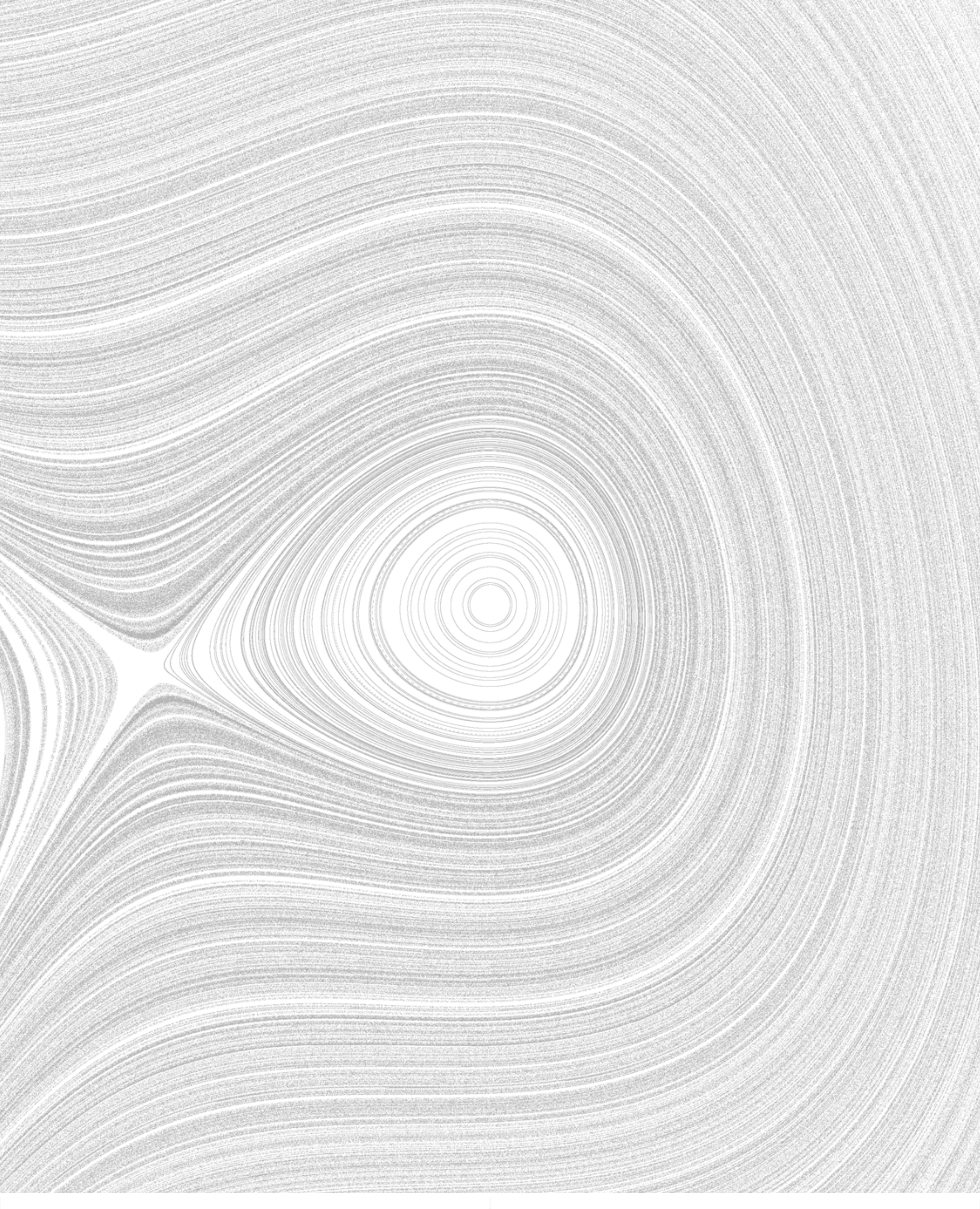

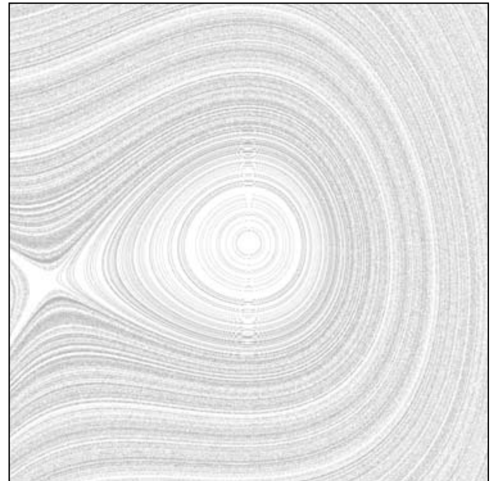

Chapter 7

INTEGRATING VIA DATA AND MEDIA SERVICES

By Shashank Tiwari

So far, you know how to combine Flex and Java using HTTP and web services. The last chapter surveyed a bunch of alternative mechanisms to achieve this. Most of these mechanisms involve loosely coupled text-based data interchange. Most of them interact by pulling data. Only one of them, Hessian, transmits binary data. Only one, Hessian again (with additional infrastructure powered by the new Java IO), allows data push.

Now, we delve into more tightly coupled scenarios and efficient binary data transmission using AMF (Action Message Format). The AMF specification can be accessed online at `http://opensource.adobe.com`. This chapter looks at both pull- and push-based interactions—using data services and media servers. Adobe offers two alternatives for data services—the commercial LifeCycle Data Services (LCDS) and the open source BlazeDS—and it offers a suite of products for media servers: Flash Media Server (FMS) products. There are a few open source alternatives to these as well.

In this chapter, I will analyze these products in the context of their applicability to rich, engaging enterprise-grade applications. Functionally they can be divided into the following three topics:

- Remoting and RPC
- Messaging and data push
- Media streaming

At this point of the book, remoting and RPC should be familiar territory, so let's start there.

Remoting and RPC

Flex applications can access the Java server side using data services. They can access Java objects and invoke remote methods on them. The Flex framework includes a client-side component called `RemoteObject`. This object acts as a proxy for a data service destination on the server. When configured properly, this object handle can be used to invoke RPCs. Before we get into the nitty-gritty of this object and destination configuration, let's step back and look at the data services architecture.

Data services architecture

Figure 7-1 is a pictorial summary of the data services architecture. The view is biased to highlight the functional elements. It includes technical aspects but skips the internal details in many places. As you look deeper into the nuts and bolts in this chapter, many of these details will emerge.

As Figure 7-1 depicts, data services includes the following:

- Gateway to intercept server-bound calls
- Parser to make sense of AMF messages
- Serializer and deserializer to transform objects between ActionScript 3.0 (AS3) and Java
- Manager to coordinate with and delegate responsibility to server-side objects
- Messaging service provider to send and receive messages

By data services, I mean a class of products that enable remoting and messaging over AMF and protocols like Real Time Messaging Protocol (RTMP). RTMP is a proprietary protocol developed by Adobe Systems for streaming audio, video, and data over the Internet. More information on RTMP can be found on Wikipedia at `http://en.wikipedia.org/wiki/Real_Time_Messaging_Protocol` and at `http://osflash.org/documentation/rtmp`.

As mentioned, there are two data services implementations from Adobe and a few open source alternatives. Following are the most popular ones:

- **LifeCycle Data Services (Adobe)**: `http://www.adobe.com/products/livecycle/dataservices/`
- **BlazeDS (open source from Adobe)**: `http://opensource.adobe.com/wiki/display/blazeds/`
- **Granite Data Services (GDS)**: `http://www.graniteds.org/`
- **WebORB for Java**: `http://www.themidnightcoders.com/weborb/java/`
- **OpenAMF**: `http://sourceforge.net/projects/openamf/`

OpenAMF is not a very active project at the time of writing. The last release dates back to 2006. This project's mission was to port AMFPHP to Java. AMF was a closed specification back then, and AMFPHP was a reverse-engineered open source option for PHP servers. OpenAMF is closer to Flash remoting than data services.

This chapter sticks mostly to LCDS and BlazeDS, but the same concepts apply to the alternatives. LCDS and BlazeDS are quite similar in form and structure and share the same codebase. BlazeDS can be thought of as a marginally scaled-down version of LCDS.

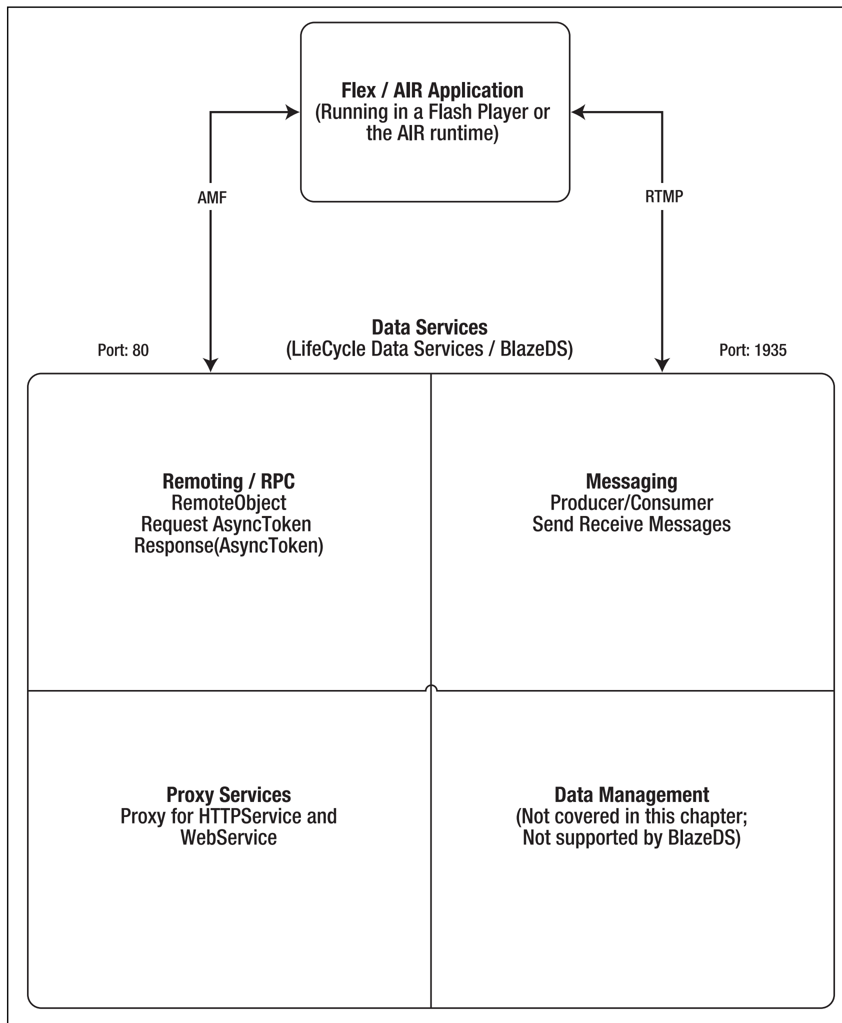

Figure 7-1. Data services architecture: an overview

Data services uses AMF3 (Action Message Format version 3) to transmit binary data between Flex and Java. The genesis of this product lies in the Flash remoting server, which used to support AMF0 as the binary protocol. Flash remoting still exists, in many forms, but data services replaces it as a better alternative.

Whichever data service we choose, it is a web application from a Java server perspective. Let's dig a bit deeper to see what this means. (Of course, we are talking only about data services for Java. There are remoting servers for other technologies, but that is beyond the scope of this book.)

It's a web application

Web applications are applications built on the technology that powers the Web. The HTTP protocol and the associated programming paradigm are a prominent part of this technology set. In Java, the raw low-level HTTP and related infrastructure is abstracted out as a higher-level API and managed components. At the heart of this abstraction is the Servlet specification, which wraps HTTP methods and HTTP protocol handling in objects. Web applications written in Java are packaged with all assets, associated libraries, class files, and any other resources into a special archive file format: a web application archive (WAR). These WAR files are deployed on a servlet container or an application server (which contains a servlet container).

Most data services, especially LCDS and BlazeDS, are web applications and exist as WAR files. The remoting gateway is a servlet, and data service elements are web components. A little later, in the section "Downloading and deploying the web application," you will see how deploying LCDS or BlazeDS is identical to deploying any other web application distributed in a WAR file format.

In both BlazeDS and LCDS, the primary communication responsibilities are handled by a message broker, which is created on startup by a servlet called the `MessageBrokerServlet`. The application server's standard class loader loads it like any other servlet. This message broker is extremely flexible, and all types of available services and endpoints can be configured for it fairly easily. All such configurations reside in an XML configuration file, which I talk about later in the section "Configuring data services."

Protocols, channels, destinations, and endpoints

One of the primary advantages of data services is their use of AMF to transmit data between Flex and Java. AMF is a binary protocol, and the Flash Player natively supports it. Therefore, transmission using AMF is fast and efficient. AMF is a high-level (application layer) protocol that uses HTTP for communication. Almost all data services dialog happens over HTTP or its secure alternative, HTTPS. AMF specification is now available under the open source license and is accessible for download at `http://download.macromedia.com/pub/labs/amf/amf3_spec_121207.pdf`. When remoting, AMF is marshaled and unmarshaled at both ends (Java and Flex) for the data interchange to work.

The data services messaging module and media server use RTMP. LCDS and FMS support RTMP, but BlazeDS does not. BlazeDS uses HTTP tunneling and AMF long pooling to achieve a push-based model. Red5 (`http://osflash.org/red5`), an open source alternative to Flash Media Server, partially reverse-engineers RTMP and provides streaming capabilities over this derived protocol.

Apart from AMF over HTTP and RTMP, the Flash Player also supports Transmission Control Protocol (TCP) over Sockets. TCP is a protocol from the Internet protocol suite that facilitates reliable ordered delivery of byte streams. Secure versions of the protocol, such as AMF and HTTP over Secure Sockets Layer (SSL) and RTMP over Transport Layer Security (TLS), can be used as well. Both SSL and TLS are cryptographic protocols that facilitate secure communication over the Internet. TLS is a newer generation protocol compared to SSL. Although similar, SSL and TLS are not interchangeable. TLS 1.0 is a standard that emerged after SSL 3.0. These protocols involve endpoint authentication, message integrity, and key-based encryption. Both protocols support a bunch of cryptographic algorithms including RSA. RSA is a popular cryptographic algorithm for public key cryptography, which involves two different keys, one to encrypt a message and another to decrypt it.

The Flash Player does not support the entire repertoire of protocols and even misses the ubiquitous User Datagram Protocol (UDP), which is very useful for multicasting. **Multicasting** is a method of information delivery to multiple destinations simultaneously whereby messages are delivered over each link of the network only once. Copies of the message are created only if the links to the destinations bifurcate.

You will not be able to take advantage of protocols like UDP with data services.

Protocols help make effective and efficient communication possible, but higher-level abstractions increase the usability of these protocols. These higher-level abstractions help you focus on business logic and reduce the burden of dealing with low-level communication handling. One such useful higher-level construct in Flex is called **destination**.

Destinations are one of the key abstractions available in the Flex framework and data services. Server-side entities are mapped to logical names and configured to be invoked using these logical names. These configured server-side elements, or destinations, have a handle (logical name) and expose server-side functionality to remote clients. Many Flex client components, especially those that facilitate remoting—for example `RemoteObject`—map to a destination. The section "Configuring data services," which comes a little later, illustrates the configuration files. In that section, you will learn how to define, configure, and use a destination. Then, in the section "Extending data services for advanced remoting use cases," you will see how to use custom extensions as destinations. In data services, almost all server-side elements that facilitate remoting and messaging are configured as destinations.

`HTTPService` and `WebService`, when routed through a data service proxy, also map to a destination.

When you interact with a server-side service via a destination, you use a messaging channel to communicate back and forth. A **messaging channel** is a bundle that defines a protocol and an endpoint set. An endpoint set means a URI, a listening port number, and an endpoint type definition. For example, an AMF channel could be established with the help of the AMF channel implementation class (`mx.messaging.channels.AMFChannel`) and an endpoint definition, which could be a combination of an endpoint type (`flex.messaging.endpoints.AmfEndpoint`) and its availability via a URI (say `/sampleapp/messagebroker/amf`) over a certain listening port (which by default for AMF is 8100).

Sometimes, services need special adapters to communicate with server-side elements. These adapters may translate the message and act as the classical conduit that helps different programming interfaces communicate and interact with each other. (An **adapter** by definition is something that modifies an API to make it adapt to the required integration scenario.) Data services define a set of built-in adapters and provide an API to create your own.

That is enough theory; time now to roll our sleeves up and see data services in action. We first install data services and then quickly create a small example application to see how it works.

Installing a data service

The last section claimed data services to be a Java web application. You will see that claim reinforced by deploying a data service like any other Java web application in a Java application server (with a servlet container). Once we have deployed it successfully, we will go ahead and configure it so that we are ready to build an example application.

Downloading and deploying the web application

For the purposes of illustration, we will pick BlazeDS as the data service and choose JBoss Application Server (AS) as the Java application server. If the data service is LCDS and the application server is any other, such as Apache Tomcat, BEA WebLogic, IBM WebSphere, Apache Geronimo, Caucho Resin, or GlassFish, the situation is not very different. Each of these application servers has its own directory structure and styles of deployment. Deploying BlazeDS or LCDS involves the same level of complexity as deploying any other Java web application packaged as a WAR. Adobe Labs has published some notes on the specific installation instructions for a few of the popular application servers. These notes are online at `http://labs.adobe.com/wiki/index.php/BlazeDS:Release_Notes#Installing_BlazeDS`.

The first step is to get all the required software and to install it.

Getting the software and installing it BlazeDS is available under the open source GNU LGPL license. Go to `http://opensource.adobe.com/wiki/display/blazeds/BlazeDS` and download the latest stable release build. You will have a choice to download the binary or the source versions. The binary version is what you should choose unless you intend to make modifications to the code. The distribution is available as a ZIP file. When you unzip the archive file, you will find a file called `blazeds.war`. This is the WAR file that has everything in it needed to use BlazeDS. In addition, you may find the following files:

- `blazeds-samples.war`: A set of sample applications
- `blazeds-console.war`: Monitoring application for BlazeDS deployments

If this is the first time you are deploying BlazeDS, it's recommended you deploy `blazeds-samples.war` and verify that the sample applications run without a problem.

The other piece of the puzzle is the Java application server. I will assume that you have downloaded and installed one for your use. In this example, we download the latest stable release of JBoss AS from the community download page accessible at `http://www.jboss.org/projects/download/`. At the time of writing, version 4.2.2.GA is the latest stable release. This may differ depending on when you download it. The JBoss AS download is an archive file that is ready to use as soon as it's expanded in the file system. (It may at most require a few environment variable settings.) On the PC, we just unzip it within a directory on the main partition.

The JBoss AS directory server appears as shown in Figure 7-2.

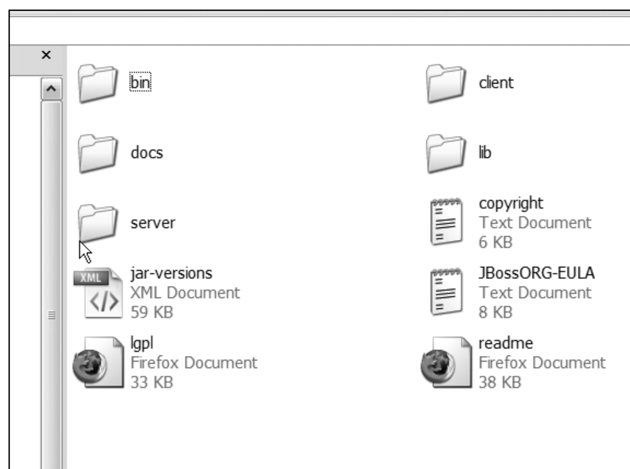

Figure 7-2. JBoss AS directory structured after unzipping it on a PC

BlazeDS runs with any Java application server that supports JDK 1.4.2+ or JDK 5. Most current versions of application servers support these JDK versions.

Deploying the WAR file Deploying a web application in JBoss is as elementary as traversing down the server ➤ default ➤ deploy folders and copying the WAR files there. Once this is done, start up the server. Go to the `bin` directory and start the server with the "run" script. To verify deployment, fire up your browser and request the samples application. In our case, JBoss is bound to

port 8080 for HTTP requests, and so the URL for samples looks like this: `http://localhost:8080/blazeds-samples/`. If you are able to access the samples application, you are ready to move to the next step of configuring data service.

With LCDS, the deployment is no different. The WAR file in LCDS has a different name—`flex.war` as opposed to `blazeds.war`—and it has a few extra features compared to BlazeDS, but from a deployment standpoint things don't change.

GDS, the open source alternative to LCDS, has a slightly different approach to software distribution. GDS is distributed as multiple bundles, each integrating with one Java server-side technology. Also, GDS is distributed as source and not binary, which means you need to set up the development environment to build the software. If you are a Java developer who also writes Flex applications, you may already have the development environment set up. The required pieces of software are

- Eclipse 3.2+ (with JDK 5+)
- Flex 3 SDK (If you use Flex Builder, you already have it.)
- Flex 3 Ant tasks (`http://labs.adobe.com/wiki/index.php/Flex_Ant_Tasks`)

Once the environment is set up, you could get hold of one of these bundles:

- `graniteds-ejb3-1.0.0.zip`: Hooks to EJB
- `graniteds-seam-1.0.0.zip`: Integrates with Seam (stateful web beans)
- `graniteds-spring-ejb3-1.0.0.zip`: Provides Spring services
- `graniteds-guice-1.0.0.zip`: Provides services for Google Guice
- `granite-pojo-1.0.0.zip`: Interfaces with plain Java
- `granite-chat-1.0.0.zip`: Includes Java New I/O (NIO) and Comet-based data push

The choice of bundle depends on your requirements. Also, the bundle version numbers could vary depending on when you download the files. As of now, the release version is 1.0.0. Each of these bundles is an Eclipse project. Once you get the bundle you need, unzip it and import it into Eclipse. Subsequently, you build it using Ant tasks.

Configuring data services

In BlazeDS and LCDS, there is more to configure than code. In both these cases, a message broker servlet is the central manager of all communication and service invocations. Configuring data services is equivalent to configuring this message broker. In the `web.xml` file where you set up this servlet, you define the configuration file it should read. The portion of `web.xml` where you define this is as follows:

```
<servlet>
    <servlet-name>MessageBrokerServlet</servlet-name>
    <display-name>MessageBrokerServlet</display-name>
    <servlet-class>flex.messaging.MessageBrokerServlet</servlet-class>
    <init-param>
        <param-name>services.configuration.file</param-name>
        <param-value>/WEB-INF/flex/services-config.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
```

You may notice that the value for the services.configuration.file is /WEB-INF/flex/services-config.xml. This is the default configuration, and most often there is no need to change it. However, if you need the configuration file to be at a different location or have a different name, you know where to make the modifications so that the new values are picked up.

All service and endpoint configurations are specified in this configuration file, which by default is called services-config.xml. From a functional standpoint, a data service tries to accomplish the following:

- Facilitate remote procedure calls to Java objects on the server and transmit data between AS3 and Java classes.
- Provide proxy services, especially for HTTPService and WebService, where security restrictions (the lack of a cross-domain definition via crossdomain.xml) disallow these services otherwise.
- Send/Receive messages between Flex and Java and between two different Flex clients.
- Manage data for the application. This topic is not discussed in this book at all. BlazeDS does not provide this off the shelf, but LCDS does.

Therefore, Adobe partitions the configuration file into four different pieces, each corresponding to one of the functional responsibilities just listed. Each of these pieces resides in a separate file, and all are included in the original configuration file by reference. The default names for these four files, in the same order in which they correspond to the functional areas listed previously, are

- remoting-config.xml
- proxy-config.xml
- messaging-config.xml
- data-management-config.xml

The portion of services-config.xml at /flex/WEB-INF from our BlazeDS installation, where three of these four files are included (BlazeDS does not have data management features), is as shown here:

```
<?xml version="1.0" encoding="UTF-8"?>
<services-config>
    <services>
        <service-include file-path="remoting-config.xml" />
        <service-include file-path="proxy-config.xml" />
        <service-include file-path="messaging-config.xml" />
    </services>
```

A few aspects like logging, security, and channel definitions are cross-cutting concerns and are used across services, so these are defined in services-config.xml itself. All other service configurations and definitions typically fall in one of the four files (or three if we are using BlazeDS) I spoke about.

In this chapter, there is no intent to cover every single aspect of configuration. Only a few important ones are sampled and explained. For an exhaustive syntax-level account of each allowed configuration, it's advisable to refer to the LiveDocs. For BlazeDS, you could refer specifically to a LiveDocs section titled “About service configuration files,” which can be found online at http://livedocs.adobe.com/blazeds/1/blazeds_devguide/help.html?content=services_config_2.html.

Let's survey a few configuration options to get a flavor of things.

Common configuration Let's start with logging. BlazeDS and LCDS use log4j for logging. Logging-related configurations reside in `services-config.xml` itself. The most important aspect of configuration is the logging level. The permissible values and their respective meanings are as follows:

- `ALL`: Logs every single message.
- `DEBUG`: Includes internal Flex activities. This is an appropriate level during development and troubleshooting, and is an incremental expansion beyond `INFO`. Therefore, all errors, warnings and information messages are included.
- `INFO`: Logs additional information that may be pertinent to developers or administrators. This level builds on top of the `WARN` level.
- `WARN`: Includes warnings as well as errors.
- `ERROR`: Logs only errors that cause service disruption.
- `NONE`: Logs nothing.

If you are familiar with log4j logging levels, then you have seen this before.

Next come channel definitions. In terms of importance, this rates above the logging-level definitions. Channels are the vital protocol and endpoint combination that make communication possible between the Flex client and the server. In BlazeDS, the default AMF channel configurations look like this:

```
<channels>
    <channel-definition id="my-amf" class=➥
"mx.messaging.channels.AMFChannel">
        <endpoint url="http://{server.name}:{server.port}/➥
{context.root}/messagebroker/amf"
class="flex.messaging.endpoints.AMFEndpoint"/>
    </channel-definition>

    <channel-definition id="my-secure-amf"
class="mx.messaging.channels.SecureAMFChannel">
        <endpoint url="https://{server.name}:{server.port}/➥
{context.root}/messagebroker/amfsecure"
class="flex.messaging.endpoints.SecureAMFEndpoint"/>
        <properties>
            <add-no-cache-headers>false</add-no-cache-headers>
        </properties>
    </channel-definition>

    <channel-definition id="my-polling-amf"
class="mx.messaging.channels.AMFChannel">
        <endpoint url="http://{server.name}:{server.port}/➥
{context.root}/messagebroker/amfpolling"
class="flex.messaging.endpoints.AMFEndpoint"/>
        <properties>
            <polling-enabled>true</polling-enabled>
            <polling-interval-seconds>4</polling-interval-seconds>
        </properties>
    </channel-definition>
</channels>
```

Three different AMF channels are defined using the preceding configuration. In each case, a fully qualified class name specifies the class that implements the channel. The channel is accessible via a configured endpoint. The endpoints include a set of tokens, namely `server.name`, `server.port`, and `context.root`. When a SWF is loaded in a browser, as happens with all Flex applications, these tokens are replaced with the correct values, and the endpoints are configured properly. In AIR and even with RTMP channels (in RTMP, `server.port` needs a specific port number definition), these tokens are not resolved automatically, and so channels don't work as expected, if configured using tokens.

To test a channel, it may a good idea to try and access the endpoint URL with the browser and see whether you get a success message, such as `200 OK`, or not. Where required, channels could accept additional properties. As an example, a polling AMF channel defines the polling interval using properties. An interesting fact is that property settings can create entirely different channels. AMF and polling AMF channels have the same class for implementing the channels, but they have different sets of properties and therefore different behavior.

Using `services-config.xml`, the channel configurations are done at compile time. It's also possible to configure channels and associate them with destinations at run time. More information about configuration at run time is covered in the section "Configuring at run time" later in this chapter.

The third important common configuration pertains to security. Exhaustive and complex security definitions are possible with data services, but we will go with the defaults for now. We shall deal with security configurations in data services in the section "Additional useful data services tips" later in this chapter.

Although this chapter has not exhaustively covered the configuration options, you know the basics of configuration by now. You will learn more about configuration as you explore the other topics that relate to data services.

Our new focus is to get data services into action. The three configuration files, `remoting-config.xml`, `proxy-config.xml`, and `messaging-config.xml`, configure services, so we will look at these in the next few sections as we implement such services using BlazeDS or LCDS.

Calling remote methods and serializing objects

You know AMF is an efficient binary protocol and a data service lets you exchange data between Flex and Java. Let's dig deeper so you can see what you need to do to leverage this mechanism in your Flex application. Because "seeing is believing" and, like pictures, working code speaks louder than words, we will first create a simple example application that will establish RPC over AMF using a data service. In this example, our data service is BlazeDS.

A simple example in action

The example application is fairly simple. It displays a list of people and the country they come from. A person is identified by an ID, first name, and last name. A country is identified by a name. The initial list is populated from an XML data file that has four names in it. A user is allowed to add names to the list and delete existing ones.

To demonstrate data services and remoting, the list and the methods to manipulate the list are kept on the server. These remote Java objects and their methods are accessed from a Flex client.

We create this example using Flex Builder 3, but we could also do without it and compile directly using `mxmlc`, the command-line compiler, or use Flex Ant tasks, available from within Eclipse. The

command-line compiler and the Flex Ant tasks are available free of charge. The command-line compiler comes bundled with the free Flex 3 SDK. The Flex Ant tasks software bundle can be downloaded from `http://labs.adobe.com/wiki/index.php/Flex_Ant_Tasks`.

As a first step, we create a new Flex project, choose Web application as the application type and J2EE as the server technology. We name this project VerySimpleRemotingExample. Figure 7-3 shows these settings in the New Flex Project dialog.

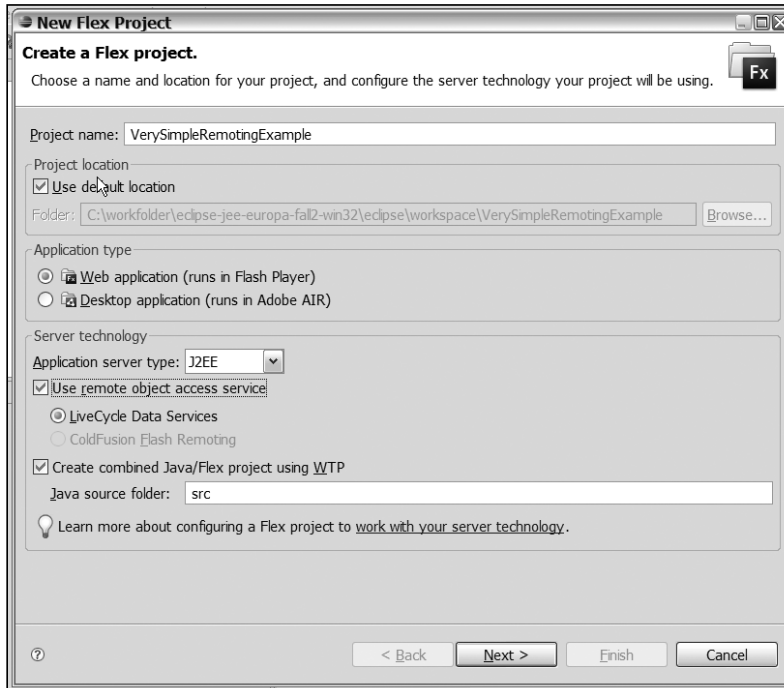

Figure 7-3. The initial screen for new Flex project creation in Flex Builder 3

You will see that Flex Builder offers only two choices for a data service, namely LifeCycle Data Services and ColdFusion Flash Remoting. At first, this often causes confusion among developers, when using BlazeDS. Choose LifeCycle Data Services as the option in the current dialog box when using BlazeDS. In the next dialog box, you will have an opportunity to point to either the BlazeDS WAR or the BlazeDS directories, deployed on your application server.

In the very first screen, a choice is offered to create a combined Java/Flex project using WTP. WTP, which stands for Web Tools Platform, is the set of tools available within Eclipse to create Java web applications. The WTP project is hosted at `http://www.eclipse.org/webtools/`. You can go to the project page to get more information on WTP.

The "data services" software available for the Flex framework consists of web applications that run in a web container. Any custom logic you write with the data services either will be part of a web application or will be accessed from a web application. Therefore, if you are starting from scratch, choosing to create a joint project is often useful. If your Java-side application already exists, as in the case of

integration with existing systems, do not choose this option. Depending on the choice at this stage, the next screen varies.

If you choose the option to create the combined project, the screen lets you define the following:

- Target runtime: The application server on which you wish to deploy the data services application. Eclipse allows you to configure the application server run time. A JBoss AS instance is configured for the purpose. The details of JBoss AS configuration are out of the scope of this chapter. However, a quick search on the Web or a look at the JBoss AS documentation (available online at `http://www.jboss.org/jbossas/docs/`) could help you configure a JBoss AS successfully.
- Context root: The root of the web application, usually the same as the application name.
- Content folder: The folder for the source files.
- Flex WAR file: The location of the data services WAR file. The LCDS WAR file is called `flex.war`, and the BlazeDS WAR file is called `blazeds.war`. We are using BlazeDS, so we choose `blazeds.war` from wherever we have locally saved it on our file system. You may recall I said you would get a chance to define BlazeDS-specific files. Here is where you do it.
- Compilation options: Options that let you compile the application either locally or on the server when the page is viewed. If you are a servlet or JSP developer, you know that you can either precompile these artifacts or compile them when accessed, initialized, and served based on user request. During development, it's advisable to compile locally in order to catch any errors and warnings as soon as possible.
- Output folder: The folder where the compiled and packaged files are saved.

Figure 7-4 shows what this screen looks like.

As I said before, it's possible you may be trying to wire up a Flex interface to your existing Java web application or have different teams work on the two pieces fairly independent of each other. In such cases, you do not choose the combined project option. If you reject the combined project option, your next screen will vary from the one in Figure 7-4. This time you are presented with a dialog to specify the following:

- Root folder: This is the folder where the existing web application is deployed. Make sure this folder contains a `WEB-INF/flex` folder.
- Root URL: This is the URL of the web application. In our case, we configure JBoss AS to listen for HTTP requests on port 8080, and we run it on the same machine from which we access the Flex interface via a browser, so the value is `http://localhost:8080/MyWebApplication`.
- Context root: The root of the web application, typically the same as its name.
- Compilation options: Already explained, in the context of the joint Java/Flex application.
- Output folder: The folder where the compiled and packaged files are saved.

Figure 7-5 shows this alternative screen, which was captured before any choices were made. This is intentional. Flex comes with an integrated JRun Java application server. You can choose to use it or ignore it in favor of any other Java application server. We ignored it and chose JBoss. However, if you decided to choose JRun, you have the advantage of Flex Builder's default configurations coinciding with what JRun wants. In Figure 7-5, you see these choices grayed out because the Use default location for LifeCycle Data Services server option is selected.

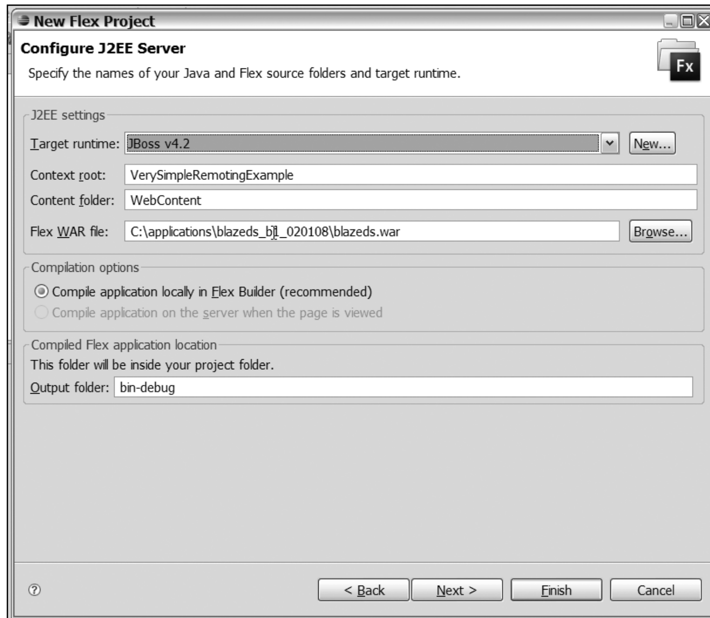

Figure 7-4. The dialog box that appears if you choose to go with the combined Java/Flex project using the WTP option

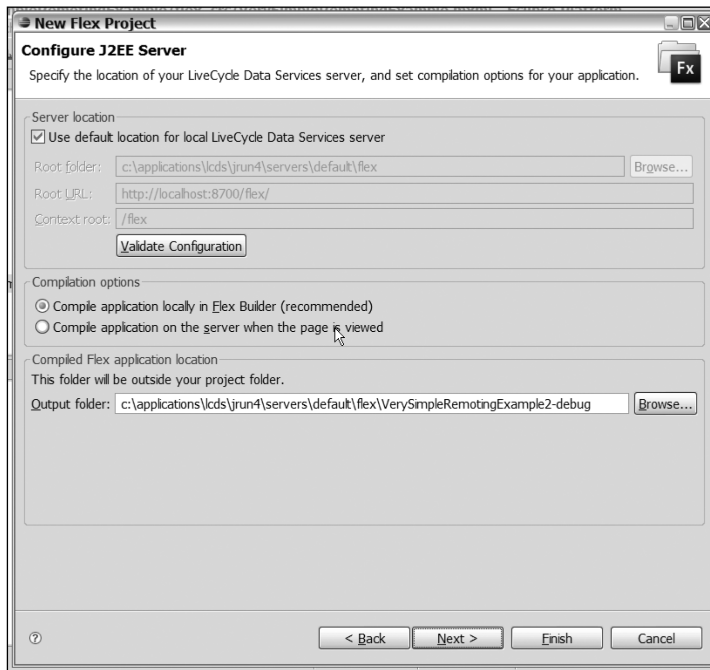

Figure 7-5. The dialog box that appears when an existing Java web application is referenced as the target for the data services application

In either of the two cases, once you make your selections, you move to the next screen, shown in Figure 7-6, which lets you add any external library files to the project and asks you to confirm the project creation. When you click the Finish button, the project is created.

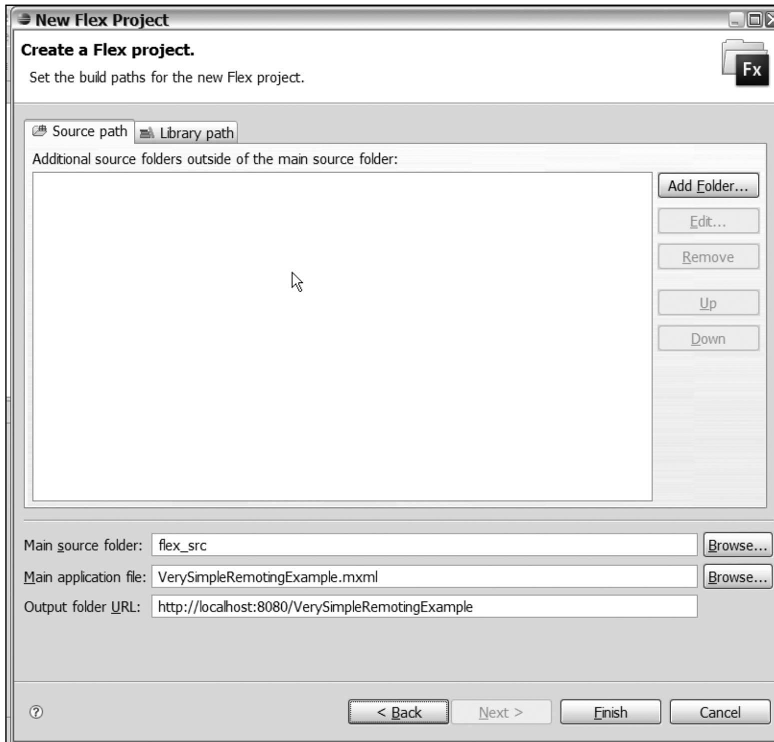

Figure 7-6. Final wizard screen for new Flex project creation

A new Flex (with data services) project is created in Flex Builder. The directory structure of our newly created project looks as illustrated in Figure 7-7. In this figure, notice a `WEB-INF` folder appears in the `WebContent` folder. This folder contains most things pertaining to BlazeDS. The `flex` folder in `WEB-INF` contains all the configuration files. The `lib` folder contains all the JARs that implement the BlazeDS functionality.

Now we are set up, and it's time to start building the example application. To represent the data, we define a `Person` class, which has the following attributes:

- `id`
- `firstName`
- `lastName`
- `country`

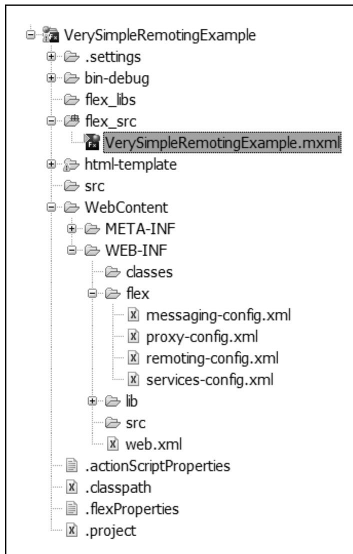

Figure 7-7. The new Flex project, VerySimpleRemotingExample, in Flex Builder 3

We also define a collection of persons that we call `PeopleCollection`. `PeopleCollection` contains `Person` objects. The object to represent these abstractions is created both in Java and AS3. Here is what the Java class looks like:

```
package advancedflex3.ch07.remoting;
public class Person
{
   private int id;
   private String firstName;
   private String lastName;
   private String country;

   public int getId() {
       return id;
   }

   public void setId(int id) {
       this.id = id;
   }

   public String getFirstName() {
       return firstName;
   }

   public void setFirstName(String firstName) {
       this.firstName = firstName;
   }
```

```
    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getCountry() {
        return country;
    }

    public void setCountry(String country) {
        this.country = country;
    }
}
```

If you are a Java developer, you will quickly recognize this class as a JavaBean type of class. It has a set of properties and set of accessor methods to get and set the attribute values. We create a similar structure in AS3. Again, we will have a set of attributes and a set of accessors to get/set those attributes. The attribute names in the Java class and the AS3 class are identical because that is how the fields are matched. This is what the AS3 class looks like:

```
package advancedflex3.ch07.remoting
{
    [RemoteClass(alias="advancedflex3.ch07.remoting.Person")]
    [Managed]
    public class Person
    {
        public var id:int;
        public var firstName:String;
        public var lastName:String;
        public var country:String;

    }
}
```

Looking at the code, you may wonder where the accessor methods are. In AS3, a class with public variables has implicit accessor methods. Therefore, the lack of explicit get and set method pairs is not a problem in this situation, and things will still work. If the properties (attributes) were private, explicit getters and setters would be required. Also, notice that the `RemoteClass` metadata links the AS3 class to its Java counterpart by specifying the fully qualified class name of the Java class as a value of the `alias` parameter. Further, you see this class annotated with another metadata element: `Managed`. Annotating a class with the `Managed` metadata tag is another way of implementing the `mx.data.IManaged` interface. The `IManaged` interface extends the `IPropertyChangeNotifier` interface. `IPropertyChangeNotifier` is a marker interface, which defines a contract such that all implementing classes dispatch a property change event for all properties of the class and any nested class exposed publicly as properties. As a result, as changes occur, the remote server class and the local class are kept in sync.

By explicitly defining a counterpart to our server-side Java class, we establish a typed association. Therefore, on data interchange, the deserialization of the serialized Java class involves mapping of the data elements into this ActionScript class. If we did not define such a class on the ActionScript side, the deserialization would still happen successfully, except that this time the ActionScript class would be dynamically created. From a performance and optimization perspective, typed associations are preferred over dynamic creation.

Next, we create a collection class that holds the set of `Person` objects on both sides of the wire, and define properties and accessor methods on it. Again, we keep the names and definitions consistent on both sides.

Our collection class on the Java server side is called `PeopleCollection.java`, and this is how it looks:

```
package advancedflex3.ch07.remoting;

import java.io.*;
import java.net.URLDecoder;

import javax.xml.parsers.*;
import org.w3c.dom.*;
import java.util.List;
import java.util.ArrayList;

public class PeopleCollection
{
    public PeopleCollection() {

    }

    public List getPersons() {

        List list = new ArrayList();

        try {
            String filePath = URLDecoder.decode(getClass().➥
getClassLoader().getResource➥
("advancedflex3/ch07/PeopleCollection.xml").➥
getFile(), "UTF-8");;
            DocumentBuilderFactory factory =
DocumentBuilderFactory.newInstance();
            factory.setValidating(false);
            Document doc =
factory.newDocumentBuilder().➥
parse(new File(filePath));
            NodeList personNodes = doc.getElementsByTagName("person");
            int length = personNodes.getLength();
            Person person;
            Node personNode;
            for (int i=0; i<length; i++) {
                personNode = personNodes.item(i);
```

```
                person = new Person();
                person.setId(getIntegerValue(personNode, "id"));
                person.setFirstName(getStringValue➥
(personNode, "firstName"));
                person.setLastName(getStringValue➥
(personNode, "lastName"));
                person.setCountry(getStringValue➥
(personNode, "country"));
            }
        } catch (Exception e) {
            e.printStackTrace();
        }

        return list;
    }

    private String getStringValue(Node node, String name) {
        return ((Element) node).getElementsByTagName➥
(name).item(0).getFirstChild().getNodeValue();
    }

    private int getIntegerValue(Node node, String name) {
        return Integer.parseInt(getStringValue(node, name) );
    }
}
```

You will notice that this class reads the data from an XML file called PeopleCollection.xml. This XML file contains four records, each of which corresponds to an attribute of the Person class. Here are the entries in that file:

```
<peopleCollection>
    <person>
        <id>1</id>
        <firstName>John</firstName>
        <lastName>Smith</lastName>
        <country>United States</country>
    </person>
    <person>
        <id>2</id>
        <firstName>Amit</firstName>
        <lastName>Sharma</lastName>
        <country>India</country>
    </person>
    <person>
        <id>3</id>
        <firstName>Alex</firstName>
        <lastName>Smirnov</lastName>
        <country>Russia</country>
    </person>
```

```
    <person>
        <id>4</id>
        <firstName>Ying</firstName>
        <lastName>Chen</lastName>
        <country>China</country>
    </person>
</peopleCollection>
```

That takes care of the data on the server side. Let's now configure these classes with data services. Once we have done that successfully, we will move on to consuming this remote data by calling remote methods from the Flex client.

We get into the WEB-INF/flex directory and open remoting-config.xml. You already know that remoting-config.xml is one of the constituent files of services-config.xml, which configures the message broker servlet and allows data services to deliver the expected functionality.

We configure the PeopleCollection object as a remoting destination so that it becomes accessible from the Flex client. Here is the snippet from remoting-config.xml:

```
<service id="remoting-service"
    class="flex.messaging.services.RemotingService">
    <adapters>
        <adapter-definition id="java-object"
class="flex.messaging.services.➥
remoting.adapters.JavaAdapter"
default="true"/>
    </adapters>

    <default-channels>
        <channel ref="my-amf"/>
    </default-channels>

    <destination id="peopleCollection">
        <properties>
            <source>advancedflex3.ch07.remoting.➥
PeopleCollection</source>
        </properties>
    </destination>
</service>
```

The adapters and default channels we use here were preconfigured with the BlazeDS distribution. It's possible to configure additional adapters and modify the default channels. The piece we add to this configuration file is the one that defines the destination. That configuration maps a logical name (handle) to the class, identified by the fully qualified class name, which holds the collection of persons. In Flex, the logical name will be referenced from within RemoteObject to invoke methods on this class. If you are familiar with any distributed computing technology like web services, EJB, COM, CORBA, or any other RPC-supporting technology, you will find this extremely familiar in approach and style.

A destination allows many more property settings beyond the essential source property determination. One of these is the scope property, which takes three valid values: Application, Session, and

Request. Application-scoped remote objects get bound to the ServletContext and are available to the entire web application. Session-scoped objects are bound to a FlexSession instance, which abstracts the session of the current conversation. Request-scoped objects are instantiated on each invocation. Other properties commonly used are the attribute-id and the factory properties. Object instances are bound to the name specified as the value of the attribute-id property. The factory property specifies the factory that creates the object. In the case of managed components, this property plays a major role. Later in the chapter, in the section "Creating custom factories," you will see this being used.

On the Java server side we are done, except for compiling the code and copying the class files over in the proper directory structure, which maps to the package structure, to WEB-INF/classes. This is how BlazeDS would be able to load these classes. It is a standard Java web application requirement to have the compiled Java classes available either in the WEB-INF/classes folder or as a packaged archive in the WEB-INF/lib folder for it to be available to the web application.

The last part of this example is the Flex application code, which calls the remote methods, gets the data across the wire, and populates a data grid with that data. For now, all of this is put in a single MXML file. The source of that MXML file is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
xmlns:mx="http://www.adobe.com/2006/mxml"
layout="absolute"
creationComplete="initApp()">
<mx:Script>
    <![CDATA[
        import advancedflex3.ch07.remoting.Person;
        import mx.rpc.events.ResultEvent;
        import mx.collections.ArrayCollection;
        import mx.controls.Alert;

        [Bindable]
        private var items:ArrayCollection;

        private function initApp():void
        {
            ro.getPersons();

        }

        private function resultHandler(event:ResultEvent):void
        {
            items = event.result as ArrayCollection;
            var length:int = items.length;

        }
    ]]>
</mx:Script>
    <mx: RemoteObject
id="ro"
```

```
destination="peopleCollection"
result="resultHandler(event)"/>
    <mx:DataGrid dataProvider="{items}" />
</mx:Application>
```

RemoteObject maps to the PeopleCollection Java class via a destination, which we configured in remoting-config.xml. In the preceding MXML file, we could call the getPersons method of the PeopleCollection class using the remote object handle. If we wanted to manipulate the data further and manipulate the items in this collection, we could do that, because each of the items in the collection is a Person object that has a type association in AS3. In the interest of keeping our focus on the basic mechanics of remoting, I will avoid discussing any details of such business logic. However, the code for the application contains an implementation for updating the collection, adding new items and deleting existing ones, and sorting items in the collection. You can peruse the source code to understand how it's implemented.

Now that you have seen a basic remoting example, the next section builds on this knowledge to cover two important related topics:

- Java to/from AS3 serialization/deserialization
- Asynchronous communication

We will start with serialization and deserialization.

Java to Flex serialization basics

The simplest of our examples established that objects get serialized and deserialized between Java and AS3, across the wire. To analyze this subject further, it is appropriate to compare and contrast the available data types in the two languages and see how they map to each other. Table 7-1 presents this comparison tersely.

Table 7-1. Juxtaposing AS3 and Java Data Types

AS3	Java
int/uint	Integer
Number	Double
Object	Map
String	String
Boolean	Boolean
Date	java.util.Date
Array (dense)	List
Array (sparse)	Map
XML	org.w3c.dom.Document
flash.utils.ByteArray	byte[]
undefined	null

This table is not exhaustive but should serve as an introductory guide. Here are a few notes to help you understand the mappings:

- Java `float`, `long`, `BigDecimal`, and `BigInteger` convert to AS3 `Number`.
- `Date`, `Time`, and `Timestamp` (from the `java.sql` package) and `java.util.Calendar` map to AS3 `Date`.
- In AS3, “true” and “false” strings can also represent a Boolean value.
- Dense arrays are those that have no holes. As an example, if an array has four elements and I put the fifth element at position 9, then in a sparse array (the opposite of a dense array) there will be no values, what I call holes, in position 4 to 8. ActionScript sparse arrays tally with `java.util.Map` to avoid sending null values across.
- Depending on the interface type at the Java end, a dense ActionScript array maps to an implementation type. For example, `List` and `Collection` become `ArrayList`, `Set` becomes `HashSet`, and `SortedSet` becomes `TreeSet`.
- Typed associations can be established between custom types with the help of corresponding bean type classes on each side and the `[RemoteClass]` metadata tag. Our simple example uses such a mapping.

In the case of custom type associations, it’s possible to include extra properties in the bean type object as far as the existing ones on the source (i.e., the server) match the other properties in the ActionScript type. For example, you may have an object represent an order with a single line item. In such an object, you may want the serializable object to contain the unit price of the item and the quantity of the item, but not the total value (which is essentially a product of the other two properties). You may want the AS3 type to have a third property to hold the total value, which you could calculate by multiplying the other two property values. If you did this and made this AS3 object the associated counterpart of the Java object (which effectively has one property less), nothing would break; everything will still work fine. It’s advisable to follow this technique and reduce the serialization overhead, especially when derived and calculated values are involved.

Serialization and deserialization with transmission over AMF is a very efficient process and beats almost all other transmission alternatives.

I mentioned earlier on that I would be covering two more special topics related to remoting. This section dealt with one, and now we will move on to the other—asynchronous communication, the only style that Flex adopts.

Asynchronous communication

All types of communication in the Flex framework are asynchronous. This has its advantages but also poses some challenges. To list a few, some advantages are as follows:

- Calls are nonblocking and therefore resources are not held up.
- Asynchronous communication and events go well together and help create loosely coupled systems.
- This type of communication allows clean separation of duties: the call invoker and the response receiver could be different entities.
- The sequence in which calls are issued, or their linearity, have little significance, and so there is usually no need to queue calls.

No choice has advantages without disadvantages. The disadvantages of asynchronous communication are often caused by the same things that make it advantageous. A few common challenges of asynchronous communication are as follows:

- Transaction management is not trivial because we are not talking about call and response in the same cycle. To roll back, it's important to have a common token to relate back to the event.
- Flows that need calls in a linear order need to be managed externally.
- In many cases, the invoker has little control over the response, i.e., when it arrives and how it is handled.

A common established way to manage asynchronous communication is to espouse the Asynchronous Completion Token (ACT) pattern, which originated at the Department of Computer Science and Engineering at Washington University in St. Louis. This pattern's universal applicability in the area of asynchronous communication caused it to become popular rather quickly. You can find a description and an explanation of the pattern online at `http://www.cs.wustl.edu/~schmidt/PDF/ACT.pdf`.

The Flex framework adopts ACT ideas natively in its framework constructs. Almost all asynchronous communication in Flex follows this pattern:

- A call is made and an asynchronous token is returned.
- Every response dispatches events, indicating either success or failure.
- Event listeners are registered with the object that is capable of receiving the response. Sometimes the implementation of a marker interface makes a class eligible to receive responses.
- When a response arrives, events are dispatched. The response comes back with the asynchronous token. Call and response are correlated using this token.
- The appropriate handler gets the response. On receipt, it processes the response.

By default, a call is dispatched as soon as it's made, and there is never any certainty about when the response will come. Let's take a simple example of an editable collection in a data grid, where this behavior could make our life difficult. Say we have a few rows of data. Now we select one of the rows of data and edit the values, and then we do the same with another row. When we are editing the second row, we realize our first row modification wasn't accurate, and so we need to revisit it. Under normal circumstances, the first row modification call has already been dispatched by now, and rolling back the modification is cumbersome. This situation can get more complex if our operation deletes a row and we need to recover it, or if we make two modifications to the same data element and the first modification completes after the second one. In general, to take care of these types of complexities, we need to allow the following:

- Sequencing of calls in some order
- Batching of calls into a logical bundle
- Locking of data elements, where applicable
- Definition of transactional boundaries to facilitate commit and rollback

In LCDS, the data management module takes care of these complicated scenarios, but in most other cases you need to take care of this yourself.

Extending data services for advanced remoting use cases

You have seen the basics of remoting to simple Java classes, which a few years back were given the interesting name of Plain Old Java Objects (POJOs). Now let's look at a couple of advanced use cases.

Supporting additional data types

Although AS3 and Java are strikingly similar in programming style, static type system, and syntax, the mapping between AS3 and Java objects is far from optimal. One of the key reasons for this is the lack of AS3 parallels for many Java data types, a glaring example of which can be seen in the two languages' collection types. Java has many advanced collection data types, whereas AS3 has very few. For example, there is no equivalent of a `SortedSet` in AS3. Even if such data types were added to AS3, how could they be mapped to the existing Java data types? There is no way of translating automatically between the two. For instance, a strongly typed enumerated type can be created in AS3 to resemble a Java 5 `Enum`, but it's by no means easy to ensure that serialization and deserialization happen between the two smoothly.

Adding support for additional data types is possible but not trivial. In order to understand the path to this addition, it's important to understand the `PropertyProxy` interface. `PropertyProxy` in the `flex.messaging.io` package allows customized serialization and deserialization of complex objects. It has access to each of the steps in the serialization and the deserialization process. During serialization, a `PropertyProxy` is asked to provide the class name, its properties, and its peculiarities. During deserialization, a `PropertyProxy` instantiates an object instance and sets the property values. `PropertyProxy` is a higher-order interface that has been implemented for many different data types. In the BlazeDS Javadocs, you will see the following classes implementing the `PropertyProxy` interface:

- `AbstractProxy`
- `BeanProxy`
- `DictionaryProxy`
- `MapProxy`
- `PageableRowSetProxy`
- `SerializationProxy`
- `StatusInfoProxy`
- `ThrowableProxy`

When adding support for a specific new data type, you could start by either implementing the `PropertyProxy` for that type or extending one of its available implementations.

Creating custom factories

Data services (both LCDS and BlazeDS) can load and instantiate simple Java classes without a problem. However, they don't work without modification if the remote object is a managed object like an EJB or a Spring bean. This is only natural, because these data services cannot automatically instantiate these objects. Managed objects are instantiated, maintained, and garbage collected within the managed environment or container they reside in. If these objects need to be consumed within data services, they warrant a factory mechanism, which can get hold of a managed object instance and make it accessible within a data service namespace.

The custom factory varies depending on how the managed object is accessed. Both LCDS and BlazeDS include a so-called factory mechanism to include objects that reside in other namespaces. Theoretically, the idea is simple and goes like this:

1. Implement a common interface, which defines a method to create a factory.
2. Configure this factory creator so that it can be instantiated and used when needed.
3. Pass the name of this configured factory creator to a destination (which points to the managed object), so that it knows which factory to use to get hold of an object instance.
4. Use the factory to look up an instance of the managed object. Though called a factory, this factory is not really creating anything. It's looking up an instance from a different namespace.
5. Use the object as you would use any other simple Java class. In other words, once bound to a destination and configured, it is ready for RPC.

Practically, most work goes into implementing the lookup method. Start with a custom class and have that custom class implement the `FlexFactory` interface. Implement the three most important methods, namely

- `initialize`: To get the factory instance configured and ready
- `createFactoryInstance`: To create a `FactoryInstance`
- `lookup`: To return a handle of the object the factory is meant to get hold of

Then configure the factory in `services-config.xml` like this:

```
<factories>
    <factory
id="ejbFactory"
class="flex.samples.factories.EJBFactory"/>
</factories>
```

We configure an EJB factory here. Now we can refer to this factory from within a destination setting. A possible case could be the following:

```
<destination id="MyEJBPoweredService">
    <properties>
        <factory>ejbFactory</factory>
        <source>MyUsefulBean</source>
    </properties>
</destination>
```

We have successfully created a custom factory and it's ready for use. Once again, I show only the logic that helps you learn the topic at hand—namely, creating custom factories.

This chapter has covered a fair bit on remoting, although it has merely scratched the surface. Next, I give you a look at the second most important feature of BlazeDS: messaging.

Messaging and real-time updates

Flex reconfirms the supremacy of event-driven systems. It thrives on asynchronous communication, and it defines the loosely coupled robustness that excites all of us to create fascinating applications using it. A concept that goes hand in hand with events and asynchronous communication is message-driven communication. Flex includes it in its repertoire of core features.

Messaging in Flex, with its adapters and channels, makes it possible to push data up from a server or a client to another client. Although today data push can be done in Java using the Java NIO Comet technique, the BlazeDS messaging-based data push provides an easy option to create rich, engaging, real-time event-driven systems. More information on Java NIO and Comet can be obtained from the following links:

- `http://en.wikipedia.org/wiki/New_I/O`
- `http://en.wikipedia.org/wiki/Comet_(programming)`

Essential messaging and pushing data

To understand messaging in Flex, we will dive into a sample application first and then explore the different aspects of the model in the light of the application. The application is a collaborative form that is filled in by two users at the same time. Changes on either side are propagated to the other one. A possible use case could be travel booking by a customer in association with an agent, where the agent assists in real time by validating the data and adding expertise to find the best deal.

There are two main roles in the world of messaging: message producer and message consumer. A **message producer**, as the name suggests, produces messages, and a **message consumer** consumes messages. Our collaborative application acts as both a producer and a consumer. As a producer, it sends all updates out as messages. Subscribed consumers receive these messages and process them. The collaborative application is used by two users, although in its current shape the application itself doesn't restrict the number of users collaborating simultaneously. Each instance of the application acts as both consumer and producer, so an instance acting as a consumer receives its own messages, sent in the capacity of a producer. Updates, when applied to the producing instance, have no impact because the data is already present there; after all, it's the source of the updates.

In a real-life situation, you may want to bind these roles to specific parts of the form and have logic to skip and apply updates, but here we adopt the most elementary approach. You can get hold of the code and extend it to include any or all of these features.

Our collaborative application is in a single MXML file, which appears as shown here:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp()">
    <mx:Script>
        <![CDATA[
            import mx.messaging.messages.*;
            import mx.messaging.events.*;

            private function processUserFormInput➥
```

```
(uName:String, uRequest:String):void {
                /* Set first message parameter to */
/* identify the message sender: a user or an agent */
                sendMessage("user",uName, uRequest);
            }

            private function processAgentFormInput➥
(aName:String, aComments:String):void {
                sendMessage("agent",aName, aComments);
            }

            private function initApp():void {
                consumer.subscribe();
            }

            private function messageHandler(event: MessageEvent):void {
                var paramArray:Array = (event.message.body).split(":");
                if(paramArray[0] == "user") {
                    userName = paramArray[1];
                    userRequest = paramArray[2];
                }else if(paramArray[0] == "agent") {
                    agentName = paramArray[1];
                    agentComments = paramArray[2];
                }

            }

            private function sendMessage➥
(param0:String,param1:String, param2:String):void {
                var message: AsyncMessage = new AsyncMessage();
                message.body = param0 + ":" + param1 + ": " + param2;
                producer.send(message);
            }
        ]]>
    </mx:Script>

    <mx:Producer id="producer" destination="CollaborationTopic"/>
    <mx:Consumer id="consumer"
destination="CollaborationTopic"
message="messageHandler(event)"/>

    <mx:Form id="collaborativeForm1" defaultButton="{updateUserInput}">
        <mx:FormItem label="User Name">
            <mx:TextInput id="userName"/>
        </mx:FormItem>
        <mx:FormItem label="User Request">
            <mx:TextInput id="userRequest"/>
        </mx:FormItem>
        <mx:FormItem>
```

```
            <mx:Button label="Update User Inputs" id="updateUserInput"
                click="processUserFormInput➥
(userName.text, userRequest.text);"/>
        </mx:FormItem>
    </mx:Form>
    <mx:Form
id="collaborativeForm2"
defaultButton="{updateAgentInput}">
       <mx:FormItem label="Agent Name">
            <mx:TextInput id="agentName"/>
        </mx:FormItem>
        <mx:FormItem label="Agent Comments">
            <mx:TextInput id="agentComments"/>
        </mx:FormItem>
        <mx:FormItem>
            <mx:Button label="Update Agent Input" id="updateAgentInput"
                click="processAgentFormInput➥
(agentName.text, agentComments.text);"/>
        </mx:FormItem>
    </mx:Form>
</mx:Application>
```

The most interesting new controls in the preceding code are those that relate to a producer and a consumer. The Producer and the Consumer point to a server-side destination, which (as you will have noticed) have the same value. Data services provide messaging services, which are accessible through destinations like the remoting services. Most data services support the two main messaging domains: point-to-point and publish-and-subscribe. Point-to-point messaging is about sending messages from one queue to the other. Publish-and-subscribe messaging is a way of distributing information where topics act as the central conduit. Consumers subscribe to a topic, and producers send messages to it. All subscribed consumers receive all the messages.

Although the concept of messaging domains and the related roles are fairly universal, each programming language or platform has its own set of frameworks and tools to handle them. In Java, messaging is defined by a specification called Java Message Service (JMS). BlazeDS has the ability to plug in adapters to process messages using custom parsers, formatters, and message handlers. By default, it includes adapters for AS3 and JMS. This example does not involve any communication with JMS or Java, but it is restricted to two or more Flex clients, and so utilizes the ActionScript adapter. Here is the configuration snippet from messaging-config.xml, which may explain things a bit further:

```
<service id="message-service"
    class="flex.messaging.services.MessageService"
    messageTypes="flex.messaging.messages.AsyncMessage">
    <adapters>
        <adapter-definition id="actionscript"
        class="flex.messaging.services.messaging.
            adapters.ActionScriptAdapter" default="true"/>
        <adapter-definition id="jms"
            class="flex.messaging.services.➥
messaging.adapters.JMSAdapter"/>
    </adapters>
```

```
        <destination id=" CollaborationTopic">

        <adapter ref="actionscript"/>
        <properties>
            <server>
                <max-cache-size>1000</max-cache-size>
                <message-time-to-live>0</message-time-to-live>
                <durable>true</durable>
                <durable-store-manager>
                    flex.messaging.durability.FileStoreManager
                </durable-store-manager>
            </server>
            <network>
                <session-timeout>0</session-timeout>
                <throttle-inbound policy="ERROR" max-frequency="50"/>
                <throttle-outbound policy="REPLACE" max-frequency="500"/>
            </network>
        </properties>

        <channels>
            <channel ref="samples-rtmp"/>
            <channel ref="samples-amf-polling"/>
        </channels>
        </destination>
    </service>
```

This configuration file is divided into three parts to make it easier for you to understand the details part by part and not get consumed in the complexity of a large configuration file. The three parts are as follows:

- Adapters
- Channels
- Destinations

Configuring a message service is not very different from configuring any other service, such as remoting or data management.

Two types of adapters are defined in this configuration. One is for ActionScript and the other is for JMS. The collaborative form destination uses the ActionScript adapter. Destinations use the channels to communicate, and send and receive messages. Channels are abstractions that bundle a protocol and an endpoint. Besides these, a destination can take a few properties to define its behavior. The preceding snippet has settings for the following properties:

- `Server`: Parameters affecting the storage of messages can be defined here. Maximum cache size, file size, file store root, and durability are typical parameters.
- `Network`: Timeout and inbound and outbound throttle policies can be set here.

This quick example should have established the elegant message features available with data services. Different data services implement message push differently. Protocols vary. LCDS uses RTMP to stream

messages, BlazeDS relies on AMF polling, and GDS uses Java NIO Comet. In all cases, the feature itself is very useful. We built a trivial example, but sophisticated real-time applications can be built on the basis of a reliable message infrastructure.

Before we close this discussion on messaging, we will look at integration with JMS and highlight the performance criteria that may be of paramount importance in production applications. You will also learn how to write a custom message adapter.

Messaging and JMS We will now take our last example, the collaborative form, and transform it into a real-time negotiation application. The negotiation involves two parties, a buyer and a seller. A buyer makes a bid and specifies the preferred price and quantity. A seller makes a bid by stating the desired price and the quantity. The matchmaking part of the negotiation is not included here. It's assumed the two parties go back and forth with a few bids and offers and finally close on a price and quantity that is acceptable to both. To keep the negotiation process neutral and unbiased, information about the last price at which somebody else traded is displayed in real time. We assume the system that collates and emits the last trade price is a Java application and is capable of sending these prices out as JMS messages. In the current context, how we consume this external data within our application is the main agenda. The Flex application consumes JMS messages with the help of data services, which rely on the JMS adapter to receive and send messages to JMS.

The application interface is simple and could look like Figure 7-8 on startup. We are assuming the JMS subscription triggers on creation being complete, and so we have fetched some data already.

Figure 7-8. A view of the negotiation application at startup

The code behind this application is equally simple, and its structure is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp()" >
    <mx:Script>
        <![CDATA[
            import mx.messaging.events.MessageEvent;
```

```
            private function processBid(zip:String, pn:String):void {
                // Process the Bid ?
                //match it with the average last trade and the ask
            }

            private function processAsk(zip:String, pn:String):void {
                // Process the Ask ?
                //match it with the average last trade and the bid
            }

            [Bindable]
            var lastTradeValue:String = "USD 99.99";

            private function messageHandler(event:MessageEvent):void {
                // Get the message and process it
                // set the value of lastTradeValue
                //based on the received message
            }

            private function initApp():void {
                //consumer.subscribe();
            }
        ]]>
    </mx:Script>

    <!-- <mx:Consumer id="consumer"
destination="lastTradePrice"
message="messageHandler(event)" /> -->
    <mx:Panel horizontalAlign="center">

    <mx:Form id="buyerBid" defaultButton="{submitBid}">
        <mx:FormItem label="Bid Quantity">
            <mx:TextInput id="bidQuantity"/>
        </mx:FormItem>
        <mx:FormItem label="Bid Price">
            <mx:TextInput id="bidPrice"/>
        </mx:FormItem>
        <mx:FormItem>
            <mx:Button label="Submit Bid" id="submitBid"
                click="processBid(bidQuantity.text, bidPrice.text);"/>
        </mx:FormItem>
    </mx:Form>

    <mx:Form id="sellerAsk" defaultButton="{submitAsk}">
        <mx:FormItem label="Ask Quantity">
            <mx:TextInput id="askQuantity"/>
        </mx:FormItem>
        <mx:FormItem label="Ask Price">
            <mx:TextInput id="askPrice"/>
```

```
            </mx:FormItem>
            <mx:FormItem>
                <mx:Button label="Submit Ask" id="submitAsk"
                    click="processAsk(askQuantity.text, askPrice.text);"/>
            </mx:FormItem>
        </mx:Form>
        <mx:Label
    text="Last Trade Price: {lastTradeValue}"
    textAlign="center"
    fontWeight="bold" />
        </mx:Panel>
    </mx:Application>
```

The highlight of this example isn't the application functionality but the integration with JMS, so we jump to the configuration file to see how that was made to work smoothly. Here is the snippet of `messaging-config.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<service
id="message-service"
class="flex.messaging.services.MessageService">
    <adapters>
        <adapter-definition
id="actionscript"
class="flex.messaging.services.messaging.➥
adapters.ActionScriptAdapter"
default="true" />
        <adapter-definition
id="jms"
class="flex.messaging.services.➥
messaging.adapters.JMSAdapter"/>

</adapters>
    <default-channels>
<channel ref="my-streaming-amf"/>
<channel ref="my-polling-amf"/>
    </default-channels>
 <destination id="dashboard_chat">
 <properties>
   <server>
<durable>false</durable>
   </server>
   <jms>
     <destination-type>Topic</destination-type>
<message-type>javax.jms.TextMessage</message-type>
<connection-factory>ConnectionFactory</connection-factory>
<destination-jndi-name>topic/testTopic</destination-jndi-name>
<delivery-mode>NON_PERSISTENT</delivery-mode>
```

```
<message-priority>DEFAULT_PRIORITY</message-priority>
<acknowledge-mode>AUTO_ACKNOWLEDGE</acknowledge-mode>
<transacted-sessions>false</transacted-sessions>
  </jms>
 </properties>

<channels>
    <channel ref="my-polling-amf"/>
 </channels>
 <adapter ref="jms"/>
 </destination>

</service>
```

JMS defines extensive possibilities for message durability, acknowledgements, and message filtering. Data services extend most of those features to Flex. The default JMS adapter in BlazeDS and LCDS supports these features.

In the preceding example, we send text messages, but it's also possible to send object messages. Data services remoting works in association with messaging to serialize and deserialize between Java and ActionScript.

JMS in Flex started out by supporting only publish-and-subscribe messaging, but point-to-point communication has now been included. The Flex framework also includes filters and subtopics to allow fine-grained and rule-based message filtering. The `Consumer` control has a `selector` property to filter messages, which takes a string value. SQL expressions can also be defined to filter messages. For example, to filter messages based on `headerProp`, a header property, you could have a criterion like this:

```
headerProp > someNumericalValue
```

Message headers are accessible via a message handle. They are stored as an associative array and can be modified as required. It's also possible to add newer members to this hash. For example:

```
var message:AsyncMessage = new AsyncMessage();
message.headers = new Array();
message.headers["newProp"] = newValue;
```

This was just a part of the code but it confirms that arbitrary numbers of header manipulations are possible with messages. Selector tags operate only on the headers and not on the message body.

An alternative to using selectors is to use subtopics. Whereas a selector evaluates every message through an expression, a subtopic creates subcategories within a destination. The subcategories can be created using specific names. Before sending a message out, a producer sets the subtopic. A small code snippet might look like this:

```
var message:AsyncMessage = new AsyncMessage();
producer.subtopic = "subTopicLevel1.subTopicLevel2.subTopicLevel3 ";
producer.send(message);
```

A consumer sets the subtopic at the time of subscription. A subtopic is set before the `subscribe` method is called. Following is an example code snippet that does this:

```
consumer.destination = "ConfiguredDestination";
consumer.subtopic = " subTopicLevel1.subTopicLevel2.subTopicLevel3";
consumer.subscribe();
```

It's possible to use wildcards with subtopics. For example, there may be multiple subtopics at level 2, and you may want to receive messages that fall in all those subcategories. In that case, instead of subscribing to each of them individually, you could subscribe to `subTopicLevel1.*` and get the same effect.

By now, you know the message service is a useful feature and provides fairly sophisticated ways to build complex applications. You are also familiar with the built-in adapters that work behind the scenes to create robust integration points. Despite these features, the variety of messaging styles and infrastructure requires the ability to write custom adapters to work with scenarios beyond ActionScript and JMS.

The good news is that BlazeDS and LCDS provide a clean and simple API to write custom adapters.

Writing custom message adapters

Writing a custom message adapter involves the following steps:

1. Use the BlazeDS (or LCDS) API to write a custom message adapter.
2. Configure the adapter in `services-config.xml` so that destinations can use it.
3. Refer to this custom adapter in a message destination.
4. Create producers and consumers and send and receive messages using this custom adapter.

I will focus on the first two of these four steps. Also, when I talk about writing the adapter, I will focus on the API and the requirements and not delve into implementation of any specific behavior.

All message adapters in BlazeDS extend directly or indirectly from `flex.messaging.services.ServiceAdapter`. `MessagingAdapter` is the base class for publish-and-subscribe messaging, and it inherits directly from the `ServiceAdapter`. Depending on the desired behavior, and therefore potential reusability of `MessagingAdapter` implementation, you can choose to start with either the `ServiceAdapter` or the `MessagingAdapter`.

All adapters are required to implement the `invoke` method, which is invoked on receipt of a message. The `invoke` method is expected to take a `Message` object and return an `Object` type. It's common to use the `AsyncMessage` class as the class that implements the `Message` interface.

In the ActionScript adapter, the `invoke` method sends the message out to all subscribers by calling the `pushMessageToClients` method. This is the custom behavior that you need to implement depending on what you expect from the adapter.

The `MessagingAdapter` class also defines a few other methods, including those that help initialize it, allow subtopics, and set security constraints. The `initialize` method takes two arguments: a string ID and a `ConfigMap` object. What you define in the `services-config.xml` or its associated configuration files becomes available via the `ConfigMap` object. You have seen JMS adapter–specific property

settings in the last example. You can similarly set properties to your custom adapter. For example, you may call your custom adapter `customAdapter` and have its properties within the `<customAdapter>` `</customAdapter>` XML tags. The `initialize` method will be able to get all the properties from such an XML node and will be able to initialize the adapter as you instruct it to.

Writing a custom adapter is a large topic and could fill up an entire chapter, if not a small book; however, I have walked you through the essentials and will stop at that. Before moving on, I would like to mention an open source initiative called dsadapters that I launched a few months back to provide adapters for many common messaging situations and platforms. The project is online at `http://code.google.com/p/dsadapters/`. You might find your desired custom adapter in this project and thus save yourself some time and energy.

Advanced issues in messaging

This section picks up an assortment of topics that pertain to advanced messaging concepts. In none of these do I get under the hood. The text only briefly surveys the topics and gives you a preview of some ideas that lie ahead.

Pushing over sockets

As discussed in the last chapter, it's possible to create socket connections over TCP/IP to an external host and port as long as the security restrictions are satisfied. If the messaging entities (i.e., producers and consumers) are available over a unique network address, creating the connection would be effortless. Socket connections can be of two types: those that transmit text and XML data, and those that transmit binary data. It may be possible to use sockets with binary data to push data efficiently. However, there are drawbacks to this approach, namely the following:

- Non-HTTP ports could have restrictions across a firewall.
- The scalability is seriously suspect, as each logical connection maps to a physical connection.

Therefore, though sockets may be a possibility, it is probably wiser to stick with data services.

Connection scalability

LCDS uses RTMP to push data, whereas BlazeDS uses AMF polling to send the data through to the client. In AMF polling, data push, though possible, is not scalable. It has limitations because of inefficiencies in the mechanism. BlazeDS does not have RTMP and probably never will, unless RTMP becomes open source.

In the meanwhile, you could use Comet-style persistent connections to push data. The Servlet 3.0 specification is trying to come up with a uniform standard to create HTTP 1.1–style persistent connections for data push using the NIO-based framework Comet. However, even before the specification is ready and accepted, the Apache foundation has already implemented a way to do this in Apache Tomcat. The Jetty team has achieved similar success.

For BlazeDS to use this scalable option, the message broker servlet needs to be modified to listen to Comet events. That way a blocked long-polling connection can be created with no threads being utilized on the server. Such connections can easily scale and have up to 30,000 or more connections on a single 64-bit machine.

Transactions

Message-based systems can support transactions and allow for explicit commit and rollback. With JMS, it's possible to have transactions in a messaging session or have more robust arrangements with the help of the Java Transaction API (JTA). In data services such as LCDS and BlazeDS, it's possible to use a JMS **transacted session**. A transacted session supports transactions within a session. Therefore, a roll-back in a transacted session will roll back all the sends and receives in that session. However, it will have no impact on transactions that are outside of the session. In other words, if the JMS messages interacting with Flex also affect a transaction in another enterprise application or the database, a roll-back will only impact the JMS session and not these external systems.

Turning this feature on is as simple as setting the `transacted-sessions` value in the configuration file to `true`. This is what it looks like:

```
<jms>
....
    <transacted-sessions>true</transacted-sessions>
....
</jms>
```

Using server-side proxies

`HTTPService` and `WebService` were covered in enough detail in the last chapter. There we emphasized that these services promote loose coupling and can be used to access data in a Flash Player independent of a server infrastructure. The only mandatory requirement was that the external host provide a cross-domain security policy definition. This requirement was never a concern if the host already had a `crossdomain.xml` file allowing access or if it was possible to request the host maintainers to put one in place. However, in the vast expanse of the World Wide Web, it's not always possible for such a good arrangement to be in place. In situations where we are unable to access data from an external host due to security restrictions, it's viable to fetch it via data services. In this role, data services provide the proxy service for `HTTPService` and `WebService` components.

The server-side configurations for proxy settings are made in the `proxy-config.xml` file. This file is included by reference in `services-config.xml`. The `HTTPService` sends HTTP requests down to the proxy, and the `WebService` sends SOAP web service calls down to the proxy. In either case, we need a service adapter to take these requests and translate them into the final call. For example, an HTTP request needs to end up with a URL invocation. You saw messaging service adapters in the context of messaging. The service adapters for HTTP and web service proxies implement a similar set of classes to create a service adapter. Default HTTP proxy and SOAP web service proxy adapters are available in BlazeDS.

For `HTTPService`, you can define a URL for the proxy setting or set up a set of dynamic URL(s) that are resolved to the appropriate URL based on the URL value set in the client-side HTTP call. For web services, you define either the WSDL URL or the SOAP endpoint URL pattern. These URL configurations are done with destination configuration. It's also possible, especially with `HTTPService`, to define a default destination and have a set of dynamic URLs with it. Then every `HTTPService` call via data services is routed through this destination to the HTTP proxy adapter.

The proxy service itself can be configured with a number of properties. We present a sample of the configuration file available in the distribution and in the documentation and explain the properties in context. Here is the sample configuration:

```
<service
id="proxy-service"
class="flex.messaging.services.HTTPProxyService">
    <properties>
        <connection-manager>
            <max-total-connections>100</max-total-connections>
            <default-max-connections-per-host>
2</default-max-connections-per-host>
</connection-manager>
<!-- Allow self-signed certificates;
should not be used in production -->
        <allow-lax-ssl>true</allow-lax-ssl>
        <external-proxy>
            <server>10.10.10.10</server>
            <port>3128</port>
            <nt-domain>mycompany</nt-domain>
            <username>flex</username>
            <password>flex</password>
            </external-proxy>
</properties>
</service>
```

The HTTP proxy adapter in BlazeDS uses the Apache `HttpClient` as the user agent. The configuration `max-total-connections` translates to the use of a multithreaded concurrent connections manager for `HttpClient`. `max-connections-per-host` sets the default number of connections if the host supports hardware clustering. The `allow-lax-ssl true` value means self-signed certificates will work. If the connection to the host is made through an external proxy, the external proxy can be specified in the configuration file as well. Authentication credentials like the password can be passed to the external proxy.

Additional useful data services tips

Data services are useful and extensible pieces of software. You have seen how they can be extended to support additional features with the help of custom factories and custom service adapters. What I discuss next are the interesting features around run-time configuration and application security that data services offer.

Configuring at run time

So far, almost all the configurations I have spoken about have related to compile-time configuration, where the entries are made in the configuration files. However, it's also possible to make many of these configurations and settings at run time. Run-time configuration makes systems more flexible and amenable to tweaking at the time of use. At run time, you can define channels, create consumers, set up subscriptions, and affect destination configurations. However, I only show you one of these possibilities here: channel definitions.

Defining channels at run time At run time, channels can be created on the client with the help of a ChannelSet object, which may contain one or more Channel objects. Essentially, the process is first to create a ChannelSet object and then dynamically create channels and add channels to it. After this, the channel set is associated with the channelSet property of the RemoteObject. This is what a sample piece of code may look like:

```
var cs:ChannelSet = new ChannelSet();
var newChannel:Channel = new AMFChannel("my-amf", endpointUrl);
cs.addChannel(newChannel);
remoteObject.channelSet = cs;
```

The endpoint URL can be defined dynamically at run time. ChannelSet has the ability to search among the set of configured channels. Each channel can define a failover URL.

Application security

In many cases, especially in enterprise scenarios, you may need to restrict access to server-side destinations. It's possible to define a secure destination without much trouble. Data service configurations allow definitions for both authentication and authorization. **Authentication** means confirming one's identity, and **authorization** relates to the server-side resources that an authenticated user can access.

Security is configured using security constraints. These constraints can be defined at multiple levels, namely the following:

- **Global**: One set of definitions for all destinations. Usually such configuration would reside in the common configuration area: within services-config.xml itself.
- **Destination specific**: Security constraints for a destination. You can define such constraints inline within the destination configuration tags.
- **Fine-grained**: For remoting destinations, you could create an include list by using multiple include-method tags (or using the exclude-method tag). Such lists will ensure that only the included methods are callable on the remote destinations. Calling any other method would cause an error.

Authentication mechanisms can be custom or basic. This implies that you could leverage your existing authentication systems using the custom route.

Following is an example of a destination-level configuration that uses custom authentication:

```
<destination id="ro">
    <security>
        <security-constraint>
            <auth-method>Custom</auth-method>
            <roles>
                <role>roUser</role>
            </roles>
        </security-constraint>
    </security>
</destination>
```

HTTPService, WebService, and RemoteObject support passing and invalidation of login credentials using the setCredentials and the logout methods, respectively. You can also pass credentials to remote services using setRemoteCredentials.

A simple example of setCredentials with RemoteObject is as follows:

```
var myRemoteObject:RemoteObject = new RemoteObject();
myRemoteObject.destination = "SecureDestination";
myRemoteObject.setCredentials("userName", "myPassword");
myRemoteObject.send({param1: 'param1Value'});
```

Implementing custom security and setting up fine-grained access control can be tedious, but with BlazeDS you can utilize such constructs without adding any additional overhead.

Leveraging the media server

So far, it's been all about data services. Data pull and push are critical for most modern applications. Equally important and becoming ever more popular is the availability of information as media assets, audio and video. The Flash Media Server suite of products is a solution for large-volume on-demand media streaming. In this section, I introduce you to the fundamentals of this product and whet your appetite to seek more information.

Flash Media Server (FMS) is a licensed product, but a free developer edition can be downloaded from the Adobe web site. The latest stable release version at the time of writing is version 3. Red5 is an open source alternative to this product.

Although FMS can be accessed from multiple venues—the Web (using the Flash Player), desktops (within AIR), and mobile devices (using FlashLite)—we will focus on getting it from the Web. An application deployed in the Flash Player (i.e., as a SWF) and built using Flex or Flash can use FMS to stream media, such as a long, high-quality video, within its context. Before we put such an arrangement to the test, it's logical to question the importance and need for such an infrastructure. Videos can be played and downloaded from Flash applications without using FMS. Video player components and controls within Flex can help load, run, and display video content with ease and elegance. Videos can be dropped into a web server and accessed via a Flex application without any problem. Why then should you use FMS?

The question is valid and the answer lies in understanding the different approaches to media delivery to a Flash Player over the Internet. The situations talked about so far are all related to the type of media delivery known as progressive download. In **progressive download**, the media file itself is external to the SWF. (Only for really small media files may it make sense to embed the media file. The SWF would be terribly bloated if large media files were embedded, and your application would take forever to download and start up.)

When the media file, say a video, is accessed, it is downloaded locally to the user agent machine, and when it buffers up a sufficient amount, it is played back. The media resource is transmitted over the regular HTTP channel on port 80, just like any web page or other web resource. In contrast to this download and playback model, there is also the model of streaming bits through a persistent connection and

playing it as soon as it comes through. This type of model is called **streaming**. Files are kept external to the SWF in this case as well, so the advantages of this are available with streaming just as with progressive download. In addition, streaming provides a few additional benefits, namely the following:

- Adjustment of transmission bitrate depending on network bandwidth availability
- Fast start as you don't have to wait for the download to complete
- Advanced control over the media portions and playback
- Ability to seek and navigate to a portion of the media
- Capture and play live as events occur
- Include interactive features
- Enhance security and enforce digital media rights

FMS makes streaming possible and uses it as the de facto mechanism to deliver media. This should answer the question why you should consider using FMS.

To achieve streaming, FMS uses RTMP as the underlying protocol. LCDS uses the same protocol for data push as part of the messaging infrastructure. RTMP is replaced with AMF polling in BlazeDS for data push because RTMP is a proprietary protocol and BlazeDS is an open source product.

FMS supports a variety of variations of the RTMP protocol, as summarized in Table 7-2.

Table 7-2. RTMP Flavors in Flash Media Server

Protocol Name	Description	Port Used
RTMP	Default RTMP	1935 (default, then 443, 80)
RTMPT	Tunneled over HTTP	80
RTMPS	RTMP over SSL	443
RTMPE	Encrypted RTMP (better than SSL)	1935 (then 443, 80)
RTMPTE	Encrypted RTMP tunneled over HTTP	80

FMS supports a lot of different media formats, but I am not going to list them here. On the Flash Player, FLV is the most popular and best supported media format for videos. You can use FLV in FMS.

Now that we have this supercharged infrastructure, let's see how we could use it with a Flex application. A Flex application is delivered as a SWF to a client machine user agent (a browser). This SWF file has the ability to connect to an external FMS, running on an external host, using RTMP. FMS is accessed pretty much the way Flex applications access external data from other hosts. Unlike regular data access, the protocol isn't HTTP, and the connections are persistent in the case of FMS.

A simple example is included to demonstrate the following:

1. Connect from Flex to FMS.
2. Make a remote method call over a net connection.
3. Get a video stream input.

The assumption is that you have the Flash Media Server installed and running. If not—if you just want to play with it—you can download a developer edition of the Flash Media Server and install it.

The first thing to do to interact with FMS from Flex is to connect to it. When you connect to an FMS3 server, you connect to an instance of the server. Therefore, the connection URL will point to such an instance. This is what the connection code might look like:

```
var nc:NetConnection = new NetConnection();
nc.addEventListener(NetStatusEvent.NET_STATUS, netStatus);
nc.addEventListener(SecurityErrorEvent.➥
SECURITY_ERROR, netSecurityError);
nc.connect("rtmp://localhost/MyApplication");
```

`NetConnection` is the primary class used to connect to FMS. When connecting from Flex, you use the AS3 version of the class, which adopts the event-driven model like the connection classes you have seen before.

Now that the connection is established, we will make a remote call to a server resource. This server resource is the media resource that we will be streamed across to our Flex application. We use the `NetStream` class to consume it and display it in our Flex application.

I will ignore most of the manipulation code on the Flex side here and just show small pieces of the call and the `NetStream` code, the intent being to simply introduce FMS.

Typically, a call to the server is made like this:

```
nc.call("serverMethod", MyResponder, "MethodParameter");
```

If the call returns a video stream, we can read the stream using the `NetStream` class. Then we can bind the video to a `UIComponent`, which we can add to the stage.

Programming FMS with Flex does not involve fundamentally different paradigms. It's all ActionScript programming of the kind you do with your other applications. The only new element is the API, with which you'll want to familiarize yourself. As with all Adobe products, the Flash Media Server 3 developer guide (LiveDocs) is the best place to go to get that information. It's available online at `http://livedocs.adobe.com/flashmediaserver/3.0/hpdocs/help.html?content=Book_Part_31_deving_1.html`.

Summary

This chapter rapidly covered a fair number of topics related to data services and the media server, with the focus more on the former. The introduction to media server was meant to be a high-level overview.

The chapter started with an overview of the data services architecture. Then you explored the steps involved in installing data services and configuring it. Subsequently you built an example application and saw data services in action.

The review of the features of data services topics included Java to Flex serialization, asynchronous communication, support for additional data types, custom adapters, connection scalability, data push over sockets, and transactions. Server-side proxy and its usage for HTTP-based calls as well as web services were also illustrated.

Both data services and media servers position the Adobe Flex and AIR technologies as viable choices for some serious applications for the present and the future. In the age of event-driven, real-time, responsive rich systems, where the Web is transforming itself into a read-write media network, technologies as these are bound to shine.

In the next chapter, we deal with Flex and PHP integration. Although PHP does not offer the same type of data services as Java does, the remoting possibilities from Flex to PHP are many. Fundamental concepts that relate to RPC and the media server, which you learned in this chapter, will be useful when dealing with analogous concepts in the world of PHP as well.