WINDOWS AZURE SERVICE BUS

Whether software runs in the cloud or on premises, it often needs to interact with other software. To provide a broadly useful way to do this, Windows Azure offers Service Bus. This article takes a look at this technology, describing what it is and why you might want to use it.

Contents

Service Bus Fundamentals

Different situations call for different styles of communication. Sometimes, letting applications send and receive messages through a simple queue is the best solution. In other situations, an ordinary queue isn't enough; a queue with a publish-and-subscribe mechanism is better. And in some cases, all that's really needed is a connection between applications—queues aren't required. Service Bus provides all three options, letting your applications interact in several different ways.

Service Bus is a multi-tenant cloud service, which means that the service is shared by multiple users. Each user, such as an application developer, creates a *namespace*, then defines the communication mechanisms she needs within that namespace. Figure 1 shows how this looks.

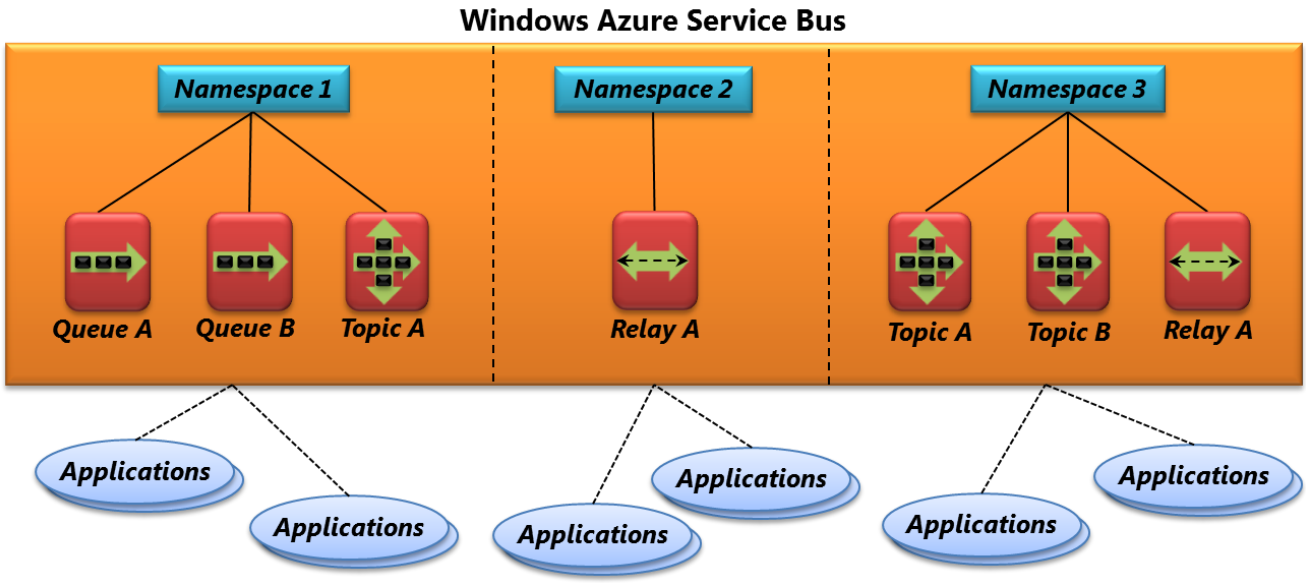

Figure 1: Service Bus provides a multi-tenant service for connecting applications through the cloud.

Within a namespace, you can use one or more instances of three different communication mechanisms, each of which connects applications in a different way. The choices are:

- *Queues*, which allow one-directional communication. Each queue acts as an intermediary (sometimes called a *broker*) that stores sent messages until they are received.
- *Topics*, which provide one-directional communication using *subscriptions*. Like a queue, a topic acts as a broker, but it allows each subscription to see only messages that match specific criteria.
- *Relays*, which provide bi-directional communication. Unlike queues and topics, a relay doesn't store in-flight messages—it's not a broker. Instead, it just passes them on to the destination application.

When you create a queue, topic, or relay, you give it a name. Combined with whatever you called your namespace, this name creates a unique identifier for the object. Applications can provide this name to Service Bus, then use that queue, topic, or relay to communicate with one another.

To use any of these objects, Windows applications can use Windows Communication Foundation (WCF). For queues and topics, Windows applications can also use a Service Bus-defined Messaging API. Queues and topics can be accessed via HTTP as well, and to make them easier to use from non-Windows applications, Microsoft provides SDKs for Java, Node.js, and other languages.

It's important to understand that even though Service Bus itself runs in the cloud (that is, in Microsoft's Windows Azure datacenters), applications that use it can run anywhere. You can use Service Bus to connect applications running on Windows Azure, for example, or applications running

inside your own datacenter. You can also use it to connect an application running on Windows Azure or another cloud platform with an on-premises application or with tablets and phones. It's even possible to connect household appliances, sensors, and other devices to a central application or to one other. Service Bus is a generic communication mechanism in the cloud that's accessible from pretty much anywhere. How you use it depends on what your applications need to do.

Queues

Suppose you decide to connect two applications using a Service Bus queue. Figure 2 illustrates this situation.

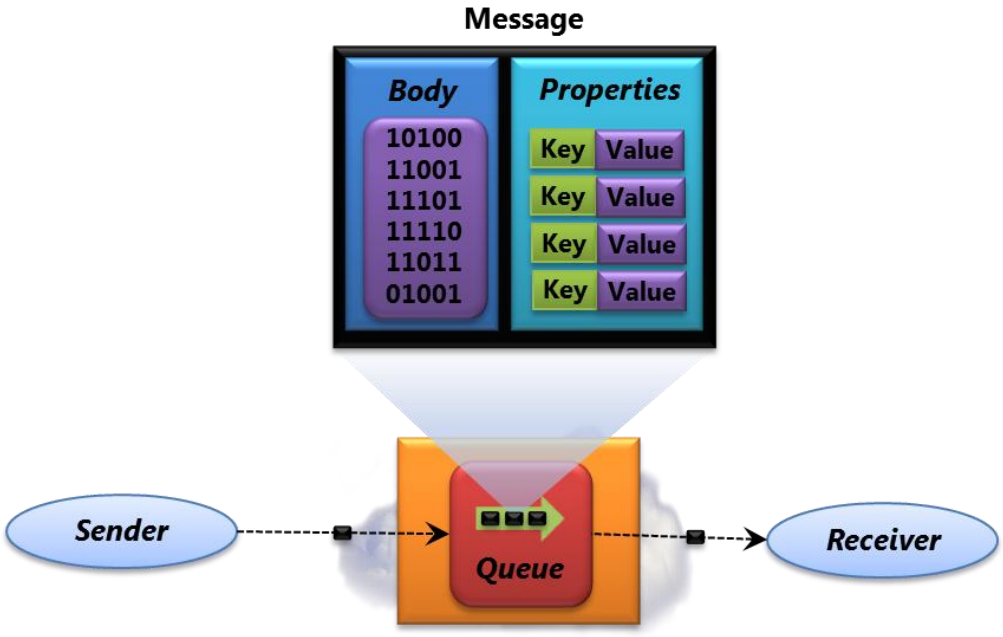

Figure 2: Service Bus queues provide one-way asynchronous queuing.

The process is simple: A sender sends a message to a Service Bus queue, and a receiver picks up that message at some later time. A queue can have just a single receiver, as Figure 2 shows, or multiple applications can read from the same queue. In the latter situation, each message is typically read by just one receiver—queues don't provide a multi-cast service.

Each message has two parts: a set of properties, each a key/value pair, and a binary message body. How they're used depends on what an application is trying to do. For example, an application sending a message about a recent sale might include the properties *Seller="Ava"* and *Amount=10000*. The message body might contain a scanned image of the sale's signed contract or, if there isn't one, just remain empty.

A receiver can read a message from a Service Bus queue in two different ways. The first option, called ReceiveAndDelete, removes a message from the queue and immediately deletes it. This is simple, but if the receiver crashes before it finishes processing the message, the message will be lost. Since it's been removed from the queue, no other receiver can access it.

The second option, PeekLock, is meant to help with this problem. Like ReceiveAndDelete, a PeekLock read removes a message from the queue. It doesn't delete the message, however. Instead, it locks the message, making it invisible to other receivers, then waits for one of three events:

- If the receiver processes the message successfully, it calls Complete, and the queue deletes the message.
- If the receiver decides that it can't process the message successfully, it calls Abandon. The queue then removes the lock from the message and makes it available to other receivers.
- If the receiver calls neither of these within a configurable period of time (by default, 60 seconds), the queue assumes the receiver has failed. In this case, it behaves as if the receiver had called Abandon, making the message available to other receivers.

Notice what can happen here: The same message might be delivered twice, perhaps to two different receivers. Applications using Service Bus queues must be prepared for this. To make duplicate detection easier, each message has a unique MessageID property that by default stays the same no matter how many times the message is read from a queue.

Queues are useful in quite a few situations. They let applications communicate even when both aren't running at the same time, something that's especially handy with batch and mobile applications. A queue with multiple receivers also provides automatic load balancing, since sent messages are spread across these receivers.

Topics

Useful as they are, queues aren't always the right solution. Sometimes, Service Bus topics are better. Figure 3 illustrates this idea.

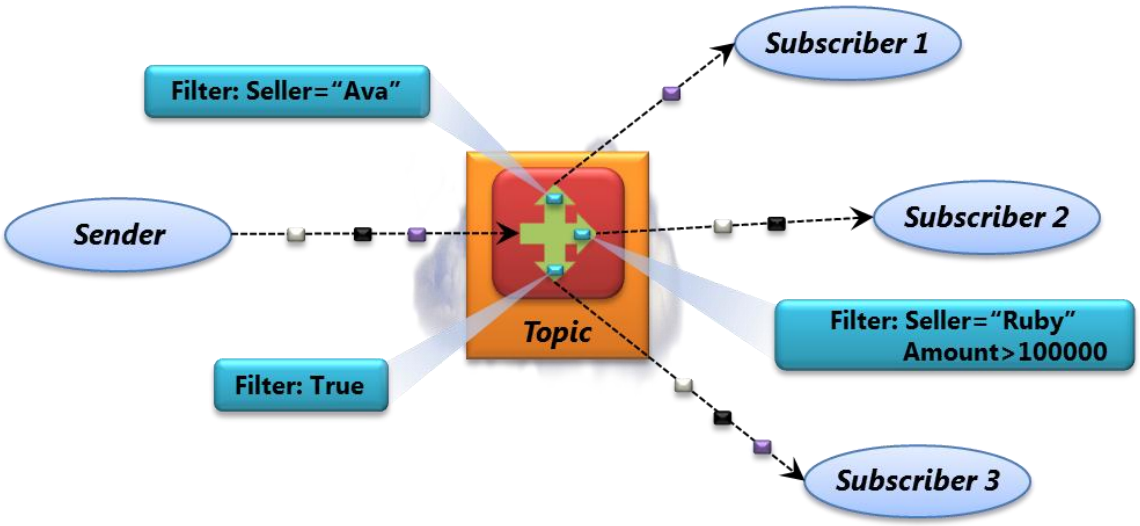

Figure 3: Based on the filter a subscribing application specifies, it can receive some or all of the messages sent to a Service Bus topic.

A topic is similar in many ways to a queue. Senders submit messages to a topic in the same way that they submit messages to a queue, and those messages look the same as with queues. The big difference is that topics let each receiving application create its own subscription by defining a *filter*. A subscriber will then see only the messages that match that filter. For example, Figure 3 shows a sender and a topic with three subscribers, each with its own filter:

- Subscriber 1 receives only messages that contain the property *Seller="Ava"*.
- Subscriber 2 receives messages that contain the property *Seller="Ruby"* and/or contain an *Amount* property whose value is greater than 100000. Perhaps Ruby is the sales manager, and so she wants to see both her own sales and all big sales regardless of who makes them.
- Subscriber 3 has set its filter to *True*, which means that it receives all messages. This application might be responsible for maintaining an audit trail, for example, and so it needs to see everything.

As with queues, subscribers to a topic can read messages using either ReceiveAndDelete or PeekLock. Unlike queues, however, a single message sent to a topic can be received by multiple subscribers. This approach, commonly called *publish and subscribe*, is useful whenever multiple applications might be interested in the same messages. By defining the right filter, each subscriber can tap into just the part of the message stream that it needs to see.

Relays

Both queues and topics provide one-way asynchronous communication through a broker. Traffic flows in just one direction, and there's no direct connection between senders and receivers. But what

if you don't want this? Suppose your applications need to both send and receive, or perhaps you want a direct link between them—you don't need a place to store messages in between. To address problems like this, Service Bus provides relays, as Figure 4 shows.

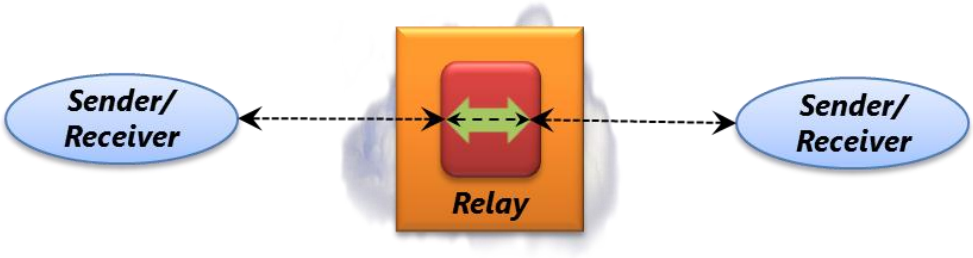

Figure 4: Service Bus relay provides synchronous, two-way communication between applications.

The obvious question to ask about relays is this: Why would I use one? Even if I don't need queues, why make applications communicate via a cloud service rather than just interact directly? The answer is that talking directly can be harder than you might think.

Suppose you want to connect two on-premises applications, both running inside corporate datacenters. Each of these applications sits behind a firewall, and each datacenter probably uses network address translation (NAT). The firewall blocks incoming data on all but a few ports, and NAT implies that the machine each application is running on probably doesn't have a fixed IP address. Without some extra help, connecting these applications over the public Internet is problematic.

A Service Bus relay provides this help. To communicate bi-directionally through a relay, each application establishes an outbound TCP connection with Service Bus, then keeps it open. All communication between the two applications will travel over these connections. Because each connection was established from inside the datacenter, the firewall will allow incoming traffic to each application—data sent through the relay—without opening new ports. This approach also gets around the NAT problem, because each application has a consistent endpoint throughout the communication. By exchanging data through the relay, the applications can avoid the problems that would otherwise make communication difficult.

To use Service Bus relays, applications rely on Windows Communication Foundation (WCF). Service Bus provides WCF bindings that make it straightforward for Windows applications to interact via relays. Applications that already use WCF can typically just specify one of these bindings, then talk to each other through a relay. Unlike queues and topics, however, using relays from non-Windows applications, while possible, requires some programming effort; no standard libraries are provided.

Unlike queues and topics, applications don't explicitly create relays. Instead, when an application that wishes to receive messages establishes a TCP connection with Service Bus, a relay is created automatically. When the connection is dropped, the relay is deleted. To let an application find the

relay created by a specific listener, Service Bus provides a registry that allows locating a specific relay by name.

Relays are the right solution when you need direct communication. Think about an airline reservation system running in an on-premises datacenter, for example, that must be accessed from check-in kiosks, mobile devices, and other computers. Applications running on all of these systems could rely on Service Bus relays in the cloud to communicate, wherever they might be running.

Connecting applications has always been part of building complete solutions, and it's hard to see this problem ever going away. By providing cloud-based technologies for doing this—queues, topics, and relays—Service Bus aims at making this essential function easier and more broadly available.

About the Author

David Chappell is Principal of Chappell & Associates (www.davidchappell.com) in San Francisco, California. Through his speaking, writing, and consulting, he helps people around the world understand, use, and make better decisions about new technologies.