CHAPTER 7

Notification Services and the Service Broker

In the last chapter, I explained how you can use a feature called Replication to send data from one system to another. That discussion was framed in regard to high availability, but of course you can use Replication for many other applications that require data interchange.

In this chapter, we examine a more targeted approach for data distribution. Using Notification Services, you can trigger specific datagrams to users based on conditions in your database, from notifying someone that a stock price has changed to alerting that a machine has just gone offline. Any type of event that you track in your database is available for use in Notification Services. Unlike Replication, you can use Notification Services to send data to an e-mail address, a cell phone, or using the *Simple Messaging Service* (SMS) to any device that conforms to that standard.

There are a couple of ways you can use Notification Services. You can treat it primarily as a task that the maintenance *database administrator* (DBA) is responsible for or you can enable it for your developers. In this chapter, I focus on the DBA side of things, but you should be aware of its many applications.

You can use SQL Server 2005 as part of a robust *service-oriented architecture* (SOA). An SOA allows you to build applications in a new way, distributing the load across multiple servers using messages between them. SQL Server 2005 provides the Service Broker as a programmable object that you can use as a store-and-forward mechanism. Although you might not write these programs, as the DBA you will be responsible to set up and manage this part of the framework for an SOA.

In the second part of the chapter, I explain how you can help develop and manage an SOA using the Service Broker. I show you a sample application using the Service Broker in the "Take Away" section at the end of the chapter.

In the early days of computing, users did not get immediate responses from their programs. Developers wrote code in either machine language or something close to it, saved it on a media such as magnetic tape or punch cards, and submitted it to a computer. The computer would run the program and create an output, either more punch cards or a printout. The users would receive the printouts as their results, disconnected from the computer. This is called "batch processing"; and most of us could not imagine working this way any more.

In modern applications, users connect to the database either directly or through a data-access layer in the system. They input data and receive immediate feedback using a video screen. This book was produced with a program using that computing method, and you might even be reading it that way.

But there are times when batch processing or disconnected computing is useful. In some applications, the user is not near the computer when the results are tabulated or created but may still need the data. Other systems might also need access to the data but not all at once. The feedback between the data and the user does not have to be immediate.

The value in this paradigm is that you can spread out the system over large areas and balance the loads between the components. Not needing the immediate feedback lowers the amount of resources a system requires.

You are already using at least one disconnected system in your daily routine. E-mail, for instance, is created on one computer at a certain time, stored on another computer at a later time, and picked up and read on still another computer at a later time. The e-mail server provides the reception, storage, and delivery of the message as a service. In effect, e-mail is a part of an SOA. In an SOA, a server provides interfaces and data that are available to any program that knows how to talk to the service. The service can also be coded to accept input, or perform some other action based on the connection.

SOA systems are not new; in fact, Microsoft provides many SOA mechanisms in programs such as BizTalk or as separate add-ons to SQL Server 2000. In Microsoft SQL Server 2005, Microsoft includes the Service Broker directly integrated into the database engine.

If Microsoft already has SOA systems available in other products, why include it in SQL Server 2005? The reason has to do with the requirements that an SOA has. If you are going to allow systems to be disconnected, your SOA has to guarantee that the traffic between the systems is encapsulated into messages so that the sending program is identified, so it can receive the proper answer from the service. You also have to make sure that the messages are ordered properly. For instance, the program might send the third line of a purchase order before it sends the header. The service needs to be aware of the order and the encapsulation so that it can respond to the purchase order only when it is complete, just as in a database transaction.

To manage all this, SOA systems use a database. In other mechanisms, you need to manage not only the SOA system but a complete database as well. By including the SOA within the database engine, the data, metadata, tracking data, and mechanisms are all contained within the same architecture. There is just one system to learn, implement, and manage.

Another type of SOA is a "push"-oriented data system. I explained a little about this type of data movement in the last chapter when we examined SQL Server's Replication features. In that environment, data is either sent or picked up by another database. Although that is a useful feature, you will often need to send data to a medium other than a database, such as e-mail or an SMS device. In SQL Server 2005, Microsoft includes the Notification Services system that can do just that.

Many people in your technology department are normally involved in creating an SOA, simply because of the vast array of technical skills required to implement and maintain it. As the DBA, you will be asked to manage and maintain the system, working with developers, business analysts, and others.

Notification Services

Notification Services is one of those products that does what it says: It provides a means of notifying a user that an event has occurred in a database. The system uses several components to accomplish this goal, from the databases that store data about the events to the control files and databases that shepherd the process.

Let's begin by examining a few applications for Notification Services and how you can distribute them. The most obvious uses involve applications that require immediate feedback for a user. These include changes in prices, levels, inventory, and any other information that is time sensitive.

Other uses for Notification Services might not be as obvious. Marketing studies show that clients are most frustrated when they feel that a company is not responsive. You can code your applications to notify a customer as to which stage an order is in, such as when it is complete, when it leaves the building, or any other action that makes a change to a status code in a database. You can also set up a notification that an order was received and when it was placed.

For the system administrator, you can use Notification Services to help reactively manage your systems. Normally, a reactive mode of managing anything in IT is a bad idea, but using Notification Services you can build in the look-ahead logic so that a server watches objects (like backups or other maintenance) for you and can alert you when a threshold is reached.

So just what is Notification Services? It is an infrastructure within SQL Server 2005 that involves a service, instances, and applications, and a set of programs your developers create to interface with the system. In this chapter, I focus on the Notification Services instances and applications.

The easiest way to create the system is programmatically using *notification management objects* (NMOs). You can find several examples of this type of programming if you installed the sample and applications on your server.

You can also create and manage Notification Services using the native tools within SQL Server 2005 using XML files. We take a look at the structure of these files and examine the results of its implementation so that you can see how to manage and maintain it. In practice, most applications are coded using NMOs, but you can always create the instance and application XML files with an Export from the Object Explorer right-click menu in SQL Server Management Studio.

Notification Services Architecture

There is a lot of excellent information in Books Online regarding Notification Services, but it is not arranged in a holistic view of the system. There is a good reason for that. Notification Services uses several components within the SQL Server 2005 platform to accomplish its goals. In addition to all the capabilities that Microsoft delivers, your developers can extend every part of the system with custom programs and interfaces. To get a picture of how this all fits together, let's start at the back of the system and work forward to the user who receives the notification.

Because Notification Services has so many components, I am going to cover it three times. In the first, I give you a broad overview, then I explain it again with a little more detail, and then I cover the various basic components by explaining the *Instance Configuration File* (ICF) and the *Application Definition File* (ADF). Even with all of this information, you are only seeing a quick overview of this topic.

In Notification Services, several terms are used that already have meanings within SQL Server, such as *instance, publisher, subscriber,* and so forth. In this chapter, pay close attention to the definitions that Notification Services uses for these common terms.

At the back of the process is SQL Server 2005, which stores the data the users need to see and all of the metadata that Notification Services requires to operate. On top of SQL Server is the NSServer.exe program, which runs all of the instances, which make up the notifications for the users. To create these instances, you have two options: Your developers can create them with programs that use NMOs, or you can create them using an XML document called an ICF.

With an instance created, you create one or more Notification Services applications within it that listen for events, which are changes that your users are interested in knowing about. Events might involve database activities but also can track changes in Analysis Services, files on a file system, and other activities.

These events are matched with subscriptions, which are sent to the subscribers (users) over various protocols to devices such as cell phones or e-mail. Once again, your developers can create Notification Services applications using NMOs or you can create them with an XML document called an ADF detailing the parts of the application.

If you create the instances and applications using XML files, you import them using SQL Server Management Studio or the NSControl.exe command-line program.

That is the first overview, which leaves out quite a bit of detail. Let's examine the process a bit further. After the instance and applications are created and running, the instance uses a program called NSServer.exe that runs in the background watching for events using an event provider. Event providers watch for database changes, file system changes, and other events. It then applies matching rules to pair up the subscriptions and the events and then sends the data on to a provider. A provider is a piece of software that receives the data and formats it for delivery.

Providers are called by the application, and a generator matches up subscriptions to the events. Subscriptions are groups of data that a subscriber (the user) cares about. Notification Services populates the entire subscription, which is read by a distributor. Distributors use special-delivery services to talk not only to other databases but to file systems, e-mail and telephones, and any other kind of communication system your developers create.

The distributor formats the output using a content formatter and then sends it on to the delivery protocol provider, which can send data over SMTP, HTTP, and SMS. The subscriber receives the data in that format, over that protocol.

As the third explanation of the process, let's take an even deeper look at two of the primary components that you will work with: the ICF and the ADF. Using these XML files, you set up the instances and applications that enable your developers to create a complete Notification Services system. All the pieces I have been talking about begin to come together as you examine these files.

Instances and the Instance Configuration File

An instance of Notification Services is a collection of applications that run as a unit on a SQL Server. You can have one or more instances on a single SQL Server. You can create an instance programmatically using NMOs or by using SQL Server Management Studio and an XML file called an ICF. That is the approach I take here.

DBA 101: XML

If you are not familiar with XML, it is not difficult to learn. It is an ASCII file with special characters (called tags) in front of the text you want to mark off in some way. It is the same principle as HTML, but the tags in XML are not set by an independent body. You can make up any tag you want. All you have to remember is to "start" the tags (in this format: <thing>) and "end" or close the tag (using the format </thing>), add a bit of header information, and keep the tags nested properly (in which case, the document is said to be well formed).

In XML, the words that are tagged are called *elements*, and any other information about those words are called *attributes*. Elements are enclosed by tags, and attributes go inside the brackets. Here is an example of an element called `name` with the value of `Buck`, and an attribute of `first`:

```
<name type="first">Buck</name>
```

Another item that you will see in XML files is the comment. It looks like this:

```
<!— This is a comment —>
```

Comments are actually a form of directives. Directives are special characters (such as question marks) that tell the XML parser to act in a certain way (in this case, ignoring the text that follows).

An important difference from HTML to keep in mind is that XML is case sensitive. Buck is not the same as buck. When you are matching the tags, this often comes back to bite you.

There is a lot more to know about XML, of course, but these basics will help you read the files I show here.

The ICF contains the name of the instance, a database for its control, the name of the applications that can talk to the instance, and encryption and delivery information. You can also place a version stamp in the file, which I recommend for production applications. The format for the file is detailed in Books Online under the topic "Instance Configuration File Reference."

I display the minimal ICF file here so that we can talk a little about the parts that are required. In a moment, I show you the results of importing an ICF file into the database using SQL Server Management Studio. If you want a useful example of Notification Services, it is best to consult the examples from Microsoft, if you installed them. Setting up a complete system with an interface and databases and explaining it all would take up much more than a single chapter and would involve you in parts of the system that a DBA does not normally handle. I do recommend you use the examples, however, because they will shortcut all the development for you and allow you to concentrate on managing a Notification Services system.

Here is the minimal version that you need to fill out for a basic instance. Each section is commented out (`<!— comment —>`) so that we can more easily discuss it:

```
<?xml version="1.0" encoding="utf-8"?>
<NotificationServicesinstance
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://www.microsoft.com/MicrosoftNotificationServices
/ConfigurationFileSchema">

<!— Notification Services instance Name —>
<instanceName></instanceName>

<!— Database Engine instance —>
<SqlServerSystem></SqlServerSystem>

<!— Applications —>
<Applications>
 <Application>
  <ApplicationName></ApplicationName>
  <BaseDirectoryPath></BaseDirectoryPath>

<ApplicationDefinitionFilePath></ApplicationDefinitionFilePath
>
 </Application>
</Applications>

<!— Delivery Channels —>
<DeliveryChannels>
 <DeliveryChannel>
  <DeliveryChannelName></DeliveryChannelName>
  <ProtocolName></ProtocolName>
 </DeliveryChannel>
</DeliveryChannels>

</NotificationServicesinstance>
```

It looks like there is a lot going on here, but it is actually quite easy to follow. The first five lines of the file represent an XML header, which details what the file is and how it will be used. That does not change for any of your instances. The rest of the sections control the creation of the instance, similar to the old .INI files of earlier Windows programs.

<!— Notification Services instance Name —>

The first section, specified in the `<instanceName>` element, sets the instance name for the system as well as the service it creates to run this instance. From here on out, the applications and other parts of this notification system's grouping will refer to this name. I usually keep these short and as descriptive as possible. The instance Name becomes the name of the service that runs in the operating system, prefaced by NS$. For example, the following entry creates a service called NS$AWInventory:

```
<instanceName>AWInventory</instanceName>
```

<!— Database Engine instance —>

The `<SqlServerSystem>` element sets the name of the SQL Server instance (no relation to the Notification Services instance set up with this file) where the Notification Services instance will run. It is the name of the server where you want the service to run.

<!— Applications —>

Each instance holds one or more applications that can talk to it. The `<Applications>` element is a parent element that contains other information that relates to the applications that will run under this instance. In effect, it is pointing to the ADF that you will also need to create prior to importing the instance using the ICF.

If you try to import an ICF before the ADF is complete, you will get an error.

This section includes the name of the application and its location and name on the hard drive. You can store these files in a shared folder, as long as the SQL Server and service accounts have access to it.

DBA 101: XML Hierarchies

If you read down the file until you see the "opening" tag of `<Applications>` and the "closing" tag `</Applications>`, you will see that there are other items between them. In XML, this is how you nest items to make them "children" of other elements.

In this example, I create only one application for the instance, but you can have as many as the resources for your system allows. It is best to plan out the entire system before you begin this process so that the instances are broken out logically to contain the applications that make the most sense to group together.

The first child element in the `<Applications>` tag is `<Application>`, which is another parent tag. You repeat this element for each application that the instance will host. Within that parent element is the `<ApplicationName>` element, which sets the name of that particular application.

Each application needs a directory to store its ADF. This file, which I create in a moment, holds the same kind of information created for the instance but sets up the application.

The next element, `<ApplicationDefinitionFilePath>`, sets the name of the ADF file. If you store only the filename here, the file needs to be stored directly in the BaseDirectoryPath location.

The reason you have two elements available is because in actual production, you will normally have several applications that run on a single instance. You want to keep these applications separate so that various developers can have the access they need to work with only the applications they are assigned to. Keeping the applications in separate directories allows you a greater level of security.

<!— Delivery Channels —>

The instance controls how the system communicates with the outside world. The first parent element, called `<DeliveryChannels>`, begins the section describing all the ways that this instance can communicate.

You open the delivery method with the `<DeliveryChannel>` element, and then type in the `<DeliveryChannelName>` element to have the type you want, from e-mail to file. Within that instruction, you need to specify the protocol the channel will use. You do that with the value of the `<ProtocolName>` element.

There are lots of ways to send information, each with its own protocol settings. So that we can maintain the overview process here, I will direct you to Books Online using the topic search I mentioned earlier to learn about the various methods you have available.

With all of the basic elements defined, we need to create the ADF.

Applications and the Application Definition File

One or more applications are contained within a single instance of Notification Services. When you create an application, you will also create or specify a database that stores the events, the subscriptions, notification data, and other application metadata about the application. If you do not have a database, you can have the XML file (or the NMO program) create one for you.

Once again, we will look at a basic XML file and examine the elements it contains:

```
<?xml version="1.0" encoding="utf-8" ?>
<Application xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://www.microsoft.com/MicrosoftNotificationServices
/ApplicationDefinitionFileSchema">

 <!— Version —>
 <!— History —>
 <!— Database Definition —>
 <!— Event Classes —>

 <!— Subscription Classes —>
 <SubscriptionClasses></SubscriptionClasses>

 <!— Notification Classes —>
 <NotificationClasses></NotificationClasses>

 <!— Event Providers —>

 <!— Generator Settings —>
 <Generator>
  <SystemName>%SystemName%</SystemName>
 </Generator>

 <!— Distributor Settings —>
 <Distributors>
  <Distributor>
   <SystemName>%SystemName%</SystemName>
  </Distributor>
 </Distributors>
 <!— Application Execution Settings —>
 <!— Important: At minimum, you should define
  a vacuuming schedule and turn off some or all
  distributor logging. —>

</Application>
```

This file has quite a few subelements, so once again I recommend that you look up the topic in Books Online using the search topic “Application Definition File Reference.” There you will find all of these elements hyperlinked, in alphabetic order. Using this minimal template and those search results, you can quickly build the file you want.

In this minimal file, the first four lines set the header for the XML file and tell SQL Server 2005 how to handle it.

<!— Version —>

The first section here is the `<Version>` element. Within that are various child elements that set a version number and build. The `<Version>` element is optional, but I normally include it so that I can track where I am in the build process. It is a good habit to get into.

<!— History —>

The `History` section is similar to the `Version` section, in that it provides tracking data for your system. You can include data here to track what changes you have made to the ADF.

<!— Database Definition —>

In the `<Database Definition>` elements, you set the name, schema, and physical structures for the application. If you do not provide one, the system creates a database using the defaults for the server. In a moment, I create a Notification Services application and show you the types of things that end up in this database.

<!— Event Classes —>

It is in the `Event Classes` section that things really become interesting. In this section, beginning with the `<EventClasses>` parent element, you set up one or more `<EventClass>` elements that define what events the system responds to.

<!— Subscription Classes —>

In the `Subscription  Classes` section, using the parent tag of `<SubscriptionClasses>`, you set up one or more subscriptions for the application using `<SubscriptionClass>` elements. Here you describe the name, the schema for the subscription, any indexes you want on the tables, and event and schedule rules. All of these elements set up how often the events are captured and what can be subscribed to.

<!— Notification Classes —>

Using the `<NotificationClasses>` parent tag in this section, you set up one or more notifications for the application using `<NotificationClass>` elements. These elements define where your application stores the notifications, what filter is used to format them, and what protocols the transport uses. You can also set whether the notifications are sent one at a time or if they use a "digest," which groups them all together to be sent at the end of a specified period.

All of these settings are similar to what you will see in a newsgroup; and if you think about it that way, you will have the concept.

<!— Event Providers —>

As I explained earlier, an event is a change in data that you care about. For instance, a stock price change or a project change would be an event. Microsoft delivers several of these with SQL Server 2005, and you can also write your own. Here are a few of the event providers that your system can use.

Analysis Services	MDX statements
File Watcher	XML output, requires a schema document
SQL Server	Runs a SQL statement to detect changes; Can be non-Microsoft because it uses linked servers
Custom	Uses IEventProvider and IScheduledEventProvider

The most prevalent event that I have seen used is the SQL Server event. This allows you to watch a table for changes and deliver the information to a user via a subscription.

Each of the event types has specific elements they require to be able to process them.

<!— Generator Settings —>

The generator is the component within Notification Services that matches changes in data (events) to those who should learn about the changes (subscribers). It runs in the background on your server using the NSServer.exe program.

In this section, you use elements to define the name of the system that will act as the generator and set the amount of threads it will use.

<!— Distributor Settings —>

The distributor is the component within Notification Services that formats and sends the data. You can store the distributor on one or more servers to help balance the load on your system. You use elements in this section to set the name of the distributor, the threads it uses, and an optional duration setting.

<!— Application Execution Settings —>

In this section, you set up all of the information around the processing of your events, such as limiting the amount of time that is spent on a particular task as well as the order in which applications are processed. An important element to set is the `<Vacuum>` value, which sets how often the cleanup process runs.

With all of those components in mind, let's take a look at how all of it fits together within the framework of a complete application. I have a single application that I plan to set up to watch an inventory level. I have created the ICF and the ADF and placed them in a directory on my local test system. I will import those into my instance of SQL Server 2005.

Installing the Instance and Creating the Application

In Figure 7-1, I have opened SQL Server Management Studio and right-clicked the Notification Services object in the Object Explorer.

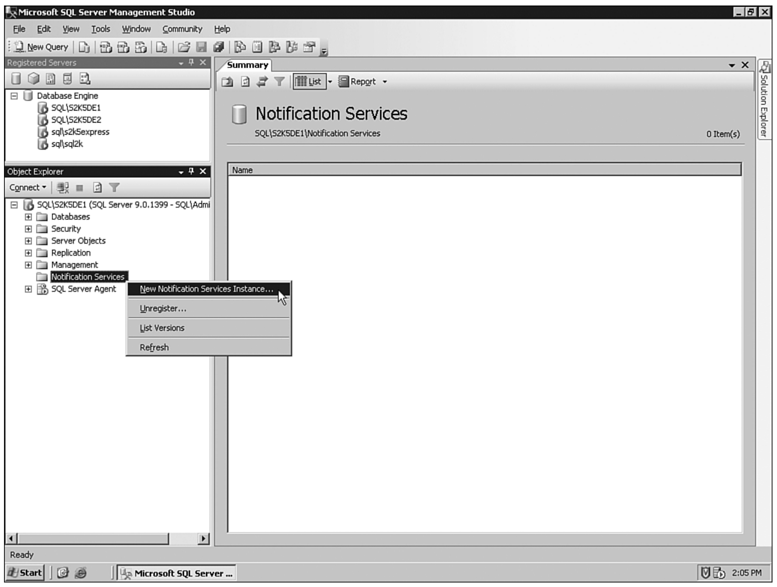
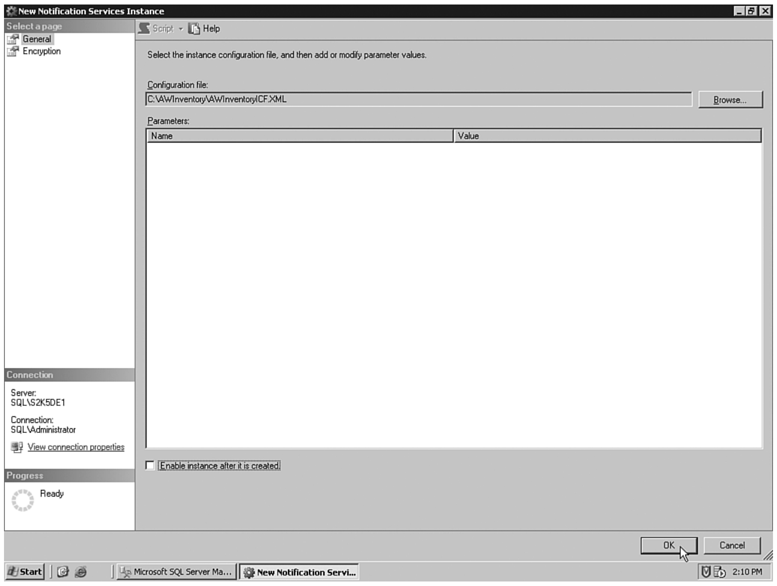


Figure 7-1

Within that panel, I specify the location of the ICF, as shown in Figure 7-2.


New Notification Services Instance
Select a page
General
Encryption
Script
Help
Browse...
Connection
View connection properties
Progress
OK
Cancel
Start
Microsoft SQL Server Ma...
New Notification Servi...
2:10 PM

Select the instance configuration file, and then add or modify parameter values.

Configuration file: C:\AWInventory\AWInventoryICF.XML

Parameters:

Name	Value

Enable instance after it is created

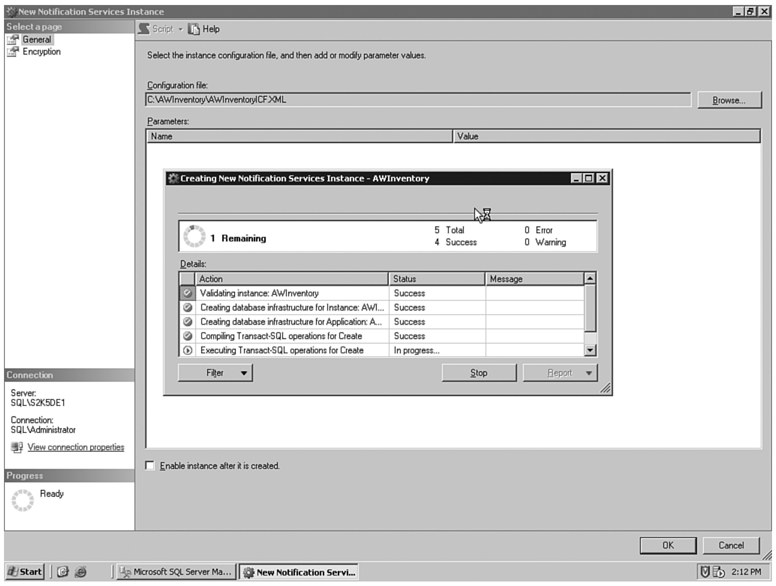
Server: SQL\S2K5DE1

Connection: SQL\Administrator

Ready

Figure 7-2

With that set, I click the **OK** button to process the file. My system shows the screen in Figure 7-3 while it processes.


New Notification Services Instance
Select a page
General
Encryption
Script
Help
Browse...
Connection
View connection properties
Progress
Filter
Stop
Report
OK
Cancel
Start
Microsoft SQL Server Ma...
New Notification Servi...
2:12 PM

Select the instance configuration file, and then add or modify parameter values.

Configuration file: C:\AWInventory\AWInventoryICF.XML

Parameters:

Name	Value

Creating New Notification Services Instance - AWInventory

1 Remaining

5 Total
4 Success
0 Error
0 Warning

Details:

Action	Status	Message
Validating instance: AWInventory	Success	
Creating database infrastructure for Instance: AWI...	Success	
Creating database infrastructure for Application: A...	Success	
Compiling Transact-SQL operations for Create	Success	
Executing Transact-SQL operations for Create	In progress...	

Enable instance after it is created.

Server: SQL\S2K5DE1

Connection: SQL\Administrator

Ready

Figure 7-3

Now I will enable the system by right-clicking it in the Object Explorer again. This process sets the distributors, generators, and subscriptions for the system. You can see that in Figure 7-4.

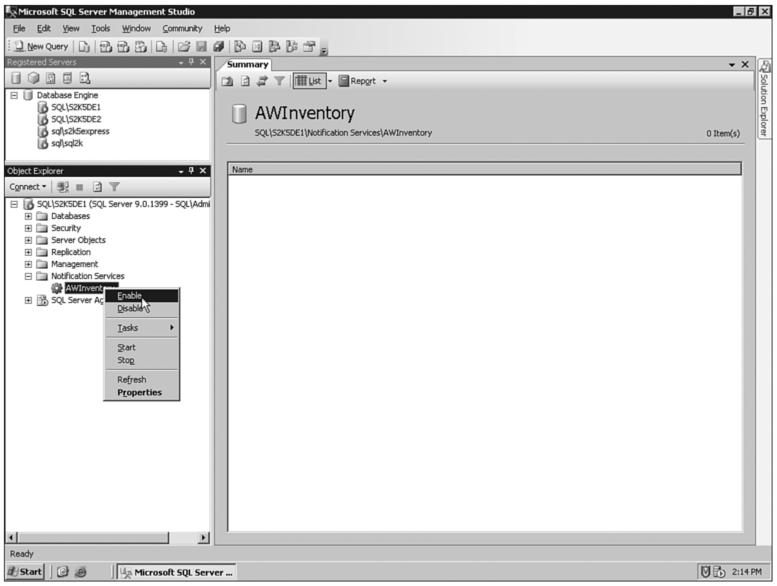


Figure 7-4

When that process completes, I right-click the service and select **Tasks,...** and then **Register** from the menu that appears, as shown in Figure 7-5.

This brings up a panel where I create the service, set the authorization for the service, and set how the service will access the database. You can see that in Figure 7-6.

I receive feedback that shows that the system is creating registry entries, creating the service, and adding new performance counters. With all that complete, I again right-click the name of the service and select **Start** from the menu that appears. After confirming that I want the system to start, the databases I specified in the ICF (AWInventoryNSMain) and the ADF (NSMetaData) are created, and the service starts.

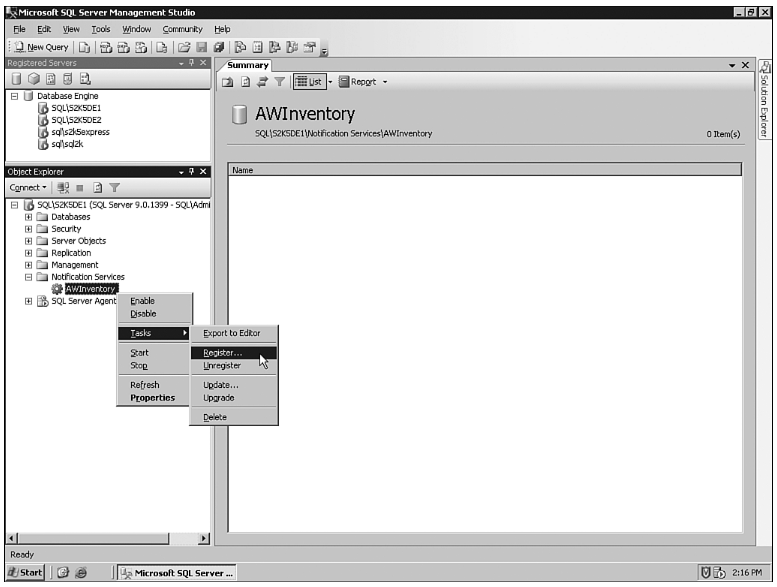
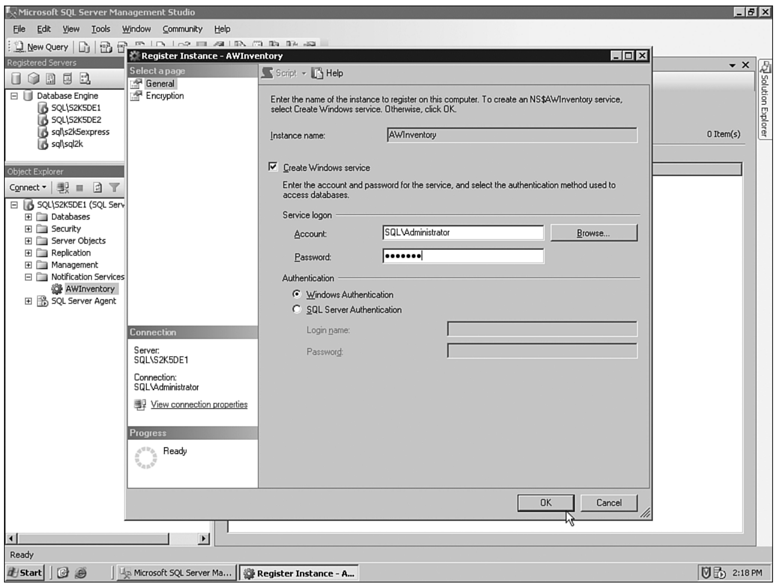


Figure 7-5


Microsoft SQL Server Management Studio
Register Instance - AWInventory
Select a page
General
Encryption
Script
Help
Connection
View connection properties
Progress
OK
Cancel
Start
Microsoft SQL Server Ma...
Register Instance - A...
2:18 PM

Enter the name of the instance to register on this computer. To create an NS$AWInventory service, select Create Windows service. Otherwise, click OK.

Instance name: AWInventory

☑ Create Windows service

Enter the account and password for the service, and select the authentication method used to access databases.

Service logon

Account: SQL\Administrator

Password: •••••••

Authentication

(●) Windows Authentication

( ) SQL Server Authentication

Login name:

Password:

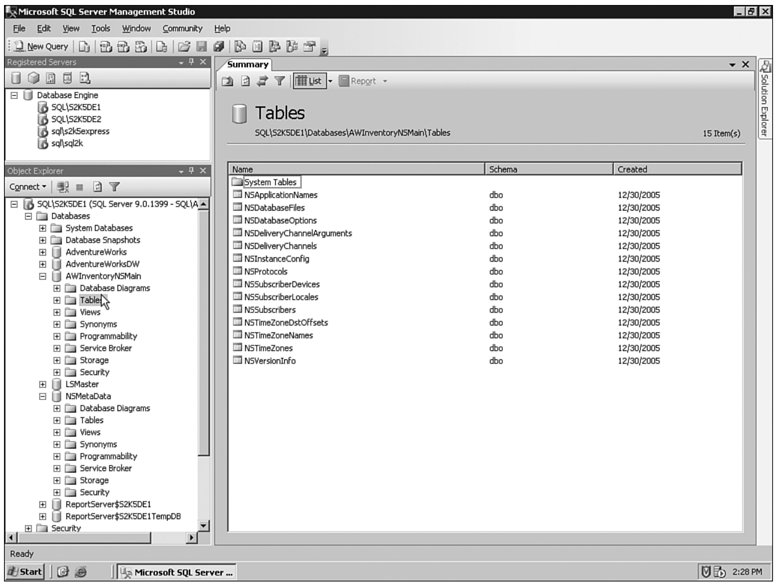
Server:
SQL\S2K5DE1

Connection:
SQL\Administrator

Ready

Figure 7-6

Figure 7-7 shows a list of the tables within the instance database called AWInventoryNSMain.



Tables

SQL\S2K5DE1\Databases\AWInventoryNSMain\Tables 15 Item(s)

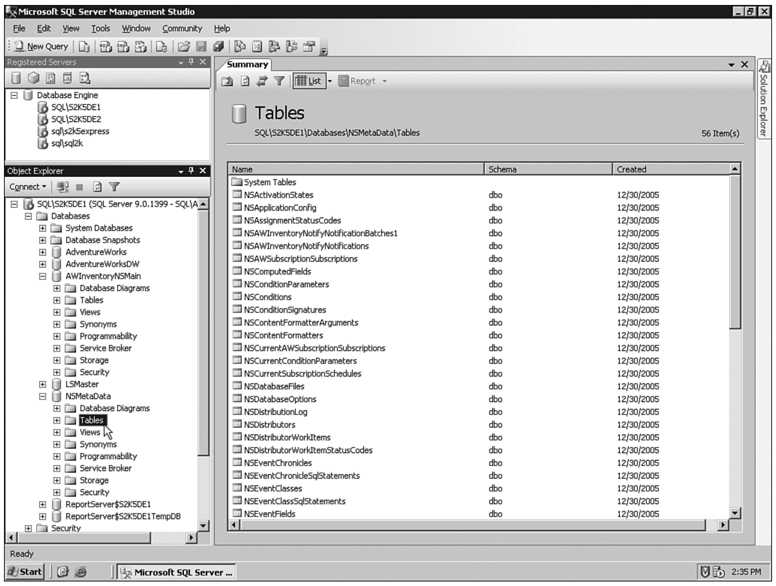
Name	Schema	Created
System Tables		
NSApplicationNames	dbo	12/30/2005
NSDatabaseFiles	dbo	12/30/2005
NSDatabaseOptions	dbo	12/30/2005
NSDeliveryChannelArguments	dbo	12/30/2005
NSDeliveryChannels	dbo	12/30/2005
NSInstanceConfig	dbo	12/30/2005
NSProtocols	dbo	12/30/2005
NSSubscriberDevices	dbo	12/30/2005
NSSubscriberLocales	dbo	12/30/2005
NSSubscribers	dbo	12/30/2005
NSTimeZoneDstOffsets	dbo	12/30/2005
NSTimeZoneNames	dbo	12/30/2005
NSTimeZones	dbo	12/30/2005
NSVersionInfo	dbo	12/30/2005

Figure 7-7

This database contains the elements I explained earlier within the ICF document. There are a few other metadata tables here, especially involving the time settings such as time zone and locales.

The tables in the application database called NSMetaData also contain the elements I explained in the ADF and other metadata. You can see all of that in Figure 7-8.

The service reads these databases and accesses the information, ready for the client programs to subscribe, similar to what I explained for replication.



Tables

SQL\S2K5DE1\Databases\NSMetaData\Tables   56 Item(s)

Name	Schema	Created
System Tables		
NSActivationStates	dbo	12/30/2005
NSApplicationConfig	dbo	12/30/2005
NSAssignmentStatusCodes	dbo	12/30/2005
NSAWInventoryNotifyNotificationBatches1	dbo	12/30/2005
NSAWInventoryNotifyNotifications	dbo	12/30/2005
NSAWSubscriptionSubscriptions	dbo	12/30/2005
NSComputedFields	dbo	12/30/2005
NSConditionParameters	dbo	12/30/2005
NSConditions	dbo	12/30/2005
NSConditionSignatures	dbo	12/30/2005
NSContentFormatterArguments	dbo	12/30/2005
NSContentFormatters	dbo	12/30/2005
NSCurrentAWSubscriptionSubscriptions	dbo	12/30/2005
NSCurrentConditionParameters	dbo	12/30/2005
NSCurrentSubscriptionSchedules	dbo	12/30/2005
NSDatabaseFiles	dbo	12/30/2005
NSDatabaseOptions	dbo	12/30/2005
NSDistributionLog	dbo	12/30/2005
NSDistributors	dbo	12/30/2005
NSDistributorWorkItems	dbo	12/30/2005
NSDistributorWorkItemStatusCodes	dbo	12/30/2005
NSEventChronicles	dbo	12/30/2005
NSEventChronicleSqlStatements	dbo	12/30/2005
NSEventClasses	dbo	12/30/2005
NSEventClassSqlStatements	dbo	12/30/2005
NSEventFields	dbo	12/30/2005

Figure 7-8

Security

There are two parts to the security of your Notification Services system. The first is to secure the locations where the ICFs and ADFs are located.

The reason you want these files secure is that those who can access them can change them to reflect more data in the `Events` section than you might want. The small files I described earlier are not representative of what you create in a production environment. The production files are much larger, so you might miss the addition of a new event or a change to a current one.

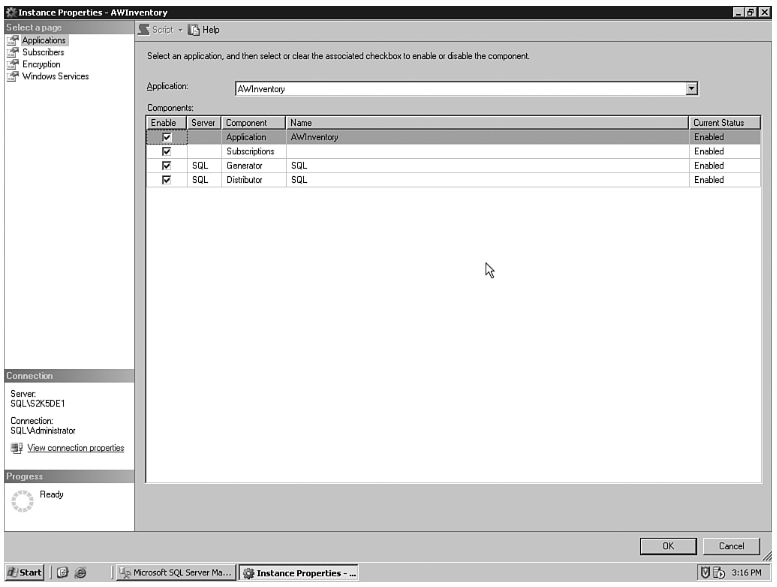
After you have your instance running, you can alter it later by re-importing the ICF and ADF. That is where the danger comes in on leaving these areas unsecured. If your developers are using NMOs to code the application rather than the control files, this is less of an issue.

The second part of securing the system is against the subscription data. To secure this data, you simply use Windows accounts or SQL Server accounts.

Monitoring and Performance Tuning

Notification Services can be used on servers that scale up, and they can also be used on a cluster. Another method for tuning the application is to spread it out onto multiple servers, including the generators and other parts of the application. That is why it is important to plan out the system ahead of time.

You can monitor a Notification Services system in SQL Server Management Studio by right-clicking the instance you are interested in and selecting the **Properties** item. Once inside, click the **Applications** item in the left pane, and in the Components for that application, check the **Status** column, as shown in Figure 7-9.



Select an application, and then select or clear the associated checkbox to enable or disable the component.

Application: AWInventory

Components:

Enable	Server	Component	Name	Current Status
✓		Application	AWInventory	Enabled
✓		Subscriptions		Enabled
✓	SQL	Generator	SQL	Enabled
✓	SQL	Distributor	SQL	Enabled

Server:
SQL\S2K5DE1

Connection:
SQL\Administrator

Figure 7-9

You can also check the Subscriber status from this resource box. Another monitoring tool is a suite of Performance Monitor objects and counters that you can access from the System Monitor in Windows, as you can see in Figure 7-10.

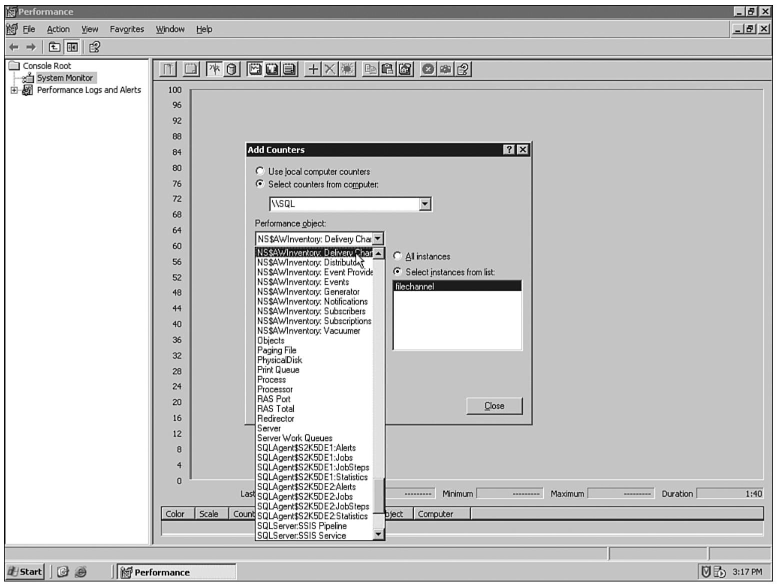

Performance
File Action View Favorites Window Help
Console Root
System Monitor
Performance Logs and Alerts
Add Counters
Use local computer counters
Select counters from computer:
\\SQL
Performance object:
NS$AWInventory: Delivery Char
NS$AWInventory: Delivery Char
NS$AWInventory: Distributor
NS$AWInventory: Event Provide
NS$AWInventory: Events
NS$AWInventory: Generator
NS$AWInventory: Notifications
NS$AWInventory: Subscribers
NS$AWInventory: Subscriptions
NS$AWInventory: Vacuumer
Objects
Paging File
PhysicalDisk
Print Queue
Process
Processor
RAS Port
RAS Total
Redirector
Server
Server Work Queues
SQLAgent$S2K5DE1:Alerts
SQLAgent$S2K5DE1:Jobs
SQLAgent$S2K5DE1:JobSteps
SQLAgent$S2K5DE1:Statistics
SQLAgent$S2K5DE2:Alerts
SQLAgent$S2K5DE2:Jobs
SQLAgent$S2K5DE2:JobSteps
SQLAgent$S2K5DE2:Statistics
SQLServer:SSIS Pipeline
SQLServer:SSIS Service
All instances
Select instances from list:
filechannel
Close
Minimum
Maximum
Duration
1:40
Color
Scale
Computer
Start
Performance
3:17 PM

Figure 7-10

You can monitor all parts of the system based on what you are interested in examining, from the subscriber subsystem to the collection methods.

Service Broker

In replication, data is exported from one server to another, but it always stays in the format of a database. In Notification Services, data is delivered to subscribers, using phones, e-mail, or other formats. Although you can send data between systems using these technologies, they are often used in a unidirectional method, sending data from a server to another system or user. Earlier I explained that there are distinct advantages in using a disconnected system such as replication and Notification Services, which are in effect store-and-forward mechanisms. The SQL Server 2005 Service Broker extends this concept to the entire application.

With the Service Broker, you can build an entire infrastructure that is "loosely coupled." That means that the system does not receive immediate feedback from each application that the tasks are complete, just that the task has been received.

Microsoft uses the analogy of the Post Office to explain the Service Broker. For the DBA learning about an SOA, this is a great way to think about the process. I use that analogy to describe what happens at a physical Post Office and then extend it to show the parts of the Service Broker. In the "Take Away" section, I show you a simple Service Broker program example that you can extend to do useful work on your production servers.

Using the Post Office to Understand the Service Broker

Let's assume you want to place an order for a new computer through the mail. You write a letter, place it in an envelope with instructions for the final location (the address), and drop it into any receptacle that the Post Office recognizes as a pick-up point. The message is picked up and brought to the local Post Office, sorted, and sent to the next Post Office in the chain all the way down the line until it reaches the Post Office closest to the recipient. That Post Office delivers the message to the recipient, who opens the envelope and reads your instructions. They initiate several actions in multiple departments, and you receive a confirmation message and then a follow-on package from the company, which completes the transaction.

Most of us use this kind of system every day, without considering the advantages it provides. For one thing, it provides a distribution of function. You write the message and receive the goods, without any knowledge or concern about the route the message will take. The Post Office carrier picks up the envelope without knowledge of its contents or final destination. The Post Office sorts the message and routes it without knowledge of the message. The recipient reads the message without concern for its previous route or future destination. The process is then reversed. As long as everyone does his or her part, there is no need to have an active governing system that needs to know about every component, from the routes to the message content. In other words, one person does not have to follow the letter from beginning to end.

Another advantage is that a single message might cause a flurry of activity at the receiving end. If you order something through the mail, that triggers several departments at a company, such as purchasing, shipping, marketing, and so on. All with just one small order that you sent: you do not have to send an individual message to each department to have them start working on your order. In fact, each department is working on several orders at one time.

The Service Broker works the same way. Each component has a specific function, and items can be "dropped off" at each point with the guarantee that the component will perform its function having no knowledge or control of any of the other parts. Developers can write complex systems by coding their individual parts with the Service Broker and letting each work on multiple activities at once.

Let's take a look at each of the parts of the Service Broker as it applies to the Post Office analogy. Along the way I explain how the two relate and some of the information you will need to implement and maintain your own system.

Just as in the case of Notification Services, the Service Broker introduces entirely different uses for the same terminology used in replication and elsewhere in IT parlance. Watch this section for new definitions of words such as *type* and *service.*

The Postal System and the Service Broker

As a conceptual "umbrella" to the whole process is the idea of a postal system. Someone (in the United States, Benjamin Franklin) originally institutes and lays out the rules for getting messages from sender to receiver. In the case of the Service Broker, the SQL Server 2005 database engine works in terms of conversations, which are similar to a single-direction motion in the postal system. For instance, the postal carrier picks up your mail from your mailbox and carries it to the local Post Office. That is a conversation in Service Broker terminology.

Happily, the message in the Service Broker is the same as the message in the postal system, referring both to the enclosure (envelope) and the payload inside it. The only difficulty is making sure you define whether you are talking about the whole thing or just the payload inside. I try to make that clear as we move along.

In a simple example where the recipient is in the same town that you are, the message leaves the local Post Office and moves to the recipient's mailbox. That is another conversation. Taken together, the two conversations along with the message form what Microsoft calls a conversation group.

In T-SQL, you can examine the conversation group with the command `GET CONVERSATION GROUP`. You will often use this command to get the status of a particular part of the system.

When your letter gets to the Post Office, it is stored there until it moves on. In the Service Broker, this is handled by storage called the queue.

Once inside the Post Office, the letter moves from place to place to be properly sorted. Those internal processes that keep everything in the right place in the Service Broker are called dialogs. The Post Office stamps a "cancellation" across the message to show when it was received. If you are in the United States, April 15 (the day our tax forms are due) is one date you want to make sure you see on your envelope. Dialogs in the Service Broker do the same thing. They stamp each message with an order number so that they are guaranteed to be correct. This is important in transaction-type applications because one line in an order might change the one prior to it.

The Post Office has employed several people to do all the moving, sorting, and delivering. In the Service Broker, the employees are called services, and activations make them work. We discuss those further in a moment.

The Post Office actually handles more than just one type of letter. You might send a postcard, or letters of various sizes. You might send a package, too, and even specify special handling instructions. In the case of the Service Broker, this is referred to as a message type. Because applications can send everything from text to music files, each is treated a little differently. Most of the issues I have run into using the Service Broker involve a mismatch of types.

The letter is finally read by the recipient. In the Service Broker, the system sends an EndDialog message, which ends the conversation.

That is the broad overview of the physical layout; now let's drill in a little to the traffic flow from the sender to the receiver.

The Mail—Message Types and Contents

The postal process starts with the mail message itself. Whenever you look at a piece of mail, it has at least two components: the envelope and the contents of the message inside. The envelope has a required size, it must have the addresses for both the sender and receiver in certain places and be readable, and it must have the right amount of postage. You form a contract between yourself and the Post Office that you will follow the rules and they will deliver the letter.

Envelope—Message Types

To send a letter, you have to subscribe to the rules they lay out for envelopes and postage. In the Service Broker, these are analogous to message types. A Service Broker message type has two basic values: `XML` and `NONE`.

In the `XML` type, SQL Server attempts to check the validity of the document prior to processing it. Anything that can use text can use the `XML` type. You can also specify that you want to process it with a particular XML schema document to verify the contents.

If you are sending anything other than text, such as a binary file, you can use the message type of `NONE`. That type instructs SQL Server not to check the validity of the contents.

The message type is specified when you send the message with the `SEND` and `RECEIVE` T-SQL commands, and you can create new ones with the `CREATE MESSAGE TYPE` command in T-SQL. I show you this process in the "Take Away" section at the end of the chapter.

Contents

You can send entire books in a single pass through the postal service, but that is not always a good idea for the Service Broker. You are working with a computer system that uses your regular network lines, so it is usually best to break down the messages into smaller pieces.

You create the message in your originating program, called the initiator. Most of the time the documents are passed around using XML files.

The Sender—Initiators, Contracts

When you send mail, you are responsible for writing the message on some type of paper, for recording the address properly (both yours and the recipient's) on the envelope, and using proper postage. The postal service guarantees that if you follow those rules, they will deliver the letter. That is a contract, and in the Service Broker you have exactly the same type of arrangement.

The user or a program, called the initiator, enters the information they want to send or receive. The program runs a stored procedure, which creates a message and begins a conversation with the proper Service Broker service program, which I explain in a moment.

The Postal Carrier—Physical Transport, Protocols, and Remote Service Bindings

Although it is invisible much of the time, you are still using some method to move this data from client to server. In the Post Office, postal carriers drive vans or walk a route to pick up and deliver mail. In the Service Broker, physical transports (such as the network infrastructure) and protocols (such as TCP/IP) handle the movement from one location to another.

That might seem obvious, but it is important to consider it in the overall picture. If the Post Office ignored the fact that they use vehicles in their work, they might not perform the routine maintenance they need or plan for redundant vehicles. It is the same in the Service Broker. You need to be aware that there are physical parts to your system and plan for maintenance and backups in case they fail.

You can secure the entire communication process with certificates if you want. This provides a high level of encryption so that the message is difficult to break.

The Post Office—Endpoints, Queues, and Activations

I have been explaining the Service Broker using Microsoft's postal service analogy, but I have moved in the direction of the client to the receiver. I have done that on purpose so that you can see how it all fits together, but in the postal service as well as the Service Broker, that is not how things really get built. Before a user ever sits down to write a letter, the Post Office has to be built first. That is the same way that it is in the Service Broker; you normally build the middle first and work out from there.

Each Post Office has a set of doors that only the postal carriers are allowed to use. In the Service Broker, this is called an endpoint. Endpoints are network connections directly into a service. The conversation groups I described earlier send messages between services using endpoints. Much of the DBA's time is spent creating and tracking endpoints.

In the Service Broker, the queue is similar to a single storage location at the Post Office. A queue is the data repository for all the messages. Unlike the Post Office, one message can be sent from a queue to multiple locations. I guess it is more like the Post Office than I care for; many people do get lots of the same junk mail.

In the Post Office, an employee is assigned to work on certain kinds of mail, in certain areas of the Post Office. In the Service Broker, each queue is associated with a stored procedure or a managed program in the CLR layer, called a service program. When a message is placed in the queue, the service program is activated and begins to process the message. Most of the work the developer does is within these stored procedures. In effect, these stored procedures are the workers in the Post Office.

This brings up another advantage in having Service Broker running directly within the database. This architecture removes the need for multiple services to be installed on the operating system. Because the queue is associated with a stored procedure, the stored procedure "wakes up" each time there is work to do, but not before. Nothing has to run in the background, constantly watching for work to do.

Another important part of this analogy is that whereas a postal worker might send the message to the final recipient, the worker might also send it to another postal worker for some additional processing. In the Service Broker, the stored procedures might do the same thing. Depending on what the stored procedure code is, the message might be processed and a final message sent or sent to yet another procedure for more processing.

The Recipient—Calling Programs

When the processing is complete, the conversation is terminated. Your original calling program can either collect the status from the Service Broker or move on without checking.

Security

Unlike other database applications, the Service Broker is based on passing messages between systems. Because of that, you are not as concerned with individual user accounts. It is a bit more like the application security explained in Chapter 4, "Security," because a single account is used to access the system on the user's behalf.

In the Service Broker, you focus your security on two areas: dialog and transport. Dialogs have to do with the messages, and transport has to do with the network. Implementing both makes the system secure.

Dialog security is set up by creating a user in the application database and then a certificate for that user. From there, you can use a Service Binding mechanism to associate the user and the Service Broker. The application uses that user to send and receive data.

For transport security, you set up a server login and a certificate in the master database and then send the certificate to the calling programs and the DBA for the application. The system uses this login to process the transactions.

Monitoring and Performance Tuning

To manage and monitor the system, you get four new dynamic management views and three Performance Monitor objects and counters.

The dynamic management views are as follows:

`sys.dm_broker_activated_tasks`	Shows all of the stored procedure activations
`sys.dm_broker_connections`	Shows all of the Service Broker network connections
`sys.dm_broker_forwarded_messages`	Shows the messages that are in the process of forwarding
`sys.dm_broker_queue_monitors`	Shows the queue monitor that manages activation for a queue

To display the Performance Monitor values, add the following objects and counters to your trace.

SQL Server, Broker Activation Object	Shows stored procedure activations
SQL Server, Broker Statistics Object	Shows general Service Broker information
SQL Server, Broker / DBM Transport Object	Shows the interaction between the Service Broker and database mirroring

The two most important tuning strategies for the Service Broker are planning the layout and optimizing the stored procedures that act as the service programs in the application. Proper planning for the infrastructure is normally why you have implemented the Service Broker to begin with. When you analyze a solution, always ask whether an asynchronous architecture is possible or preferable, and then implement the layout accordingly.

Like most applications, the biggest bang for the effort is in tuning the T-SQL code that runs within the stored procedures. One strategy that you might want to alert the developers about is the use of the `WAITFOR` statement. This instructs the stored procedure to wait for an event to occur prior to looping through the code. Adding this statement is important in a disconnected architecture because the message might not be complete when the service program is activated.

Take Away

Both Notification Services and the Service Broker are more about the programming constructs than administration. Other than the security implications and a few more Performance Monitor objects and counters, there is not a lot for the DBA to do in the system from day to day.

I thought it might be useful to work through a simple example of a Service Broker application because it is the one that is the most accessible from standard T-SQL statements. Creating your own application helps you to understand the process, and you can even use this simple program as a starting point for a more advanced application.

I will keep the comments down to what I think you should know along the way, because seeing the entire application process laid out is useful to understanding the process quickly. You should be familiar with the terms I explained in the chapter to follow along. If you are interested in extending this system, create the application on your test system just as I have here. Then review the T-SQL statements that make up the process to expand what the system can do.

Service Broker Example

In this section, I create a simple example of adding an employee to a database. The requirement is for a junior human resources worker to be able to add an entry into the company's employee system. Because her manager wants to review the entry prior to the employee receiving a permanent ID number, we have decided to create an application that takes her entry and places it in a "holding" table until it is reviewed.

On my test system, I created a database called ServiceBrokerExample that has one table called Employee. That table has one column called EmployeeInfo, with an xml column setting, because that is what my application expects. Here is the code for all that in case you want to try it on your system:

```
— Create the database
CREATE DATABASE ServiceBrokerExample
GO

— Create the table
USE [ServiceBrokerExample]
GO
CREATE TABLE [dbo].[Employee](
      [EmployeeInfo] [xml] NULL
) ON [PRIMARY]
GO
```

With that all set, I begin the process by creating a message type for the system. I instruct the type to check to make sure my data is in proper XML format, but in production you will also often reference a full XML schema document:

```
— Create the Message Type
CREATE MESSAGE TYPE
 [ServiceBroker/Example/Employee/AddEmployee]
 VALIDATION = WELL_FORMED_XML
GO
```

I then create a contract for the system, for each party to use. You can see that it uses the AddEmployee message type that I just created:

```
— Create the Contract
CREATE CONTRACT
 [ServiceBroker/Example/Employee/AddEmployeeContract]
 ([ServiceBroker/Example/Employee/AddEmployee]
 SENT BY INITIATOR
 )
GO
```

Here are my “postal workers” that read the data from the queue and insert it into the database. I am using an XML conversion function to move the data along and place it into the database. There are two stored procedures here: one to do the inserts, and the other to check and empty the queue. Do not let the complexity stop you; read through each section line by line to see what’s happening here:

```
— Create the Insert SP
CREATE PROCEDURE [dbo].[AddEmployee]
      @MB xml
AS
INSERT INTO Employee(EmployeeInfo)
VALUES (@MB)
GO
```

And now the stored procedure that reads the queue:

```
— Create the Update SP
CREATE PROCEDURE [dbo].[ProcessEmployee]
AS
BEGIN
 BEGIN TRAN
  DECLARE @CH uniqueidentifier
  DECLARE @MB varbinary(max)
 —dequeues the message
  WAITFOR
  (
  RECEIVE TOP(1) @CH = conversation_handle, @MB =
message_body
  FROM EmployeeQueue
  ),
  TIMEOUT 1500
  —process the message

  EXECUTE ProcessEmployee @MB
  END CONVERSATION @CH
 COMMIT TRAN
 END
```

With the workers in place, I need to set up the “Post Office”—the queue that will store all the data. When I create the queue, I assign the service program (in this case, my stored procedure) that is assigned to process it:

```
— Create the Queue
CREATE QUEUE EmployeeQueue
 WITH STATUS = ON,
 ACTIVATION
 (
 PROCEDURE_NAME = ProcessEmployee,
 MAX_QUEUE_READERS = 5,
 EXECUTE AS SELF
 )
GO
```

I am almost there. Now I create the service that responds to the requests from the conversations; I will also tie that to the contract I created earlier:

```
— Create the Service
CREATE SERVICE AddEmployeeService
 ON QUEUE [EmployeeQueue]
 ([ServiceBroker/Example/Employee/AddEmployeeContract])
GO
```

The system is now ready, and I can examine all the objects using the SQL Server Management Studio. With the server ready for Service Broker conversations, I can set up a full sample event. I am only sending a snippet of the code I would really use as the XML document, but this snippet makes the code easier to read:

```
— Begin the dialog
DECLARE @CH uniqueidentifier
DECLARE @EmployeeName XML
SET @Employeename = '<name>Buck</name>'

BEGIN DIALOG CONVERSATION @CH
 FROM SERVICE AddEmployeeService
 TO SERVICE
'[ServiceBroker/Example/Employee/AddEmployeeService]'
 ON CONTRACT
[ServiceBroker/Example/Employee/AddEmployeeContract];

 SEND ON CONVERSATION @CH
      MESSAGE TYPE
[ServiceBroker/Example/Employee/AddEmployee] (@EmployeeName)
GO
```

To check the results, I query the dynamic management views I mentioned earlier, as well as the destination table:

```
— Look at the results
SELECT * FROM sys.dm_broker_activated_tasks
SELECT * FROM sys.dm_broker_connections
SELECT * FROM sys.dm_broker_forwarded_messages
SELECT * FROM sys.dm_broker_queue_monitors
GO

SELECT *
FROM
Employee
GO
```