, you must first obtain an account from the Azure Services Platform portal (see Additional Resources). Once you have registered and been approved, you will receive an invitation code that you can use to create a new .NET Services *solution*.

First, you’ll need to sign into the .NET Services portal using your Windows Live ID, and then you can select “Create a New Solution”. At this point, you enter the invitation code you received from Microsoft and press “Sign Up”. Your new solution will act as the container for your .NET Service Bus endpoints.[[7]](#footnote-8)

Figure 5 shows what the .NET Services portal looks like after creating a new solution. You’ll notice that I’ve created two solutions within my account, one called “pluralsight” and another called “skonnard".[[8]](#footnote-9) The solution name is also the username for the solution and you’ll receive an initial password for the solution once you’ve created it, which you can always change later if you wish.

From this page, you can click on one of the getting started links to learn more about how you can begin using each of the .NET Services or you can click on one of the solution names to manage the solution. When you click on a solution, you’ll be taken to a solution management page that allows you to do things like manage solution credentials, and manage different aspects of the individual .NET Services. See Figure 6 for an example of what this page looks like for my “pluralsight” solution.

If you click on the .NET Service Bus link, you’ll be taken to a page that describes your .NET Service Bus solution, providing a quick reference to the URI naming structure you must use, along with a link to the service registry for the solution (see Figure 7). As indicated on the Web page, the base URI for all endpoints within my “pluralsight” solution must be http://servicebus.windows.net/services/pluralsight/. I can append paths to the right of that base URI as I see fit. Also, if you browse to the solution’s base URI, you’ll receive an Atom feed representing your solution registry (it will be empty right now).

Next you need to download the Microsoft® .NET Services SDK and install it. Once you have the SDK installed, you’re ready to start writing code to leverage the .NET Service Bus in your applications.

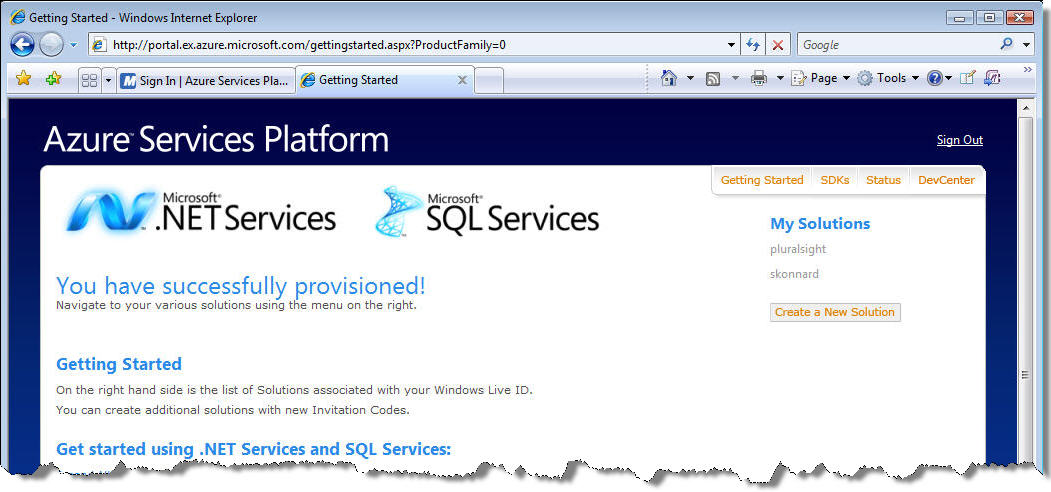


Figure : Creating a .NET Services Solution

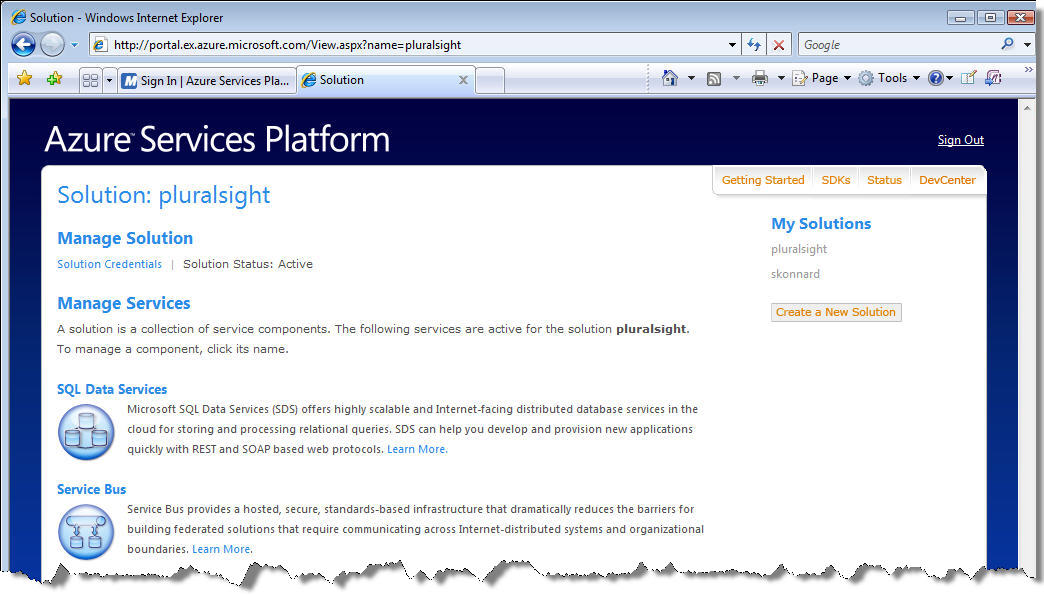


Figure : Managing a .NET Services Solution



Figure : Getting Started with a .NET Service Bus Solution

Let me show you a quick example of how you can configure a simple WCF application to take advantage of the .NET Service Bus. We’ll start with the following simple WCF service contract and implementation:

[ServiceContract]

public interface IHelloServiceBus

{

[OperationContract]

string SayHello(string name);

}

class HelloServiceBus : IHelloServiceBus

{

public string SayHello(string name)

{

string greeting = string.Format("Hello {0}!", name);

Console.WriteLine("Returning: {0}", greeting);

return greeting;

}

}

We’ll host this service in the following console application that reads the service configuration details from the application configuration file:

class Program

{

static void Main(string[] args)

{

Console.WriteLine("\*\*\*\* Receiver \*\*\*\*");

ServiceHost host = new ServiceHost(typeof(HelloServiceBus));

host.Open();

Console.WriteLine("Press [Enter] to exit");

Console.ReadLine();

host.Close();

}

}

And we’ll start by using the following endpoint definition in the application configuration file. Notice how the endpoint uses NetTcpBinding and a local address of net.tcp://localhost:8080/helloservicebus:

<configuration>

<system.serviceModel>

<services>

<service name="Microsoft.ServiceBus.Samples.HelloServiceBus">

<endpoint address="net.tcp://localhost:8080/helloservicebus"

binding="netTcpBinding"

contract="Microsoft.ServiceBus.Samples.IHelloServiceBus" />

</service>

</services>

</system.serviceModel>

</configuration>

Next, we can write a client application that invokes the service. The following code shows how to do this using the same IHelloServiceBus contract definition. It also assumes that the endpoint details (for “RelayEndpoint”) will be read from the client’s application configuration file:

class Program

{

static void Main(string[] args)

{

Console.WriteLine("\*\*\*\* Sender \*\*\*\*");

Console.WriteLine("Press <Enter> to start sending messages.");

Console.ReadLine();

ChannelFactory<IHelloServiceBus> channelFactory =

new ChannelFactory<IHelloServiceBus>("RelayEndpoint");

IHelloServiceBus channel = channelFactory.CreateChannel();

string response = channel.SayHello(".NET Service Bus");

Console.WriteLine(response);

channelFactory.Close();

}

}

And finally, the client’s application configuration file needs to have the equivalent endpoint definition in order to communicate with the service via the TCP endpoint it exposes:

<configuration>

<system.serviceModel>

<client>

<endpoint address="net.tcp://localhost:8080/helloservicebus"

binding="netTcpBinding"

contract="Microsoft.ServiceBus.Samples.IHelloService"

name="RelayEndpoint" />

</client>

</system.serviceModel>

</configuration>

If you run the two applications, you’ll see “Hello .NET Service Bus!” displayed in both console windows. In this case there’s a direct TCP connection between the client and the service applications.

Now let’s look at what it takes to introduce the .NET Service Bus as a relay between the client and service applications. First we’ll need to reconfigure the service host to listen on the .NET Service Bus, and then we’ll need to reconfigure the client to send messages through the .NET Service Bus.

We can reconfigure the service to listen on the .NET Service Bus by simply changing the binding from NetTcpBinding to NetTcpRelayBinding. When doing so, we’ll also need to specify a valid .NET Service Bus address for the endpoint. Since I have a solution named “pluralsight", I can use an address of sb://servicebus.windows.net/services/pluralsight/helloservicebus.

I also need to provide credentials in order to prove to the relay service that I’m allowed to listen within that solution address space. These credentials will be presented to the .NET Access Control Service in order to receive a token for the .NET Service Bus. There are several ways you can do this but I’ll simply provide a username and password for this example.[[9]](#footnote-10) You can specify the username and password in code or configuration. I’m using the latter approach through the <transportClientEndpointBehavior> element. The following shows the complete configuration for the .NET Service Bus-enabled service:

<configuration>

<system.serviceModel>

<services>

<service name="Microsoft.ServiceBus.Samples.HelloServiceBus">

<endpoint address=

"sb://servicebus.windows.net/services/pluralsight/helloservicebus"

behaviorConfiguration="default"

binding="netTcpRelayBinding"

contract="Microsoft.ServiceBus.Samples.IHelloServiceBus" />

</service>

</services>

<behaviors>

<endpointBehaviors>

<behavior name="default">

<transportClientEndpointBehavior credentialType="UserNamePassword">

<clientCredentials>

<userNamePassword

userName="[solution-name]" password="[solution-password]" />

</clientCredentials>

</transportClientEndpointBehavior>

</behavior>

</endpointBehaviors>

</behaviors>

</system.serviceModel>

</configuration>

When the WCF service host opens with this configuration, it will first send the client credentials to the .NET Access Control Service and acquire a security token for listening on the .NET Service Bus. It will then establish a TCP connection with the relay service and present the security token it acquired. Assuming the service is allowed to listen on that address (meaning the token contains the necessary claim), the relay service will create a listener for relaying messages to our local WCF service.

Reconfiguring the client application is very similar. First, we need to change the endpoint to use the NetTcpRelayBinding and the same .NET Service Bus address that we configured our service to listen on. We also need to configure the client with credentials. As with the service, clients must also prove that they are allowed to send messages to a particular address on the .NET Service Bus by acquiring a token from the .NET Access Control Service. The following shows the complete client configuration:

<configuration>

<system.serviceModel>

<client>

<endpoint address=

"sb://servicebus.windows.net/services/pluralsight/helloservicebus"

binding="netTcpRelayBinding"

contract="Microsoft.ServiceBus.Samples.IHelloServiceBus"

behaviorConfiguration="default"

name="RelayEndpoint" />

</client>

<behaviors>

<endpointBehaviors>

<behavior name="default">

<transportClientEndpointBehavior credentialType="UserNamePassword">

<clientCredentials>

<userNamePassword

userName="[solution-name]" password="[solution-password]" />

</clientCredentials>

</transportClientEndpointBehavior>

</behavior>

</endpointBehaviors>

</behaviors>

</system.serviceModel>

</configuration>

With these changes in place, we can run the service host application followed by the client application, and we’ll see the same result as before only this time the communication was relayed through the .NET Service Bus (see Figure 8), making it possible to traverse a variety of nasty network obstacles in the way.

|  |  |
| --- | --- |
| HelloSender.jpg | HelloReceiver.jpg |
|  |  |

Figure : HelloServiceBus Sample in Action

Now that you’re familiar with the various .NET Service Bus concepts and how to get started with a simple solution, we’re ready to dive deeper into the details. The following sections will explore the key areas of the .NET Service Bus architecture along with their features and capabilities.

# Naming and Discovery

Understanding how to name your services on the .NET Service Bus is of central importance. Before exploring the .NET Service Bus approach, let’s consider how the Domain Naming System (DNS) works.

DNS is designed to map domain names to IP addresses. When you browse to a Web site, the first thing that happens is a DNS lookup to figure out what IP address the human friendly domain name resolves to. Since DNS relies on public IP addresses, it doesn’t work for identifying hosts sitting behind NAT devices without the help of a layered service like Dynamic DNS. Today it’s common for a single IP address to identify an entire network of hosts sitting behind a single NAT device. Ultimately, the DNS model is less than ideal for naming and identifying endpoints in a service oriented world.

Unlike DNS, the .NET Service Bus naming system is optimized for naming service endpoints in a host-independent fashion. You can think of the naming system as a global forest of federated naming trees projected onto host-independent URI’s. Each solution maps to a naming tree, hence, each solution name must be globally unique. The naming trees are federated because each solution owner controls the names within his solution. They are “trees” because of the hierarchical nature of the namespace (names within name within names). There’s a natural projection for these names onto URI’s, but the resulting URI’s are completely host-independent – you can have multiple services all running on different hosts that share the same solution name. These characteristics of the .NET Service Bus naming system provide a more granular, endpoint-level approach that complements DNS nicely.

## Naming System

The root of the .NET Service Bus naming system is resolvable through traditional DNS techniques. The naming system then relies on host-independent criteria – specifically the solution name – to distinguish between different domains of control within the naming system. Then it’s up to solution owners to control the hierarchical namespace with their respective solutions as illustrated in Figure 9.[[10]](#footnote-11)

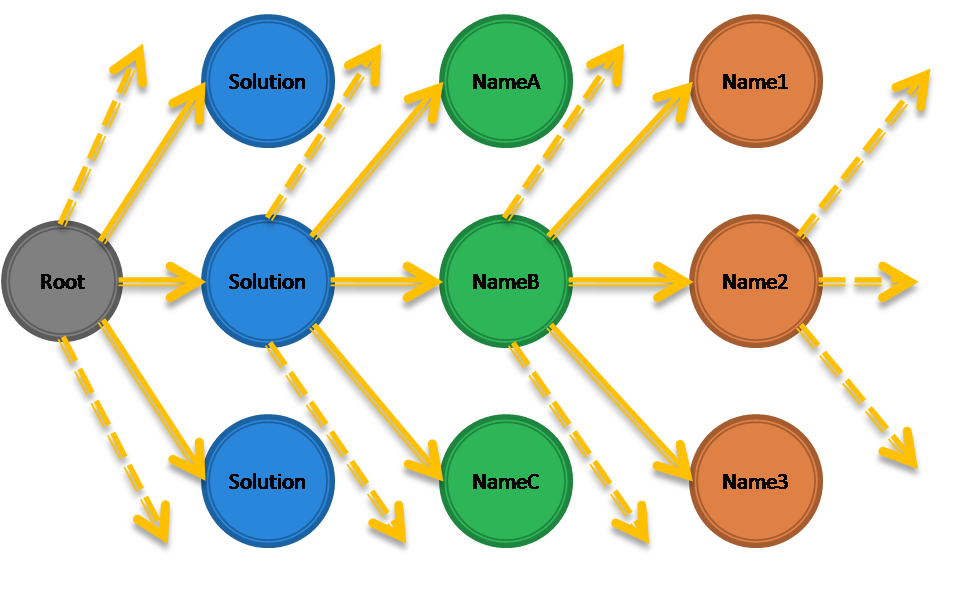


Figure : .NET Service Bus Naming System

The way you project .NET Service Bus names onto URI’s is as follows:

[scheme]://[solution-naming-scope]/[name]/[name]/...

As of the CTP release, the .NET Service Bus supports two URI schemes: “sb” and “http”. You’ll use “http” for all HTTP-based endpoints and the “sb” scheme for everything else. The *solution-naming-scope* identifies a unique naming tree by combining the root of the naming system with the solution name.

The root of the naming system for the CTP release is “servicebus.windows.net”. And you need to use the following URI format to combine the root of the naming system with your solution name:

[scheme]://servicebus.windows.net/services/[solution]/[name]/[name]/...

For example, all endpoint names within my “pluralsight” solution must start with sb://servicebus.windows.net/services/pluralsight/ or, if you’re using HTTP, http://servicebus.windows.net/services/pluralsight. Everything that comes after the solution in the path is part of the user-defined namespace. I can create an unlimited number of endpoint names within my solution by adding a hierarchy of names after my solution name. Suppose we have an office in Boston, Salt Lake City, and Los Angeles, and each one needs to publish some endpoints. We could assign each office a unique name and give the office control over defining names within its hierarchy[[11]](#footnote-12):

sb://servicebus.windows.net/services/pluralsight/boston/

sb://servicebus.windows.net/services/pluralsight/saltlakecity/

sb://servicebus.windows.net/services/pluralsight/losangeles/

Then suppose each of the offices needs to expose an endpoint for a (location specific) training registration service. They could accomplish that by adding “training” to the end of their URI’s:

sb://servicebus.windows.net/services/pluralsight/boston/training

sb://servicebus.windows.net/services/pluralsight/saltlakecity/training

sb://servicebus.windows.net/services/pluralsight/losangeles/training

And if the Boston office, as the headquarters, needs to expose a payment processing service, it can simply create another endpoint name using “payment” within its namespace:

sb://servicebus.windows.net/services/pluralsight/boston/payment

It’s key to understand that these service endpoints don’t have to be hosted on the same server or on the same network or even in the same geography. It’s the job of the .NET Service Bus, and specifically the relay service, to determine where the endpoints are physically located at runtime.

In a future (post CTP) release, the .NET Service Bus will support a different URI format for specifying the solution-naming-scope. Instead of specifying the solution name in the path of the URI, they will support the following format where you specify the solution name as part of the domain name:

[scheme]://[solution].servicebus.windows.net/[name]/[name]

Notice how the solution if added to the left of the domain name and how “services” is no longer part of the path.[[12]](#footnote-13) Going back to the payment service example, the new URI would look like this:

sb://pluralsight.servicebus.windows.net/boston/payment

The .NET Service Bus does not provide an API for directly interacting with the naming system. Instead, you access with the naming system through the service registry functionality.

## Service Registry

The .NET Service Bus provides a service registry for publishing and discovering service endpoint references within a solution. It allows you to publish endpoint references as either simple URI’s or as official WS-Addressing endpoint references. Others can then discover a solution’s endpoint references by browsing to the solution’s base address and retrieving an Atom feed that contains the information.[[13]](#footnote-14)

The service registry exposes a solution’s endpoint references through a linked tree of Atom 1.0 feeds. You navigate the service registry by simply navigating the naming system via HTTP, browsing to each level within a solution’s naming structure you wish to inspect. When you browse to the solution’s base HTTP address, you’ll receive the root Atom 1.0 feed describing the first level of nested names. If you then browse to one of the nested names, you’ll receive another Atom 1.0 feed that describes the second level of names nested within the container name. This continues until you reach a leaf name in the tree.

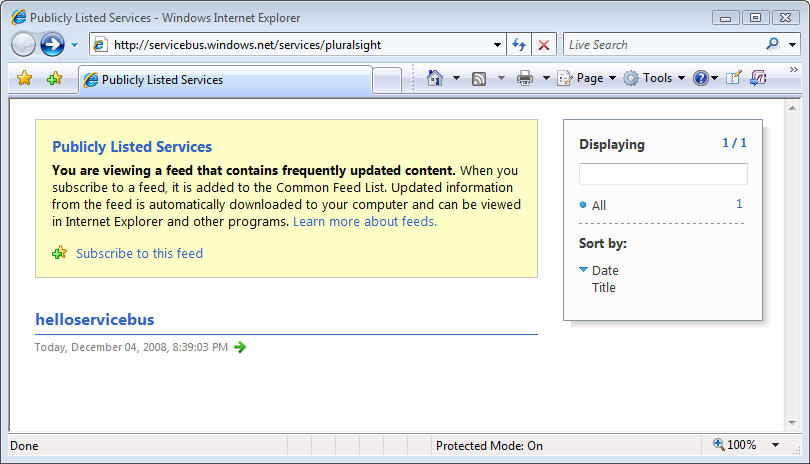


Figure : Browsing to a Solution's Service Registry

There are two ways to publish endpoint references into the registry: automatically or manually. The most common approach is to let the .NET Service Bus take care of it for you when you register listeners with the relay service. The relay service automatically populates the service registry with endpoint references as you create listeners unless you instruct it not to do so.[[14]](#footnote-15) For example, if I re-run the host for my HelloServiceBus service shown earlier and then browse to my solution’s base HTTP address, I see an Atom 1.0 feed rendered within Internet Explorer listing a single endpoint reference (see Figure 10).

Let’s look at a more interesting example. Suppose you were to reconfigure the HelloServiceBus host application shown earlier with the following endpoint configuration:

<configuration>

<system.serviceModel>

<services>

<service name="Microsoft.ServiceBus.Samples.HelloServiceBus">

<endpoint address=

"sb://servicebus.windows.net/services/pluralsight/boston/training"

behaviorConfiguration="default"

binding="netTcpRelayBinding"

contract="Microsoft.ServiceBus.Samples.IHelloServiceBus" />

<endpoint address=

"sb://servicebus.windows.net/services/pluralsight/boston/payment"

behaviorConfiguration="default"

binding="netTcpRelayBinding"

contract="Microsoft.ServiceBus.Samples.IHelloServiceBus" />

<endpoint address=

"sb://servicebus.windows.net/services/pluralsight/saltlakecity/training"

behaviorConfiguration="default"

binding="netTcpRelayBinding"

contract="Microsoft.ServiceBus.Samples.IHelloServiceBus" />

<endpoint address=

"sb://servicebus.windows.net/services/pluralsight/losangeles/training"

behaviorConfiguration="default"

binding="netTcpRelayBinding"

contract="Microsoft.ServiceBus.Samples.IHelloServiceBus" />

</service>

</services>

...

</system.serviceModel>

</configuration>

Now when you run the host application and browse to the solution’s base HTTP address, you’ll see the root Atom 1.0 feed shown in Figure 11. Notice how the root feed only contains the first level of names from the endpoints we defined, specifically “boston”, “losangeles”, and “saltlakecity”.

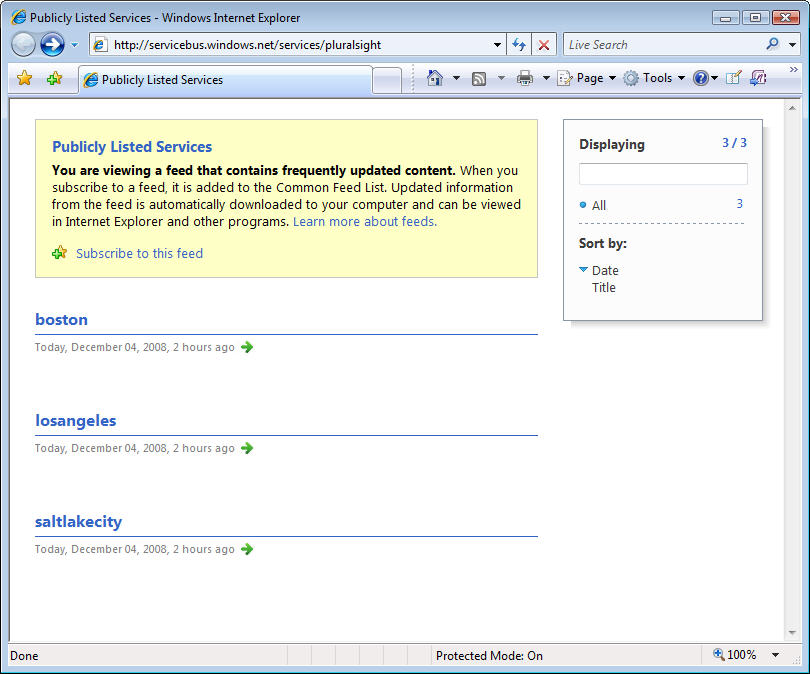


Figure : The Root Atom 1.0 Feed

Now if you click on “boston”, you’ll issue a request for another Atom 1.0 feed describing the names within “boston” level of the hierarchy. Figure 12 shows what the nested feed for “boston” looks like. If we were to browse to the “losangeles” or “saltlakecity” feeds, you’d only see “training” in the feed.

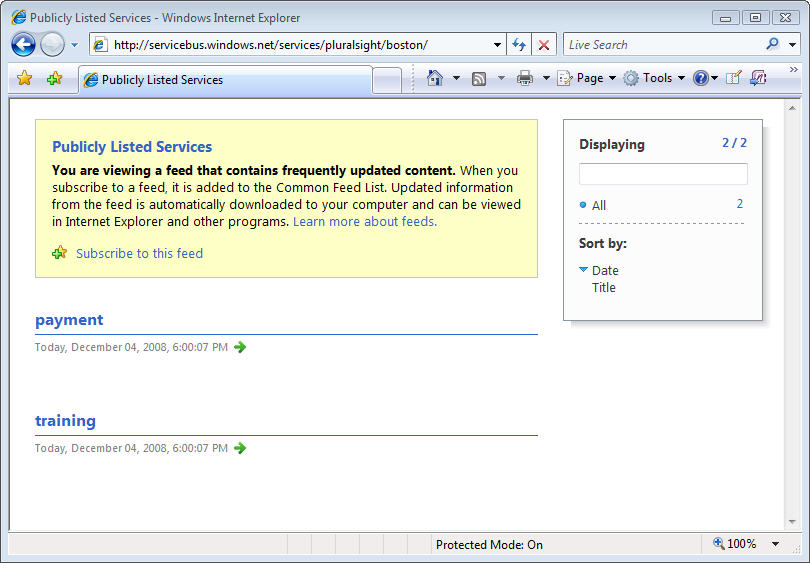


Figure : A Nested Atom 1.0 Feed

The other way you can publish endpoint references into the service registry is by programmatically interacting with the service registry through the Atom Publishing Protocol (APP) or WS-Transfer. You can simply issue HTTP PUT requests with an Atom 1.0 entry to any name in your solution’s naming tree, and the endpoint reference will be added. You delete endpoint references by issuing an HTTP DELETE request to the name you wish to delete. When using this approach, names are automatically created and deleted in the naming system as you create or delete service registry entries.[[15]](#footnote-16)

# Messaging

At the heart of the .NET Service Bus you’ll find a scalable, general purpose relay service (see Figure 4) that provides a variety of advanced messaging features for flexible Internet connectivity.

## The Relay Service

The relay service makes it easy to build solutions capable of communicating through firewalls and NAT devices using a variety of different messaging patterns. The relay service supports one-way messaging, request/response messaging, peer-to-peer messaging, event distribution for Internet-scope publish/subscribe scenarios, and connection-oriented bi-directional socket communication.

When using the relay service, you’re essentially delegating the transport-level “listening” responsibility to the relay service in the cloud. In this case, your services no longer have to create local transport listeners; instead they rely on the relay service to handle the specified transport communication details. It’s the relay service’s responsibility to simply forward incoming messages to your local service.

The relationship between your services and the relay service is analogous to the relationship between Windows applications that build on top of HTTP.SYS. Applications that use HTTP.SYS don’t actually create HTTP transport listeners themselves. Instead, they simply register an address with the HTTP.SYS driver and ask it to forward along all messages it receives on the registered address. This allows for centralization of the HTTP transport logic and reduces surface area, which is good from a security perspective. The relay service plays a similar role to that of HTTP.SYS but it resides up in the cloud and your local services register to have it pass along messages received at a particular address.

You must initiate the connection between your local service and the relay service.[[16]](#footnote-17) When using .NET, you’ll accomplish this via the WCF programming model and a new suite of WCF “relay” bindings that ship with the .NET Services SDK. Behind the scenes, the relay bindings map to new transport binding elements designed to create WCF listeners that integrate with relay service. In most connection modes, the WCF listener creates a secure outbound socket connection into the relay service that supports bidirectional communication. During the process it authenticates, specifies a name to listen on, and tells it what type of listener to establish. The WCF relay bindings provide a variety of messaging options that make it easy to overcome a variety of tough Internet connectivity challenges even in the most locked-down environments. We’ll be discussing these options in detail throughout the following sections.

## Listeners and Naming

When choosing names for your listeners, follow the guidelines we discussed in the Naming section. Although you can organize listeners by whatever criteria makes sense for your solution, it’s important to realize that only a single listener can listen on a particular URI space, except when using the NetEventRelayBinding, which is specifically designed for multiple listeners on a single URI.

In all other cases, if you attempt to open a second listener on a URI that’s already being used by another listener, the attempt to open the second listener will fail. Furthermore, if the second URI is within the URI scope of the first URI, the attempt to open the second listener will also fail. For example, if the first listener uses /foo/bar and then the second listener attempts to open /foo/bar/baz, the attempt will fail. Multiple listeners can use URI’s that share the same base address, as is the case with /foo/bar and /foo/baz, but one URI cannot be the complete prefix of another URI or you’ll see an error.

The relay service routes messages to listeners using a longest-prefix-match algorithm (not necessarily an exact match), which also makes it possible for the target service to directly inspect the URI path segments and the query string to do custom processing, which comes in handy for RESTful scenarios.

## Ports Required by the Relay Service

The relay service only requires a few outbound ports to be open, specifically ports 808, 818, 819, and 828. [[17]](#footnote-18) It uses port 808 for outbound TCP connections and port 828 for outbound TCP/SSL connections. They also use ports 818 and 819 for certain advanced TCP-based connectivity modes that we’ll discuss later. It’s important to note that you don’t have to open any inbound ports on your firewall or perform any kind of port mapping on your NAT/router device in order to use the relay service.

If you’re operating in a tightly managed network environment that blocks all outbound socket connections, except for those used by HTTP/HTTPS, you can use a special HTTP-based connectivity option where the WCF listener actually polls for messages from the relay service over an HTTP connection. This approach usually always works even the most locked-down network environments.

From a security perspective, the relay service functions as a demilitarized zone (DMZ), completely isolated from the service's local environment. The DMZ handles all network traffic externally, making it possible to filter out unwanted traffic in the cloud before it enters your service environment. Furthermore, the relay service hides all details about the network location of the local listener (in effect making it anonymous), thereby reducing the potential surface area for attack (a la HTTP.SYS). The relay service is integrated with the .NET Access Control Service, which provides authentication and authorization features, giving you a flexible security gate in the cloud that is quite simple to manage.

If you’re concerned by the idea of the relay service “punching” holes in your firewall, it’s important to consider the security features provided by the relay service. A firewall is designed to block access to network resources at the port level. Conceptually, the relay service (combined with the security features of the .NET Access Control Service) is similar to a firewall in that sense, only the relay service is designed to block undesired access to your service resources on an endpoint by endpoint basis.

## WCF Relay Bindings

The primary programming model for working with the .NET Service Bus on the .NET platform is WCF. The .NET Services SDK comes with a set of new WCF bindings that automate the integration between your WCF services and clients with the relay service. In most cases, all you need to do is replace the current WCF binding that you’re using with one of the .NET Service Bus bindings.

Figure 13 lists all of the .NET Service Bus WCF bindings and the standard WCF bindings they correspond to. The most commonly used WCF bindings, such as BasicHttpBinding, WebHttpBinding, WSHttpBinding, and NetTcpBinding, all have a corresponding .NET Service Bus binding with a very similar name (just insert “Relay” before “Binding”). There are only a few relay bindings – NetOneWayRelayBinding and NetEventRelayBinding – that don’t have a corresponding binding within the .NET Framework.

|  |  |
| --- | --- |
| Standard WCF Binding | Equivalent Relay Binding |
| BasicHttpBinding | BasicHttpRelayBinding |
| WebHttpBinding | WebHttpRelayBinding |
| WSHttpBinding | WSHttpRelayBinding |
| WS2007HttpBinding | WS2007HttpRelayBinding |
| WSHttpContextBinding | WSHttpRelayContextBinding |
| WS2007HttpFederationBinding | WS2007HttpRelayFederationBinding |
| NetTcpBinding | NetTcpRelayBinding |
| NetTcpContextBinding | NetTcpRelayContextBinding |
| N/A | NetOnewayRelayBinding |
| N/A | NetEventRelayBinding |

Figure : WCF Relay Bindings

The relay bindings work just like the standard WCF bindings for the most part. For example, they support the different WCF message versions (SOAP 1.1, SOAP 1.2, and None), the various WS-\* security scenarios, reliable messaging, streaming, metadata exchange, the Web programming model (e.g., [WebGet] and [WebInvoke]), and many more standard WCF features. There are only a few WCF features not supported by design including atomic transaction flow and transport level authentication.

If you’re familiar with how WCF works under the covers, you might be interested to know how the new bindings (shown in Figure 13) map to transport binding elements. Figure 14 specifies the transport binding element for each of the new relay bindings. There are four new WCF transport binding elements that they all map to: HttpRelayTransportBindingElement, HttpsRelayTransportBindingElement, TcpRelayTransportBindingElement, and OnewayRelayTransportBindingElement.

|  |  |
| --- | --- |
| Relay Binding | Transport Binding Element |
| BasicHttpRelayBinding | Http(s)RelayTransportBindingElement |
| WebHttpRelayBinding | Http(s)RelayTransportBindingElement |
| WSHttpRelayBinding | Http(s)RelayTransportBindingElement |
| WS2007HttpRelayBinding | Http(s)RelayTransportBindingElement |
| WSHttpRelayContextBinding | Http(s)RelayTransportBindingElement |
| WS2007HttpRelayFederationBinding | Http(s)RelayTransportBindingElement |
| NetTcpRelayBinding | TcpRelayTransportBindingElement |
| NetTcpRelayContextBinding | TcpRelayTransportBindingElement |
| NetOnewayRelayBinding | OnewayRelayTransportBindingElement |
| NetEventRelayBinding | OnewayRelayTransportBindingElement |

Figure : WCF Relay Binding Elements

These new WCF primitives are ultimately the ones that provide the low-level channel integration with the relay service. If you were to compare the binding elements used by WSHttpBinding and WSHttpRelayBinding, you’d notice that the collections are almost identical. The only significant difference between them is the transport binding element as illustrated in Figure 15.

WSHttpBinding uses the standard HttpTransportBindingElement. On the client side, this binding element maps to a client channel that uses System.Net.WebRequest for the HTTP communication. And on the server side, it maps to a listener built in terms of System.Net.HttpListener for the HTTP.SYS integration.

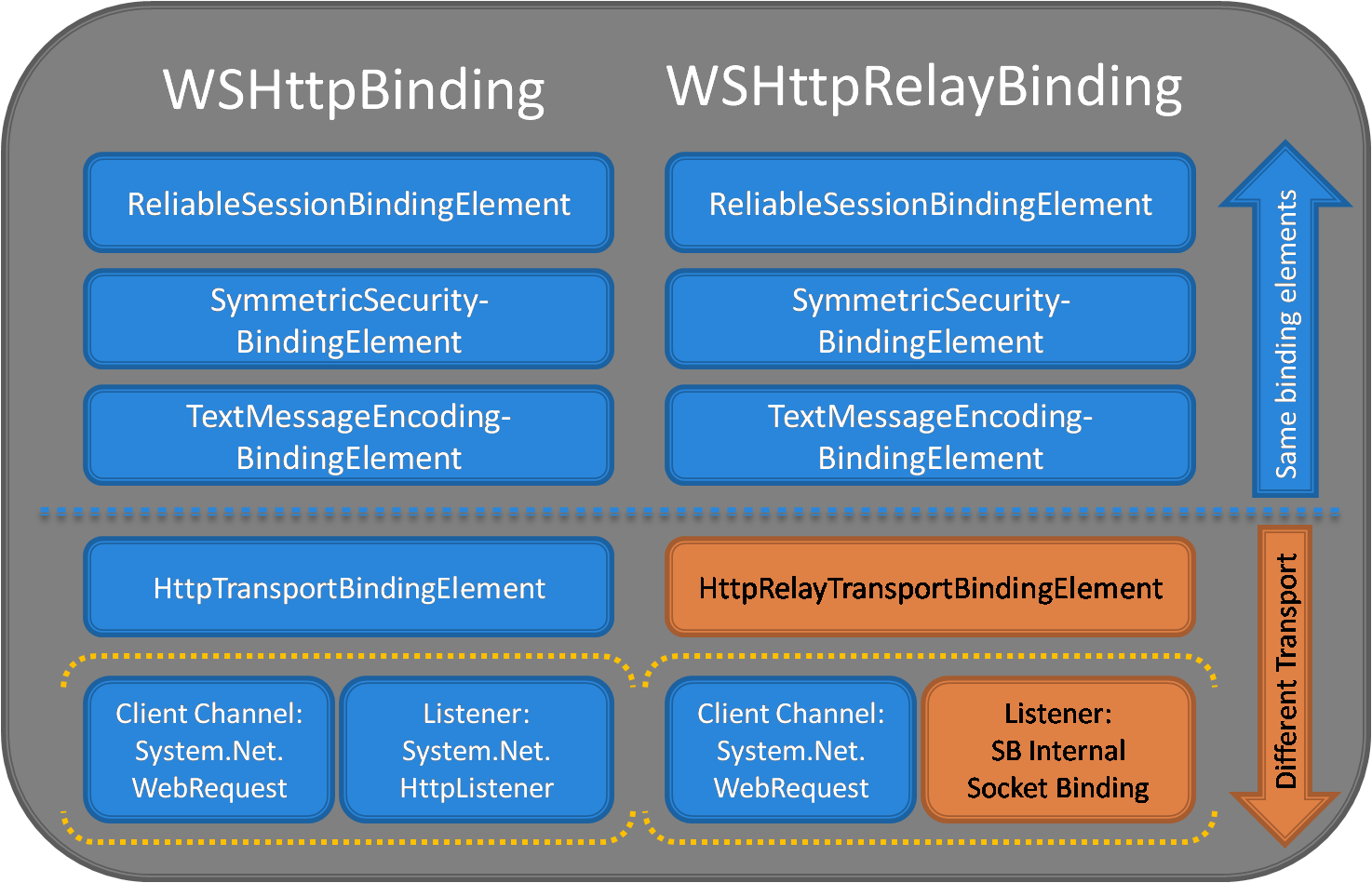


Figure : Binding Stacks Compared

WSHttpRelayBinding, on the other hand, maps to the HttpRelayTransportBindingElement, which interestingly uses the same type of client channel, meaning the way WCF clients talk to the relay service is exactly the same, but uses a different listener on the server side. The new WCF listener is the one that knows how to integrate with relay service and process “relayed” messages. Notice how all of the binding elements layered above the transport binding element are exactly the same in both cases. This illustrates that the relay service has been designed to compose nicely with the various WS-\* protocols.

If we were to redraw this figure for the other WCF bindings, it would look very similar in that the primary difference will be found in the transport listener components. In the following sections, we’re going to dive into the details of the main WCF relay bindings and show how to use them.

## NetOnewayRelayBinding

The NetOnewayRelayBinding is the most constrained of the all the relay bindings because it only supports one-way messages. However, it’s also the most successful relay binding at providing listeners with an inbound communication path in constrained network environments because it provides support for both TCP and HTTP-based connections with the relay service on the receiver side of the equation.

By default, the NetOnewayRelayBinding uses SOAP 1.2 over TCP along with a binary encoding of the messages, although these communication settings are configurable through standard binding configuration techniques.[[18]](#footnote-19) Services that use this binding must always use the “sb” protocol scheme.

When a WCF service uses this binding, the WCF listener attempts to establish an outbound connection with the relay service in order to create a bidirectional socket. In this case, it always creates a secure TCP/SSL connection through outbound port 828. During the connection process the listener authenticates (by supplying a security token acquired from the .NET Access Control Service), specifies a name to listen on within the relay service, and tells the relay service what type of listener to create.

When a WCF client uses this binding, it uses the NetTcpRelayBinding channel components and creates a TCP connection with the relay service via port 808 (TCP) or 828 (TCP/SSL), depending on the binding configuration. During the connection process it must authenticate with the relay service (again by supplying a security token acquired from the .NET Access Control Service). Once the client has established the TCP connection, it can then begin sending one-way messages to the relay service for it to “relay” to the target service through the receiver’s established TCP connection (see Figure 16).

Figure 16 illustrates something to note about the .NET Service Bus architecture. When senders and receivers establish connections with the relay service, they are load balanced across an array of front-end nodes, which collectively provide the necessary scalability for this type of service. When the sender sends a message, the sender’s front-end node communicates with the backend naming and routing fabric in order to successfully relay the message to the other front-end node attached to the receiver. This approach ultimately provides a scalable communication path between senders and receivers.

If you set the binding’s security mode property to either Transport or TransportWithMessageCredential, the channel will require SSL protection. In this case, all traffic sent to and from the relay service will be protected via SSL, however, the message will pass through the relay service in the clear.[[19]](#footnote-20) If you want to ensure full privacy, you’ll want to use the Message security mode, in which case you can encrypt everything but the addressing information within the message passing through the relay service.

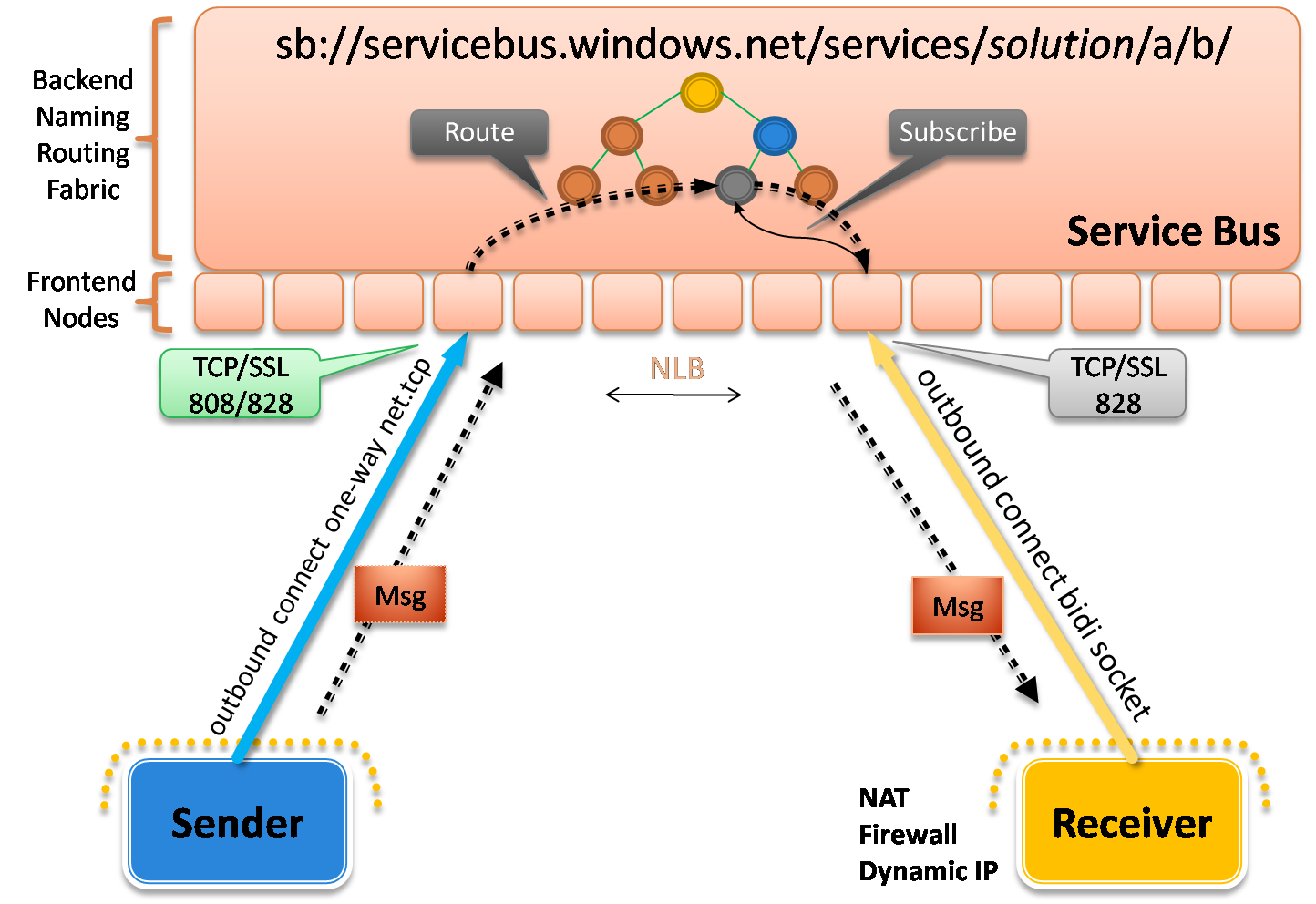


Figure : NetOnewayRelayBinding

The NetOnewayRelayBinding requires all operations on the service contract to be marked as one-way operations (IsOneWay=true). Assuming that’s the case, using this WCF binding is as simple as specifying it on your endpoint definitions and supplying the necessary credentials.

The .NET Services SDK comes with a simple sample that illustrates how the NetOnewayRelayBinding works. It defines a one-way service contract that looks like this:

[ServiceContract(Name = "IOnewayContract",

Namespace = "http://samples.microsoft.com/ServiceModel/Relay/")]

public interface IOnewayContract

{

[OperationContract(IsOneWay = true)]

void Send(int count);

}

The service implementation simply prints the supplied count to the console window. The service host application configures the service with a single endpoint using NetOnewayRelayingBinding:[[20]](#footnote-21)

<configuration>

<system.serviceModel>

<services>

<service name="Microsoft.ServiceBus.Samples.OnewayService">

<endpoint address=

"sb://servicebus.windows.net/services/pluralsight/oneway"

behaviorConfiguration="credentials"

binding="netOnewayRelayBinding"

contract="Microsoft.ServiceBus.Samples.IOnewayContract" />

</service>

</services>

...

</system.serviceModel>

</configuration>

The client application is configured with an equivalent endpoint configuration (also based on NetOnewayRelayBinding), and it sends 25 messages to the relay service through the endpoint. When you run the solution, both the client and the service host application will prompt you for your solution name and credentials. Figure 17 shows the service host application console after running the sample. All 25 messages sent by the client successfully pass through the relay and arrive at our local WCF service.

|  |  |
| --- | --- |
| NetOnewayRelayBinding_Send.jpg | NetOnewayRelayBinding_Received.jpg |

Figure : Running the NetOnewayRelayBinding Sample

When using NetOnewayRelayBinding, the receiver’s connection to the relay service is made over TCP by default. If you’re operating in a locked down network environment that doesn’t allow any outbound TCP connections beyond HTTP, you can configure the NetOnewayRelayBinding to use a more aggressive connection mode that causes the receiver to establish an HTTP connection with the relay service (instead of a raw TCP connection) for retrieving relayed messages. The ConnectivityMode enum defines the three connectivity modes you available to you. As you can see, TCP is the default mode.

|  |  |
| --- | --- |
| ConnectivityMode | Description |
| Tcp (default) | Listeners create TCP connections with the relay service through port 828 (SSL). |
| Http | Listeners create an HTTP connection with the relay service and poll for messages on an internal HTTP-based message buffer (making it easier to work around port constraints). |
| AutoDetect | This mode automatically selects between the Tcp and Http modes based on an auto-detection mechanism that probes whether either connectivity option is available for the current network environment and prefers Tcp. |

Figure : Oneway ConnectivityMode Options

When using the HTTP mode, the relay service does something a little different – it creates an in-memory message buffer behind the scenes that’s used to hold all of the relayed messages. The receiver then uses a polling technique to retrieve the relayed messages from the buffer over HTTP. Hence, the sender transmits messages over TCP while the receiver retrieves messages over HTTP (see Figure 19).

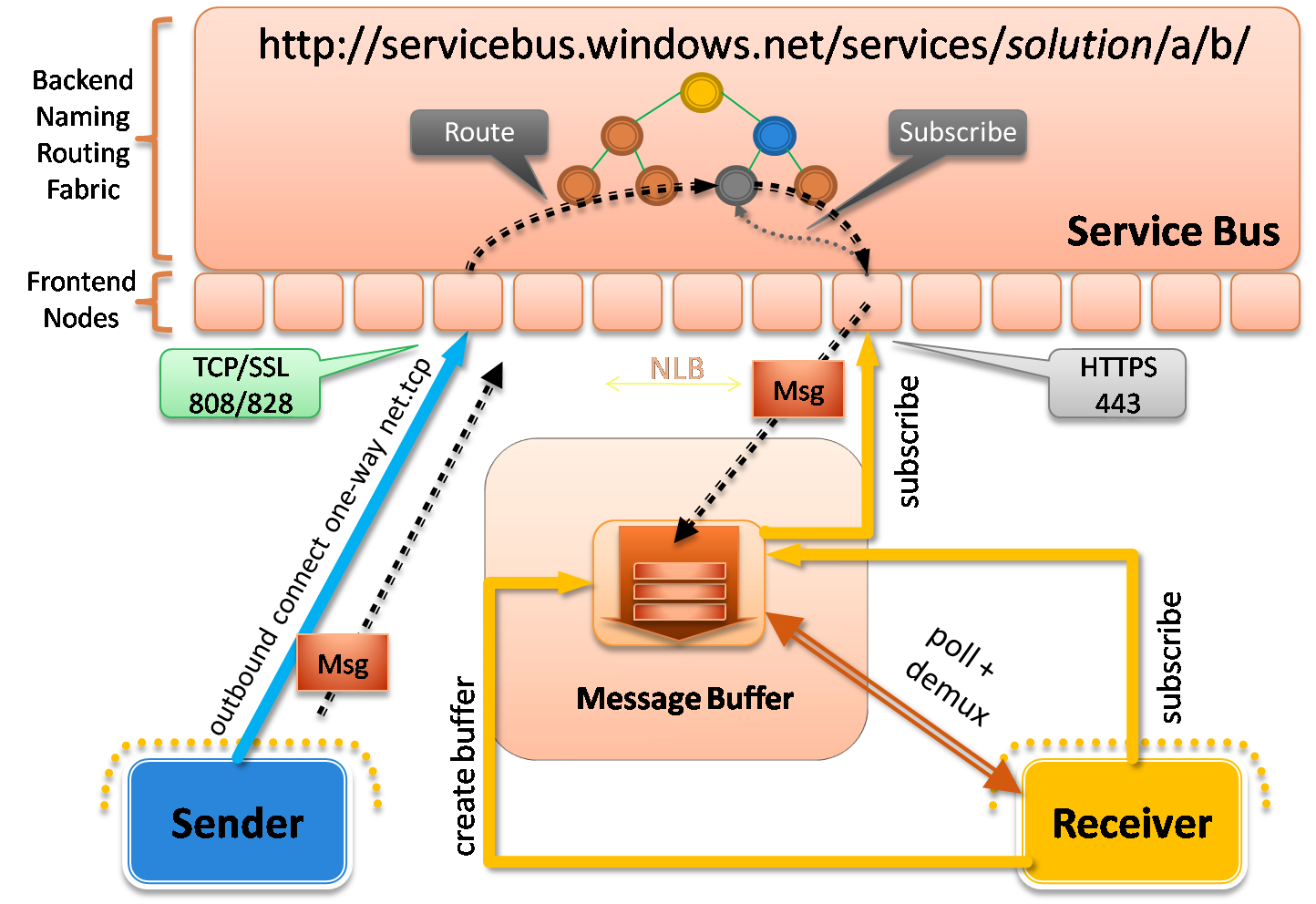


Figure : NetOnewayRelayBinding HTTP Connectivity Mode

This is all behind the scenes and transparent to use as a developer. Both the WCF sender and the receiver are simply configured to use NetOnewayRelayBinding with the HTTP connectivity mode.

The AutoDetect mode instructs WCF to automatically determine whether to use TCP or HTTP for the receiver’s connection. If TCP is possible on the given network, it prefers that mode. Otherwise, it will automatically switch to the HTTP mode in order to making things work through the firewall.

You set the one-way connectivity mode at the AppDomain-level through the static ServiceBusEnvironment class.[[21]](#footnote-22) It provides a OnewayConnectivity property where you can specify one of the three ConnectivityMode values described in Figure 18. All one-way connections use this same setting (even those created by NetEventRelayBinding endpoints). This seems like a reasonable design since all one-way connections within a particular AppDomain reside on the same network and deal with the same network challenges.

The following code illustrates how to modify the sample service host application to use the HTTP connectivity mode through the static ServiceBusEnvironment class:

...

ServiceBusEnvironment.OnewayConnectivity.Mode = ConnectivityMode.Http;

ServiceBusEnvironment.OnewayConnectivity.HttpModeMessageBufferLocation =

new Uri("http://servicebus.windows.net/services/pluralsight/" +

"92BC2784-BCEA-43dd-99A3-BE7EEE2EA87A");

ServiceHost host = new ServiceHost(typeof(OnewayService), address);

host.Open();

...

Notice how when using the HTTP mode, you must also specify an address for the internal HTTP message buffer.[[22]](#footnote-23) If you re-run the SDK sample with this code in place, you’ll see the same results as before but you’ll probably notice a little more delivery delay given how the HTTP polling mechanism works.

This approach only works for one-way messaging. If you need this aggressive approach in a bidirectional fashion, you can create a custom WCF binding that composes two NetOnewayRelayBinding connections into a composite duplex channel that uses this HTTP mode in both directions.[[23]](#footnote-24)

## NetEventRelayBinding

The NetEventRelayBinding is very similar to the NetOnewayRelayBinding in terms of how it’s implemented. The binding defaults and security options are identical to those for NetOnewayRelayBinding. In addition, the mechanics around how listeners/clients interact with the relay service are essentially the same as for the NetOnewayRelayBinding (review the previous section for details). In fact, the NetEventRelayBinding class actually derives from the NetOnewayRelayBinding class.

The main difference with the NetEventRelayBinding is that is allows you to create multiple listeners on the same .NET Service Bus address (see Figure 20). When a client sends a message to such an address, the .NET Service Bus multicasts the message to all listeners currently subscribed to that address.[[24]](#footnote-25)

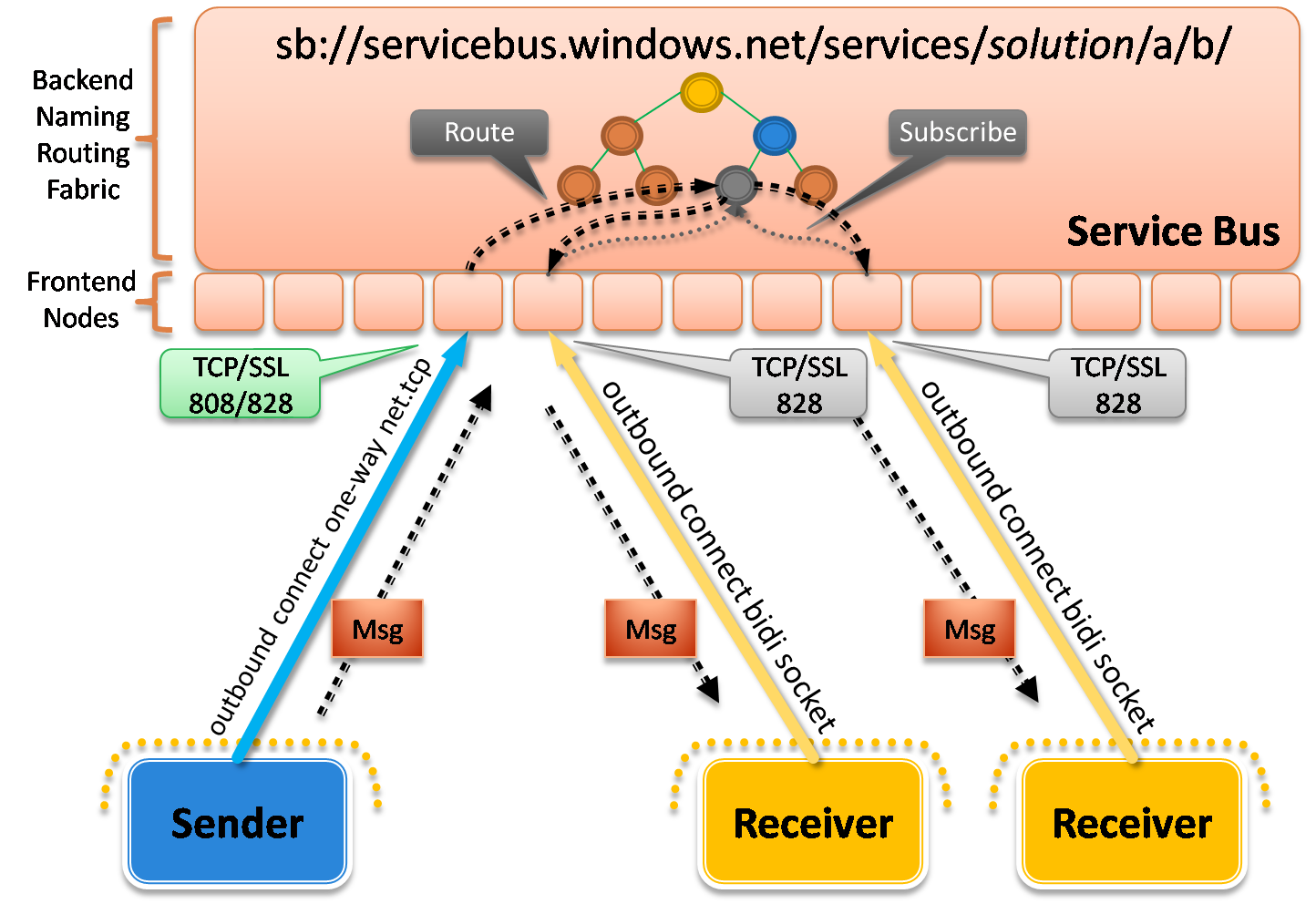


Figure : NetEventRelayBinding

This binding gives you the foundation for implementing publish/subscribe architectures but it doesn’t provide any assurances around message delivery or order. This reality makes it feel more like UDP multicasting but it’s not quite as error-prone since the relay service relies on TCP for communication.

The NetEventRelayBinding supports the same ServiceBusEnvironment.OnewayConnectivity options as the NetOnewayRelayBinding. When you configure the OneWayConnectivity property on the ServiceEnvironment class, it takes effect for all endpoints that use either one-way binding. Hence, you can use the aggressive HTTP connectivity mode for all of your NetEventRelayBinding listeners if they are being hosted within a locked-down network environment that blocks outbound TCP connections.

The .NET Services SDK comes with another sample that illustrates how to use NetEventRelayBinding to build a simple chat room application. It contains a single console application project that acts as both the sender and the receiver. Each instance of the chat room application registers a new listener within the relay service using the same address. The application then lets you enter chat messages, and it sends them to the relay service targeting the same address, at which point the relay service distributes the message to all running chat room applications subscribed to the same address. The following illustrates what the WCF configuration looks like for the chat room console application:[[25]](#footnote-26)

<configuration>

<system.serviceModel>

<client>

<endpoint name="RelayEndpoint"

contract="Microsoft.ServiceBus.Samples.IMulticastContract"

binding="netEventRelayBinding"

bindingConfiguration="default"

address="sb://servicebus.windows.net/services/pluralsight/chat/" />

</client>

<services>

<service name="Microsoft.ServiceBus.Samples.MulticastService">

<endpoint name="RelayEndpoint"

contract="Microsoft.ServiceBus.Samples.IMulticastContract"

binding="netEventRelayBinding"

bindingConfiguration="default"

address="sb://servicebus.windows.net/services/pluralsight/chat/" />

</service>

</services>

</system.serviceModel>

</configuration>

If you run three instances of the chat room application and enter some messages in one of them, you will see the messages show up in the other chat room windows as illustrated in Figure 21.

|  |  |
| --- | --- |
| Chat1.jpg | Chat2.jpg |
| Chat3.jpg | |

Figure : NetEventRelayBinding Chat Room Sample

Both the NetOnewayRelayBinding and the NetEventRelayBinding are designed for flexible unicast and multicast one-way messaging. They don’t correspond directly to any of the standard WCF bindings but are loosely related to NetMsmqBinding and NetPeerTcpBinding in terms of their messaging semantics.

## NetTcpRelayBinding

The NetTcpRelayBinding is the one I used in the first HelloServiceBus sample towards the beginning of this paper. It’s also the one Microsoft recommends that you use by default unless you have a good reason to do otherwise. The reason is because the NetTcpRelayBinding is very closely aligned with the standard WCF NetTcpBinding and has been highly optimized for efficiency and throughput. The key difference is that NetTcpRelayBinding creates a publicly-reachable TCP endpoint in the relay service.

By default, the NetTcpRelayBinding supports SOAP 1.2 over TCP and it uses binary serialization for efficiency. Although its configuration is very similar to that of the NetTcpBinding, their underlying TCP socket layers are different and are therefore not directly compatible with one another. This means that client applications will also need to be configured with NetTcpRelayBinding in order to integrate.

Let’s look at how this binding works when used with in the default “relayed” connection mode (see Figure 22). First the receiver establishes a secure outbound TCP connection with the relay service. During the process, it must authenticate, specify and address to listen on, and specify what type of listener to create within the relay. Up to this point, it’s very similar to the NetOnewayRelayBinding.

When the sender establishes a connection with the relay service and authenticates at a particular address, the relay service creates a socket-to-socket forwarder component. This component then routes a control message through the backend routing fabric down to the receiver, instructing the receiver to establish an outbound connection back to the socket forwarder. Once the receiver has connected, the relay service contains an efficient socket forwarder for relaying a bidirectional byte stream. This is precisely what happened behind the scenes in the HelloServiceBus example I showed you earlier.

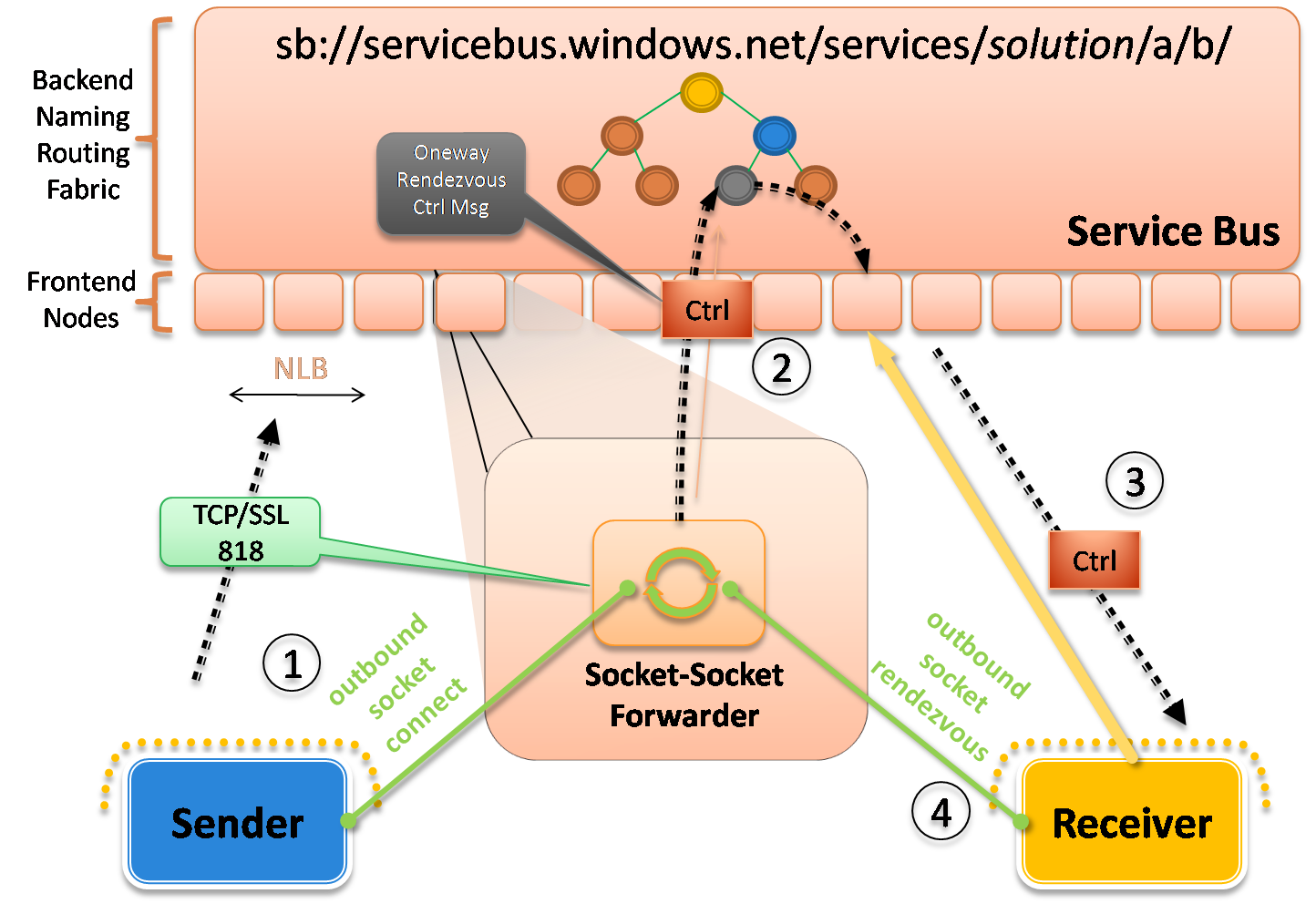


Figure : NetTcpRelayBinding – Relayed Mode

The NetTcpRelayBinding actually supports three different connection modes that control how the sender and receiver communicate with one another through the relay service (see Figure 23).

|  |  |
| --- | --- |
| TcpConnectionMode | Description |
| Relayed (default) | All communication is relayed through the relay service. The SSL-protected control connection is used to negotiate a relayed end-to-end socket connection that all communication flows through. Once the connection is established the relay service acts like a socket forwarder proxy relaying a bi-directional byte stream. |
| Hybrid | The initial communication is relayed through the relay service infrastructure while the sender and receiver negotiate a direct socket connection to each other. The coordination of this direct connection is governed by the relay service. The direct socket connection algorithm is capable of establishing direct connections between two parties that sit behind opposing firewalls and NAT devices. The algorithm uses only outbound connections for firewall traversal and relies on a mutual port prediction algorithm for NAT traversal. Once a direct connection can be established the relayed connection is automatically upgraded to a direct connection without message or data loss. If the direct connection cannot be established, data will continue to flow through the relay service as usual. |
| Direct | This mode is identical to the Hybrid mode in the CTP release. In a future release this mode will only use a direct connection, assuming one can be established, and it will never relay any application data through the relay service. |

Figure : TcpConnectionMode Options

The Relayed mode I just described is the default. It also supports an interesting Hybrid mode, which instructs the relay service to do its best to establish a *direct connection* between the sender and receiver, so no data needs to pass through the relay anymore. It’s considered a “hybrid” mode because it starts by relaying information through the relay while it attempts to upgrade to a direct connection. If successful, it will switch over to a direct connection without any data loss (see Figure 24). If it cannot establish a direct connection for whatever reason, it will continue to use the relay service.

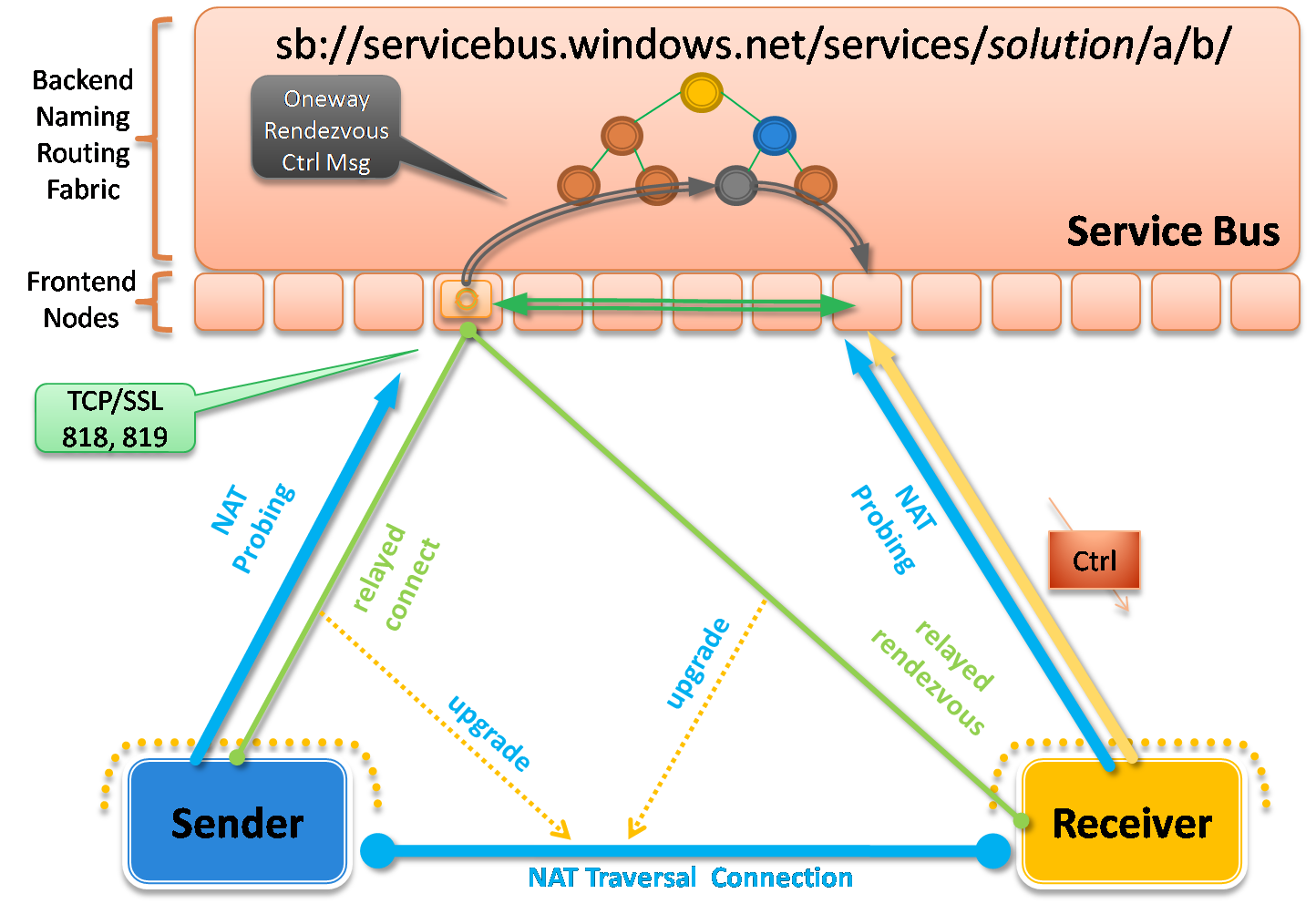


Figure : NetTcpRelayBinding – Hybrid Mode

The way the relay service accomplishes this is through a special mutual port predication algorithm based on probing information from the sender and receiver. The relay service looks at this probing information and does its best to predict what ports are going to be open on their respective NAT devices. It can then provide that information to the sender/receiver so that they can attempt to establish a direct connection with one another. If the relay service predicts correctly, the connection will succeed, otherwise it can try again until it decides to give up and to stick with the relayed connection.

This approach is similar to the approach used by many of today’s instant messaging applications when transferring files between users. Next time you use that feature, pay attention to the initial file transfer speed and whether or not it significantly speeds up at some point during the process. If you notice a significant boost in transfer speed, you just witnessed the upgrade to a direct connection.

There’s a sample that ships with the .NET Services SDK that illustrates how this works. The service is configured with a single NetTcpRelayBinding endpoint configured to use the Hybrid connection mode:[[26]](#footnote-27)

<configuration>

<system.serviceModel>

<bindings>

<netTcpRelayBinding>

<binding name="default" connectionMode="Hybrid">

<security mode="None" />

</binding>

</netTcpRelayBinding>

</bindings>

<services>

<service name="Microsoft.ServiceBus.Samples.HelloService">

<endpoint address=

"sb://servicebus.windows.net/services/pluralsight/direct"

contract="Microsoft.ServiceBus.Samples.IHelloContract"

binding="netTcpRelayBinding"

bindingConfiguration="default" />

</service>

</services>

...

</system.serviceModel>

</configuration>

The corresponding client application is also configured with an equivalent endpoint definition (using the Hybrid connection mode as well). The service implementation actually does nothing in this sample. The sample is designed to measure how many messages the client can send to the service’s “no-op” operation every 250 milliseconds. It prints the number to the client’s console window (see Figure 25).

Notice how the number of messages sent by the client dramatically jumps several seconds into running the program. This highlights exactly when the connection was upgraded to a direct connection.

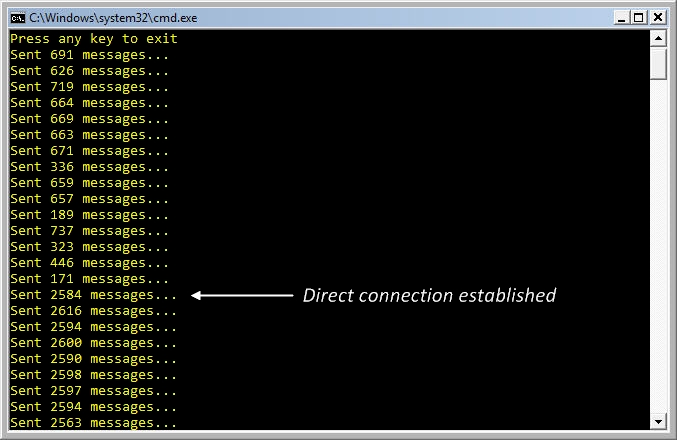


Figure : Running the NetTcpRelaying Binding "Hybrid" Sample

NetTcpRelayBinding also supports a Direct connection mode, which instructs the relay service to only use a direct connection, assuming it’s possible, and never to use the relay service for application data.[[27]](#footnote-28)

Obviously, when you expose NetTcpRelayBinding endpoints through the .NET Service Bus, they will only be consumable by WCF client applications also configured to use the NetTcpRelayBinding. Such a configuration provides the best performance and throughput but doesn’t accommodate the widest range of clients. When reach and interoperability are more important application requirements, you should turn to one of the various HTTP bindings supported by the .NET Service Bus.

## HTTP Relay Bindings

All of the bindings discussed thus far require clients to use WCF on the clients-side of the interaction. When you need non-WCF clients to integrate with your .NET Service Bus endpoints, you should pick one of the various HTTP bindings supported by the relay service. The .NET Service Bus comes with several HTTP bindings – WebHttpRelayBinding, BasicHttpRelayBinding, WSHttpRelayBinding, and WS2007HttpRelayBinding – that all offer wider reach and more interoperability because they can support any client that knows how to use the standard protocols supported by each of these bindings.

WebHttpRelayBinding and BasicHttpRelayBinding provide the greatest reach because they’re based on simple HTTP/REST and basic SOAP respectively. The other HTTP relay bindings are capable of providing additional layers of functionality through the WS-\* protocols. When you use the WSHttpRelayBinding, for example, clients will have to support the same suite of WS-\* protocols enabled on the endpoint. The main downside to the HTTP relay bindings is they do require comparatively more processing overhead.

Regardless of which HTTP relay binding you use, the mechanics of what happens within the relay service is largely the same (see Figure 26). The receiver first establishes a secure outbound TCP connection (yes, TCP connection) with the relay service. Then, clients can begin sending messages to the HTTP/SOAP endpoint exposed by the relay service using standard HTTP/SOAP techniques. This means WCF is no longer necessary on the client side – any HTTP/SOAP compatible programming library will do.

When an incoming message arrives on one of the front nodes, a control message is then routed down to the receiver indicating how to create a rendezvous connection back with the sender’s front-end node, thereby establishing a direct HTTP-to-socket forwarder for relaying the HTTP messages.[[28]](#footnote-29)

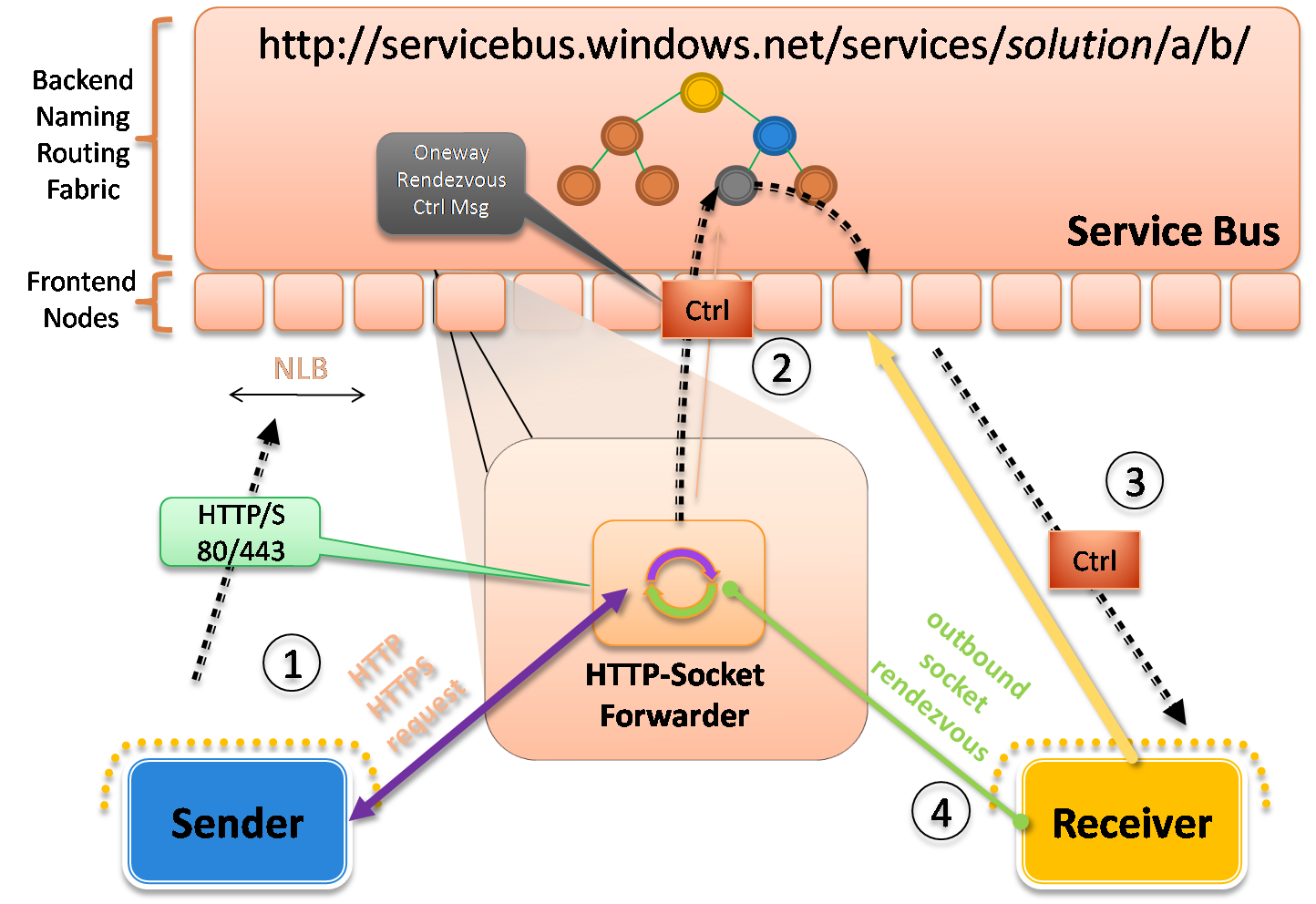


Figure : HTTP Relay Bindings

The relay service knows how to route SOAP 1.1, SOAP 1.2, and plain HTTP (REST) messages transparently. You control the messaging style and the various WS-\* protocols you wish to employ by configuring one of the HTTP relay bindings like you would any other WCF binding.

Let’s start with an example of how you can publish a RESTful service through the .NET Service Bus. The .NET Services SDK comes with a complete sample called SimpleShare that makes it possible to share files and directories on your local machine with consumers through the .NET Service Bus. The service exposes a simple RESTful service contract using the WCF 3.5 Web programming model as shown here:

[ServiceContract(Name = "ISimpleShareHttpContract",

Namespace = "http://samples.microsoft.com/SimpleShare")]

public interface ISimpleShareHttpContract

{

[OperationContract, WebGet(UriTemplate="/\*")]

Message Get();

}

The implementation of the Get method inspects the incoming HTTP request’s URI path segments and determines how to map them to the local file system. When you run the local service host application, you specify which directory you want to “share” as the starting point along with a share name. The host application is configured with a single WebHttpRelayBinding endpoint that looks something like this:[[29]](#footnote-30)

<configuration>

<system.serviceModel>

<services>

<service name="Microsoft.ServiceBus.Samples.SimpleShareService">

<endpoint name="RelayEndpoint"

contract="Microsoft.ServiceBus.Samples.ISimpleShareHttpContract"

binding="webHttpRelayBinding"

address=

"http://servicebus.windows.net/services/pluralsight/share/" />

</service>

</services>

...

</system.serviceModel>

</configuration>

As an example, I’ll run the sample specifying “c:\Demos” for the directory to share and “demoshare” for the share name. You can see what the running console application looks like in Figure 27.

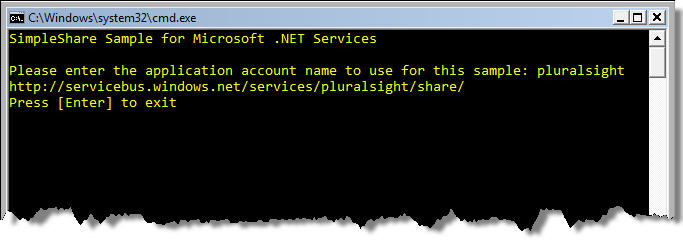


Figure : Running the SimpleShare Host Application

Now I have a WCF RESTful service running locally that I’ve published an endpoint for within the relay service. And the relay service knows how to route incoming HTTP requests to my local WCF service. With this running, anyone can browse to <http://servicebus.windows.net/services/pluralsight/share> in order to begin browsing my local shared directory, assuming they’re authorized to do so (see Figure 28).

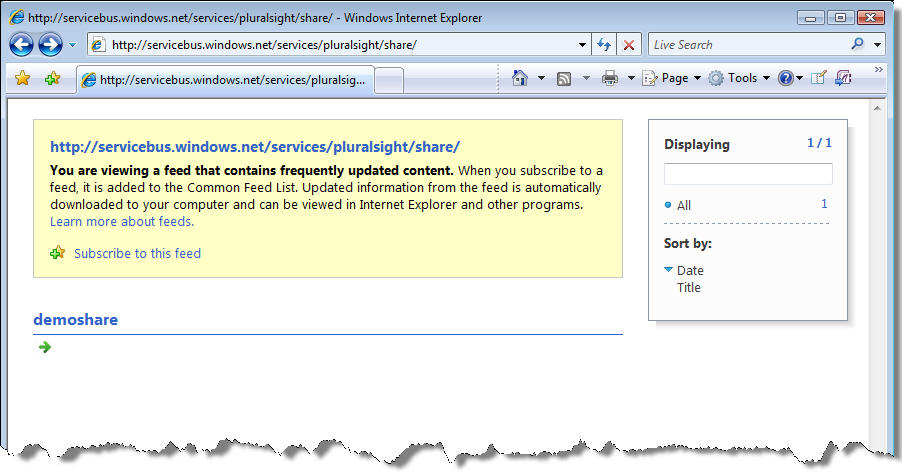


Figure : Browsing to the HTTP Endpoint

Notice how the information is being returned as an RSS feed, which is being rendered within Internet Explorer. The service allows you to continue browsing the share by clicking on directories within the feed. For example, if you click on “demoshare”, you’ll see the results shown in Figure 29.

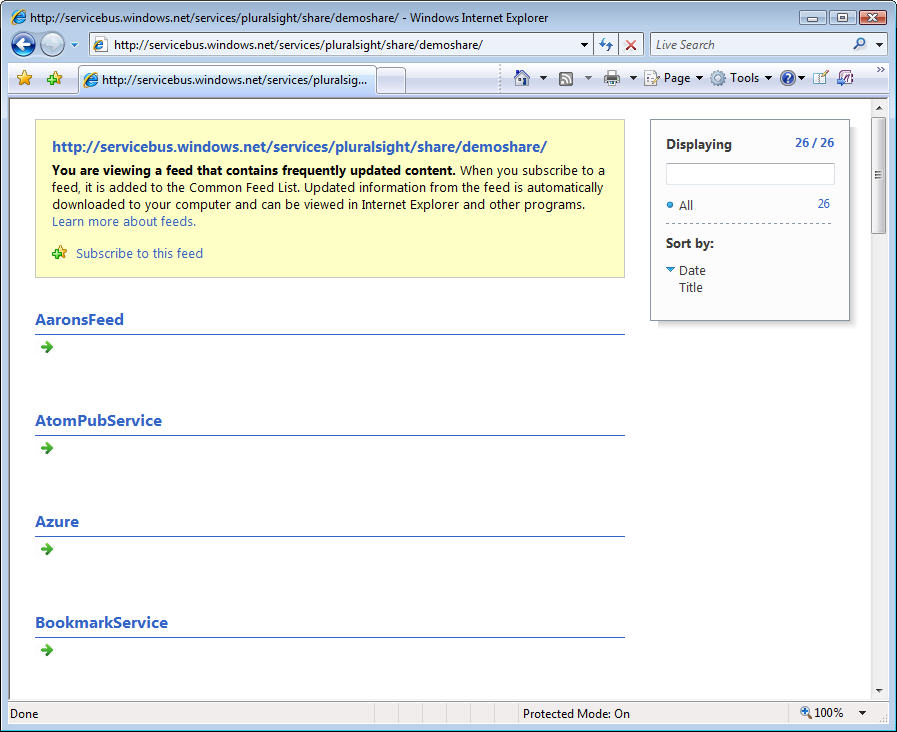


Figure : Browsing to Subdirectories within the Share

When you find a file that you’d like to retrieve (say a large zip file), you can then issue an HTTP GET request to retrieve the zip file through the relay service. In general, this SDK sample is a good example of how you can expose WCF-based RESTful services to consumers through the .NET Service Bus.[[30]](#footnote-31)

There are numerous samples in the SDK that illustrate how to use the WSHttpRelayBinding in a variety of configurations. The following example shows how to configure a WSHttpRelayBinding endpoint to use message-based security, with username tokens, and a customer validator component:

<configuration>

<system.serviceModel>

<behaviors>

<serviceBehaviors>

<behavior name="usernamePasswordServiceBehavior">

<serviceCredentials>

<serviceCertificate findValue="localhost"

storeLocation="LocalMachine"

storeName="My" x509FindType="FindBySubjectName" />

<userNameAuthentication userNamePasswordValidationMode="Custom"

includeWindowsGroups="false" customUserNamePasswordValidatorType=

"SimpleUsernamePasswordValidator, WSHttpRelayMsgSecUserNameService" />

</serviceCredentials>

</behavior>

</serviceBehaviors>

</behaviors>

<bindings>

<wsHttpRelayBinding>

<binding name="default">

<security mode="Message" relayClientAuthenticationType="None">

<message clientCredentialType="UserName"/>

</security>

</binding>

</wsHttpRelayBinding>

</bindings>

<services>

<service name="Microsoft.ServiceBus.Samples.EchoService"

behaviorConfiguration="usernamePasswordServiceBehavior">

<endpoint name="RelayEndpoint"

contract="Microsoft.ServiceBus.Samples.IEchoContract"

binding="wsHttpRelayBinding"

bindingConfiguration="default"

address=

"http://servicebus.windows.net/services/pluralsight/echo" />

</service>

</services>

...

</system.serviceModel>

</configuration>

You simply configure the WSHttpRelayBinding endpoint for security and reliable messaging features like you normally would for the WSHttpBinding. In the above example, the relay service does not perform any authentication – it simply relays the messages and lets the service take care of it.

In general, the BasicHttpRelayBinding and WSHttpRelayBinding are closely aligned with the equivalent standard WCF bindings and their various options. The main feature not supported by the WSHttpRelayBinding is transaction flow and that’s by design.[[31]](#footnote-32) Be sure to check out the other SDK samples for more examples on how you can use these HTTP bindings in various configurations.

# Access Control

The .NET Service Bus requires each receiver to be *authenticated* and *authorized* for a particular URI before it will create a listener for that receiver. Clients must also be authenticated and authorized, by default, before the .NET Service Bus will relay messages on their behalf. You can, however, disable client authentication altogether in the relay service when you configure your receiver endpoints.

## Authentication and Authorization

Authentication and authorization have traditionally been very distinct concerns. Authentication is about identifying the client whereas authorization is about deciding what each client is allowed to do. Although the .NET Service Bus allows you to handle these security concerns independently of one another, the default experience in the CTP release blurs the line between them a little.

.NET Service Bus authentication can be performed by any identity provider trusted by the .NET Access Control Service. And then it’s the job of the .NET Access Control Service to perform claim transformation logic in order to produce the claims required by the .NET Service Bus for authorization purposes.

The .NET Access Control Service implicitly trusts Windows Live ID (WLID) as an identity provider. And through the next version of Active Directory Federation Services (code-named “Geneva Server”), you’ll be able to use your own Active Directory identities as well. In addition to these identity provider options, the .NET Access Control Service includes its own built-in identity provider for solution accounts to help you get started today, but eventually this feature will go away and you will need to use a mainstream provider like WLID or Geneva Server. The built-in identity provider authenticates based on the solution name and password provided during solution creation. The .NET Access Control Service can federate with multiple identity providers at the same time, thereby acting as a central access control authority.[[32]](#footnote-33)

The .NET Service Bus has been specifically designed to trust the .NET Access Control Service. The .NET Service Bus looks for specific claims in the *security tokens* provided by senders and receivers (which they acquired from the .NET Access Control Service). There are only two *claims* it looks for in the security token: “#Listen” and “#Send”. When a receiver attempts to create a listener, the .NET Service Bus looks for the “#Listen” claim within the security token provided by the .NET Access Control Service. If it finds it, and the token hasn’t been tampered with, it authorizes the creation of a new listener. The same thing happens on the client side, only in this case, the .NET Service Bus looks for the “#Send” claim instead.

When a sender or receiver presents either a solution credential or a security token (acquired from a trusted identity provider), the .NET Access Control Service makes a rules-driven decision about whether to issue an authorization token (containing the “#Send” or “#Listen” claim) to the target service, which is in this case the .NET Service Bus. As long as you’ve provided a valid identity, the .NET Access Control Service will issue the appropriate authorization token for authorization in the .NET Service Bus.

The .NET Access Control Service signs and encrypts the resulting security token so the claims can only be read by the target service (again the .NET Service Bus). As a result, the .NET Service Bus is the only one able to decrypt the message and read its contents. Once the .NET Service Bus verifies and processes the authorization token found in the message, it removes it before relaying the message on to the receiver. The .NET Service Bus authorization token would have no meaning to the receiver anyway.

Figure 30 illustrates the .NET Service Bus access control model I’ve been describing from both the sender and receiver’s perspective. The numbered yellow boxes enumerate the steps taken by the receiver while the number blue boxes enumerate the steps taken by the sender.

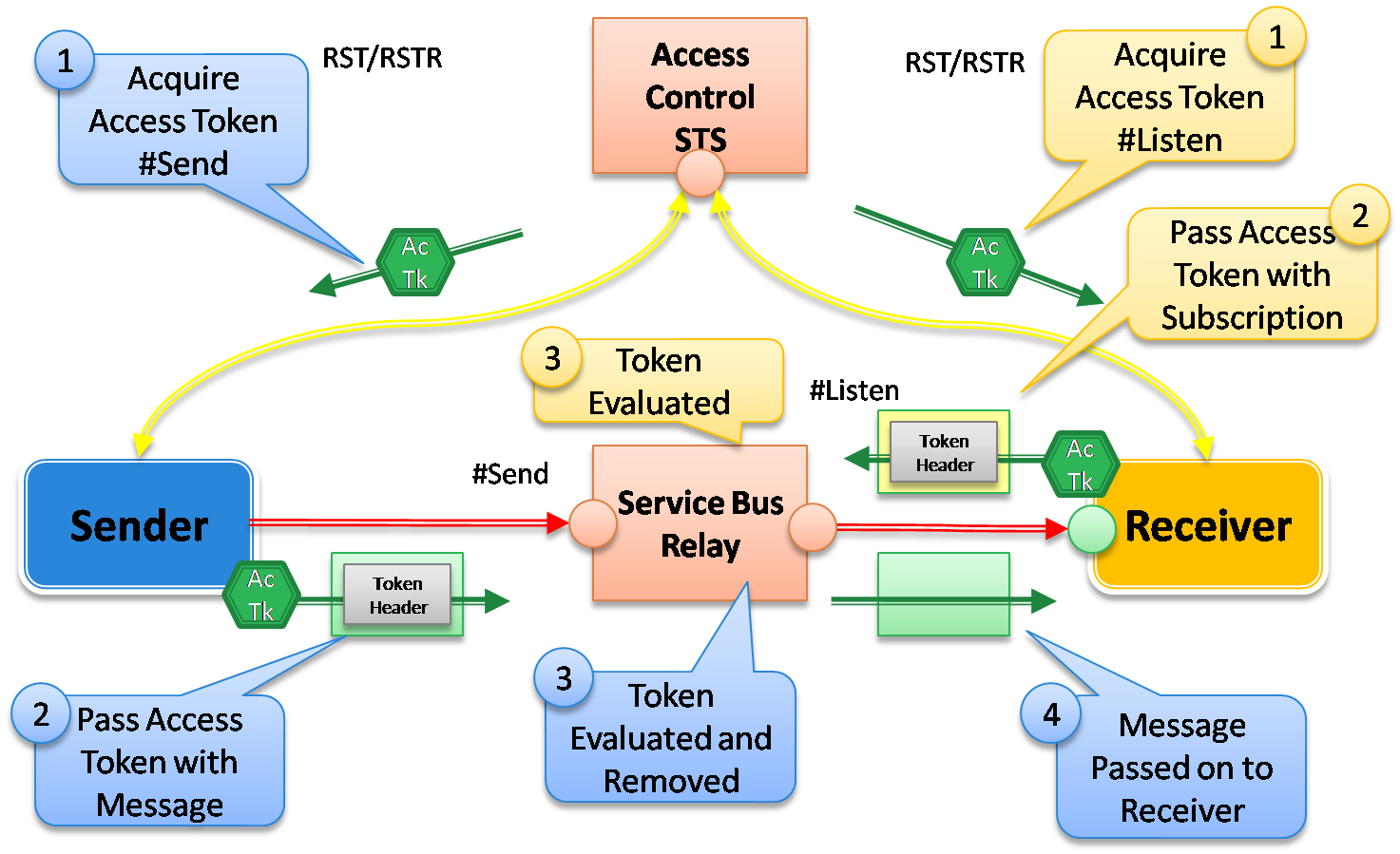


Figure : Access Control Model for the .NET Service Bus

## WCF Programming Model for Access Control

The WCF programming model for the .NET Service Bus provides a few simple abstractions that make it easy to influence the .NET Service Bus access control model shown in Figure 30. You specify what credentials to use for a particular .NET Service Bus endpoint through the TransportClientEndpointBehavior, which I showed how to use in the very first HelloServiceBus example.

The credentialType property can be set to one of the several values described in Figure 31. You can use Windows CardSpace information cards, X.509 certificates, username/password credentials, or no credentials at all. There are numerous SDK samples that illustrate these different credential options.

|  |  |
| --- | --- |
| CredentialType Value | Description |
| CardSpace | The client credential is a self-issued Windows CardSpace information card that is registered with the .NET Access Control Service. |
| AutomaticRenewal | The client credential is a self-issued Windows CardSpace information card that is registered with the .NET Access Control Service. The difference is the access token will be automatically renewed as needed. |
| FederationViaCardSpace | The client credential is a managed Windows CardSpace information card issued by an identity provider trusted by the .NET Access Control Service. |
| UserNamePasssword | The client credential is a username/password credential for the .NET Service Bus solution registered with the .NET Access Control Service. |
| X509Certificate | The client credential is a X.509 certificate for the .NET Service Bus solution that has been registered with the .NET Access Control Service. |
| Unauthenticated | No client credential is provided. |

Figure : CredentialType Values

You register Windows CardSpace information cards and X.509 certificates with the .NET Access Control Service through the Azure Services Platform portal. Once you’ve registered a Windows CardSpace information card or X.509 certificate with a solution, you can use it for authentication.

You can also control whether or not clients are required to authenticate with the .NET Service Bus by configuring the binding for the receiver. You do this through the relayClientAuthenticationType property on the <security> element supported by each relay binding. If you set the property to “None”, clients will not be required to authenticate in order to relay messages to the receiver. If you set it to “RelayAccessToken”, clients will be required to present authorization tokens to the relay service. The following shows how to reconfigure the SimpleShare example to disable client relay authentication:

<configuration>

<system.serviceModel>

<services>

<service name="Microsoft.ServiceBus.Samples.SimpleShareService">

<endpoint name="RelayEndpoint"

contract="Microsoft.ServiceBus.Samples.ISimpleShareHttpContract"

binding="webHttpRelayBinding"

bindingConfiguration="default"

address="http://servicebus.windows.net/services/pluralsight/share/" />

</service>

</services>

<bindings>

<webHttpRelayBinding>

<binding name="default">

<security relayClientAuthenticationType="None" />

</binding>

</webHttpRelayBinding>

</bindings>

...

</system.serviceModel>

</configuration>

It’s also possible to implement a “hybrid” security mode by combining message-based security (WS-Security) with a relay client authentication mode of “RelayAccessToken”. In order to accomplish this you would specify a standard security mode of “Message” (on the binding) along with “RelayAccessToken” for the relay client authentication mode. And then you’d need to configure the client with two separate credentials: one via the regular WCF ClientCredentials property (for gaining access to the target service) and the other via the TransportClientEndpointBehavior (for gaining access to the .NET Service Bus).

## Web-Style (REST) Authentication

If you set the relayClientAuthenticationType to “RelayAccessToken” when using the WebHttpRelayBinding, the relay service provides a defense layer over your RESTful HTTP service that requires clients to be authenticated and authorized through a .NET Service Bus authorization token. There are two ways to provide the required authorization token to the relay service: either through interactive credentials or programmatically through a RESTful interface.

If a client issues an HTTP request to a .NET Service Bus endpoint without providing an authorization token, the relay service will redirect the request to an interactive login page (see Figure 32). The interactive login page accepts CardSpace or username/password credentials associated with the solution. After the user enters valid credentials, the user will be redirected back to the original request, along with the required authorization token, and the request will then be successfully relayed.[[33]](#footnote-34)

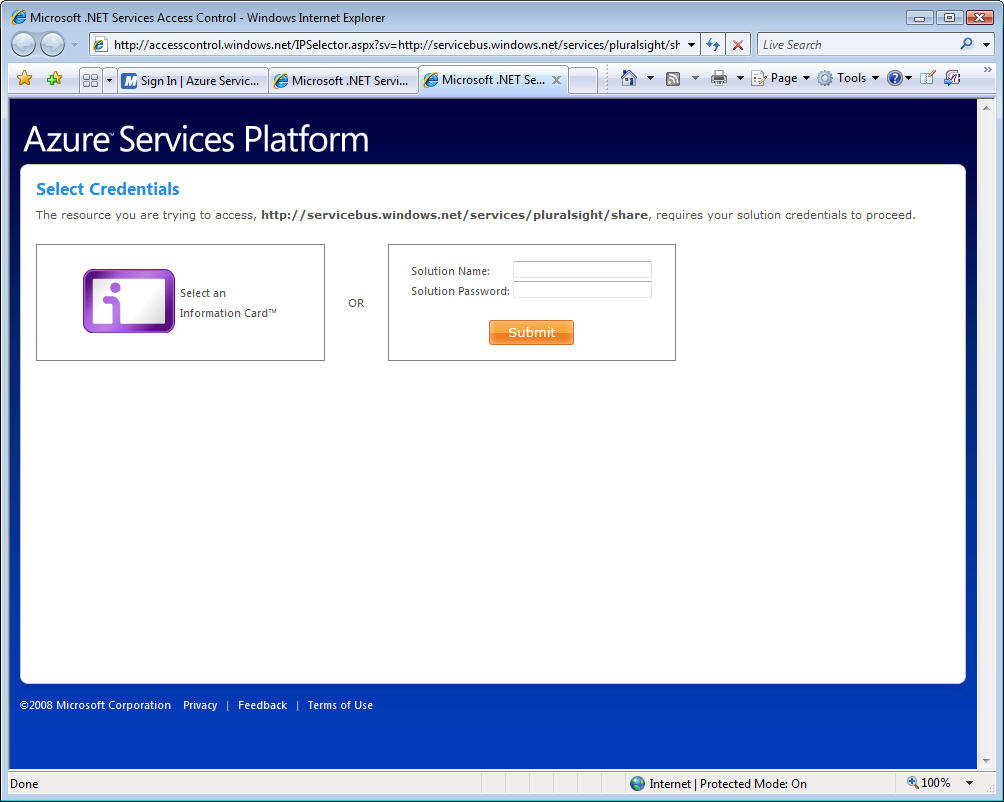


Figure : Interactive Login Page

It’s also possible to perform this authorization step programmatically through a RESTful interface to the .NET Access Control Service. Today, you can request an authorization token by simply issuing the following HTTP GET request to the .NET Access Control Service:[[34]](#footnote-35)

https://accesscontrol.windows.net/isssuetoken.aspx?u={solution-name}&p={password}

The response contains a reference cookie (in plain text format) to a token held within the .NET Access Control Service. The client can use the cookie to gain access to the relay service by adding the cookie value to outgoing HTTP requests in a custom HTTP header named “X-MS-Identity-Token”. When using this technique, Microsoft strongly recommends using HTTPS to protect the cookie value on the wire.

For more information on the .NET Access Control Service, and to learn specifically about how you can use it in conjunction with your own services (not just through the .NET Service Bus), see the accompanying whitepaper called A Developer’s Guide to the .NET Access Control Service.

# Guidance

One of the hardest questions to answer when moving towards the .NET Service Bus is: which binding should I use? This section provides some general guidance to help answer that important question.

Microsoft recommends you use the NetTcpRelayBinding by default unless you specifically need a communication feature provided by one of the other bindings. This was the binding that I used in the HelloServiceBus example towards the beginning of the paper. The NetTcpRelayBinding provides the most compact wire format and the most efficient communication path through the relay service infrastructure. It will therefore yield the highest throughput of all the WCF relay bindings and it will impose the lowest amount of processing overhead on the listening service.

There are several connectivity modes to choose from when using the NetTcpRelayBinding. If your scenario requires transferring large messages in a peer-to-peer fashion, you should prefer the Hybird/Direct connectivity modes over Relayed. However, remember that you won’t be able to use certain features such as WS-ReliableMessaging and WS-Security with the Hybrid/Direct modes. Hence, Relayed is a good default connection mode unless you’re specifically trying to optimize P2P transfers.

If you’re working in locked-down network environments, the NetTcpRelayBinding may not work given the outbound TCP ports it required by listeners. In situations like this, you’ll want to turn to the NetOnewayRelayBinding or the NetEventRelayBinding. These bindings provide the most aggressive NAT/firewall traversal options because they provide an optional HTTP connectivity mode that enables listeners to receive messages over HTTP ports (80/443) that are usually unblocked. The main scenario for choosing the NetEventRelayBinding is when you need event distribution for multicast scenarios. [[35]](#footnote-36)

The NetOneWayRelayBinding and NetEventRelayBinding are constrained to one-way messaging. Hence, if you need request/response style messaging, these bindings aren’t the best fit. And unless you need their aggressive traversal capabilities (e.g., in a locked-down network), the NetTcpRelayBinding should probably be your preferred choice even for one-way communication due to its more efficient behavior.

When your requirements call for interoperability with non-Microsoft clients, the NetTcpRelayBinding, NetOnewayRelayBinding, and NetEventRelayBinding are not going to work. This is when you should turn to the HTTP relay bindings. The HTTP relay bindings provide the highest-degree of interoperability with the rest of the world because they expose endpoints based on open standards including HTTP, SOAP 1.1, and SOAP 1.2. When you use the HTTP relay bindings, any compatible Web services client will be able to communicate with your .NET Service Bus endpoints. The primary downside to using the HTTP relay bindings is they come with the greatest amount of processing overhead on the listening service.[[36]](#footnote-37)

Choosing between the various HTTP relay bindings boils down to what style of service you’re trying to build (REST vs. SOAP) and what additional Web services protocols you want to support (WS-\*). If your requirements call for REST, you want to use the WebHttpRelayBinding. If your requirements call for SOAP, you have several choices. You’ll want to use the BasicHttpRelayBinding when you’re building basic profile compliant SOAP services, and the WSHttpRelayBinding/WS2007HttpRelayBinding when you’re building more advanced SOAP-based services that leverage the layered WS-\* protocols.

In the area of security, Microsoft encourages you to take full advantage of message-based security techniques by signing and encrypting your messages that will pass through the service relay (this is the “hybrid” mode I referred to in the Access Control section). The relay security model composes nicely with WS-Security thereby making it possible to ensure privacy and tamper protection. The service relay does not read or touch any of the information within the body of your messages – it only looks at the URI and, in some cases, the WS-Addressing headers in order to determine how to relay the message.

One final important note: the CTP release of the .NET Service Bus does not support hosting your on-premise WCF services within IIS/WAS. As mentioned earlier, the local service host process is responsible for initiating the relationship with the relay service. Since IIS/WAS services rely on a process activation model, the host process doesn’t actually run until the first message arrives. This presents an interesting dilemma when using the .NET Service Bus. Microsoft is actively working on a solution for a future release but as of the CTP release, you should avoid IIS/WAS hosting when using the .NET Service Bus.

# Summary

The .NET Service Bus provides a secure, standards-based messaging fabric for connecting applications across the Internet, making it possible to traverse difficult firewall and NAT obstacles when necessary. It consists of global naming system, a discoverable service registry, numerous messaging and connectivity options, and a claims-based access control mechanism. .NET developers can take advantage of the .NET Service Bus by simply choosing from a new set of WCF bindings and the rest of their WCF code remains essentially the same. Ultimately, the messaging fabric offered by the .NET Service Bus dramatically reduces the barriers for making software + services a reality on the Windows platform.

# Additional Resources

We’ve provided links to several resources below that will further your education on the suite of Microsoft® .NET Services and the .NET Service Bus in particular.

## Microsoft® .NET Services Whitepaper Series

* An Introduction to Microsoft .NET Services for Developers
  + <http://go.microsoft.com/?linkid=9638347>
* A Developer’s Guide to the Microsoft® .NET Service Bus (*this paper*)
  + <http://go.microsoft.com/?linkid=9638348>
* A Developer’s Guide to the Microsoft® .NET Access Control Service
  + <http://go.microsoft.com/?linkid=9638349>
* A Developer’s Guide to the Microsoft .NET Workflow Service
  + <http://go.microsoft.com/?linkid=9638350>

## .NET Service Bus Resources

* Azure Services Platform
  + <http://www.microsoft.com/azure/services.mspx>
* Register for Azure Services
  + <http://www.microsoft.com/azure/register.mspx>
* Microsoft® .NET Services
  + <http://www.microsoft.com/azure/netservices.mspx>
* Java SDK for Microsoft .NET Services
  + <http://www.jdotnetservices.com/>

# About the Author

Aaron Skonnard is a cofounder of Pluralsight, a premier Microsoft .NET training provider offering both instructor-led and online training courses. Aaron is the author of numerous books, articles, and whitepapers, as well as Pluralsight’s REST, Windows Communication Foundation, and BizTalk courses. Aaron has spent years developing courses, speaking at conferences, and teaching developers throughout the world. You can reach him at <http://pluralsight.com/aaron>.

# Acknowledgements

Clemens Vasters deserves most of the credit for the content found in this whitepaper. His recent PDC sessions, initial documentation drafts, and other writings set the vision for what you see here. Thanks to Clemens for his willing assistance and illustrative figures that made my job that much easier.

1. Microsoft® .NET Services is the new, more appropriate name, for the original *BizTalk Services* initiative. [↑](#footnote-ref-2)
2. For more information on Microsoft® .NET Services, the .NET Access Control Service, and the .NET Workflow Service, see the accompanying papers in the Microsoft .NET Services Whitepaper Series referenced at the end of this paper. [↑](#footnote-ref-3)
3. This terminology was used in the BizTalk Services documentation but is no longer an official term used by Microsoft. [↑](#footnote-ref-4)
4. RIA = Rich Internet Application [↑](#footnote-ref-5)
5. No one refutes the fact that the Internet is full of bad guys. These days, if you have anything of value, you must design against the threat of being attacked. Hence, most corporate environments shield themselves heavily with multiple firewall layers. [↑](#footnote-ref-6)
6. An inbound port is required for an external node to initiate communication with an application sitting within the firewall. [↑](#footnote-ref-7)
7. See Introduction to Microsoft .NET Services for Developers for a complete walkthrough of how to get started. [↑](#footnote-ref-8)
8. This is only possible if you have multiple invitation codes. Each solution requires a unique invitation code. [↑](#footnote-ref-9)
9. You will receive a password after creating your .NET Service Bus solution. You use the solution name for the username. [↑](#footnote-ref-10)
10. The fact that you can define name hierarchies within your solutions gives you the opportunity to further delegate control over specific sub-trees to different groups or teams within your organization. [↑](#footnote-ref-11)
11. You would use “http” instead of “sb” if these names identified HTTP-based endpoints. [↑](#footnote-ref-12)
12. There has also been some discussion about making it possible to grow the domain name to the left of the solution name. As an example, the Boston office’s payment service could be named as sb://boston.pluralsight.servicebus.windows.net/payment. Microsoft has said nothing official about this feature but it’s interesting to note that it’s possible within the naming system. [↑](#footnote-ref-13)
13. This can be done programmatically by simply issuing an HTTP GET request to the solution’s base address. [↑](#footnote-ref-14)
14. The .NET Services SDK comes with a NameSettings endpoint behavior that allows you to control whether a given endpoint is published in the service registry. You do this by setting the behavior’s DiscoveryType property to either DiscoveryType.Public or DiscoveryType.Private, and you can also control the published endpoint name via the behavior’s DisplayName property. [↑](#footnote-ref-15)
15. You will need to provide a security token (acquired from the .NET Access Control Service) when interacting with the service registry in this way. Microsoft has not yet provided full documentation on how to use APP or WS-Transfer to interact with the service registry. Look for that to become available in one of the future releases. [↑](#footnote-ref-16)
16. It’s important to note that the CTP release does not support IIS/WAS activation scenarios for hosting local services. [↑](#footnote-ref-17)
17. Microsoft has applied for well-known .NET Service Bus port number through the Internet Assigned Numbers Authority (IANA), the organization responsible for global coordination of IP addressing and Internet protocols. Once they’ve received the official port numbers assigned by IANA, they will update the .NET Service Bus implementation accordingly. [↑](#footnote-ref-18)
18. The NetOnewayRelayBinding currently constrains messages to 60 KB on the wire. [↑](#footnote-ref-19)
19. Although visible, Microsoft has stated that the relay service does not observe the messages passing through it. [↑](#footnote-ref-20)
20. Note that I’ve inserted an example address for one of my solutions here. The SDK sample prompts you for a solution name and dynamically generates the address at runtime. [↑](#footnote-ref-21)
21. ServiceBusEnvironment is a static class providing access to common environment parameters. It provides information about the current host names for .NET Service Bus and .NET Access Control endpoints and to the global ConnectivitySettings for all NetOnewayRelayBinding and NetEventRelayBinding based endpoints that are active in the current application domain. [↑](#footnote-ref-22)
22. The address of the HTTP message buffer must refer to a location within in the current solution’s namespace. It’s a good idea to use a unique identifier, such as a GUID, to avoid collisions with other endpoints within the same solution. [↑](#footnote-ref-23)
23. This is a more advanced topic beyond the scope of this whitepaper. Look for more information on this scenario, and perhaps a sample, in a future release of the .NET Services SDK. [↑](#footnote-ref-24)
24. In the CTP release, the NetEventRelayBinding constrains you to a maximum of 20 concurrent listeners. [↑](#footnote-ref-25)
25. Note that I’ve inserted an example address for one of my solutions here. The SDK sample prompts you for “chat session” name and uses it to dynamically generate the event distribution address at runtime. [↑](#footnote-ref-26)
26. Note that I’ve inserted an example address for one of my solutions here. The SDK sample prompts you for a solution name and dynamically generates the address at runtime. [↑](#footnote-ref-27)
27. In the CTP release, the Direct mode works just like the Hybrid mode but it will work as desribed in a future release. Also, WS-ReliableMessaging does not work properly in conjunction with Hybrid/Direct at this point in time. [↑](#footnote-ref-28)
28. This is essentially the opposite of what happens with the NetOnewayRelayBinding when using the HTTP connection mode. In that case, the sender creates a TCP connection with the relay service while the receiver pulls messages down over HTTP. If you want the receiver to also use HTTP, you’ll have to create a custom composite binding based on two one-way HTTP channels. [↑](#footnote-ref-29)
29. Note that I’ve inserted an example address for one of my solutions here. The SDK sample prompts you for a solution name and dynamically generates the address at runtime. [↑](#footnote-ref-30)
30. In the CTP release, the WebHttpRelayBinding is not fully compatible with the WebScriptEnablingBehavior. The special endpoint that generates the JavaScript proxy is not reachable through the .NET Service Bus at this time. [↑](#footnote-ref-31)
31. Transaction flow requires the distributed transaction coordinators of the communicating parties to establish a communication path to each other. When going through the .NET Service Bus, establishing this relationship is very challenging.  In addition, since atomic transactions require resources to be locked, it’s general not a recommended practice at Internet scope where latency is an issue. Thus, transaction flow is not supported by any of the bindings at this time. [↑](#footnote-ref-32)
32. Most of the samples that ship with the .NET Services SDK use the identity provider built into the .NET Access Control Service. The fact that the .NET Access Control Service comes with a built-in identity provider is what makes the authentication/authorization process feel like a single step and can be somewhat misleading. If you use a different identity provider, the distinction will definitely be clearer. [↑](#footnote-ref-33)
33. This interactive login page is temporary solution that is already scheduled to be replaced in a future release with a more flexible version that allows using federated identity providers and their respective login sites. [↑](#footnote-ref-34)
34. The GET endpoint is also a temporary solution made available in the CTP release for experimentation purposes. [↑](#footnote-ref-35)
35. The NetEventRelayBinding is primarily focused on application integration scenarios and currently optimized for a small number of event subscribers (up to 10). It’s not designed for handling consumer scenarios with significantly larger subscriber bases. Also, in the CTP release, all solutions are throttled to a maximum of 5 endpoints listening concurrently. [↑](#footnote-ref-36)
36. The HTTP connection path is currently optimized for transferring larger payloads with moderate request rates and not for high request rate scenarios. [↑](#footnote-ref-37)