Learning Flash Media Server 3
by William B. Sanders

Publisher: O'Reilly
Pub Date: March 31, 2008
Print ISBN-13: 978-0-596-51590-4
Pages: 273

Table of Contents

Overview

If you're interested in recording and streaming media using Flash Media Server 3 (FMS3) and Adobe's Real-Time Messaging Protocol, this unique 267-page PDF-only book is the perfect primer. It is not a reference, but a systematic guide to developing FMS3 applications using ActionScript 3.0, with chapters that focus on specific aspects of the server and how they work. FMS3 is very different from regular web servers. Because its open-socket server technology stays connected until users quit the application, you can stream audio, video, text, and other media in real time. FMS3 is also quite different from previous versions, a fact that web developers familiar with Flash Media Server 2 or Flash Communication Server 1.5 will quickly discover. Don't worry. With *Learning Flash Media Server 3* and a little experience with Flash CS3 and ActionScript 3.0, anyone can get up to speed in no time. You'll learn how to install FMS3, organize your development environment with Apache web server, and use the management console before diving into the whys and hows of:

- Recording and playing back streaming audio and video in VP6 and H.264 formats
- Using the new Flash Media Encoder to stream and record video
- Camera and microphone settings
- Non-persistent client-side remote shared objects
- Two-way audio-video communications
- Broadcasting and server-side bandwidth control
- Working with server-side files: the file class
- Server-side shared objects
- Server-side streams
- Setting up a software load handler using FMS3's new server-side NetStream
- Bringing in data and working with configuration files

At the heart of every chapter is a core set of code that shows the minimum requirements needed for different procedures. Beyond that, *Learning Flash Media Server 3* provides you with plenty of options for using FMS3's different versions -- the full-feature server, the streaming-only server, and the limited-user development server. It's a whole new world of media, and this book puts you right at the doorstep. Ready to enter?

Learning Flash Media Server 3
by William B. Sanders

Publisher: O'Reilly
Pub Date: March 31, 2008
Print ISBN-13: 978-0-596-51590-4
Pages: 273

Table of Contents

Copyright

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://safari.oreilly.com). For more information, contact our corporate/institutional sales department: (800) 998-9938 or corporate@oreilly.com.

Editor: Steve Weiss

Editor: Audrey Doyle

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. Learning Flash Media Server, the image of a pearl-spotted barbet, and related trade dress are trademarks of O'Reilly Media, Inc.

Chapter 1. Getting Started with Flash Media Server 3

1.1. The New Flavors for Flash Media Server 3

The release of Flash Media Server 3 (FMS3) comes in three different server configurations. First is Flash Media Interactive Server 3 (FMIS3), with full features for interactive streaming, and incorporating features of the previous Origin and Edge editions. Similar to the full Flash Media Server 2 version, FMIS3 has more features: This new edition does not restrict the number of concurrent connections or bandwidth usage.

The second version of FMS3 is the Flash Media Streaming Server 3 (FMSS3). This server is limited to streaming video and audio. Think of it as similar to progressive downloading, except FMSS3 uses server streaming. If your main goal is to stream audio and video from a site, this option provides a much lower cost server than FMIS3, with no limits on concurrent users or bandwidth. Using the Flash Media Encoder 2, you can use FMSS3 for live video streaming of either H.264 or On2 VP6 video.

The third version of FMS3 is the free Flash Media Development Server 3 (FMDS3). With the same features as the FMIS3 server, this version is limited to 10 concurrent users and cannot be used in a production environment. You can use it to develop applications that will run either on your own computer or a local area network (LAN), for either the FMIS3 or FMSS3 servers.

To avoid confusion, this book refers to all three versions simply as FMS3. You will need to keep in mind that the FMSS3 server cannot be used for recording video or interactive chats, and has other limitations. However, using the Development version of the server, you can create and test all of the examples in this book. You can even record your own FLV video files and play them on either of the other two servers.

Chapter 1. Getting Started with Flash Media Server 3

1.1. The New Flavors for Flash Media Server 3

The release of Flash Media Server 3 (FMS3) comes in three different server configurations. First is Flash Media Interactive Server 3 (FMIS3), with full features for interactive streaming, and incorporating features of the previous Origin and Edge editions. Similar to the full Flash Media Server 2 version, FMIS3 has more features: This new edition does not restrict the number of concurrent connections or bandwidth usage.

The second version of FMS3 is the Flash Media Streaming Server 3 (FMSS3). This server is limited to streaming video and audio. Think of it as similar to progressive downloading, except FMSS3 uses server streaming. If your main goal is to stream audio and video from a site, this option provides a much lower cost server than FMIS3, with no limits on concurrent users or bandwidth. Using the Flash Media Encoder 2, you can use FMSS3 for live video streaming of either H.264 or On2 VP6 video.

The third version of FMS3 is the free Flash Media Development Server 3 (FMDS3). With the same features as the FMIS3 server, this version is limited to 10 concurrent users and cannot be used in a production environment. You can use it to develop applications that will run either on your own computer or a local area network (LAN), for either the FMIS3 or FMSS3 servers.

To avoid confusion, this book refers to all three versions simply as FMS3. You will need to keep in mind that the FMSS3 server cannot be used for recording video or interactive chats, and has other limitations. However, using the Development version of the server, you can create and test all of the examples in this book. You can even record your own FLV video files and play them on either of the other two servers.

1.2. What Is a Media Server?

If you've used Flash Communication Server or Flash Media Server 2 and understand its use, you can skip this section. Flash Media Server 3 is the latest version of Flash Media Server and has several improvements. However, if you're new to working with open socket technology, including FMS3, read on.

FMS3 is an open socket server. The key difference between open socket servers and Web servers is that as soon as you receive information from a Web server, the connection is broken. It may look as if you're still connected to the Web server, especially with a Flash page that's animating materials. However, that's not the way it works. If you open a Web page, the Web server sends you the page along with all associated graphics, text, and other media; and your computer sends a message back that says, "Got it!" (or something to that effect), and the connection closes. With an open socket server, the connection stays open until you quit the application or trigger an event that cuts the connection. Because the connection remains open, you can stream audio, video, text, and any other media available on the Internet, in real time. You just can't do that with a regular Web server because it has an entirely different architecture.

1.2.1. Enter RTMP

When you use a regular Web page, you're most likely using HTTP (Hypertext Transfer Protocol), which allows you to look at Web pages. To be able to work with streaming media, Adobe developed Real-Time Messaging Protocol (RTMP). Generally, when you use FMS3, you first connect to a Web server via HTTP and then to Flash Media Server using RTMP. Because of this arrangement, you're working simultaneously with different protocols-one for the Web site and the other for the streaming media.

1.2.2. Special Languages and Documentation

To deal with all the things you can do with FMS3, you have two additional language Application Program Interfaces (APIs). Both APIs are extensions of ActionScript and are called Client-Side Media ActionScript and Server-Side Media ActionScript, or in this book, simply as CS ActionScript and SS ActionScript. With ActionScript 3.0, which is also used in this book, the client-side ActionScript has been fully integrated into the language. As a result, the documentation no longer comes with a special client-side ActionScript documentation file. Instead, the entire ActionScript 3.0 language reference is included. The *Server-Side ActionScript Language Reference* is still included, however. The full content of these APIs is richly described in PDF files that come with FMS3. (When you install FMS3, these files are installed on your computer in `Program Files  Adobe  Flash Media Server 3  documentation`.) Once you've installed FMS3, you should immediately print this documentation and put it in loose-leaf binders, or at least put the files in a folder on your desktop so that they're easy to look up. You'll be using the documentation all the time. The FMS3 documentation is general and complete; in contrast, this book is not a reference. Rather this book is designed to get beginners started, with a series of specialized chapters that show how to use different features of FMS3.

1.3. Installing FMS3

You can install FMS3 on either a Windows or Linux OS. For a Windows server, you need at least Windows 2000, which means you can install it on your Windows Home, XP, or Vista edition. If you're installing it on your company's server, you'll probably be using Windows 2003 Standard Edition, Windows 2005 Server, or the new Windows 2008 Server. For your Linux box, either Red Hat Enterprise version 3 or 4 will do the trick. You'll need a 3.2 GHz or faster Pentium 4 (or some other X86-compatible processor), 1 GB of RAM, and 50 MB of hard drive space. For this book, I used a 3 GHz Pentium 4 with 1 GB of RAM running on Windows XP, Windows Vista on another server, and Red Hat version 4 on a Linux server.

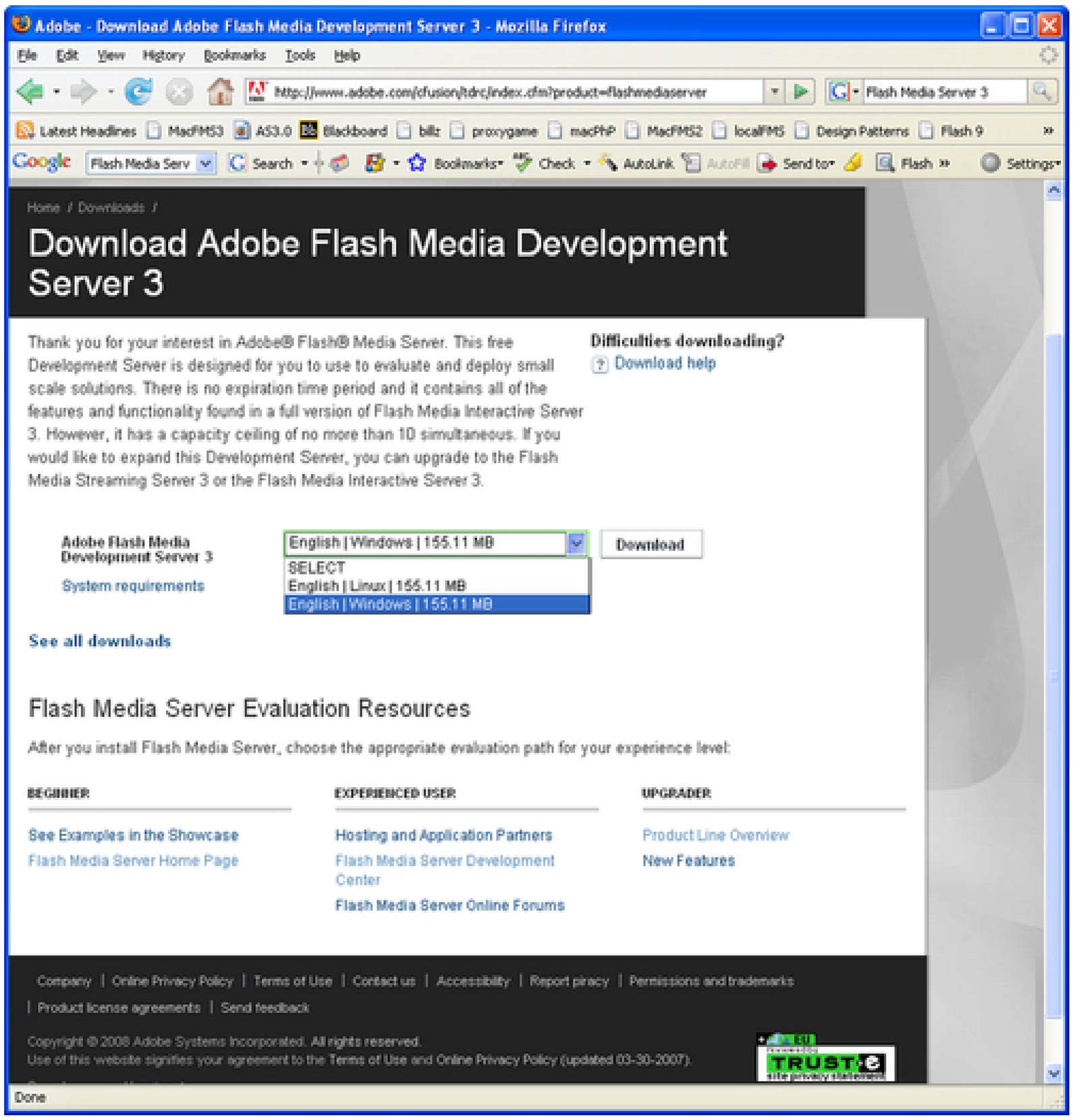
> NOTE
>
> You cannot install FMS3 on Mac OS or Apple OS server. However, you can develop FMS3 applications on a Macintosh and install FMS3 on the "MacTel" version of a Macintosh running Windows. If you run both Windows and Mac OS X at the same time on your Macintosh-using Parallels-you can run the Flash application on the Macintosh side and FMS on the PC side. Also, you can install FMS3 on a Linux server, but you cannot develop applications on a Linux machine. (Adobe doesn't make a Linux version of Flash.) This apparent paradox recognizes that the two main server technologies are Windows and Linux, and the two main development platforms are Windows and Macintosh.

As you will soon see, this doesn't mean that you cannot develop and test FMS3 apps on your Macintosh. Rather, it means that you'll just have to install FMS3 on a remote server (which would include your Windows computer sitting on the same desktop or even in your lap). The big advantage of running your application on one computer and FMS3 on another is that your processor doesn't have to concentrate on two things at once. So if you've got a Macintosh, don't worry. Most of the applications developed for this book were done using a Macintosh with the FMS3 server connected via a LAN to a PC running Windows XP. Of course, if you have a newer Macintosh with the Intel processor, you can run them both on the same machine, as long as Mac OS and Windows OS are running in parallel.

1.3.1. Installation Steps for Windows

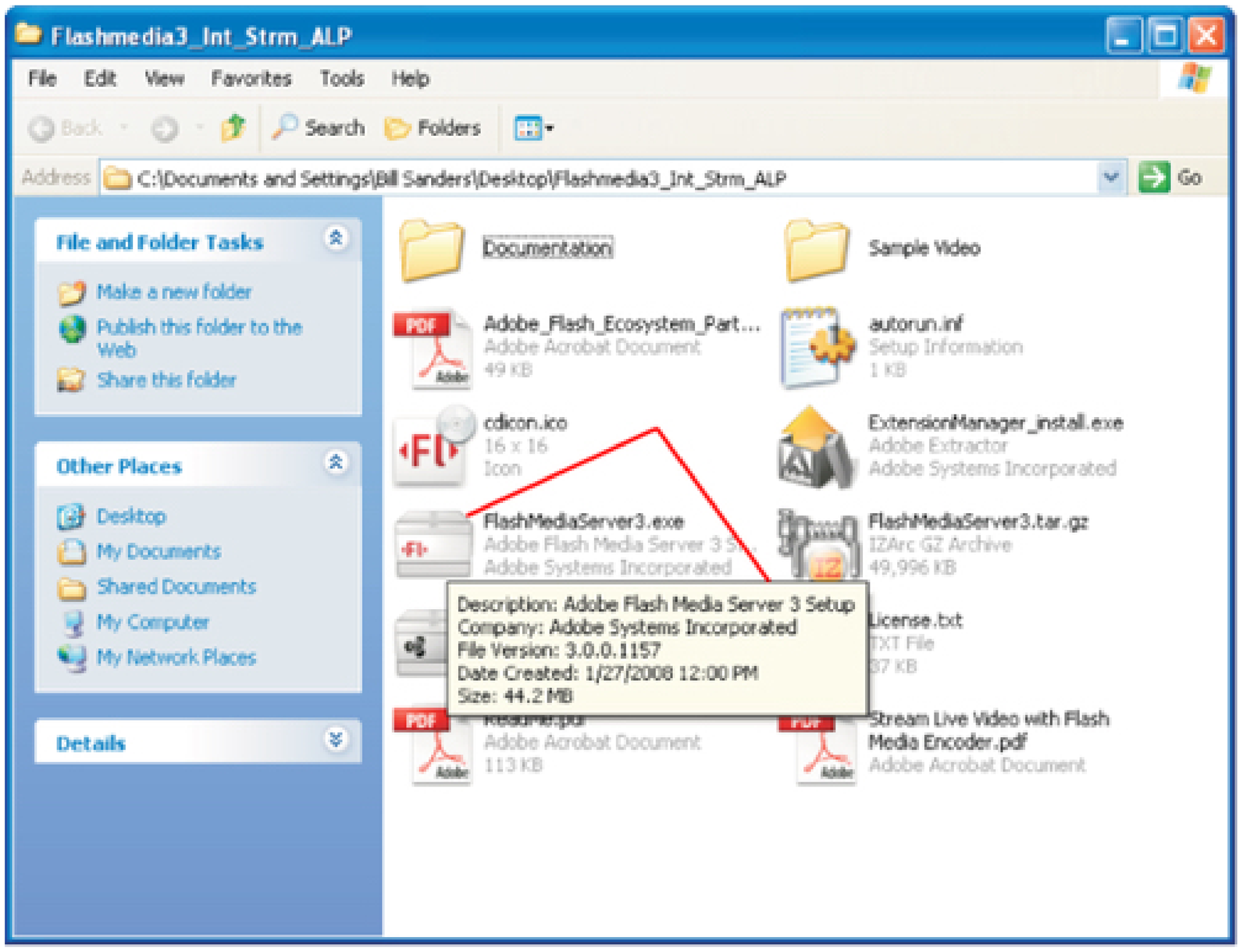
Installing FMS3 is very easy on either Windows OS or Linux. First, you'll look at an example of how to install FMS3 on a Windows XP computer using the Development Version freely available online at www.adobe.com/products/flashmediaserver. Figure 1-1 shows the download page. Be sure to select the correct version for your server (or computer).

Figure 1-1. Selecting FMS3 version



Once downloaded, uncompress the file and open the Flashmedia3_Int_Strm_ALP folder on your desktop or somewhere equally convenient, and double-click FlashMediaServer3.exe, as shown in Figure 1-2.

Figure 1-2. Setup file in FMS3 folder

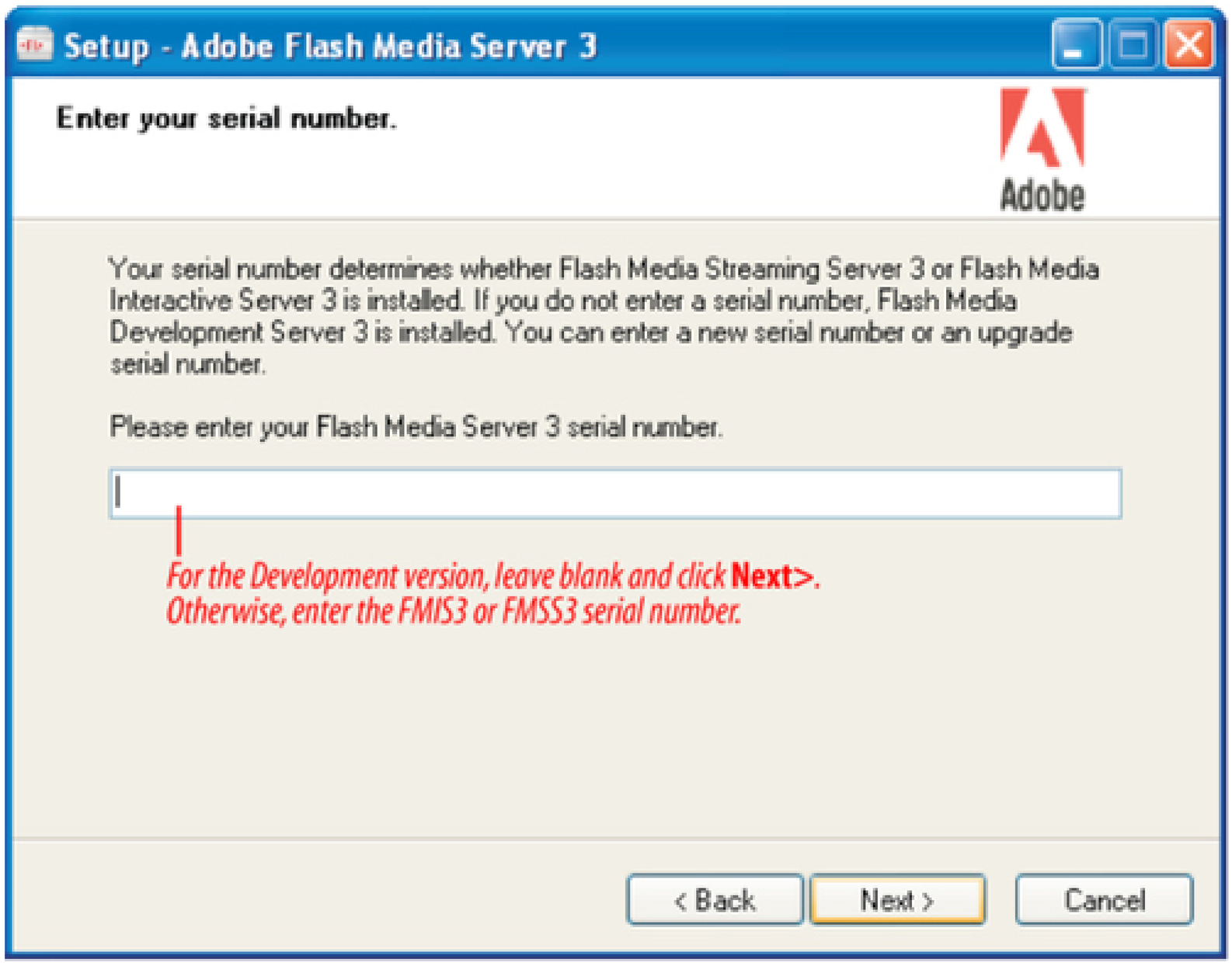


As soon as you start the setup process, the Flash Media Server 3 Setup Wizard appears, as shown in Figure 1-3.

Figure 1-3. FMS3 Setup Wizard

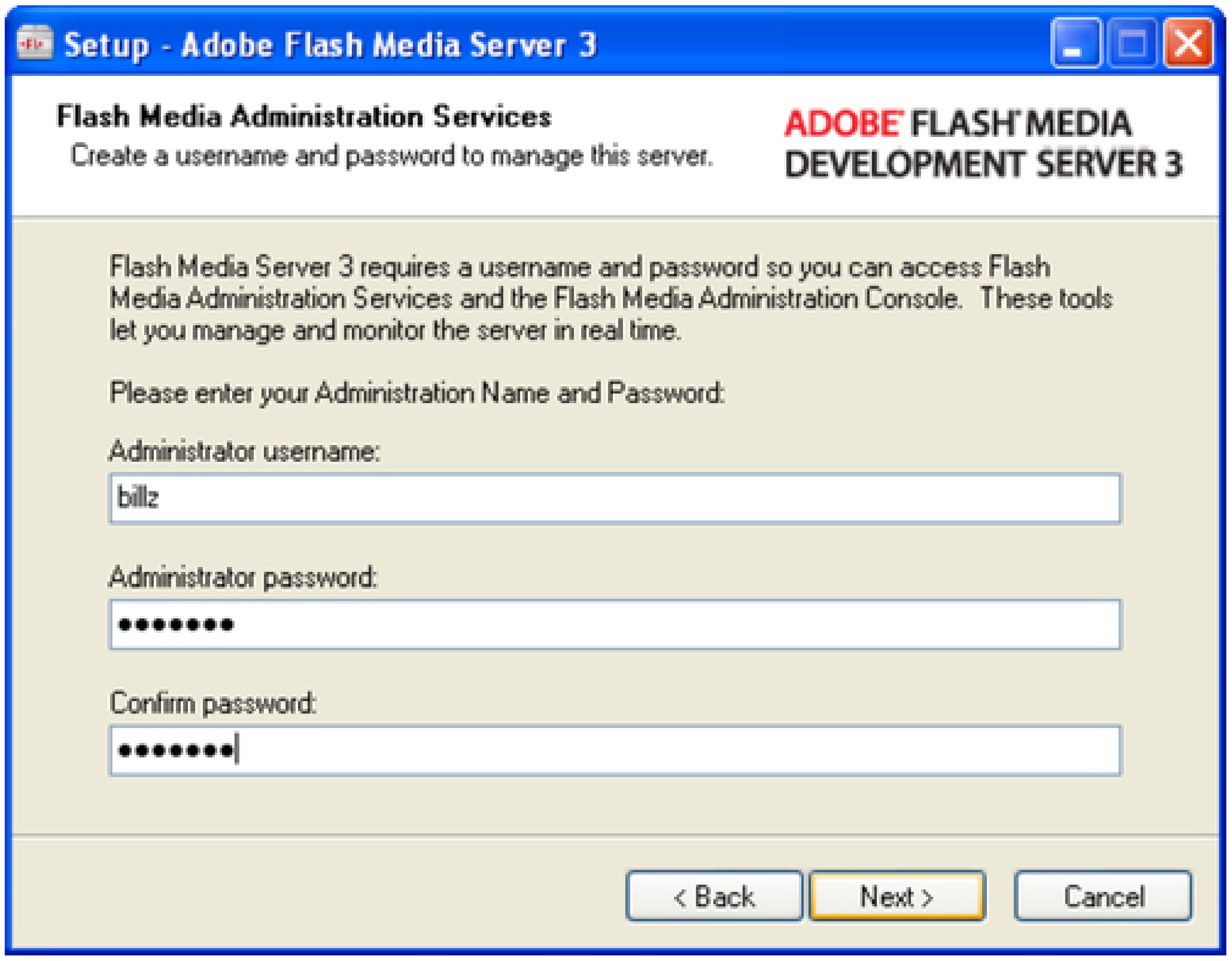
For the next step, you'll need your serial number, if you have one; keep it handy. Then click the Next button. The screen shown in Figure 1-4 appears, requesting your serial number.

Figure 1-4. Serial Number entry box



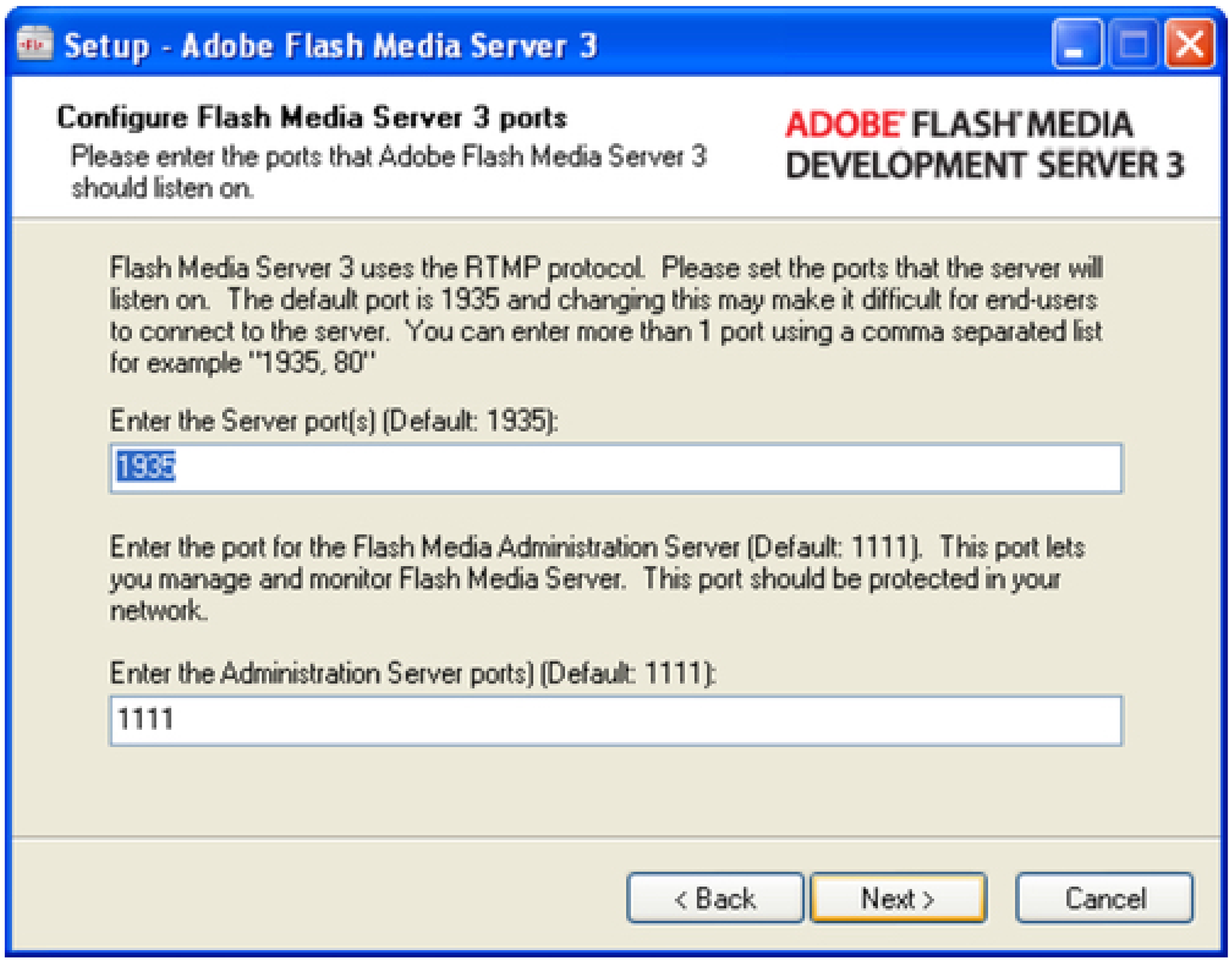
Enter a serial number to have that number determine whether the server is configured as FMIS3 or FMSS3; if you do not enter a serial number, the server will be configured as the FMDS3 (Development) server. Press the Next button. Figure 1-5 shows the page that appears requesting a user name and password for the module you will need to access for the Flash Media Administration Services:

Figure 1-5. Administrator user name and password entry



This page is crucial because you have to remember both the administrator name and password-write them down. The name and password are both case-sensitive; so don't get fancy. The Flash Media Administration Console lets you see what's going on with your application on the server, and you need to get it right. When you've entered your administrator name and password, click Next to open the window to establish the port settings for both the Flash Media Server and the Flash Media Administration console. Figure 1-6 shows the window with the default settings:

Figure 1-6. Setting ports



The port configuration is generally straightforward. Use the default ports shown in Figure 1-3. By default, FMS3 is configured to listen to Port 1935; unless you have a very good reason, leave it at that. (One good reason to change your default port is if your system already has an application that listens to Port 1935.) However, you can include additional ports to listen to. For example, you can list:

```
1935,80,443
```

This means you can have a sequence of ports to listen to if you're using more than the RTMPE (or RTMP) protocols, such as RTMPT (tunnel via HTTP) or RTMPS (tunnel via HTTPS).

Likewise, the administrative service (FMS3 console) listens to port 1111. That's a different service, and you don't want it to have the same port as FMS3.

As you develop more complex applications in FMS3, you may want to reconfigure your ports. Check out the Flash Media Server 3 TechNotes at the Adobe Web site for more information.

To keep it simple, for now just use the defaults and click Next.

The last page that appears before installation has all of the details of the key directories and the port values. Write these down and keep them where you can easily find them. Figure 1-7 shows what the summary includes:

Figure 1-7. Final check before installation

Once you've verified that the settings are satisfactory, click Install.

Now you should see the installer grinding away to install FMS3. When it has successfully completed its tasks, the screen shown in Figure 1-8 appears. You can change the defaults here if you don't want to see the ReadMe.pdf notice, start the server, or have the server start automatically every time you reboot or turn on your computer. Starting and letting the server run all the time won't be a chore for your processor when there's nothing for it to serve. So, if you leave the default configuration, you'll be all set to develop FMS3 applications every time your computer is on.

Figure 1-8. Completing Setup Wizard screen

Once you have everything installed for Flash Media Server 3 on your computer, you can look in Start All Programs Adobe Flash Media Server 3. There you will see selections of the documentation, the Flash Media Administration Console, starting and stopping both the media and administration servers and even an uninstaller. Figure 1-9 shows what appears if your installation succeeded:

Figure 1-9. Flash Media Server 3 selections

1.3.2. Installation Steps for Linux

Installing FMS3 on Linux is pretty easy as well. Basically, all you need to do is to extract the files from the compressed archive (FlashMediaServer3.tar.gz) and then untar the installation file. I extracted the file using the graphic user interface.

Unpacking the install package creates a directory and places the package within it. Even though you can see the icons that make up the package, you have to go into the Terminal to install the software. From the Terminal, navigate to the new directory, in which you will see a file named installFMS.

Type:

```
./installFMS
```

in the Terminal to install your server. You must enter and remember a unique Administrative name and password. You can simply use the default values for the server port (1935) and admin service (1111), and everything should work as expected.

1.4. Organizing Your Development Environment

If you follow the default installation, your Flash Media Server 3 should start up every time you turn on your computer. Within this environment you need to set up your work environment so that you can conveniently test your applications.

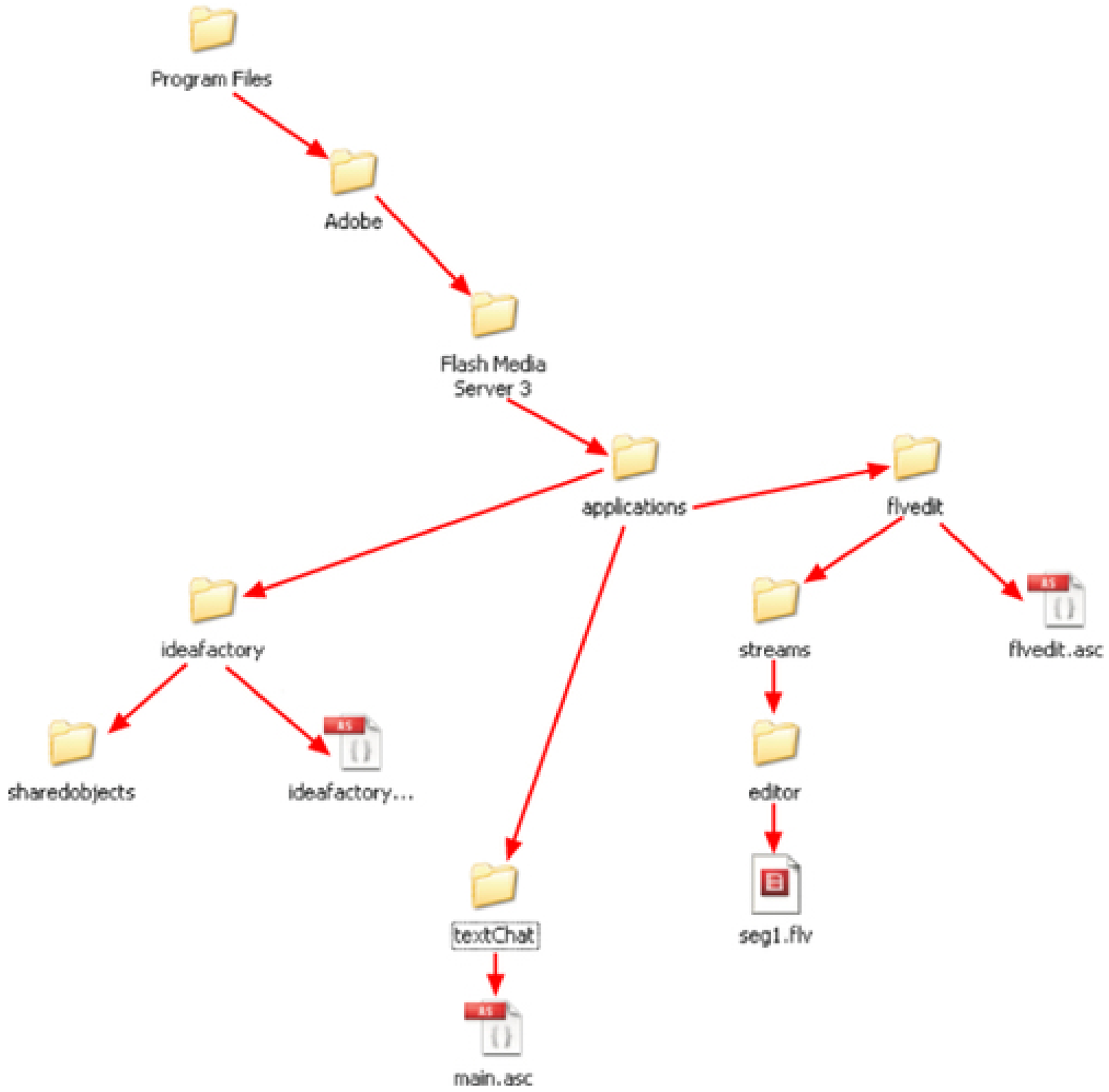
1.4.1. Server-Side Files

When you install FMS3 on Windows, your typical installation is going to set up your server-side files with the following path:

```
C:\Program Files\Adobe \Flash Media Server 3\applications
```

All of your server-side application folders go into the folder named *applications*. For example, Figure 1-10 shows a visual view of the paths with three applications in the applications folder named ideafactory, flvedit, and textChat. The application folders must be named with the application name. That is, if your application is named "vidChat", you must name the folder "vidChat." The folder can be empty or contain other files. If you have written server-side code for your application, the file must be named main.asc or the same name as your application, such as vidChat.asc. Throughout this book I use the same name as the applications for the ASC files to reduce confusion when discussing more than one application with server-side code.

Figure 1-10. Path to server-side applications

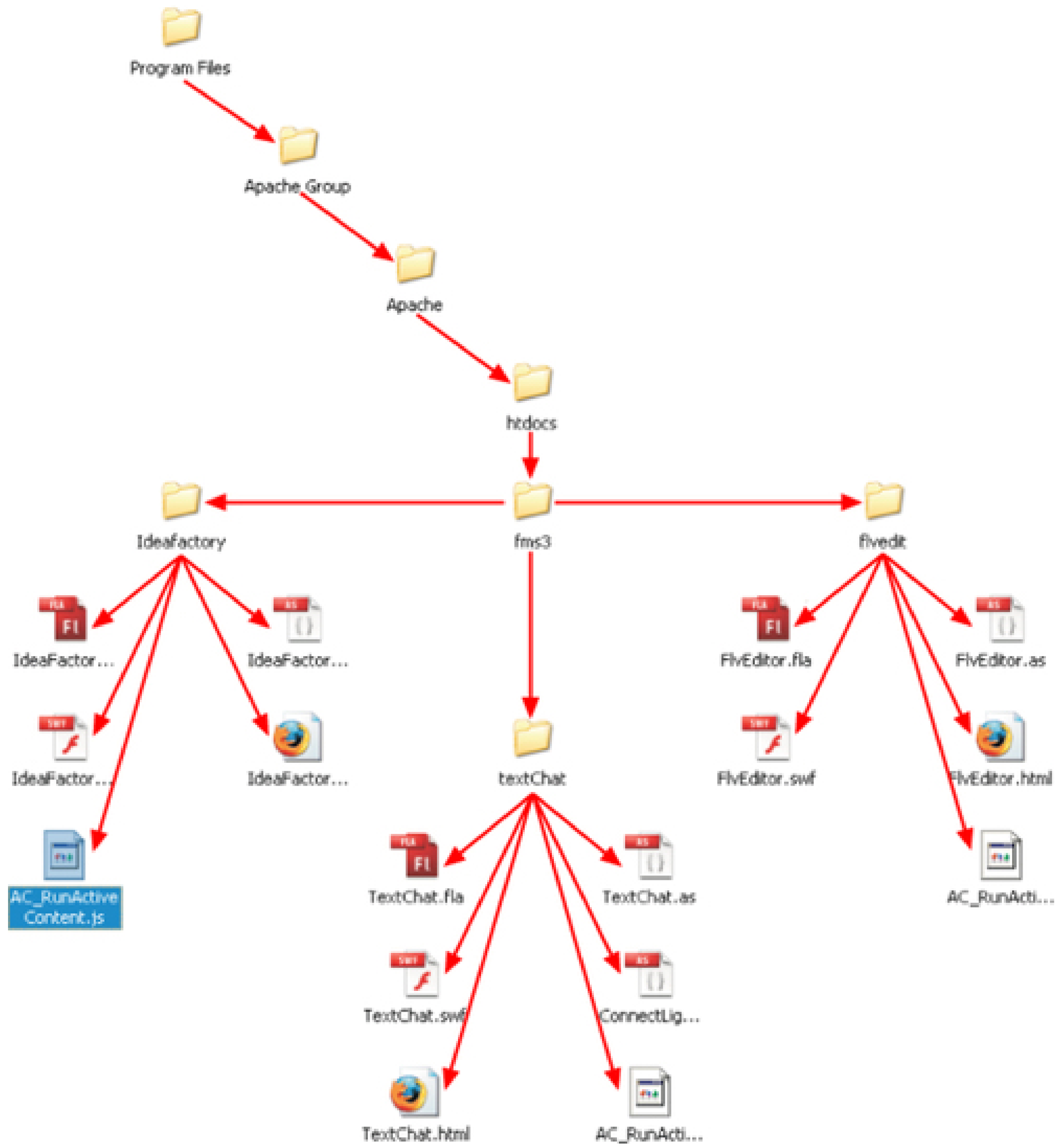


In Figure 1-10, the ideafactory and flvedit applications contain ASC files with the same name as the application folder, and the textChat application's ASC file is named mainASC, but it could have been named textChat.asc. Likewise, all of the others ASC files could have been named main.asc. Use whatever system is convenient for you. As you look at different applications in this book, you'll see different folders associated with server-side files.

1.4.2. Client-Side Files

Your client-side files are the Flash FLA, AS, and SWF files that you're used to working with in Flash. If you have your server on your computer, they can go anywhere. When you use the Control Test Movie sequence with your application, your server will be found and everything should work fine. However, if you want to test your applications using a LAN or just using your browser, you need to place them where they will be recognized by your Web server using the HTTP protocol on your localhost (127.0.0.1) or local LAN IP address. That is, all your HTML and SWF files need to be placed in the Web server's root directory. Because it's easier to develop and test applications, you also can place the FLA and AS files in the same folders as the SWF and HTML files. Figure 1-11 shows a typical setup on a Windows XP box using an Apache Web server.

Figure 1-11. Client-side files set up for remote access



Depending on the root directory for your Web server, your setup will vary. For example, if you're using a Windows IIS server, the path would be:

```
C:\Inetpub\wwwroot\FMS3apps
```

The same concept is at work, but you're just using a different root.

NOTE

If you're working with two different "roots"-one for the server side and one for the client side-make a shortcut to each. It will save you a lot of time and make development go much smoother.

You can develop applications on your Windows PC perfectly well without using a Web server at all. If you're most comfortable developing and testing your applications on a single computer, you can develop your client-side applications wherever you want. Your desktop is as good a location as any.

1.4.3. Setting Up with a Macintosh

My particular setup uses a Macintosh for development, which connects through a LAN to FMS3 on my Windows XP and Vista computers. The good news is that Macintosh computers come with Apache servers already installed, so all you have to do is figure out where to place your client-side applications to hook up to FMS3 on the server. The next section shows how a LAN setup works with the Macintosh using the default path:

```
Home\sites\FMS3apps
```

The Home directory on Macintosh computers is usually the computer's name, in my case, billsanders. Figure 1-12 shows the path to FMS3 applications on a Macintosh. To make a connection, you need to use the connection through your Web server. You will connect to FMS3 on a remote server because FMS3 doesn't run on Mac OS. The next section shows how to connect to local and remote servers, including how to set up a LAN link using a Macintosh as the primary development computer.

Figure 1-12. Path to client-side applications on Macintosh

1.5. Testing FMS3 Connections

The protocol to connect to Flash Media Server 3 is RTMP-Real-Time Messaging Protocol. The newest version of RTMP is RTMPE (*E* for Encryption). The examples show both. The older RTMP is a bit faster, but RTMPE performs better for encryption and is more secure. You set up a net connection, and then place the RTMP path in your `NetConnection` instance. The basic routine is as follows:

```
nc = new NetConnection( );
nc.connect("rtmp:/appName");
```

If your path is to a server on the same computer (or physical server) as your application, you use only a single "/" slash; however, if your path is to a remote server or LAN, you use a double slash "//" as follows:

1.5.1.

1.5.1.1. Same computer:

```
nc.connect("rtmp:/FMSapp")
```

1.5.1.2. Different computers:

```
nc.connect("rtmp://www.myDomain.com/FMSapp")
```

Since I develop on a LAN, I used the latter. Also, I like to put the RTMP string into a string variable. Then when I have to change RTMP paths, I just comment out the old string and remove comment slashes from what I want to use. The examples in this book use the single slash, but the LAN version is commented out. Also, if you use a LAN, make sure that your RTMP uses the IP address for the FMS3 server location, and not the IP address where your client-side materials are being developed.

Once you've got your folder arrangements set up, you need to make a connection to test whether your FMS3 is performing in conjunction with your applications. To do this, you need a simple FMS3 application to test. If you don't have Flash installed, download the latest version from the Adobe.com Web site and install it. You will need the Flash IDE to follow the examples in this book. The following steps show you how to create a simple FMS3 application to test:

1. Open a new Flash file (ActionScript 3.0); this is an FLA file. Save it as FMS3Connect.fla.

2. Add a logo to the upper left corner of the Stage. (Optional).

3. Open the Property inspector from the Window menu. In the Document Class text box, type **`FMS3Connect`**. Save the file.

4. Open a new ActionScript file and save it as FMS3Connect.as. The class name, FMS3Connect, is the same as the file name minus the .as extension.

5. Add the Example 1-1 script and save the file again.

Example 1-1. FMS3Connect.as

Code View:

```
package
{
  import flash.net.NetConnection;
        import flash.display.Sprite;
  import flash.text.TextField;
  import flash.events.NetStatusEvent;

  public class FMS3Connect extends Sprite
  {
        private var nc:NetConnection;
        private var rtmpNow:String;
        private var msg:String;
        private var connectText:TextField;
        private var posX:Number;

        function FMS3Connect ()
        {

             nc=new NetConnection();
             nc.addEventListener (NetStatusEvent.NET_STATUS,checkConnect);
             rtmpNow="rtmpe://192.168.0.11/connect";
             //rtmpNow="rtmpe:/connect";
             nc.connect (rtmpNow);
        }
        private function checkConnect (event:NetStatusEvent):void
        {
             connectText=new TextField();
             msg=event.info.code;
             connectText.width=250;
             connectText.text=msg;
             addChild (connectText);
             posX=connectText.stage.stageWidth;
             connectText.x=(posX/2)-((msg.length/2)*(6));
             connectText.y=175;
        }
  }
}
```

6. On the server-side, add a folder and name it connect. The location of this folder is essential for FMS3 to work. For example, a typical path would be:

   Code View:

   ```
   C:\Program Files\Adobe \Flash Media Server 3\applications\ connect
   ```

Remember, all server-side files must be placed on the same computer as the FMS3 server, even if you're using

a LAN. (You'd never put server-side folders on a Macintosh!)

The folder name is the application name. This concept may take a bit of getting used to, but eventually it'll be as easy as pie. Figure 1-13 illustrates what your application should show. The first time you may have to wait a few seconds, especially if you're using a LAN.

You will soon learn that the prettiest message in the world is "NetConnection.Connect.Success." That means your application has successfully connected to Flash Media Server 3.

Figure 1-13. Message indicating successful connection to server

1.6. Using the FMS3 Administration Console

To see what's going on in your server, you need to use the FMS3 Administration console. It shows you what's going on behind the scenes on your media server. If your FMS3 is on Windows, navigate to Start All Programs → Adobe → Flash Media Server 3 Management Console, as shown in Figure 1-9.

When you initially open the Administration console, you will be asked to enter the Administrative user name and password. This is the name and password you wrote down when you installed your server. (See Figure 1-5.) Type your user name and password, and if your server is in a secure location, you can click the check boxes to remember the information and automatically open the console when you start it. For development, I use the Administration console all the time, so I have the user name and password automatically entered. However, if your development environment is not secure, you should require that the user enter the user name and password.

Launch the application FMS3Connect, developed in the previous section. In the Administration console, select the ViewApplications → Clients option. When the application successfully launches, you will see the name of the application, the number of clients currently using the application, plus other information about the client as shown in Figure 1-14.

Figure 1-14. Flash Media Administration Console

As you become more familiar with FMS3, you will find the Administration console more useful. In subsequent chapters, you'll see how it can provide invaluable feedback on what your server is doing.

1.7. Using This Book

Learning with Flash Media Server 3 is designed to offer you options, but at the heart of every chapter is a core set of code that shows the minimum requirements needed for different procedures. Later chapters will have some material that is more complex, more sophisticated, and optimized for getting the most out of FMS3. The chapters will include the minimalist set of code for different operations. This code is presented as a procedure with the sole purpose of clarifying what must be in place for something to happen. It won't be sophisticated, but it will work and help you understand how to accomplish different goals with FMS3. In the more advanced chapters, the same operations may be rendered as a class or even a component. Sometimes in the same chapter you will see a range of solutions for the same task, ranging from a simple procedure to an object-oriented class to a design pattern.

1.7.1. Flash and ActionScript 3.0

Unlike previous versions of Flash Media Server, FMS3 arrives with the introduction of a major ActionScript revision. ActionScript 3.0 is the first true ECMAScript 4 implementation of ActionScript. In the future, expect more languages to adopt one ECMA standard or another. As Internet languages adopt these standards, different languages will overlap more and require less work to learn a new one. Because ActionScript 3.0 represents a very different way of programming in ActionScript-more like Java, C++ and C#-you may need to take some time to go over it. All of the client-side code is in ActionScript 3.0 and the server-side script is in ActionScript 1.0; so the server-side script may actually be more familiar to you. In fact, if you don't know Flash and ActionScript, before you go too far with FMS3, you'd better spend some time getting up to speed on both. All of the examples in this book use Flash CS3 and ActionScript 3.0, which means that most of the code won't work with earlier versions of Flash. However, if you are using Flex 2 or later, you should be able to work around some of the Flash CS3 elements and reproduce the examples.

If you've used Flash Communication Server 1.5 or Flash Media Server 2, you'll probably find some differences; however, all of the code in this book was developed using FMS3. So if something doesn't work with FCS 1.5 or FMS2, it's because some features were added to FMS3 that are not available in FCS 1.5 or FMS2.

1.7.2. Client-Side and Server-Side ActionScript

In addition to the ActionScript you use with Flash, three other ActionScripts are used with FMS3, the most important and immediate being Client-side and Server-side ActionScript. Client-side ActionScript is generated in the ActionScript files just as regular ActionScript, except that it contains certain classes, methods, and properties that are either unique to using FMS3 or used in unique ways. Server-side ActionScript is almost identical to JavaScript except it is saved with an .asc file extension and contains some uniquely FMS3 related classes not found in either JavaScript or ActionScript. The ASC files are written in a special ActionScript Communication File editor that looks almost identical to the ActionScript File editor. The third ActionScript used with FMS3 is Server Management ActionScript, used for FMS3 management. Think of it in the same context as you would Network Administration for administering FMS3. Installing Flash Media Server 3 also installs a full set of PDF files explaining both Client-side and Server-side ActionScript in:

```
C:\Program Files\Adobe\Flash Media Server 3\documentation
```

This documentation includes a full set of FMS3 documentation in the following folders and PDF files:

- *ActionScript 3.0 Language and Components Reference* documents ActionScript™ 3.0 (client-side AS Reference for both FMS3 and non-FMS3 applications). *flashmediaserver_AS3LR*(folder).

- *Adobe Flash Media Server ActionScript 2.0 Language*, *flashmediaserver_AS2LR.pdf*.
- *Server-Side ActionScript Language Reference for Adobe Flash Media Interactive Server*, *flashmediaserver_SSLR.pdf*.
- *Adobe Flash Media Server Administration API Reference*, *flashmediaserver_AdminsitrationAPI.pdf*.
- *Adobe Flash Media Interactive Server Plug-in API Reference*, *flashmediaserver_plug_in_API* (folder).
- *Adobe Flash Media Server Configuration and Administration Guide*, *flashmediaserver_config_admin.pdf*.
- *Adobe Flash Media Server Development Guide*, *flashmediaserver_dev_guide.pdf*.
- *Adobe Flash Media Server Installation Guide*, *flashmediaserver_install.pdf*.
- *Adobe Flash Media Interactive Server Plug-In Developer Guide* (Note: This is for FMIS or FMDS), *flashmediaserver_plugin_dev.pdf*.
- *Adobe Flash Media Server Technical Overview*, *flashmediaserver_tech_overview.pdf*.

As a starting point, print out the two *Server-Side ActionScript Language Reference for Adobe Flash Media Interactive Server* and *Adobe Flash Media Server Development Guide* documents and organize them in a loose-leaf binder. Keep the folder with the ActionScript 3.0 reference handy for looking up different client-side code. This book includes all of the ActionScript 3.0 code for the applications, but having a full-language reference at hand is always important. This book was designed with these available documents in mind. For the basic how-to, this book should serve you well, but you should also look at the FMS3 file on developing applications. The more angles you look at FMS3 from, the better you'll understand it.

Chapter 2. Recording and Playing Back Streaming Audio and Video

2.1. Streaming and Broadcasting

Flash Media Server 3 can both record and play back streaming audio and video. The nature of streaming is not the same as broadcasting. In broadcasting, the sender sends out a single signal; everyone who connects to the channel sending the signal gets the same stream. It's like dropping a pebble in the water-a concentric circle of waves sends out a "signal." No matter where viewers or listeners are, when the wave reaches them, they get the same wave-like a TV picture or radio transmission-as everyone else.

When FMS3 sends out a stream of audio and video, it creates a separate stream for each recipient. Its "broadcasting" works more like the spokes on a wagon wheel, where everyone connected gets his or her own stream. So here the term "broadcasting," really means a form of *streamcasting*-a technology where everyone gets a separate stream. If one person is listening, my application sends only a single stream; but if 20 people are listening, it generates 20 streams. Because the server automatically creates a separate stream for each user connected, you don't have to create all those streams in your coding. However, in deciding how to set up your application, for bandwidth considerations you need to consider the number of streams it may generate.

If you've ever viewed online video, you've probably seen different kinds without really realizing it. If you click on a video and have to wait a long time, what you're really waiting for is for the video file to first download and then for your computer to play the video that's saved to your hard drive. That works fine except you have to wait until the file is fully downloaded before you can watch it. Also, it's saved on your hard drive taking up space. A second type of video processing is called *progressive download*. It's like a hybrid of a video download and streaming. As the video file is downloaded, it begins to play rather than wait for the whole thing to download. However, the file will end up on the user's hard drive, and processing is not as smooth as true streaming. The third type of video processing, if you have FMS3 on your server, lets you stream video files from the server. This chapter explains how to get started with both creating and streaming Flash Video (FLV) files using FMS3.

Chapter 2. Recording and Playing Back Streaming Audio and Video

2.1. Streaming and Broadcasting

Flash Media Server 3 can both record and play back streaming audio and video. The nature of streaming is not the same as broadcasting. In broadcasting, the sender sends out a single signal; everyone who connects to the channel sending the signal gets the same stream. It's like dropping a pebble in the water-a concentric circle of waves sends out a "signal." No matter where viewers or listeners are, when the wave reaches them, they get the same wave-like a TV picture or radio transmission-as everyone else.

When FMS3 sends out a stream of audio and video, it creates a separate stream for each recipient. Its "broadcasting" works more like the spokes on a wagon wheel, where everyone connected gets his or her own stream. So here the term "broadcasting," really means a form of *streamcasting*-a technology where everyone gets a separate stream. If one person is listening, my application sends only a single stream; but if 20 people are listening, it generates 20 streams. Because the server automatically creates a separate stream for each user connected, you don't have to create all those streams in your coding. However, in deciding how to set up your application, for bandwidth considerations you need to consider the number of streams it may generate.

If you've ever viewed online video, you've probably seen different kinds without really realizing it. If you click on a video and have to wait a long time, what you're really waiting for is for the video file to first download and then for your computer to play the video that's saved to your hard drive. That works fine except you have to wait until the file is fully downloaded before you can watch it. Also, it's saved on your hard drive taking up space. A second type of video processing is called *progressive download*. It's like a hybrid of a video download and streaming. As the video file is downloaded, it begins to play rather than wait for the whole thing to download. However, the file will end up on the user's hard drive, and processing is not as smooth as true streaming. The third type of video processing, if you have FMS3 on your server, lets you stream video files from the server. This chapter explains how to get started with both creating and streaming Flash Video (FLV) files using FMS3.

2.2. Minimalist Project

To get started you'll create a little project containing the minimum number of elements needed to create an FLV file using ActionScript 3.0. The following are the packages and classes you'll need (client-side):

Packages

fl.controls.Button;

fl.controls.TextInput;

flash.display.Sprite;

flash.net.NetConnection;

flash.net.NetStream;

flash.events.NetStatusEvent;

flash.events.MouseEvent;

flash.events.Event;

flash.net.ObjectEncoding;

flash.media.Camera;

flash.media.Microphone;

flash.media.Video;

Key Classes

NetConnection

NetStream

Camera

Microphone

Video

Button

The package list shows all the packages you will need for the final application in this chapter that both records and plays back FLV files using FMS3. Of these packages, one of the most important to import is the

`ObjectEncoding` package. FMS2 (in case you're using it instead of FMS3) requires the `NetConnection` property `defaultObjectEncoding` to change the AMF default from 3 to 0 using the following line:

```
NetConnection.defaultObjectEncoding = flash.net.ObjectEncoding.AMF0;
```

You will see this property used throughout the book for FMS2 users who want to employ ActionScript 3.0 in their client-side code. Just uncomment the `defaultObjectEncoding` lines.

One more important chore before you get to the actual coding. You need a server-side folder named vid2. This will be your FMS application folder, and while it won't need a server-side (ASC) file, it's needed to run your FMS application. After you run your first application, you will find additional folders inside the vid2 folder.

2.2.1. Camera and Microphone

To set up an application to record video, you will need a Webcam connected to your computer and a microphone. Some Webcams have built-in microphones, others have no microphones, and still others come with ones that must be plugged into the microphone port of your computer. Many computers come with built-in microphones, especially on portable computers and most versions of Macintosh computers. You also may find that a Webcam comes installed in your computer. Some Windows-based portable computers, and iMac and MacBook model Macintosh computers have built-in Webcams as well as microphones.

Examples in this chapter include a few settings for the camera, but most of the settings and their explanations are in Chapter 3. The following line sets the camera to a 240 x 180 capture width and height. The frames-per-second (FPS) is set to 24.

```
cam.setMode (240,180,24);
```

The camera quality is set fairly high, 90 out of 100 using the `Camera.setQuality` method in the only other line that is used with camera settings.

```
cam.setQuality (0,90);
```

For the minimum applications in this chapter, that is all you need to know. But for fine-tuning your application, you will spend time working with these settings and others for optimizing the efficiency of your application.

2.2.2. Minimum Code to Record FLV File

With ActionScript 3.0, you can create your entire application with a minimum of manual Stage placement. For this, and most of the other examples in this book, you will see how easy it is to create everything you need using ActionScript 3.0. To get started, you need to open a regular Flash file and save it as MinRecord.fla. To this file, add a Button and TextInput component to the Library panel. Once you add these components, you can position them on the Stage by creating instances of each using the component name as the class name. So to add a button, for example, you simply create a new instance name using Button as the data type. The following code segment shows how:

```
var recordBtn:Button=new Button();
```

In addition, you have to import the correct package, `fl.controls.Button`. Finally, you add the component to the Stage using the `addChild()` method. Then using the instance's x and y properties, you can place it where

you like on the Stage.

To display your video, you need a `Video` instance. The video instance is used to display what your Camera object "sees." Likewise, you will need a `Microphone` instance to deal with audio data you are transmitting to be recorded. To connect and stream the audio and video, you need both a `NetConnection` instance and a `NetStream` instance. Putting it all together, you will do the following:

1. Create camera and microphone objects for capturing video and audio.

2. Place a video instance on the Stage to see what the camera sees.

3. Make a connection to the server with either an absolute or relative RTMP address (`NetConnection.connect("rtmp://url/app/inst")`) and create a stream that will carry the video and audio to the media server where they will be recorded in an FLV file.

The following pseudo-code illustrates the sequence used to carry the audio and video to the media server:

1. First make instances of a camera, microphone, and video.

```
var cam:Camera = Camera.getCamera();
var mic:Microphone = Microphone.getMicrophone();
var vid:Video=new Video(w,h);
```

2. Set up your net connection and stream.

```
var nc:NetConnection=new NetConnection();
nc.connect("rtmp://www.myFMShost.com/appName/myInstance");
var ns:NetStream=new NetStream(nc);
```

3. Attach video to camera and net stream to camera and microphone.

```
vid.attachCamera(cam); //This is so that you can see video
ns.attachAudio (mic); //Give audio a ride on the stream
ns.attachCamera (cam); //Give video a ride on the stream
```

4. Publish the net stream.

```
ns.publish ("FLVname","record");
```

NOTE

When you instantiate an instance of either a camera or microphone object, you use a `get` method instead of `new`, as is the case with most other objects. The correct statement to instantiate a camera object is

```
var myCam:Camera = Camera.getCamera();
```

and for a microphone,

```
var myMic:Microphone = Microphone.getMicrophone();
```

Using the RTMP protocol described in Chapter 1, "Getting Started with Flash Media Server 3," the stream is directed to the Flash media server where your FLV file resides in the instance folder inside the streams folder, all within your application folder. In this case the folder arrangement on the FMS server looks like: applications → vid2 → streams → recordings → flv files.

In Example 2-1, you bring everything together. The main instances are all declared as private variables. In that way, the variables are available only to the class and its subclasses. Everything else is placed into a method (a function in the class) or is part of the constructor (generally the first function using the class name). An important method, `checkConnect`, uses the `NetStatusEvent` to determine whether the net connection is successful. The event listener is added to the `NetConnection` instance (`nc`) using the following line:

```
nc.addEventListener (NetStatusEvent.NET_STATUS,checkConnect);
```

If the connection is successful, only then does it instantiate the `NetStream`. It is important to maintain this practice throughout the book so that your application does not try to create a net stream when it is not connected to FMS.

To get everything up and running follow these steps:

1. Open a new Flash file (ActionScript 3.0); this is an FLA file. Save it as MinRecord.fla.

2. Open the Property inspector from the Window menu. In the Document Class text box, type `MinRecord`. Save the file.

3. Open a new ActionScript file and save it as MinRecord.as in the same folder as the MinRecord.fla file.

4. In the MinRecord.as file, add the Example 2-1 script and save the file again.

Example 2-1. MinRecord.as

Code View:

```
package
{
    import fl.controls.Button;
    import fl.controls.TextInput;
    import flash.display.Sprite;
    import flash.net.NetConnection;
    import flash.net.NetStream;
    import flash.events.NetStatusEvent;
    import flash.events.MouseEvent;
    import flash.events.Event;
    //import flash.net.ObjectEncoding;
    import flash.media.Camera;
    import flash.media.Microphone;
    import flash.media.Video;

    public class MinRecord extends Sprite
    {
          private var nc:NetConnection;
          private var ns:NetStream;
          private var rtmpNow:String;
          private var msg:Boolean;
          private var cam:Camera;
          private var mic:Microphone;
          private var vid1:Video;
          private var recordBtn:Button;
          private var stopBtn:Button;
          private var textInput:TextInput;

          //Constructor
          function MinRecord ()
          {
               //NetConnection.defaultObjectEncoding = flash.net.ObjectEncoding.AMF0;
               nc=new NetConnection();
               nc.addEventListener (NetStatusEvent.NET_STATUS,checkConnect);
               rtmpNow="rtmp://192.168.0.11/vid2/recordings";
               //rtmpNow="rtmp:/vid2";
               nc.connect (rtmpNow);
               addMedia ();
               addUI ();
               recordBtn.addEventListener (MouseEvent.CLICK,startRecord);
               stopBtn.addEventListener (MouseEvent.CLICK,stopRecord);
          }

          private function addMedia ():void
          {
               cam=Camera.getCamera();
               cam.setMode (240,180,24);
               cam.setQuality (0,90);
               mic=Microphone.getMicrophone();
               vid1=new Video(cam.width,cam.height);
               vid1.attachCamera (cam);
               addChild (vid1);
               vid1.x=100;
               vid1.y=50;
          }

          private function addUI ():void
          {
               recordBtn=new Button();
```

```
            recordBtn.label="Record";
            recordBtn.x=100;
            recordBtn.y=50+(cam.height) +5;
            recordBtn.width=70;
            addChild (recordBtn);
            stopBtn=new Button();
            stopBtn.label="Stop Record";
            stopBtn.x=recordBtn.x+85;
            stopBtn.y=recordBtn.y;
            stopBtn.width=75;
            addChild (stopBtn);
            textInput=new TextInput();
            textInput.x=recordBtn.x;
            textInput.y=recordBtn.y + 30;
            addChild (textInput);
        }

        private function checkConnect (e:NetStatusEvent):void
        {
            msg=(e.info.code=="NetConnection.Connect.Success");
            if (msg)
            {
                ns = new NetStream(nc);
            }
        }

        private function startRecord (e:Event):void
        {
            if (ns)
            {
                recordBtn.label="Recording";
                ns.attachAudio (mic);
                ns.attachCamera (cam);
                ns.publish (textInput.text,"record");
            }
        }

        private function stopRecord (e:Event):void
        {
            recordBtn.label="Record";
            ns.close ();
        }
    }
}
```

An important operation in the code is the need to close the `NetStream` instance (`ns.close()`) when you finish recording. Otherwise, the recording won't save the FLV file. Figure 2-1 shows what you see as you're making a recording.

Figure 2-1. Recording a Flash video

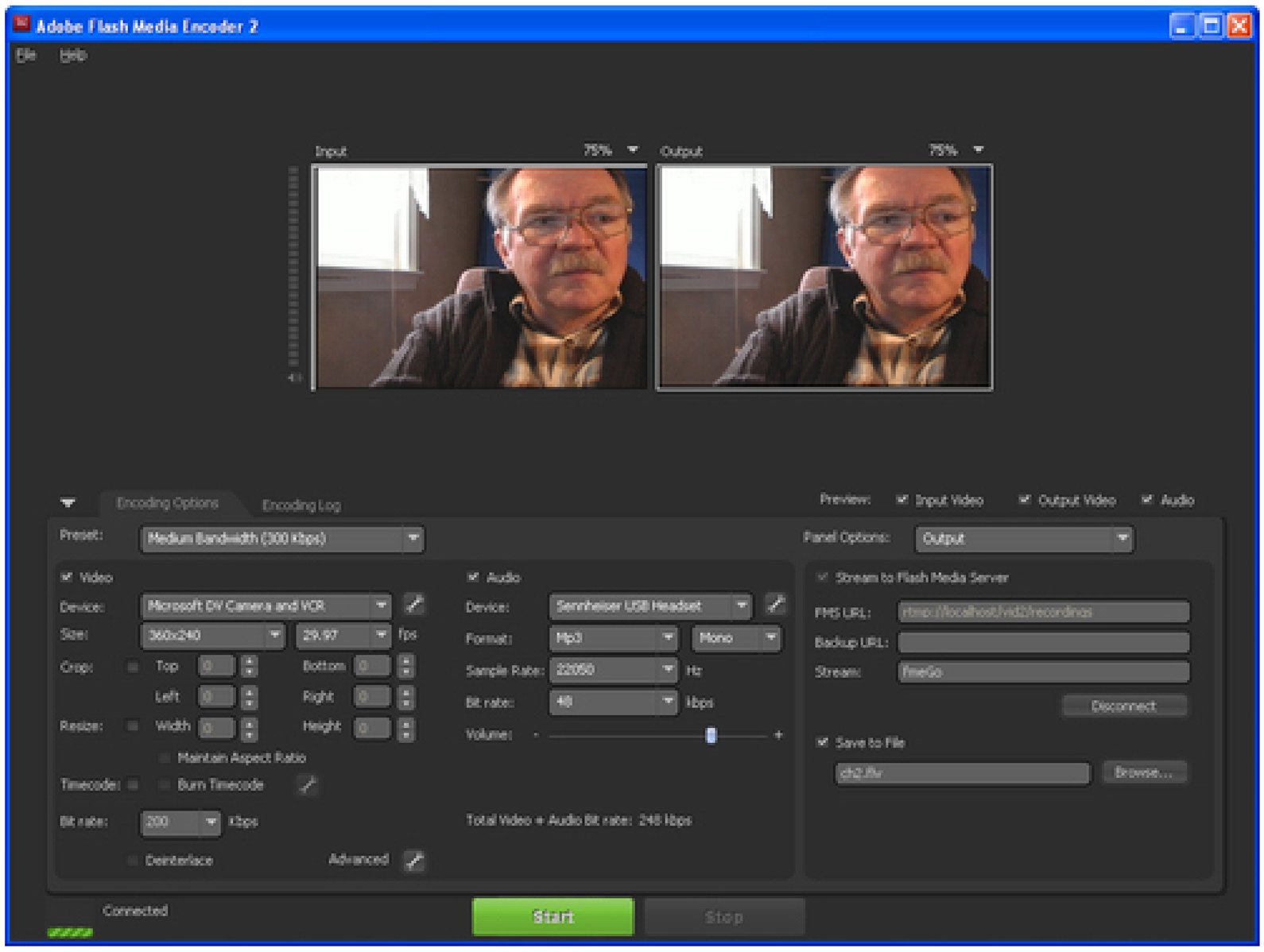
That done, you now need to create the minimum code required to play back a video. In this next section, you can use a lot of the code you used in the MinRecord application.

2.2.3. Flash Media Encoder 2 for FMSS Users

Using Flash Media Streaming Server you can stream video and audio, but you cannot record FLV files. Fortunately, you can download Flash Media Encoder 2 (FME2) from http://www.adobe.com/products/flashmediaserver/flashmediaencoder/ and use it with any version of FMS3. At the time of this writing, FME2 is available only for Windows platforms. (Of course you can always record using the Development version of FMS3 with the applications you see in this book.)

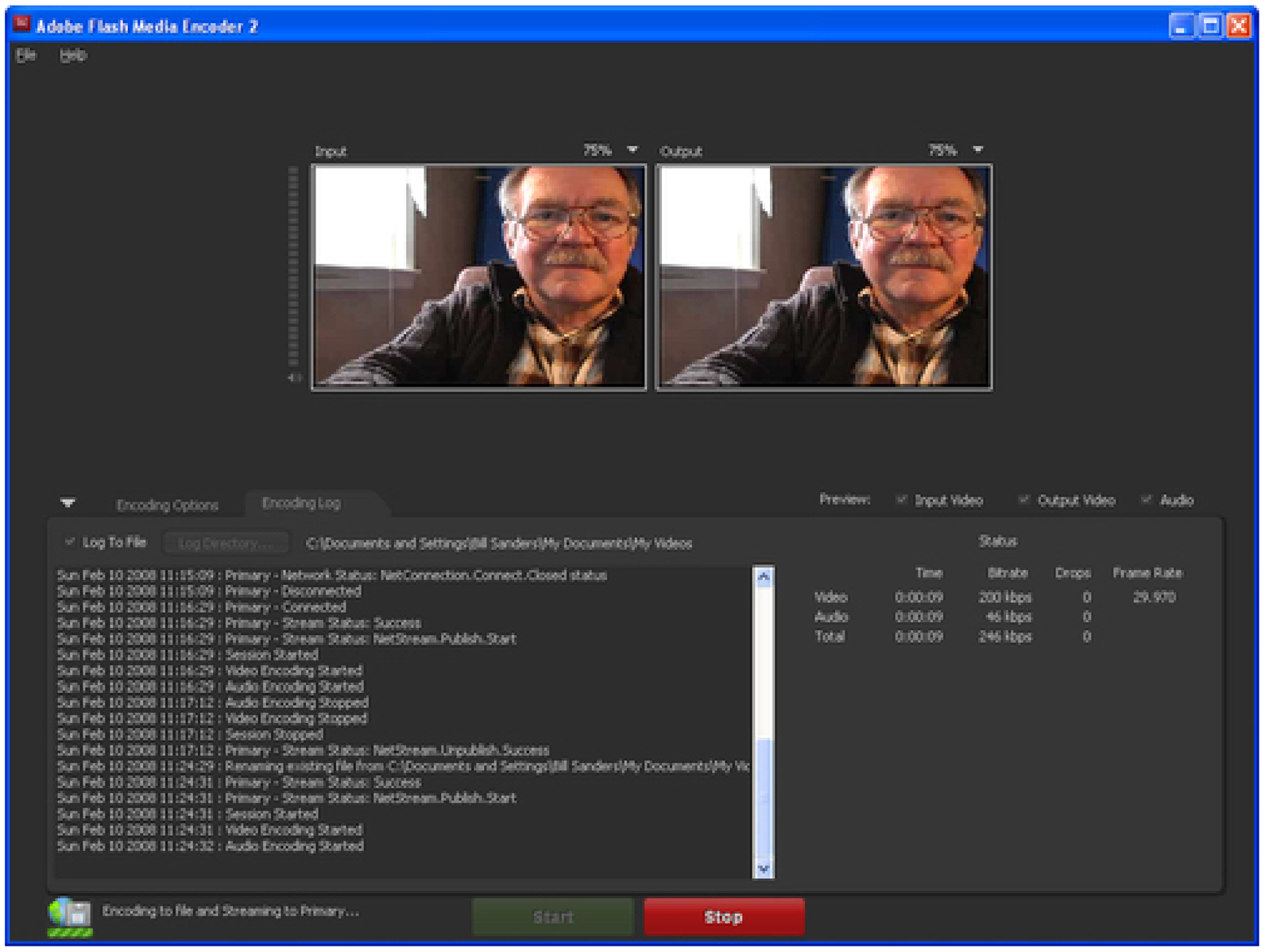
To set up a Flash Media Encoder 2, first add the name of the file (ch2.flv). (Here Figure 2-2 shows FME2 set to the same FMS3 folder as the application in the previous section- Example 2-1-, listed as rtmp://localhost/vid2/recordings in the FMS URL.) Turn on the Save to File option. Press the Connect button (it changes to Disconnect, as shown in Figure 2-2):

Figure 2-2. Flash Media Encoder 2 settings



To begin recording, click Start. (Chapter 3, "Setting Your Camera and Microphone," introduces you to live streaming and how FME2 can be used for live streams.) Figure 2-3 shows FME2 recording the video.

Figure 2-3. Flash Media Encoder 2 recording and streaming video



The FLV file continues being recorded until you press the Stop button. Once you press Stop, you can move your FLV file from the default My Documents/My Videos folder in Windows. Place it at the following path for use in the next section:

```
...applications\vid2\streams\recordings\ch2.flv
```

The applications folder is located at the root where your Flash Media server resides.

> NOTE
>
> How to Play H.264 Files
>
> You may have heard that FMS3 now lets you stream H.264 files such as MP4 files. Well you can. Unlike FLV files that don't require a reference to the .flv extension, you need to provide more information for H.264 files. A typical MP4 file might have a name like parade.mp4. First, change the name to parade.f4v. Then, to address the file, type mp4:parade.f4v. H.264 files require the special format. FMS3 does not record in the H.264 format, but you can find a lot of other devices that do. So if you have your heart set on H.264 format, you're all set with FMS3.

2.2.4. Minimum Code to Stream FLV

To be able to play back your video, you need a second script. This script won't need a camera or microphone

because you won't be transmitting any streams outward. You just want the stream to bring in what you have recorded, or as you'll see in subsequent chapters, what someone else is streaming to you. As a result, you do not need to import either the Camera or Microphone classes. However, you still need to import the Video class so that you have an object that can capture and display the streaming video coming from your FLV file

2.2.4.1. Playing FLV Files

The code for playing FLV files is very simple. The NetStream object simply plays instead of publishing a stream. The following code shows the key methods you need, where `ns` is the `NetStream` instance name:

```
myVideo.attachNetStream (ns);
ns.play ("myFlv");
```

An important difference between a progressive download and FMS, is that you do not include the file's .flv extension when calling the file to play. As Example 2-2 shows, the method for starting the FLV file to play uses information from the TextInput component. So to specify a name to play, simply type the name of the file before the .flv extension. Thus, to save a file as myFlv.flv, simply type **`myFlv`**.

2.2.4.2. Metadata

Whenever you play an FLV file, the Flash Player sends out a metadata event. For playback in general you do not need the metadata, but in some applications it can be crucial. For example, one metadata property is the duration of an FLV file, which can come in handy if you're creating a scrubber bar.

The metadata in an FLV file can vary widely. For those that you create yourself using FMS, you can rely on a fairly standard set of properties. The following is an example generated by the MinPlay application listed in Example 2-2:

```
creationdate = Mon May 28 11:36:04 2007
createdby = FMS 3.0
duration = 9.828
audiocodecid = 5
videocodecid = 2
canSeekToEnd = false
```

If you create an FLV using the Flash Video Encoder, you'll get a different set of properties as the following list shows:

```
canSeekToEnd = true
width = 720
videocodecid = 4
videodatarate = 400
duration = 30.563
height = 480
framerate = 29.969985961914063
audiodelay = 0.038
audiocodecid = 2
audiodatarate = 96
```

The code you need to capture metadata requires an object to host the `onMetaData` event that can be used by a `NetStream.client` property. The `client` simply specifies the object on which callback methods are invoked.

The following code segment shows how to set up an object to capture metadata information.

```
var metaSniffer:Object=new Object();
ns.client=metaSniffer;
metaSniffer.onMetaData=getMeta;
```

The callback function itself can retrieve a single metadata property or, as in the following segment, all of them.

```
function getMeta (mdata:Object):void
  {
        for (var prop:Object in mdata)
        {
             trace (prop+" = "+mdata[prop]);
        }
  }
```

For now, you really don't need the metadata, but it is an important element in all playbacks using FLV files. If you do not have a metadata handler in your code when you test it, in Flash you will get an error message even though your application will work fine.

2.2.4.3. Build Your Own FLV Player

At this point, you're all ready to build your own player. This one is simple and does not have all of the features of your FLVPlayer component, but it does a good job of playing back FLV files and stopping when you want.

1. Create a new Flash file (ActionScript 3.0) and save it as MinPlay.fla.

2. In the Property inspector in the Document Class text box, type **MinPlay** and save the FLA file again.

3. Open a new ActionScript 3.0 file and save it as MinPlay.as in the same folder as the MinPlay.fla file.

4. In the MinPlay.as file, enter the code in Example 2-2 and save the file again.

Example 2-2. MinPlay.as

Code View:

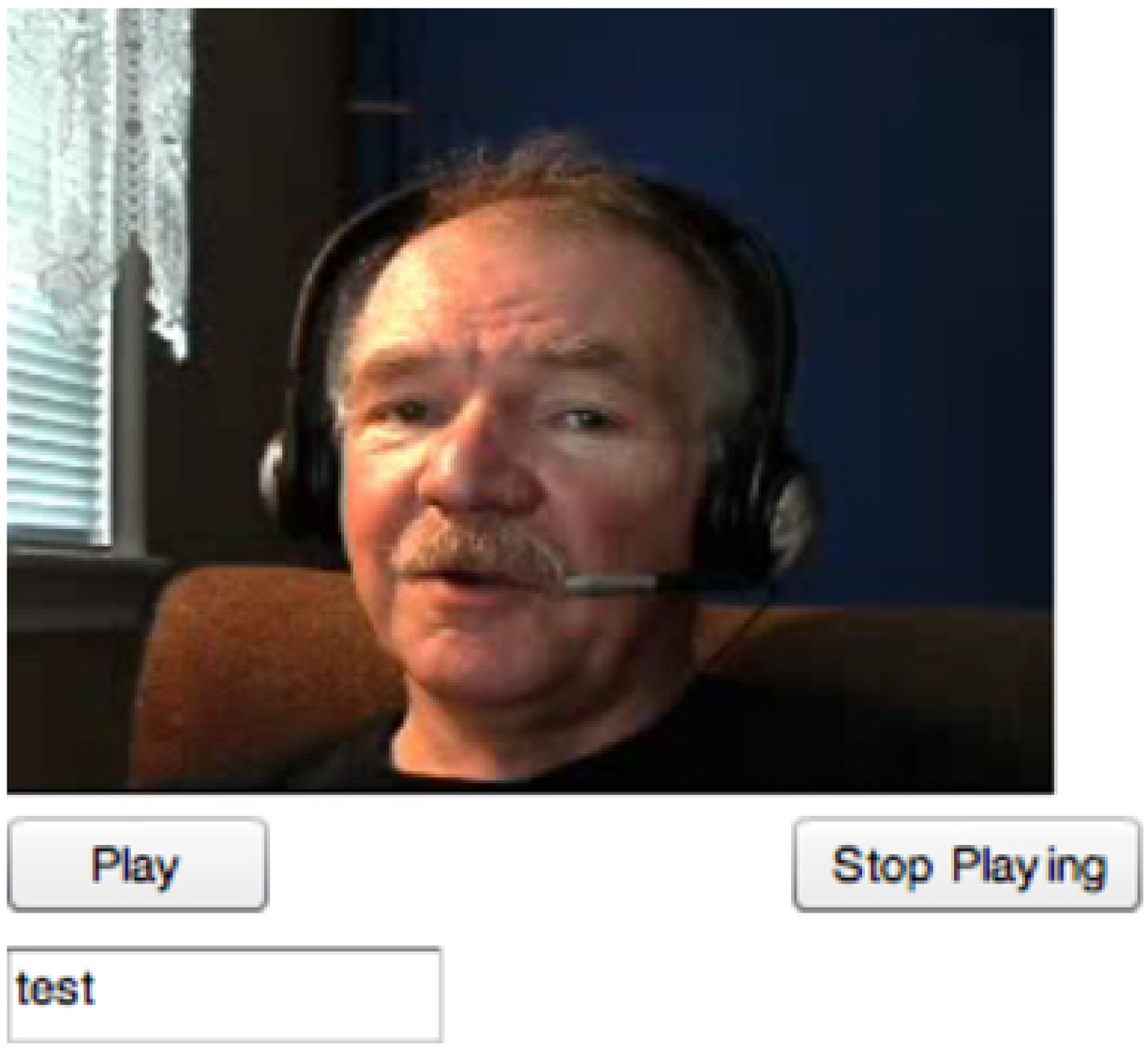
```
package
{
  import fl.controls.Button;
  import fl.controls.TextInput;
  import flash.display.Sprite;
  import flash.net.NetConnection;
  import flash.net.NetStream;
  import flash.events.NetStatusEvent;
  import flash.events.MouseEvent;
  import flash.events.Event;
  //import flash.net.ObjectEncoding;
```

```
import flash.media.Video;

public class MinPlay extends Sprite
{
	private var nc:NetConnection;
	private var ns:NetStream;
	private var rtmpNow:String;
	private var msg:Boolean;
	private var vid1:Video;
	private var playBtn:Button;
	private var stopBtn:Button;
	private var textInput:TextInput;
	private var metaSniffer:Object;
	private var dur:Number;

	function MinPlay ()
	{
		//NetConnection.defaultObjectEncoding = flash.net.ObjectEncoding.AMF0;
		nc=new NetConnection();
		nc.addEventListener (NetStatusEvent.NET_STATUS,checkConnect);
		rtmpNow="rtmp://192.168.0.11/vid2/recordings";
		//rtmpNow="rtmp:/vid2/recordings";
		nc.connect (rtmpNow);
		addMedia ();
		addUI ();
		playBtn.addEventListener (MouseEvent.CLICK,startPlay);
		stopBtn.addEventListener (MouseEvent.CLICK,stopPlay);
	}

	private function addMedia ():void
	{
		vid1=new Video(240,180);
		addChild (vid1);
		vid1.x=100;
		vid1.y=50;
	}

	private function addUI ():void
	{
		playBtn=new Button();
		playBtn.label="Play";
		playBtn.x=100;
		playBtn.y=180 +55;
		playBtn.width=60;
		addChild (playBtn);
		stopBtn=new Button();
		stopBtn.label="Stop Playing";
		stopBtn.width=80;
		stopBtn.x=playBtn.x+180;
		stopBtn.y=playBtn.y;
		addChild (stopBtn);
		textInput=new TextInput();
		textInput.x=playBtn.x;
		textInput.y=playBtn.y + 30;
		addChild (textInput);
	}

	private function checkConnect (e:NetStatusEvent):void
	{
```

```
            msg=(e.info.code=="NetConnection.Connect.Success");
            if (msg)
            {
                  ns = new NetStream(nc);
                  metaSniffer=new Object();
                  ns.client=metaSniffer;
                  metaSniffer.onMetaData=getMeta;
            }
      }

      private function getMeta (mdata:Object):void
      {
            for (var prop:Object in mdata)
            {
                  trace (prop+" = "+mdata[prop]);
            }
      }

      private function startPlay (e:Event):void
      {
            if (ns)
            {
                  playBtn.label="Playing";
                  vid1.attachNetStream (ns);
                  ns.play (textInput.text);
            }
      }

      private function stopPlay (e:Event):void
      {
            playBtn.label="Play";
            ns.play (false);
            ns.close ();
      }
  }
}
```

Now test your FLV player. Figure 2-4 shows what you should see when you enter the name of your FLV file and click the Play button.

Figure 2-4. Playback of FLV file



While the video is playing, the Play button shows Playing; when you click the Stop Playing button, the label reverts to Play. This is a minor user interface enhancement. In Chapter 3 and Chapter 6 you will see more ways to let users know what's going on in their application.

2.2.5. Playback Using the FLVPlayer Component

With the FLVPlayer component, you can easily play your FLV files. To do so, you'll need a key main.asc file. Follow these steps to get the file:

1. Open your browser and go to the following URL: www.adobe.com/support/documentation/en/flash/samples/

2. Select the Samples.zip file. At this writing, the file you need is for ActionScript 2.0 samples, not the ActionScript 3.0 samples. Be sure to select the first Samples.zip and not the one for ActionScript 3.0.

3. In the Samples folder, open the ComponentsAS2 folder and then the FLVPlayback folder, which contains the main.asc file that you need to make the FLVPlayer component work. The file will go into the server-side folder.

4. In your FMS3 applications folder, create a folder named player. Place the main.asc file in the player folder.

5. Add a folder named streams in the player folder. Then in the streams folder, add a folder named _definst_. In this folder, place the FLV files you want to play. For the purposes of this application, rename one of the FLV files as test.flv. Now you're all set to play your videos using FMS3.

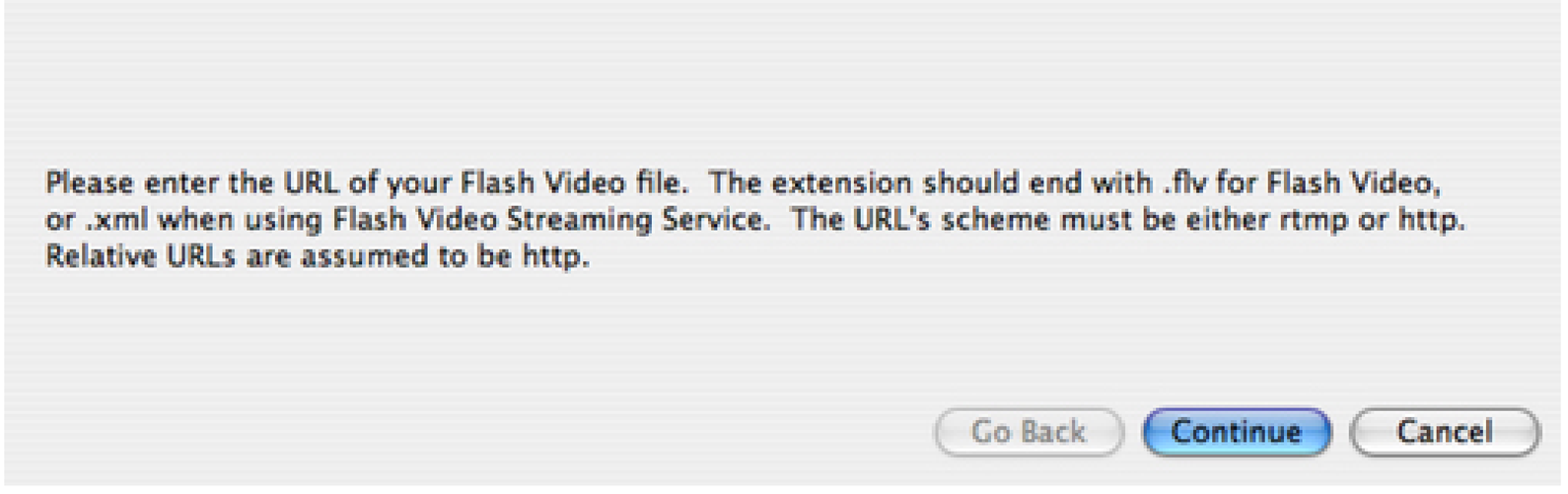
Once you've set up your server-side folders and files, you'll need to open a new Flash file (ActionScript 3.0) and save it as FLVPlayer.fla. Then do these steps:

1. In your FLVPlayer application, select File Import Import Video

2. When the Select Video window opens, click the radio button, "Already deployed to a web server. . . ."

3. Type `rtmp://localhost/player/test.flv` in the URL window (or the correct Web server or IP address for your particular setup) as shown in Figure 2-5. Click Continue.

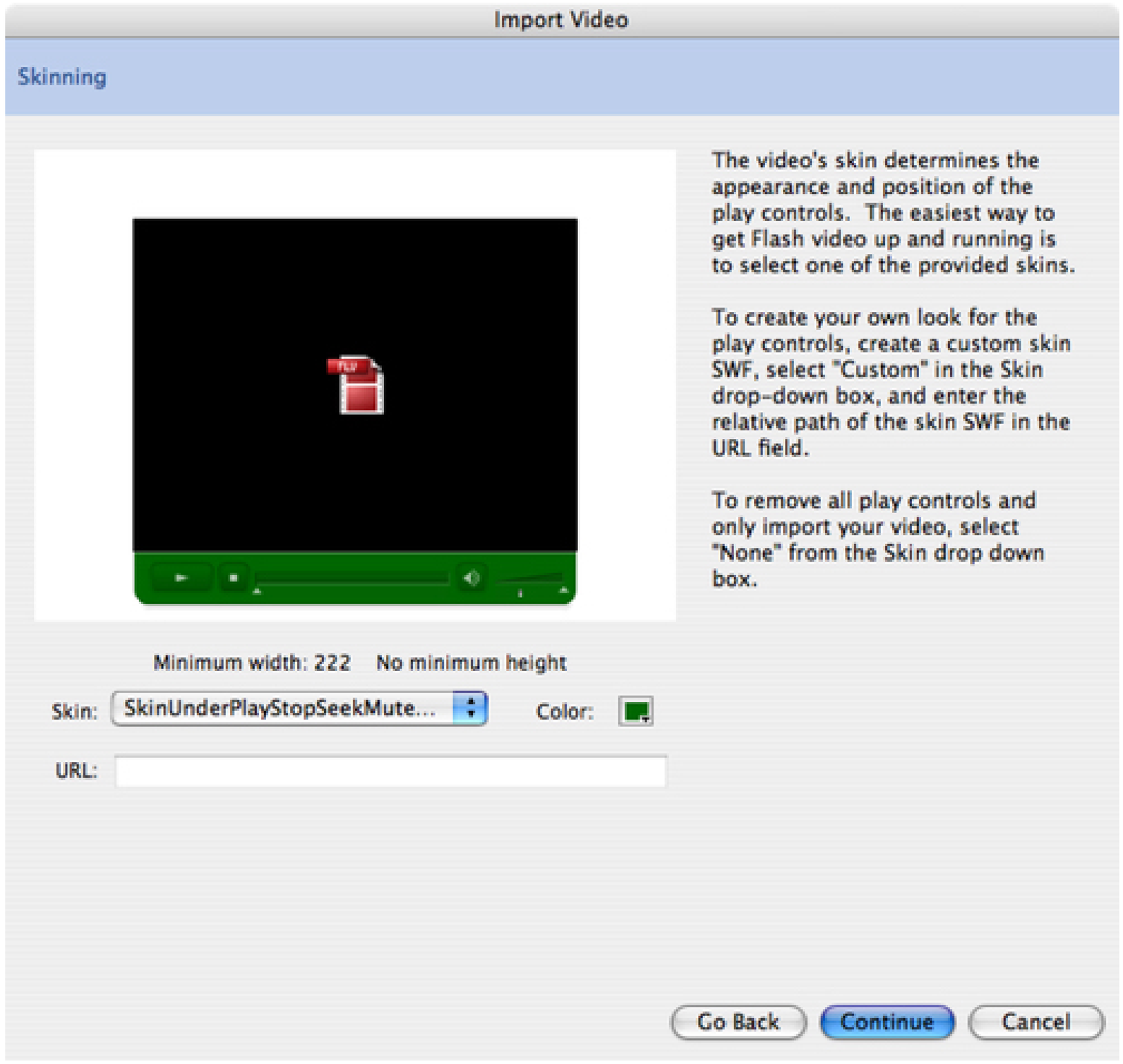
Figure 2-5. Specifying the URL for RTMP protocol



4. In the Skinning window, select a skin from the Skin pop-up menu and a color from the Color box. Figure

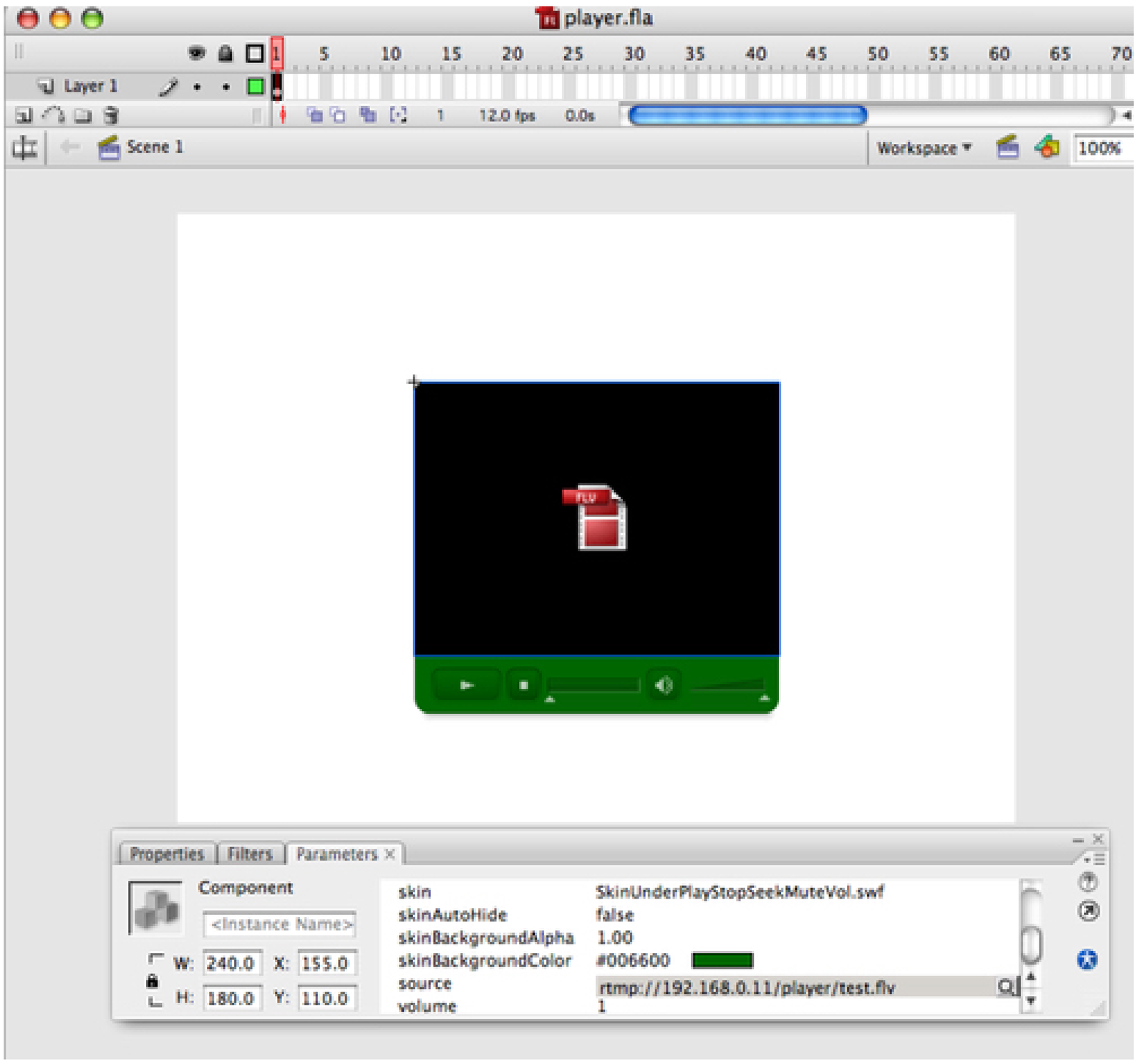
2-6 shows a typical selection. You can leave the URL window blank. If you later create a custom skin for your player, then select the URL where you stored the skin. Click Continue.

Figure 2-6. Choosing a control skin and color



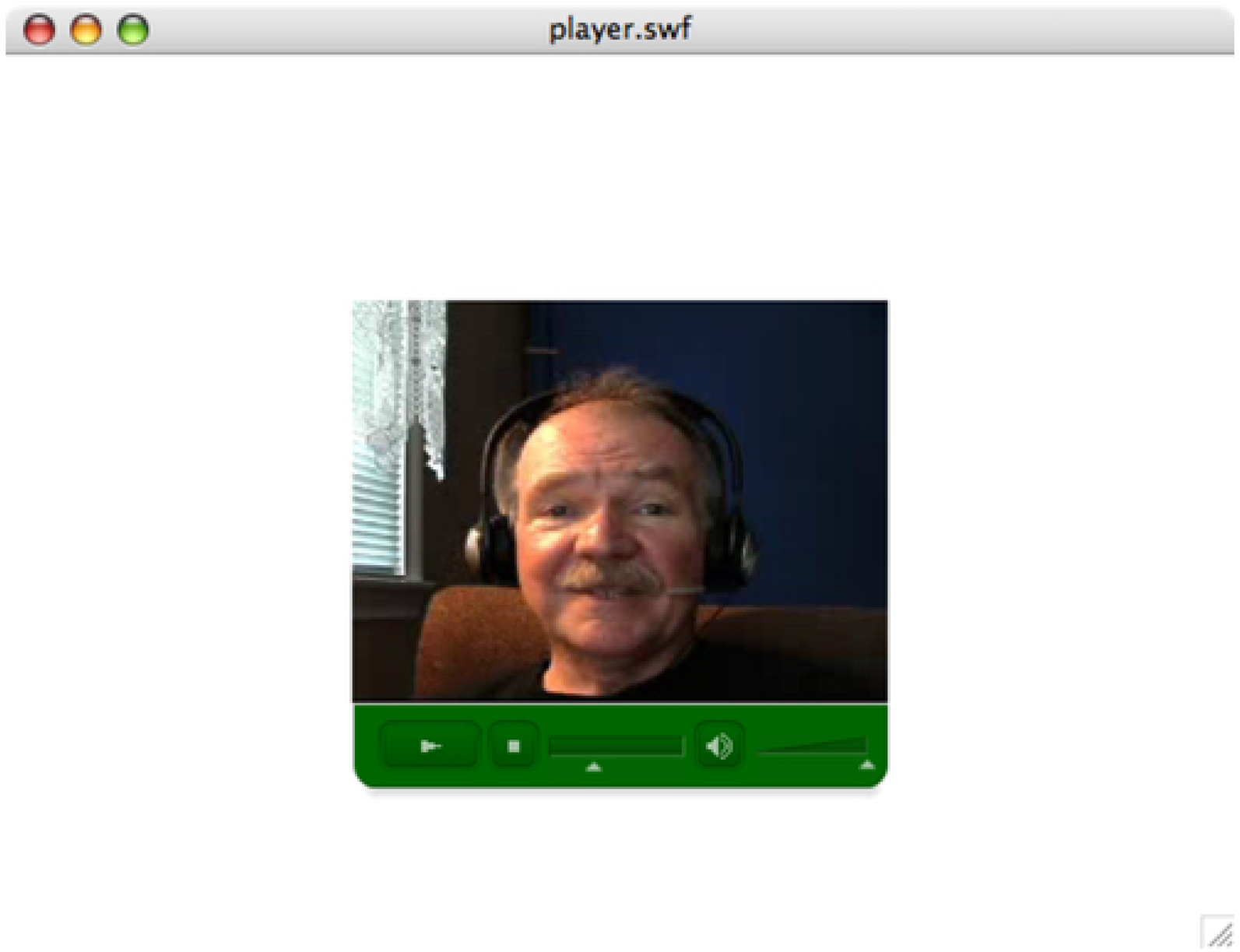
5. Read the Finish Video Import window for an explanation of what happens when you click Finish. Then click Finish. Figure 2-7 displays what appears on your Stage.

Figure 2-7. FLVPlayer on the Stage



6. Test your video by selecting Control Test Movie. Your video should play, as shown in Figure 2-8.

Figure 2-8. Video playing from FMS3



Publish the application just as you would any Flash application, but when you place the HTML and SWF files on the production server, include the second SWF file for the player skin as well.

Using the FLVPlayer built into Flash CS3 has the advantage of containing all the controls you need, such as the scrubber and mute button. Dynamically passing different FLV files is a bit more problematic, though. You need to plan ahead and know where you are going to be placing your main.asc file, your FLV (or H.264) files. The FLVPlayback also has a unique set of properties, methods and events that can be employed. These FLVPlaybackr properties are found in the fl.video package in the FLVPlayback class.

2.3. Combined Record and Playback Application

This final minimalist application can be used for both recording and playback. It uses two video windows so that you can keep the live view while reviewing a recorded video. Also, it uses the MovieClip object as a subclass instead of as a Sprite class in setting up the RecordPlay class. This change illustrates how you might incorporate a logo based on a MovieClip object in your application..

Follow these steps to develop the application:

1. Create a new Flash file (ActionScript 3.0), set the Stage size in the Property inspector to 650 x 400 pixels. If you like, add a background color. Save the file as RecordPlay.fla

2. Use your own logo or make one up on the Stage. (Just draw a simple logo, select it, press the F8 button, and select Movie Clip as Type, click Export for ActionScript, and click OK.)

3. In the Property inspector in the Document Class text box, type `RecordPlay` and save the FLA file again.

4. Next, open an ActionScript 3.0 file and save it as RecordPlay.as in the same folder as the RecordPlay.fla file.

5. In the RecordPlay.as file, enter the code in Example 2-3 and save the file again.

Example 2-3. RecordPlay.as

Code View:

```
package
{
  import fl.controls.Button;
  import fl.controls.TextInput;
  import flash.display.MovieClip;
  import flash.net.NetConnection;
  import flash.net.NetStream;
  import flash.events.NetStatusEvent;
  import flash.events.MouseEvent;
  import flash.events.Event;
  //import flash.net.ObjectEncoding;
  import flash.media.Camera;
  import flash.media.Microphone;
  import flash.media.Video;

  public class RecordPlay extends MovieClip
  {
        private var nc:NetConnection;
        private var ns:NetStream;
        private var rtmpNow:String;
        private var msg:Boolean;
        private var cam:Camera;
        private var mic:Microphone;
```

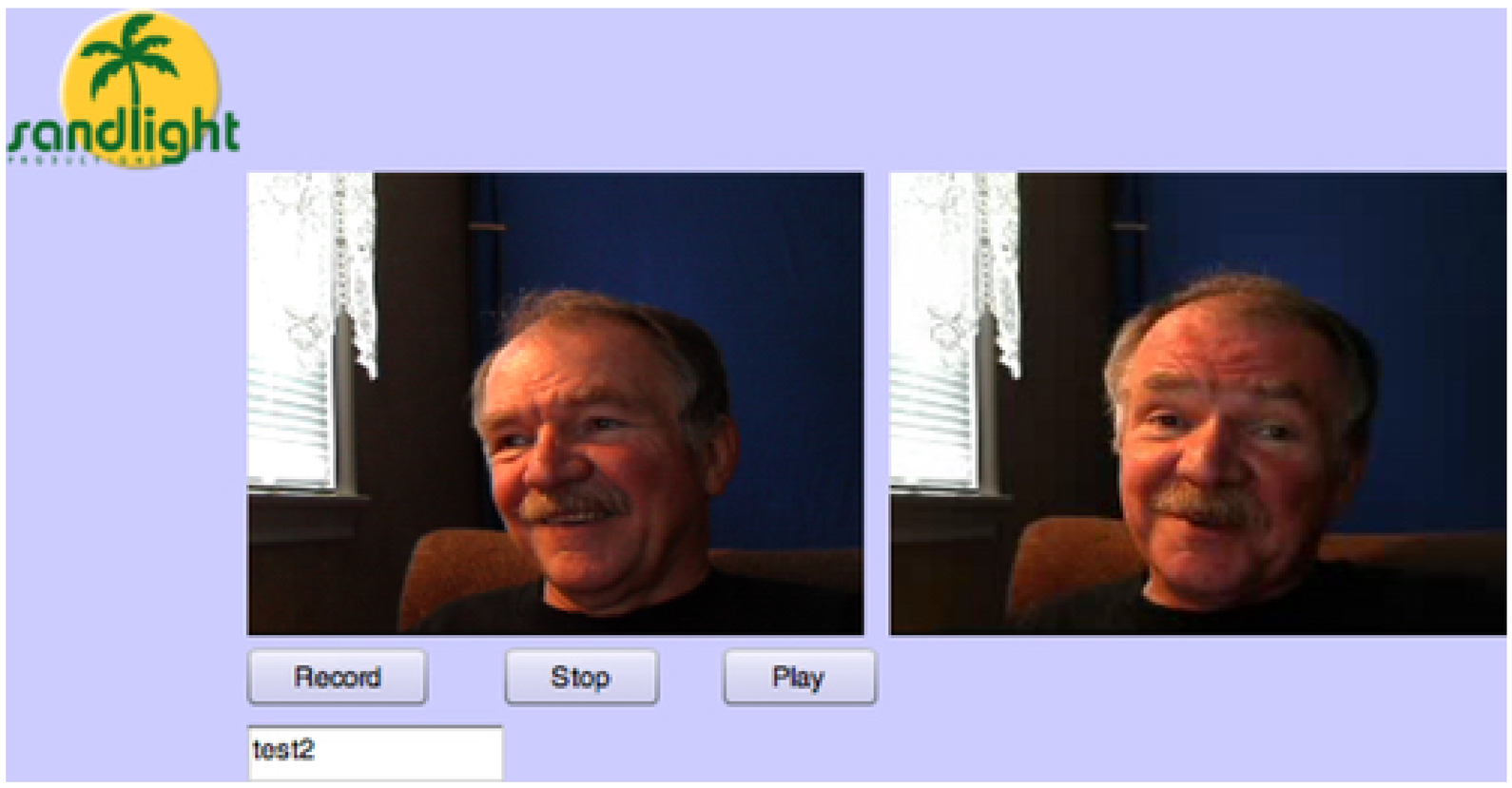
```
private var vid1:Video;
private var vid2:Video;
private var recordBtn:Button;
private var stopBtn:Button;
private var playBtn:Button;
private var textInput:TextInput;
private var metaSniffer:Object;
private var dur:Number;

function RecordPlay ()
{
     //NetConnection.defaultObjectEncoding = flash.net.ObjectEncoding.AMF0;
     nc=new NetConnection();
     nc.addEventListener (NetStatusEvent.NET_STATUS,checkConnect);
     rtmpNow="rtmp://192.168.0.11/vid2/recordings";
     //rtmpNow="rtmp:/vid2/recordings";
     nc.connect (rtmpNow);
     addMedia ();
     addUI ();
     recordBtn.addEventListener (MouseEvent.CLICK,startRecord);
     stopBtn.addEventListener (MouseEvent.CLICK,stopAll);
     playBtn.addEventListener (MouseEvent.CLICK,startPlay);
}

private function checkConnect (e:NetStatusEvent):void
{
     msg=(e.info.code=="NetConnection.Connect.Success");
     if (msg)
     {
          ns = new NetStream(nc);
          metaSniffer=new Object();
          ns.client=metaSniffer;
          metaSniffer.onMetaData=getMeta;
     }
}

private function getMeta (mdata:Object):void
{
     //Dummy to avoid error
}

private function addMedia ():void
{
     cam=Camera.getCamera();
     cam.setKeyFrameInterval (12);
     cam.setMode (240,180,15);
     cam.setQuality (0,80);
     mic=Microphone.getMicrophone();
     mic.rate=11;
     videoSetup ();
}

private function videoSetup ():void
{
     vid1=new Video(cam.width,cam.height);
     vid1.attachCamera (cam);
     vid1.x=100;
     vid1.y=70;
     addChild (vid1);
```

```
            vid2=new Video(cam.width,cam.height);
            vid2.x=vid1.x+vid1.width+10;
            vid2.y=vid1.y;
            addChild (vid2);
      }

      private function addUI ():void
      {
            recordBtn=new Button();
            recordBtn.label="Record";
            recordBtn.x=100;
            recordBtn.y=70+(cam.height) +5;
            recordBtn.width=70;
            addChild (recordBtn);
            stopBtn=new Button();
            stopBtn.label="Stop";
            stopBtn.x=recordBtn.x+100;
            stopBtn.y=recordBtn.y;
            stopBtn.width=60;
            addChild (stopBtn);
            playBtn=new Button();
            playBtn.label="Play";
            playBtn.x=stopBtn.x+85;
            playBtn.y=recordBtn.y;
            playBtn.width=60;
            addChild (playBtn);
            textInput=new TextInput();
            textInput.x=recordBtn.x;
            textInput.y=recordBtn.y + 30;
            addChild (textInput);
      }

      private function startRecord (e:Event):void
      {
            if (ns)
            {
                  recordBtn.label="Recording";
                  ns.attachAudio (mic);
                  ns.attachCamera (cam);
                  ns.publish (textInput.text,"record");
            }
      }

      private function stopAll (e:Event):void
      {
            playBtn.label="Play";
            recordBtn.label="Record";
            ns.close ();
      }

      private function startPlay (e:Event):void
      {
            if (ns)
            {
                  playBtn.label="Playing";
                  vid2.attachNetStream (ns);
                  ns.play (textInput.text);
            }
      }
```

```
    }
}
```

The final step is to test your application. As with the other two applications in this chapter, this one uses the same RTMP address. You can develop as many different client-side applications as you want based on the same server-side application. By using different instance names in the RTMP address, you can automatically generate different server-side folders within a single application folder. Figure 2-9 shows what your application looks when you test it.

Figure 2-9. Both live local and recorded video playing



With this simple application, you can now record any video you want and play it back, or let someone else anywhere in the world play it back.

> You're going to be using your server-side folders a lot. To save time, create a shortcut to your applications folder so you don't have to dig through all of your folders and subfolders every time you need to access your server-side folders.

The settings for the microphone and camera use a good deal of bandwidth and are not optimized for getting the most out of your application. Rather, the settings give you the best possible video for limited use. The next chapter examines settings you can use with both the camera and microphone to optimize your application. Later chapters will detail different aspects of recording and playing back FLV files. To get started, make sure that you first get used to the basic elements used in the application in this chapter.

Chapter 3. Setting Your Camera and Microphone

3.1. Camera and Microphone Methods for Setting Parameters

One of the ongoing challenges for FMS3 developers is optimizing the settings for cameras and microphones. The Camera and Microphone classes provide methods for setting several different parameters. Depending on your media environment- your hardware (Webcams and microphones) and connection bandwidth-the settings will affect your application's performance.

Every Webcam has a "native" capacity, but whether that capacity can actually be realized is another matter. For example, most USB 2 and IEEE 1394 (FireWire) Webcams list a video capture speed of 30 frames per second (fps) and a video capture resolution of 640 x 480. Chances are that unless you have a pretty hefty bandwidth capability, lots of RAM memory, and a big fat processor, you're not going to be able to get that type of resolution in even a two-way audio/video chat.

Several Webcams provide the specifications for the "in-between" video capture resolutions.

The following are typical settings built into Webcams and digital video cameras:

- 160 x 120
- 176 x 144
- 320 x 240
- 360 x 240
- 352 x 288
- 640 x 480

The native video capture resolutions are generally set to a 4:3 (width:height) ratio; although it's not necessary to maintain that ratio, while you're getting started it's not a bad idea. FMS3 uses fairly sophisticated algorithms to resolve settings, but by using the 4:3 ratio you're going to be able to optimize what your camera sees. However, if you're interested in more details about how the settings work and using different aspect ratios, look at these resources

http://flash-communications.net/technotes/setMode/

http://www.peldi.com/fmswiki/index.php?title=Generating_Optimal_setQuality%2C_setMode_and_setRate_Parameters

3.1.1. Instantiating Camera and Microphone Objects

Unlike many other classes where an object instance is instantiated using a constructor, Camera and Microphone instances use the `get( )` method. The parameters for both `Camera.getCamera( )` and `Microphone.getMicrophone( )` refer to the drivers in one's local system. Usually, the process is nothing more than using the instantiating process without any parameters, such as,

```
var my_cam:Camera = Camera.getCamera( );
```

The instance `my_cam` refers to the currently selected driver to which your physical camera is connected. For example, if you're using a Logitech Pro 5000 camera plugged into a USB port the selected driver will likely be "QuickCam," the driver name provided with that particular camera.

However, if you have more than one camera connected to your computer and they have different drivers, you can select one or the other using a zero-based array number. If you have an IEEE 1394 (FireWire) driver and a USB driver, you can reference one or the other by its position in the Adobe Flash Player Settings window. For example, Figure 3-1 shows the currently selected driver at the top and then all six drivers in the system listed below. Their positions are:

1. Microsoft DV Camera and VCR
2. Creative WebCam Live! Pro #3
3. Creative WebCam Live! Pro #4
4. VHScrCap
5. Creative WebCam (VFW)
6. Creative WebCam Live! Pro (VFW)

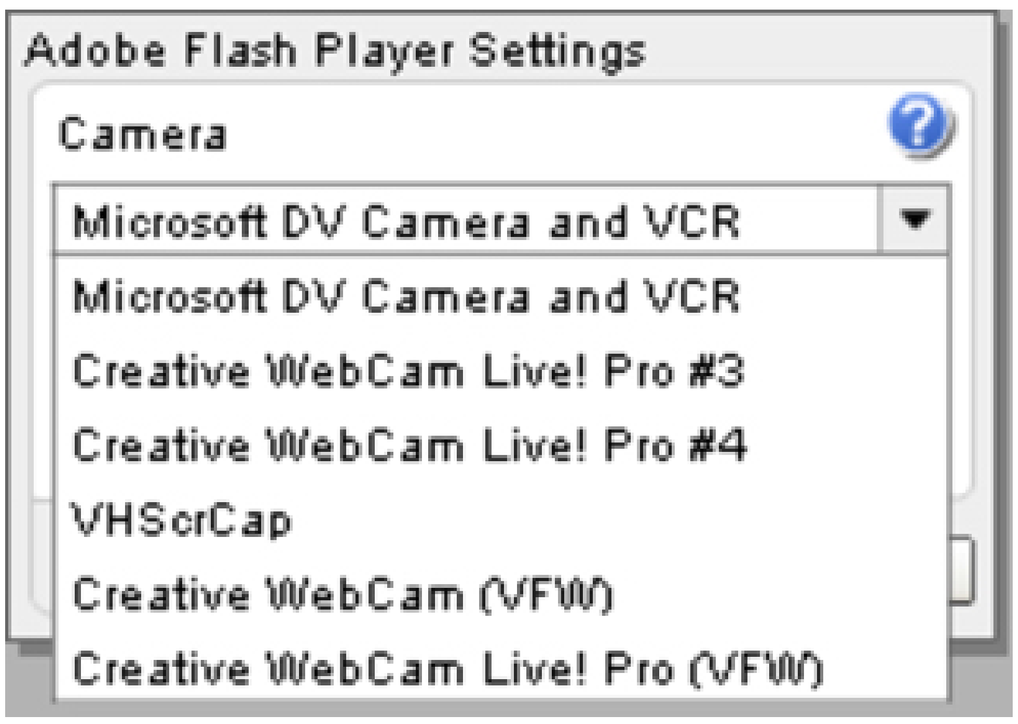
Thus, the statement,

```
var my_cam:Camera = Camera.getCamera(1);
```

sets the current camera to a camera connected to the Creative WebCam Live! Pro #3 driver. A driver with a number after it, such as #3 and #4, indicates that more than one of the same model camera is attached to the system. In this case, two Creative WebCam Live! Pro cameras are hooked up to the same computer. Two other Creative WebCam drivers are also installed, and they can be used by other Creative WebCams if they are

attached. The VHScrCap driver is a unique driver that has no real camera. It makes the screen a "camera." When it is selected, the screen becomes a "video" that can be used to show others what's on your screen. (You can get this driver for systems running Windows OS at www.hmelyoff.com/.)

Figure 3-1. Camera drivers in the Camera Tab of the Settings window



Generally, you should instantiate the camera using no parameters in the `Camera.getCamera( )` statement. This way, the user gets the default camera they're using. If you set it to another value, you risk the user not having a camera in the defined slot (for example, 4 or 5) or in an inappropriate slot, such as a digital video camera in a slot for an audio/video chat that would consume too much bandwidth.

The Microphone object works the same way. The `Microphone.getMicrophone()` statement gets the currently selected microphone driver and the microphone using that driver. Figure 3-2 shows five drivers in a Macintosh computer. One is for the microphone in an iSight camera, another for the built-in microphone, and the third accesses the microphone in a digital video camera.

Using either `Camera.names` or `Microphone.names`, you can access the array of driver names in your current computer.

3.1.2. Default Camera and Microphone Settings

When using Camera and Microphone classes for any FMS3 application, either explicitly set the parameters on each setting method or do nothing and use the default settings. Getting your application just right involves a good number of different settings. All of the methods available have default values. To get an idea of the settings in a minimal application, look at their default settings on a Macintosh, as shown in Figure 3-2:

Figure 3-2. Microphone drivers in the Microphone tab of the Settings window

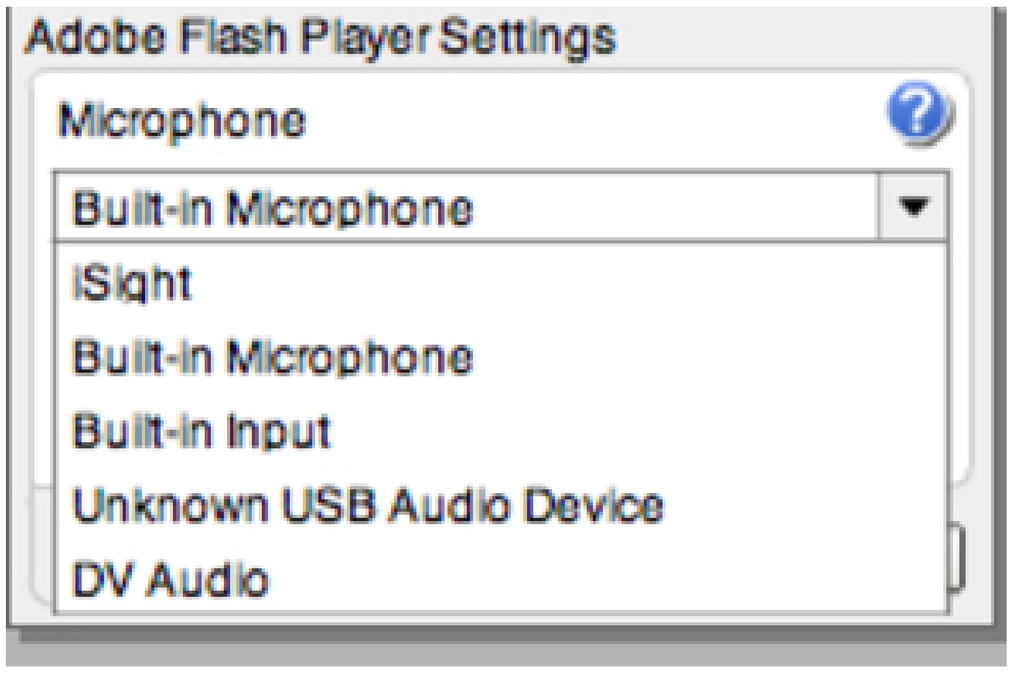


Table 3-1. Default Camera and Microphone settings

Method	Default Setting
Camera	
`Camera.setKeyFrameInterval( )`	15. This means that every 15th frame is a keyframe while the other frames are interpolated by the video compression algorithm. The lower the value, the more your computer has to work and the more bandwidth used
`Camera.setMode( )`	`160, 120, 15, true`. Width, height, and frames per second. The optional parameter specifies whether the application should favor the capture size or the frame rate
`Camera.setMotionLevel( )`	50, 2000. The first parameter refers to how much motion needs to be detected to fire an `activity` event. Acceptable values range from 0 to 100. The second parameter refers to the number of milliseconds (seconds/1000) that should elapse before the `activity` event is fired. Camera. `setMotionLevel` simply detects motion and does not affect bandwidth usage. Video is still sent even if `activity` is detected beyond the setting.
`Camera.setQuality( )`	16384, 0. The first parameter is the amount of bandwidth (in bytes) that can be used to generate optimum camera quality. Since bandwidth is generally measured in bits, the default bandwidth of 131,072 bits/second means that about 131kb of bandwidth is all that can be used to make your video look as good as possible. The second parameter is the quality of the video. A value of 0 means to do the best possible with the bandwidth value in the first parameter. The default settings ask the server, "Give me the best picture possible with 131 KB of bandwidth." You can set the quality from 0 to 100. With the exception of 0, the value indicates the quality you want from your video. A value of 100 is the best. If you set the first bandwidth parameter to 0 and the second parameter to a value other than 0, you will get the quality indicated by the second parameter. Thus, a setting of 0, 85 means that the server gives you all the bandwidth you need to keep the quality level at 85.
Microphone	

Method	Default Setting
Microphone.rate	8. The capture rate can be one of five kilohertz (kHz) values-5, 8, 11, 22, and 44. If your audio capture device does not support the assigned value, it jumps to the next level. So if you set your capture rate to 5 and that rate is not supported by your sound card, it would jump to 8 kHz.
Microphone.gain	50. This value controls the volume of your microphone. Technically, it refers to the amount to multiply the signal before sending it out. The values range from 0 (mute) to 100.
Microphone.setSilenceLevel( )	10, 2000. The first parameter specifies the amount of sound required to activate the microphone by invoking an activity event, and the second parameter is the number of milliseconds it takes for the microphone to turn off once the sound is determined to have stopped. As soon as the sound is above the silence level, the microphone turns on again.
Microphone.setUseEchoSuppression()	false. This setting reduces the effects of feedback from the speakers to the microphone. This setting can be made in both the Settings window on the Microphone tab as well as dynamically using ActionScript. If you're using external speakers instead of a headset for your sound, suppress the echo by setting the parameter to true using this method dynamically.

Chapter 3. Setting Your Camera and Microphone

3.1. Camera and Microphone Methods for Setting Parameters

One of the ongoing challenges for FMS3 developers is optimizing the settings for cameras and microphones. The Camera and Microphone classes provide methods for setting several different parameters. Depending on your media environment- your hardware (Webcams and microphones) and connection bandwidth-the settings will affect your application's performance.

Every Webcam has a "native" capacity, but whether that capacity can actually be realized is another matter. For example, most USB 2 and IEEE 1394 (FireWire) Webcams list a video capture speed of 30 frames per second (fps) and a video capture resolution of 640 x 480. Chances are that unless you have a pretty hefty bandwidth capability, lots of RAM memory, and a big fat processor, you're not going to be able to get that type of resolution in even a two-way audio/video chat.

Several Webcams provide the specifications for the "in-between" video capture resolutions.

The following are typical settings built into Webcams and digital video cameras:

- 160 x 120
- 176 x 144
- 320 x 240
- 360 x 240
- 352 x 288
- 640 x 480

The native video capture resolutions are generally set to a 4:3 (width:height) ratio; although it's not necessary to maintain that ratio, while you're getting started it's not a bad idea. FMS3 uses fairly sophisticated algorithms to resolve settings, but by using the 4:3 ratio you're going to be able to optimize what your camera sees. However, if you're interested in more details about how the settings work and using different aspect ratios, look at these resources

http://flash-communications.net/technotes/setMode/

http://www.peldi.com/fmswiki/index.php?title=Generating_Optimal_setQuality%2C_setMode_and_setRate_Parameters

3.1.1. Instantiating Camera and Microphone Objects

Unlike many other classes where an object instance is instantiated using a constructor, Camera and Microphone instances use the `get( )` method. The parameters for both `Camera.getCamera( )` and `Microphone.getMicrophone( )` refer to the drivers in one's local system. Usually, the process is nothing more than using the instantiating process without any parameters, such as,

```
var my_cam:Camera = Camera.getCamera( );
```

The instance `my_cam` refers to the currently selected driver to which your physical camera is connected. For example, if you're using a Logitech Pro 5000 camera plugged into a USB port the selected driver will likely be "QuickCam," the driver name provided with that particular camera.

However, if you have more than one camera connected to your computer and they have different drivers, you can select one or the other using a zero-based array number. If you have an IEEE 1394 (FireWire) driver and a USB driver, you can reference one or the other by its position in the Adobe Flash Player Settings window. For example, Figure 3-1 shows the currently selected driver at the top and then all six drivers in the system listed below. Their positions are:

1. Microsoft DV Camera and VCR
2. Creative WebCam Live! Pro #3
3. Creative WebCam Live! Pro #4
4. VHScrCap
5. Creative WebCam (VFW)
6. Creative WebCam Live! Pro (VFW)

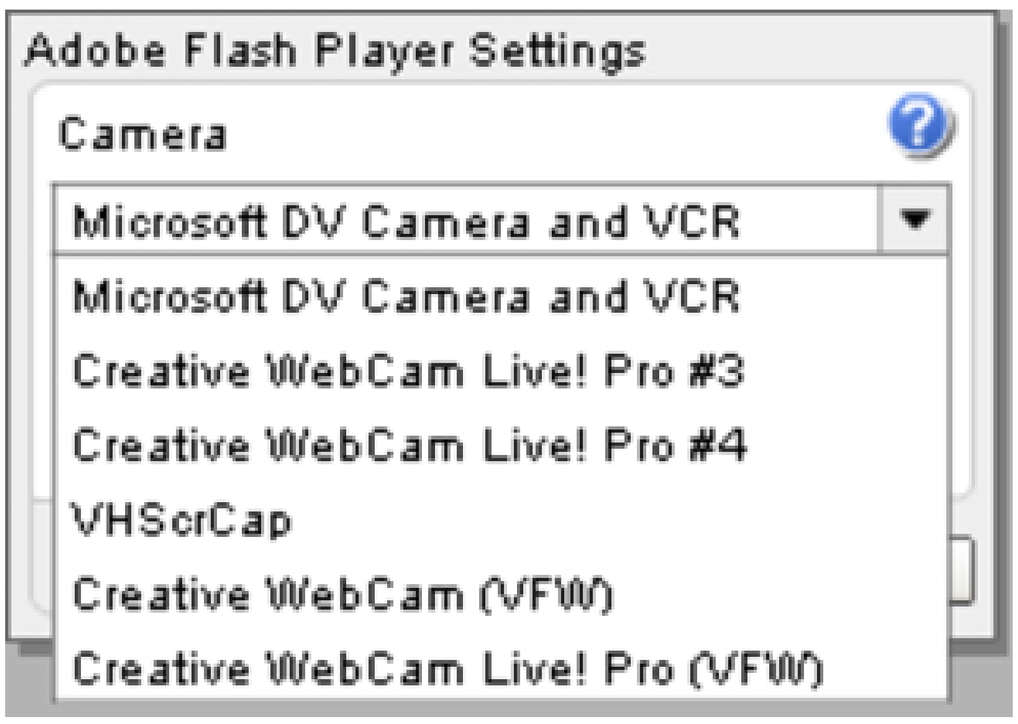
Thus, the statement,

```
var my_cam:Camera = Camera.getCamera(1);
```

sets the current camera to a camera connected to the Creative WebCam Live! Pro #3 driver. A driver with a number after it, such as #3 and #4, indicates that more than one of the same model camera is attached to the system. In this case, two Creative WebCam Live! Pro cameras are hooked up to the same computer. Two other Creative WebCam drivers are also installed, and they can be used by other Creative WebCams if they are

attached. The VHScrCap driver is a unique driver that has no real camera. It makes the screen a "camera." When it is selected, the screen becomes a "video" that can be used to show others what's on your screen. (You can get this driver for systems running Windows OS at www.hmelyoff.com/.)

Figure 3-1. Camera drivers in the Camera Tab of the Settings window



Generally, you should instantiate the camera using no parameters in the `Camera.getCamera( )` statement. This way, the user gets the default camera they're using. If you set it to another value, you risk the user not having a camera in the defined slot (for example, 4 or 5) or in an inappropriate slot, such as a digital video camera in a slot for an audio/video chat that would consume too much bandwidth.

The Microphone object works the same way. The `Microphone.getMicrophone()` statement gets the currently selected microphone driver and the microphone using that driver. Figure 3-2 shows five drivers in a Macintosh computer. One is for the microphone in an iSight camera, another for the built-in microphone, and the third accesses the microphone in a digital video camera.

Using either `Camera.names` or `Microphone.names`, you can access the array of driver names in your current computer.

3.1.2. Default Camera and Microphone Settings

When using Camera and Microphone classes for any FMS3 application, either explicitly set the parameters on each setting method or do nothing and use the default settings. Getting your application just right involves a good number of different settings. All of the methods available have default values. To get an idea of the settings in a minimal application, look at their default settings on a Macintosh, as shown in Figure 3-2:

Figure 3-2. Microphone drivers in the Microphone tab of the Settings window

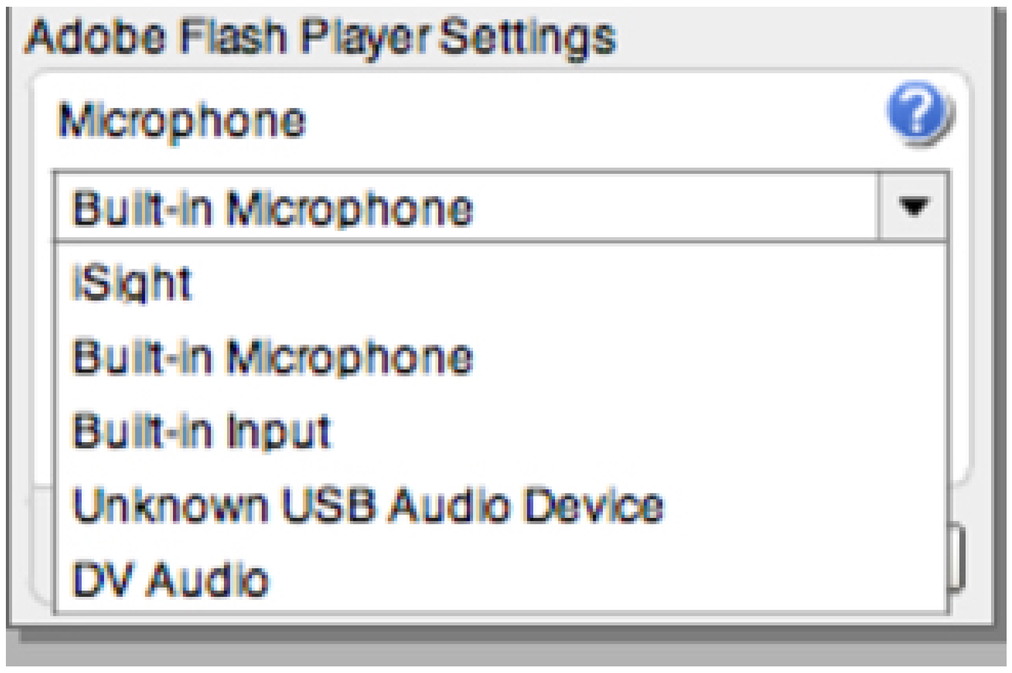


Table 3-1. Default Camera and Microphone settings

Method	Default Setting
Camera	
`Camera.setKeyFrameInterval( )`	15. This means that every 15th frame is a keyframe while the other frames are interpolated by the video compression algorithm. The lower the value, the more your computer has to work and the more bandwidth used
`Camera.setMode( )`	`160, 120, 15, true`. Width, height, and frames per second. The optional parameter specifies whether the application should favor the capture size or the frame rate
`Camera.setMotionLevel( )`	50, 2000. The first parameter refers to how much motion needs to be detected to fire an `activity` event. Acceptable values range from 0 to 100. The second parameter refers to the number of milliseconds (seconds/1000) that should elapse before the `activity` event is fired. Camera. `setMotionLevel` simply detects motion and does not affect bandwidth usage. Video is still sent even if `activity` is detected beyond the setting.
`Camera.setQuality( )`	16384, 0. The first parameter is the amount of bandwidth (in bytes) that can be used to generate optimum camera quality. Since bandwidth is generally measured in bits, the default bandwidth of 131,072 bits/second means that about 131kb of bandwidth is all that can be used to make your video look as good as possible. The second parameter is the quality of the video. A value of 0 means to do the best possible with the bandwidth value in the first parameter. The default settings ask the server, "Give me the best picture possible with 131 KB of bandwidth." You can set the quality from 0 to 100. With the exception of 0, the value indicates the quality you want from your video. A value of 100 is the best. If you set the first bandwidth parameter to 0 and the second parameter to a value other than 0, you will get the quality indicated by the second parameter. Thus, a setting of 0, 85 means that the server gives you all the bandwidth you need to keep the quality level at 85.
Microphone	

Method	Default Setting
Microphone.rate	8. The capture rate can be one of five kilohertz (kHz) values-5, 8, 11, 22, and 44. If your audio capture device does not support the assigned value, it jumps to the next level. So if you set your capture rate to 5 and that rate is not supported by your sound card, it would jump to 8 kHz.
Microphone.gain	50. This value controls the volume of your microphone. Technically, it refers to the amount to multiply the signal before sending it out. The values range from 0 (mute) to 100.
Microphone.setSilenceLevel( )	10, 2000. The first parameter specifies the amount of sound required to activate the microphone by invoking an activity event, and the second parameter is the number of milliseconds it takes for the microphone to turn off once the sound is determined to have stopped. As soon as the sound is above the silence level, the microphone turns on again.
Microphone.setUseEchoSuppression()	false. This setting reduces the effects of feedback from the speakers to the microphone. This setting can be made in both the Settings window on the Microphone tab as well as dynamically using ActionScript. If you're using external speakers instead of a headset for your sound, suppress the echo by setting the parameter to true using this method dynamically.

3.2. Minimalist Project

Even a minimal project requires sending out an audio/video stream and sending it back in again. The local video doesn't help at all to see the results of your setting. Thus, you're going to have to stream in the video you're sending out to simulate what the person receiving your stream will see. You don't have many ways to simulate different kinds of connections; if you want to try out different connection speeds, you'll have to enlist the help of some friends whose connections differ from yours. Still, you can get an idea of what happens with different settings using a LAN or streaming in and out of the same computer. You'll need the following classes and objects (client-side):

Classes and Methods

NetConnection

NetStream

Camera

setMode( )

setQuality( )

setKeyFrameInterval( )

setMotionLevel( )

Microphone

setMode( )

setQuality( )

setKeyFrameInterval( )

setMotionLevel( )

Video

This first minimalist project lets you hardcode the different parameter values for the camera and microphone and then view the results of your settings. ("Hardcode" refers to directly tying in the parameters using literal values.) Later in this chapter, you'll be able to experiment with a more robust application.

Before starting, let's look at the absolute minimum code required to set up your camera and microphone:

1. Import Camera, Microphone, Video, NetConnection, NetStream, NetStatusEvent and ObjectEncoding packages.

2. Instantiate Camera and Microphone objects and their key methods for setting their parameters. These are

used with your hardware to send out the digital video and sound. A video instance is instantiated by assigning a variable as a Video class instance.

3. Create a connection to the server (FMS3). This is done with the NetConnection class.

4. Create a `NetStream` instance to stream video to the server where it will be streamed back to view the video object.

3.2.1. Minimum Code to Set Up Camera and Microphone and Display Results

This code uses two video objects and two net stream objects. One stream captures the audio and video displayed in the local video and sends it out to the server. The other stream object brings the stream in and displays it in the second video. Look at the key camera and microphone settings. You can increase or decrease the values to provide a better picture and motion. A general rule of thumb is that as you increase the quality, you also increase the amount of bandwidth used to stream the audio and video. Likewise, as you decrease some aspect of quality, you decrease the bandwidth.

Open a new ActionScript 3.0 file, enter the following code, and save the file as MinCam.as:

Example 3-1. MinCam.as

Code View:

```
package
{
  import flash.display.Sprite;
  import flash.media.Camera;
  import flash.media.Microphone;
  import flash.media.Video;
  import flash.net.NetConnection;
  import flash.net.NetStream;
  //import flash.net.ObjectEncoding;
  import flash.events.NetStatusEvent;

  public class MinCam extends Sprite
  {
        private var cam:Camera;
        private var mic:Microphone;
        private var vid1:Video;
        private var vid2:Video;
        private var nc:NetConnection;
        private var nsOut:NetStream;
        private var nsIn:NetStream;
        private var rtmpNow:String;
        private var msg:Boolean;

        public function MinCam ()
        {
             //NetConnection.defaultObjectEncoding=flash.net.ObjectEncoding.AMF0;
             rtmpNow="rtmp://192.168.0.11/vid3";
             cam=Camera.getCamera();
             mic=Microphone.getMicrophone();

             //Camera Settings
             cam.setKeyFrameInterval (15);
             cam.setMode (240,180,15,false);
```

```
            cam.setMotionLevel (35,3000);
            cam.setQuality (40000 / 8,0);

            //Microphone Settings
            mic.gain =85;
            mic.rate=11;
            mic.setSilenceLevel (25,1000);
            mic.setUseEchoSuppression (true);

            //Video Setup
            vid1=new Video(cam.width,cam.height);
            vid2=new Video(cam.width,cam.height);
            addChild (vid1);
            vid1.x=10,vid1.y=20;
            addChild (vid2);
            vid2.x=vid1.width+15,vid2.y=20;

            //Attach local video and camera
            vid1.attachCamera (cam);

                  //Connect
                  nc=new NetConnection;
                  nc.connect (rtmpNow);
                  nc.addEventListener (NetStatusEvent.NET_STATUS,checkConnect);
            }

            //Create NetStream instances
            private function checkConnect (e:NetStatusEvent):void
            {
                  msg=e.info.code == "NetConnection.Connect.Success";
                  if (msg)
                  {
                        nsOut=new NetStream(nc);
                        nsIn=new NetStream(nc);
                        //NetStream
                        nsOut.attachAudio (mic);
                        nsOut.attachCamera (cam);
                        nsOut.publish ("camstream");
                        vid2.attachNetStream (nsIn);
                        nsIn.play ("camstream");

                  }
            }
      }
}
```

Many aspects of this script are crucial, but for getting started, the main important features are the microphone and camera settings. Before going to the next section, try experimenting with them to see how any changes you make affect the video in the right video window and the sound. (The settings were intentionally set so that you would get less than optimum output. See if you can improve the output by changing the parameter values.)

Figure 3-3 shows the expected output.

Figure 3-3. Local and streamed video

When testing your application, use different settings when you're looking at the output. As a rule of thumb, try reducing the settings to optimize bandwidth until you don't like the sound and video. Then revert to the last settings you had. However, setting decisions depends on many different parameters, as the next section discusses.

3.2.2. Setting Decisions

In setting your camera and microphone parameters, think of the settings as those for a giant pie-making machine. Imagine that you have a machine with a big pipe that pops out pies. The bigger the pipe, the more and better pies it can produce. (That's your bandwidth.) You can set the quality of your pies from "edible" to "fantastic." (That's your camera quality and microphone rate.) Your pies can range from tiny tarts to great big pies. (That's your camera mode width and height.) Further, you can produce a few pies at a time, requiring fewer resources from your machine, to lots and lots of pies, requiring more resources. (That's your fps.) Finally, you've got your pie eaters. With just a few pie eaters, it's easy to keep up with plenty of good pies, but as the number increases, you've got to lower the quality of the pies to make enough for everyone.

As flawed as that analogy may be, it gives you an idea of the trade-offs in creating an application using FMS3. If you have plenty of bandwidth, most of your other problems will diminish until the number of people using the application at the same time increases beyond what the bandwidth can handle.

To help see how this works, Adobe suggests the settings in Table 3-2 for the Camera quality. These settings show how the different trade-offs are made for a single setting. While factoring in these values, you have to consider the asynchronous speeds of most DSL and cable services. For example, one DSL provider advertises 1.5 Mbps (megabits per second) down speeds and up speeds of only 384 Kbps (kilobits per second). ("Down" speed refers to the speed at which bits are sent from the server to your computers, while "up" speed is the speed that your computer sends bits to the server.) So while you may have more than a megabit of bandwidth to receive data, your system can only send data out at a rate of less than half a megabit. Further, the location and type of server your data goes out to and comes in from will affect your performance. You can get a better idea of your Internet connection speeds at different test sites on the Web, for example, at www.speakeasy.net/speedtest/.

This site shows both your download and upload speeds, and you'll be able to plan your settings accordingly. To get an accurate idea of what you can expect to get, test your connection speed at different times during the day and with different locations. You will be surprised at the range of speeds. My cable modem connection ranged from 15 Mbps /733 Kbps (down/up) between Connecticut and NYC, to 5.6 Mbps/638Kbps between Connecticut and LA. While I found a huge difference in the download speeds, the upload speed stayed fairly consistent. The differences are not strictly a function of distance. One of my slowest speed tests was between Connecticut and Atlanta, resulting in 3.2 Mbps/725 Kbps. Ten minutes later, the same test between Connecticut and Atlanta showed 11.7 Mbps /732 Kbps. Lots of other factors impact your bandwidth, including the number and quality of

routers between you and the target, time of day, day of week and even special occasions such as emergencies that may impact the use of Internet traffic. While school is in session, far more students, teachers, and researchers use the Internet. Since the settings I'm using provide the upload speeds for the camera and microphone, I can expect pretty consistent output.

NOTE

One of the enduring dilemmas in FMS3 is the use of bytes for making and reading both client-side and server-side bandwidth settings. The problem is that bandwidth is measured in bits! (A byte is made up of 8 bits.) A 56k (kilobit) modem maxes out at 56,000 bits; not 56,000 bytes. This means that all settings-to make sense in bits-must be multiplied or divided by 8. So if you want to set your bandwidth to 100 Kbit, you must write 100000/8 in your settings. To help make things clearer, the examples use the format where values are divided by 8 to show the number of bits actually being used. On top of that, a kilobit is typically 1024 bits. However, bandwidth measurement is not a real kilobit but rather 1000 bits. So when you see 50 Kbit in relationship to bandwidth, it's just 50 x 1000 and not 50 x 1024 as it would be in other types of bit and byte measurement standards.

Table 3-2. Camera Image and Motion Quality

Speed (bits/second)	Image Quality	Motion Quality	Settings BW/Quality
56 Kbit	Lower	Higher	32000/8,0
56 Kbit	Higher	Lower	0,65
700 Kbit	Lower	Higher	96000/8,0
700 Kbit	Higher	Lower	0,90
3 Mbit	Lower	Higher	3200000/8,0
3 Mbit	Higher	Lower	0,100

Later chapters cover bandwidth and bandwidth detection more in discussing server-side script. By detecting the amount of bandwidth clients have, you can dynamically change settings to best accommodate these users.

3.2.3. Microphone Bandwidth Usage and Rate Settings

To get a more precise idea of how much bandwidth is consumed by different rate settings on the microphone, Table 3-3 provides a breakdown. The last column is the most important because it provides bits/second in terms that are actually used-so `Microphone.rate=8` generates 16 Kbit of bandwidth.

Table 3-3. Microphone rates and bandwidth usage

Microphone.rate	kHz	bytes/sec	bits/sec
5	5.512	1378	11.025 Kbit
8	8.000	2000	16 Kbit
11	11.025	2756	22.05 Kbit

Microphone.rate	kHz	bytes/sec	bits/sec
22	22.050	5513	22 44.1 Kbit
44	44.100	11025	88.2 Kbit

An easy way to remember the bps taken up by the rate setting is to double the rate value. For example, a rate of 5 is about 10 Kbits (actually its 11.025 Kbits) and a rate of 22 is about 44 Kbits. Keep in mind that the rate setting for the microphone is added to any other bandwidth-consuming settings used in the application.

3.2.4. Microphone Settings and Latency

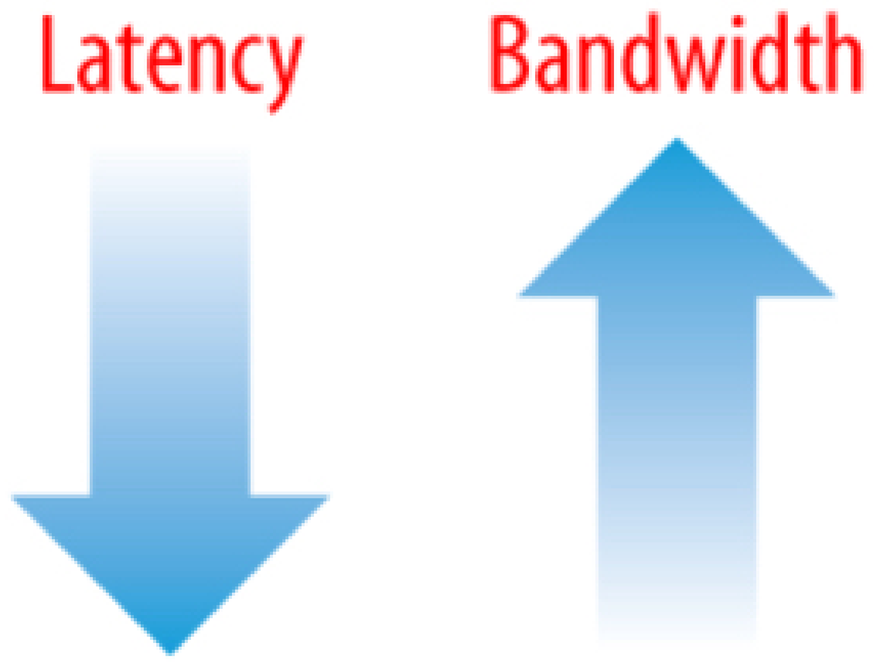
Besides the amount of bandwidth your application generates, another important consideration is the amount of latency your application has. *Latency* describes the difference between the time the sound arrives and when the video appears. If you've ever seen a film where the movement of an actor's mouth and his audio aren't quite synchronized because the speech was translated from one language to another, you have an idea of the effect of latency. (Those old Kung Fu movies from the 1970s seem to show this effect the best.) Figure 3-4 shows the relationship between the microphone rate setting and the latency between audio and video:

Table 3-4. Microphone Settings and Latency

Microphone.rates	Audio/video latency in milliseconds
5	50ms
8	32ms
11	25ms
22	12ms
44	6ms

By comparing Table 3-4 with Table 3-3 you can clearly see that higher rate settings use more bandwidth but also reduce latency. For example, if you compare the settings of 5 and 44 in Table 3-3, you will see that the higher setting uses about 8 times as many bits per second. However, if you look at Table 3-4, you will see that a rate setting of 5 has about 9 times the latency that a setting of 44. Figure 3-4 shows the simple yet clear relationship between latency and bandwidth usage:

Figure 3-4. Inverse relationship between latency and bandwidth usage



Like everything else in Flash Media Server 3, you will find trade-offs between bandwidth and the quality of one parameter or another. This particular relationship is counter-intuitive in certain respects. Some might think that the lower the bandwidth, the lower the latency because fewer packets are passed. However, the opposite is true.

3.3. Dynamically Testing Your Camera and Microphone Settings

To more easily see the effects of different settings, this next application uses minimum code for the actual settings, but it employs several different UI components for user input. The purpose of the application is two-fold. First, it allows you to explore the optimum values for setting your camera and microphone values dynamically. Second, it helps you understand how to use different UI components for making dynamic settings. Before getting started, look at Figure 3-5 and Figure 3-6, an "instance map," to see what components are used and their instance names. They include two movie clips with embedded video objects and a slider movie clip containing another movie clip used in changing volume.

This application uses the Stage to set up both TextInput and Button components. You need to change some settings to get the component instance names to be recognized by ActionScript 3.0, but you'll save a lot of time getting the components just where you want them on the Stage and in coding. To get going, follow these steps:

1. Open a new Flash file and create four layers, naming them from top to bottom, Text, TextInputs, Buttons and Background in the Timeline. Save the file as DyControls.fla..

2. Drag a Button, TextInput and Slider component from the Components panel to the Library panel.

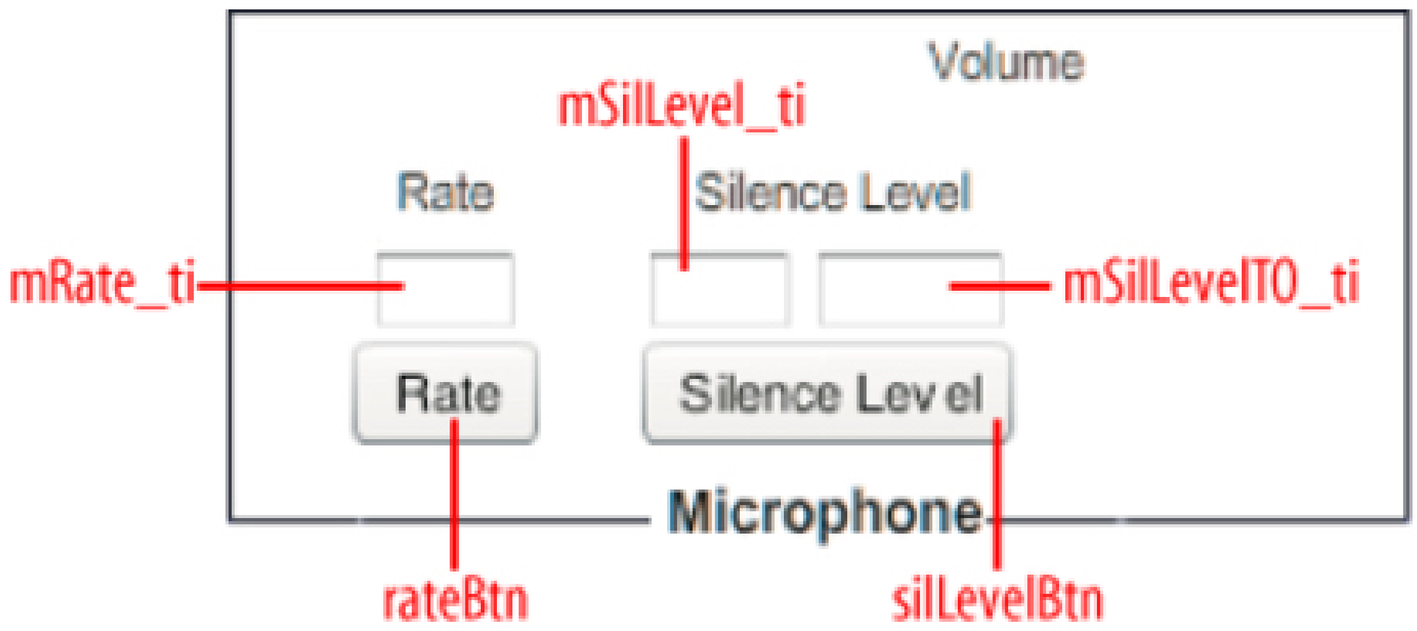
3. Select File Publish Settings Flash tab Settings Button and then next to Stage, deselect Automatically Declare Stage Instance. This crucial step enables ActionScript 3.0 to recognize the instance names.

4. Click the TextInputs layer, and drag ten TextInput components to the Stage, placing them as shown in Figure 3-5 and Figure 3-6. These figures also show the instance names to use for each component's instance name in the Property inspector.

Figure 3-5. Instance name for Camera input

5. Click the Buttons layer, and drag five Button components to the Stage, placing them as shown in Figure 3-5 and Figure 3-6. These figures also show the instance names to use for each component's instance name in the Property inspector.

Figure 3-6. Instance name for Microphone input



6. Click the Text layer, and using Figure 3-5, Figure 3-6, and Figure 3-7 as a guide, type in the appropriate labels.

7. Once all of your components and text labels are on the Stage, select the Background layer and enclose the camera settings in a red rectangle; then do the same for the microphone settings using a blue rectangle. This will help you quickly distinguish between the key settings.

8. In the Property inspector, type **DyControls** in the Document Class text box and save the FLA file. Your Stage is all set. Next, you will write the script.

9. Open a new ActionScript file and save it as DyControls.asin the same folder as the DyControls.flyfile.

10. In the DyControls.as file, enter the Example 3-2 script, and save the file again.

Example 3-2. DyControls.as

Code View:

```
package
{
  import fl.controls.Slider;
  import fl.events.SliderEvent;
  import flash.display.DisplayObject;
  import flash.display.Sprite;
  import flash.media.Camera;
  import flash.media.Microphone;
  import flash.media.Video;
  import flash.net.NetConnection;
  import flash.net.NetStream;
  //import flash.net.ObjectEncoding;
  import flash.events.NetStatusEvent;
  import flash.events.MouseEvent;
  import flash.events.ActivityEvent;

  public class DyControls extends Sprite
```

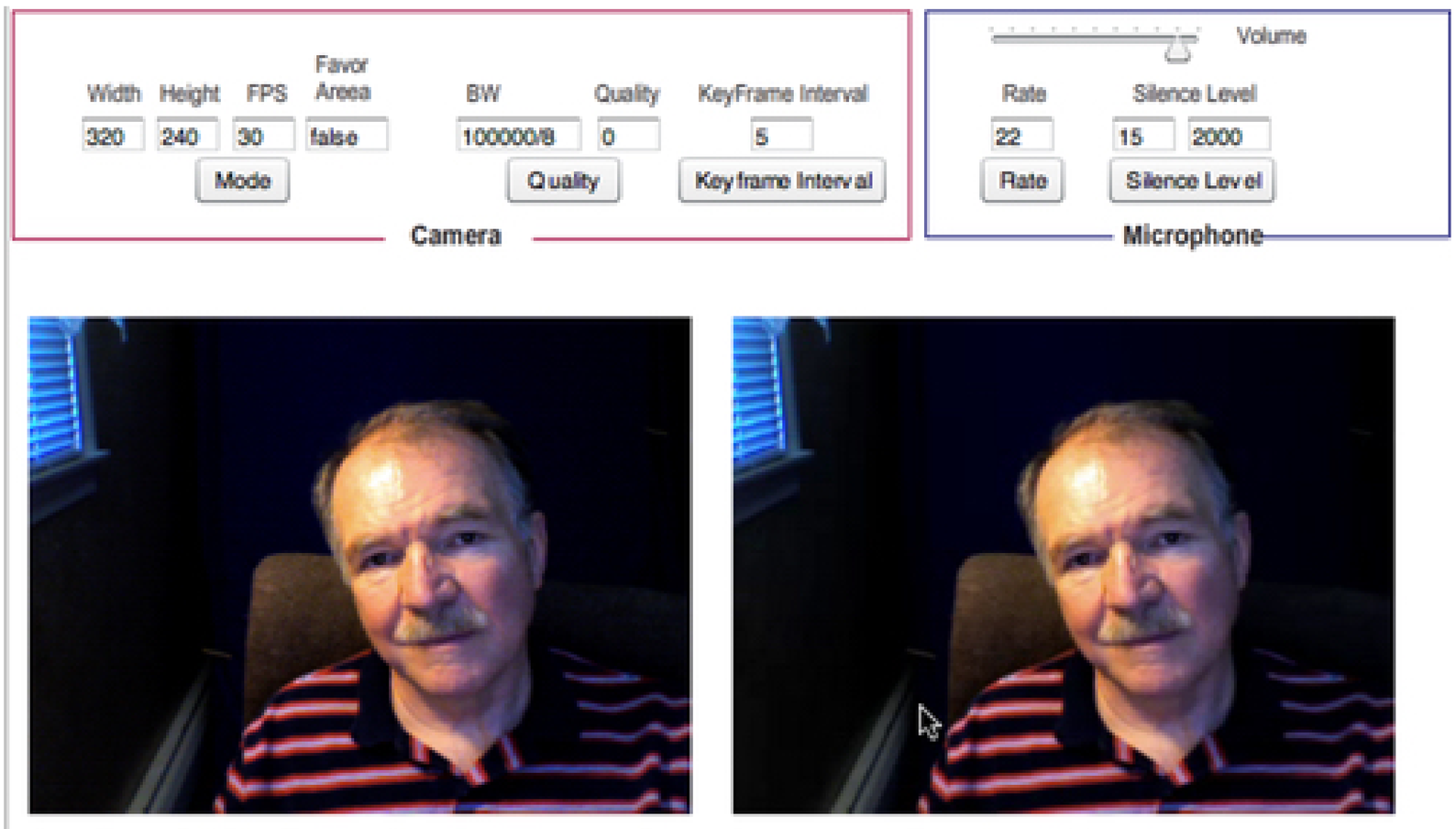
```
{
	private var cam:Camera;
	private var mic:Microphone;
	private var vid1:Video;
	private var vid2:Video;
	private var nc:NetConnection;
	private var nsOut:Object;
	private var nsIn:NetStream;
	private var rtmpNow:String;
	private var msg:Boolean;

	//Camera
	//*Mode
	private var cmode:Cmode;
	private var cModeW:Number;
	private var cModeH:Number;
	private var cModeF:uint;
	private var cModeA:Boolean;
	//*Quality
	private var cQualB:Number;
	private var cQualQ:uint;
	//*KeyFrame Internal
	private var keyFrame:uint;

	//Microphone
	//*
	private var gain_slider:Slider;
	private var mSilLevel:uint;
	private var mSilLevelTO:uint;

	public function DyControls ()
	{
	     //NetConnection.defaultObjectEncoding=flash.net.ObjectEncoding.AMF0;
	     rtmpNow="rtmp://192.168.0.11/vid3";
	     cam=Camera.getCamera();
	     mic=Microphone.getMicrophone();

	     //Default Camera Settings
	     cam.setKeyFrameInterval (15);
	     cam.setMode (240,180,15,false);
	     cam.setMotionLevel (35,3000);
	     cam.setQuality (200000/8,0);

	     //Microphone Settings
	     mic.gain =85;
	     mic.rate=11;
	     mic.setSilenceLevel (25,1000);
	     mic.setUseEchoSuppression (true);
	     gain_slider=new Slider();
	     gain_slider.width = 100;
	     gain_slider.snapInterval = 10;
	     gain_slider.tickInterval = 10;
	     gain_slider.maximum = 100;
	     gain_slider.value = 85;
	     addChild (gain_slider);
	     gain_slider.x=475,gain_slider.y=14;

	     //Video Setup
	     vid1=new Video(cam.width,cam.height);
```

```
            vid2=new Video(cam.width,cam.height);
            addChild (vid1);
            vid1.x=10,vid1.y=150;
            addChild (vid2);
            vid2.x=vid1.width+30,vid2.y=150;
            //Attach local video and camera
            vid1.attachCamera (cam);

            //Connect
            nc=new NetConnection;
            nc.connect (rtmpNow);
            nc.addEventListener (NetStatusEvent.NET_STATUS,checkConnect);

            //UI Event Listeners
            modeBtn.addEventListener (MouseEvent.CLICK,setCmode);
            qualBtn.addEventListener (MouseEvent.CLICK,setQual);
            keyFrameBtn.addEventListener (MouseEvent.CLICK,setKeyFrame);
            gain_slider.addEventListener (SliderEvent.CHANGE, volumeControl);
            rateBtn.addEventListener (MouseEvent.CLICK,setMicRate);
            silLevelBtn.addEventListener (MouseEvent.CLICK,setSilLevel);

            //Activity Detection
            cam.addEventListener (ActivityEvent.ACTIVITY, checkCamAction);
            mic.addEventListener (ActivityEvent.ACTIVITY, checkMicAction);
        }

        //Create NetStream instances
        private function checkConnect (e:NetStatusEvent):void
        {
            msg=e.info.code == "NetConnection.Connect.Success";
            if (msg)
            {
                nsOut=new NetStream(nc);
                nsIn=new NetStream(nc);
                //NetStream
                nsOut.attachAudio (mic);
                nsOut.attachCamera (cam);
                nsOut.publish ("camstream");
                vid2.attachNetStream (nsIn);
                nsIn.play ("camstream");
            }
        }

        //UI Callbacks
        private function setCmode (e:MouseEvent):void
        {
            cModeW=Number(cModeW_ti.text);
            cModeH=Number(cModeH_ti.text);
            cModeF=uint(cModeF_ti.text);
            cModeA=Boolean(cModeA_ti.text);
            cam.setMode (cModeW,cModeH,cModeF,cModeA);
            vid1.width=cam.width;
            vid1.height=cam.height;
            vid2.width=cam.width;
            vid2.height=cam.height;
            vid2.x=vid1.width+30,vid2.y=150;
        }

        private function setQual (e:MouseEvent):void
```

```
            {
                cQualB=Number(cQualB_ti.text);
                cQualQ=Number(cQualQ_ti.text);
                cam.setQuality (cQualB,cQualQ);
            }

            private function setKeyFrame (e:MouseEvent):void
            {
                keyFrame=Number(keyFrame_ti.text);
                cam.setKeyFrameInterval (keyFrame);
            }

            private function volumeControl (e:SliderEvent):void
            {
                mic.gain=gain_slider.value;
            }

            private function setMicRate (e:MouseEvent):void
            {
                mic.rate=Number(mRate_ti.text);
            }

            private function setSilLevel (e:MouseEvent):void
            {
                mSilLevel=Number(mSilLevel_ti.text);
                mSilLevelTO=Number(mSilLevelTO_ti.text);
                mic.setSilenceLevel (mSilLevel,mSilLevelTO);
            }

                private function checkMicAction (e:ActivityEvent):void
            {
                trace ("Microphone Activity detected");
            }

            private function checkCamAction (e:ActivityEvent):void
            {
                trace ("Camera Activity detected");
            }
        }
    }
```

When you test your application you can see the different qualities in picture, sound, and motion as you change the values in the TextInput boxes and click the button. Figure 3-7 shows some values and the results. Experiment with different settings to see their effects. All Camera settings are on the left and all Microphone settings on the right.

Figure 3-7. Running version of application



When you run the application, remember that the video you get is based on your system and connections, and where you're running your application. You should have a fairly consistent video in the left and right video, and an image and motion that will change as you change the setting. So if you're creating an audio/video application for low-end DSL users, use the application settings that generate less bandwidth and see if the results in the right video are acceptable. If you're targeting an audience with high-speed Internet connections, use those higher settings, and you should see better quality results. You can get higher quality and low bandwidth by the simple trick of making the image smaller.

3.4. Key Considerations

This last section briefly considers some elements of the settings you're making.

fps The number of frames per second acts as a multiplier on your application. A 320 x 240 frame has 320 x 240 pixels to ship across the Internet-about 77 Kbits. A setting of 15 fps = 15 x 77 Kbits -about 1.2 Mbits. With those same conditions, a setting of 30 fps would be about 2.4 Mbits. A smaller window of 160 x 120 requires about 19 Kbitsper frame, so a setting of 15 fps requires 285 Kbits. If the quality of the picture is important but the movement is not, such as in an audio/video chat where only your mouth is moving or you're sitting quietly listening and looking, you can gear down the fps setting to save bandwidth.

Quality: The quality (amount of compression) affects how good your image is going to look, and how much bandwidth it will consume. In turn, this will affect how much is used in each frame. Higher quality means more bits per frame.

Width and Height: Camera mode settings and video dimensions need not be the same. Your video screen can be set to 160 x 120, but if you set your camera mode to 320 x 240, you'll get a better picture. Conversely, if you set your mode to 160 x 120 and your video to 320 x 240, you'll get a bigger video picture but lower quality. However, if you deviate from the 4:3 ratio, you definitely want to match your camera mode settings with the video setting. For example, suppose you want to have a smaller picture with a 2:5 ratio. Your video dimensions should also have a 2:5 ratio. In the application, matching the video to the camera settings is done by the following simple algorithm:

```
video.width=camera.width;
video.height=camera.height
```

That makes it easy because the video is simply reflecting the camera's mode setting.

Silence and Motion Levels: Setting the silence level impacts the amount of bandwidth used. When the sound is below the silence level, no sound is sent. This is a very important setting because it actually affects the amount of bandwidth your application uses. Setting the camera's activity level works in a similar way, except that it does not affect bandwidth. Rather, when the motion from the camera reaches the level set for detecting motion, the `Camera.onActivity` event is launched and triggers any function you have set to fire when the activity is detected. The video is displayed regardless of the motion level setting.

With these elements in mind, practice with your settings both with the supplied applications in this chapter and with ones you build yourself. Testing is the only certain way of getting the most from your applications.

3.5. Adjusting Camera and Audio with Flash Media Encoder

Another way to set up and test your adjustments with the camera and microphone is to use the Flash Media Encoder (FME) with a live trial. The importance of testing your streams with a live trial is that when you have a live broadcast, what you see and what the user sees may be very different. Using FME, you can both see the input (local) and the output (streaming). You can use all of the built-in format adjustments in FME. Likewise, you can test your output on another system to be sure everything is the way you want.

In this next example, a digital video camera is connected, and FME provides the native sizes of the camera. A Canon Optura 100 was used in the test with a setting of 360 x 240. This setting is native the camera, and while it does not adhere to the 4:3 ratio but uses a 3:2 ratio instead. To see how the DV camera looks on a remote site, a Macintosh player application displays the output sent by the FME. Example 3-3 provides the listing for the remote player:

Example 3-3. FMElive.as

Code View:

```
package
{
  import flash.display.Sprite;
  import flash.net.NetConnection;
  import flash.net.NetStream;
  import flash.events.NetStatusEvent;
  import flash.media.Video;

  public class FMElive extends Sprite
  {
        private var good:Boolean;
        private var vid:Video;
        private var nc:NetConnection;
        private var ns:NetStream;
        private var rtmpNow:String;
        private var metaSniffer:Object;

        public function FMElive()
        {
             setVid();
             rtmpNow="rtmp://192.168.0.11/fme/viewer";
             nc=new NetConnection();
             nc.addEventListener(NetStatusEvent.NET_STATUS, checkConnect);
             nc.connect(rtmpNow);
        }

        private function getMeta (mdata:Object):void
        {
             //Live
        }

        private function checkConnect(e:NetStatusEvent):void
        {
             good=(e.info.code=="NetConnection.Connect.Success");
             if(good)
             {

                  ns=new NetStream(nc);
                  vid.attachNetStream(ns);
```

```
                ns.play("liveFME");
                metaCheck();
            }
        }

        private function setVid():void
        {
            vid=new Video();
            vid.x=95;vid.y=80;
            vid.width=360,vid.height=240;
            addChild(vid);
        }

        private function metaCheck():void
        {
            metaSniffer=new Object();
            ns.client=metaSniffer;
            metaSniffer.onMetaData=getMeta;
        }

    }
}
```

Create an FLA file with the Document Class set to *FMElive*. In the server-side applications folder, add a new folder named *fme*.

Launch your Flash Media Encoder selecting any device driver. Set the FMS URL to:

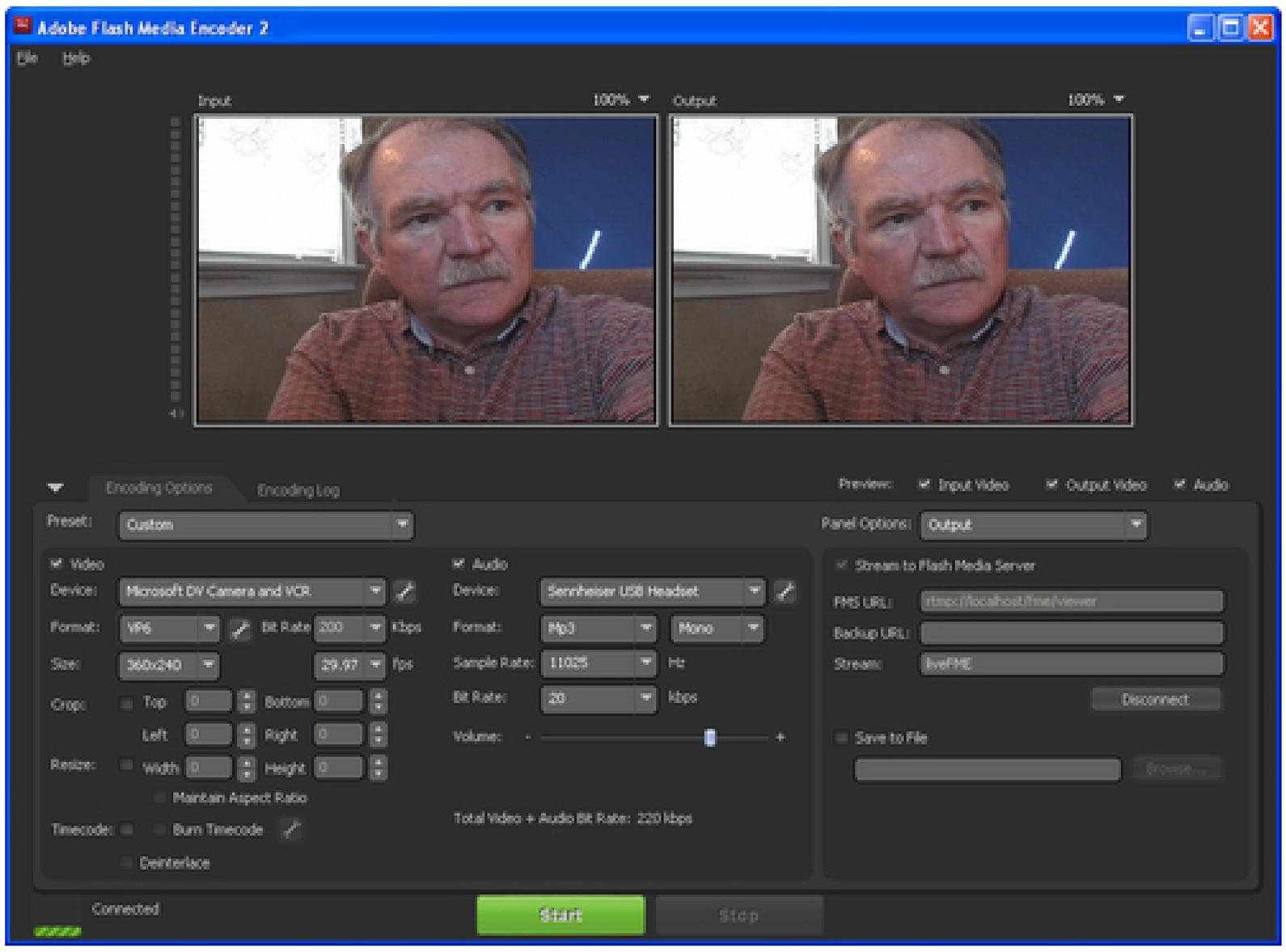
```
rtmp://localhost/fme/viewer
```

Set the Stream name to:

```
liveFME
```

Leave the Save to File unchecked and the name for the file blank. Finally make sure that the Stream to Flash Media Server box is checked-it's right above the FMS URL window. Figure 3-8 shows sample settings. Your Device window for both Video and Audio are likely to be different, as are any default settings. However, as long as the FMS URL and Stream names are those specified, you will be able to stream live video and audio from your Flash Media Encoder.

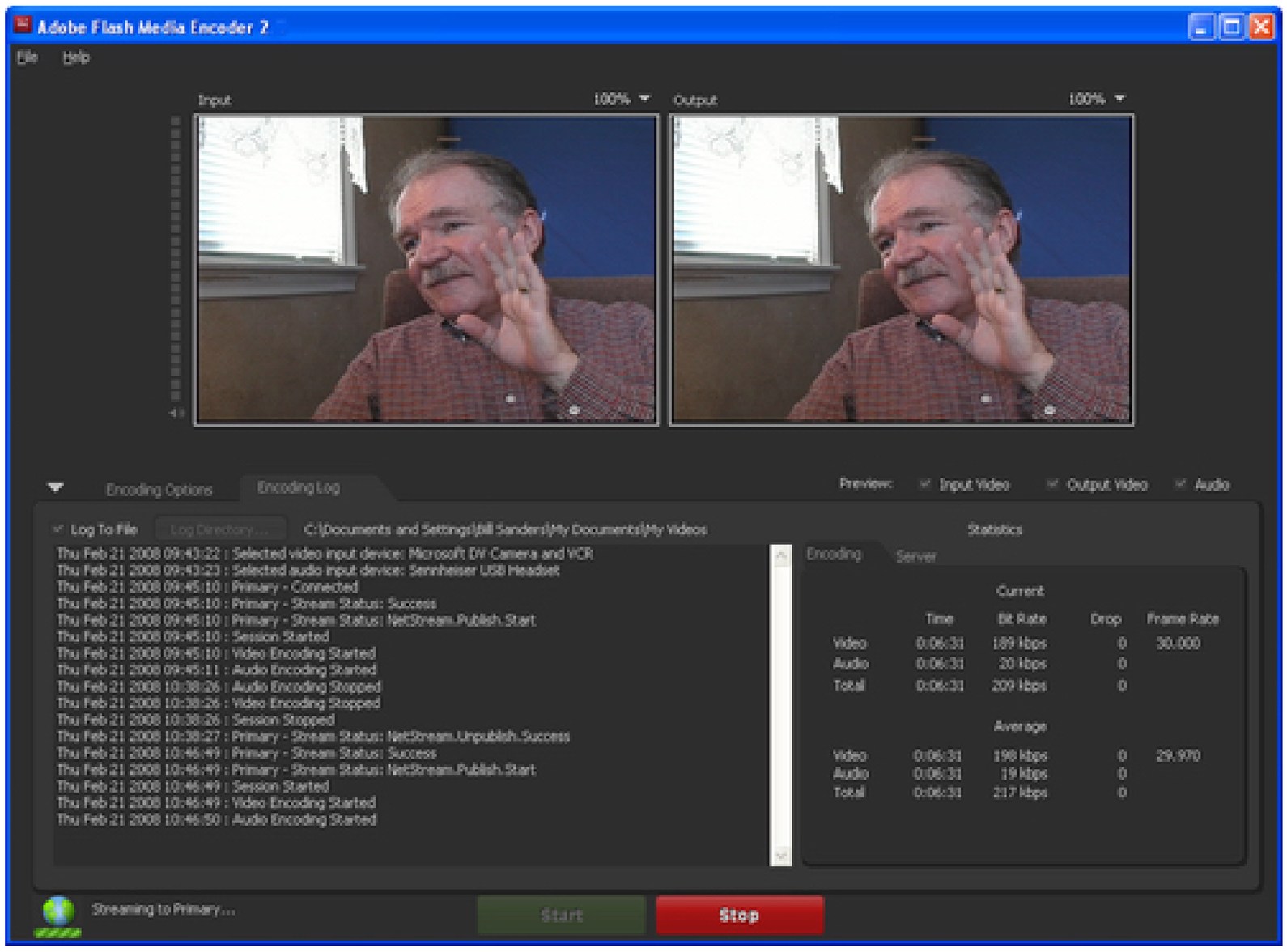
Figure 3-8. Flash Media Encoder settings prior to starting stream



Once all of your settings are ready, first click the Connect button, and if successful, you will see the Connect button change to Disconnect as Figure 3-8 shows.

As soon as you click the green Start button, you will begin streaming. Open the FMElive application and you should be able to see streamed video from the FME. Figure 3-9 is a collage showing the FME and the FMElive application displaying the streamed video:

Figure 3-9. Live stream from FME displayed locally and remotely



In this particular test, the stream was sent to the application on a Macintosh. As you will see, the output image in the right video window on the FME is displayed on the remote site. Generally, you will be able to see better video in the left FME video window because it displays what is coming directly from the camera and microphone.

Once you begin streaming, you will see the log of events in the left bottom panel and in the right panel, the statistics of use. The Statistics shows information for current and average bit rate, frame rate and the number of dropped frames. If you have the Log to File option checked, the log in the left panel is saved to a text file. The log can help you both understand the connection sequence and debug any problems.

To make adjustments to the Encoding Option, you must first stop the stream and then re-start it. You *do not* have to disconnect. Try changing the different settings, and when you arrive at the ones that work best for you system, write them down. Later when coding another application, you can then use them without having to go through the same process. If you like, you can use the Flash Media Encoder for a broadcast station, broadcasting to users through the FMElive application.

Chapter 4. Nonpersistent Client-Side Remote Shared Objects

4.1. Sharing Data on Multiple Connections

The first time I encountered the remote shared object I thought it had to be the most important contribution to the Internet since the invention of the browser. That opinion has changed little over the years, especially in the case of any application that needs to send data and interact with others in a "live" situation.

So what are shared objects and what are they used for? Shared objects are objects that can be displayed and controlled on one browser and seen on another. So if you have an application that displays text using a text input field, you can display that text in both your own browser as well as in all others that are looking at the same application. Besides text, you can move movie clips on one screen, and they move on other screens. You can click a button on one browser to launch an event on all browsers using shared objects.

When a shared object is set up in a Flash Media Server 3 application, it sets up a "subscription" that others who have launched the same application automatically subscribe to. Think of shared objects as magazines and everyone who logs on gets a subscription to the magazine. The "magazines" are delivered in the form of real-time data. So, if you're sending (sharing) the text in a text field, as soon as you enter the data, it's seen on the remote computer.

4.1.1. Persistent and Nonpersistent Shared Objects

Nonpersistent shared objects are something like variables. When you quit an application, all of the values in the variables disappear. They do not persist. In the same way, the data in a nonpersistent shared object is lost as soon as users leave the application.

Persistent shared objects are more like data in a database. The data from persistent shared objects is stored on the server's hard drive, and hence, persist. When you have data that you need to access from different sites, persistent shared objects are very handy. However, this chapter deals only with nonpersistent shared objects. In Chapter 8, where more advanced shared objects concepts are discussed, both server-side and persistent shared objects will be examined.

4.1.2. Uses of Nonpersistent Remote Shared Objects

While any list of nonpersistent remote shared objects will fall far short of their possible uses, the following gives you an idea of how they can be used:

- Remotely controlled slide show.

- Text chat.
- Games with opponents at different sites.
- DJ application where songs are remotely selected and played.
- A doorbell.
- A live online auction.
- Customer service application.

With these applications in mind, let's look at how to create and use shared objects with FMS3.

Chapter 4. Nonpersistent Client-Side Remote Shared Objects

4.1. Sharing Data on Multiple Connections

The first time I encountered the remote shared object I thought it had to be the most important contribution to the Internet since the invention of the browser. That opinion has changed little over the years, especially in the case of any application that needs to send data and interact with others in a "live" situation.

So what are shared objects and what are they used for? Shared objects are objects that can be displayed and controlled on one browser and seen on another. So if you have an application that displays text using a text input field, you can display that text in both your own browser as well as in all others that are looking at the same application. Besides text, you can move movie clips on one screen, and they move on other screens. You can click a button on one browser to launch an event on all browsers using shared objects.

When a shared object is set up in a Flash Media Server 3 application, it sets up a "subscription" that others who have launched the same application automatically subscribe to. Think of shared objects as magazines and everyone who logs on gets a subscription to the magazine. The "magazines" are delivered in the form of real-time data. So, if you're sending (sharing) the text in a text field, as soon as you enter the data, it's seen on the remote computer.

4.1.1. Persistent and Nonpersistent Shared Objects

Nonpersistent shared objects are something like variables. When you quit an application, all of the values in the variables disappear. They do not persist. In the same way, the data in a nonpersistent shared object is lost as soon as users leave the application.

Persistent shared objects are more like data in a database. The data from persistent shared objects is stored on the server's hard drive, and hence, persist. When you have data that you need to access from different sites, persistent shared objects are very handy. However, this chapter deals only with nonpersistent shared objects. In Chapter 8, where more advanced shared objects concepts are discussed, both server-side and persistent shared objects will be examined.

4.1.2. Uses of Nonpersistent Remote Shared Objects

While any list of nonpersistent remote shared objects will fall far short of their possible uses, the following gives you an idea of how they can be used:

- Remotely controlled slide show.

- Text chat.
- Games with opponents at different sites.
- DJ application where songs are remotely selected and played.
- A doorbell.
- A live online auction.
- Customer service application.

With these applications in mind, let's look at how to create and use shared objects with FMS3.

4.2. Instantiating Remote Shared Objects

This chapter will demonstrate three different remote shared objects with minimal code. The first is a pointer that lets you point to any location on another person's page. This kind of pointer can come in handy if you're making a presentation to a remote location and you have to point out different things to the audience. Second, you will see how to make a minimal text chat where the text entered into an input text box shows up on all users' screens. And third, using a shared movie clip, you will see how to open Web pages on remote viewers' screens. These are all minimal examples, but they show how you can do different things with remote shared objects. The rest is up to your imagination.

When using a class, you can scope the shared object outside the constructor, just as you would any other variable.

```
private var so:SharedObject;
```

Instantiating a remote shared object, though, is a little different than most class instances. Figure 4-1 shows the general format for instantiating a shared object.

Figure 4-1. Instantiating SharedObject class

Instead of using the `new` statement, the `getRemote()` method creates the remote shared object. Using the same data type (`SharedObject`), you can also create a local shared object with the method, `getLocal()`. Given the focus on FMS3, you will not find any examples using local shared objects in this book.

4.2.1. Setting Up Shared Object Storage: Slots

When you set up a shared object instance, you need to set up a storage system based on the shared object's properties. Unlike earlier versions of Flash Communication Server and Flash Media Server, Flash Media Server 3 uses the `SharedObject.setProperty()` method to assign values to a shared object property. By assigning different shared object properties you can create slots using most kinds of data types. The following shows a simple property slot:

```
var memberName:String= name_txt.text;
so.setProperty("cliName", memberName);
```

The shared object has a data property, `cliName`. So it now has a storage slot for a string name that can be shared with others connected to the same application. In the same way, you can set up as many slots as your

application needs, using the appropriate data type for the data to be stored. Whenever the shared object property changes, it fires a `SyncEvent` to be used by an event handler that updates the information for all connected to the same shared object.

4.2.2. Inspecting a Shared Object

To get a better idea of what's going on in a shared object, look at its contents. It should appear something like the contents of an array or any object that has more than a single element. Add a folder to your server-side folders named basicSO; this folder will be used for all of the examples in this chapter. In the following example, several different types of data are added to the slots. The properties include literals as well as variables to illustrate the range of data that can be put into shared objects.

4.2.2.1. The SyncEvent Class

When you use remote shared objects that are used by other clients, an important class is SyncEvent. As the name implies, the event is used in synchronizing events on your browser with those on other browsers using the same remote shared objects. By using the `SYNC` constant, as soon as the client connects to a remote shared object, a sync event fires up. Whenever a client changes a shared object's data property, the action invokes a sync event. However, because different kinds of actions related to shared objects generate events, you need a way to determine the nature of the event.

4.2.2.2. The **`changeList`** Property

In ActionScript 3.0, one of the key properties in successfully dealing with events generated by actions related to shared objects is the `SyncEvent.changeList` property. The `changeList` property is an array that stores shared object properties. A key property of the `changeList` property is the `code` property that indicates what type of event occurred. When first connected to a shared object, the code generates a `code` value of "clear." This code value can be used to help track the number of users connected to the same shared object.

When a remote user changes the value of a shared object, two `code` events are generated. A "success" value indicates that the client's changes were successfully made, and a "change" value indicates that someone else changed the shared object. So the client who makes the change receives a "success" code, and everyone else connected to the same shared object gets a "change" code value. For this first example, both the "clear" and "success" states are used to direct the program to methods for creating slots for and assigning values to a shared object, and then for showing the context of those slots.

To see how all of this works, you can trace this next application through these key operations:

- Makes a `NetConnection` (`nc`) in the constructor function (`BasicSO`). This generates a `NetStatusEvent`.

- Creates a shared object in the `doSO` function and connects the shared object to the `NetConnection` (`nc`) instance generating a sync event.

- With the `seeSo` function, determines the nature of the event using the `SyncEvent.changeList` property. It uses the zero array element in the `changeList` property rather than iterating through the whole array. A finding of "clear" reports that the shared object connection has been established and calls the function to set the shared object properties. If it finds "success," it shows the changes that have been made on your client.

This application's purpose is to help you understand something about how shared objects work, but as designed, it focuses on both the steps in the process and how to change values in shared object. At this stage, though, it doesn't share anything remotely. The following steps show how to set it up:

1. Create a new Flash file and save it as BasicSO.fla.

2. In the Document Class window in the Property inspector, type **`BasicSO`** and resave the file.

3. Open a new ActionScript file and save it as BasicSO.as.

4. In the BasicSO.as file, enter the script in Example 4-1, and save the file again.

Example 4-1. BasicSO.as

Code View:

```
package
{
  import flash.display.Sprite;
  import flash.net.SharedObject;
  import flash.events.NetStatusEvent;
  import flash.events.SyncEvent;
  import flash.net.NetConnection;
  //import flash.net.ObjectEncoding;

  public class BasicSO extends Sprite
  {
        private var test_so:SharedObject;
        private var nc:NetConnection;
        private var monthDay:Date;
        private var good:Boolean;

        public function BasicSO ()
        {
             //NetConnection.defaultObjectEncoding=flash.net.ObjectEncoding.AMF0;
             //Connect
             var rtmpNow:String="rtmp://192.168.0.11/basicSO";
             nc=new NetConnection;
             nc.connect (rtmpNow);
             nc.addEventListener (NetStatusEvent.NET_STATUS,doSO);
        }

        private function doSO (e:NetStatusEvent):void
        {
             good=e.info.code == "NetConnection.Connect.Success";
             if (good)
             {
                  //Set up shared object
                  test_so=SharedObject.getRemote("test",nc.uri,false);
                  monthDay=new Date;
                  test_so.connect (nc);
                  test_so.addEventListener (SyncEvent.SYNC,seeSO);
             }
        }

        private function seeSO (se:SyncEvent):void
        {
```

```
            trace (se.changeList[0].code);
            switch (se.changeList[0].code)
            {
                  case "clear" :
                        loadSO ();
                        break;

                  case "success" :
                        showSO ();
                        break;
            }
      }

      private function loadSO ():void
      {
            test_so.setProperty ("city","Katmandu");
            test_so.setProperty ("dayOfMonth",monthDay.getDate());
            test_so.setProperty ("verity",true);
            test_so.setProperty ("bunch",250);
      }

            }

            private function showSO ():void
            {
                  trace (test_so.data.city);
                  trace (test_so.data.dayOfMonth);
                  trace (test_so.data.verity);
                  trace (test_so.data.bunch);
            }
        }
}
```

5. Test the file by choosing Control Test Movie.

When you run the example, the Output window shows:

```
clear
success
Katmandu
15
true
250
```

As you can see, shared objects take on the basic data types in ActionScript as other variables and properties, and they can be extracted in the same way as other objects and variables.

4.2.3. Syncing Shared Objects

Whenever the value of any shared object property attribute, or "slot," changes, that change must be broadcast to all of those connected. The event handler for this process is `SyncEvent.SYNC,callback`. Each time any data attribute changes, a change event triggers `SyncEvent.SYNC,callback`, and in this way, all of the changed

attributes in all of the connections are updated. The general format is the following:

```
share_so.addEventListener (SyncEvent.SYNC,syncSO);
....
private function syncSO(event: SyncEvent)
{
   //Update variables with so.data property attribute values
   variable1= my_so.data.attribute1;
   variable2= my_so.data.attribute2;
   variable3= my_so.data.attribute3;
}
```

The process for this change is fairly straightforward. Somewhere in the script a user changes one of the slots in the shared object data property. As soon as that property is changed, it triggers the `SyncEvent.SYNC,callback`; and when that happens, the code in the `SyncEvent.SYNC,callback` container updates the changes throughout the connections in the application.

For the whole process to work correctly, you need to link the shared object and the connection to the server, `SharedObject. connect(NetConnection)`. To be able to work with remote shared objects meaningfully, you will need a line that connects the shared object instance to FMS3 using the `NetConnection` instance as a parameter. For example:

```
billz_so.connect(nc);
```

makes the necessary connection for the shared object instance `billz_so` to send and receive the changes to shared objects using the `NetConnection` instance `nc`. In all of the sample projects in this chapter, watch for the `SharedObject. connect(NetConnection)` statement, and don't ever forget to include it in your applications. (When you debug your code, often you will find that the shared object connection has been left out. Without it, your shared objects will not run correctly.)

4.3. Minimalist Project for Shared Movie Clip

To see how shared objects work with a remote page, this first minimalist project uses a movie clip as a shared object. This example shows how to set up and move a movie clip around the Stage-your Stage and the Stage of anyone connected to the same shared object. In creating applications that have online slide shows that work something like a PowerPoint presentation, you need all viewers to see the pointer move to the location onscreen where you want them to focus. Using an island movie clip as a pointer, this application has island names in each of the Stage's four corners. The four names are simply positions on the Stage that let you see that when you move the pointer onscreen, it will go to the same positions on all screens connected to the application. You'll use these classes and objects for the application:

Classes

NetConnection

SharedObject

NetStatusEvent

SyncEvent

MouseEvent

Objects

Movie Clip (1)

Follow these steps to walk through the project:

1. Open a new Flash document and save it as MinSOmc.fla.

2. In the four corners of the Stage, use the Text tool to type the name of four different islands, as shown in Figure 4-2.

3. Draw the image of an island as shown in Figure 4-2 with the drawing tools. (Alternatively, draw an arrow pointer or anything else.)

4. Select the shape and press F8 to open the Convert To Symbol dialog box. Type **`Island`** in the Name text box and select Movie Clip as the Type. Turn on the Export for ActionScript checkbox. You should now see the class name Island in the Class window with the Base class, flash.display.MovieClip. Click OK and save the file.

5. Delete the movie clip from the Stage. It will still be in the Library panel as a class, and you can reference it just as any other class.

6. Open a new ActionScript file and save it as MinSOmc.as.

7. Add the code in Example 4-2, and save the file again.

Example 4-2. MinSOmc.as

Code View:

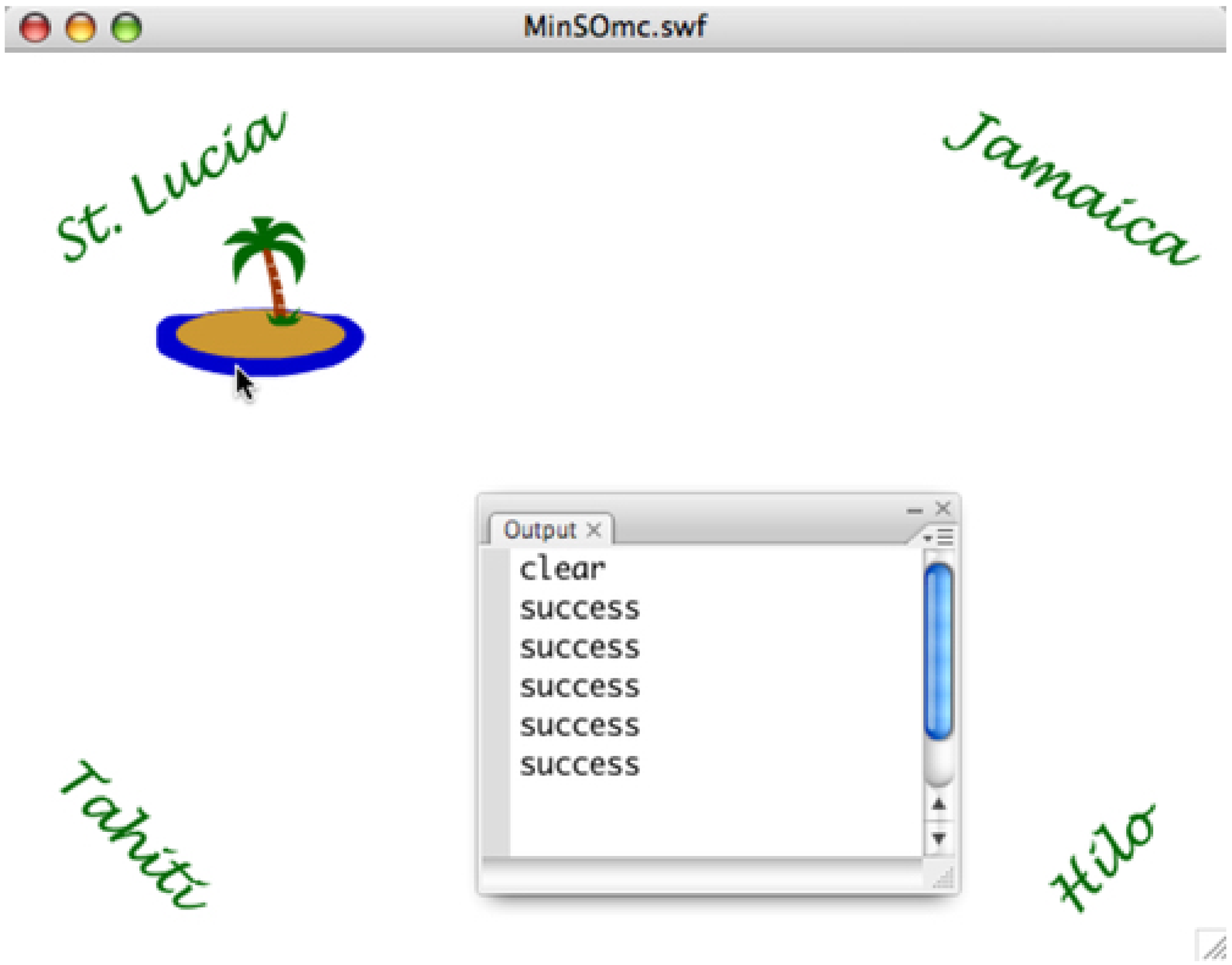
```
package
{
  import flash.display.Sprite;
  import flash.net.SharedObject;
  import flash.events.NetStatusEvent;
  import flash.events.SyncEvent;
  import flash.net.NetConnection;
  import flash.events.MouseEvent;
  //import flash.net.ObjectEncoding;

  public class MinSOmc extends Sprite
  {
        private var pointer_so:SharedObject;
        private var nc:NetConnection;
        private var good:Boolean;
        private var island:Island;

        public function MinSOmc()
        {
              //NetConnection.defaultObjectEncoding=flash.net.ObjectEncoding.AMF0;
              //Connect
              var rtmpNow:String="rtmp://192.168.0.11/basicSO";
              nc=new NetConnection  ;
              nc.connect (rtmpNow);
              nc.addEventListener (NetStatusEvent.NET_STATUS,doSO);
              island=new Island  ;
              addChild (island);
              island.x=200;
              island.y=200;
              island.addEventListener (MouseEvent.MOUSE_DOWN,beginDrag);
              island.addEventListener (MouseEvent.MOUSE_UP,endDrag);
        }

        private function doSO (e:NetStatusEvent):void
        {
              good=e.info.code == "NetConnection.Connect.Success";
              if (good)
              {
                    //Shared object
                    pointer_so=SharedObject.getRemote("point",nc.uri,false);
                    pointer_so.connect (nc);
                    pointer_so.addEventListener (SyncEvent.SYNC,doUpdate);
              }
        }
```

```
        private function doUpdate (se:SyncEvent):void
        {
             for (var cl:uint; cl < se.changeList.length; cl++)
             {
                  trace(se.changeList[cl].code);
                  if (se.changeList[cl].code == "change")
                  {
                       switch (se.changeList[cl].name)
                       {
                            case "xpos" :
                                 island.x=pointer_so.data.xpos;
                                 break;
                            case "ypos" :
                                 island.y=pointer_so.data.ypos;
                                 break;
                       }
                  }
             }
        }

        private function beginDrag (e:MouseEvent)
        {
             island.addEventListener (MouseEvent.MOUSE_MOVE,moveMc);
             island.startDrag ();
        }

        private function endDrag (e:MouseEvent)
        {
             island.stopDrag ();
        }

        private function moveMc (e:MouseEvent)
        {
             e.updateAfterEvent ();
             pointer_so.setProperty ("xpos",island.x);
             pointer_so.setProperty ("ypos",island.y);
        }
   }
}
```

Because this application uses the basicSO server-side application, you don't need to add a new folder to the server-side applications folder.

Figure 4-2 shows what you can expect to see when you test the application and start moving the shared object movie clip around the Stage. Initially, connection of the shared object fires a sync event, and the result is a message of "clear," as can be seen in the Output window. Then the movement of the movie clip changes the values of the shared object, and because you are looking at local movement, the sync code shows "success." If you moved the object on a remote client, you would see "change"; but because it is only running in the test mode without a remote client, you won't see any "change" code unless you publish the code and open a browser and run it as a separate client.

Figure 4-2. Movie clip shared object



To see how this all works, begin with the `moveMC` function and the lines:

```
pointer_so.setProperty ("xpos",island.x);
pointer_so.setProperty ("ypos",island.y);
```

Those lines set the `pointer_so` data value of the `xpos` and `ypos` properties based on the position of the shared movie clip. When the change occurs, it fires a `SYNC` event. The move action initiates a `SyncEvent.changeList` "change" code on the remote client's player. (On the local client-the one that drags the movie clip-the code is "success," not "change.") The script then iterates through the array of clients, and then the shared object's properties (named `xpos` and `ypos`.) The loop first finds each client and where change is indicated, and then updates the value of the movie clip's position.

Outlined, the process looks like this:

- A client moves the shared object (the island movie clip).
- The movement updates the value of the shared object on the client's application.
- The change in the shared object initiates a sync event with a `changeList` code "change" for all remote clients.

- The `doUpdate()` function assigns the value of the shared object properties to the local movie clip `x` and `y` properties, thereby moving the movie clip.

As noted in the introduction to this chapter, you can do a lot of practical things using shared objects. Using multiple modules, you can create one module (a control modules) to remotely control objects on the pages of other modules. However, users on the other modules cannot control any shared objects even though they can see the changes made by the control module. In this way, you can set up an application where you have a speaker point to different parts of a page while giving a presentation. Likewise, in gaming, you can do all kinds of interesting things with two or more players controlling movie clips that appear and move other remote clients.

4.4. Minimalist Project for Shared Text

Having seen a minimalist project for a movie clip, you'll now look at one for sharing text. The concepts are essentially the same, but instead of basing the changes in the shared object slots on the movement of the mouse position on the Stage, the changes are based on typed input by the user.

In this application, the user changes the value of the shared object property attribute (text content) by typing the value and pressing a button to fire a function. The function enters the value of the text input field and assigns that value to the shared object attribute. Here are the basic elements of the project:

Classes

NetConnection

SharedObject

NetStatusEvent

SyncEvent

MouseEvent

TextArea

InputText

Button

Objects

TextArea component

InputText component

Button component

Button component

Follow these steps to create the application:

1. Open a new Flash file (ActionScript 3.0) and save it as TextSO.fla.

2. Open the Library panel (press Ctrl+L (Windows) or Command+L (Mac OS)) and drag a TextArea, InputText, and Button component into the Library. Save the file again.

3. Open a new ActionScript file and save it as TextSO.as.

4. Enter the script in Example 4-3 and resave the file.

Example 4-3. TextSo.as

Code View:

```
package
{
  import fl.controls.TextArea;
  import fl.controls.Button;
  import fl.controls.TextInput;
  import flash.display.Sprite;
  import flash.events.SyncEvent;
  import flash.events.NetStatusEvent;
  import flash.events.MouseEvent;
  import flash.net.SharedObject;
  import flash.net.NetConnection;

  public class TextSO extends Sprite
  {
        private var button:Button;
        private var text_so:SharedObject;
        private var nc:NetConnection;
        private var textArea:TextArea;
        private var textInput:TextInput;
        private var rtmpGo:String;
        private var good:Boolean;

        public function TextSO ()
        {
             //Set up UIs
             textArea=new TextArea();
             textArea.setSize (200,300);
             textArea.move (20,20);
             addChild (textArea);

             textInput=new TextInput();
             textInput.move (20,330);
             addChild (textInput);

             button=new Button();
             button.width=50;
             button.label="Send";
             button.move (125,330);
             button.addEventListener (MouseEvent.CLICK,sendMsg);
             addChild (button);

             rtmpGo = "rtmp://192.168.0.11/basicSO";
             nc = new NetConnection( );
             nc.connect (rtmpGo);
             nc.addEventListener (NetStatusEvent.NET_STATUS,doSO);
        }

        private function doSO (e:NetStatusEvent):void
        {
             good=e.info.code == "NetConnection.Connect.Success";
```

```
            if (good)
            {
                  //Set up shared object
                  text_so=SharedObject.getRemote("test",nc.uri,false);
                  text_so.connect (nc);
                  text_so.addEventListener (SyncEvent.SYNC,checkSO);
            }
      }

      private function checkSO (e:SyncEvent):void
      {
            for (var chng:uint; chng<e.changeList.length; chng++)
            {
                  switch (e.changeList[chng].code)
                  {
                        case "clear" :
                              break;

                        case "success" :
                              trace (text_so.data.msg);
                              break;

                        case "change" :
                              textArea.appendText (text_so.data.msg + "\n");
                              break;
                  }
            }
      }

      private function sendMsg (e:MouseEvent):void
      {
            text_so.setProperty ("msg",textInput.text);
            textArea.appendText (textInput.text + "\n");
      }
  }
}
```

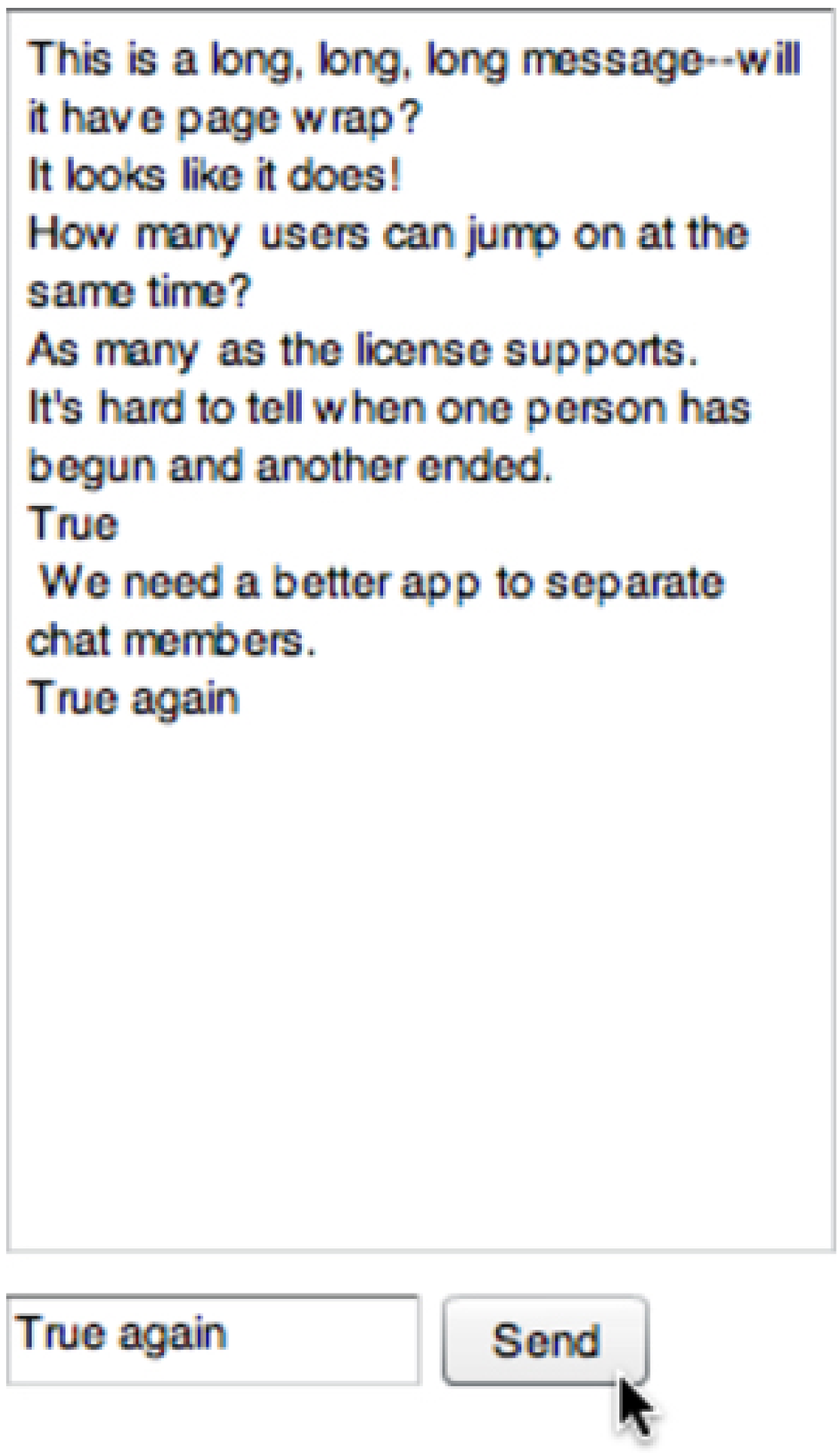
When you test this application, you will need to run two copies of it. Even though it's a minimum application, it's perfectly functional and you could chat with others all day long with this little app. Just remember this sequence when coding chat applications:

- Change value of shared object property attribute (for example, text content).
- Trigger `SyncEvent.SYNC` event due to change in the shared object `msg` data property.
- Change local property values by user input and assignment to the TextArea component.
- Change remote property values with shared object value.

When you look at client-side shared objects as a simple sequence, it's not so hard to see how they are set up in a script.

Figure 4-3 shows what you can expect to see in your browser.

Figure 4-3. Shared text



The script maintains a good deal of the structure from the shared object that moves the movie clip on remote clients. In the next application, you will see a very similar code structure as well.

4.5. Minimalist Project for Shared Function

This next little minimalist project is very cool. It allows you to open a Web page on someone else's browser. You can open a Web page from any domain, not just your local one. Here are the classes and objects you'll use:

Classes

- NetConnection
- SharedObject
- NetStatusEvent
- SyncEvent
- MouseEvent
- URLRequest
- InputText
- Button

Functions

- navigateToURL

Objects

- InputText component
- Button component

In setting up a link to another URL, ActionScript 3.0 employs the URLRequest class and `navigateToURL()` function, both of which are part of the `flash.net` package. Because the function's argument is the string with the URL, the argument will be the shared object. Figure 4-4 shows a chunk of code from the project that traces the changing of the shared object to the firing of the function. The five key elements are numbered from 1 to 5.

Figure 4-4. Path of shared object events

```
60          private function checkSO (e:SyncEvent):void
61          {
62              for (var chng:uint; chng<e.changeList.length; chng++)
63              {
64                  switch (e.changeList[chng].code)
65                  {
66                      case "clear" :
67                          break;
68
69                      case "success" :
70                          trace (web_so.data.web);
71                          break;
72
73                      case "change" :
74                          webRequest=new URLRequest(web_so.data.web);
75                          navigateToURL (webRequest,"_blank");
76                          break;
77                  }
78              }
79          }
80
81          private function sendWeb (e:MouseEvent):void
82          {
83
84              web_so.setProperty ("web", "http://"+ textInput.text);
85
86          }
```

- In step 1, a data property is both defined ("web") and assigned a value (`http://` plus the value of the string in the text input box).

- The change in a shared object causes a sync event and fires the `checkSO` function shown in step 2.

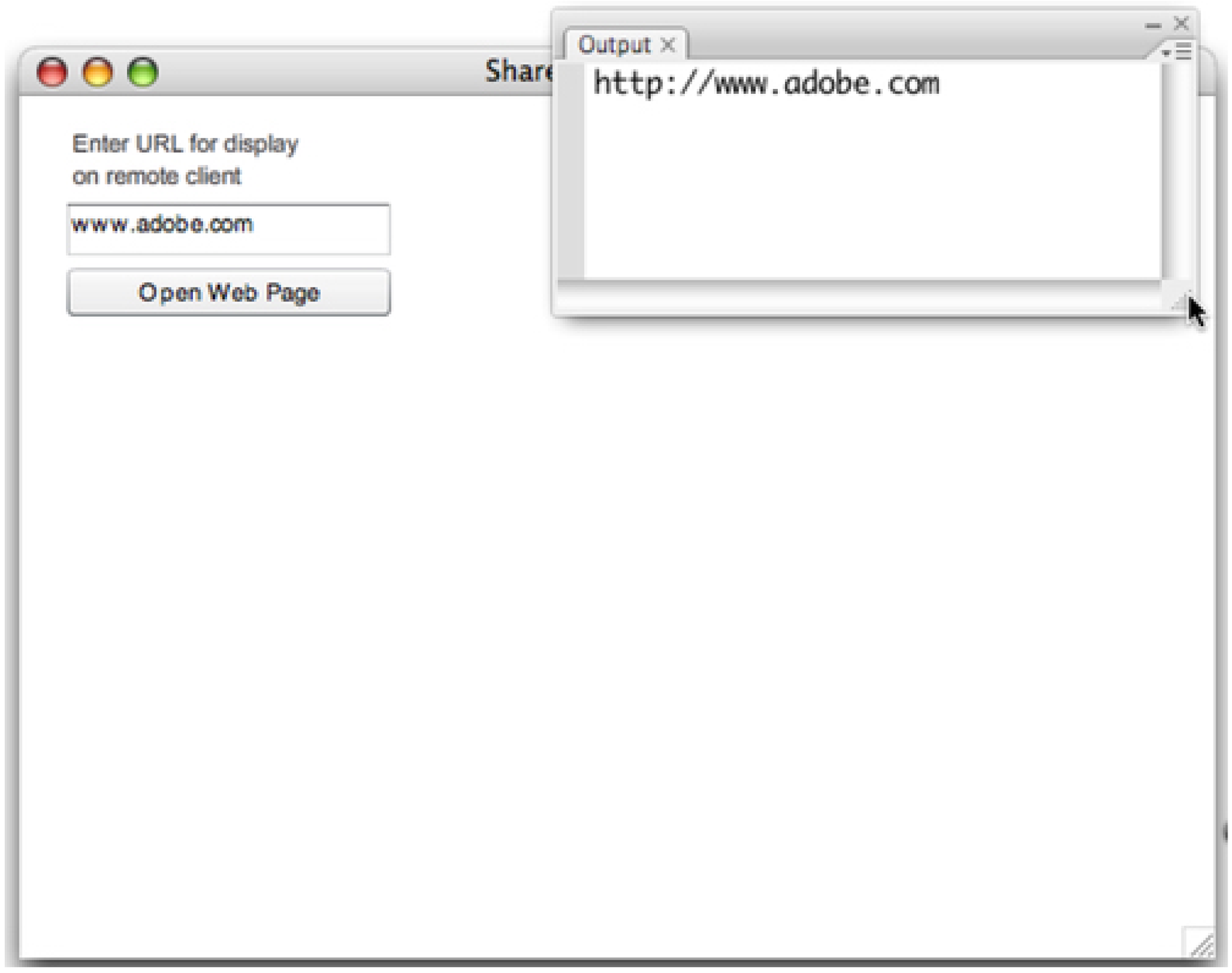
- In step 3, the script iterates through the `changeList` array looking for the code value.

- Step 4 shows the script finding the code value "change."

- Step 5 launches the Web site stored in the `web_so.data.web` slot where a "change" value has been identified.

To help you see what else is going on in this process, the code value "success" displays the value of the `web_so.data.web` in the Output window. If you wanted, you could use that event code to open the same Web site in the client's window. Figure 4-5 shows the application at work.

Figure 4-5. Opening Web page on remote client



Use Figure 4-5 as a guide and implement the following steps to create the full application:

1. Open a new Flash file and save it as SharedFunction.fla.

2. Using an 11-point Arial font, type in the Static text box: **`Enter URL for display on remote client`**. Position the text at x=20, y=14.

3. Open the Library panel (press Ctrl+L (Windows) or Command+L (Mac OS)) and drag an InputText and Button component into the Library. Save the file again.

4. Open a new ActionScript file and save it as SharedFunction.as.

5. Add the script shown in Example 4-4, and resave the file.

Example 4-4. SharedFunction.as

```
Code View:
package
{
  import fl.controls.Button;
```

```
import fl.controls.TextInput;
import flash.display.Sprite;
import flash.events.SyncEvent;
import flash.events.NetStatusEvent;
import flash.events.MouseEvent;
import flash.net.SharedObject;
import flash.net.NetConnection;
import flash.net.URLRequest;
import flash.net.navigateToURL;

public class SharedFunction extends Sprite
{
	private var button:Button;
	private var web_so:SharedObject;
	private var nc:NetConnection;
	private var textInput:TextInput;
	private var rtmpGo:String;
	private var webURL:String;
	private var good:Boolean;
	private var webRequest:URLRequest;

	public function SharedFunction ()
	{
		//Set up UIs

		textInput=new TextInput();
		textInput.setSize(150,24);
		textInput.move (20,50);
		addChild (textInput);

		button=new Button();
		button.width=150;
		button.label="Open Web Page";
		button.move (20,80);
		button.addEventListener (MouseEvent.CLICK,sendWeb);
		addChild (button);

		//NetConnection
		rtmpGo = "rtmp://192.168.0.11/basicSO";
		nc = new NetConnection( );
		nc.connect (rtmpGo);
		nc.addEventListener (NetStatusEvent.NET_STATUS,doSO);
	}

	private function doSO (e:NetStatusEvent):void
	{
		good=e.info.code == "NetConnection.Connect.Success";
		if (good)
		{
			//Set up shared object
			web_so=SharedObject.getRemote("web",nc.uri,false);
			web_so.connect (nc);
			web_so.addEventListener (SyncEvent.SYNC,checkSO);
		}
	}

	private function checkSO (e:SyncEvent):void
	{
		for (var chng:uint; chng<e.changeList.length; chng++)
```

```
            {
                  switch (e.changeList[chng].code)
                  {
                        case "clear" :
                              break;

                        case "success" :
                              trace (web_so.data.web);
                              break;

                        case "change" :
                              webRequest=new URLRequest(web_so.data.web);
                              navigateToURL (webRequest,"_blank");
                              break;
                  }
            }
      }

      private function sendWeb (e:MouseEvent):void
      {

            web_so.setProperty ("web", "http://"+ textInput.text);

      }
  }
}
```

6. Switch to the SharedFunction.fla file and choose File Publish to generate an HTML file.

7. If you are developing on a single computer that you're using as both a server and host, such as a typical Windows XP or Vista machine, open your browser in your local Web host. (Typically the reference will be something like http://localhost/myFMSapps/ or http://127.0.0.1/myFMSapps/.) Navigate to SharedFunction.html and open the application. Now, from your Flash CS3 application, choose Control Test Movie to try out the application. Enter a Web URL without the "http://" portion, and click the button.

When you do the test, you will see a Web page open only on the remote client. The full URL name will appear in the Output window to give you an idea of what the remote client will actually open. It's more dramatic, though, if you test the application on a remote client. Even with a LAN, if you have your browser open on a computer other than the one on which you're developing the application, you will see the Web page on the remote site (another computer on the LAN) open.

4.6. An Upgraded Text Chat

Now that you have an idea of how shared objects work, this next application shows how to make a pretty good text chat application. Up to this point, the treatment of shared objects has been kept simple so that you could see the basics. However, now you need to reconsider the user interface for the text chat application.

In any application, you cannot think simply of the code. This is especially true when using Flash Media Server 3. You have to think of the interaction and the flow of actions by two or more people. You want to make the experience of your application as easy as possible-so easy that no one using your application notices it. That is, of course, unless you want the application to bring something to the user's attention.

4.6.1. Chatter Considerations

When people engage in a text chat, their hands are on the keyboard. Once they finish a message, instead of having to remove their hands from the keyboard to click a button with the mouse, they should have the option of sending their message using the keyboard. So that's the first thing to add to the basic chat created in the TextSO example. When a writer has finished with the current message, the writer can either click a Send button or just press the Enter or Return key.

In addition, different people in the chat room should be identified when they send a message. Adding a name to a message has two important consequences. First, it separates each message from others, which helps in clear communication. Second, the sender is identified so that everyone knows who said what.

As a last step, to make life easier for the chatter, as soon as the chatter enters the message in the input window, the message should disappear. That clears the message window for the next message to be typed in. It's a little thing, but it makes the chat flow much smoother.

To recap, here are the items you need to create to enhance the text chat application:

1. Allow the user to press the Enter or Return key and submit a message without having to click the Submit button.

2. Separate messages in the chat window according to author.

3. After the user submits a message, the input message field is cleared.

4.6.2. A Better Chat Application

This upgraded real-time chat application needs to have more events added and a second `TextInput` instance where participants can enter their names. To make it easy, the application will also include a focus event (`FocusEvent`) so that when a person enters his or her name, the prompt message in the box disappears. Further, the application should verify that the user entered a name and did not leave the prompt message or a blank in the name window.

To give participants the opportunity to use keyboard message entry, the script will include the ComponentEvent class. This class can listen for different events stored in constants, including `ENTER` to trap pressing the Enter or Return key. Because the application should listen only for Enter/Return, the script does not use the KeyboardEvent class. Doing so would have required the application to listen to every key press, and in a chat application, this would have been a lot of work for the processor.

Follow these steps to create this application:

1. Open the TextSO.fla application, and change the Document Class value in the Property inspector to TextChat. Choose File → Save As and save the file as TextChat.fla. You can open a brand new Flash file, but this saves some time because you're going to need to use the Button, TextInput and TextArea components already in the Library.

2. Using an 18-point display font, enter the Static text label **`Text Chat Center`** positioned at x=16, y=17 on the Stage. Figure 4-6 shows how the label will look when the application is running.

3. Open a new ActionScript file and save it as TextChat.as. Enter the code in Example 4-5 and resave the file.

Example 4-5. TextChat.as

Code View:

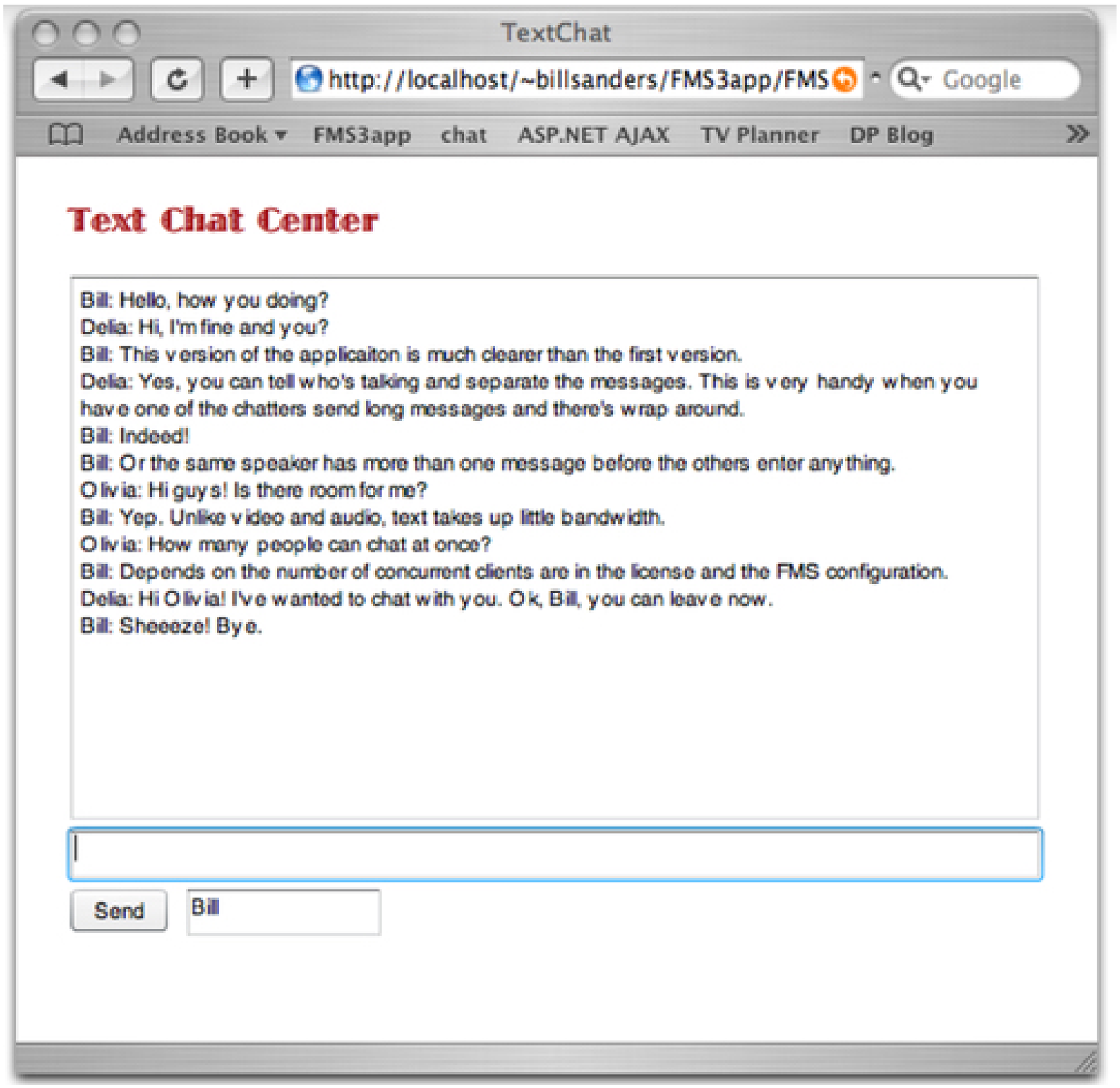
```
package
{
  import fl.controls.TextArea;
  import fl.controls.Button;
  import fl.controls.TextInput;
  import flash.display.Sprite;
  import flash.events.SyncEvent;
  import flash.events.NetStatusEvent;
  import flash.events.MouseEvent;
  import flash.events.FocusEvent;
  import flash.net.SharedObject;
  import flash.net.NetConnection;
  import fl.events.ComponentEvent;

  public class TextChat extends Sprite
  {
        private var button:Button;
        private var text_so:SharedObject;
        private var nc:NetConnection;
        private var textArea:TextArea;
        private var textInput:TextInput;
        private var chatName:TextInput;
        private var rtmpGo:String;
        private var good:Boolean;
        private var catchKey:Boolean;
        private var noName:Boolean;

        public function TextChat ()
        {
             //Set up UIs
             textArea=new TextArea();
             textArea.setSize (500,280);
             textArea.move (20,54);
             addChild (textArea);

             textInput=new TextInput();
             textInput.setSize (500,24);
```

```
            textInput.move (20,340);
            textInput.addEventListener (ComponentEvent.ENTER,checkKey);
            addChild (textInput);

            button=new Button();
            button.width=50;
            button.label="Send";
            button.move (20,370);
            button.addEventListener (MouseEvent.CLICK,sendMsg);
            addChild (button);

            chatName=new TextInput;
            chatName.setSize (100,24);
            chatName.move (80, 370);
            chatName.text="<Enter Name>";
            chatName.addEventListener (FocusEvent.FOCUS_IN,cleanName);
            addChild (chatName);

            rtmpGo = "rtmp://192.168.0.11/basicSO";
            nc = new NetConnection( );
            nc.connect (rtmpGo);
            nc.addEventListener (NetStatusEvent.NET_STATUS,doSO);
      }

      private function doSO (e:NetStatusEvent):void
      {
            good=e.info.code == "NetConnection.Connect.Success";
            if (good)
            {
                  //Set up shared object
                  text_so=SharedObject.getRemote("test",nc.uri,false);
                  text_so.connect (nc);
                  text_so.addEventListener (SyncEvent.SYNC,checkSO);
            }
      }

      private function checkSO (e:SyncEvent):void
      {
            for (var chng:uint; chng<e.changeList.length; chng++)
            {
                  switch (e.changeList[chng].code)
                  {
                        case "clear" :
                              break;

                        case "success" :
                              break;

                        case "change" :
                              textArea.appendText (text_so.data.msg + "\n");
                              break;
                  }
            }
      }

      private function cleanName (e:FocusEvent):void
      {
            chatName.text="";
      }
```

```
        private function sendMsg (e:MouseEvent):void
        {
             noName=(chatName.text=="<Enter Name>" || chatName.text=="");
             if (noName)
             {
                  textArea.appendText ("You must enter your name \n");
             }
             else
             {
                  text_so.setProperty ("msg",chatName.text +": "+ textInput.text);
                  textArea.appendText (chatName.text +": "+textInput.text + "\n");
                  textInput.text="";
             }
        }

        private function checkKey (e:ComponentEvent):void
        {
             noName=(chatName.text=="<Enter Name>" || chatName.text=="");
             if (noName)
             {
                  textArea.appendText ("You must enter your name \n");
             }
             else
             {
                  text_so.setProperty ("msg",chatName.text +": "+ textInput.text);
                  textArea.appendText (chatName.text +": "+textInput.text + "\n");
                  textInput.text="";
             }
        }
    }
}
```

Figure 4-6 shows how the application should appear in a browser when you test it.

Figure 4-6. Text chat with name-delineated messages



Once you have all of the code in place, test it with another user (or open two browsers on your computer). This kind of application is useful for both a standalone real-time text chat or as part of another application as a chat module. If you put it up on a server, you can chat with people all over the world and with as many as the license will support. You might even consider reconfiguring your Flash Media Server to accept more concurrent users but use less bandwidth, because text transfer takes up so little bandwidth compared to audio and video.

Chapter 5. Two-Way Audio-Video Communications

5.1. Face-to-Face Communication

The dream of face-to-face communication via something like a combined television/ telephone has been around for at least the last 50 years. However, the promise of such communication between individuals in remote locations has only been realized and widespread lately. With open socket technology and high-speed Internet, such face-to-face audio/visual (A/V) communication over the Web is simple and becoming more common.

This chapter looks at the basic ingredients for creating a two-way A/V application using FMS3-either FMIS3 or the Development FMS3. FMSS cannot be used for interactive applications. The applications in this chapter all use classes and objects briefly discussed in previous chapters, especially in Chapter 3, "Setting Your Camera and Microphone," which examined the Camera, Microphone, and Video object,s and the NetStream class. You might want to review Chapter 3 before getting started so that you're less likely to get lost in the discussion.

Chapter 5. Two-Way Audio-Video Communications

5.1. Face-to-Face Communication

The dream of face-to-face communication via something like a combined television/ telephone has been around for at least the last 50 years. However, the promise of such communication between individuals in remote locations has only been realized and widespread lately. With open socket technology and high-speed Internet, such face-to-face audio/visual (A/V) communication over the Web is simple and becoming more common.

This chapter looks at the basic ingredients for creating a two-way A/V application using FMS3-either FMIS3 or the Development FMS3. FMSS cannot be used for interactive applications. The applications in this chapter all use classes and objects briefly discussed in previous chapters, especially in Chapter 3, "Setting Your Camera and Microphone," which examined the Camera, Microphone, and Video object,s and the NetStream class. You might want to review Chapter 3 before getting started so that you're less likely to get lost in the discussion.

5.2. The NetStream Bundle

It sometimes helps to think of the NetStream class as a pack mule. You load it up with different things and send it on its way. Once the stream arrives at its destination, it unloads its bundle to be received as audio and video. Like pack mules, every destination requires a different stream. So if you are sending a bundle to Kowloon, Sydney, Yokohama, Mumbai, and Lima, you'll need five mules, one for each destination. As far as you the developer is concerned, you will need to create only a single `NetStream` instance to send the same video and audio to all of those places. Even though you're creating a single instance, FMS3 generates a separate stream for each one. That's important to keep in mind because you've got to pay the bandwidth price for each place that receives the stream. (Think of buying each of your pack mules an airline ticket paid for in bandwidth bucks.)

In addition to sending out bundles, you want to receive bundles. You send out audio and video and you want to receive audio and video. So, for each object you want to receive, you will need one `NetStream` instance. Bandwidth-wise, this can get expensive quickly. Figure 5-1 shows the difference between two connections and four.

Figure 5-1. Connections and streams

Two connections require two streams; but four connections require 12 streams. You can calculate the number of streams generated with this formula:

$$S = c^2 - c$$

S represents the number of streams generated and *c* represents the number of connections.

Figure 5-1 illustrates that two connections generate only two streams (2 x 2 = 4; 4 – 2 = 2) while four connections generate twelve (4 x 4 = 16; 16 - 4 = 12).

As you can see, as the number of connections increases numerically, the number of streams increases exponentially. For instance, if you double the number of connections from 4 to 8, the number of streams jumps from 12 to 56-more than quadrupling the number of streams and the effect on bandwidth.

5.3. The NetStream Class and Live Streams

Chapter 2 and Chapter 3 introduced the NetStream class, but they didn't cover it in any detail. This chapter begins by examining the NetStream class as it is used for live streaming, and then goes into the minimalist example. Using the NetStream class, the goal is to capture audio and video input from the camera and microphone and send them to another person to be seen and heard. This process involves the following `NetStream` methods:

- `NetStream.connect(myNetConnection)`
- `NetStream.attachAudio(microphone)`
- `NetStream.attachVideo(camera)`
- `NetStream.publish("streamName")`
- `NetStream.play("streamName")`

In looking at the sequence required to generate a two-way A/V connection, the stream is connected to the server via `NetConnection`, to establish a link between the user and the application on the server. The publishing stream needs to be attached to both the publishing camera and microphone. That stream is sent to the FMS3 server, and the server sends it to the connected clients. The following script segment shows this sequence:

```
//Stream Out
nsOut=new NetStream(nc);
nsOut.attachAudio (mic);
nsOut.attachCamera (cam);
nsOut.publish("left","live");
```

Here's what the script does:

- Creates a stream associated with a net connection to FMS3.
- Attaches the microphone to the stream.
- Attaches the camera to the stream.
- Publishes the stream using a unique stream name.

Now that you have a publishing module, set up a playing module that captures and plays the stream that was sent. You don't want to capture the same stream that you just published, so you will need a stream name other

than "left." The play stream will be called right, keeping in mind that it represents the current recipient. (The other recipient plays "right" and receives "left.") To capture the incoming stream, you need to attach the stream to the video object on the Stage. The stream containing both audio and video is treated as a single unit-a video. You don't need to attach the sound to an object to have it play back, as you did when sending video. So, you attach the stream to the video object using the `attachVideo( )` method. Once the stream is being sent to the right place (the embedded video object), all you need to do is to play the stream, as the following code segment shows:

```
//Stream In
nsIn=new NetStream(nc);
nsIn.play("right");
vidStream.attachNetStream(nsIn);
```

The sequence for reading and displaying the incoming stream, then, would be:

- Create a stream associated with a net connection to FMS3.
- Attach the stream to the video object.
- Play the stream.

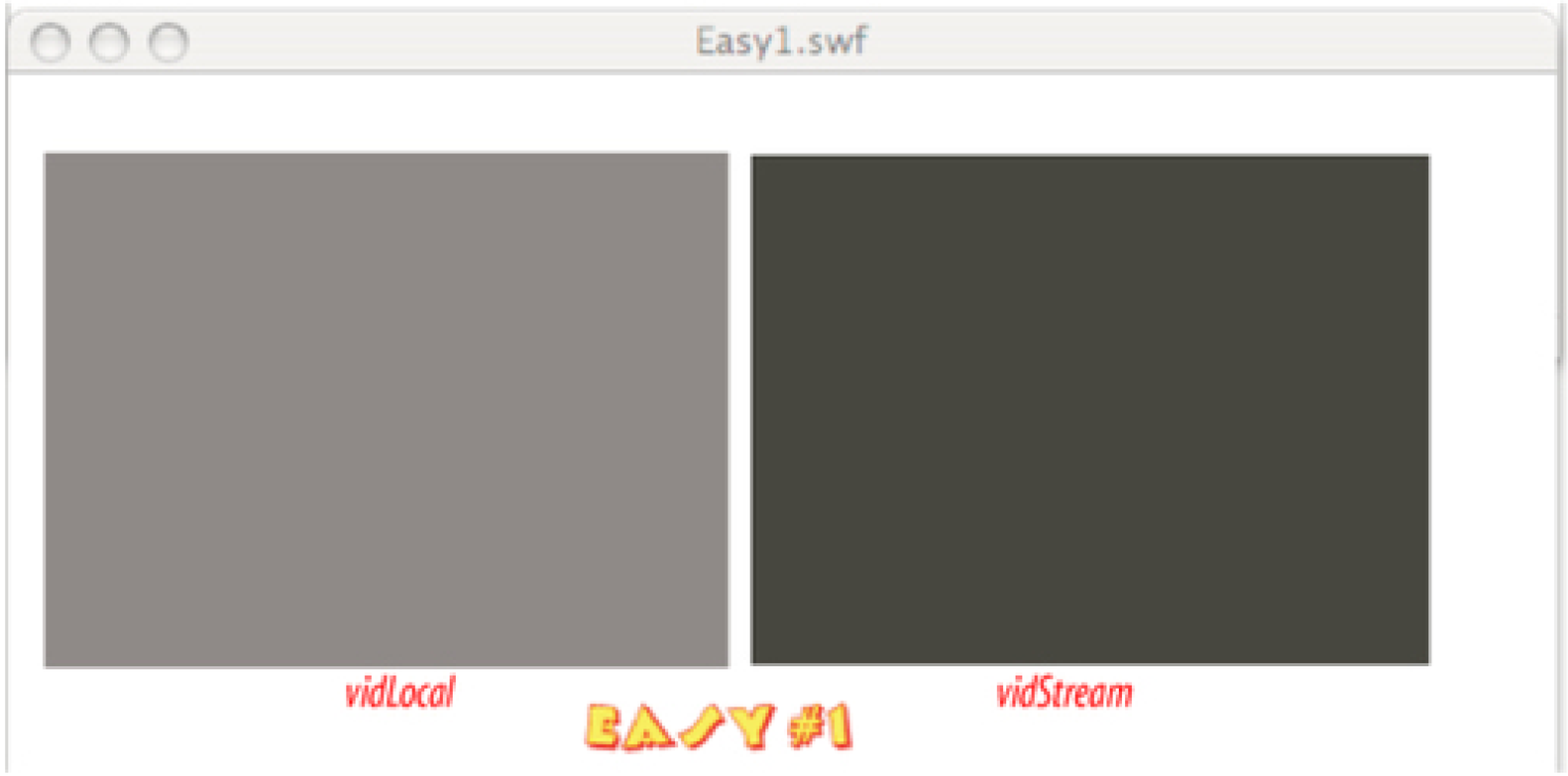
There's more to creating a two-way A/V application, but the core of doing so is working with the NetStream class' methods. Once you get the order straight in your mind, the rest is pretty easy.

5.4. The World's Easiest Two-Way A/V Chat Application

To make a minimalist two-way A/V chat application requires two modules. Keeping in mind that the FMS3 application is simply the reference name you use in the RTMP URL, you can have several modules with a single application. That is, the application name is the name of the folder on the server-side.

That's what you're going to do to make this application. The application name will be easy. You will have two modules, Easy1 and Easy2. Each module will connect to FMS3 through the easy application stored on the server-side. For this application, each module will be written to receive the stream sent by the other. As you will see, a simple code swap does all the real work. So the task really involves writing one module, duplicating it, and then making a few edits. Figure 5-2 shows the areas where the two videos will appear, and their instance names:

Figure 5-2. Video instances



To get started, review the following classes and objects you will need:

Classes

NetConnection

NetStream

NetStatusEvent

Camera

Microphone

Video

Follow these steps to make this chat application:

1. Open a new Flash file and save it as Easy1.fla.

2. In the Tools panel, select the Text tool and add a 24-point Static text label, **Easy #1**. Position the label at x=205, y=223.

3. Open the Property inspector, and in the Document Class text box, type **Easy1**. Resave the file.

4. Open a new ActionScript 3.0 Flash file and save it as Easy1.as in the same folder as the Easy1.fla file.

5. In the Easy1.as file, add the script shown In Example 5-1 and save the file again.

Example 5-1. Easy1.as

Code View:

```
package
{
  import flash.display.Sprite;
  import flash.events.NetStatusEvent;
  import flash.net.NetConnection;
  import flash.net.NetStream;
  import flash.media.Camera;
  import flash.media.Microphone;
  import flash.media.Video;

  public class Easy1 extends Sprite
  {
        private var nc:NetConnection;
        private var good:Boolean;
        private var rtmpNow:String;
        private var nsIn:NetStream;
        private var nsOut:NetStream;
        private var cam:Camera;
        private var mic:Microphone;
        private var vidLocal:Video;
        private var vidStream:Video;


        public function Easy1 ()
        {
             rtmpNow="rtmp://your.web.com/easy";
             nc=new NetConnection();
             nc.connect (rtmpNow);
             nc.addEventListener (NetStatusEvent.NET_STATUS,checkCon);
             setCam ();
             setMic ();
             setVideo ();
        }
```

```
        private function checkCon (e:NetStatusEvent):void
        {
             good=e.info.code == "NetConnection.Connect.Success";
             if (good)
             {
                  nsOut=new NetStream(nc);
                  nsOut.attachAudio (mic);
                  nsOut.attachCamera (cam);
                  nsOut.publish("left","live");

                  nsIn=new NetStream(nc);
                  nsIn.play("right");
                  vidStream.attachNetStream(nsIn);
             }
        }

        private function setCam()
        {
             cam=Camera.getCamera();
             cam.setKeyFrameInterval (9);
             cam.setMode (240,180,15);
             cam.setQuality (0,80);
        }

        private function setMic()
        {
             mic=Microphone.getMicrophone();
             mic.gain=85;
             mic.rate=11;
             mic.setSilenceLevel(15,2000);
        }

        private function setVideo()
        {
             vidLocal=new Video(cam.width,cam.height);
             addChild(vidLocal);
             vidLocal.x=15; vidLocal.y=30;
             vidLocal.attachCamera(cam);

             vidStream=new Video(cam.width,cam.height);
             addChild(vidStream);
             vidStream.x=(vidLocal.x+ cam.width +10); vidStream.y=vidLocal.y;
        }
   }
}
```

6. Make sure that the line `rtmpNow="rtmp://your.web.com/easy"` reflects the FMS URL to your own URL, whether it's the localhost on your own computer, a LAN IP address, or the URL of a remote server.

7. When you have entered the code, resave the file. Then open the Easy1.fla file. Choose File Publish to publish your HTML, SWF, and JavaScript files (AC_RunActiveContent.js).

   This completes the first module. Test it; you should see yourself in the left window. The next steps simply

require a few changes to the first module.

8. Open the Easy1.fla file, if it's not already open. Choose File Save As and rename the file Easy2.fla.

9. Open the Property inspector, and in the Document Class text box, type **Easy2**. Change the Static text on the Stage from *Easy #1* to *Easy #2*. Resave the FLA file.

10. Open the Easy1.as file and choose File Save As to rename and save the file as Easy2.as.

11. Change the class name and the constructor function from `Easy1` to `Easy2` so that they read:

```
public class Easy2 extends Sprite
....
public function Easy2()
```

12. In the `checkCon` function, change the following two lines:

```
nsOut.publish("left","live");     nsOut.publish("right","live");
....
nsIn.play("right");     nsIn.play("left");
```

13. Save the Easy2.as file. Your two-module application is complete.

Step 12 shows the key lines in the application. Easy1 publishes a stream named left, and plays a stream named right. Easy2 does the opposite so that each module plays that the other's stream.

To test this application, you will need two cameras and ideally two computers. One user runs Easy1.html and the other Easy2.html. Both modules must be able to access the FMS3 server whether on a LAN or accessing a remote server over the Internet. Figure 5-3 shows what you should see when you run both modules.

Figure 5-3. Two-way Audio-Video chat

If your IP address can be accessed remotely, you can chat with anyone in the world. Just use your Web server as the root for your materials, as described in "Section 1.5" in Chapter 1.

5.5. A Better Two-Way Chat Application

While a two-module chat application is easy enough to create and works perfectly well, its sole purpose is to show the minimal code required to create a two-way chat. A better option is a two-way chat that has a single module and that can tell which way to set the streams. The optimal approach is to write a server-side script. The concept of a FMS3 application is best appreciated when you look at a script in which all users who use the same application get different information. You're familiar with variables in a non-FMS application that change but use no server-side data such as a database. However, if someone else uses the same application, what you do has no effect on the other's use of the same application. Take for example an online game that has no server-side component. Your score won't affect anyone else's score who's playing the same game. One way to think about it is that the application is not a shared one.

In Chapter 3, "Nonpersistent Client-Side Remote Shared Objects," you learned about shared objects and saw how several users connected through a common application could affect others. No server-side scripts were employed in those applications because the client-side script communicated through FMS without requiring a script on the server. This chapter introduces you to a server-side script that tracks whether one or two users are using the same application at the same time. The script has much in common with the two-module application discussed in the previous section, but instead of two modules, this next application uses only one. Because the work done by two modules is handled by the server-side script, you need only a single module.

5.5.1. Keeping Track of Users

Before getting started on the server-side script, you have to shift your thinking a bit: Instead of using ActionScript 3.0, essentially you'll be using ActionScript 1.0 with a few classes added for Flash Media Server 3.0. (If you're not familiar with ActionScript 1.0, think of the script as a slightly modified JavaScript.) That means that the data is untyped (no data type is assigned) and the only classes you can create are done using prototype.

However, this first script is only going to do one thing. Using an array with two elements, it will use the `Array.pop()` method to provide one of two strings, left and right. When a client leaves, the element is placed back on the array using `Array.push()`. To understand how all of this is going to work, you first need to understand something about two major server-side classes, Application and Client.

Server-side ActionScript (SSAS) has very few classes-15 in total. Three of those classes deal with XML and two with SOAP. In reality, you'll spend most of your time with only four classes: Application, Client, SharedObject and Stream. For this application, you need only the first two.

5.5.1.1. The Application Class

As its name implies, the Application class deals with the entire application. Each application, no matter how many clients are connected, has a single application object. What makes this class unusual is that it is automatically instantiated as `application`. So when you write SSAS code, you don't have to create an object-it's been done automatically for you. Thus, to use this class, you simply write:

```
application.doSomething...
```

That can be confusing; think of it as one less step to instantiate an instance of the class.

The remaining chapters that explore server-side code will go into more server-side classes, as well as cover more properties, methods and events used with the Application class. For now, you'll use three events and two methods:

Application Events

- onAppStart
- onConnect
- onDisconect

Application Methods

- rejectConnection
- acceptConnection

The client can trigger application events simply by connecting to the application. The first client to open the application triggers the onAppStart event. Subsequent users (clients) won't affect that event, and as long as the application is running-which can be up to 20 minutes after the last client has left the application-the event does not launch.

Each new connection to the application triggers the onConnect event. Every new client who connects to the application through the RTMP process makes a connection, and that connection can be accepted or rejected by the Application methods, acceptConnection or rejectConnection.

Whenever a client disconnects from the application, the client is removed from the client array. You can also use the eventof a disconnection to employ the Application.onDisconnect event to create an unnamed function to be used for finding which clients have disconnected. This appears in the server-side script.

5.5.1.2. The Client Class

Like the Application class, the Client class automatically creates an instance of itself. Each client in an application becomes part of a client array for the application; each client is an array element. For example, supposed Nancy, Pete and Juan are all connected to the same application, with Nancy being first and Juan being last to connect. You could reference them as follows:

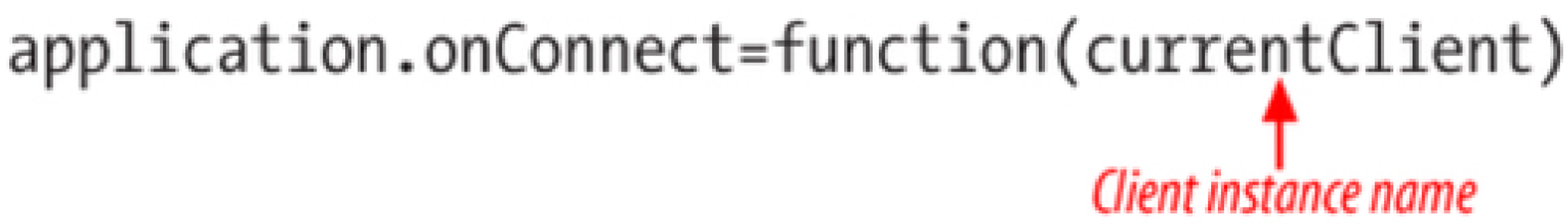
```
Nancy = application.clients[0];

Pete = application.clients[1];

Juan= application.clients[2];
```

Notice that the Application property name is clients with an *s*. (You can save a lot of debugging time by remembering that.) As the server-side script shows, the Client instance name is currentClient. It's sort of like stating,

```
currentClient = new Client();
```

However, that's not how the Client instance name comes into being. Rather, the client gets its reference name from the event function used to connect it to the server. Figure 5-4 shows the correct naming procedure.

Figure 5-4. Naming Client instance



The instance name `currentClient` is used for all of the clients, but keep in mind that each user connected is part of the `Application.clients` array. Like all classes, the instances can use the built-in properties, methods and events. However, you can also create your own properties and methods, just as with any other class. In this chapter's application, the properties and methods are created for the application (user property and method). The script contains one user method and one user property.

The user method is written as:

```
currentClient.streamSelect = function();
```

What's interesting about the method is that it is invoked from the client-side. Look at the client-side script to see how that is called.

The property added to the `currentClient` instance is called `cliNow` and is assigned the value from the `vidStreams` array created at the top of the script.

```
currentClient.cliNow=vidStreams.pop();
```

In the array at the top of the script, only two elements appear, the literals right and left. The value in the `currentClient.clinow` property is returned to the client-side in the `streamSelect` method.

5.5.2. The Server-Side Script

To create a server-side script, that generates a client name for only two participants and rejects more than two clients, follow these steps:

1. Open a new ActionScript Communication file and save the file as deux.asc

2. Enter the code in Example 5-2 and save the file again.

   Example 5-2. deux.asc

Code View:

```
//Two-element array
vidStreams=["right","left"];

//Application first starts
application.onAppStart = function()
{
  trace("The deux is out of the deck");
};

//A currentClient (user) attempts to connect
application.onConnect = function(currentClient)
{
 //Reject connection if array is empty
 //(Two users are currently using application)
  if (vidStreams.length <= 0)
 {
  application.rejectConnection(currentClient);
 }

 //Store array element in currentClient property
  currentClient.cliNow=vidStreams.pop();

  application.acceptConnection(currentClient);
  currentClient.streamSelect = function()
  {
        trace("Stream "+currentClient.cliNow+" used");
  //Sent the property to Client object
        return currentClient.cliNow;
  };
};

application.onDisconnect = function(currentClient)
{
 //When currentClient leaves put the element back in array
 vidStreams.push(currentClient.cliNow);
}
```

3. In the applications folder in your Flash Media Server 3 folder, create a new folder named deux.

4. Save the file in the deux folder, naming it deux.asc. Alternatively, name the file main.asc, depending on your preferences. (I prefer using the application name because if I name them all main.asc and then one gets inadvertently replaced or moved, the file is very difficult to find.)

Before going on to the client-side script, you need to understand essentially what this server-side script does. Whenever a client connects to the application, it pops an element off the array and returns it to the client through the client-side script. The script uses the string that it's sent to determine what name to use for the outgoing stream or incoming stream. Once the two-element array is empty, it won't allow any other clients to connect. When a client leaves, it replaces the element in the array, and now different client can join the chat. If both clients leave, two more clients can use it. Among other things, this script guarantees a private online audio

video chat. (Unless, of course, you talk to yourself.)

5.5.3. The Client-Side Strategy

Once you have finished the server-side script, you're all set for the client-side script. You'll simply make a change to the two-module application from earlier in the chapter, and convert it into a one-module chat application.

To make sure that the streaming audio and video goes to the right output, you simply need to get a unique string from the server-side script. The server-side script returns one of two strings to name the streams-left and right. The AS 3.0 Responder class is used to capture server-side information that you can use with the client script.

5.5.3.1. The Responder Class

The Responder class has parameters for working with returned data as well as error handling. This application focuses only on the first parameter, and so the example does not use the error handling. (Later examples will use the error-handling feature.) Figure 5-5 shows the call to the server and the responder's role in capturing the returned data:

Figure 5-5. Responder instance and call to server

In Figure 5-5, consider the `Responder` instance as specifying a callback function to catch whatever is returned by the call to the server. The `NetConnection.call()` [`nc.call("streamSelect",responder);`] method includes the name of the server-side function. The previous call points to the line,

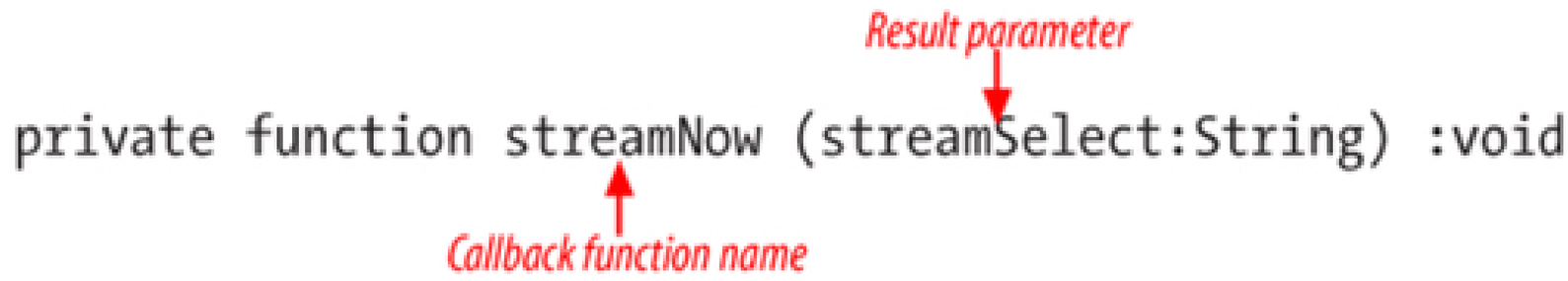
```
currentClient.streamSelect = function();
```

in the server-side script. (See "deux.asc. ActionScript Communication File" for the complete server-side script.)

5.5.3.2. Responder Callback Function

Once the call is made and the responder callback has been specified, the callback function requires a parameter to capture the data returned from the server. Figure 5-6 shows how the callback function is set up and what each part does.

Figure 5-6. Responder instance and call to server



In Figure 5-6, the callback parameter is the key to capturing what is sent from the server because it contains the value of the returned string. Now, the script uses that value as a unique stream name. The callback function contains the following switch statement:

```
switch (streamSelect)
{
  case "left" :
        outStream="left";
        inStream="right";
        break;
  case "right" :
        outStream="right";
        inStream="left";
        break;
}
```

The switch statement uses the returned value in `streamSelect` (the parameter in the callback function) to decide how to name the streams. If the returned value is "left," the outgoing stream is named left and the incoming stream is named right. Conversely, if the returned value is "right," the outgoing and incoming streams have the opposite names.

This procedure may seem fairly complex just to name a couple of streams. But keep in mind that using this script, two users across the globe can successfully coordinate communication. Also, this coordination can be done securely using only a single module.

5.5.4. The Client-Side Applications

The client-side files include a logo as well as the application name. The logo happens to be a MovieClip placed in the Library panel, and so the MovieClip has to be imported. (If your logo is not a MovieClip, you can omit the line that imports the MovieClip class.) Follow these steps to create the FLA file and then the ActionScript (.as) file:

1. Open a new Flash file (ActionScript 3.0) and save it as Deux.fla.

2. On the Stage, place your logo in the upper left corner. (Optional.)

3. In the middle of the Stage, use the Text tool to create the Static Text label **Deux** in 32-point text. Position the label at x=250, y=30. (The example uses Bauhaus 93 font.)

4. In the Document Class text box in the Property inspector, type **Deux**. Save the file.

5. Open a new ActionScript file, and save it as Deux.as in the same folder where you saved the Deux.fla file.

6. In the Deux.as file, enter the script in the Example 5-3, and resave the file.

Example 5-3. Deux.as

Code View:

```
package
{
  import flash.display.Sprite;
  import flash.display.MovieClip;
  import flash.events.NetStatusEvent;
  import flash.net.NetConnection;
  import flash.net.NetStream;
  import flash.media.Camera;
  import flash.media.Microphone;
  import flash.media.Video;
  import flash.net.Responder;

  public class Deux extends Sprite
  {
        private var nc:NetConnection;
        private var good:Boolean;
        private var netOut:NetStream;
        private var netIn:NetStream;
        private var cam:Camera;
        private var mic:Microphone;
        private var responder:Responder;
        private var vidOut:Video;
        private var vidIn:Video;
        private var outStream:String;
        private var inStream:String;

        public function Deux ()
        {
             var rtmpNow:String="rtmp://192.168.0.11/deux";
             nc=new NetConnection;
             nc.connect (rtmpNow);
             nc.addEventListener (NetStatusEvent.NET_STATUS,getStream);
        }

        private function getStream (e:NetStatusEvent):void
        {
             good=e.info.code == "NetConnection.Connect.Success";
             if (good)
             {
                  responder=new Responder(streamNow);
                  nc.call ("streamSelect",responder);
             }
        }

        private function streamNow (streamSelect:String):void
        {
             setCam ();
             setMic ();
             setVid ();
```

```
switch (streamSelect)
{
      case "left" :
           outStream="left";
           inStream="right";
           break;
      case "right" :
           outStream="right";
           inStream="left";
           break;
}

//Publish local video
netOut=new NetStream(nc);
netOut.attachAudio (mic);
netOut.attachCamera (cam);
vidOut.attachCamera (cam);
netOut.publish (outStream, "live");

     //Play streamed video
     netIn=new NetStream(nc);
     vidIn.attachNetStream (netIn);
     netIn.play (inStream);
}

private function setCam ():void
{
     cam=Camera.getCamera();
     cam.setMode (240,180,15);
     cam.setQuality (0,85);
}

private function setMic ():void
{
     mic=Microphone.getMicrophone();
     mic.rate=11;
     mic.setSilenceLevel (12,2000);
}

private function setVid ():void
{
     vidOut=new Video(240,180);
     addChild (vidOut);
     vidOut.x=25;
     vidOut.y=110;

     vidIn=new Video(240,180);
     addChild (vidIn);
     vidIn.x=vidOut.x+260;
     vidIn.y=110;
        }
  }
}
```

When you run the program, you should see pretty much what you saw using the Easy application. However, both current users will see the same thing. Figure 5-7 shows what you can expect to see when the application runs correctly.

Figure 5-7. Two-way chat with one module

By using a single module, you can use a single file name. This application was tested worldwide. From Connecticut, USA, chats were conducted with users in Bangkok, Singapore, Portugal, England, South Africa, Italy and other locations. Pictured is a developer from Digital Samba in Barcelona, Spain. So even though it's a simple application, it can be used to communicate worldwide.

5.6. Four-Way Conference Application

Going from a two-way chat application to multiple participant conference application is a matter of scale and bandwidth adjustment. On the server-side, the only requirement is to add two more elements to the array and change the name. However, before starting to work on the actual application, you'll look at the bandwidth requirements.

5.6.1. Two Equals Two: Four Equals Twelve

At the beginning of the chapter, Figure 5-1 shows how the number of streams grows from two in a two-way A/V chat to 12 in a four-way A/V conference. With that growth in the number of streams, cutting down on bandwidth is going to essential. To begin, look at the size of the videos sent across the Internet. In the Deux application, the camera is set to 240 x 180, generating roughly 43.2 kilobits (Kbits) pixels. Multiply that by the number of frames per second (fps = 15) and number of streams (2) and the application generates roughly 1.3 megabits (Mbits) per second. That does not include the audio, but for now, 1.3 Mbits can be used as a target bandwidth for the four-way conference. (The nice thing about audio is that in most conferences, only one person talks at a time, and so its size may not differ that much in a two- or four-person A/V application.)

Using 1.3 Mbits as the target bandwidth, the first consideration is the size of the video transmitted. In the Deux application, both participants have a lot of blank space on either side. Cutting out the extra horizontal space makes the speaker clearly visible, and you've saved a lot of unused bits to lug across the Internet. The four-way conference application, named Quad, cuts the camera width to 80 from 240. To better center the speaker, the height is cut to 100 from 180. Now, instead of a 240 x 180 matrix of bits, the matrix is 80 x 100. Next, cut down the fps to 12 from 15. Do the math, and the number of bits should be 1.15 Mbits-actually less than the two-person chat. Check the following steps:

a. 80 x 100 = 8,000.

b. 8,000 x 12 = 96,000 (where 12 is the number of streams).

c. 96,000 x 12 =1,152,000 (where 12 is also the frames per second).

When you've completed those calculations, you are in a position to create a site that is less likely to freeze up and give the clients a bad experience.

5.6.2. Positioning the Clients and their Streams

With four people in a conference, giving them a frame of reference helps them have a sense of the overall conference. To do this, the application has backdrops that frame where the videos appear. So, for example, if two users are at the site, they can see where the next two users appear. The logo is placed in the middle of the application to suggest a round table where everyone is on equal standing. However, since the view is on a vertical plane (a computer monitor), the person in the top position may appear more powerful than the others, and the person at the bottom in the lowest status.

Fortunately, because this is a virtual conference, the placement can put everyone in the top position from the user's own vantage point. Simply by sending all local video to the top position, each user sees him- or herself at the top, surrounded by the rest.

To get this effect and to distribute all of the streams correctly, the switch statement again comes in handy. The following code segment shows how everything is distributed:

Code View:

```
switch (streamSelect)
{
  case "left" :
        outStream="left";
        inStream1="right";
        inStream2="top";
        inStream3="bottom";
        break;
  case "right" :
        outStream="right";
        inStream1="left";
        inStream2="top";
        inStream3="bottom";
        break;
  case "top" :
        outStream="top";
        inStream1="left";
        inStream2="right";
        inStream3="bottom";
        break;
  case "bottom" :
        outStream="bottom";
        inStream1="left";
        inStream2="top";
        inStream3="right";
        break;
}
```

As with the Deux application, all of the data for the cases in the switch statement are supplied from the server-side script. Instead of just two, `left` and `right`, now the `streamSelect` values are `left, right, top`, and `bottom`. These names *do not* act to position the different streams. Rather they are simply four convenient names. The next code segment shows how the streams are distributed:

```
//Publish local video
netOut=new NetStream(nc);
netOut.attachAudio (mic);
netOut.attachCamera (cam);
vidOut.attachCamera (cam);
netOut.publish (outStream, "live");

//Play streamed video
netIn1=new NetStream(nc);
vidBottom.attachNetStream (netIn1);
netIn1.play (inStream1);

netIn2=new NetStream(nc);
vidLeft.attachNetStream (netIn2);
netIn2.play (inStream2);

netIn3=new NetStream(nc);
vidRight.attachNetStream (netIn3);
```

```
netIn3.play (inStream3);
```

The name `vidOut` is the name of the video object in the top position, and `netOut` is the name of the outgoing stream from the local client's camera and microphone. The other incoming streams are allocated to the left, right, and bottom video objects, and the stream names are those generated in the `switch` statement.

5.6.3. Building the Conference Application

You're ready to build the application. Follow these steps to build the application, beginning with placing objects on the Stage:

1. Open a new Flash file (ActionScript 3.0). Open the Property inspector, and in the Document Class text box, type **Quad**. Save the file as Quad.fla.

2. Using Figure 5-8 as a guide, use the Rectangle tool to draw four 80 x 100 rectangles on the Stage. The exact positions of the rectangles are shown in Figure 5-8. (Do not enter the x and y coordinates as labels. They are positioning guides only.)

Figure 5-8. Placement of backdrops

3. Place your logo in the center of the backdrops, as shown in placement_of_backdrops, and the Quad text label at the top. Save the file.

4. Open a new ActionScript file and save it as Quad.as.

5. Enter the script in Example 5-4, and resave the file.

Example 5-4. Quad.as

Code View:

```
package
{
  import flash.display.Sprite;
  import flash.display.MovieClip;
  import flash.events.NetStatusEvent;
  import flash.net.NetConnection;
  import flash.net.NetStream;
  import flash.media.Camera;
  import flash.media.Microphone;
  import flash.media.Video;
  import flash.net.Responder;

  public class Quad extends Sprite
  {
        private var nc:NetConnection;
        private var good:Boolean;
        private var netOut:NetStream;
        private var netIn1:NetStream;
        private var netIn2:NetStream;
        private var netIn3:NetStream;
        private var cam:Camera;
        private var mic:Microphone;
        private var responder:Responder;
        private var vidOut:Video;
        private var vidBottom:Video;
        private var vidLeft:Video;
        private var vidRight:Video;
        private var outStream:String;
        private var inStream1:String;
        private var inStream2:String;
        private var inStream3:String;

        public function Quad ()
        {
             var rtmpNow:String="rtmp://192.168.0.11/quad";
             nc=new NetConnection;
             nc.connect (rtmpNow);
             nc.addEventListener (NetStatusEvent.NET_STATUS,getStream);
        }

        private function getStream (e:NetStatusEvent):void
        {
             good=e.info.code == "NetConnection.Connect.Success";
             if (good)
             {
                  responder=new Responder(streamNow);
                  nc.call ("streamSelect",responder);
             }
        }
```

```
private function streamNow (streamSelect:String):void
{
     setCam ();
     setMic ();
     setVid ();

     switch (streamSelect)
     {
          case "left" :
               outStream="left";
               inStream1="right";
               inStream2="top";
               inStream3="bottom";
               break;
          case "right" :
               outStream="right";
               inStream1="left";
               inStream2="top";
               inStream3="bottom";
               break;
          case "top" :
               outStream="top";
               inStream1="left";
               inStream2="right";
               inStream3="bottom";
               break;
          case "bottom" :
               outStream="bottom";
               inStream1="left";
               inStream2="top";
               inStream3="right";
               break;
     }

     //Publish local video
     netOut=new NetStream(nc);
     netOut.attachAudio (mic);
     netOut.attachCamera (cam);
     vidOut.attachCamera (cam);
     netOut.publish (outStream, "live");

     //Play streamed video
     netIn1=new NetStream(nc);
     vidBottom.attachNetStream (netIn1);
     netIn1.play (inStream1);

     netIn2=new NetStream(nc);
     vidLeft.attachNetStream (netIn2);
     netIn2.play (inStream2);

     netIn3=new NetStream(nc);
     vidRight.attachNetStream (netIn3);
     netIn3.play (inStream3);
}

private function setCam ():void
{
     cam=Camera.getCamera();
     cam.setMode (80,100,12);
```

```
            cam.setQuality (0,80);
            cam.setKeyFrameInterval(12);
        }

        private function setMic ():void
        {
            mic=Microphone.getMicrophone();
            mic.rate=11;
            mic.setSilenceLevel (12,2000);
        }

        private function setVid ():void
        {
            vidOut=new Video(80,100);
            addChild (vidOut);
            vidOut.x=235;
            vidOut.y=43;

            vidLeft=new Video(80,100);
            addChild (vidLeft);
            vidLeft.x=89;
            vidLeft.y=145;

            vidRight=new Video(80,100);
            addChild (vidRight);
            vidRight.x=379;
            vidRight.y=145;

            vidBottom=new Video(80,100);
            addChild (vidBottom);
            vidBottom.x=235;
            vidBottom.y=253;
        }
    }
}
```

6. Create a new folder and name it Quad. Place the folder in your server-side applications folder.

7. Open the Deux.asc file, and change the first line to read,

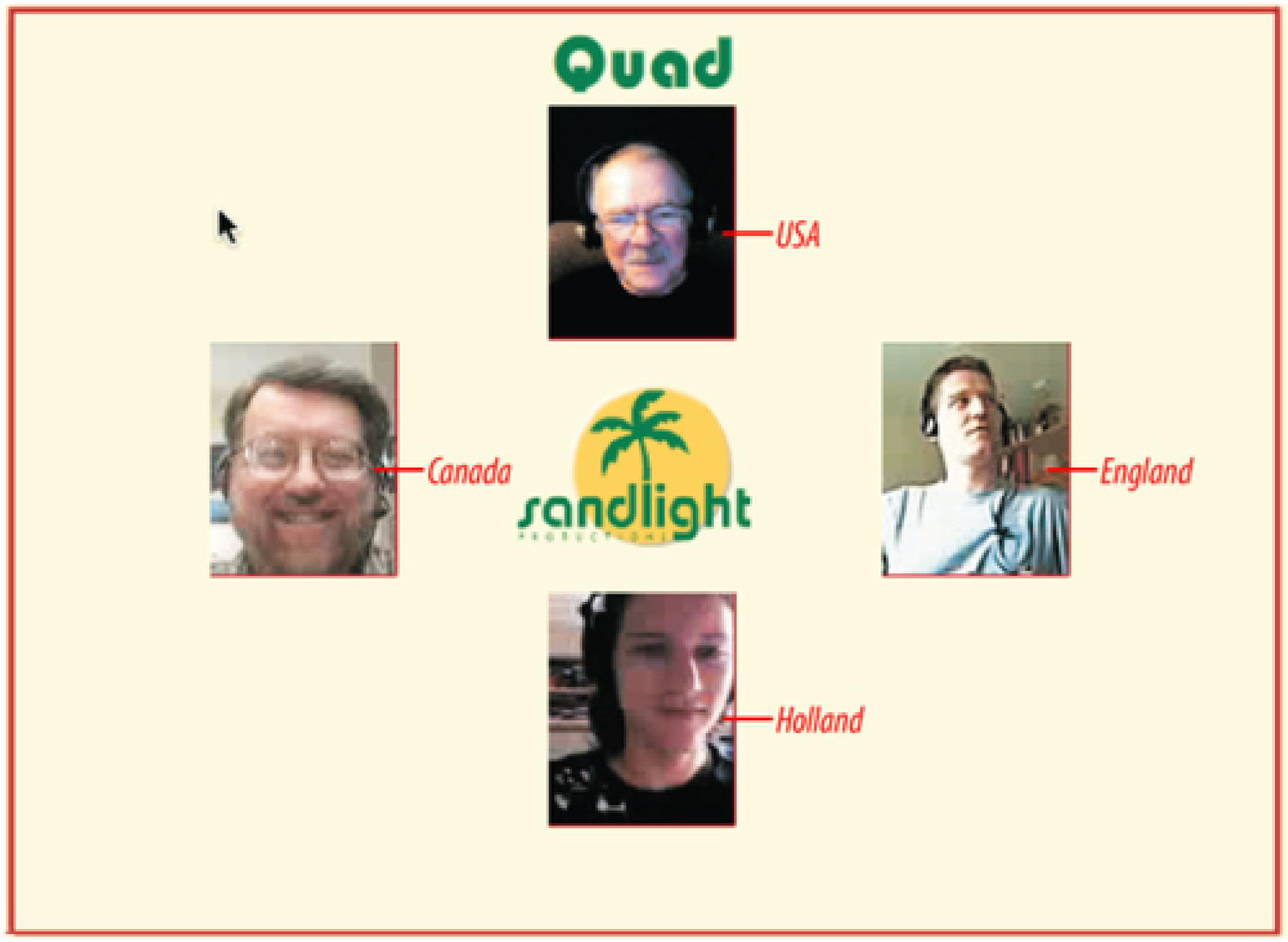
```
vidStreams=["bottom", "top",right","left"];
```

8. Choose File Save As and save the file as Quad.asc in the Quad folder

9. Reopen the Quad.fla file (or select the Quad.fla tab if it's already open) and select File Publish to generate the HTML, SWF, and JavaScript files. Once you place those files on your Web server, you're all set for your online conference.

Invite three friends with video cameras and microphones to connect to your application. As each one enters the online conference, they will see the others and the green backdrops where no one has yet appeared. If someone leaves the conference, you will see that person's last image in the video window until a new person comes in. Figure 5-9 shows four people from four different countries chatting on the application. The perspective shows the local user at the top position; however, the three others also see themselves in the top position. The placement of video instances generates an arrangement of equality (or power) not possible in nonvirtual space.

Figure 5-9. Four-way conference in four countries



The purpose of this application has been to keep it simple. You can add different enhancements, such as using the `Video.clear()` method to clear the screen after a visitor has left the conference.

5.7. Moving On to More Server-Side Applications

Now that you've seen how to work with audio and video on the client-side (with a little help from the server-side), the next step is to work with audio and video on the server-side. Chapter 6, "Broadcasting and Server-Side Bandwidth Control," introduces another type of A/V application-broadcast, better described as a one-to-many application. The main focus, though, will be on introducing the server-side of the FMS3 equation. You'll begin with some simple uses of the server-side. But as you'll quickly see, once you start using server-side script, you'll have far more options and control over the applications you build.

Chapter 6. Broadcasting and Server-Side Bandwidth Control

6.1. Casting Many Streams

The idea of making a worldwide broadcasting system may seem a bit ambitious, but using FMS3, it's actually quite easy. All you need to do is send out a stream containing audio and video using a "Studio" module, and have a "TV" module set to pick it up. You can even set up the bandwidth allocation so that the Studio module uses more client-to-server bandwidth, and the TV module is weighted so that most of the bandwidth is allocated from the server to the client. In addition, you can switch between live and recorded streams and bring in live reports from remote locations. Before you go out and hire a news team, remember that FMS3 generates a new stream for every client connected to the application. FMS3 has both bandwidth and connected-client limits, and so depending on your version of FMS3, you'll be able to serve more or fewer clients. However, you can create a broadcasting application that optimizes bandwidth so that you can serve as many clients as possible while still delivering server side the best A/V possible.

Chapter 6. Broadcasting and Server-Side Bandwidth Control

6.1. Casting Many Streams

The idea of making a worldwide broadcasting system may seem a bit ambitious, but using FMS3, it's actually quite easy. All you need to do is send out a stream containing audio and video using a "Studio" module, and have a "TV" module set to pick it up. You can even set up the bandwidth allocation so that the Studio module uses more client-to-server bandwidth, and the TV module is weighted so that most of the bandwidth is allocated from the server to the client. In addition, you can switch between live and recorded streams and bring in live reports from remote locations. Before you go out and hire a news team, remember that FMS3 generates a new stream for every client connected to the application. FMS3 has both bandwidth and connected-client limits, and so depending on your version of FMS3, you'll be able to serve more or fewer clients. However, you can create a broadcasting application that optimizes bandwidth so that you can serve as many clients as possible while still delivering server side the best A/V possible.

6.2. Switching Cameras

As anyone who has worked in a TV studio knows, the producer switches between cameras, with most productions having about three different cameras. So before getting to the studio, this section looks at switching from one camera to another-a key skill for setting up your own worldwide video-broadcasting corporation.

6.2.1. Selecting a Camera

Like most of the changes discussed in this book, a key change in ActionScript 3.0 alters the way to reference a camera. In prior versions of ActionScript, a camera was referenced by an integer parameter. However, in ActionScript 3.0, the reference to a camera is now a string parameter. The new way, though, still uses an integer classed as a string to do the job. This means that you make the reference as `"4"` instead of 4. The number is the position of the camera in the Settings window. The camera at the top of the list is in the "0" position and each camera is referenced by its relative position in the list. Thus, a reference to the fifth camera is to `"4"` just like a 0-based array. The following line shows how to reference the fifth camera in the list:

```
var cam2:Camera=Camera.getCamera("4");
```

The Camera class instance, `cam2`, is now the fifth camera in the Settings window. (Actually, it's the camera that uses the fifth driver.)

You might remember that if no parameter is used, the camera defaults to the last selected camera. For example, if you last used the camera in position `"5"` (sixth on the list) the line,

```
var cam1:Camera=Camera.getCamera();
```

would mean that a reference to `cam1` would address the sixth camera rather than the first one in the Flash Player Settings window-the one in position number `"0"`.

6.2.2. Two-Camera Switch

To see how to switch cameras, the following application requires a minimum of two cameras. If you only have one camera, borrow one or use one of the virtual Webcams you can find online. (An interesting camera driver treats your screen as a Webcam, providing you with a nice screen capture tool. It's available for Windows at

`http://www.hmelyoff.com/index.php?section=8`

Follow these simple steps to build the application:

1. Open a new Flash file (ActionScript 3.0) and save it as CameraSwitch.fla.

2. Open the Library panel and drag a Button component into the Library.

3. In the Document Class text box in the Property inspector, type **CameraSwitch** and save the file.

4. Open a new ActionScript file and save it as CameraSwitch.as.

5. In the CameraSwitch.as file, add the script in Example 6-1 and save the file.

Example 6-1. CameraSwitch.as

Code View:

```
package
{
    import flash.media.Camera;
    import flash.media.Microphone;
    import flash.media.Video;
    import flash.display.Sprite;
    import flash.net.NetConnection;
    import flash.net.NetStream;
    import flash.events.MouseEvent;
    import fl.controls.Button;

    public class CameraSwitch extends Sprite
    {
          private var cam1:Camera;
          private var cam2:Camera;
          private var btn1:Button;
          private var btn2:Button;
          private var mic:Microphone;
          private var vid:Video;


          public function CameraSwitch ()
          {
               vid=new Video(320,240);
               vid.x=100,vid.y=30;
               addChild (vid);

               mic=Microphone.getMicrophone();
               cam1=Camera.getCamera();
               cam1.setMode (320,240,15);
               cam2=Camera.getCamera("4");
               cam2.setMode (320,240,15);

               btn1=new Button();
               btn1.width=70;
               btn1.x=vid.x;
               btn1.y=vid.y+vid.height+10;
               btn1.label="Camera 1";
               addChild (btn1);
               btn1.addEventListener (MouseEvent.CLICK,camOne);

               btn2=new Button();
               btn2.width=70
               btn2.x=btn1.x+100;
               btn2.y=vid.y+vid.height+10;
               btn2.label="Camera 2";
```

```
                addChild (btn2);
                btn2.addEventListener (MouseEvent.CLICK,camTwo);

            }
            private function camOne (e:MouseEvent)
            {
                vid.attachCamera (cam1);
            }
            private function camTwo (e:MouseEvent)
            {
                vid.attachCamera (cam2);
            }
    }
}
```

Before you test the program, check your Flash Player Settings window and make sure that it lists at least two drivers. If you only have two drivers, determine which one is the default, and then enter the string number of the other. (For example, if you only have two drivers and the first one is the default ["0"], then set `cam2` to "1".)

The following sections feature studio applications. If you only have a single camera, you probably won't want to add a second camera. However, if you want to turn your studio into a more robust application, then just incorporate the code in Example 6-1.

6.3. The Minimum Studio

Building a minimal broadcast studio is a very simple operation requiring nothing more than a module with a stream that sends out live audio and video. To make it even easier, especially in the long run, this application includes a reusable class for doing nothing more than setting up streams going out and streams coming in. This class works like a utility class that you can re-use to simplify the work of creating streaming audio/video. A nice-to-have minimum is a video object to enable the "anchor" to see him- or herself during the broadcast. Likewise, an "on the air" message lets the broadcaster know that the studio is connected to FMS3.

6.3.1. Utility Stream Class

The first order of business is to create the utility class to ease the work of streaming audio and video in and out. It uses the following classes:

Classes

NetConnection

NetStream

Camera

Microphone

Video

This utility class extends the NetStream class. The class has two public methods, one to steam audio/video out (`streamOut`) and one to stream it in (`streamIn`). Remember that a stream needs a microphone instance, a camera instance, and a string for the stream name to stream out A/V. The `streamOut` method has parameters for each of those three elements.

Conversely, to play a video coming in, all you need is the name of the video object and the stream name. Because this first minimalist application is only streaming video out, you may wonder, why bother with the `streamIn` method? The purpose of this class is not to make *one* application smoother to develop, but *several*. Later in the chapter, you'll learn how to create a "receiver" that works like a TV set, and you'll be glad that the `streamIn` method is included. (In later chapters, you'll wonder how you've lived this long without it.)

Finally, one element you may not be familiar with is the `super` statement. This utility class is an extension of the NetStream class. To create a `NetStream` instance, you need to include the `NetConnection` parameter. The super statement invokes the superclass NetConnection so that when you construct a `StreamAV` instance, you can include the necessary `NetConnection` parameter. Follow these steps to create this utility class.

1. Open a new ActionScript file and save it as StreamAV.as.

2. Add the code in Example 6-2, and resave the file in the same folder you plan to use for the rest of the application.

Example 6-2. StreamAV.as

```
Code View:
package
{
  import flash.net.NetConnection;
  import flash.net.NetStream;
  import flash.media.Camera;
  import flash.media.Microphone;
  import flash.media.Video;

  public class StreamAV extends NetStream
  {
        public function StreamAV (nc:NetConnection)
        {
             super (nc);
        }

        public function streamOut (mic:Microphone,cam:Camera,stream:String):voic
        {
             this.attachAudio (mic);
             this.attachCamera (cam);
             this.publish (stream,"live");
        }

        public function streamIn (vid:Video,stream:String):void
        {
             vid.attachNetStream (this);
             this.play (stream);
        }
  }
}
```

6.3.2. The Basic Broadcast Studio

The first Studio module you'll develop is very simple. All it does is to show the "TV Anchor" (you) and the "on the air" message to let you know that you're connected. Chapter 5, "Two-Way Audio-Video Communications," showed how to stream in and out. This application module simply streams out-however, with help from the StreamAV class.

Classes

- NetConnection
- NetStatusEvent
- Camera
- Microphone

Video

Sprite

TextField

StreamAV

Why isn't the NetStream class listed in this application? The new utility class StreamAV handles that for you. In both the Broadcast and TV applications, you use the StreamAV class; as a result, the applications require fewer imports because they're handled by StreamAV.

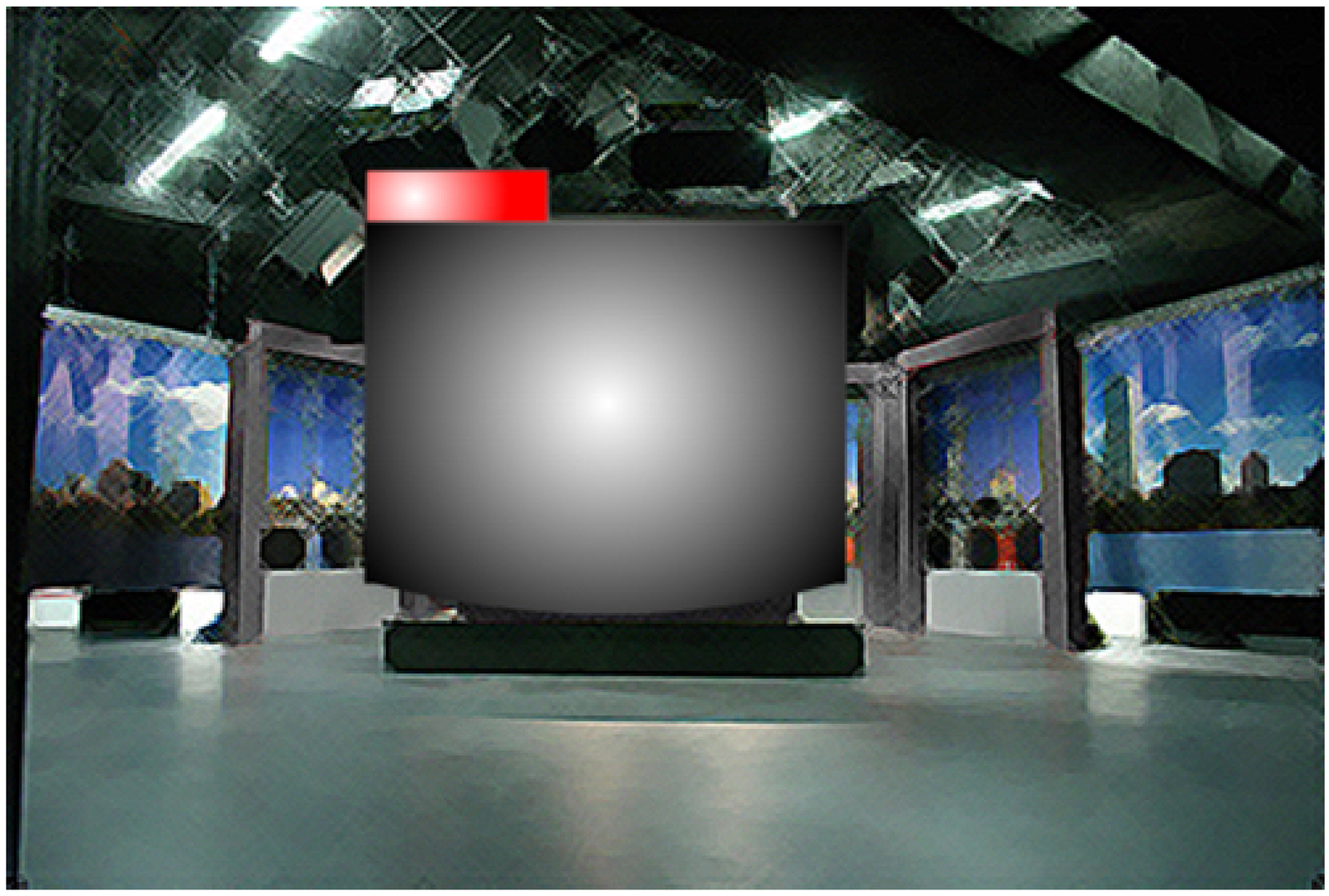
I created a "studio" background to have some fun, but you can make your own studio as simple or complex as you like. The following instructions provide the minimal steps for creating a studio. You will need a folder named studio on the server side in your FMS3 applications folder, and you might want to add it before you get started. Later in the chapter, you'll add features to the basic studio, so look at Figure 6-1 to get an idea of a layout and the room you'll need. For now, you need only the elements in the minimal list above.

1. Create a new folder named studio and add it to your FMS3 applications folder

2. Open a new Flash file and save it as Broadcast.fla in the same folder as the StreamAV.as file.

3. Optionally, add a studio background using a photograph, existing drawing, or drawing you create using Flash tools. Using the Rectangle tool, add a rectangle with the dimensions, w=200, h=150, positioned at x=150, y=190 over the background. This will serve as the monitor screen. In the upper left corner positioned exactly on top of the first rectangle, add a second rectangle with the dimensions w=75, h=21, y=21, y=68.5. Figure 6-1 shows the general idea, and Figure 6-2 shows what you'll see when you execute the script. Save the Broadcast.fla file with your client-side folders.

Figure 6-1. Studio backdrop



4. Open a new ActionScript file and save it as Broadcast.as in the same folder with the StreamAV.as and Broadcast.fla files.

5. In the Broadcast.as file, add the script in Example 6-3 and resave the file.

Example 6-3. BroadCast.as

Code View:

```
package
{
  import flash.net.NetConnection;
  import flash.events.NetStatusEvent;
  import flash.media.Camera;
  import flash.media.Microphone;
  import flash.media.Video;
  import flash.display.Sprite;
  import flash.text.TextField;

public class Broadcast extends Sprite
{
      private var nc:NetConnection;
      private var cam:Camera;
      private var mic:Microphone;
      private var vid:Video;
      private var rtmpNow:String;
      private var onAir:String;
      private var broadcast:StreamAV;
      private var good:Boolean;
      private var txtField:TextField;

      public function Broadcast ()
```

```
        {
            //Set Camera and Microphone

            nc=new NetConnection ;
            nc.addEventListener (NetStatusEvent.NET_STATUS,checkConnect);
            cam=Camera.getCamera();
            cam.setMode(240,180,15);
            cam.setQuality(0,85);
            mic=Microphone.getMicrophone();
            mic.rate=11;
            vid=new Video(240,180);
            vid.x=150;
            vid.y=90;
            addChild(vid);
            txtField=new TextField();
            txtField.x=150;
            txtField.y=75;
            addChild(txtField);
            rtmpNow="rtmp://192.168.0.11 /studio";
            nc.connect (rtmpNow,"anchor");
        }

        //Connection
        private function checkConnect (e:NetStatusEvent)
        {
            good=e.info.code == "NetConnection.Connect.Success";
            if (good)
            {
                txtField.text="On The Air";
                broadcast=new StreamAV(nc);
                vid.attachCamera (cam);
                broadcast.streamOut (mic,cam,"presentation");
            }
        }
    }
}
```

When you run this module, you should be able to see yourself in the studio monitor and a small "on the air" sign indicating that you're connected to the server. The single stream is sending out the input from your current camera and microphone. That's all it does and nothing more. Figure 6-2 shows the module in action. (Note the "on the air" sign above the video.)

Figure 6-2. Broadcast studio

Now that you have a broadcast studio, you need a module to receive the streamed materials. This is essentially a TV set where users can watch what's coming from your video streamcast.

6.3.3. The Minimum TV

If you thought that the minimum studio was easy, the minimum TV is even easier. All you need is a video object on the Stage and you're good to go. These minimum elements are the entire collection of classes you'll need:

Classes

NetConnection

NetStatusEvent

Video

Sprite

StreamAV

Like the studio, you can make your TV as simple or elaborate as you want. All you need, though, is the code. I used an old-fashioned TV set as the background, but otherwise, it's pretty simple. Follow these steps to create the TV:

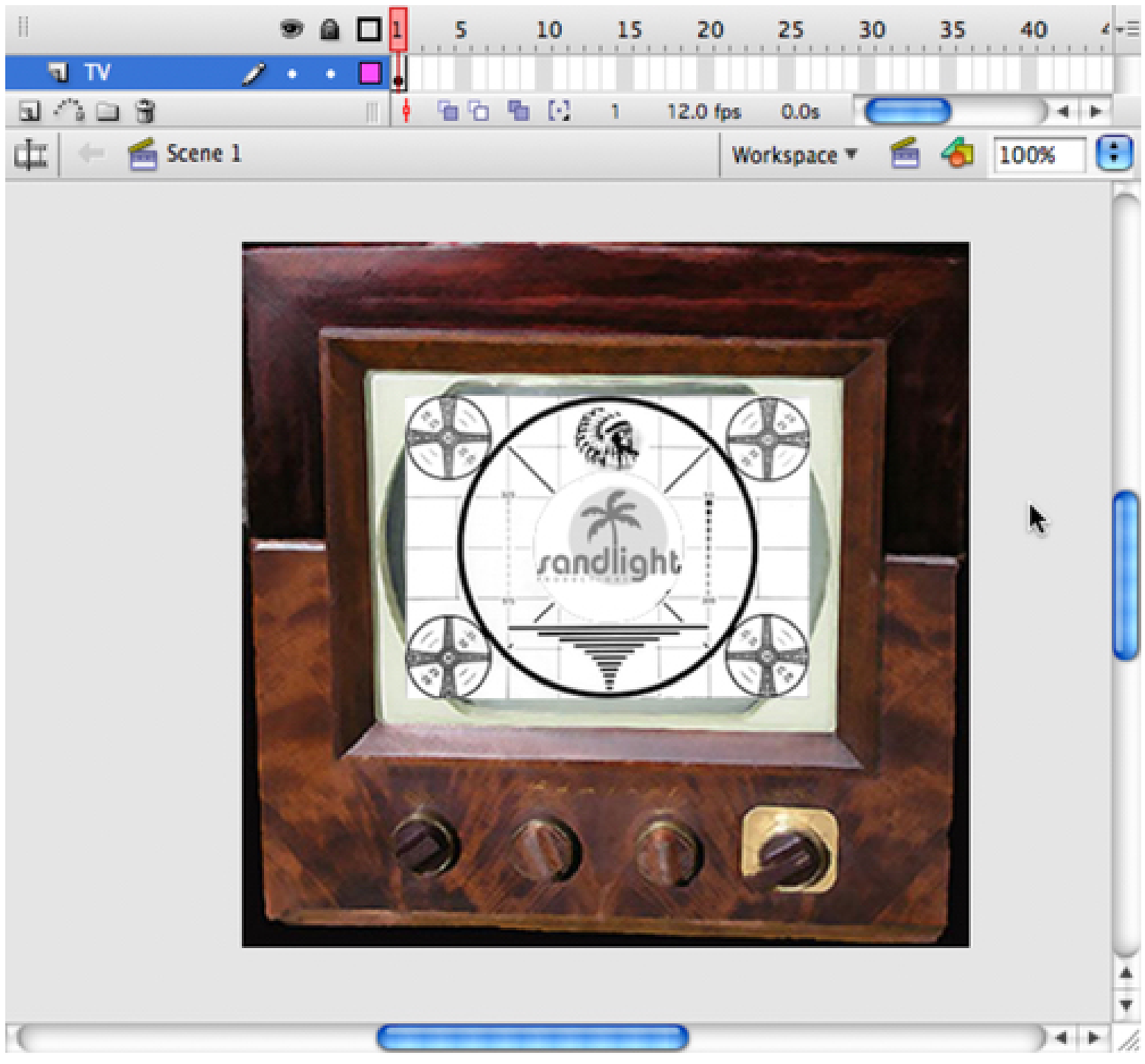
1. Open a new Flash file and set the Stage size to 360 x 350.

2. (You can set your Stage smaller, but you cannot set it to smaller than 215 x 138. Smaller than that size, it's too small for the Settings window and it won't work. For this particular application, you don't want it smaller than 215 x 150 because your video size is set to 200 x 150. You can set it larger as well. This particular size is based on the size of the TV image I made.)

3. Save the file as TV.fla in the same folder where you saved the StreamAV.as file. If you want to keep the TV and Broadcast modules in separate folders, that's fine. Just be sure that each folder contains the Stream.as file.

4. Optionally, create a graphic TV set to act as a backdrop for your image, such as the one shown in Figure 6-3. The Test pattern is set to 200 x 150 and is positioned exactly where the video image goes. When everything is positioned where you want it, save the FLA file.

Figure 6-3. Stage for TV module



5. Open a new ActionScript file and save it as TV.as in the same folder with TV.fla.

6. In the TV.as file, add the Example 6-4 script and resave the file.

Example 6-4. TV.as

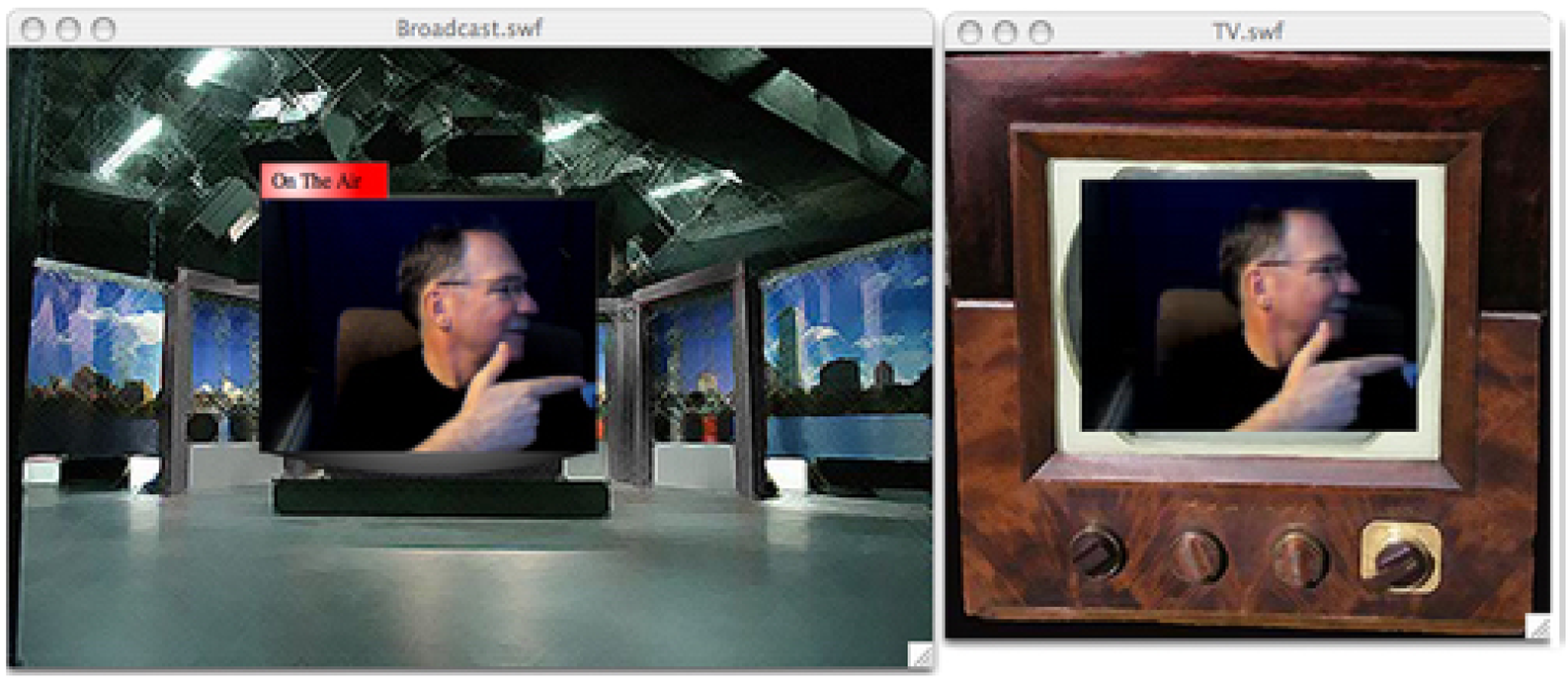
```
Code View:
package
{
  import flash.net.NetConnection;
  import flash.events.NetStatusEvent;
  import flash.media.Video;
  import flash.display.Sprite;

  public class TV extends Sprite
  {
        private var nc:NetConnection;
        private var vid:Video;
        private var rtmpNow:String;
        private var tvSet:StreamAV;
        private var good:Boolean;


        public function TV()
        {
             nc=new NetConnection ;
             nc.addEventListener (NetStatusEvent.NET_STATUS,checkConnect);
             vid=new Video(200,150);
             vid.x=81;
             vid.y=76;
             addChild (vid);
             rtmpNow="rtmp://192.168.0.11 /studio";
             nc.connect (rtmpNow,"viewer");
        }

        //Connection
        private function checkConnect (e:NetStatusEvent)
        {
             good=e.info.code == "NetConnection.Connect.Success";
             if (good)
             {
                   tvSet=new StreamAV(nc);
                   tvSet.streamIn (vid,"presentation");
             }
        }
   }
}
```

Depending on the image you used, you may have to change the x and y settings for the video instance. I found that by creating a 200 x 150 rectangle and moving it on the Stage to where I wanted the video to appear, I could use the values in the Property inspector to get the exact coordinates. Once those coordinates were found, I just deleted the rectangle.

When you've finished, test your broadcast system. First, load the Broadcast module and run it. Once it shows

"on the air," you're ready to run the TV module. If your TV displays the image that you broadcast from your Studio module, everything is working. Likewise, you should be able to hear what is sent from the studio. However, you will neither see nor hear what the user of the TV module does. (Just like real TV!) Figure 6-4 shows what you should see when both modules are working correctly:

Figure 6-4. Studio stream seen on TV module



You may have noticed in the script that the `NetConnection.connect()` parameter included an additional value. Until now, all that you've seen in the `connect()` method is the name of the application (for example, studio). However, both modules have an optional parameter. In this case, the optional parameter identifies the user as either the "anchor" or the "viewer," depending on which module is running. On the server side, which you will learn about in the next section, this optional parameter helps to uniquely identify who is using the application. You may want to limit who can and cannot use your application, and the use of a second parameter with a user identifier can be very handy.

6.4. Introduction to the Server Side

The script you've been using up to this point is ActionScript 3.0, which includes what was known as Client Side Communication ActionScript prior to ActionScript 3.0. It represents a subset of ActionScript. The set of classes, properties, methods, and events used with Flash Media Server 3 in ActionScript 3.0 that deal with communicating with FMS 3 can usefully be referred to as "Client Side ActionScript," or more simply, CSAS.

The language used to prepare scripts for launching from the server is called Server-Side ActionScript (SSAS). Like ColdFusion, PHP, and .NET, SSAS scripts have their source in the server, and communicate directly with the server. SSAS is both similar to, and different from, CSAS and ActionScript in general. In fact, it's closer to JavaScript and ActionScript 1.0 than it is to ActionScript 3.0. The special SSAS classes differentiate it from the latest version of JavaScript, but you can expect it to develop over time into the ECMAScript standard, as ActionScript 2.0 and ActionScript 3.0 have been doing. (For an update on ActionScript 3.0, see www.adobe.com/products/flex/technologies/.)

6.4.1. Using SSAS

Writing scripts in SSAS is different than writing scripts in the Actions panel or in a Script window. Keep in mind that the script runs from the server, and relies on events occurring on the server. To get started, let's look at three of these events:

- Starting the application.
- Connecting to the application.
- Disconnecting from the application.

Accepting the client is a method, and not an event. But accepting a connection is an essential part of the connection process. So in addition to the three events, the connection acceptance also is included. While working on the server-side script, you'll want to debug it as well. To do this, as you do with client-side scripts, you'll use the `trace()` statement. The trace information does not appear in the Output window. Instead, you must open your FMS3 Application console; then select View Applications Live Log. The Live Log window displays the results of executing the `trace()` statements, as shown in Figure 6-5. However, first you need to write the SSAS file.

1. Open a new ActionScript Communication file and save it as studio.asc in the server-side folder named studio.
2. In the studio.asc file, add the code in Example 6-5 and save the file again.

Example 6-5. studio.asc

```
application.onAppStart = function()
{
  trace("The Broadcast has begun!");
};

application.onConnect = function(client,nameNow)
{
  client.name=nameNow;
  trace(client.name + " has connected");
  application.acceptConnection(client);
  trace(client.name + " has been accepted");
}

application.onDisconnect = function(client)
{
  trace(client.name + " has left.");
}
```

The script uses the SSAS class Application for reporting what was occurring on the server side. The three events trapped include the application's start (`onAppStart`), the connection of the client (`onConnect`), and the client's disconnect (`onDisconnect`). In addition, the `Application.acceptConnection()` allows the client to make the connection. If the `Application.rejectConnection()` method were used as a response to that event, the client would be disconnected.

Once you have saved the server-side script, open the Broadcast module. The first three `trace( )` results will appear in the FMS3 Administration console, as shown in Figure 6-5. When the application starts, the first message comes up. That will be the only time you see the application startup message because clients can come and go without restarting the application. Then you will see that the client named "anchor" has connected and then the acceptance message. Next, when you open the Viewer module, the console displays the "viewer" connect message and has its connection accepted. Finally, the viewer leaves, and then the anchor. At this point the application named studio shows that 0 clients are connected.

Figure 6-5. Trace statements in the Administration console

Now you have successfully created a server-side script that accepts connections. Its sole purpose is to give you a guided tour of what happens when a client connects to a server and how the server can recognize unique

clients.

6.4.2. Setting Server-Side Client Bandwidth Limits

Let's do something practical with a server-side (ASC) script. Because controlling bandwidth is so important when working with streaming audio and video, you'll look at how to set bandwidth limits for different clients. This application will have only two client names, anchor and viewer, so you can set the bandwidth by looking at the name of the client. The Broadcast module will only be sending streams out, and the Viewer module will be receiving streams but not sending any out.

Using the SSAS `Client.setBandwidthLimit()` method, you can specify how much server-to-client and client-to-server bandwidth the different clients can use. The general format of the setting statement is:

```
client.setBandwidthLimit(s2c, c2s);
```

where *s2c* is server-to-client and *c2s* is client-to-server. By setting different values to variables that will be assigned as parameters, you can trap the name of the client and, depending on whether it's the anchor or a viewer instance, assign each the appropriate amount. The following script sets up the bandwidth limits for the two modules. When you save it, just overwrite the existing studio.asc file using the code in Example 6-6.

Example 6-6. studio.asc

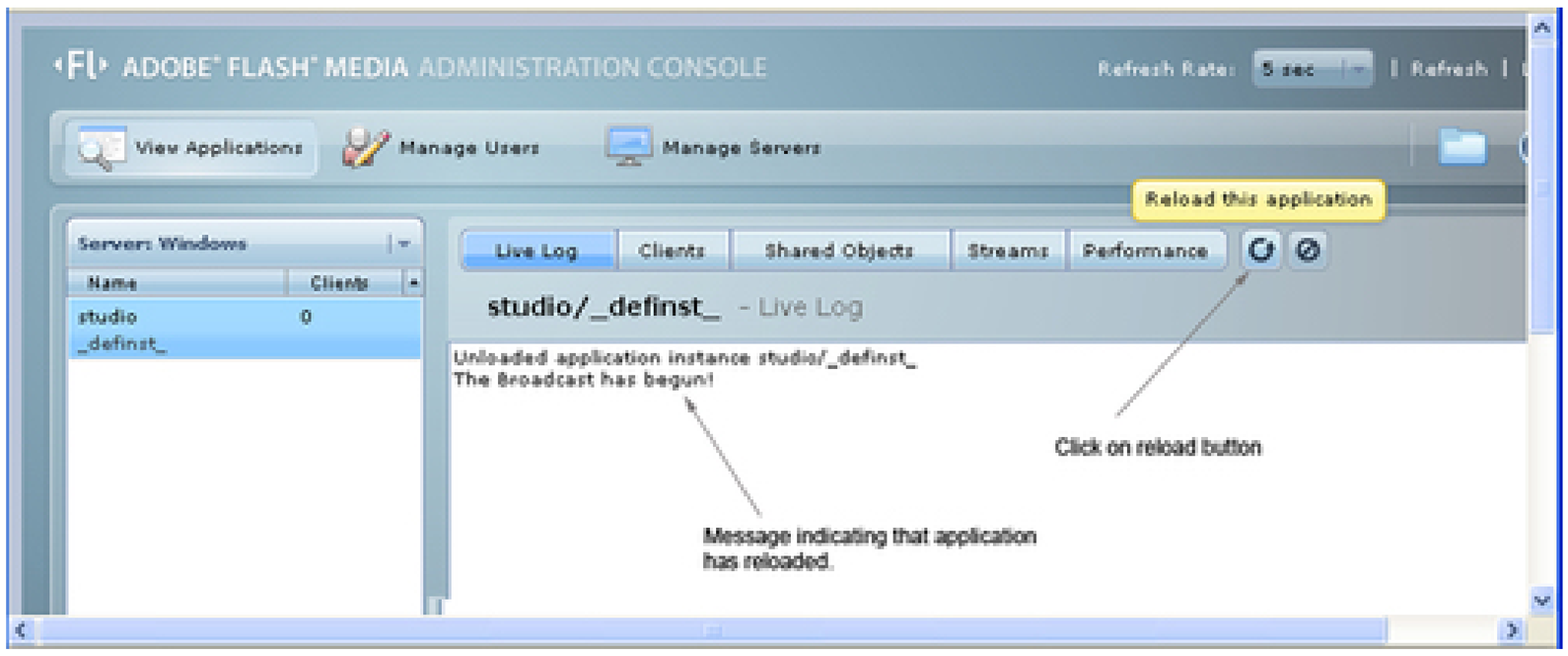
Code View:

```
application.onAppStart = function()
{
  trace("The Broadcast has begun!");
};

application.onConnect = function(client,nameNow)
{
  client.name=nameNow;
  trace(client.name + " has connected");
  application.acceptConnection(client);
  trace(client.name + " has been accepted");
  var s2c;
  var c2s;
  if(client.name == "anchor")
  {
        s2c=1000;
        c2s= 100000/8;
  }
  else
  {
        s2c=100000/8;
        c2s=1000;
  }
  client.setBandwidthLimit(s2c,c2s);

  trace(s2c);
  trace("server to client=" + client.getBandwidthLimit(1));
  trace(c2s);
  trace("client to server=" + client.getBandwidthLimit(0));

}

application.onDisconnect = function(client)
```

```
{
  trace(client.name + " has left.");
}
```

To test the new SSAS file, you must restart the application. You can do that in the FMS3 console by selecting the View Applications tab and then clicking the Reload Application button (the circle with the arrowhead). Figure 6-4 shows the selected application and the button to reload the application. This removes the old version of the application from the FMS3 cache and loads the new version of the file. (This is an easy step to forget, leaving you wondering why your changes haven't taken effect.)

Figure 6-6. Reload applications before running new server-side scripts



Reloading the application disconnects the clients. So once you've reloaded, run both the anchor and viewer instances again to see the feedback in the FMS3 console. The following output appears once both modules are running:

```
anchor has connected
anchor has been accepted
1000
server to client=1000
12500
client to server=12500
viewer has connected
viewer has been accepted
12500
server to client=12500
1000
client to server=12500
```

Now, both module instances have different bandwidth limits. The client-to-server bandwidth on the Broadcast module is higher because the module must send streams from the client to the server, but it doesn't get any back from the viewer instance. By the same token, the Viewer module needs more bandwidth coming from the server to the client because it only receives streams, but sends none. When you make such adjustments, you

can optimize the amount of bandwidth you use in your application.

6.4.3. Launching a Server Action from the Client Side

As you've seen so far, all of the events used in the server-side script were launched from the server using Application events. You can also initiate a server method from the client side and pass variables entered from the client side to a server-side script with the `NetConnection.call()` function. For example, to dynamically change the value of the bandwidth, you can send a message along with values that will launch the server-side function and pass values to set the bandwidth limit. The general format for making such a call when you're not expecting a return but you're sending data is:

```
nc.call("ssFunction", null, arg1, arg2);
```

If all you want to do is to fire off the function without passing any values to it, you can use the format:

```
nc.call("ssFunction");
```

On the server-side script, you will need a function that "catches" the call from the client side. The client class on the server side is the "catcher" of function calls. The format is:

```
client.funcName = function(arg) {
//code
}
```

For example:

```
client.setBW = function(ser2c, c2ser) {
client.setBandwidthLimit(ser2c, c2ser);
}
```

If you're running a broadcast studio, you need an easy way to change the bandwidth limits so that you can easily test your application and set the limits to what you want. The actual client that you control is the Broadcast Studio module, so you don't have to worry about the TV module. It's going to have to stay set using literals. Otherwise the viewers could ramp up their own bandwidth limits way beyond what you want them to have. To make it easy, this next application, which is a modification of the original Broadcast module, uses a List component. However, before you can control anything on the server, you need to change the studio.asc script to the one listed in Example 6-7.

Example 6-7. studio.asc

```
application.onAppStart = function()
{
  trace("The Broadcast has begun!");
};

application.onConnect = function(client,nameNow)
{
  client.name=nameNow;
  trace(client.name + " has connected");
  application.acceptConnection(client);
  trace(client.name + " has been accepted");
  client.setBW=function(ser2c, c2ser)
  {
        client.setBandwidthLimit(ser2c, c2ser);
        trace("New server to client= " + (ser2c*8)+"bps");
        trace("New client to server= " + (c2ser*8)+"bps");
  }
}

application.onDisconnect = function(client)
{
  trace(client.name + " has left.");
}
```

The server-side function receives the values passed from the client in the two parameters, `ser2c` and `c2ser`. These values are then used to set the bandwidth limit. The output is shown in the FMS3 console in bps, or bits per second. Remembering that everything is done in bytes in FMS, the bytes are converted to bits (your actual bandwidth unit of measure) by multiplying the bytes by 8. (Remember that each byte is 8 bits, so 100 bytes is 800 bits.)

To get both sides working together, you need to change the Broadcast.as and Broadcast.fla files. However, because the change is fairly significant, follow these steps to create a new application from the old one:

1. Open the Broadcast.fla and Broadcast.as files, and save them as BroadcastSetBW.fla and BroadcastSetBW.as, respectively.

2. Select the BroadcastSetBW.fla tab, and in the Property inspector change the name of the Document class to BroadcastSetBW. Now, drag a List component into the Library and save the file.

3. Select the BroadcastSetBW.as file tab and enter the code in Example 6-8. (You can just edit the code from the original Broadcast.as file.) Save the file.

Example 6-8. BroadcastSetBW.as

Code View:

```
package
{
    import fl.data.DataProvider;
    import fl.controls.List;
    import flash.net.NetConnection;
    import flash.events.NetStatusEvent;
```

```
import flash.media.Camera;
import flash.media.Microphone;
import flash.media.Video;
import flash.display.Sprite;
import flash.text.TextField;
import flash.events.Event;

public class BroadcastSetBW extends Sprite
{
      private var nc:NetConnection;
      private var cam:Camera;
      private var mic:Microphone;
      private var vid:Video;
      private var rtmpNow:String;
      private var onAir:String;
      private var broadcast:StreamAV;
      private var good:Boolean;
      private var txtField:TextField;
      private var list:List;
      private var stuff:DataProvider;
      private var cli2sv:uint;

      public function BroadcastSetBW ()
      {
           //Set Camera and Microphone
           nc=new NetConnection ;
           nc.addEventListener (NetStatusEvent.NET_STATUS,checkConnect);
           cam=Camera.getCamera();
           cam.setMode (200,150,15);
           cam.setQuality (0,85);
           mic=Microphone.getMicrophone();
           mic.rate=11;

           vid=new Video(200,150);
           vid.x=150;
           vid.y=90;
           addChild (vid);

           txtField=new TextField();
           txtField.x=155;
           txtField.y=72;
           addChild (txtField);

           doList ();

           rtmpNow="rtmp://192.168.0.11 /studio";
           nc.connect (rtmpNow,"anchor");
      }

      //Connection
      private function checkConnect (e:NetStatusEvent)
      {
           good=e.info.code == "NetConnection.Connect.Success";
           if (good)
           {
                txtField.text="On The Air";
                broadcast=new StreamAV(nc);
                vid.attachCamera (cam);
                broadcast.streamOut (mic,cam,"presentation");
```

```
            }
        }

        //Create List and Data Provider
        private function doList ():void
        {
            list=new List();
            stuff=new DataProvider();
            stuff.addItem ({label:"Select BW"});
            stuff.addItem ({label:"Minimum", data:(10000/8)});
            stuff.addItem ({label:"Small", data:(20000/8)});
            stuff.addItem ({label:"Medium", data:(50000/8)});
            stuff.addItem ({label:"Large", data:(100000/8)});
            list.dataProvider=stuff;
            list.move(355,90);
            list.width=65;
            list.selectedIndex=0;
            addChild (list);
            list.addEventListener(Event.CHANGE,changeBW);
        }

        private function changeBW(e:Event):void
        {
            cli2sv=list.selectedItem.data;
            nc.call("setBW",null,100,cli2sv);
            list.selectedIndex=0;
        }
    }
}
```

To better see the relationship between client-side and server-side scripts, Figure 6-7 shows the relevant parts of each with arrows indicating the path the parameters take. The trace statements show that the parameters were correctly passed (or not!) Because the Broadcast module won't be receiving any streams, the setting for the server-to-client is static. Thus, the only the server-side bandwidth setting that needs to be controlled is the client-to-server bandwidth limit. To see the code making calls to the server in action, the next section explains how to set up a totally dynamic studio, including the module for changing the bandwidth limits.

Figure 6-7. Client-side call to server-side function

```
87          private function changeBW(e:Event):void
88          {
89              cli2sv=list.selectedItem.data;
90              nc.call("setBW",null,100,cli2sv);          Client side
91              list.selectedIndex=0;
92          }
```

```
12      client.setBW=function(ser2c, c2ser)                Server side
13      {
14          client.setBandwidthLimit(ser2c, c2ser);
15          trace("New server to client= " + (ser2c*8)+"bps");
16          trace("New client to server= " + (c2ser*8)+"bps");
17      }
18  }
```

To better see the values passed through the List component, Table 6-1 shows all of the values used.

Table 6-1. List labels and data values

Label	Data Value
Minimum	10000/8
Small	20000/8
Medium	50000/8
Large	100000/8

All of the values are set with bps (bits per second) in mind because bandwidth is measured in bits, not bytes. (I know it's redundant, but it does bear repeating.) So, the minimum is providing a rate of only 10 Kbps and the maximum is 100 Kbps. Figure 6-8 shows what you can expect to see in the Broadcast module:

Figure 6-8. Broadcast module showing List component for changing bandwidth limits

To see the effect that changing bandwidth limits has on the studio and TV modules, test the TV module with the new studio module. The default bandwidth setting works well. But as you keep decreasing bandwidth for the studio to use to send a stream to the TV module, you will experience freezing. As you increase the bandwidth limits, the freezing and latency disappears. Try experimenting with this module to get a feel for how the changes, especially the latency and freezes, affect the application.

6.5. Dynamic Camera, Microphone, and Bandwidth Controls

You may have wondered why so few values were assigned to the Camera and Microphone objects in the initial broadcast applications. That was simply to focus on the minimal code for creating a broadcast-type application. This section adds several controls to the Broadcast module. However, adding these controls will eliminate the "studio" atmosphere to focus better on the different controls and what they do. You'll be able to see the image on the TV module change dynamically as you change the different controls. In that way, you can better see how all the settings are important. However, you will be using exactly the same TV module with no changes at all. Because this application uses several different controls, primarily user interface components, you will see far more classes and events in the class list than in previous examples. As with the other examples, you'll use the StreamAV class; so make sure that the StreamAV.as file is in the same folder as the new Broadcast module.

Classes

NetConnection

NetStatusEvent

Camera

Microphone

Video

Sprite

TextField

StreamAV

DataProvider

MovieClip

List

TextInput

Button

Slider

SliderDirection

SliderEvent

RadioButton

RadioButtonGroup

TextField

Event

MouseEvent

Objects

Static Text fields

Drawn rectangle

backdrop_mc (instance)

As you will see, the bulk of the work is done in the file, BroadcastStudio.as, dynamically assigning different UI components to the Stage. However, to get started, you'll need to place several different Static text labels on the Stage and create a MovieClip object and a rectangle drawing using the Tools panel. To help you, Figure 6-9 shows all of the objects with their x and y coordinates (x,y) and the dimensions and positions of two drawn objects.

Figure 6-9. Objects and instance names

All of the labels are created with Static text, using 11-point Arial Black for the bold and Arial for the regular font labels. (You can substitute any fonts you want, but that may throw off the coordinates a bit which just requires a little position tweaking.) Follow these steps to create this application:

1. Open a new Flash file (ActionScript 3.0) and save it as BroadcastStudio.fla in the same folder with a copy of the StreamAV.as file. Set the Stage size to 600 x 500 and save the file again. This example uses a light gray background color; you can change it or leave it as white. However, be sure to make it light enough so that black text is visible. All of the labels should be black because the dynamic labels are black.

2. Using Figure 6-9 as a guide, add the different Static text labels using the Text tool. The easiest way to work with the labels is to first type them in Static text, and then using the Property inspector, position them to the correct coordinates. For example, a position of 20,60 would mean that you type 20 in the Property inspector's X: window and 60 in the Y: window. Save the file.

3. Using the Rectangle tool, draw a 200 x 150 rectangle with rounded corners set to 9. Fill it with a black-to white radial (select this radial fill in the Swatches panel) with no stroke. Select the rectangle and press F8 to convert it to a MovieClip object. In the Property inspector, name the instance `backdrop_mc`. Position the object as indicated in Figure 6-9.

4. With the Rectangle tool, just above the movie clip, draw a 75 x 21 rectangle, and apply a red radial fill with no stroke. Position the rectangle as indicated in Figure 6-9. Save the file.

5. Open the Components panel and add the following components to the Library panel: Slider, TextInput, RadioButton, Button, and List. (This process is easier if you separate the Library panel from the dock and then open the Component panel and drag the components to the bottom part of the Library panel.) Save the file. This should be all you need to do with the FLA file.

6. Open a new ActionScript file and save it as BroadcastStudio.as in the same folder as BroadcastStudio.fla.

7. In the BroadcastStudio.as file, enter the script in Example 6-9 and save the file again.

Example 6-9. BroadcastStudio.as

Code View:

```
package
{
  import fl.data.DataProvider;
  import flash.display.MovieClip;
  import fl.controls.List;
  import fl.controls.TextInput;
  import fl.controls.Button;
  import fl.controls.Slider;
  import fl.controls.SliderDirection;
  import fl.events.SliderEvent;
  import fl.controls.RadioButton;
  import fl.controls.RadioButtonGroup;
  import flash.net.NetConnection;
  import flash.events.NetStatusEvent;
  import flash.media.Camera;
  import flash.media.Microphone;
  import flash.media.Video;
  import flash.display.Sprite;
  import flash.text.TextField;
  import flash.events.Event;
  import flash.events.MouseEvent;

  public class BroadcastStudio extends Sprite
  {
        private var nc:NetConnection;
        private var cam:Camera;
```

```
        private var mic:Microphone;
        private var vid:Video;
        private var rtmpNow:String;
        private var onAir:String;
        private var broadcast:StreamAV;
        private var good:Boolean;
        private var txtField:TextField;

        //BW
        private var list:List;
        private var stuff:DataProvider;
        private var cli2sv:uint;

        //Camera
        private var qualBw:TextInput;
        private var qualQual:TextInput;
        private var modeW:TextInput;
        private var modeH:TextInput;
        private var modeFPS:TextInput;
        private var keyFrameI:TextInput;
        private var camBtn:Button;

        //Microphone
        private var rateGroup:RadioButtonGroup;
        private var r5:RadioButton;
        private var r8:RadioButton;
        private var r11:RadioButton;
        private var r22:RadioButton;
        private var r44:RadioButton;
        private var silLevelL:TextInput;
        private var silLevelT:TextInput;
        private var silL:uint;
        private var silT:uint;
        private var mGain:Slider;
        private var micBtn:Button;

        public function BroadcastStudio ()
        {
             //Set Camera and Microphone
             nc=new NetConnection ;
             nc.addEventListener (NetStatusEvent.NET_STATUS,checkConnect);

             //Camera default
             cam=Camera.getCamera();
             cam.setMode (192,144,15);
             cam.setQuality (0,85);
             //
             qualBw=new TextInput();
             addChild (qualBw);
             qualBw.x=20,qualBw.y=35;
             qualQual=new TextInput();
             addChild (qualQual);
             qualQual.x=(qualBw.x+qualBw.width+5),qualQual.y=35,qualQual.width=30;
             modeW=new TextInput();
             modeH=new TextInput();
             //
             modeFPS=new TextInput();
             addChild (modeW);
             modeW.width=30,modeW.x=20,modeW.y=95;
```

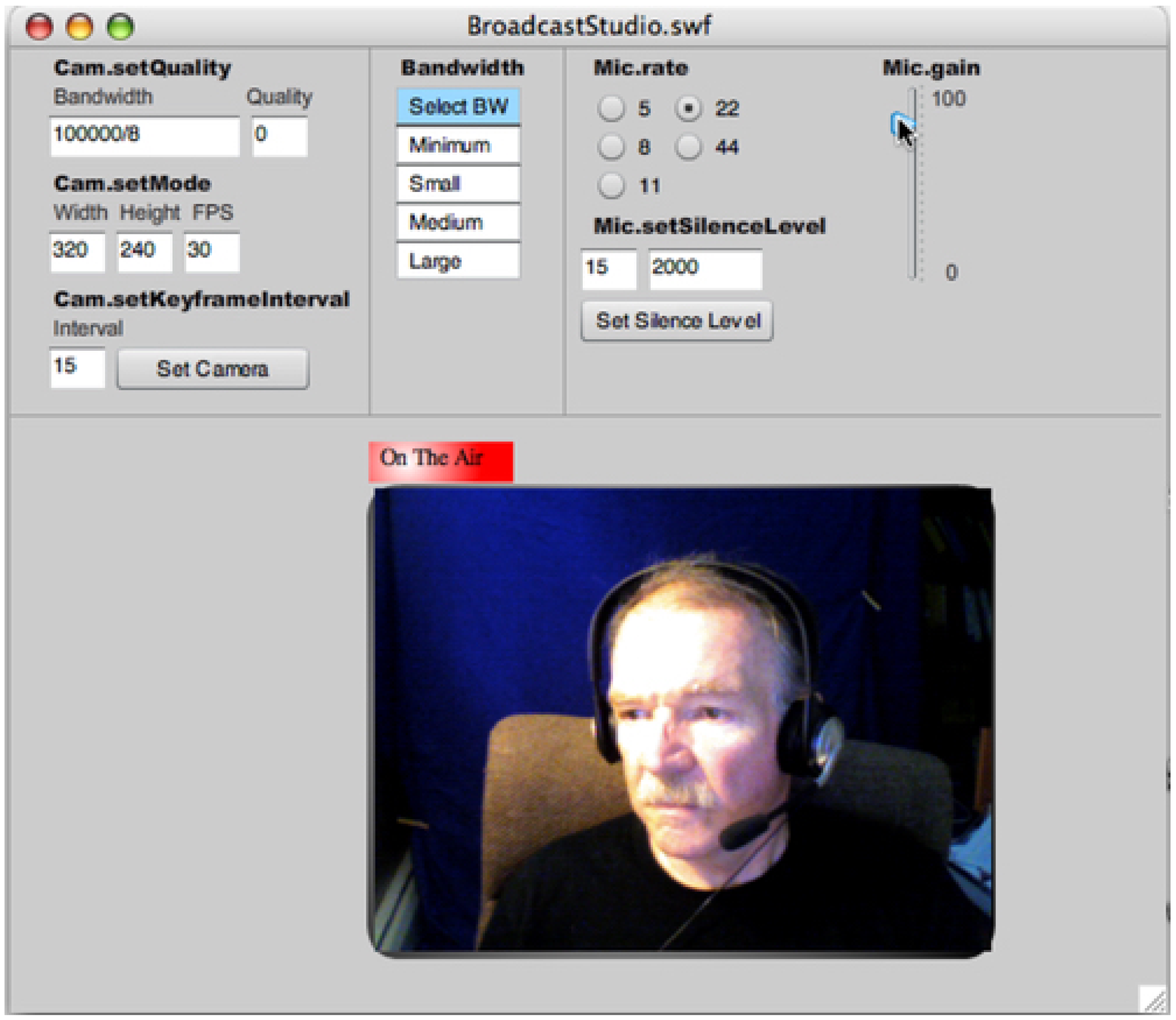
```
addChild (modeH);
modeH.width=30,modeH.x=(modeW.x+modeW.width+5),modeH.y=95;
addChild (modeFPS);
modeFPS.width=30,modeFPS.x=(modeH.x+modeH.width+5),modeFPS.y=95;
keyFrameI=new TextInput();
addChild (keyFrameI);
keyFrameI.width=30,keyFrameI.x=20, keyFrameI.y=155;
camBtn=new Button();
camBtn.label="Set Camera";
addChild (camBtn);
camBtn.x=(keyFrameI.width+keyFrameI.x+5),camBtn.y=155,
camBtn.addEventListener (MouseEvent.CLICK,setCam);

//Microphone default
mic=Microphone.getMicrophone();
mic.rate=11;
micBtn=new Button();
micBtn.addEventListener (MouseEvent.CLICK,setMic);
rateGroup=new RadioButtonGroup("Rate");
r5=new RadioButton(),r8=new RadioButton();
r11=new RadioButton(),r22=new RadioButton();
r44=new RadioButton();
r5.move(300,20),r8.move(300,40);
r11.move(300,60),r22.move(340,20);
r44.move (340,40);
r5.label="5",r8.label="8",r11.label="11";
r22.label="22",r44.label="44";
r5.group=rateGroup,r8.group=rateGroup;
r11.group=rateGroup, r22.group=rateGroup;
r44.group=rateGroup;
addChild(r5),addChild(r8);
addChild(r11),addChild(r22);
addChild (r44);
rateGroup.addEventListener (MouseEvent.CLICK,setMic);

silLevelL=new TextInput();
silLevelL.move (296,104);
silLevelL.width=30;
addChild (silLevelL);
silLevelT=new TextInput();
silLevelT.move (331,104);
silLevelT.width=60;
addChild (silLevelT);
micBtn=new Button();
micBtn.move (296,130);
micBtn.label="Set Silence Level";
addChild (micBtn);
micBtn.addEventListener (MouseEvent.CLICK,setSilence);

mGain=new Slider();
mGain.move (470,20);
mGain.minimum=0;
mGain.maximum=100;
mGain.tickInterval=5;
mGain.value=70;
mGain.width=100;
addChild (mGain);
mGain.direction= SliderDirection.VERTICAL;
mGain.addEventListener (SliderEvent.CHANGE,setGain);
```

```
            //Video default
            vid=new Video(cam.width,cam.height);
            vid.x=189;
            vid.y=228;
            addChild (vid);

            txtField=new TextField();
            txtField.x=190;
            txtField.y=205;
            addChild (txtField);

            doBW ();

            rtmpNow="rtmp://192.168.0.11 /studio";
            nc.connect (rtmpNow,"anchor");
      }

      //Cwonnection
      private function checkConnect (e:NetStatusEvent)
      {
            good=e.info.code == "NetConnection.Connect.Success";
            if (good)
            {
                  txtField.text="On The Air";
                  broadcast=new StreamAV(nc);
                  vid.attachCamera (cam);
                  broadcast.streamOut (mic,cam,"presentation");
            }
      }

      //Create List and Data Provider
      private function doBW ():void
      {
            list=new List();
            stuff=new DataProvider();
            stuff.addItem ({label:"Select BW"});
            stuff.addItem ({label:"Minimum", data:(10000/8)});
            stuff.addItem ({label:"Small", data:(20000/8)});
            stuff.addItem ({label:"Medium", data:(50000/8)});
            stuff.addItem ({label:"Large", data:(100000/8)});
            list.dataProvider=stuff;
            list.move (200,20);
            list.width=65;
            list.selectedIndex=0;
            addChild (list);
            list.addEventListener (Event.CHANGE,changeBW);
      }

      //Change the client bandwidth
      private function changeBW (e:Event):void
      {
            cli2sv=list.selectedItem.data;
            nc.call ("setBW",null,100,cli2sv);
            list.selectedIndex=0;
      }

      private function setCam (e:MouseEvent):void
      {
            cam.setQuality (Number(qualBw.text),Number(qualQual.text));
```

```
            cam.setMode
(Number(modeW.text),Number(modeH.text),Number(modeFPS.text));
            cam.setKeyFrameInterval (Number(keyFrameI.text));
            backdrop_mc.width=(cam.width+6);
            backdrop_mc.height=(cam.height+6);
            setVid ();
        }

        private function setMic (e:MouseEvent):void
        {
            mic.rate=Number(rateGroup.selection.label);
        }

        private function setSilence (e:MouseEvent):void
        {
            silL=Number(silLevelL.text);
            silT=Number(silLevelT.text);
            mic.setSilenceLevel (silL,silT);
        }

        private function setGain (e:SliderEvent):void
        {
            mic.gain=e.target.value;
        }

        private function setVid ():void
        {
            vid.width=cam.width;
            vid.height=cam.height;
        }
    }
}
```

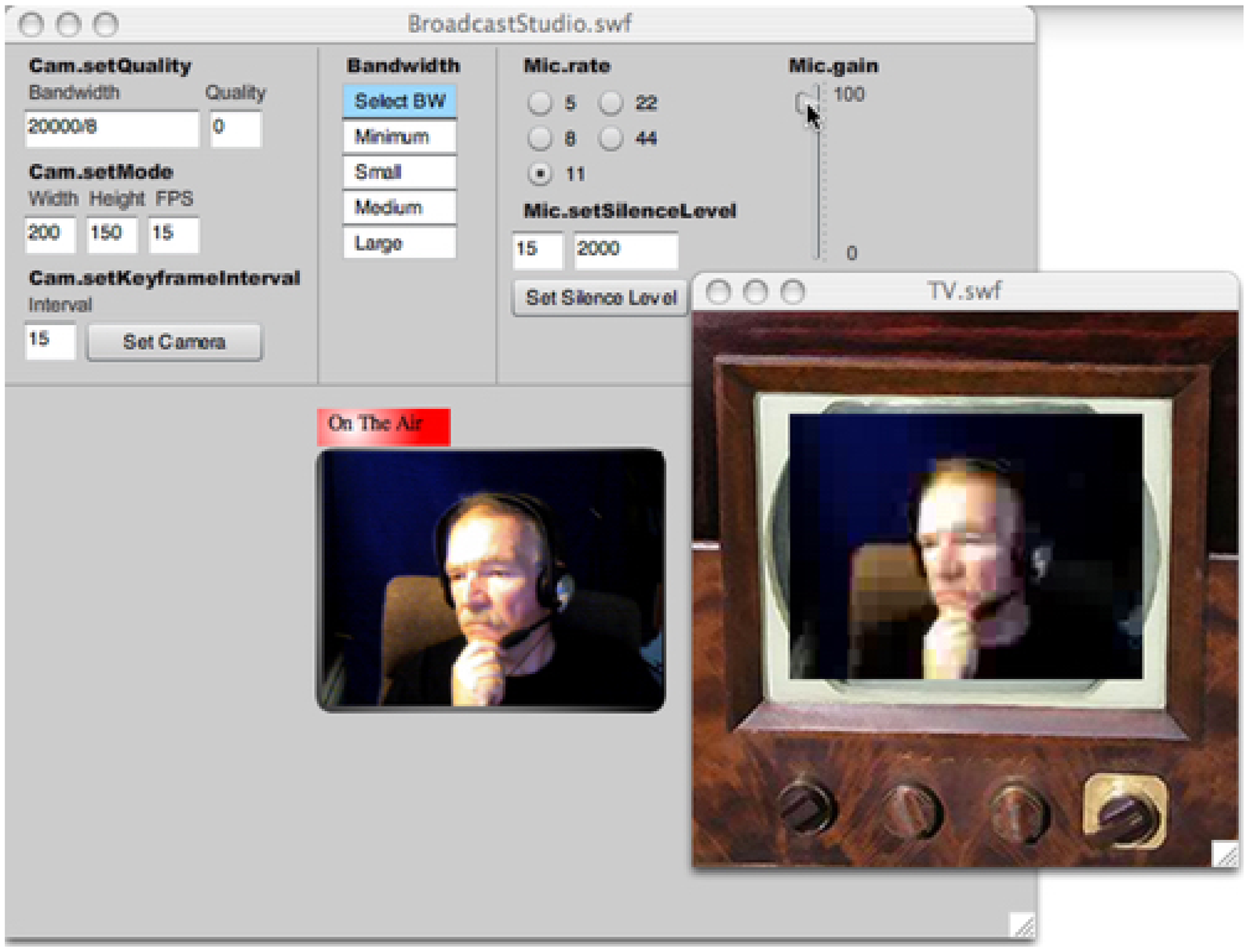
When you've finished, test the script. Figure 6-10 shows what you will see with the different options filled in and the video screen changed from the default value:

Figure 6-10. Dynamic changes in BroadcastStudio module



With this control panel, you have lots of different options. The camera controls are set with a single button; so be sure not to leave any blank options when you press the Set Camera button. The current setting shows that the mode quality is set around the available bandwidth. It gives the best picture for 100 Kbits; however, the image seen in the Broadcast Studio module only reflects the different image size generated by the link between the mode setting and the size of the video window. However, if you set the quality to 10 or 100, it won't change what you see locally. Where it makes a difference is in the stream that is sent to the TV module. For example, Figure 6-11 shows a very low bandwidth setting in the camera quality bandwidth (only 20 Kbits); if you compare the image in the studio with that on the TV module, you can see the difference.

Figure 6-11. Studio changes only show up in streamed view



Connecting your camera directly to the video object gives the best possible picture. You need to use the BroadcastStudio module with the TV module to really see all of the effects. For example, try optimizing quality, FPS, and keyframe interval (use a low interval value such as 5 or less) and then set the bandwidth to minimum or small. Your picture won't take long to freeze up!

This application is meant to show how much you can get with the least possible bandwidth. For example, if you're expecting more viewers, you might want to change the video resolution on the TV module to 160 x 120 (the default) or even less from 200 x 150, as was done in the Quad application (see "Section 5.6.3" in Chapter 5.) Above all, this application demonstrates how to set up a video and audio with lots of different user interface controls in ActionScript 3.0, and the effect of each setting or combination of settings on a streaming application.

6.6. Bandwidth Checker

Until the release of FMS3, the only way to check your system's current bandwidth was with server-side code and algorithms that would extract bandwidth based on pings or some similar measure. A very interesting new feature is a call to the server, `checkBandwidth`. Because the call is to a built-in function, you can check your bandwidth with client-side code. The `checkBandwidth` command is invoked with a call to the method from a NetConnection object:

```
//nc is a NetConnection instance
nc.call ("checkBandwidth",null);
```

It looks exactly like a call to a server-side function, but with the `checkBandwidth` method, you don't write any server-side code. While you don't write the method, it still belongs to the server-side Client class. You have to write a special class to get the returned information using two specific methods, named `onBWCheck` and `onBWDone`. The class is invoked as a `NetConnection.client` instance.

6.6.1. Setting Up the Client Class

The first step is to write a client class that contains the methods required to check the bandwidth and return a value. Also, by including a public getter method, you can get the returned bandwidth for your main class. The client class can go into the same file as your main class; but by keeping it separate, it can be easily identified and reused for any application where you need to get the bandwidth. To get started, create the BWClient.as file shown in Example 6-10. It is the class to be instantiated by the `NetConnection.client` instance.

Example 6-10. BWClient.as

Code View:

```
package
{
    class BWClient
    {
          private var currentBW:Number;

          public function onBWCheck (... rest):Number
          {
               return 0;
          }
          public function onBWDone (... rest):void
          {
               if (rest.length>0)
               {
                    currentBW=rest[0];
               }
          }
          public function sendInfo ():Number
          {
               return currentBW;
          }
    }
}
```

The `...(rest)` parameter accepts any number of comma-delimited arguments. The first method, `onBWCheck` returns a 0 because it has to return something. Its role is to start the ball rolling on the call to `NetConnection.call("checkBandwidth", null)`. The key method in this class is the one that recognizes that the operation is completed, `onBWDone`. The rest[0] element contains the current bandwidth speed in kilobits. To be able to do something in the main class, the `sendInfo()` method is a public one that is used to retrieve the current bandwidth by the main class.

Next, the CheckBW class calls the client class (BWClient), extracts the bandwidth, and shows the user what it is. Of course, the bandwidth could be used to do anything from adjusting the camera and sound settings, to making other adjustments to optimize a FMS3 application. Enter the code in Example 6-11 and save it as CheckBW.as.

Example 6-11. CheckBW.as

Code View:

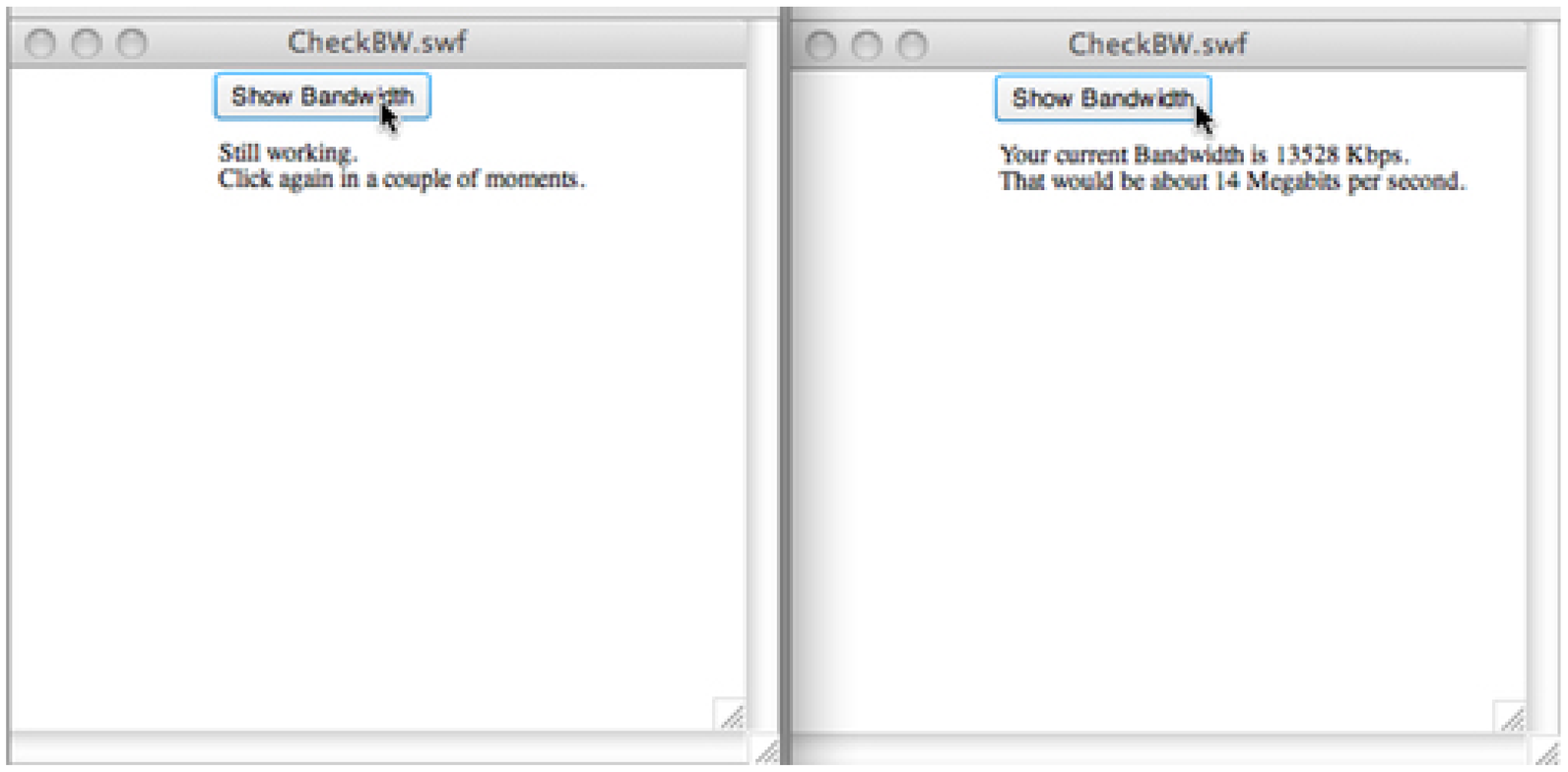
```
package
{
    import flash.display.Sprite;
    import flash.net.NetConnection;
    import flash.events.NetStatusEvent;
    import flash.events.MouseEvent;
    import flash.text.TextField;
    import flash.text.TextFieldAutoSize;
    import fl.controls.Button;

    public class CheckBW extends Sprite
    {
          private var nc:NetConnection;
          private var rtmpNow:String;
          private var good:Boolean;
          private var btn:Button;
          private var txtFld:TextField;
          private var checkUp:Object;
          private var currentBW:Number;

          public function CheckBW ()
          {
               btn=new Button();
               btn.x=200, btn.y=50;
               btn.label="Show Bandwidth";
               addChild (btn);
               btn.addEventListener (MouseEvent.CLICK,showBW);

               txtFld=new TextField();
               txtFld.multiline=true;
               txtFld.x=200, txtFld.y=80;
               txtFld.autoSize = TextFieldAutoSize.LEFT;
               addChild (txtFld);

               rtmpNow="rtmp://192.168.0.11/studio/";
               nc=new NetConnection ;
               nc.addEventListener (NetStatusEvent.NET_STATUS,checkNcall);
               nc.client=new BWClient();
               nc.connect (rtmpNow,"BWsnooper");
               checkUp=new Object();
```

```
            }
            private function checkNcall (e:NetStatusEvent):void
            {
                  good=e.info.code == "NetConnection.Connect.Success";
                  if (good)
                  {
                        nc.call ("checkBandwidth",null);
                  }
            }

            private function showBW (e:MouseEvent)
            {
                  checkUp.see=nc.client.sendInfo();
                  if (checkUp.see >0)
                  {
                        currentBW=nc.client.sendInfo();
                        txtFld.text="Your current Bandwidth is "+currentBW +" Kbps.\n";
                        txtFld.appendText ("That would be about
"+Math.round(currentBW/1000)+ " Megabits per second.");
                  }
                  else
                  {
                        txtFld.text="Still working.\n";
                        txtFld.appendText ("Click again in a couple of moments.");
                  }
            }
      }
}
```

The CheckBW class calls the `checkBandwidth` method as soon as the connection is established. However, to see what the bandwidth is, the user clicks a button that calls the `showBW` method. It uses the `BWClient` instance (`nc.client`) to call the getter method, `sendInfo()`. Using an object instance property (`checkUp.see`), the program stores the returned bandwidth value and passes it to a numeric variable, `currentBW`. In turn, the information is passed to a text field for the user's information.

One of the interesting features is that even though the `checkBandwidth` method is called as soon as a connection is established, it takes a while for the bandwidth information to be returned to the client. So the output message in the text field advises the user to be patient and give the button another click if the process is incomplete. Figure 6-12 shows the output before and after the bandwidth value has been returned.

Figure 6-12. Before and after bandwidth has been returned



As you can see in the CheckBW class, the RTMP connection is to the same application that has been used with the other applications in this chapter (studio). However, you can use it with any application simply by using the same RTMP address.

6.7. Conclusion

Now that you have a start with writing your own server-side code, you are in a position to expand the functionality of your FMS3 applications. With the ability to create your own server-side code, you can control far more aspects of your application and make them more interactive or simply improve their functionality by tweaking the bandwidth.

The new `Client.checkBandwidth` method, while requiring no server-side script, acts as though a server-side script does indeed exist. In every other way on the client side, from calling the function to getting information back, the `checkBandwidth` method must be treated just as though you can see it in a server-side script. Think of it as an invisible method residing on the server-side because that is exactly what it is and how it responds to client-side calls.

Chapter 7. Working With Server-Side Files: The File Class

7.1. Recording Data

This chapter explores recording data using Flash Media Server 3. We'll look at the server side File class. This class allows developers to create applications that can store information in text (TXT) files, as well as binary and UTF-8 files, in a secure location on the server side along with the ASC files and recorded FLV files.

NOTE

As you start using more server-side scripts, you'll see a number of differences between ActionScript 3.0 on the client-side and server-side ActionScript (SSAS). ActionScript 3.0 meets the ECMAScript 4 standard, while SSAS is JavaScript with a few additional classes for Flash Media Server. Most important is the use of data typing-stating a data type for your objects. In ActionScript 3.0, the correct way to type data in declarations, parameters, and function returns is to place a colon (:) in front of the data type. For example, the following are all accepted in ActionScript 3.0, while none are accepted in SSAS:

```
private var myList: Array=new Array("A", "B", "C");
protected var manyWords:String;
public function shedArea(H:Number, W:Number):Number
private function getData( ) :void ....
```

So as you get into the habit of typing data in your client-side code, remember that your server-side data is not typed. The data typing issue can be seen in the code in previous chapters that discussed server-side ActionScript, but not in this chapter, because so much of the code is server side ActionScript, The point is re-emphasized here, because so much of this chapter's code is server-side ActionScript that requires no typing when objects or variables are declared.

Chapter 7. Working With Server-Side Files: The File Class

7.1. Recording Data

This chapter explores recording data using Flash Media Server 3. We'll look at the server side File class. This class allows developers to create applications that can store information in text (TXT) files, as well as binary and UTF-8 files, in a secure location on the server side along with the ASC files and recorded FLV files.

NOTE

As you start using more server-side scripts, you'll see a number of differences between ActionScript 3.0 on the client-side and server-side ActionScript (SSAS). ActionScript 3.0 meets the ECMAScript 4 standard, while SSAS is JavaScript with a few additional classes for Flash Media Server. Most important is the use of data typing-stating a data type for your objects. In ActionScript 3.0, the correct way to type data in declarations, parameters, and function returns is to place a colon (:) in front of the data type. For example, the following are all accepted in ActionScript 3.0, while none are accepted in SSAS:

```
private var myList: Array=new Array("A", "B", "C");
protected var manyWords:String;
public function shedArea(H:Number, W:Number):Number
private function getData( ) :void ....
```

So as you get into the habit of typing data in your client-side code, remember that your server-side data is not typed. The data typing issue can be seen in the code in previous chapters that discussed server-side ActionScript, but not in this chapter, because so much of the code is server side ActionScript, The point is re-emphasized here, because so much of this chapter's code is server-side ActionScript that requires no typing when objects or variables are declared.

7.2. The File Class

The File class is handy for storing data without using middleware or an SQL database. You can just specify what you want, write it to the file, and then retrieve it when needed. While files created with the File class are not shared objects, the information stored in these files can be accessed by anyone who opens the application with the appropriate client-side read calls set. Thus, like an application with a database and middleware, you can turn FMS3 into a mini-database system for storing and retrieving information.

Working with the File class involves two basic functions: writing data and reading it. Being a server-side class, most of the heavy lifting is done by the server. The client side's job is to retrieve data from the server side. Up to this point, you've seen how to call a server-side function and pass data in the form of arguments from the client side. Now, you will have to learn how to retrieve data from the server side and use it there. First, though, you'll learn how to instantiate a file and write data to the file on the server side.

7.2.1. Creating a File Class Instance

To create a file instance requires a single parameter for a name. The name must be made up of simple text characters-UTF-8 encoded characters. Essentially, you specify a string with the name of the file you'll use to store your data. For example, the line:

```
var myBigFile = new File("myStuff.txt");
```

creates an instance named `myBigFile` and a text file named myStuff.txt. All of the methods and properties of the File class can now be used with `myBigFile`, and all of the data associated with `myBigFile` will be stored in and retrieved from myStuff.txt. You can now write to and read from the file.

7.2.2. Writing to a File

Once you create a File instance, you can write to the file specified in the constructor parameter. To write to a file, use the `File.open` method to open it to write, append, or read using text, binary, or UTF8 data. The general format is: fileInstance.open("text", "write");

Once the file is open, the format for writing to the file is:

```
fileInstance.write(data);
```

The data can be anything passed to the parameter, typically a string or numeric value. Before you write to the file, a good practice is to verify that it's open using the `File.isOpen` method. If the file has been opened successfully, it returns a Boolean `true`; otherwise, it returns a `false`. Once you have written to a file, the next step would be to close it. So, a typical sequence would be:

```
if(fileInstance.isOpen)
{
  fileInstance.write(data);
  fileInstance.close( );
}
```

If you use "`append`" as a parameter when opening the file instance instead of "`write`," the pointer goes to the

end of the file. But if you select "`write`," you will overwrite the entire file. That is, it will not clear the file and then write at the file's beginning. It just writes at the beginning of the file.

7.2.3. Reading a File

To read a server-side file, only a few changes are required from writing a file. Instead of using "`write`" you use "`read`" and one of the several read methods. You can select from the following:

```
fileInstance.read(n)
```

Reads (n) number of characters from the file where (n) is an integer.

```
fileInstance.readAll( )
```

Reads the entire file.

```
fileInstance.readByte( )
```

Reads the next byte from the file, and then returns a numeric value of the byte. This is the only read method that works if the file is opened in the binary mode.

```
fileInstance.readln( )
```

Reads the next line based on the line-break character. In Linux, the character is `\n`, and either `\r` or `\n` in a Windows server.

A typical read sequence would be:

```
fileInstance.open("text", "read");
if(fileInstance.isOpen)
{
  var myData=fileInstance.readAll( );
  fileInstance.close( );
  return myData;
}
```

The sequence is similar to the write sequence, but it has a critical new element you need to examine, the `return` statement. The contents of the file are passed to the `myData` variable, and that is returned to somewhere, but where? And how? It needs to go to the client-side script so that it can be displayed. To see how this is accomplished, look at a complete server-side script. This script will be a simple, minimalist one that will write data to a text file and then display its contents. All data input and output will originate on the client side. The following script uses the client name fileWriter instead of client, as used in server-side scripts in previous chapters. The two key functions are `fileWriter.WriteNow` and `fileWriter.ReadNow`. Both of these functions are called from the client side, as discussed in Chapter 6 in the section on "Section 6.3.2". However, this script uses a `return` statement to send information back to the client side. Follow these steps to begin the project:

1. In the server-side applications folder, create a folder named fileIt.

2. Open an ActionScript Communication file and save it as filtIt.as in the fileIt folder.

3. In the filtIt.asc file, enter the code in Example 7-1 and save the file.

Example 7-1. fileIt.asc

Code View:

```
application.onAppStart=function( )
{
  trace("Basic File script is up");
};

application.onConnect=function(fileWriter, name)
{
  fileWriter.id = name;
  application.acceptConnection(fileWriter);
  var sandFile = new File("sandlight.txt");

  fileWriter.WriteNow = function(cliMsg)
  {
        sandFile.open("text", "append");
        if (sandFile.isOpen)
        {
             sandFile.write(cliMsg);
             sandFile.close( );
        }
  };

  fileWriter.ReadNow = function( )
  {
        sandFile.open("text", "read");
        if (sandFile.isOpen)
        {
             contentNow = sandFile.readAll( );
             sandFile.close( );
             return contentNow;
        }
  };
};

application.onDisconnect=function(fileWriter)
{
  trace(fileWriter.id+" has left.");
};
```

When you've finished writing and saving the script, you will need a client-side script to enter data into a text file and to read from the file. This next section shows how ActionScript 3.0 sends data to, and retrieves it from, a text file.

7.2.4. Minimal Client-Side Write and Read File Data

All you really need on the client side for the basic operations involved in writing data to a file and getting it back is some way to call and send data to the server-side functions and capture returned data. From previous chapters, you know that the `NetConnection.call( )` function will send parameters to the server side. However, getting data back from the server is another matter. Fortunately, it's not difficult. Keeping in mind that the `NetConnection.call()` method has three parameters-server-side function name (a string), a Responder object to capture what has been returned from the server side, and optional parameters for sending data to the server side-all you need to do is to make a call with a parameter to send data to be written and another with a call to be read.

The `NetConnection.call()`for sending data has been discussed in detail in Chapter 6. Here, you'll use `NetConnection.call()` to retrieve server-side data. The basic format is:

```
NetConnection.call("readDataFunc",responder);
```

The second parameter is a reference to an object that can be used to capture any values returned by the called function. (A complete discussion of the Responder class can be found in Chapter 5.) Figure 7-1 shows the relationship between the client-side call and the returned value.

Figure 7-1. Client-side calls and server-side returns

Client Side Script

```
61      private function writeFile (e:MouseEvent)
62      {
63          nc.call ("WriteNow",null,textInput.text+"\n");
64          textInput.text="";
65      }
66
67      private function readFile (e:MouseEvent)
68      {
69          responder=new Responder(showFile);
70          nc.call ("ReadNow",responder);
71      }
72
73      private function showFile (fileBack:String)
74      {
75          textArea.appendText (fileBack);
76      }
```

Server Side Script

```
12      fileWriter.WriteNow = function(cliMsg)
13      {
14          sandFile.open("text", "append");
15          if (sandFile.isOpen)
16          {
17              sandFile.write(cliMsg);
18              sandFile.close( );
19          }
20      };
21
22      fileWriter.ReadNow = function( )
23      {
24          sandFile.open("text", "read");
25          if (sandFile.isOpen)
26          {
27              contentNow = sandFile.readAll( );
28              sandFile.close( );
29              return contentNow;
30          }
31      };
```

Figure 7-1 shows the following:

- The `writeFile` function (line 61) of the client-side script calls the `WriteNow` function on line 12 of the server-side script. The `writeFile` function has a null parameter for a responder, and sends the text contents of the `TextInput` object to the server.

- In the server-side script, the `WriteNow` function captures the contents of the text contents in a parameter named `cliMsg`. Those contents are then written into the file -- `sandFile.write(cliMsg)`.

- The `readFile` function on the client side (line 67) calls the `ReadNow` function on the server-side. First, though, the script defines a `Responder` instance (line 70) to send any returned data to the `showFile` function (line 73).

- When the `ReadNow` function fires, it opens the file and returns the contents of the file in a variable, `contentNow` (line 29). The returned contents are then stored in a string parameter, `fileBack`, in the `showFile` function on the client-side (line 73).

You're ready to create the client-side application and see what can be done with a minimal script for the File class. Follow these steps:

1. Open a new Flash file (ActionScript 3.0) and save it as BasicFile.fla.

2. Open the Library panel, and from the Components panel drag in Button, TextInput and TextArea components.

3. In the Document Class text box in the Property inspector, type **`BasicFile`** and resave the file.

4. Open a new ActionScript file and save it as BasicFile.as in the same folder with the BasicFile.fla file. (These files should not be in the same folder with your server-side applications!)

5. In the BasicFile.as file, enter the code in Example 7-2 and save the file.

Example 7-2. BasicFile.as

Code View:

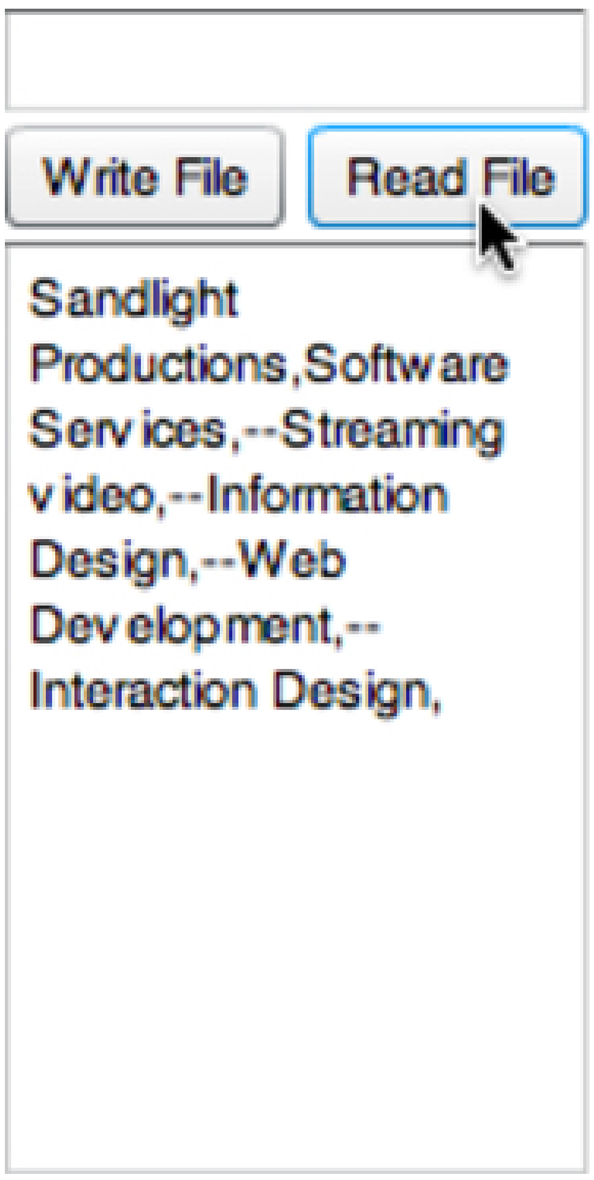
```
package
{
  import flash.net.NetConnection;
  import flash.net.Responder;
  import flash.events.NetStatusEvent;
  import flash.events.MouseEvent;
  import flash.display.Sprite;
  import fl.controls.Button;
  import fl.controls.TextInput;
  import fl.controls.TextArea;

  public class BasicFile extends Sprite
  {
        private var nc:NetConnection;
        private var textInput:TextInput;
        private var textArea:TextArea;
```

```
private var readBtn:Button;
private var writeBtn:Button;
private var rtmpNow:String;
private var good:Boolean;
private var responder:Responder;

public function BasicFile ()
{
     nc=new NetConnection();
     rtmpNow="rtmp://192.168.0.11/fileIt/storeFile";
     nc.connect (rtmpNow,"Writer");
     nc.addEventListener (NetStatusEvent.NET_STATUS,checkConnect);
     writeBtn=new Button();
     writeBtn.x=20, writeBtn.y=45;
     writeBtn.width=60;
     writeBtn.addEventListener (MouseEvent.CLICK,writeFile);
     addChild (writeBtn);

     readBtn=new Button();
     readBtn.x=85, readBtn.y=45;
     readBtn.width=60;
     readBtn.addEventListener (MouseEvent.CLICK,readFile);
     addChild (readBtn);

     textInput=new TextInput();
     textInput.x=20, textInput.y=20;
     addChild (textInput);

     textArea = new TextArea();
     textArea.width=100, textArea.height=200;
     textArea.x=20, textArea.y=70;
     addChild (textArea);
}

private function checkConnect (e:NetStatusEvent):void
{
     good=e.info.code == "NetConnection.Connect.Success";
     if (good)
     {
          writeBtn.label="Write File";
          readBtn.label="Read File";
     }
}

private function writeFile (e:MouseEvent):void
{
     nc.call ("WriteNow",null,textInput.text+"\n");
     textInput.text="";
}

private function readFile (e:MouseEvent):void
{
     responder=new Responder(showFile);
     nc.call ("ReadNow",responder);
}

private function showFile (fileBack:String):void
{
     textArea.appendText (fileBack);
```

```
            }
        }
    }
```

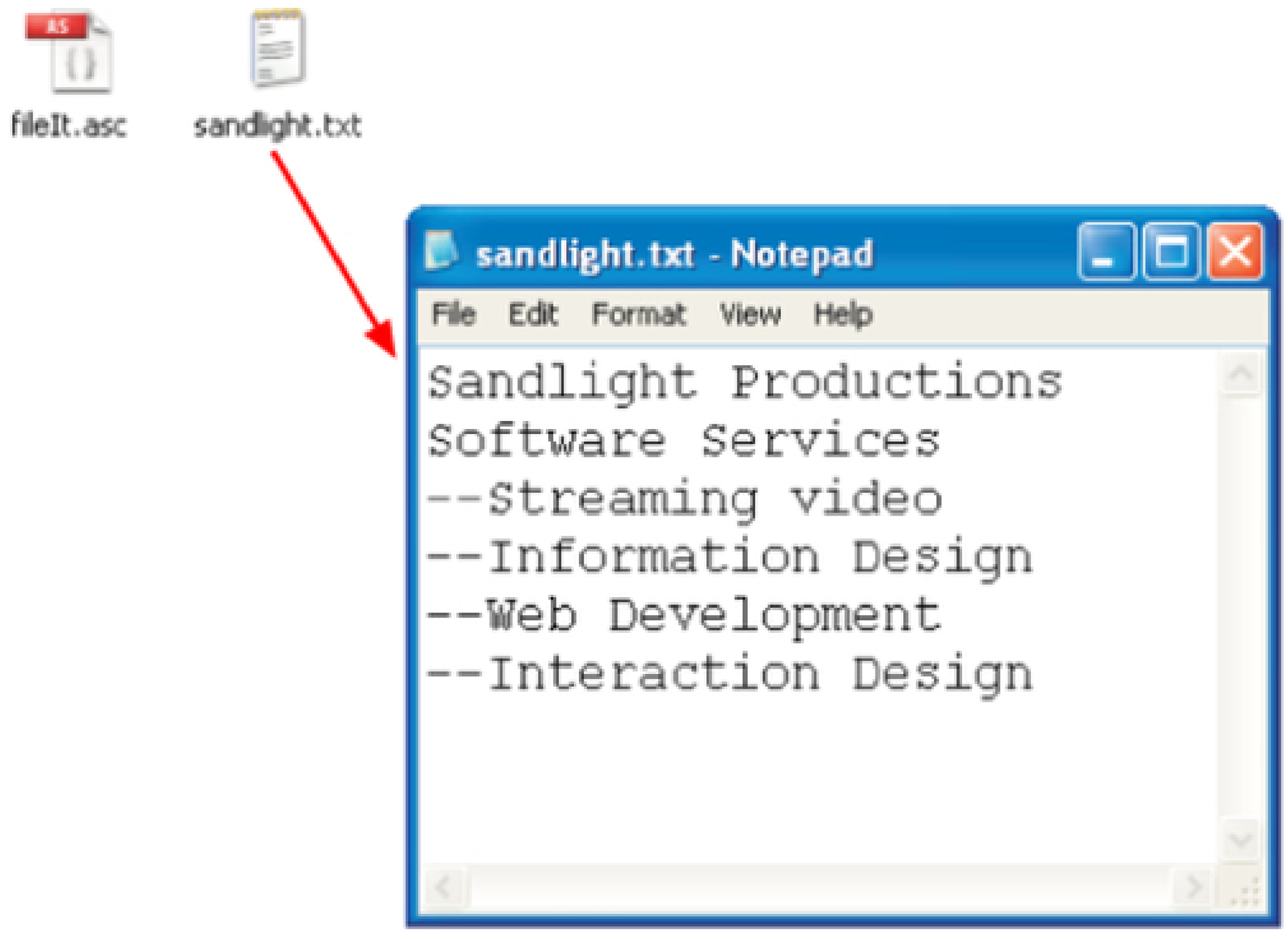
When you test the application, type in different words and press the Write File button. Once you have entered several words, press the Read file key; your entire file will appear as shown in Figure 7-2.

Figure 7-2. Displaying data from a server-side file



In looking at the output, you might assume that all of the line breaks were lost when the data was stored in the file. However, as Figure 7-3 shows, the file that was created has nicely delineated the entries.

Figure 7-3. Text file automatically created with File class



As you can see from Figure 7-3, the formatting is not because of the file formatting. Later in this chapter you'll learn some ways to format and use the server-side File class and ActionScript 3.0 to better format output. (See "Section 7.3" and "Section 7.4.") First, you'll look at a way to clear out all the text and put something else in the file.

7.2.5. Deleting Files

As you experiment with the minimal file application, you'll see the contents of the file keep growing. Is there a way to remove all of the file contents and start over?

One way to remove data from a file is to remove the file itself. Using `File.remove()`, the entire text file is deleted. But if you add more data, it reappears. The format is:

```
fileInstance.remove( );
```

To use it, all you need to do is place the method into a function and call it from the server side. Open the fileIt.asc file, and insert the following function after the `ReadNow` function:

```
fileWriter.RemoveNow = function( )
{
  sandFile.remove( );
};
```

Save the server-side file as fileItRemove.asc in a folder in the server-side applications folder named fileItRemove. Follow these steps to edit the client-side files:

1. Open the client-side BasicFile.fla file and change the Document class value in the Property inspector to `BasicRemove` and save the file as BasicRemove.fla.

2. Open the BasicFile.as file and save it as BasicRemove.as.

3. In the BasicRemove.as file, replace the script with that shown in Example 7-3. Save the file again.

Example 7-3. BasicFileRemove.as

Code View:

```
package
{
  import flash.net.NetConnection;
  import flash.net.Responder;
import flash.events.NetStatusEvent;
import flash.events.MouseEvent;
import flash.display.Sprite;
import fl.controls.Button;
import fl.controls.TextInput;
import fl.controls.TextArea;

public class BasicRemove extends Sprite
{
      private var nc:NetConnection;
      private var textInput:TextInput;
      private var textArea:TextArea;
      private var readBtn:Button;
      private var writeBtn:Button;
      private var removeBtn:Button;
      private var rtmpNow:String;
      private var good:Boolean;
      private var responder:Responder;

      public function BasicRemove ()
      {
            nc=new NetConnection();
            rtmpNow="rtmp://192.168.0.11/fileItRemove/storeFile";
            nc.connect (rtmpNow,"Writer");
            nc.addEventListener (NetStatusEvent.NET_STATUS,checkConnect);
            writeBtn=new Button();
            writeBtn.x=20, writeBtn.y=45;
            writeBtn.width=60;
            writeBtn.addEventListener (MouseEvent.CLICK,writeFile);
            addChild (writeBtn);

            readBtn=new Button();
            readBtn.x=85, readBtn.y=45;
            readBtn.width=60;
            readBtn.addEventListener (MouseEvent.CLICK,readFile);
            addChild (readBtn);

            removeBtn=new Button();
            removeBtn.x=145, removeBtn.y=45;
```

```
            removeBtn.width=90;
            removeBtn.addEventListener (MouseEvent.CLICK,removeFile);
            addChild (removeBtn);

            textInput=new TextInput();
            textInput.width=125;
            textInput.x=20, textInput.y=20;
            addChild (textInput);

            textArea = new TextArea();
            textArea.width=125, textArea.height=200;
            textArea.x=20, textArea.y=70;
            addChild (textArea);
      }

      private function checkConnect (e:NetStatusEvent):void
      {
            good=e.info.code == "NetConnection.Connect.Success";
            if (good)
            {
                  writeBtn.label="Write File";
                  readBtn.label="Read File";
                  removeBtn.label="Remove File";
            }
      }

      private function writeFile (e:MouseEvent) :void
      {
            nc.call ("WriteNow",null,textInput.text+"\n");
            textInput.text="";
      }

      private function readFile (e:MouseEvent):void
      {
            responder=new Responder(showFile);
            nc.call ("ReadNow",responder);
      }

      private function removeFile (e:MouseEvent)
      {
            nc.call ("RemoveNow",null);
      }

      private function showFile (fileBack:String):void
      {
            if (fileBack != null)
            {
                  textArea.appendText (fileBack);
            }
            else
            {
                  textArea.text="Empty File";
            }
      }
  }
}
```

Whenever you press the Remove File button, whatever text file is associated with the server-side file instance will be deleted. When you press the Read File button, you will see an "Empty File" notice in the TextArea component. However, as soon as the call to write data is made, the text file is re-instantiated and filled with the new text.

NOTE

Open the server side fileItRemove folder and then run the application. As soon as you write data, the text file will appear. Then, when you click the Remove File button, it will disappear. The command doesn't just remove the data in the file; it literally removes the file itself.

Figure 7-4 in the next section shows what the application looks like when running. It has no formatting except for the dashes placed in front of the categories.

7.3. Client-Side Formatting

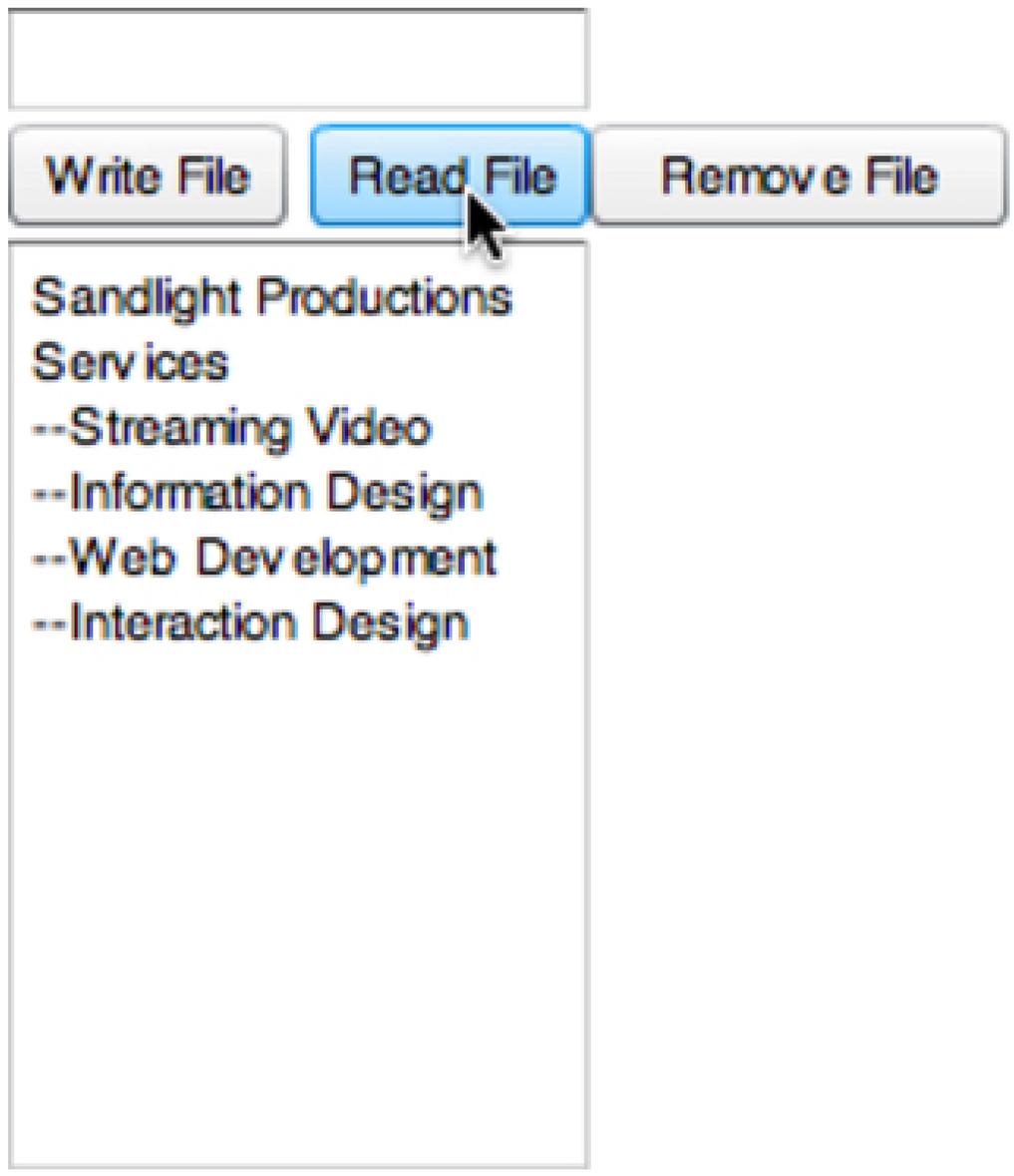
Data is formatted automatically on both the client- and server-side scripts. Up to now, the data being returned to the client side is an array. That's because when using `File.readAll()`, the elements are made up of each line of the file. When the data was entered from the client side, the `\n` escape character (new line) added the line needed to separate lines and have them go into an array. That's also where the commas came from. Commas always appear whenever you print an array as a whole. Knowing that you're dealing with an array, you can use array class methods for the formatting to better present data..

To see how this can be applied, replace the `showFile` method with the following:

```
private function showFile (fileBack:Array):void
{
  if (fileBack != null)
  {
        var backLen:uint=fileBack.length;
        for (var c:uint; c< backLen; c++)
        {
             textArea.appendText (fileBack[c] + "\n");
        }
  }
  else
  {
        textArea.text="File empty";
  }
```

By typing the result parameter (`fileBack`) as an array, you're making a much cleaner transition between the server-side object and the client-side implementation. So while strict typing is not possible on the server side, you can apply it on the client side. That provides a much clearer way to format the output. As you will see, each entry will appear on a separate line in the output, as shown in Figure 7-4.

Figure 7-4. Array-formatted output



How the data is entered into the file and what File method is used to read the data determines the options for displaying it on the client side. Planning ahead is essential to avoid a data mismatch.

7.4. Server-Side Formatting

Up to this point, `File.write( )` has been the only method employed for writing data and `File.readAll( )` for reading the contents of a file. However, you have far more options on the server-side. Table 7-1 shows all the writing and reading methods in the File class.

File Method	Action
write(p)	Writes one or more parameter values to an open file.
writeAll(arr)	Writes array parameter to an open file.
writeByte(n)	Writes byte n to the file.
write n(p)	Writes p to file with end-of-line character after last parameter. Because this is a server-side method, the end-of-line character depends on the type of server (Windows or Linux) and not on the client's platform.
read(n)	Reads n number of characters from file.
readAll( )	Reads the entire file as an array with end of line being the element delimiter.
readByte( )	Reads the next byte from file opened in binary mode.
readln( )	Reads the next line in the file as a string using line-separator characters determined by server platform (\r\n for Windows and \n for Linux).

To see how to format with both write and read methods, you'll alter the minimal File application used so far in this chapter. It will require making further changes to both the server-side and client-side scripts.

In using `File.write()`, FMS3 simply writes data to the file passed to it from the client side. Using `File.writeln()`, FMS3 adds a new line character, depending on the kind of server (Windows or Linux) you're using. This means that you don't have to add a `\n` escape character to the client-side script when the data is sent to the server-side script.

In reading the data back, instead of sending an array with a single statement as in the `File.readAll()` method, `File.readln()` reads the file a line at a time as a string. When you're searching for a term in a file, reading a single line at a time can be handy if you have a search word on each line, such as a customer number or name. Then you can extract the single line and send it to the client-side for viewing.

First, use File Save As to save the fileIt.asc file as fileEof.asc. Then change the write and read functions to the following and save it as fileIt.asc:

Code View:

```
fileWriter.WriteNow = function(cliMsg)
{
      sandFile.open("text", "append");
      if (sandFile.open)
      {
           sandFile.writeln(cliMsg);
           sandFile.close( );
      }
      trace(cliMsg);
```

```
};


fileWriter.ReadNow = function( )
{
      sandFile.open("text", "read");
      var contentNow = new Array();
      if (sandFile.isOpen)
      {
           var counter =0;
           while(!sandFile.eof())
           {
                readBuf=sandFile.readln();
                contentNow[counter]=readBuf;
                counter++;
           }
           return contentNow;
           sandFile.close( );
      }
};
```

```
  };

};
```

The major difference in these functions is employment of the `File.eof()` method. Using `File.readAll()` places the entire contents of the file into an array. In this case, the `File.eof()` function first iterates through the lines in the file and then puts the elements into the array one at a time until the end-of-file condition is met.

7.4.1. Server-Side Data Initiation

Suppose you want to find out information about who's using your application, and you want this to be initiated from the server using logon information. When users log on to your application, typically they enter a username that they make up. If you want to log a username with a user's IP address and put it into a file, you can. In this way, if you have a problem with a user, you can use the IP address to deny their connection to the application. What's more, all of this can be done on the server side. The following steps will get you started:

1. In the server side applications folder, create a folder named fileIP.

2. Open, a new ActionScript Communication file and save it as fileIP.asc in the fileIP folder.

3. In the fileIP.asc file, enter the code in Example 7-4 and save the file.

Example 7-4. fileIP.asc

```
application.onAppStart = function()
{
  trace("Trap IP is up");
};
application.onConnect = function(client, name)
{
  application.acceptConnection(client);
  client.id = name;
  var ip = client.ip;
  var ipFile = new File("ip.txt");
  ipFile.open("text", "append");
  if (ipFile.isOpen) {
        ipFile.writeln(client.id, ip);
        ipFile.close();
  }
  trace(client.id+"'s IP address is "+ip);
};
application.onDisconnect = function(client)
{
  trace(client.id+" has left.");
};
```

The script uses the `Application.onConnect( )` event handler to fire the script to open the file and record the current user's name and associated IP address. The script is simple primarily because `Client.ip` is one of the `Client` class's native properties. The client-side module is very simple as well. All you need is a Button and a TextInput component. Follow these steps to complete the application:

0. Open a new Flash file. In the Document Class text box in the Property inspector, type **`IPfile`** and save the file as IPfile.fla.

1. Open the Library panel and drag a Button and TextInput component into the Library. Save the file.

Example 7-5. IPfile.as

Code View:

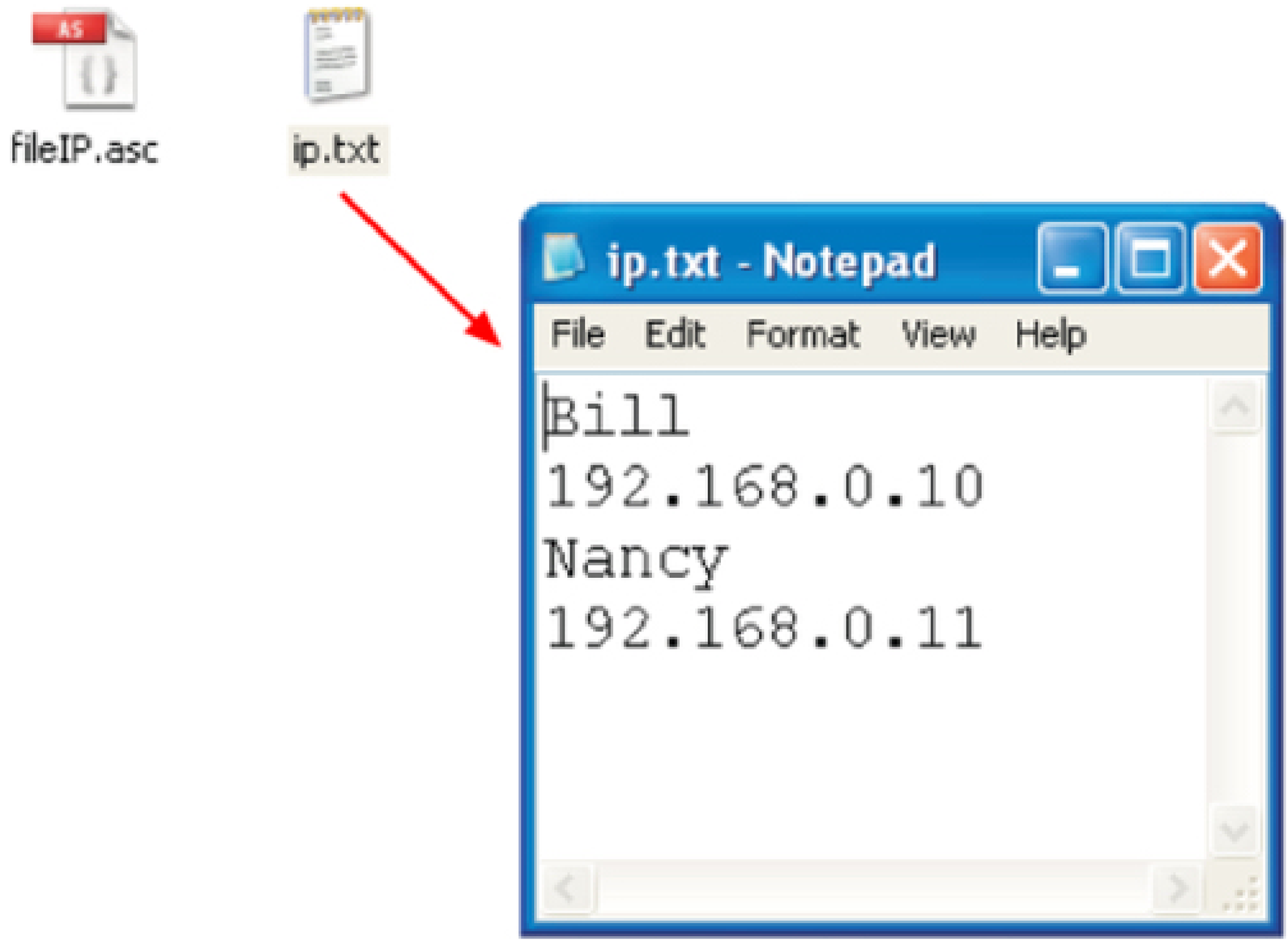
```
package
{
  import flash.net.NetConnection;
  import flash.events.NetStatusEvent;
  import flash.events.MouseEvent;
  import flash.display.Sprite;
  import fl.controls.Button;
  import fl.controls.TextInput;

  public class IPfile extends Sprite
  {
        private var nc:NetConnection;
        private var textInput:TextInput;
        private var logBtn:Button;
        private var rtmpNow:String;
        private var good:Boolean;

        public function IPfile ()
        {
             nc=new NetConnection();
             nc.addEventListener (NetStatusEvent.NET_STATUS,checkConnect);
             rtmpNow="rtmp://192.168.0.11/fileIP/getIP/";
             textInput=new TextInput();
             textInput.x=20, textInput.y=20;
             addChild (textInput);

             logBtn=new Button();
             logBtn.x=20, logBtn.y=45;
             logBtn.width=120;
             logBtn.label="Enter name and click";
             logBtn.addEventListener (MouseEvent.CLICK, logOn);
             addChild (logBtn);
        }

        private function logOn (e:MouseEvent)
        {
             nc.connect (rtmpNow,textInput.text);
        }

        private function checkConnect (e:NetStatusEvent):void
        {
             good=e.info.code == "NetConnection.Connect.Success";
             if (good)
             {
                   textInput.text="IP Stored";
             }
             else
             {
                   textInput.text="Failed Connection";
             }
        }
  }
}
```

The user enters his or her name and clicks the Logon button, labeled Enter Name And Click. This sends the name of the user to the server-side script as the second argument in the server-side line:

```
application.onConnect = function(client, name) {
```

Creating a `client.id` property with the name passed from the client side and a variable, `ip` with the value from the `client.ip` property places both into the ip.txt file using `File.writln()`., Figure 7-5 shows how the values of two different names from two different connections on a LAN are stored.

Figure 7-5. User names and IP addresses stored in file



Note that when using multiple arguments in the `File.writln()` function, the values are placed on separate lines. To retrieve the information, simply use one of the read methods on the client- or server-side scripts.

7.5. Beggar's Database

Now that you have seen different ways to use the server-side File class, it's time to use that knowledge to build a database. This database is a "Section 7.5" because of its limited capability, but it has the essential elements of a database. It stores data in discrete fields and provides a search function.

Here's the way it works:

- Each entry is stored on a separate line in the text file using the `File.writeln` method. This example uses three fields; name, email, and phone. You can substitute your own fields if you wish.

- The information is retrieved by searching on the first field-the name field. Using the `File.readln` method (function), it first finds the name passed from the client side.

- Because all the data is stored on separate lines, the next two lines in the text file will be the next two values associated with the name-the email address and phone number.

- To get the associated email and phone, two more `File.readln` statements bring them up to be stored in separate variables.

- The two variables with the email and phone information are then concatenated with a tilde (~) separator and returned to the client side.

Before looking at the client-side script, you'll first get the server-side script in place; follow these steps:

1. Create a folder in the server-side applications folder named beggarDB.

2. Open a new ActionScript Communication file and save it as beggarDB.asc in the beggarDB folder.

3. In the beggarDB.asc file, enter the script in Example 7-6 and save the file.

Example 7-6. beggarDB.asc

Code View:

```
application.onAppStart = function()
{
    trace("Beggar Database is up");
};

application.onConnect = function(clientDB)
{
    application.acceptConnection(clientDB);
    cusMail = new File("customers.txt");

    Client.prototype.storeData = function(cusName, cusEmail, cusPhone)
    {
          cusMail.open("text", "append");
          if (cusMail.isOpen)
          {
               cusMail.writeln(cusName, cusEmail, cusPhone);
               cusMail.close();
          }
          trace(cusName+"'s Email="+cusEmail+ " and phone#:"+cusPhone);
    };

    Client.prototype.findData = function(search)
    {
          var dbData;
          var foundEmail;
          var foundPhone;
          var foundData;
          cusMail.open("text", "read");
          if (cusMail.isOpen)
          {
               while (!cusMail.eof())
               {
                    dbData = cusMail.readln();
                    trace(cusMail.position);
                    if (dbData == search) {
                         foundEmail = cusMail.readln();
                         foundPhone = cusMail.readln();
                         foundData=foundEmail +"~"+foundPhone;
                         cusMail.position = cusMail.length;
                    }
               }
               cusMail.close();
          }
          return foundData;
    };
    clientDB.id = "Current user";
};

application.onDisconnect = function(clientDB)
{
    trace(clientDB.id+" has left.");
};
```

The use of the `prototype` property in the constructor allows the method to be executed once, and not every time a client connects. Using the `prototype` property means that the other clients in the `application.clients` array won't have to define the method with every new client.

Once the data entry and search retrieval are completed on the server side, you need to set up a client-side application to send data and bring the retrieved data to the server side. You should be familiar with the concepts from previous discussions and applications in this chapter. In recalling data from the server side, the client-side script uses substring methods to separate the returned email address from the phone number. The main difference is primarily in the number of label components used. The following steps show how to create the client-side portion to finish up the application:

0. Open a new Flash file and type BeggarDB in the Document Class text box. Save the file as BeggarDB.fla.

1. Open the Library panel and drag Button, TextInput, and Label components into the bottom panel of the Library panel. Save the file.

2. Add a layer. Using Static text, enter the label **`Beggar's Database`** at the top of the page. With the Line tool, draw a 1-point gray line horizontally across the page at y=163. This line separates the data entry from the data search sections of the page. Next, lock the top layer, and on the bottom layer draw a rectangle with no stroke value and a gradation fill. Rotate the rectangle so that the lighter side of the gradation is on the bottom. Now use the Align panel to size and fit the rectangle to the size of the Stage; lock the layer.

3. Open a new ActionScript file and save it as BeggarDB.as.

4. In the beggarDB.asc file, enter the script in the Example 7-7 and save the file.

Example 7-7. beggarDB.asc

Code View:

```
package
{
    import flash.net.NetConnection;
    import flash.net.Responder;
    import flash.events.NetStatusEvent;
    import flash.events.MouseEvent;
    import flash.display.Sprite;
    import flash.text.TextFieldAutoSize;
    import fl.controls.Button;
    import fl.controls.Label;
    import fl.controls.TextInput;

    public class BeggarDB extends Sprite
    {
          private var nc:NetConnection;
          private var responder:Responder;
          private var nameInput:TextInput;
          private var emailInput:TextInput;
          private var phoneInput:TextInput;
          private var searchInput:TextInput;
```

```
        private var nameL:Label;
        private var emailL:Label;
        private var phoneL:Label;
        private var searchL:Label;
        private var foundE:Label;
        private var foundP:Label;
        private var foundD:String;
        private var dbMail:String;
        private var dbPhone:String;
        private var s:uint;

        private var storeBtn:Button;
        private var findBtn:Button;

        private var rtmpNow:String;
        private var good:Boolean;

        public function BeggarDB ()
        {
             nc=new NetConnection();
             nc.addEventListener (NetStatusEvent.NET_STATUS,checkConnect);
             rtmpNow="rtmp://192.168.0.11/beggarDB/beggar";
             nc.connect (rtmpNow);

             nameL=new Label();
             nameL.text="Name:";
             nameL.x=18,nameL.y=54;
             addChild (nameL);

             emailL=new Label();
             emailL.text="Email:";
             emailL.x=18,emailL.y=90;
             addChild (emailL);

             phoneL=new Label();
             phoneL.text="Phone:";
             phoneL.x=18,phoneL.y=127;
             addChild (phoneL);

             searchL=new Label();
             searchL.text="Search\nName";
             searchL.x=18,searchL.y=180;
             searchL.height = 36;
             addChild (searchL);

             nameInput=new TextInput();
             nameInput.x=72, nameInput.y=54;
             nameInput.width=144, nameInput.height=18;
             addChild (nameInput);

             emailInput=new TextInput();
             emailInput.x=72, emailInput.y=90;
             emailInput.width=144, emailInput.height=18;
             addChild (emailInput);

             phoneInput=new TextInput();
             phoneInput.x=72, phoneInput.y=127;
             phoneInput.width=144, phoneInput.height=18;
             addChild (phoneInput);
```

```
            searchInput=new TextInput();
            searchInput.x=72, searchInput.y=180;
            searchInput.width=144, searchInput.height=18;
            addChild (searchInput);

            storeBtn=new Button();
            storeBtn.x=(216+18), storeBtn.y=90;
            storeBtn.width=70;
            storeBtn.addEventListener (MouseEvent.CLICK, storeData);
            addChild (storeBtn);

            findBtn=new Button();
            findBtn.x=(216+18), findBtn.y=180;
            findBtn.width=80;
            findBtn.addEventListener (MouseEvent.CLICK, searchData);
            addChild (findBtn);

            foundE=new Label();
            foundE.autoSize=TextFieldAutoSize.LEFT;
            foundE.x=72, foundE.y=216;
            foundE.text="Email: ";
            addChild (foundE);

            foundP=new Label();
            foundP.autoSize=TextFieldAutoSize.LEFT;
            foundP.x=72, foundP.y=234;
            foundP.text="Phone: ";
            addChild (foundP);
        }

        private function storeData (e:MouseEvent)
        {
            nc.call ("storeData",null,nameInput.text,emailInput.text,
phoneInput.text);
            nameInput.text="";
            emailInput.text="";
            phoneInput.text="";
        }

        private function searchData (e:MouseEvent)
        {
            responder=new Responder(returnData);
            nc.call ("findData",responder,searchInput.text);
            foundE.text="Email: ";
            foundP.text="Phone: ";

        }

        private function returnData (dbFound:String)
        {
            if (dbFound !=null)
            {
                s=dbFound.indexOf("~");
                dbMail=dbFound.substring(0,s);
                dbPhone=dbFound.substring(s+1,dbFound.length);
                foundE.text+=dbMail;
                foundP.text+=dbPhone;
            }
            else
```

```
                {
                        searchInput.text="Name not found.";
                }
        }

        private function checkConnect (e:NetStatusEvent):void
        {
                good=e.info.code == "NetConnection.Connect.Success";
                if (good)
                {
                        storeBtn.label="Store Data";
                        findBtn.label="Search Data";
                }
                else
                {
                        storeBtn.label="Failed";
                        findBtn.label="Failed";
                }
        }
    }
}
```

When you test the application, wait until the labels on the buttons appear. (They're a beggar's connection light.) Then enter a name, email address, and phone number, and press the Store Data button. As soon as you click the Store Data button, all of the windows for data entry clear to allow for the next entry. When you click the Search Data button, the resulting email and phone appear beneath the Search window, as shown in Figure 7-6. You can retrieve the information you want by entering the name in the Search Name text field. If the name you enter is not found, a "Name not found" notice appears in the Search window.

Figure 7-6. Objects and instance names for FileDB application

When you test your application with this data search routine, open your FMS3 console and watch the Live Log with the View Applications tab selected. You will see the file seek position (the value of the position in the FLV file) and take fewer iterations for items near the top of the file. This means that the file doesn't have to stay open as long or keep working after finding a match. In essence, it's more efficient for data storage and retrieval. You can use the File class for just about any kind of data storing. This chapter has examined many of the methods and properties of the File class; the FMS3 documentation covers many more for any file work your application may have. Subsequent chapters show more tasks that the File class can perform.

Chapter 8. Server-Side Shared Objects

8.1. What Are Server-Side Shared Objects?

Flash Media Server 3 shared objects can be divided into three groups:

Locality

- local
- remote

Source

- client-side
- server-side

Persistence

- persistent
- nonpersistent

Local shared objects are employed where data is used on the client's computer and that data is not shared with others. Because they do not require FMS3, local shared objects will not be examined here. Remote shared objects share data through FMS3, and server-side remote shared objects assign values to shared objects on the server and communicate with client-side shared objects. Server-side shared objects can also share data between multiple Flash Media Servers. Persistent remote shared objects are stored on disks in FSO files; nonpersistent shared objects exist only as long as long as the application is in use.

FMS3 video and audio streams are sent out and "subscribed to" by clients though a common FMS3 application. Each "subscription" generates a new stream. Similarly, shared objects represent a way of sharing data. A remote shared object is an object that is sent to other current application instances through connected clients

using Flash Media Server. Shared objects can be completely client-side in that they are generated, have values assigned to them, and are sent to the communication server ready for sharing. Remote client-side shared objects do not require a server-side script. (See Chapter 4, "Nonpersistent Client-Side Remote Shared Objects," for a full discussion of client-side shared objects.)

This chapter deals with both persistent and nonpersistent shared objects generated by shared objects on the server. The first half of the chapter covers nonpersistent server-side shared objects, and the second half, persistent shared objects.

Chapter 8. Server-Side Shared Objects

8.1. What Are Server-Side Shared Objects?

Flash Media Server 3 shared objects can be divided into three groups:

Locality

- local
- remote

Source

- client-side
- server-side

Persistence

- persistent
- nonpersistent

Local shared objects are employed where data is used on the client's computer and that data is not shared with others. Because they do not require FMS3, local shared objects will not be examined here. Remote shared objects share data through FMS3, and server-side remote shared objects assign values to shared objects on the server and communicate with client-side shared objects. Server-side shared objects can also share data between multiple Flash Media Servers. Persistent remote shared objects are stored on disks in FSO files; nonpersistent shared objects exist only as long as long as the application is in use.

FMS3 video and audio streams are sent out and "subscribed to" by clients though a common FMS3 application. Each "subscription" generates a new stream. Similarly, shared objects represent a way of sharing data. A remote shared object is an object that is sent to other current application instances through connected clients

using Flash Media Server. Shared objects can be completely client-side in that they are generated, have values assigned to them, and are sent to the communication server ready for sharing. Remote client-side shared objects do not require a server-side script. (See Chapter 4, "Nonpersistent Client-Side Remote Shared Objects," for a full discussion of client-side shared objects.)

This chapter deals with both persistent and nonpersistent shared objects generated by shared objects on the server. The first half of the chapter covers nonpersistent server-side shared objects, and the second half, persistent shared objects.

8.2. Working with Server-Side Shared Objects

The data property is the main property used with client-side shared objects. However, server-side shared objects have no data properties, and so shared object values have to be generated in a different manner. On the server-side, this is done with the `get` function. The following shows the difference between the client- and server-side methods of constructing shared objects:

Server-side

```
var mySerSide_so = SharedObject.get("name", Boolean);
```

Client-side

```
var myCS_so:SharedObject = SharedObject.getRemote("name", uri, Boolean);
```

As shown in Chapter 4, "Nonpersistent Client-side Remote Shared Objects," whenever a shared object is changed, the `SyncEvent.SYNC` event coordinates any data changes for all clients connected to a shared object. The same can be done with shared objects originating with server-side scripts. Using `SyncEvent.SYNC` finds shared object data as well. The following shows how the client-side SharedObject class can be used to display shared object data that originates in server-side script.

8.2.1. Making a Minimalist Room List: Server-Side Class

Using server-side script, shared objects can be used to keep track of a list of users currently connected to your computer. You can set up your shared object as soon as your application starts. In that way, all clients are connected to the same shared object. So, within the `onAppStart` function, you should have a line that instantiates the server-side `SharedObject` class. The following line constructs an application shared-object named `users_so`:

```
application.users_so = SharedObject.get("users_so", false);
```

You can reference the same shared object on the client-side as well. The shared object is essentially a server-side object whether it is created using server- or client-side coding.

An important step that you won't see in using shared objects with server-side coding is a shared object connection (`SharedObject.connect()`). The method is not part of the sever-side code, and the shared object automatically connects to the server as part of the client connecting using `NetConnection` on the client-side.

As each client enters a name and connects to the server, that name is passed to a server-side function. In turn, this same name is passed to the shared object. The sequence can be seen as:

- The parameter "username" is passed from the client-side to the server-side and placed in the `onConnect`

function.

- The parameter is passed to the client's name property.
- The name is passed to the shared object using the `setProperty` method.

When the client leaves the application, the `onDisconnect` function places a `null` value in the shared object, effectively to remove the client from the user list. Note that the shared object is instantiated in the `application.onAppstart` function at the beginning of the script. So the shared object is part of the application, and not just for the currently connected client. Follow these steps to get started:

1. Open your server-side applications folder and create a folder named userlist.
2. Open a new ActionScript Communication file and save it as userlist.asc in the userlist folder.
3. In the userlist.asc file, enter the code in Example 8-1 and save the file again.

Example 8-1. userlist.asc

```
application.onAppStart = function()
{
    trace(this.name+" is reloaded");
    this.users_so = SharedObject.get("users_so", false);
};
application.onConnect = function(currentClient, username)
{
    currentClient.name = username;
    this.acceptConnection(currentClient);
    this.users_so.setProperty(currentClient.name, username);
   trace(this.users_so.getProperty(currentClient.name));
};
application.onDisconnect = function(currentClient)
{
    trace("disconnect: "+currentClient.name);
    this.users_so.setProperty(currentClient.name, null);
};
```

The process of first setting the user's name to a property of the Client object and then setting it to a SharedObject class appears redundant. However, the name property of the Client is not the same entity as the name of the shared object. For example, someone named Nancy can have a cat also named Nancy. While the name is the same, the person Nancy and the cat Nancy are not. This applies equally to Client names and `SharedObject` names.

At this point, the user's name is now in the application's shared object named `users_so`. The next step is to use that shared object in an application. One very useful application is to place all names connected to the same application and same "room" in a List component so that you can see who's connected and who's not. Actually, the first step in the process began when the user entered his or her name in an input text box and clicked a

button to make the connection. Thus, the line:

```
nc.connect("appName/roomName", "userName");
```

connects to the application and room in the first parameter, and to the user's name in the second parameter. The user's name is the one passed on to the server-side.

8.2.2. Making a Room List: Client-Side Retrieval

While the username on the server-side shared object originates on the client side, the shared object must be made available to the client side for its contents to appear on the server side. A simple way to accomplish this is to reference the same shared object name: `users_so`, that was built using server-side script. You're actually not accessing two different places for shared objects; they're all on the server side, even if they are defined on the client-side.

It sounds less strange when the `SharedObject.getRemote()` method is used to get the shared object. That's because the shared object was actually created by server-side code, even though both server- and client-side code can create shared objects residing on the server-side. That's why on the client side the shared objects have to use the `SharedObject.connect()` method. What you need to do first is create a shared object with the code:

```
var users_so = SharedObject.getRemote("users_so", nc.uri, false);
```

Once the shared object is connected to the server using the `SharedObject.connect()` method, you need a way to transfer the data from the shared object to the `List` component via a `DataProvider` object. Because shared objects are stored in array-like elements, and each element contains a user's name, you can create a loop to iterate through the object shown in the `listPlace` function:

```
private function listPlace (e:SyncEvent)
{
    list.dataProvider=connectName;
    list.removeAll ();
    for (var uName:String in users_so.data)
    {
          if (users_so.data[uName] != null)
          {
               connectName.addItem ({label:users_so.data[uName]});
          }
    }
}
```

The names originally set using the server-side script are simply taken from the `SharedObject` using the data property. To see how it all works together, look at Example 8-2 with the entire client-side script.

Example 8-2. NameList.as

Code View:

```
package
{
    import flash.net.NetConnection;
    import flash.net.SharedObject;
```

```
import flash.events.NetStatusEvent;
import flash.events.MouseEvent;
import flash.events.SyncEvent;
import flash.display.Sprite;
import fl.controls.TextInput;
import fl.controls.Button;
import fl.controls.List;
import fl.data.DataProvider;

public class NameList extends Sprite
{
      private var nc:NetConnection;
      private var rtmpNow:String;
      private var good:Boolean;
      private var textInput:TextInput;
      private var userName:String;
      private var button:Button;
      private var list:List;
      private var users_so:SharedObject;
      private var connectName:DataProvider;

      public function NameList ()
      {
           nc=new NetConnection();
           rtmpNow="rtmp://192.168.0.11/userlist/roomlist";
           nc.addEventListener (NetStatusEvent.NET_STATUS,checkConnect);

           textInput=new TextInput();
           textInput.x=20, textInput.y=20;
           addChild (textInput);

           button=new Button();
           button.x=20, button.y=45;
           addChild (button);
           button.addEventListener (MouseEvent.CLICK,addName);
           button.label="Connect";

           list=new List();
           list.x=20, list.y=70;
           addChild (list);
           connectName=new DataProvider();

      }
      private function addName (e:MouseEvent)
      {
           nc.connect (rtmpNow,textInput.text);
      }

      private function listPlace (e:SyncEvent)
      {
           list.dataProvider=connectName;
           list.removeAll ();
           for (var uName:String in users_so.data)
           {
                if (users_so.data[uName] != null)
                {
                     connectName.addItem ({label:users_so.data[uName]});
                }
           }
```

```
            }

            private function checkConnect (e:NetStatusEvent)
            {
                 good=e.info.code == "NetConnection.Connect.Success";
                 if (good)
                 {
                       button.label="Connected";
                       users_so=SharedObject.getRemote("users_so",nc.uri,false);
                       users_so.addEventListener (SyncEvent.SYNC,listPlace);
                       users_so.connect (nc);
                 }
            }
      }
}
```

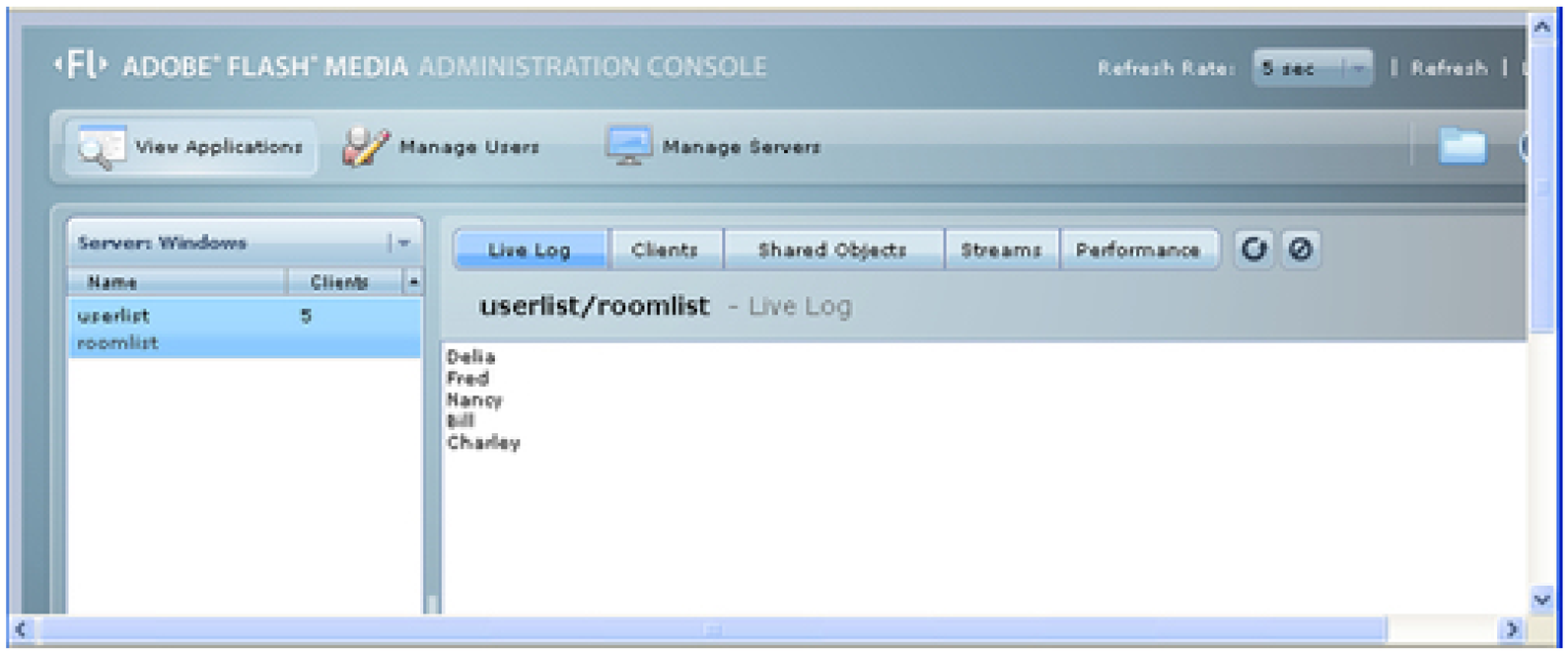
The client application is quite simple. Figure 8-1 shows the application when multiple users have connected to it.

Figure 8-1. Names of users connected to application

Each time a new user connects to the application, all names in the List component are removed; then the script loops through the shared object array and puts them all in. In this way, each user keeps getting an updated list of participants.

At the same time that all names appear in each of the clients' window, they also appear on the Administrative console through the `trace` statements in the server-side code. Figure 8-2 shows what happens when the names are added to the shared object array.

Figure 8-2. Names of connected users appear in the Administrative console



As you can see, the same names that appear in the server-side console window appear in the client-side List component.

8.3. Removing Users

One problem you or your clients may encounter is the need to remove unwanted users who logged onto your application, or kick off users in a game situation or some other condition. You can create an application to remove such users with `Application.disconnect()`. The format is:

```
application.disconnect(clientObj);
```

The trick is to get the `clientObj` target. To do that, you can use the `Application.clients` property. The `clients` property acts like an array but really isn't one: you cannot use the for-in loop. Otherwise, it works just like an array. The line:

```
application.disconnect(application.clients[n]);
```

where `n` is an integer value, removes the user in that slot. For example, imagine an application with three users, Joann, Dotty, and Fred. Joann is `application.clients[0]`, Dotty is `application.clients[1]`, and Fred is `application.clients[2]`. The line:

```
application.disconnect(application.clients[2]);
```

removes Fred. The trick is to work out a method for who is who so that you can remove the right user.

One way is to include a client name or ID. This can be done by passing the user's name to a client property. That property becomes part of all of the clients in Figure 8-2 The connected users appear in the list component when they connect `Application.clients` properties. For example, the following code segment adds a name property to each client on first connecting:

```
application.onConnect = function(currentClient, name)
{
    currentClient.name = name;
```

Suppose the name Fred is passed to the name property as the third name passed. The following line would have a value of "Fred":

```
application.clients[2].name
```

So, by looking for a match between the name you want to remove and the `Application.clients` properties, you can find the right user to remove using the following code:

```
for (var alive = 0; alive<application.clients.length; alive++)
{
    if (application.clients[alive].name == zapped)
    {
          application.disconnect(application.clients[alive]);
    }
```

```
}
```

To see how this works, you'll create a simple application. This particular application allows anyone to remove anyone else, but normally only an administrative module would allow such removal.

8.3.1. Client Removal Made Easy

This application has most of the features of the room application, adding only a new button and input component. To get started, you'll set up the server-side script. You will notice that it's the same as the Example 8-2 of a room list, except for the new section that removes the user.

1. Open a new ActionScript Communication file and save it as zap.asc in a folder named zap where you keep your server-side applications.

2. In the zap.asc file, enter the code in Example 8-3 and save the file again.

Example 8-3. zap.asc

Code View:

```
application.onAppStart = function()
{
    trace("Removal is reloaded");
    this.users_so = SharedObject.get("users_so", false);
};
application.onConnect = function(currentClient, name)
{
    currentClient.name = name;
    nowName = currentClient.name;
    this.acceptConnection(currentClient);
    this.users_so.setProperty(currentClient.name, nowName);
   //-----------------------------------------------------------------
    //Remove user
    currentClient.bust = function(zapped)
    {
          for (var alive = 0; alive<application.clients.length; alive++)
          {
               if (application.clients[alive].name == zapped)
               {
                    application.disconnect(application.clients[alive]);
               }
          }
    };
   //-----------------------------------------------------------------
};
application.onDisconnect = function(currentClient)
{
    trace("disconnect: "+currentClient.name);
    this.users_so.setProperty(currentClient.name, null);
};
```

As noted, this server-side script is very similar to the previous Example 8-2 in this chapter. To see the differences, look at the commented sections with dashed lines that focus on the added code used to remove a user.

8.3.2. Client-Side Removal Module

On the client side, set up a call to the `Client.bust` function and the appropriate shared objects so that all can see which users are still connected and which have been deleted. To create this application, you can use both the NameList.as and NameList.fla files to save a lot of time. However, as you will see, the two applications look very different.

Follow these steps to complete this removal application:

0. Load the NameList.fla and save it as it as Remove.fla. (The Library panel has all of the components you'll need since this new application uses the same ones.)

1. In the Document class text box of the Propertiy inspector, type **Remove**.

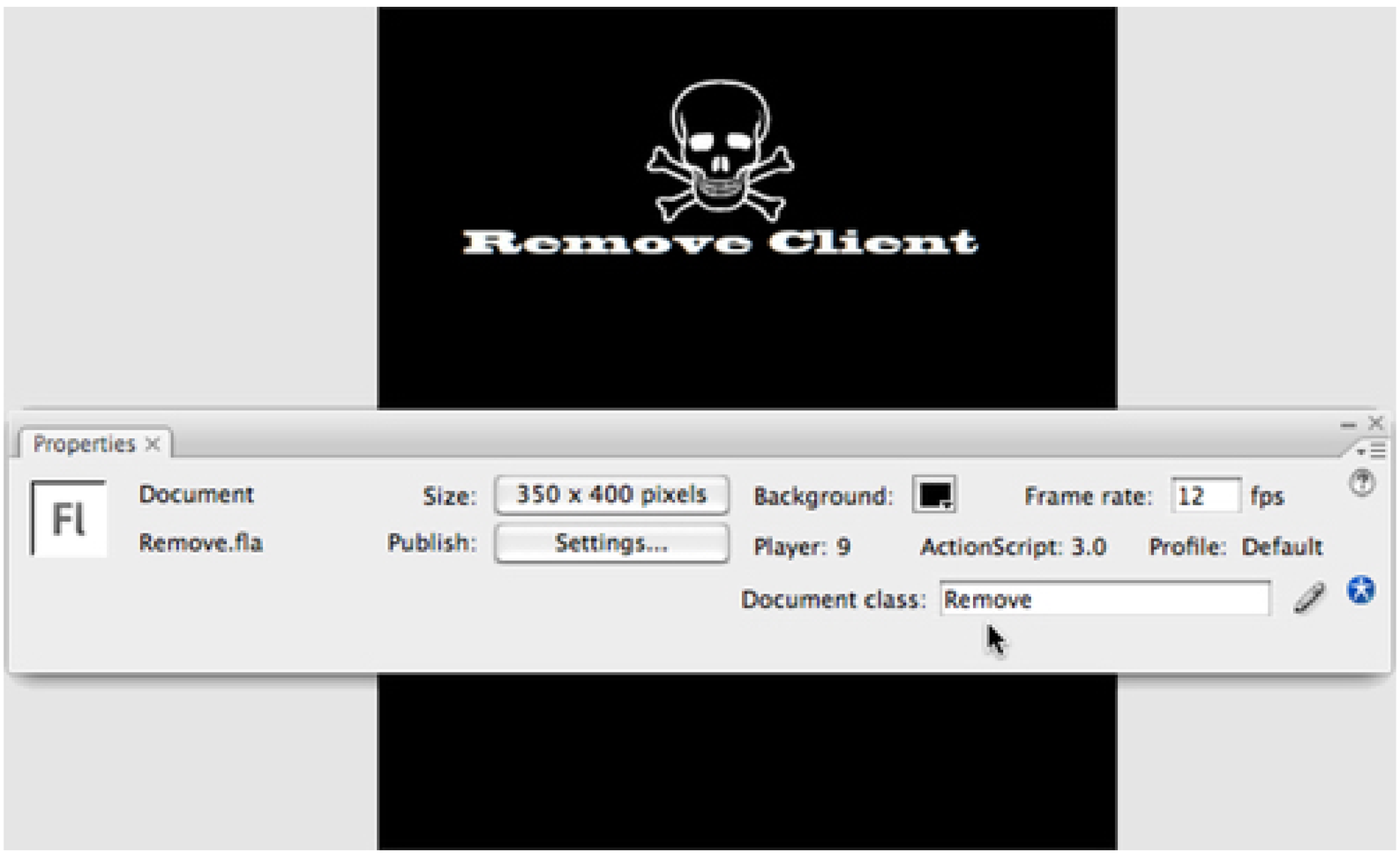
2. Change the Stage size to 350 x 400, add a black background, and resave the file.

3. Optionally, add a white skull and crossbones (70 x 70 dimensions; x=127, y=35) and a white Static text label, "Remove Client" (in this example, Blackoak Standard 14-point font, positioned at x=38, y=105). Save the file. Figure 8-3 shows the prepared Stage.

4. Open the NameList.as file, and save it as Remove.as in the same folder where you placed the Remove.fla file.

5. Leave the ActionScript in the file, and add the code in Example 8-4. (Be sure to change the class name to `Remove` and the constructor function to `Remove()`.)

Figure 8-3. Stage for application



Example 8-4. Remove.as

Code View:

```
package
{
    import flash.net.NetConnection;
    import flash.net.SharedObject;
    import flash.events.NetStatusEvent;
    import flash.events.MouseEvent;
    import flash.events.SyncEvent;
    import flash.display.Sprite;
    import fl.controls.TextInput;
    import fl.controls.Button;
    import fl.controls.List;
    import fl.data.DataProvider;

    public class Remove extends Sprite
    {
          private var nc:NetConnection;
          private var rtmpNow:String;
          private var good:Boolean;
          private var textInput:TextInput;
          private var userName:String;
          private var button:Button;
          private var killer:Button;
          private var list:List;
          private var users_so:SharedObject;
          private var connectName:DataProvider;

          public function Remove ()
          {
               nc=new NetConnection();
               rtmpNow="rtmp://192.168.0.11/zap/killer";
               nc.addEventListener (NetStatusEvent.NET_STATUS,checkConnect);
```

```
            textInput=new TextInput();
            textInput.x=108, textInput.y=128;
            addChild (textInput);

            button=new Button();
            button.x=108, button.y=153;
            addChild (button);
            button.addEventListener (MouseEvent.CLICK,addName);
            button.label="Connect";

            killer=new Button();
            killer.x=108, killer.y=280;
            killer.addEventListener (MouseEvent.CLICK,zap);
            addChild (killer);
            killer.label="Remove Selected";

            list=new List();
            list.x=108, list.y=178;
            addChild (list);
            connectName=new DataProvider();

        }
        private function addName (e:MouseEvent)
        {
            nc.connect (rtmpNow,textInput.text);
        }

        private function listPlace (e:SyncEvent)
        {
            list.dataProvider=connectName;
            list.removeAll ();
            for (var uName:String in users_so.data)
            {
                  if (users_so.data[uName] != null)
                  {
                        connectName.addItem ({label:users_so.data[uName]});
                  }
            }
        }

        private function zap (e:MouseEvent)
        {
            nc.call("bust",null,list.selectedItem.label);
        }

        private function checkConnect (e:NetStatusEvent)
        {
            good=e.info.code == "NetConnection.Connect.Success";
            if (good)
            {
                  button.label="Connected";
                  users_so=SharedObject.getRemote("users_so",nc.uri,false);
                  users_so.addEventListener (SyncEvent.SYNC,listPlace);
                  users_so.connect (nc);
            }
        }
    }
}
```

In looking at the application, the key new element is the call to the server-side function, `bust` in the private function `zap()`. The user to be deleted is selected by selecting his or her name in the List component and clicking the Remove Selected button. Clicking the Remove Selected button fires the `zap()` function that calls the server-side function, `bust`. Figure 8-4 and Figure 8-5 show what a name selected and removed.

Figure 8-4. Selecting client for removal

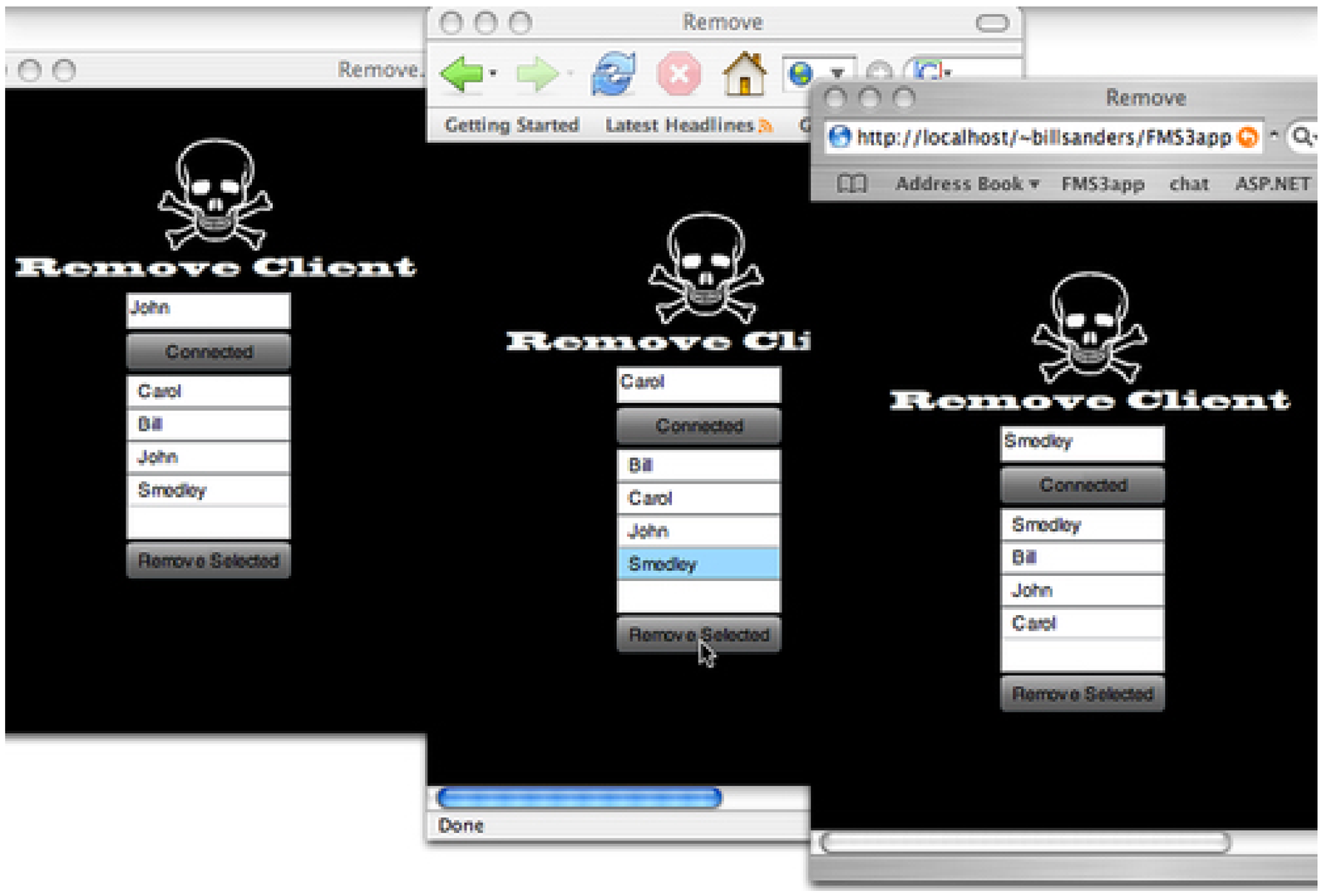


Figure 8-4 shows four clients connected, according to the names in the List component. (Three are shown and one is on a different computer.) The user named Carol selected the user named Smedley, and is clicking the Remove Selected button. Figure 8-5 shows what happens when the button is pressed.

Figure 8-5. Removed user is disconnected

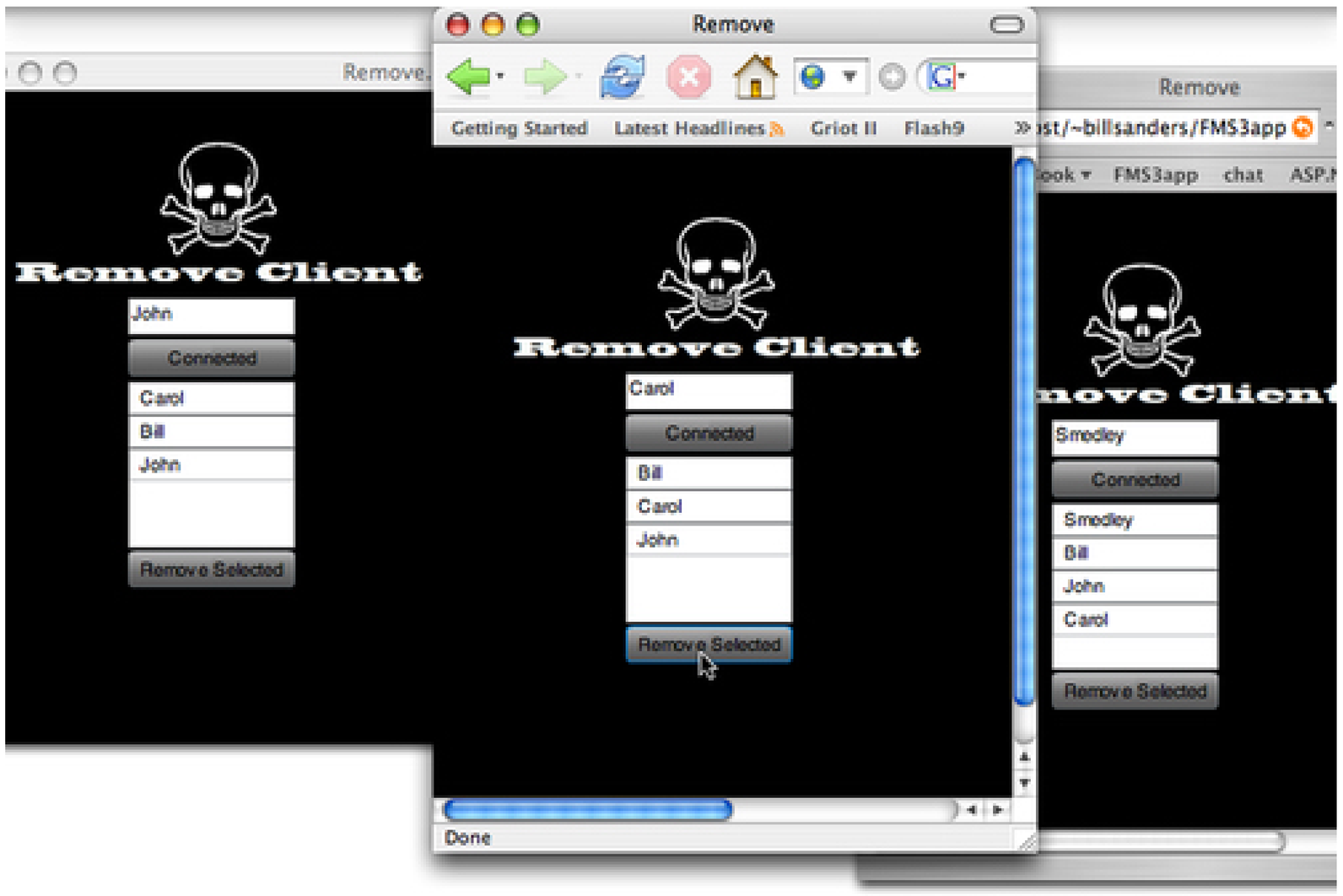


Figure 8-5 shows that John and Carol display three remaining users (as does Bill, who is on a different system). However, Smedley still shows the original four. That's because while John, Carol and Bill are still connected, Smedley is not. Because he is not connected, Smedley never gets the message from the server-side script to update (sync) all of the names. Figure 8-6 shows what is occurring on the Administration Console and the available information.

Figure 8-6. FMS console showing removed client with three remaining clients

Figure 8-6 shows the trace statement that appears whenever a user leaves, with the user name and the fact

that the user's been disconnected. It also shows that three clients are still connected.

8.4. Persistent Server-Side Shared Object

Remote persistent server-side shared objects are those shared objects created in ASC files using server-side scripts that contain data that can be shared remotely and are written to disk. While the shared objects are on the server-side, they typically communicate with client-side shared objects so that clients (users) can enter data and retrieve it.

Shared objects on the server side are remote by nature. That is, any client connected to the application that gets shared object data from the communication server does so remotely. So references to persistent server-side shared objects for the remainder of the chapter assume that such shared objects are remote as well.

8.4.1. Minimalist Server-Side Shared Object

Using Server-Side Communication ActionScript, the first step is to create a shared object using `SharedObject.get( )`. The format for a persistent server-side shared object is:

```
SharedObject.get("mySO", true)
```

More generically, the statement is:

```
SharedObject.get(name, Boolean)
```

The Boolean value signals whether the data is to be persistent (stored) or not. By default, the shared objects are not persistent (`false`), and so a `true` value needs to be assigned. Leaving the parameter blank also will result in nonpersistent shared objects. In the sample script, the server-side persistent shared object is created right at the top of the script with the following lines:

```
application.onAppStart = function()
{
    trace("Persistent SO  is running");
    myShared = SharedObject.get("cli_so", true);
};
```

Creating the shared object within the `application.onAppStart` event handler is arbitrary; other places in the script would work just as well. The instance of the server-side shared object is `myShared`. The name `cli_so` is used on the client-side as well. Using the common name on both the client- and server-sides facilitates communication between the two sides. However, for a purely server-side script, you could use any name desired.

8.4.2. Adding Properties and Slots to Server-Side Shared Objects

In earlier versions of ActionScript on the client-side, attributes were added to the shared object data property. (You can use the `setDirty()` method if you wish to use the `SharedObject.data` property. I prefer the newer method because it is closer to data assignment on the server-side.) Each attribute creates a unique data slot where values can be stored. However, the server-side shared objects have no data properties. Rather, properties are assigned using `SharedObject.setProperty()`. The statement is applied in the following format:

```
serShare_so.setProperty("nameSO", cname);
```

or more generically:

```
SharedObject.setProperty(property name, value);
```

The first parameter is simply the name (label) you wish to give the property, and the second is the value assigned to that property. A misconception of shared objects is that each shared object is a single slot where shared object data is stored. Actually, a single shared object can have as many slots as you want simply by adding properties to the shared object instance. In this minimalist example, only a single property and hence a single slot is created. The sole slot is named "storage"; think of it as a property of the `myShared` server-side shared object. Example 8-5 shows the server-side script.

Example 8-5. psosend.asc

```
application.onAppStart = function()
{
    trace("Persistent SO  is running");
    myShared = SharedObject.get("cli_so", true);
};
application.onConnect = function(client)
{
    var fromSSSO;
    application.acceptConnection(client);
    client.storeIt = function(dataNow)
    {
          myShared.setProperty("storage",dataNow);
    }
    client.shareIt = function()
    {
          fromSSSO=myShared.getProperty("storage")
          myShared.send("grabIt",fromSSSO+" from storage" );
          trace("PSO sent!")
    };
};
```

In this application the client sends data via the `client.storeIt` function. The value to be stored is passed from the client through a parameter, `dataNow`. Because the server-side script actually places the value into a shared object, you can think of it as a server-side shared object.

8.4.3. Sending Persistent Server-Side Shared Object Data to the Client Side

In a relatively simple procedure, sending shared object data from the server side to the client in past versions of ActionScript took no more than setting up a function in the client code to retrieve and use shared object data sent from the server. However, with ActionScript 3.0, capturing the sent data is far more elaborate. Sending the shared object data from the server is still very simple. The following sections explain the process of sending and capturing data using FMS3 and ActionScript 3.0.

8.4.3.1. Sending Shared Object Data

Data can be retrieved from a persistent server-side object using the `SharedObject.get()` method. The statement has the format:

```
soObj.getProperty("propertyLabel");
```

Typically, the data is passed into a variable and the variable is placed in a `SharedObject.send()` statement. For example, the following script segment sends a server-side shared object to the client:

```
var package = soObj.getProperty("propertyLabel");
soObj.send("clientCatcher",package);
```

So from the server side, sending shared object values to the client is pretty simple using `SharedObject.send()`. The tricky part using ActionScript 3.0 is catching the data and using it on the client. Using ActionScript 2.0 and earlier versions of what was called Client Side Communication ActionScript, you only needed to set up a function with a parameter representing what was sent from the server. However, with ActionScript 3.0, you need to set up something a bit more elaborate.

8.4.3.2. Catching Sends

As you can see, sending data from the server is quite simple. Capturing data by the client is a bit more involved. Figure 8-7 shows the general structure. Along with code from Example 8-5 and Example 8-6, the figure shows the connections between the initial establishing a `NetConnection.client`, a new ActionScript 3.0 property, and the resulting output from the server appearing in the output window.

Figure 8-7. Capturing server information in client

The key to the capturing code is adding an event listener to the shared object `client` property (`cli_so.client`) that can either be used as a variable or output to the Stage. To accomplish the capture, you must create a custom event listener in the `Client` class. (The `Client` class happens to be a common name, but because it is a user class, it can be named anything you want, like `SendCapture` or `ServerSideGrabber`.)

In this application most of the work is setting it up so that once the data is sent to the client through the `Client.grabIt()` method the returned information can be used on the client-side. Recovery is done by making the `Client` class an extension of the `EventDispatcher` class. Then when the data arrives in the client, that event can be used to pass it to the `SharedObject.client`.

8.4.4. Creating the Minimalist Application

Now that you have some idea of how the data is passed back to the client side and captured, it's time to put the client-side application together. Follow these steps to create the client-side data capture application:

1. Open a new Flash file (ActionScript 3.0) and save it as SendCatch.fla.

2. Open the Library panel and add a TextInput, Button, and TextArea component. Save and compact the file.

3. Open a new ActionScript file and save it as SendCatch.as.

4. In the SendCatch.as file, add the code in Example 8-6 and save the file again.

Example 8-6. SendCatch.as

Code View:

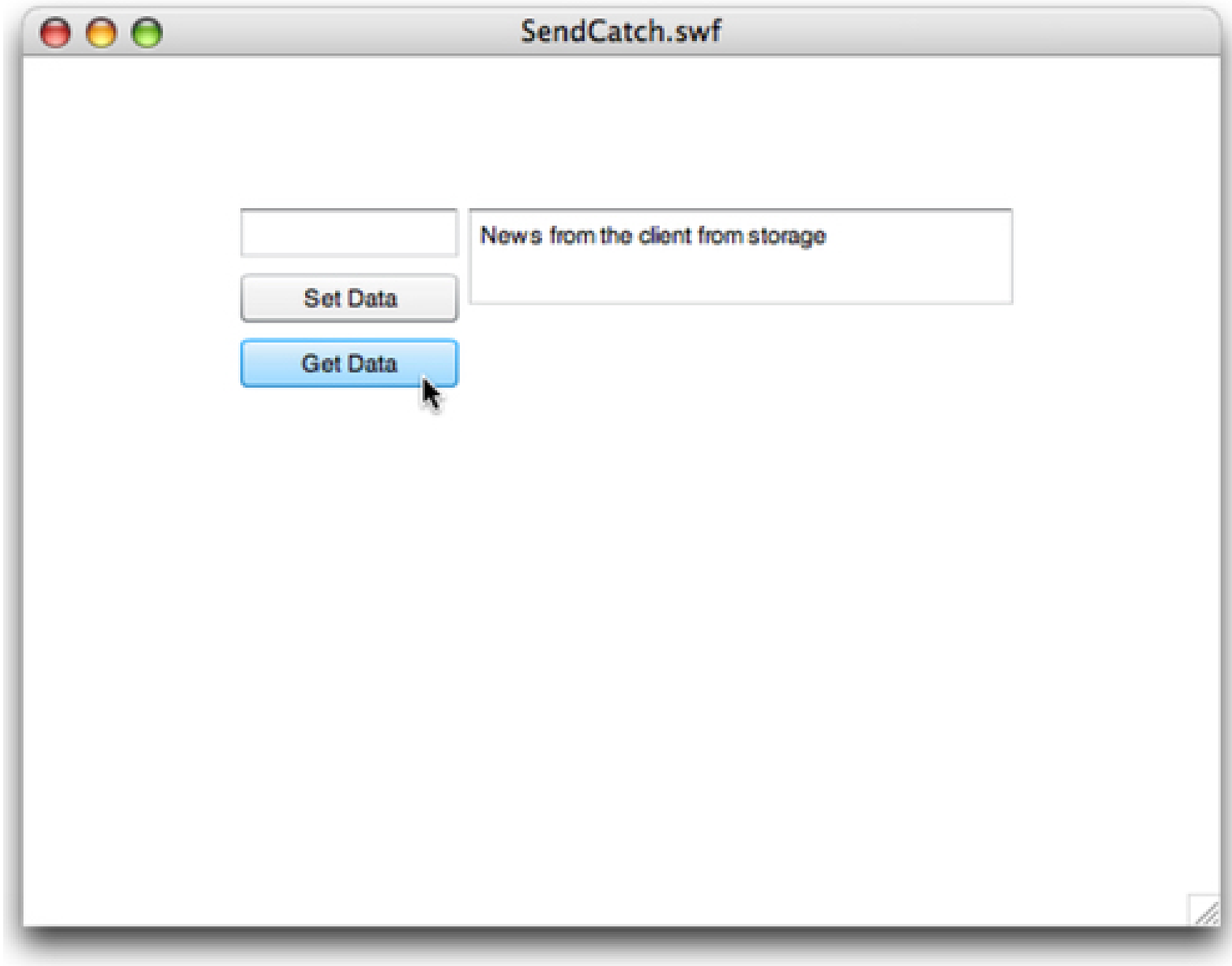
```
package
{
    import flash.display.Sprite;
    import flash.net.SharedObject;
    import flash.events.NetStatusEvent;
    import flash.events.Event;
    import flash.net.NetConnection;
    import flash.events.MouseEvent;
    import fl.controls.Button;
    import fl.controls.TextInput;
    import fl.controls.TextArea;

    public class SendCatch extends Sprite
    {
          private var sent_so:SharedObject;
          private var nc:NetConnection;
          private var good:Boolean;
          private var rtmpNow:String;
          private var button:Button;
          private var setdata:Button;
          private var textInput:TextInput;
          private var textArea:TextArea;
          private var cli_so:SharedObject;
```

```
public function SendCatch ()
{
     button=new Button();
     button.x=100,button.y=130;
     addChild (button);
     button.label="Get Data";
     button.addEventListener (MouseEvent.CLICK,getBack);

     setdata=new Button();
     setdata.x=100,setdata.y=100;
     addChild (setdata);
     setdata.label="Set Data";
     setdata.addEventListener (MouseEvent.CLICK,storeData);

     textInput=new TextInput();
     textInput.x=100,textInput.y=70;
     addChild (textInput);

     textArea=new TextArea();
     textArea.width=250;
     textArea.x=100+textInput.width+5,textArea.y=70;
     addChild (textArea);

     rtmpNow="rtmp://192.168.0.11/psosend/storage";
     nc=new NetConnection;
     nc.connect (rtmpNow);
     nc.addEventListener (NetStatusEvent.NET_STATUS,checkConnection);
}

private function checkConnection (e:NetStatusEvent):void
{
     good=e.info.code == "NetConnection.Connect.Success";
     if (good)
     {
          cli_so=SharedObject.getRemote("cli_so",nc.uri,true);
          cli_so.client=new Client();
          cli_so.connect (nc);
     }
     else
     {
          textArea.text="Connection not established";
     }
}

private function storeData (e:MouseEvent)
{
     nc.call ("storeIt",null,textInput.text);
     textInput.text="";
}

private function displaySent (e:Event)
{
     textArea.text=cli_so.client.RESULT;
     cli_so.removeEventListener (Client.SENTHERE,displaySent);
}

private function getBack (e:MouseEvent)
{
```

```
                cli_so.client.addEventListener (Client.SENTHERE,displaySent);
                nc.call ("shareIt",null);
            }
    }
}

import flash.events.EventDispatcher;
import flash.events.Event;

class Client extends EventDispatcher
{
    public static  var SENTHERE:String="received";
    public var RESULT:String;

    public function grabIt (... rest):void
    {
          RESULT=rest[0];
          dispatchEvent (new Event(Client.SENTHERE));
    }
}
```

When you've finished, test the application. Figure 8-8 shows that the message "News from the client" was sent to the server by entering it in the input text box and pressing the Set Data button. When the Get Data button is pressed, the shared object property is found on the server and sent back with the appended message "from storage."

Figure 8-8. Retrieving data from the server



Using a single property in a server-side shared object is relatively simple, but getting it back using ActionScript 3.0 is a bit more work. In the next section, you will learn how to enter more than a single property and to retrieve groups of data belonging to a single category.

8.5. The Idea Factory

To illustrate how to use more than one server-side property and generate properties on the server side, this next application stores ideas from participants who enter them from any remote application, The ideas are then stored in a persistent shared object. A use case for this application would be where some topic needs to be discussed by more than one group at different times. The user selects the idea category and then enters an idea. To see what others have entered, along with one's own ideas, the user clicks a category, and enters ideas, and then retrieves all ideas by selecting a category and then pressing a button to see all of the added materials.

8.5.1. Multiple Server-Side Properties

As with all variables, unique server-side properties need unique labels. To have more than one idea per property, the properties have to be grouped in some way that will give them both a group and individual identity. To accomplish this, the application uses a counter variable, itself stored as a persistent shared object. The categories and ideas are stored in properties named either `catSO` plus the value of the counter variable, or `ideaSO` plus the value of the counter variable. For example, suppose the value of the counter variable is 21. The category happens to be Wish List; so the label of the property would be `catSO21` with a value `Wish List`. Suppose further that the idea for the Wish List was "3D images." The property name would be `ideaSO21` and its value `3D images`.

The counter variable is stored in a server-side shared object in a property named `counter`. Whenever the `persistClient.dataAdd` call is made, the stored value of the counter property, stored in a persistent shared object slot, is passed to a variable named `checker`. With each new data entry, the value of checker increments and is stored in the server-side shared object, `serShare_so` slot used for the property that tracks the number of ideas that has been stored.

To retrieve all of the ideas for a certain category stored in this way, you need to pass the persistent shared object category name to the server side. In Example 8-7, passing the category value to the server in the following line does this:

```
persistClient.showOff = function(findMe)
```

The `findMe` parameter is simply one of the categories you're using, and `showoff` is the name of the function called from the client. Wherever the `catSO+N` value matches the `findeMe` value, it concatenates the `ideaSO+N` to the return variable (`pSO`). The following line shows this process:

```
pSO +=findMe+": "+serShare_so.getProperty("ideaSO"+sleuth)+"\n";
```

Note that the `findMe` parameter is not looking for a specific idea, but instead is returning all of the ideas in a given category. Once the category has been established, the parameter finds every single idea in the persistent shared object slot where the value is that of the category, and then gets the specific idea by matching the counter number with the idea. Thus, something like the following occurs in retrieving the data:

```
catSO3 = findMe
ideaSO3=idea to get
catSO15 = findMe
ideaSO15=idea to get
catSO21 = findMe
ideaSO21=idea to get
etc....
```

Essentially, it's a grouping algorithm rather than a search for a single matching instance.

1. Open a new ActionScript Communication file and save the file as ideafactory.asc in a folder named ideafactory where you store your server-side applications

2. In the ideafactory.asc file, add the code in Example 8-7 and save your file again.

Example 8-7. ideafactory.asc

Code View:

```
//Persistent SO
application.onAppStart = function()
{
    trace("The Idea Factory is in Business!");
    serShare_so = SharedObject.get("ideas_so", true);
};

application.onConnect = function(persistClient)
{
    application.acceptConnection(persistClient);

    persistClient.dataAdd = function(category, idea)
    {
          //Set up an index to increment with each use
          //Adds a slot to SO for a counter property/attribute
          var checker = serShare_so.getProperty("counter");
          trace(checker);
          if (checker == undefined) {
               checker=1;
               serShare_so.setProperty("counter", checker);
               trace(checker);
          }
          else
          {
               checker++;
               serShare_so.setProperty("counter", checker);
          }
          //Add data to unique slots
          serShare_so.setProperty("catSO"+checker, category);
          serShare_so.setProperty("ideaSO"+checker, idea);
          trace(serShare_so.getProperty("catSO"+checker));
          trace(serShare_so.getProperty("ideaSO"+checker));
    };

    persistClient.showOff = function(findMe)
    {
          trace("Match: "+findMe);
          var pSO;
          var checker = serShare_so.getProperty("counter");
          for (var sleuth = 1; sleuth<checker+1; sleuth++)
          {
               searcher = serShare_so.getProperty("catSO"+sleuth);
```

```
            if (searcher == findMe)
            {
                  var finder = sleuth;
                  if(serShare_so.getProperty("ideaSO"+sleuth)!=undefined)
                  {
                        if(pSO==undefined)
                        {
                  pSO =findMe+": "+serShare_so.getProperty("ideaSO"+sleuth)+"\n";
                        }
                        else
                        {
                  pSO +=findMe+": "+serShare_so.getProperty("ideaSO"+sleuth)+"\n";
                        }
                  }
            }
            else if (sleuth <1)
            {
                  pSO = "Sorry "+persistClient.ip+" no such category.";
            }
      }
      serShare_so.send("goShow", pSO);
   };

   persistClient.killer = function()
   {
         var hitMan=SharedObject.get("ideas_so",true);
         hitMan.clear();
   };
};
```

Now that you have the server-side code set up, all you need is a client-side module to store and retrieve data from the persistent server-side shared object. Since most of the work is done by the server-side code, the client task is to call the proper server-side functions and deal with the response.

8.5.2. Client-Side Modules

The goal of this very simple example has been to keep it clear to allow focusing on persistent server-side shared objects. However, once the parts are all put together, the process is a relatively straightforward one and has many applications where data needs to be shared. Follow these steps to put together an application that sends data to and retrieves it from the server:

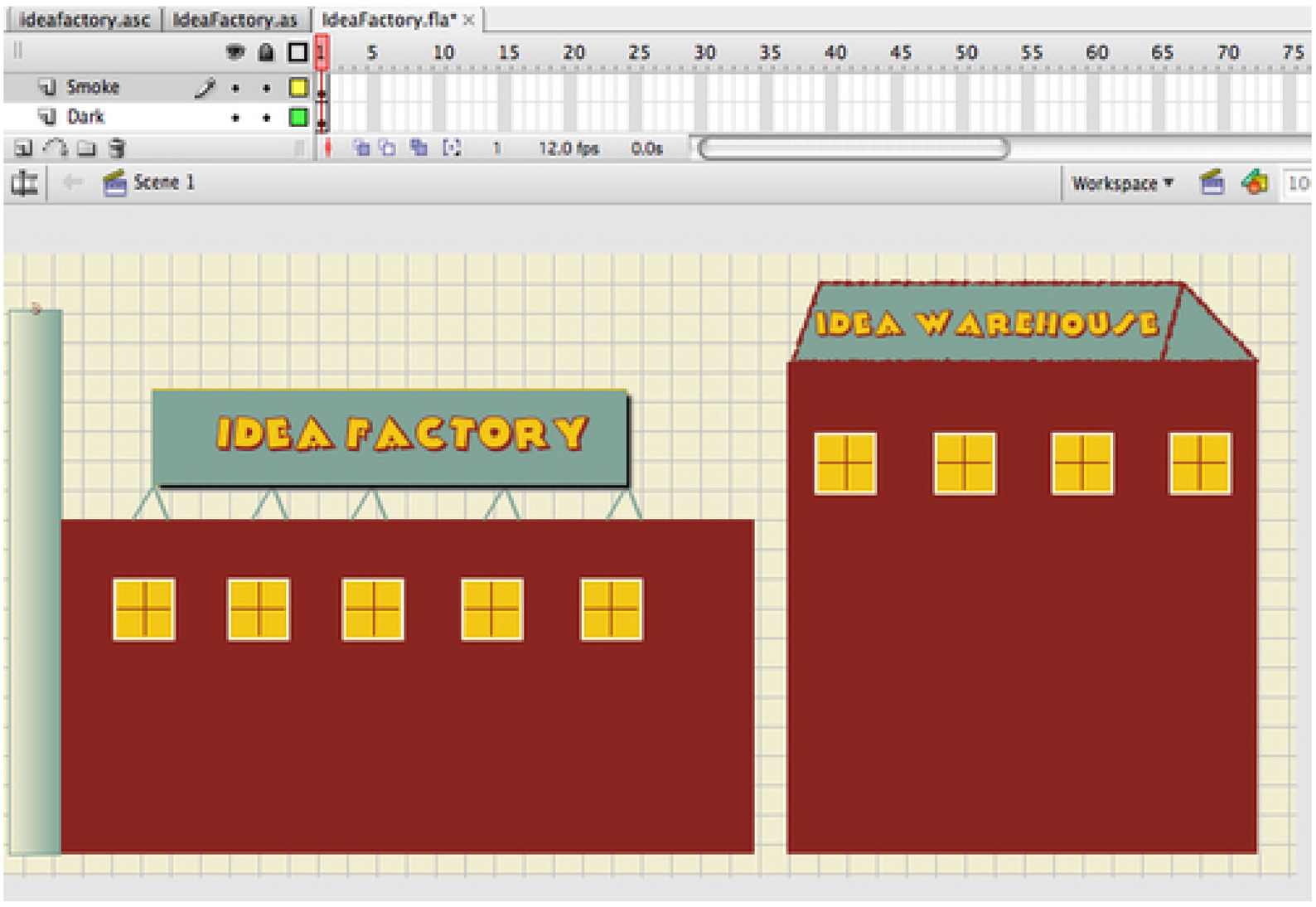
1. Open a new Flash file (ActionScript 3.0) and save it as IdeaFactory.fla. In the Document Class text box in the Property inspector, type **`IdeaFactory`**.

2. Open the Library panel and add Button, TextArea, and List components.

3. As an option, add a graphic backdrop as shown in Figure 8-9. (You actually don't need any backdrop, and I'm sure you can make a more appealing one if you decided to add one.) If you choose to include a graphic backdrop, turn on the grid for help in aligning the x and y positions of the components placed on

the Stage.

Figure 8-9. Background for idea factory



4. Optionally, add a smoke movie clip. Create a movie clip with a `stop()` statement in the first keyframe. Then set up staggered tweens of a graphic 0 or 1 moving from the top of the chimney to the Idea Warehouse's rooftop using the Timeline to create the motion tweens. Use `smoke` as the instance name for the movie clip. If you do not include this option, remove the following line of code in Example 8-8 in the `setIdea()` method:

```
smoke.play();//optional
```

5. When your background is complete, select File Save and Compact.

6. Open a new ActionScript file and save it as IdeaFactory.as in the same folder as IdeaFactory.fla.

7. In the IdeaFactory.as file, enter the code in Example 8-8 and save the file again.

Example 8-8. IdeaFactory.as

Code View:

```
package
{
    import flash.net.NetConnection;
```

```
import flash.net.SharedObject;
import flash.events.NetStatusEvent;
import flash.events.MouseEvent;
import flash.events.SyncEvent;
import flash.events.Event;
import flash.display.Sprite;
import flash.display.MovieClip;
import fl.controls.TextInput;
import fl.controls.Button;
import fl.controls.TextArea;
import fl.controls.List;
import fl.data.DataProvider;


public class IdeaFactory extends Sprite
{
      private var nc:NetConnection;
      private var rtmpNow:String;
      private var good:Boolean;
      private var catInput:List;
      private var catData:DataProvider;
      private var ideaInput:TextArea;
      private var ideaOutput:TextArea;
      private var enterIdea:Button;
      private var retrieveIdeas:Button;
      private var ideas_so:SharedObject;

      public function IdeaFactory ()
      {
           nc=new NetConnection();
           rtmpNow="rtmp://192.168.0.11/ideafactory/thinker";
           nc.addEventListener (NetStatusEvent.NET_STATUS,checkConnection);
           nc.connect (rtmpNow,"test");

           catInput=new List();
           catInput.x=396, catInput.y=252;
           addChild (catInput);
           catData=new DataProvider();
           catData.addItem ({label:"=Category="});
           catData.addItem ({label:"FMS AS 3.0"});
           catData.addItem ({label:"FMS Ser Side"});
           catData.addItem ({label:"Wish List"});
           catData.addItem ({label:"Applications"});
           catInput.dataProvider=catData;
           catInput.selectedIndex=0;

           ideaInput=new TextArea();
           ideaInput.setSize (250,72);
           ideaInput.x=126, ideaInput.y=252;
           addChild (ideaInput);

           enterIdea=new Button();
           enterIdea.x=126, enterIdea.y=(329);
           enterIdea.width=220;
           enterIdea.label="Manufacture idea and send to warehouse";
           addChild (enterIdea);
           enterIdea.addEventListener (MouseEvent.CLICK,setIdea);

           ideaOutput=new TextArea();
```

```
            ideaOutput.setSize (250,162);
            ideaOutput.x=558;
            ideaOutput.y=162;
            addChild (ideaOutput);

            retrieveIdeas=new Button();
            retrieveIdeas.x=558, retrieveIdeas.y=(329);
            retrieveIdeas.width=220;
            retrieveIdeas.label="Select category and click to view ideas";
            addChild (retrieveIdeas);
            retrieveIdeas.addEventListener (MouseEvent.CLICK,getIdeas);

      }

      private function setIdea (e:MouseEvent)
      {
            if (catInput.selectedItem.label != "=Category=")
            {
                  var inCat:String=catInput.selectedItem.label;
                  var inIdea:String=ideaInput.text
                  nc.call ("dataAdd",null,inCat,inIdea);
                  ideaInput.text="";
                  smoke.play();//optional
            }
            else
            {
                  ideaOutput.text="First select category. \n";
            }
      }

      private function getIdeas (e:MouseEvent)
      {
            if (catInput.selectedItem.label != "=Category=")
            {
                  ideas_so.client.addEventListener (Client.SENTHERE,displaySent);
                  nc.call ("showOff",null,catInput.selectedItem.label);
            }
            else
            {
                  ideaOutput.text="First select category. \n";
            }
      }

      private function displaySent (e:Event)
      {
            ideaOutput.text=ideas_so.client.resultNow;
            ideas_so.client.removeEventListener (Client.SENTHERE,displaySent);
      }

      private function checkConnection (e:NetStatusEvent):void
      {
            good=e.info.code == "NetConnection.Connect.Success";
            if (good)
            {
                  ideas_so=SharedObject.getRemote("ideas_so",nc.uri,true);
                  ideas_so.client=new Client();
                  ideas_so.connect (nc);
            }
      }
```

```
    }
}

import flash.events.EventDispatcher;
import flash.events.Event;

class Client extends EventDispatcher
{
    public static  var SENTHERE:String="received";
    public var resultNow:String;

    public function goShow (... rest):void
    {
          resultNow=rest[0];
          dispatchEvent (new Event(Client.SENTHERE));
    }
}
```

When you've entered the code, test it. Figure 8-10 shows what you can expect to see.

Figure 8-10. Displaying stored ideas

If you look carefully, you can see that the "smoke" is drifting from the Idea Factory toward the Idea Warehouse. That means that the "Manufacture ideas and send to warehouse" button was pressed. Prior to that, a user had selected the Applications category and pressed the "Select category and click to view ideas" button. Experiment with the application to get a feel for how persistent server-side shared objects work.

8.5.3. Deleting Persistent Shared Objects

If you followed the server-side script closely, you probably noticed an unused function:

```
persistClient.killer = function()
```

```
{
      var hitMan=SharedObject.get("ideas_so",true);
      hitMan.clear();
};
```

The little segment will erase all persistent shared objects, using the `SharedObject.clear` method. If you want to erase all stored data and start all over again, add the following line to the `checkConnection()` function:

```
nc.call("killer", null);
```

Once you've used it to erase your data, you can comment it out until needed again. Alternatively, you can add a dynamic object to execute the code when needed.

Persistent shared objects represent a way that data can be quickly stored and shared with all clients in an application. The retrieved data can be used to change settings in dynamic elements of the application. This simple example illustrates a practical way to share ideas. However, you can add far more variables for storage and retrieval using the server-side and client-side persistent shared objects.

Chapter 9. Server-Side Streams

9.1. Stream Management

The heart and soul of audio/video in Flash Media Server 3 is the Stream object. You can think of the Stream object as a manager for existing NetStream objects giving you far more options than with NetStream alone. For example, in a broadcast, you can switch the stream from one broadcast source to another and have both publish live and record simultaneously. Imagine an "anchor" with several "reporters" and recorded video. The anchor would be able to switch from the camera on him or her to the camera on a reporter in the field to recorded FLV files.

As a manager, the Stream object can play or record existing streams, such as live NetStream objects created on the client. It can order the playback of recorded MP3 and FLV files. Basically, you can control streams in ways not possible using the client side objects and methods alone.

Chapter 9. Server-Side Streams

9.1. Stream Management

The heart and soul of audio/video in Flash Media Server 3 is the Stream object. You can think of the Stream object as a manager for existing NetStream objects giving you far more options than with NetStream alone. For example, in a broadcast, you can switch the stream from one broadcast source to another and have both publish live and record simultaneously. Imagine an "anchor" with several "reporters" and recorded video. The anchor would be able to switch from the camera on him or her to the camera on a reporter in the field to recorded FLV files.

As a manager, the Stream object can play or record existing streams, such as live NetStream objects created on the client. It can order the playback of recorded MP3 and FLV files. Basically, you can control streams in ways not possible using the client side objects and methods alone.

9.2. Anatomy of **Stream.play( )**

Server-side Communication ActionScript lets you control streams in ways not easily done with client-side scripts. At the heart of the methods is `Stream.play`. In breaking down the parameters, you can find several different options. By beginning with the full list of options, you can examine the different ways you can use the parameters.

```
myStream.play("name",start,length,reset,remoteConnection, virtualKey)
```

The name you play and the name you get are generally different. For example:

```
application.myStream.get("serSideStreamName")
application.myStream.play("newProduct")
```

means that you have created a stream instance that returns the name "serSideStreamName". However, the stream instance itself, `myStream`, can play an existing live stream such as "newProduct", or it can play an FLV or MP3 file. The following shows a sample of what a server-side stream named "sandlight" can stream:

```
application.sandStream.get("sandlight")//Name
if(application.sandStream)
{
  application.sandStream.play("PepTalk")//Live
  application.sandStream.play("recordedTalk")//Recorded FLV
  application.sandStream.play("mp3:Blues")//MP3
}
```

So, whatever the `sandStream` instance plays, it is accessed as "sandlight". To actually see a video or hear a sound, you use a client-side `NetStream` instance. Thus, you can choose to play any number of videos or sounds using the reference to the "sandlight" stream name. For example, the following client-side script would play anything selected by the server-side stream named "sandlight":

```
csStream_ns = new NetStream(nc);
csStream_ns.play("sandlight");
view_video.attachVideo(csStream_ns);
```

On the server-side, the following script:

```
application.sandStream.get("sandlight"),
application.sandStream.play("favVid",-2)
```

would play the video "favVid". So what the user would see and hear when playing "sandlight" on the client side would be whatever was in the FLV file named "favVid". If, however, the server-side script was:

```
sandStream.play("mp3:favSound",-2)
```

the client would play the sound from the MP3 file, "favSound". The essential difference between playing an MP3 file and either a FLV or live stream, is the format. The "mp3:" preface is expected before all MP3 files. Any MP3 file will be played over the live stream "sandlight".

9.2.1. Start, Length, and Reset Parameters

On the face of it, the second parameter, `start` is fairly simple. It starts playing an FLV or MP3 file at the specified time into the file. So a start time of 3 would start playing 3 seconds into the video or MP3 file. However, the `start` parameter also contains two negative values and 0, which have their own meanings.

Table 9-1. Start Parameters

Value	Description	Example
-2	This is the default value. If set to -2 (or no parameter is set), the server attempts to play a live stream of the name in the first parameter. If no live stream is playing, it tries to play a recorded file of the same name. When neither is available, it creates a live stream and waits for a stream to be published to that name.	application.sandStream.play("mp3:flaRock",-2)
-1	If set to -1 the server attempts to play a live stream of the name in the first parameter. If no live stream of that name is playing, it waits for a stream of that name to be published.	application.sandStream.play("liveChat",-1)
0	A 0 or greater setting starts playing the file at the start time. A 0 means to start at the beginning.	application.sandStream.play("fireWorks",0)

The third parameter, `length`, plays the file for the specified number of seconds. However, it too has some special values.

Table 9-2. Length Parameters

-1	If set to -1 the server plays a live stream for as long as the stream lasts, and it plays a recorded file in its entirety.	application.sandStream.play("mp3:Rock",-2,-1)
0	A 0 setting shows only the first frame. If you have several videos and you want to show users a sample, you can use this setting for a "face page" or thumbnail.	application.sandStream.play("datingVid",-2,0)

The fourth parameter, `reset`, expects a Boolean value. The default value is `true` and resets any play list. That means when the action is called, it goes to the top or beginning. Setting this value to `false` can be very useful when you want to play a list of FLV or MP3 files. For example, the following will play a set of three with the first reset true (by default) and the other two false. Whenever the function is called, it resets to the first song in the list and then plays the other two.

```
Client.prototype.playSet = function( )
{
  this.setStream = Stream.get("songs");
  if(setStream)
  {
        this.setStream.play("mp3:flash", 0);
```

```
        this.setStream.play("mp3:Blue", 0, -1, false);
        this.setStream.play("mp3:Jazz", 0, -1, false);
  }
};
```

Using this technique, you can create any number of play lists for an online radio station, or using video, an online news station. All it needs is a client-side control that plays the stream by a reference to its name ("songs") in the above example.

9.3. Playing MP3 Files

To show how to stream a set of MP3 files, this little application is set up like an MP3 player. The key minimalist code is on the server-side, and so you'll start with it. The script provides two different sets of tunes, rock and jazz genres. The user chooses one or the other, and then it's played by a call issued to the server-side code.

The example code uses the exact file names I used in creating the example. You will need to substitute your own music and names of MP3 files. Follow these steps to create an MP3 player application:

1. Create a new folder in the server-side applications folder and name it music. In the music folder, add a folder named streams. In the streams folder, add another folder named selections. In the selections folder, add six MP3 files.

2. Open a new Communications ActionScript file, and save it as music.asc in the music folder.

3. In the music.asc file, add the code in Example 9-1 and save the file again.

Example 9-1. music.asc

Code View:

```
application.onConnect = function(user)
{
    application.acceptConnection(user);
    Client.prototype.songPlay = function(songsNow)
    {
          this.songStream = Stream.get("songs");
          if (this.songStream)
          {
               if (songsNow == "rock")
               {
                    trace("Now playing: "+songsNow);
                    //Add any MP3 files you want
                    this.songStream.play("mp3:Flash", 0);
                    this.songStream.play("mp3:Take5", 0, -1, false);
                    this.songStream.play("mp3:PrideofMan", 0, -1, false);
               }
               else
               {
                    trace("Now playing: "+songsNow);
                    this.songStream.play("mp3:Doit", 0);
                    this.songStream.play("mp3:Fever", 0, -1, false);
                    this.songStream.play("mp3:PieceHeart", 0, -1, false);
               }
          }
    };
    Client.prototype.stopPlay = function(songsNow)
    {
          this.songStream.play(false);
          trace("All play has stopped.");
    };
```

```
};
```

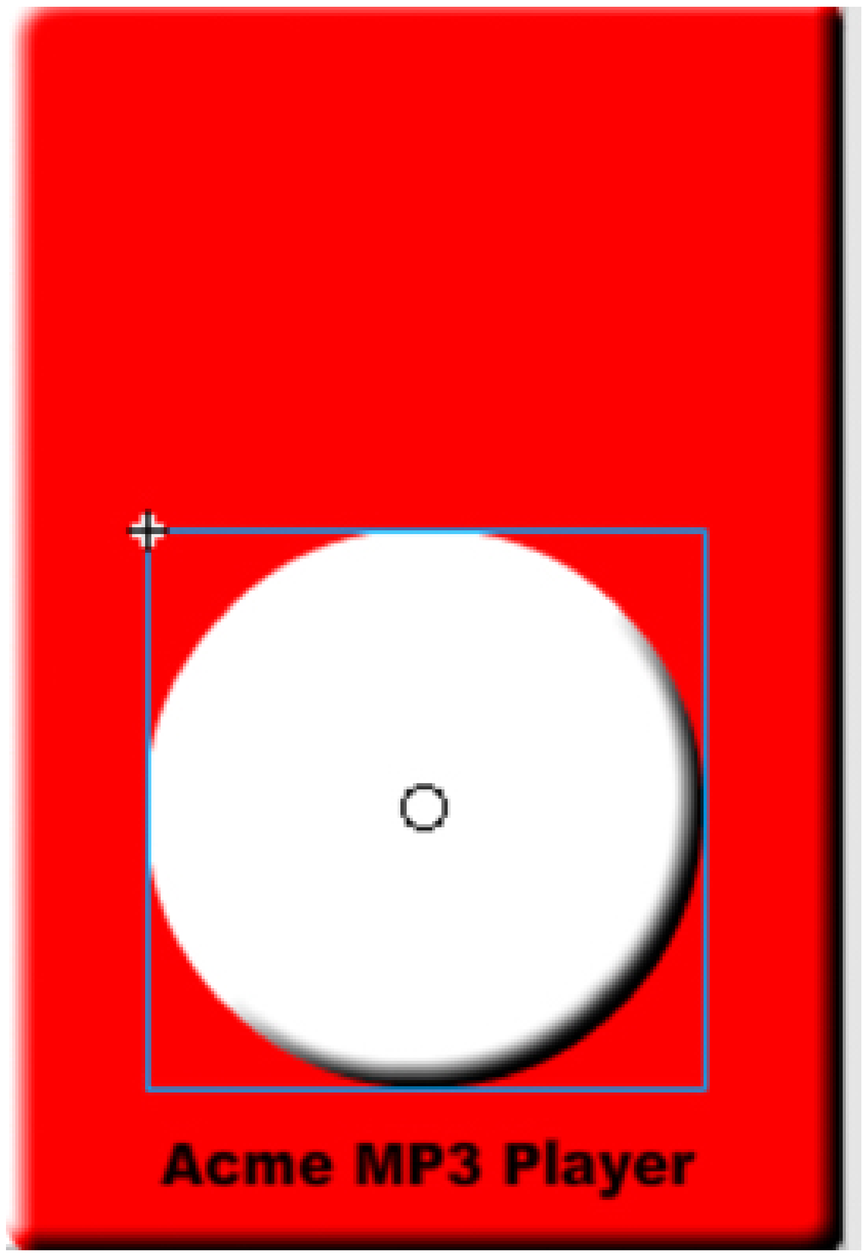
By selecting which group of sounds he or she wishes to hear, the user simply chooses one group or the other, and that selection is sent as a parameter to the server-side function. The function `songPlay` accepts the `songsNow` parameter, which is either rock or jazz. To stop everything, a different call is made to a function that sets the play value to `false`. Thus, the next step is to set up the client-side script to call either of these two functions and pass the selectable parameters.

The client-side application is a little image of an MP3 player with buttons to select the song groups, and to start and stop the music. Follow these steps to build it:

1. Open a new Flash file and save it as Music.fla.

2. Set the Stage size to 180 x 272. Using the Rectangle tool, set the corner curve to a value of 9, and draw a red rectangle that covers the Stage. Convert the rectangle to a movie clip, and add a Bevel filter to it. Using the Oval tool, draw a white circle with a diameter of 120, and position it at x=30, y=116. Convert it to a movie clip and add a Bevel filter to it. At the bottom, add a label of your choice, using Figure 9-1 as a guide. (The code is based on these dimensions and placements, and so if you change them, you will have to change the position properties in the code.)

Figure 9-1. MP3 background image



3. In the Document Class text box in the Property inspector, type **MP3** and save the file.

4. Open a new ActionScript file and save it as MP3.as in the same folder as the MP3.fla file.

5. In the MP3.as file, add the script in Example 9-2 and save the file again.

Example 9-2. MP3.as

Code View:

```
package
{
    import fl.controls.Button;
    import fl.controls.TextArea;
    import flash.display.MovieClip;
    import flash.display.Sprite;
    import flash.net.NetConnection;
    import flash.net.NetStream;
    import flash.events.MouseEvent;
    import flash.events.NetStatusEvent;
    import flash.events.AsyncErrorEvent;

    public class MP3 extends Sprite
    {
          private var playBtn:Button;
          private var backBtn:Button;
```

```
private var fwdBtn:Button;
private var textArea:TextArea;
private var rock:Boolean=true;
private var tuneGroup:String;
private var rtmpNow:String;
private var nc:NetConnection;
private var good:Boolean;
private var music:NetStream;
private var initial:Boolean=false;

public function MP3 ()
{
     nc=new NetConnection();
     rtmpNow="rtmp://192.168.0.11/music/selections";
     nc.addEventListener (NetStatusEvent.NET_STATUS,checkConnect);
     nc.connect (rtmpNow);
}

private function checkConnect (e:NetStatusEvent)
{
     good=e.info.code == "NetConnection.Connect.Success";
     if (good)
     {
          music=new NetStream(nc);
          music.addEventListener (AsyncErrorEvent.ASYNC_ERROR,tuneChange);
          doComponents ();
          nc.removeEventListener (NetStatusEvent.NET_STATUS,checkConnect);
     }
     else
     {
          trace ("Problem: "+e.info.code);
     }
}

private function tuneChange (e:AsyncErrorEvent)
{
     textArea.text="New tune starting";
}

private function rockJazz (e:MouseEvent)
{
     if (!initial)
     {
          textArea.text="Rock: \n--Flash\n--Who Are You?\n--Mexico";
          rock=true;
          initial=true;
     }
     else
     {
          textArea.text="Jazz: \n--Do It Again\n--Fever\n--Take 5";
          rock=false;
          initial=false;
     }
}

private function playStop (e:MouseEvent)
{
     initial=true;
     if (playBtn.label=="Play")
```

```
            {
                  playBtn.label="Stop";
                  if (rock)
                  {
                        nc.call ("songPlay",null,"rock");
                  }
                  else
                  {
                        nc.call ("songPlay",null,"jazz");
                  }
            }
            else
            {
                  playBtn.label="Play";
                  nc.call ("stopPlay",null);
            }
            music.play ("songs");
      }

      private function doComponents ()
      {
            backBtn=new Button();
            backBtn.width=25;
            backBtn.label="<<";
            backBtn.x=34.5,backBtn.y=165;
            addChild (backBtn);
            backBtn.addEventListener (MouseEvent.CLICK,rockJazz);

            playBtn=new Button();
            playBtn.width=37;
            playBtn.label="Play";
            playBtn.x=69,playBtn.y=164;
            addChild (playBtn);
            playBtn.addEventListener (MouseEvent.CLICK,playStop);

            fwdBtn=new Button();
            fwdBtn.width=25;
            fwdBtn.label=">>";
            fwdBtn.x=113.5,fwdBtn.y=165;
            addChild (fwdBtn);
            fwdBtn.addEventListener (MouseEvent.CLICK,rockJazz);

            textArea=new TextArea();
            textArea.width=160, textArea.height=95;
            textArea.x=10;
            textArea.y=12;
            addChild (textArea);
      }
   }
}
```

Once you have all of your files where they belong, test the application. Select either "Rock" (the default) or "Jazz"; then click the Play button. You'll hear all of the tunes you placed in the selected category. Figure 9-2 shows the completed application working in the test mode.

Figure 9-2. Completed MP3 player

You can easily change the server-side script to accommodate additional MP3 selections. Just add the MP3 files to the selections folder (on the server side) and add lines after the last line in the group. For example, the following shows two additional MP3 tunes added:

```
this.songStream.play("mp3:PieceHeart", 0, -1, false);//Last existing
this.songStream.play("mp3:Jazzwork", 0, -1, false);//New tune
this.songStream.play("mp3:Stormclouds", 0, -1, false);//New tune
```

It's a simple application, but one that you'll find quite flexible and useful whenever you want to add play lists to your application.

9.3.1. Selecting FLV Files to Play with Server-Side Script

Because the server-side Stream class acts like a net stream manager, you can assign a `Stream` instance name, using the format:

```
client.myInstance = new Stream("serStream");
```

Then, whatever `myInstance` plays is streamed as *serStream*. So, using a client-side NetStream, if you play, *serStream*, it actually plays whatever *serStream* is playing. Thus,

```
client.myInstance.play
```

This next application is a very simple one, showing how to call any file using a variable passed to the server-side. The following server-side script is saved as chooseflv.asc in a folder named chooseflv on the server-side. Also, you will need to create a streams folder and, inside this, a folder named vault. Place the FLV files you want to play in chooseflv → streams → vault.

First, create the following script:

Example 9-3. chooseflv.asc

Code View:

```
//Record live stream
application.onAppStart = function()
{
    trace("Record live app.");
};
application.onConnect = function(user)
{
    application.acceptConnection(user);
    Client.prototype.recordLive = function(flvName)
    {
          this.liveStream = Stream.get(flvName);
          if (this.liveStream)
          {
               this.liveStream.play("liveOne");
               this.liveStream.record();
          }
    };

    Client.prototype.playRecorded = function(flvName)
    {
          this.recordedStream = Stream.get("recordedOne");
          if (this.recordedStream)
          {
               this.recordedStream.play(flvName,-2);
               trace("Playing recorded");
          }
    };

    Client.prototype.stopPlayback = function()
    {
          this.recordedStream.play(false);
          trace("Playing "+flvName+" has stopped.");
    }

    Client.prototype.stopRecord = function()
    {
          this.liveStream.record(false);
          this.liveStream.flush();
          trace("Recording has stopped.");
```

```
    }
};
```

Like the previous example, the application contains only two functions that can be called from the client side. The `chooseFLV` function is primary. The client side passes the parameter `flvName`, which is used as the name of the file to play. The `stopPlay` function simply stops playing the stream. That's all there is to it-a very simple but useful application for running FLV files from the client side using server-side controls.

9.3.2. Client Names One Stream, Can Choose Many Streams

When you use the server-side `Stream` class to manage what to play, the client simply sets up a `NetStream` instance and issues a `NetStream.play()` to play a single stream name. Then, by passing different file names or names of live streams to the server, it turns over the management of what is actually seen or heard to the server. In this particular application, the server Stream uses the name, "flvPlay" as a reference:

```
this.flvStream = Stream.get("flvPlay");
```

On the client side, the `NetStream` instance plays that stream name using the line,

```
ns.play ("flvPlay");
```

The selection of what to play is handled by passing a name to the server function using the `NetConnection.call` method. Then on the server, name is passed on as "flvName" and is played on the stream "flvPlay."

```
this.flvStream.play(flvName, -2);
```

All the client script has to do is to set up a `NetStream` instance to play the server `Stream` instance and send a string to name the FLV file you want played. In fact, if you had more than a single live stream going through the server, you could select which live stream to play. (In the next section you'll see how that's done.) Follow these steps to create all of the elements for the client side of the application:

1. Open a new Flash file and save it as ChooseFlv.fla.

2. Open the Library panel and drag a TextInput and Button component into the Library.

3. In the Document Class text box in the Property inspector, type **`ChooseFlv`**. Save the file again.

4. Open a new ActionScript file and save it as ChooseFlv.as in the same folder as the ChooseFlv.fla file.

5. In the ChooseFlv.as file, enter the code in Example 9-4 and save the file again.

Example 9-4. Chooseflv.as

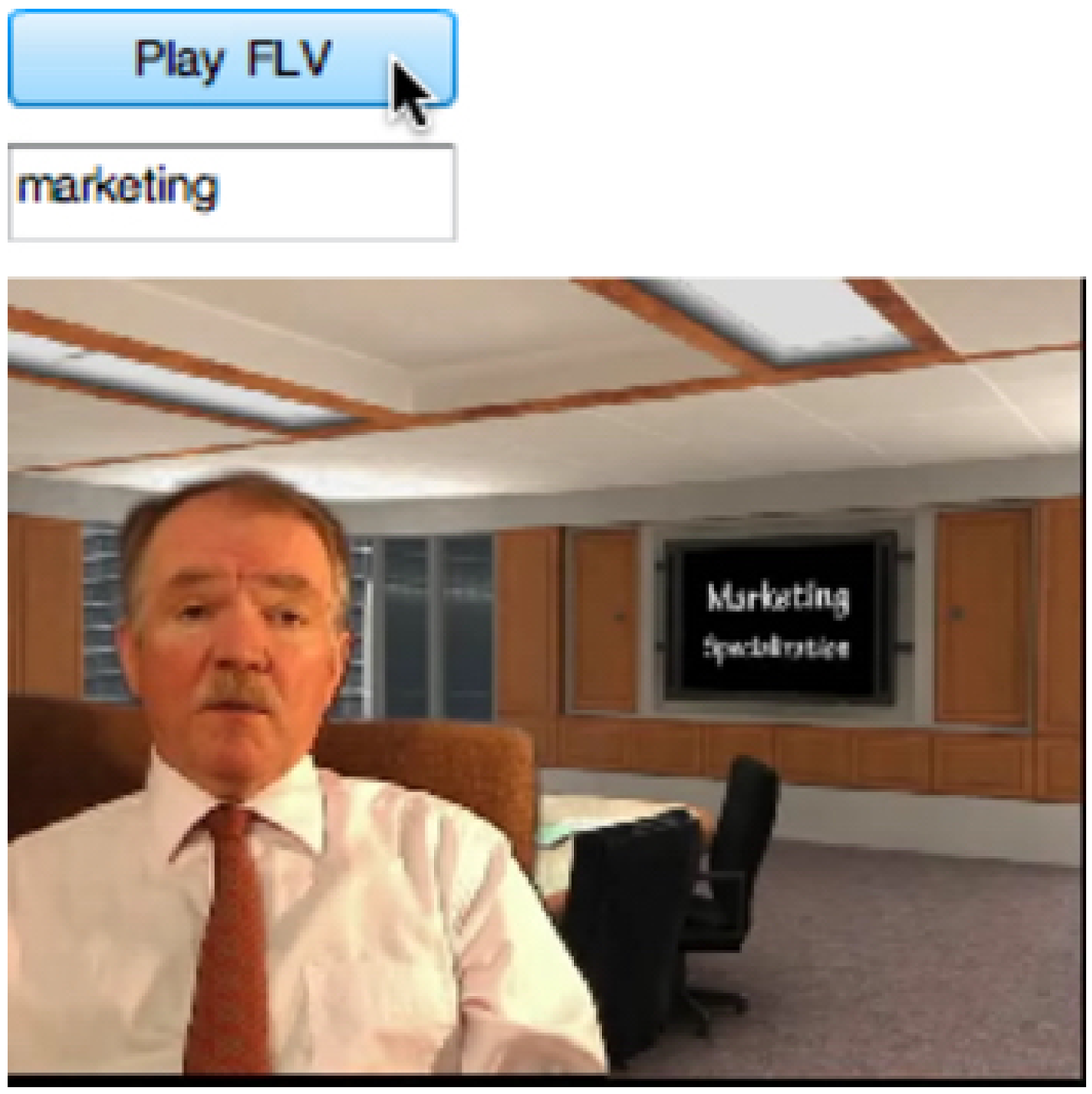
Code View:

```
package
{
    import flash.net.NetConnection;
    import flash.net.NetStream;
    import flash.events.NetStatusEvent;
    import flash.media.Video;
    import flash.display.Sprite;
    import flash.events.MouseEvent;
    import fl.controls.TextInput;
    import fl.controls.Button;


    public class ChooseFlv extends Sprite
    {
          private var nc:NetConnection;
          private var ns:NetStream;
          private var rtmpNow:String;
          private var good:Boolean;
          private var textInput:TextInput;
          private var vid:Video;
          private var button:Button;
          private var checkBtn:Boolean;
          private var dur:Number;
          private var metaSniffer:Object;

          public function ChooseFlv ()
          {
               button=new Button ;
               button.x=200,button.y=100;
               addChild (button);
               button.label="Play FLV";
               button.addEventListener (MouseEvent.CLICK,playFlv);

               textInput=new TextInput ;
               textInput.x=200,textInput.y=130;
               addChild (textInput);

               vid=new Video(240,180);
               vid.x=200,vid.y=160;
               addChild (vid);

               nc=new NetConnection ;
               rtmpNow="rtmp://192.168.0.11/chooseflv/vault";
               nc.connect (rtmpNow);
               nc.addEventListener (NetStatusEvent.NET_STATUS,checkConnect);
          }

          private function checkConnect (e:NetStatusEvent)
          {
               good=e.info.code == "NetConnection.Connect.Success";
               if (good)
               {
                    ns=new NetStream(nc);
```

```
                metaSniffer=new Object ;
                ns.client=metaSniffer;
                metaSniffer.onMetaData=getMeta;
          }
     }

     private function getMeta (mdata:Object):void
     {
          dur=mdata.duration;
     }

     private function playFlv (e:MouseEvent)
     {
          checkBtn=button.label == "Play FLV";
          if (checkBtn)
          {
                nc.call ("chooseFLV",null,textInput.text);
                vid.attachNetStream (ns);
                ns.play ("flvPlay");
                button.label="Stop";
          }
          else
          {
                nc.call ("stopPlay",null);
                ns.play (false);
                button.label="Play FLV";
          }
     }
}
```

When you run the program, be sure to have at least one FLV file in the vault folder on the server side. Figure 9-3 shows what you can expect to see when you test your application.

Figure 9-3. Selected video playing by server control



The name of the file you played is marketing.flv, which would have used the following code from a purely client-side application:

```
ns.play("marketing")
```

However, as you can see, the client-side code is the following

```
ns.play ("flvPlay");
```

In this way, the server-side Stream class can manage as many different streams as you like. The client provides the name of the resources (recorded or live) to add to the stream.

9.3.3. Simultaneous Live and Recorded Streams

A question that often comes up is how to both send a live stream and record it at the same time. For example, a client may need to record his presentations for later viewing by participants who couldn't attend the live presentation. One way to record a live chat is to play and record the live stream using the server-side Stream class. The format:

```
this.liveStream = Stream.get("serStream");
if (this.liveStream)
{
this.liveStream.play("cliLive");
this.liveStream.record( );
}
```

works to both play and record a stream. The effect is to record a live stream. The FLV file is saved as the name of the server-side stream ("serStream" in the previous example). So, if the client-side `NetStream` plays a live stream named "cliLive" it continues to be streamed live, and at the same time recorded as "serStream" in the previous example. To build an application that would use this technique, you'll start with the server-side ActionScript. This application will have two pairs of methods. One pair begins the recording of a live stream and stops the recording. The other pair starts and stops the playback of the recorded file. Follow these steps to build an application that plays a live stream and records it at the same time

1. Create a folder named liveRecord in the server-side applications folder.

2. Open a new ActionScript Communication file, and save it as liverecord.asc in the liverecord folder stored in the applications folder with the other server-side folders.

3. In the liverecord.asc file, add the code in Example 9-5 and save the file again.

Example 9-5. liverecord.asc

Code View:

```
//Record live stream
application.onAppStart = function()
{
    trace("Record live app.");
};
application.onConnect = function(user)
{
    application.acceptConnection(user);
    Client.prototype.recordLive = function(flvName)
    {
          this.liveStream = Stream.get(flvName);
          if (this.liveStream)
          {
               this.liveStream.play("liveOne");
               this.liveStream.record();
          }
    };

    Client.prototype.stopRecord = function()
    {
          this.liveStream.record(false);
          this.liveStream.flush();
          trace("Recording has stopped.");
    }

    Client.prototype.playRecorded = function(flvName)
    {
          this.recordedStream = Stream.get("recordedOne");
          if (this.recordedStream)
          {
               this.recordedStream.play(flvName,-2);
               trace("Playing recorded");
          }
    };
```

```
    Client.prototype.stopPlayback = function()
    {
          this.recordedStream.play(false);
          trace("Playing "+flvName+" has stopped.");
    }

};
```

The key function `recordLive` accepts a parameter, `flvName`, used to name the server-side `Stream` instance's stream. Another function, `stopRecord`, stops the recording. A second pair of functions plays the recorded FLV file (`playRecorded`) and another to stop the playback (`stopPlayback`). On the client side, you simply need to create `NetStream` instances to play the server-side streams whether live or recorded. Follow these steps to complete the application:

0. Open a new Flash file and save it as LiveRecord.fla.

1. Open the Library panel and place a Button and TextInput component in the Library.

2. Open the Property inspector and in the Document Class text box, type **`LiveRecord`**. Save the file again.

3. Open a new ActionScript file and save it as LiveRecord.as in the same folder as LiveRecord.fla file.

4. In the LiveRecord.as file add the code in Example 9-6 and save the file.

Example 9-6. LiveRecord.as

Code View:

```
package
{
    import flash.net.NetConnection;
    import flash.net.NetStream;
    import flash.events.NetStatusEvent;
    import flash.media.Video;
    import flash.media.Camera;
    import flash.media.Microphone;
    import flash.display.Sprite;
    import flash.events.MouseEvent;
    import fl.controls.TextInput;
    import fl.controls.Button;


    public class LiveRecord extends Sprite
    {
          private var cam:Camera;
          private var mic:Microphone;
```

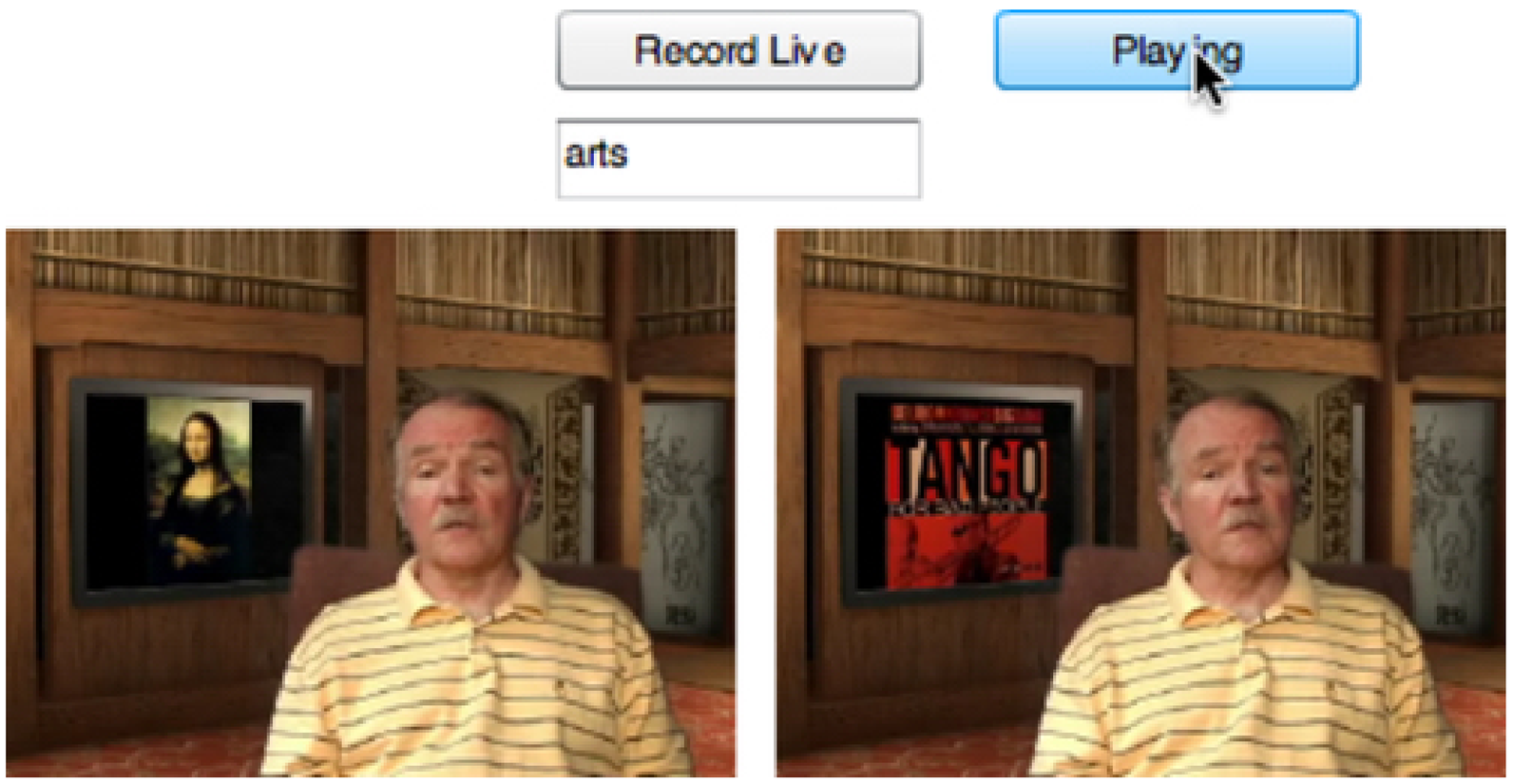
```
private var nc:NetConnection;
private var ns:NetStream;
private var in_ns:NetStream;
private var recorded_ns:NetStream;
private var rtmpNow:String;
private var good:Boolean;
private var textInput:TextInput;
private var recordVid:Video;
private var liveVid:Video;
private var button:Button;
private var playback:Button;
private var checkBtn:Boolean;
private var dur:Number;
private var metaSniffer:Object;

public function LiveRecord ()
{
     cam=Camera.getCamera();
     mic=Microphone.getMicrophone();

     button=new Button;
     button.x=200,button.y=100;
     addChild (button);
     button.label="Record Live";
     button.addEventListener (MouseEvent.CLICK,recordFlv);

     playback=new Button ;
     playback.x=320,playback.y=100;
     addChild (playback);
     playback.label="Playback FLV";
     playback.addEventListener (MouseEvent.CLICK,playFlv);

     textInput=new TextInput ;
     textInput.x=200,textInput.y=130;
     addChild (textInput);

     recordVid=new Video(200,150);
     recordVid.x=260,recordVid.y=160;
     addChild (recordVid);

     liveVid=new Video(200,150);
     liveVid.x=50,liveVid.y=160;
     addChild (liveVid);

     nc=new NetConnection ;
     rtmpNow="rtmp://192.168.0.11/liverecord/reels";
     nc.connect (rtmpNow);
     nc.addEventListener (NetStatusEvent.NET_STATUS,checkConnect);
}

private function checkConnect (e:NetStatusEvent)
{
     good=e.info.code == "NetConnection.Connect.Success";
     if (good)
     {
          liveVid.attachCamera (cam);
          ns=new NetStream(nc);
          ns.attachCamera (cam);
          ns.attachAudio (mic);
```

```
                ns.publish ("liveOne");

                in_ns=new NetStream(nc);
                in_ns.play ("liveOne");
                recordVid.attachNetStream (in_ns);

                metaSniffer=new Object();
                recorded_ns=new NetStream(nc);
                recorded_ns.client=metaSniffer;
                metaSniffer.onMetaData=getMeta;
          }
     }

     private function getMeta (mdata:Object):void
     {
          dur=mdata.duration;
     }
     private function recordFlv (e:MouseEvent)
     {
          checkBtn=button.label == "Record Live";
          if (checkBtn)
          {
                nc.call ("recordLive",null,textInput.text);
                button.label="Recording";
          }
          else
          {
                nc.call ("stopRecord",null);
                button.label="Record Live";
          }
     }

     private function playFlv(e:MouseEvent)
     {
          checkBtn=(playback.label == "Playback FLV");
          if (checkBtn)
          {
                nc.call ("playRecorded",null,textInput.text);
                recorded_ns.play ("recordedOne");
                recordVid.attachNetStream (recorded_ns);
                playback.label="Playing";
          }
          else
          {
                nc.call ("stopPlayback",null);
                playback.label="Playback FLV";
                recordVid.attachNetStream (in_ns);
          }

     }
}
```

The `Camera` and `Microphone` instances use default settings, but you can easily optimize them by adding the appropriate methods and values. One reason that this application shows the local and the live stream is to give

the user a good idea of what is being recorded (the live stream, not the local one). As a minimalist example, this application highlights the ability to both stream live video and record it at the same time. Note that the code for the left video (`liveVid`) attaches the `Camera` and `Microphone` instances directly to the video, but the right video (`recordVid`) attaches a stream (`in_ns`). Then a second `NetStream` object (`in_ns`) is played with its contents attached to the `recordVid` instance. In this way you can see the actual stream coming from the server and local `Camera` and `Microphone` instances stream simultaneously. Figure 9-4 shows a live stream on the left and the playback of a recorded stream on the right.

Figure 9-4. Local video and recorded playback



Once the live stream is established, a function call to the server-side,

```
nc.call ("recordLive",null,textInput.text);
```

takes the same stream and records it. Thus, using a single published stream, you can both play and record simultaneously. By the same token, you can also play back a recorded FLV file using calls to the server.

9.4. Changing Streams

A powerful and useful application employing client- and server-side streams together is broadcasting different streams on demand. For example, suppose you want to take the TV studio from Chapter 6, "Broadcasting and Server-Side Bandwidth Control," and be able to switch from the anchor to a reporter in the field or a recorded video-just like the evening news. The viewer sees whatever you switch to-the anchor, a reporter, or a video. This system works with a single broadcasting stream on the server side. The broadcasting stream plays whatever stream name it's assigned, while the news room continuously plays a key stream that is the server-side stream's name.

9.4.1. Server-Side Code

Like the other applications in this chapter, this one hinges on the server-side script. In fact, it hinges on just a single function, as shown in Figure 9-5. Streams are generated from either a live source or a recorded one. The name of the stream to be played is passed in the parameter `reporterNow` by the Control module. The Viewer module plays only a single stream named news, and so the viewer sees whatever is passed in `reporterNow` and played in the news stream.

Figure 9-5. Function to switch selected media to stream

Tests with the application showed that the buffer would often fill and keep playing a recorded FLV stream or even live stream long after the Control module had switched to another stream. To help alleviate that problem, I added a buffer reset that set the buffer to zero (`0`) whenever the Control module switched from one stream to another. So the `setBufferTime(0)` reflects resetting the buffers when a stream swap occurs. Finally, in order to play MP3 files using the standard .mp3 extension for the file names, the script includes a function to examine the contents of the file name and determine whether it contains the .mp3 extension. If it does, it then rearranges the file name to the format required for MP3 files (mp3: filename). After experimentation, I found that mixing MP3 files with FLV and live streams is not a good idea. Playing a group of MP3 files with this application works fine and so too does mixing live streams with FLV files. However, when an MP3 file is selected, it plays, but on switching back to a live stream or an FLV file, the program freezes for several seconds before the new stream begins playing. So while the example shows how to add MP3 files to the mix, I'd recommend using them only when working solely with MP3 files. Use a separate stream for live video files.

With this in mind, the following shows the entire server-side script. Save the script as *streamwork.asc*.

Example 9-7. streamwork.asc

Code View:

```
application.onAppStart=function()
{
     trace("News is on!");
};
application.onConnect=function(user)
{
    application.acceptConnection(user);
    Client.prototype.selectReporter = function(reporterNow)
    {
          if (reporterNow.substr(reporterNow.indexOf("."), 4) == ".mp3")
          {
               reporterNow = "mp3:"+reporterNow.substring(0, reporterNow.length-4);
          }
          this.reporterStream = Stream.get("news");
          this.reporterStream.setBufferTime(0);
          this.reporterStream.play(reporterNow, -2);
          trace("Now playing: "+reporterNow);
    };
};
```

When you've finished the server-side script, this application calls for three client-side modules, including:

- Control module that selects the different media to stream.
- Reporter module for streaming out live audio/video.
- TV set module for viewing selected stream.

All three modules connect to the same server-side application. So you can think of the application as one server-side module and three client-side modules.

9.4.2. Control Module

The Control module is like a control room in a TV studio. The producer can switch between the anchor and both live and recorded feeds. From the TVProducer class, you can add any name you want and switch what the viewer sees. The Control module is a simple one that controls which stream the viewer sees. The "producer" who runs the module adds names of live reporters or recorded files. If the file is an MP3 file, it must be entered in a format that includes the MP3 extension. Reporter names should be added exactly as the reporter enters them, requiring some coordination between the producer and recorder. For example the following would be expected:

- anchor (default).
- Tom Smith (reporter name).

- Nancy (reporter name).
- jazz.mp3 (MP3 file).
- Movie1 (FLV file).
- Movie2 (FLV file).

All of the names added to the list can be clicked to generate the live or recorded media that the viewer will see. This particular module has a top screen that shows the local camera and microphone, and a bottom screen where the producer can see what is currently selected. If both are the same, that means that the "anchor" is in the receiver's view window.

Follow these steps to set up the Control module:

1. Open a new Flash file and save it as *TVProducer.fla* in a folder with your other client-side files.
2. Open the Library panel and drag a Button, List, and TextInput component into the Library.
3. Add a Static text label "TV Producer" to the top of the stage. The label is handy for quickly differentiating it from the other modules. (You can get as fancy as you want in decorating the TV studio.) In the Property inspector Document Class text box, type **`TVProducer`** and save the file.
4. Open a new ActionScript file and save it as *TVProducer.as* with the *TVProducer.fla* file.
5. In the *TVProducer.as* file, add the code in Example 9-8 and save the file again.

Example 9-8. TVProducer.as

Code View:

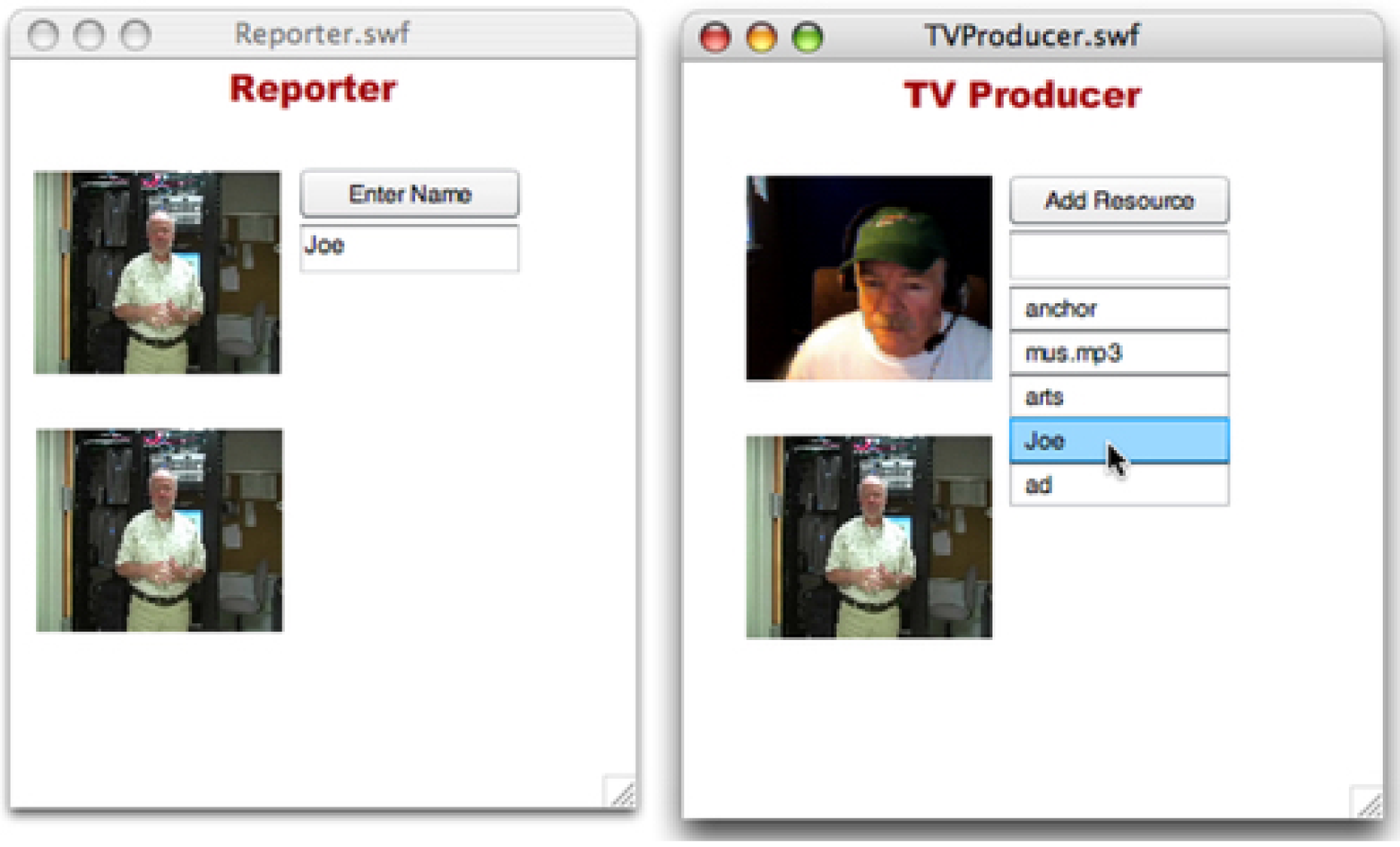
```
package
{
    import flash.net.NetConnection;
    import flash.net.NetStream;
    import flash.events.NetStatusEvent;
    import flash.events.Event;
    import flash.events.MouseEvent;
    import flash.display.Sprite;
    import flash.media.Video;
    import flash.media.Camera;
    import flash.media.Microphone;
    import fl.controls.List;
    import fl.data.DataProvider;
    import fl.controls.TextInput;
    import fl.controls.Button;
```

```
public class TVProducer extends Sprite
{
      private var nc:NetConnection;
      private var ns:NetStream;
      private var studio:NetStream;
      private var rtmpNow:String;
      private var good:Boolean;
      private var vid:Video;
      private var vidStream:Video;
      private var cam:Camera;
      private var mic:Microphone;
      private var list:List;
      private var resource:DataProvider;
      private var resourceList:Array;
      private var textInput:TextInput;
      private var button:Button;
      private var currentResource:String;
      private var metaSniffer:Object;

      public function TVProducer ()
      {
            nc=new NetConnection();
            rtmpNow="rtmp://192.168.0.11/streamwork/studio";
            nc.connect (rtmpNow);
            nc.addEventListener (NetStatusEvent.NET_STATUS,checkConnect);

            textInput=new TextInput();
            textInput.x=265, textInput.y=105;
            addChild (textInput);

            button=new Button();
            button.label="Add Resource";
            button.x=265, button.y=80;
            button.addEventListener (MouseEvent.CLICK,addResource);
            addChild (button);

            list=new List();
            list.x=265, list.y=130;
            addChild (list);
            list.addEventListener (Event.CHANGE,changeStream);

            resourceList=new Array("anchor");
            list.dataProvider=new DataProvider(resourceList);

            cam=Camera.getCamera();
            cam.setMode (112,93,10);
            cam.setQuality (100000/8,0);

            vid=new Video(112,93);
            vid.x=145,vid.y=80;
            vid.attachCamera (cam);
            addChild (vid);

            vidStream=new Video(112,93);
            vidStream.x=145,vidStream.y=197;
            addChild (vidStream);

            mic=Microphone.getMicrophone();
            mic.rate=11;
```

```
        }

        private function checkConnect (e:NetStatusEvent)
        {
             good=e.info.code == "NetConnection.Connect.Success";
             if (good)
             {
                  currentResource="anchor";
                  studio=new NetStream(nc);
                  studio.attachAudio(mic);
                  studio.attachCamera(cam);
                  studio.publish(currentResource);

                  ns=new NetStream(nc);
                  ns.play("news");
                  vidStream.attachNetStream(ns);

                  metaSniffer=new Object();
                  ns.client=metaSniffer;
                  metaSniffer.onMetaData=getMeta;
             }
             else
             {
                  trace (e.info.code);
             }
        }

        private function getMeta (mdata:Object):void
        {
             //Dummy function
        }

        private function addResource (e:MouseEvent)
        {
             resourceList.push (textInput.text);
             list.dataProvider=new DataProvider(resourceList);
             textInput.text="";
        }

        private function changeStream (e:Event)
        {
             currentResource=list.selectedItem.label;
             nc.call("selectReporter",null,currentResource);
        }
    }
}
```

When you test the application initially, all you will see is yourself and the word "anchor" in the List component. If you click "anchor," you will see yourself in the bottom window. Once you have others using the Reporter module, you can switch to them or a recorded file. For example, Figure 9-6 shows the TV Producer and Reporter modules side by side. The producer has selected the external reporter named "Joe"; so both Joe and the producer see the currently streaming video in the bottom window.

Figure 9-6. Producer selecting reporter to stream



Later in this chapter, "Section 9.4.4" describes how to create the module for the remote reporter. Until then, you can add recorded FLV and MP3 files to the studio folder inside the application's streams folder. Just add their name to the input window beneath the Add Resource button. Do not add the .flv extension to FLV files, but with MP3 files, you need to add the .mp3 extension, as shown in Figure 9-6. By clicking the different names, the producer streams whatever video he or she wants.

9.4.3. Viewer Module

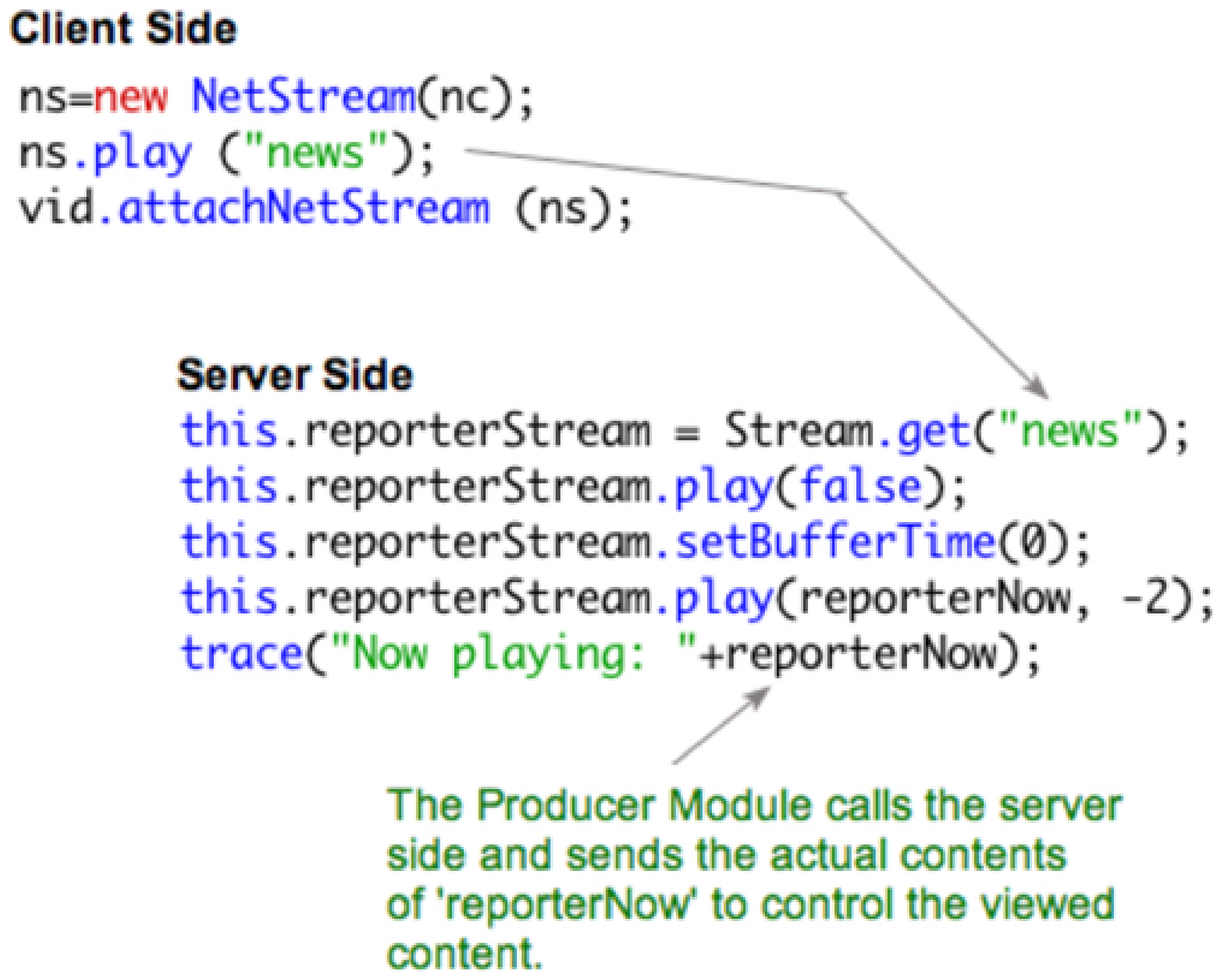
The Viewer module plays the same named stream news, but that stream changes because it is a server-side stream object subject to playing different names for the same stream instance. You'll have to imagine that your TV has only one channel, and just like the evening news where different recorded and live feeds are switched, you'll see the current selected media. (You can even add commercials if you want.)

The Viewer module is fairly passive. It streams in whatever the server is commanded by the command module. The following three lines pretty much handle this application's client-side streaming:

```
ns=new NetStream(nc);
ns.play ("news");
vid.attachNetStream (ns);
```

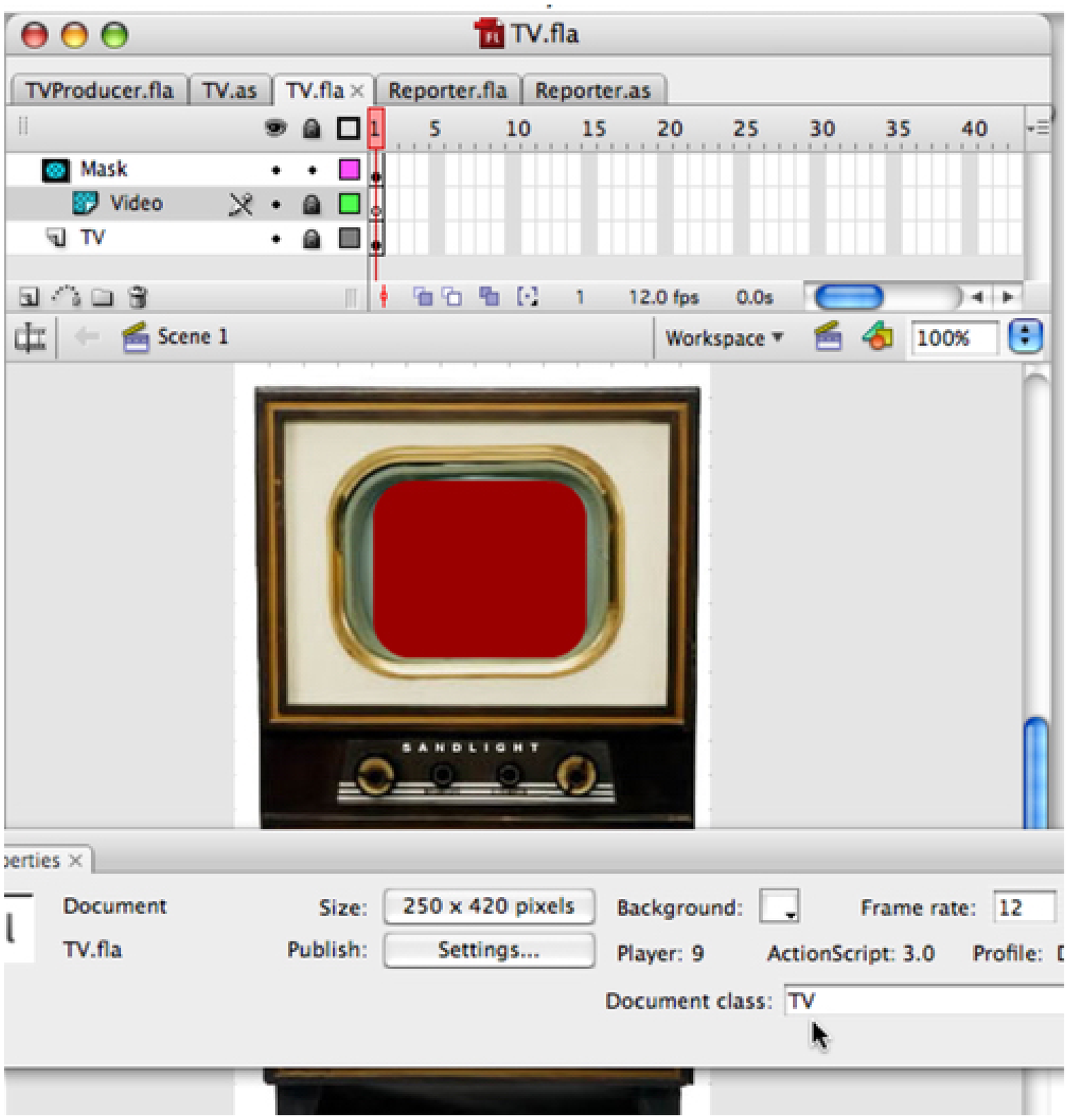
The way the server Stream class works is very cool. Using the `Stream.get( )` method, add a string to the parameter. In this case the parameter contains the string literal "news." Then when the Stream instance (`reporterStream`) plays, another string specifies which named live stream being published is played. If no live stream is currently playing, FMS looks for an FLV or MP3 file with the name in the `Stream.play( )` parameter. In this case, a variable, `reporterNow`, provides the name of the live stream or FLV or MP3 file. So when the Control module changes the contents of `reporterNow` from anchor to the name of a reporter (for example, Joe), the Viewer module (what the viewer sees) changes to play the stream being broadcast by the selected reporter. If no live stream is being played with the name specified in `reporterNow`, FMS looks for a recorded file and will play it instead when the second parameter is set to -2. Figure 9-7 shows how the code interacts.

Figure 9-7. Viewer module client-side code connection to server-side code



In the same way that the basic code for the viewer component is centered on some basic code, the FLA with the visual features of the Viewer module is equally simple. Figure 9-8 shows the layers and Stage objects with their instance names.

Figure 9-8. Setting up Stage for Viewer module



Using Figure 9-8 as guide, follow these steps to create the Viewer module:

1. Open a new Flash document, set the size to 250 x 420, and save it as TV.fla.

2. Create an image of a TV set on the stage. Name the layer TV. In the example in Figure 9-8, I created an old-fashioned one because the screen is relatively small.

3. Add two layers above the TV layer. Name the middle layer Video and the top one Mask.

4. Select the top layer, Right-click (Windows) or Control-click (Mac OS), and select Mask. The Mask layer shows the mask icon and the Video layer (see Figure 9-8). Lock the TV and Video layers.

5. Select the first keyframe in the Mask layer and create a 112 x 93 mask using the Rectangle tool, setting the curves locked at 24. Position the rounded rectangle at x=73, y=62. This gives the video rounded corners. If you made a different-size mask, such as 120 x 100, simply adjust the size and position. Likewise, if you used a different TV set graphic, adjust the position as needed on the Stage and in the code.

6. Open the Property inspector, and in the Document Class text box, type **TV**, as shown in Figure 9-8. Save the FLA file again.

7. Open a new ActionScript file and save it as *TV.as* in the same folder as the other client-side files for this project.

8. In the TV.as file, add the code in Example 9-9 and save the file again.

Example 9-9. TV.as

Code View:

```
package
{
    import flash.net.NetConnection;
    import flash.net.NetStream;
    import flash.events.NetStatusEvent;
    import flash.display.Sprite;
    import flash.media.Video;

    public class TV extends Sprite
    {
          private var nc:NetConnection;
          private var ns:NetStream;
          private var rtmpNow:String;
          private var good:Boolean;
          private var vid:Video;

          private var metaSniffer:Object;

          public function TV ()
          {
               vid=new Video(112,93);
               vid.x=73,vid.y=62;
               addChild (vid);

               nc=new NetConnection();
               rtmpNow="rtmp://192.168.0.11/streamwork/studio";
               nc.connect (rtmpNow);
               nc.addEventListener (NetStatusEvent.NET_STATUS,checkConnect);
          }

          private function checkConnect (e:NetStatusEvent)
          {
               good=e.info.code == "NetConnection.Connect.Success";
               if (good)
               {
                    ns=new NetStream(nc);
                    ns.play ("news");
```

```
                    vid.attachNetStream (ns);

                    metaSniffer=new Object();
                    ns.client=metaSniffer;
                    metaSniffer.onMetaData=getMeta;
               }
               else
               {
                    trace (e.info.code);
               }
          }
          private function getMeta (mdata:Object):void
          {
               //Dummy function
          }
     }
}
```

For the TV to show anything, you need to run the TV Producer module at the same time. Figure 9-9 shows what you can expect to see when all is set up correctly.

Figure 9-9. Selecting the video to view

9.4.4. Reporter Module

The final module in the triad is the reporter. Basically, all the Reporter module does is provide additional live streams. A unique feature of the Reporter module is that the reporter must first sign in with a name that the producer adds as a resource to the reporter's list of selections to stream. (These names must be communicated to the producer, to correctly enter them.) The name that the reporter enters is the name of the live stream the reporter will use. The TV producer is automatically named "anchor," so does not have to enter a name prior to logging on. Figure 9-6 shows what the Reporter module looks like when completed. The top video shows what the local camera sees, and the bottom window is what is currently being streamed. The ActionScript for this module is a combination of the TV producer and the TV itself: the script has the same code for streaming live video as the producer, but cannot select other streams. It streams out only its own and sees only what the producer has currently selected.

Follow these steps to create this final module:

1. Open a new Flash file and save it as *Reporter.fla*.

2. Open the Library module and add Button and TextInput modules.

3. In the Property inspector in the Document Class text box, type **`Reporter`**. Save the file.

4. Open a new ActionScript file and save it as *Reporter.as* in the same folder with the other client-side files for this project.

5. In the *Reporter.as* file, add the code in Example 9-10 and save the file again.

Example 9-10. Reporter.as

Code View:

```
package
{
    import flash.net.NetConnection;
    import flash.net.NetStream;
    import flash.events.NetStatusEvent;
    import flash.events.Event;
    import flash.events.MouseEvent;
    import flash.display.Sprite;
    import flash.media.Video;
    import flash.media.Camera;
    import flash.media.Microphone;
    import fl.controls.TextInput;
    import fl.controls.Button;

    public class Reporter extends Sprite
    {
          private var nc:NetConnection;
          private var ns:NetStream;
          private var studio:NetStream;
          private var rtmpNow:String;
          private var good:Boolean;
```

```
private var vid:Video;
private var vidStream:Video;
private var cam:Camera;
private var mic:Microphone;
private var textInput:TextInput;
private var button:Button;
private var reporter:String;
private var metaSniffer:Object;

public function Reporter ()
{

     textInput=new TextInput();
     textInput.x=265, textInput.y=105;
     addChild (textInput);

     button=new Button();
     button.label="Enter Name";
     button.x=265, button.y=80;
     button.addEventListener (MouseEvent.CLICK,makeConnection);
     addChild (button);

     cam=Camera.getCamera();
     cam.setMode (112,93,10);
     cam.setQuality (100000/8,0);

     vid=new Video(112,93);
     vid.x=145,vid.y=80;
     vid.attachCamera (cam);
     addChild (vid);

     vidStream=new Video(112,93);
     vidStream.x=145,vidStream.y=197;
     addChild (vidStream);

     mic=Microphone.getMicrophone();
     mic.rate=11;

}
private function makeConnection (e:MouseEvent)
{
     nc=new NetConnection();
     rtmpNow="rtmp://192.168.0.11/streamwork/studio";
     nc.connect (rtmpNow);
     nc.addEventListener (NetStatusEvent.NET_STATUS,checkConnect);
}

private function checkConnect (e:NetStatusEvent)
{
     good=e.info.code == "NetConnection.Connect.Success";
     if (good)
     {
          reporter=textInput.text;
          studio=new NetStream(nc);
          studio.attachAudio (mic);
          studio.attachCamera (cam);
          studio.publish (reporter);

          ns=new NetStream(nc);
```

```
                    ns.play ("news");
                    vidStream.attachNetStream (ns);

                    metaSniffer=new Object();
                    ns.client=metaSniffer;
                    metaSniffer.onMetaData=getMeta;
               }
               else
               {
                    trace (e.info.code);
               }
          }
          private function getMeta (mdata:Object):void
          {
               //Dummy function
          }
     }
}
```

The key element in this module is the "studio" `NetStream` instance. It publishes a string variable, `reporter`. The `reporter` variable contains the name the reporter uses to first connect. The name can be anything, including first and last names, but as long as it's unique and included in the controller module's List component, it will work. To work with the application, simply have someone serve as anchor and others act as reporters and viewers. If you want to further automate the process, you can store the reporter names on the server and have them automatically loaded into the List component in the TV Producer module. Also, gather up any FLV or MP3 files you want and put them into the appropriate folder (streamwork streams studio FLV and MP3 files). You can now put on a worldwide broadcast.

9.5. Server-Side NetStream Class

Flash Media Interactive Server 3 has a new class, NetStream. You can set up your application so that more than a single FMSIS3 can be used with a single stream. You can use one server to create a NetStream to another server and use both servers to balance the load as more clients connect. The following shows the general process:

- Client generates `NetStream` → Server1 gets client `NetStream` and serves it.
- Server1 generates server `NetStream` to Server2 Server2 serves stream.

With a single stream originating from a client, once the load on the first server gets filled up with clients, subsequent clients can be sent to another server. Any number of strategies can be used to allocate server resources. The following application boils down the features to a minimal configuration so that you can see how everything works.

The first module is the server-side script. You cannot use RTMPE, RTMPS, or RTMPTE when making the connection to the second server. In this application example, all components use the basic RTMP. Example 9-11 shows the server-side NetStream class instantiated along with a new `NetConnection`, but using a single named stream originating in the client.

Example 9-11. SerSideNS.asc

```
var nc;
var ns;
application.onConnect = function(client)
{
    application.acceptConnection(client);
}

application.onPublish = function(client, hotStream)
{

    if (application.name == "SerSideNS/BasicAudience")
    {
          trace("OK:"+application.name);
          nc = new NetConnection();
          nc.connect( "rtmp://192.168.0.3/SerSideNS/OverflowAudience" );
          ns = new NetStream(nc);
          trace(hotStream.name+ " is up.");
          ns.setBufferTime(2);
          ns.attach(hotStream);
          ns.publish( hotStream.name, "live" );
    }
}
```

The `onPublish` function gets the client and name of the published stream-bigShow in this example. Then it sets up a `NetConnection` and `NetStream`, very much as you'd do with a client-side script that generates a stream. The `NetConnection.connect()` method is also just similar to what you'd use in a client-side script. But

keep in mind that this script is from a server connecting to another server. The example uses two different IP addresses to illustrate this feature. Then the stream name is attached to a NetStream object and published live. That is all there is to it. Remember that all server-side code is ActionScript 1.0, and so there's no strong typing.

9.5.1. Publishing the Client-Side Stream

The client side is set up with one module publishing a stream and two modules playing it. One of the modules plays the stream from one server, and the other plays it from another. Example 9-12 shows that publishing the stream is no different from any other client-side publication.

Example 9-12. SplitAudience.as

Code View:

```
package
{
  import flash.display.Sprite;
  import flash.net.NetConnection;
  import flash.events.NetStatusEvent;
  import flash.net.NetStream;
  import flash.media.Video;
  import flash.media.Microphone;
  import flash.media.Camera;

  public class SplitAudience extends Sprite
  {
        private var nc:NetConnection;
        private var ns:NetStream;
        private var vid:Video;
        private var cam:Camera;
        private var mic:Microphone;
        private var good:Boolean;

        public function SplitAudience ()
        {
             nc=new NetConnection ;
             nc.addEventListener (NetStatusEvent.NET_STATUS,checkConnection);
             nc.connect ("rtmp://192.168.0.11/SerSideNS/BasicAudience");
        }
        private function checkConnection (e:NetStatusEvent):void
        {
             good=e.info.code == "NetConnection.Connect.Success";
             if (good)
             {
                   ns=new NetStream(nc);
                   cam=Camera.getCamera();
                   cam.setMode(220,180,24);
                   cam.setQuality(0,90);
                   mic=Microphone.getMicrophone();
                   mic.rate=22;
                   vid=new Video(220,180);
                   vid.attachCamera (cam);
                   ns.attachCamera (cam);
                   ns.attachAudio (mic);
                   addChild(vid);
                   vid.x=100,vid.y=100;
                   ns.publish ("bigShow","live");
             }
        }
```

```
    }
}
```

Any broadcast type of application could be used with the server-side `NetStream` instance because most of the work is done in the server-side code.

9.5.2. Playing the Stream from Multiple Servers

Example 9-13 and Example 9-14 are identical except for their names and RTMP values. Example 9-13 has the value:

```
rtmpNow="rtmp://192.168.0.11/SerSideNS/BasicAudience";
```

Example 9-14 is only slightly different:

```
rtmpNow="rtmp://192.168.0.3/SerSideNS/OverflowAudience";
```

Only a single server has the *SerSideNS.asc* (or main.asc) file. Its job is to open the second connection to the second server. As soon as the Publishing module (SplitAudience) starts, both servers indicate a client present in the Live Log view of the Administration console. One client is the connection between the Publishing module and the first server, and the second connection is that between the first and second servers.

Example 9-13 lists the playing module. It connects to the first server where the server-side script resides.

Example 9-13. BasicAudience.as

Code View:

```
package
{
  import flash.display.Sprite;
  import flash.net.NetConnection;
  import flash.net.NetStream;
  import flash.events.NetStatusEvent;
  import flash.events.Event;
  import flash.media.Video;

  public class BasicAudience extends Sprite
  {
        private var nc:NetConnection;
        private var ns:NetStream;
        private var rtmpNow:String;
        private var good:Boolean;
        private var vid:Video;

        function BasicAudience ()
        {
             nc=new NetConnection();
             nc.addEventListener (NetStatusEvent.NET_STATUS,checkConnect);
             rtmpNow="rtmp://192.168.0.11/SerSideNS/BasicAudience";
```

```
            nc.connect (rtmpNow);
            addMedia ();
        }

        private function addMedia ():void
        {
            vid=new Video(220,180);
            addChild (vid);
            vid.x=100;
            vid.y=50;
        }

        private function checkConnect (e:NetStatusEvent):void
        {
            good=(e.info.code=="NetConnection.Connect.Success");
            if (good)
            {
                ns = new NetStream(nc);
                if (ns)
                {
                    ns.play ("bigShow");
                    vid.attachNetStream (ns);
                }
            }
        }
    }
}
```

Be sure to start the SplitAudience module before you test the BasicAudience module. As soon as you launch the BasicAudience module, you will see that two clients are recognized in the Administration console, as shown in Figure 9-10.

Figure 9-10. Primary server shown to have two connections

One connection is the Publishing module and the other is the first playing module.

The OverflowAudience class is identical to the BasicAudience class except for the name and RTMP address. You

can use a FMS2 server as a target server, and if you do, uncomment the lines in Example 9-14:

```
//import flash.net.ObjectEncoding;
//NetConnection.defaultObjectEncoding = flash.net.ObjectEncoding.AMF0;
    var rtmpNow:String="rtmp://poobah.mwd.hartford.edu/appName";
```

The server-side code must all be stored on a FMS3 server, but when creating new server-side `NetConnection` and `NetStream` instances, the target established in the RTMP address can be to a FMS2 server.

Example 9-14 shows the entire listing.

Example 9-14. OverflowAudience.as

Code View:

```
package
{
  import flash.display.Sprite;
  import flash.net.NetConnection;
  import flash.net.NetStream;
  import flash.events.NetStatusEvent;
  import flash.events.Event;
  import flash.media.Video;
//import flash.net.ObjectEncoding;

  public class OverflowAudience extends Sprite
  {
        private var nc:NetConnection;
        private var ns:NetStream;
        private var rtmpNow:String;
        private var good:Boolean;
        private var vid:Video;

        function OverflowAudience ()
        {
             // NetConnection.defaultObjectEncoding =      flash.net.ObjectEncoding.
AMF0;

    var rtmpNow:String="rtmp://poobah.mwd.hartford.edu/appName";

             nc=new NetConnection();
             nc.addEventListener (NetStatusEvent.NET_STATUS,checkConnect);
             rtmpNow="rtmp://192.168.0.3/SerSideNS/OverflowAudience";
             nc.connect (rtmpNow);
             addMedia ();
        }

        private function addMedia ():void
        {
             vid=new Video(220,180);
             addChild (vid);
             vid.x=100;
             vid.y=50;
        }

        private function checkConnect (e:NetStatusEvent):void
        {
```

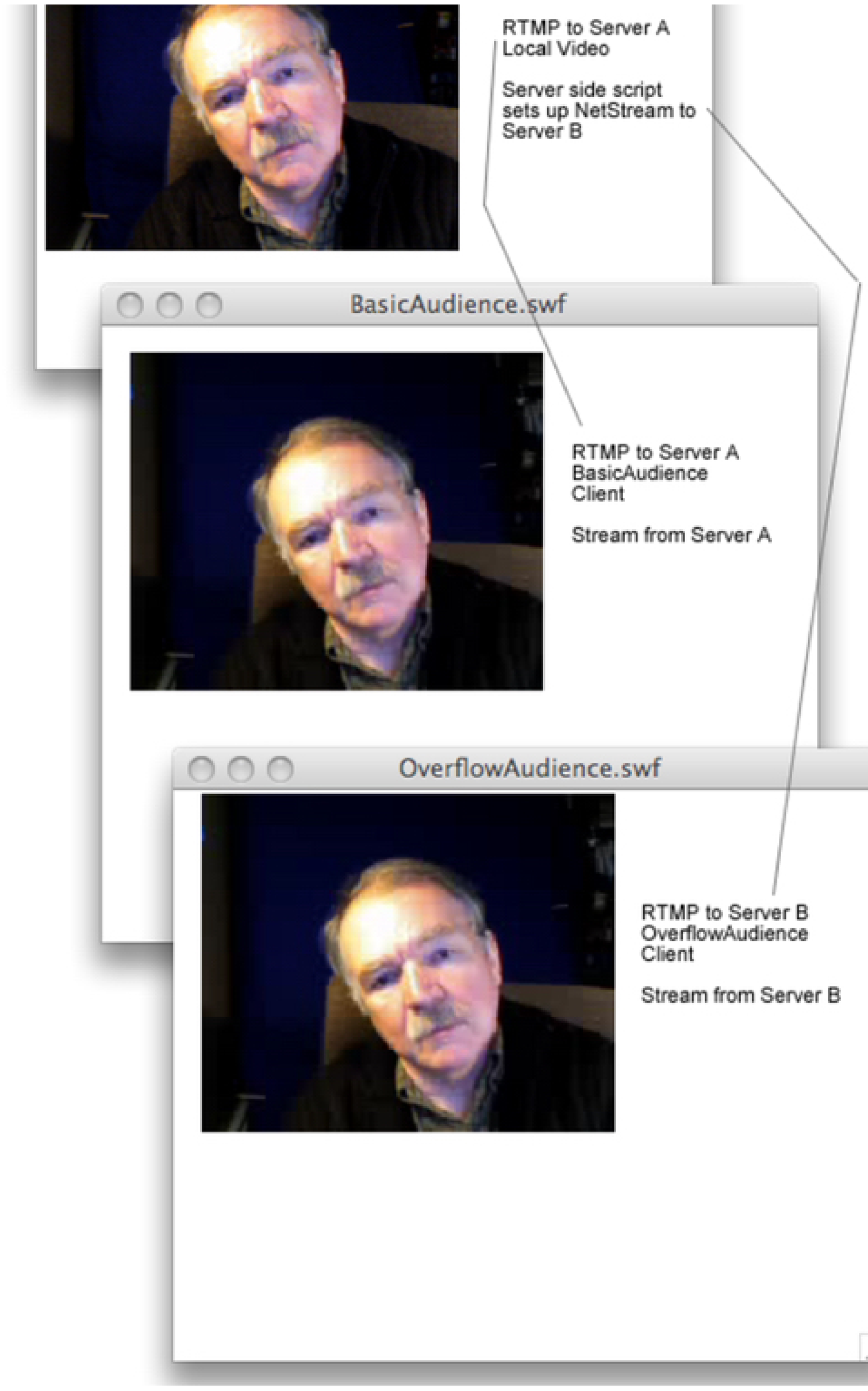
```
            good=(e.info.code=="NetConnection.Connect.Success");
            if (good)
            {
                  ns = new NetStream(nc);
                  if (ns)
                  {
                       ns.play ("bigShow");
                       vid.attachNetStream (ns);
                  }
            }
      }
  }
}
```

When you start playing the second module, you will have to look at a second Administration console connected to the secondary server where two clients are also shown. One client is still the primary server's connection, and the second is the OverflowAudience module. Its Live Log looks different than the first shown in Figure 9-11.

Figure 9-11. Secondary server showing two connections

Even though the two different Administration consoles display different information, they both show the same live stream (`bigShow`). Figure 9-12 shows what you can expect to see and the relationship between the different parts.

Figure 9-12. Local and streamed output



As you can see, all of the views are identical. The Publishing module has a direct link to the local video and audio. The first playing module is receiving its stream from the primary server and the second from the

secondary server. They all look pretty much the same, and what little latency was noticed is not greater for one server or the other.

While this example has been very simple, the addition of a server-side `NetStream` instance opens the door for multiplying the power of FMSIS3 by easing the preparation of a software-based load balancer. Rather than having to rely on a hardware load balancer, such as a special router, the code for working out a software solution is made possible with the new server-side `NetStream`. Broadcasting applications, such as those discussed in this chapter, can be channeled to wider audiences, thanks to more servers to handle the load. What's more, if you have any FMS2 servers available, they can be used to help balance the load as well as long as one FMS3 server is available to generate a server-side `NetStream` instance.

Chapter 10. Bringing in Data and Working with Configuration Files

10.1. Cue Points, Metadata, and Stream Completion

Certain kinds of data are built into FLV files for use by developers. This book features many examples of metadata, either in use or handled to avoid errors. One description of *metadata* is definitional data that provides information about, or documentation of, other data managed within an application or environment. Examples of metadata in this book illustrate that something is going on with FLV files that requires coding to handle the metadata. The metadata arising from a FLV file is delivered through a net stream. Using the `NetStream.client` property with ActionScript 3.0, you can create an object and assign it to `NetStream.client`. Then using that object, you can use the `NetStream` event handlers to pull out cue point and metadata information. Use that information for controls or additional information. For example, if you are setting up a presentation application, you can use cue points to send text to an output window to emphasize content in the video. Likewise, you can use the duration property of metadata for scaling a slider or any other use. Of course, you can use the object to trigger functions related to other `NetStream` events. For instance, the `NetStream.onPlayStatus` is sent even when a stream has been completely played. Here is some of the information contained within cue points and metadata:

Cue Point Properties

- Name
- Parameters
- Time
- Type

Typical Metadata Properties

- Duration
- Creation date
- Data rates
- Width
- Height

To set up and capture different kinds of data, the following simple routine works well, and makes it easy to deal with scope:

```
eventObject=new Object();
ns.client=eventObject;
eventObject.onMetaData=metaDataFunction;
eventObject.onCuePoint= cuePointFunction;
eventObject.onPlayStatus= endOfStreamFunction;
```

One reason why I like to use an object instead of creating a separate class to handle metadata and cue points is that I find it easier to encapsulate the object using the `private` modifier in declaring the object. Likewise, I can encapsulate the different functions that are called when the event is dispatched. Setting a separate class to handle these kinds of events has the advantage of being able to reuse it; if you prefer that method, you may want to use it instead.

Whenever you create an FLV file, some information is automatically stored as the metadata using the Adobe Flash CS3 Video Encoder or some other form of encoding, including the `NetStream.publish` record or append methods FMS.

Example 10-1 shows one way to set up and use cue points and metadata in an application. You will need an FLV file saved as CuePoint.flv in the FMS path applications/cuepoint/streams/cueball/before trying it out. In an FLA file, place an instance of a button (noncomponent) with the class name Btn, and a TextArea component in the Library with the class name TextArea.

Example 10-1. CueFMS.as

Code View:

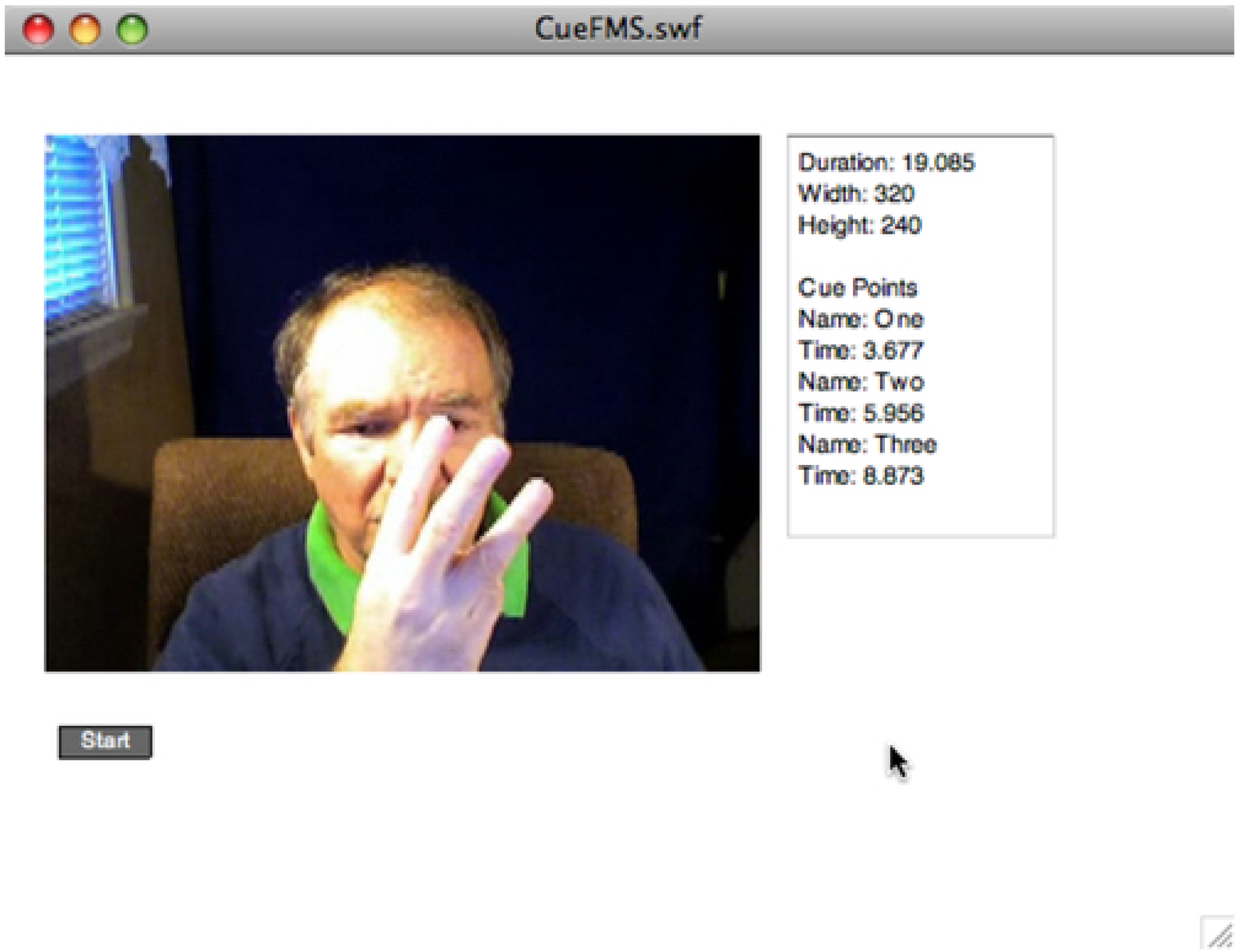
```
package
{
  import flash.media.Video;
  import flash.net.NetConnection;
  import flash.net.NetStream;
  import flash.display.Sprite;
  import flash.events.MouseEvent;
  import flash.events.NetStatusEvent;
  import fl.controls.TextArea;

  public class CueFMS extends Sprite
  {
```

```
private var nc:NetConnection;
private var ns:NetStream;
private var vid:Video;
private var metaCue:Object;
private var btn:Btn;
private var rtmpNow:String;
private var good:Boolean;
private var metaDur:Number;
private var metaWidth:Number;
private var metaHeight:Number;
private var textArea:TextArea;

public function CueFMS ()
{
     nc=new NetConnection ;
     rtmpNow="rtmp://192.168.0.11/cuepoint/cueball";

     nc.connect (rtmpNow);
     nc.addEventListener (NetStatusEvent.NET_STATUS,checkConnect);
}
private function checkConnect (e:NetStatusEvent)
{
     good=e.info.code == "NetConnection.Connect.Success";
     if (good)
     {
          ns=new NetStream(nc);
          vid=new Video;
          vid.width=320;
          vid.height=240;
          vid.x=18,vid.y=36;
          addChild (vid);

          textArea=new TextArea;
          textArea.x=350, textArea.y=36;
          textArea.width=120, textArea.height=180;
          addChild(textArea);

          btn=new Btn();
          btn.x=24,btn.y=300;
          addChild (btn);
          btn.addEventListener (MouseEvent.CLICK,startClass);

          //Capture metadata and cuedata from FLV file
          metaCue=new Object();
          ns.client=metaCue;
          metaCue.onMetaData=getMeta;
          metaCue.onCuePoint=getAcue;
          metaCue.onPlayStatus=clearVideo;
     }
}

private function startClass (e:MouseEvent)
{
     ns.play ("CuePoint");
     vid.attachNetStream (ns);
}

private function getMeta (mdata:Object):void
{
```

```
            //All metadata can be captured here.
            metaDur=mdata.duration;
            metaWidth=mdata.width;
            metaHeight=mdata.height;
            textArea.text="Duration: "+metaDur;
            textArea.appendText("\nWidth: "+metaWidth);
            textArea.appendText("\nHeight: "+metaHeight);
            textArea.appendText("\n\nCue Points");
        }

        private function getAcue (cue:Object)
        {
            textArea.appendText("\nName: "+cue.name+"\nTime: "+cue.time);
        }

        private function clearVideo (end:Object)
        {
            vid.clear ();
        }
    }
}
```

Figure 10-1 shows the output in the middle of the video. Using hand signals, I placed a cue point where each number appeared, naming the cue points after the number. In Figure 10-1, the name of the current cue point is "Three"; the view shows three fingers. Simultaneously with the cue point name, the time of the cue point is also available, and it too appears. Figure 10-1 shows that when then cue point "Three" appears, the three fingers emerge 8.873 seconds into the video.

Figure 10-1. Cue Point and metadata appear in a TextArea component



Besides the cue points, the metadata appear. However, the metadata is available when the FLV file begins to play, and shows that the video is 19.085 seconds long and has a native width of 320 and height of 240. (You can set the video to a different height and width, but the native dimensions generally provide a better video.)

Example 10-1 includes a third event, `NetStream.onPlayStatus`. Because the event only dispatches when the stream has completely finished, it's useful for clearing the screen or initiating some other end-of-play action. In this case, the only thing behind the video is a green rectangle the size of video. More importantly, you can use the event to start a new video or another kind of action related to the application.

Chapter 10. Bringing in Data and Working with Configuration Files

10.1. Cue Points, Metadata, and Stream Completion

Certain kinds of data are built into FLV files for use by developers. This book features many examples of metadata, either in use or handled to avoid errors. One description of *metadata* is definitional data that provides information about, or documentation of, other data managed within an application or environment. Examples of metadata in this book illustrate that something is going on with FLV files that requires coding to handle the metadata. The metadata arising from a FLV file is delivered through a net stream. Using the `NetStream.client` property with ActionScript 3.0, you can create an object and assign it to `NetStream.client`. Then using that object, you can use the `NetStream` event handlers to pull out cue point and metadata information. Use that information for controls or additional information. For example, if you are setting up a presentation application, you can use cue points to send text to an output window to emphasize content in the video. Likewise, you can use the duration property of metadata for scaling a slider or any other use. Of course, you can use the object to trigger functions related to other `NetStream` events. For instance, the `NetStream.onPlayStatus` is sent even when a stream has been completely played. Here is some of the information contained within cue points and metadata:

Cue Point Properties

- Name
- Parameters
- Time
- Type

Typical Metadata Properties

- Duration
- Creation date
- Data rates
- Width
- Height

To set up and capture different kinds of data, the following simple routine works well, and makes it easy to deal with scope:

```
eventObject=new Object();
ns.client=eventObject;
eventObject.onMetaData=metaDataFunction;
eventObject.onCuePoint= cuePointFunction;
eventObject.onPlayStatus= endOfStreamFunction;
```

One reason why I like to use an object instead of creating a separate class to handle metadata and cue points is that I find it easier to encapsulate the object using the `private` modifier in declaring the object. Likewise, I can encapsulate the different functions that are called when the event is dispatched. Setting a separate class to handle these kinds of events has the advantage of being able to reuse it; if you prefer that method, you may want to use it instead.

Whenever you create an FLV file, some information is automatically stored as the metadata using the Adobe Flash CS3 Video Encoder or some other form of encoding, including the `NetStream.publish` record or append methods FMS.

Example 10-1 shows one way to set up and use cue points and metadata in an application. You will need an FLV file saved as CuePoint.flv in the FMS path applications/cuepoint/streams/cueball/before trying it out. In an FLA file, place an instance of a button (noncomponent) with the class name Btn, and a TextArea component in the Library with the class name TextArea.

Example 10-1. CueFMS.as

Code View:

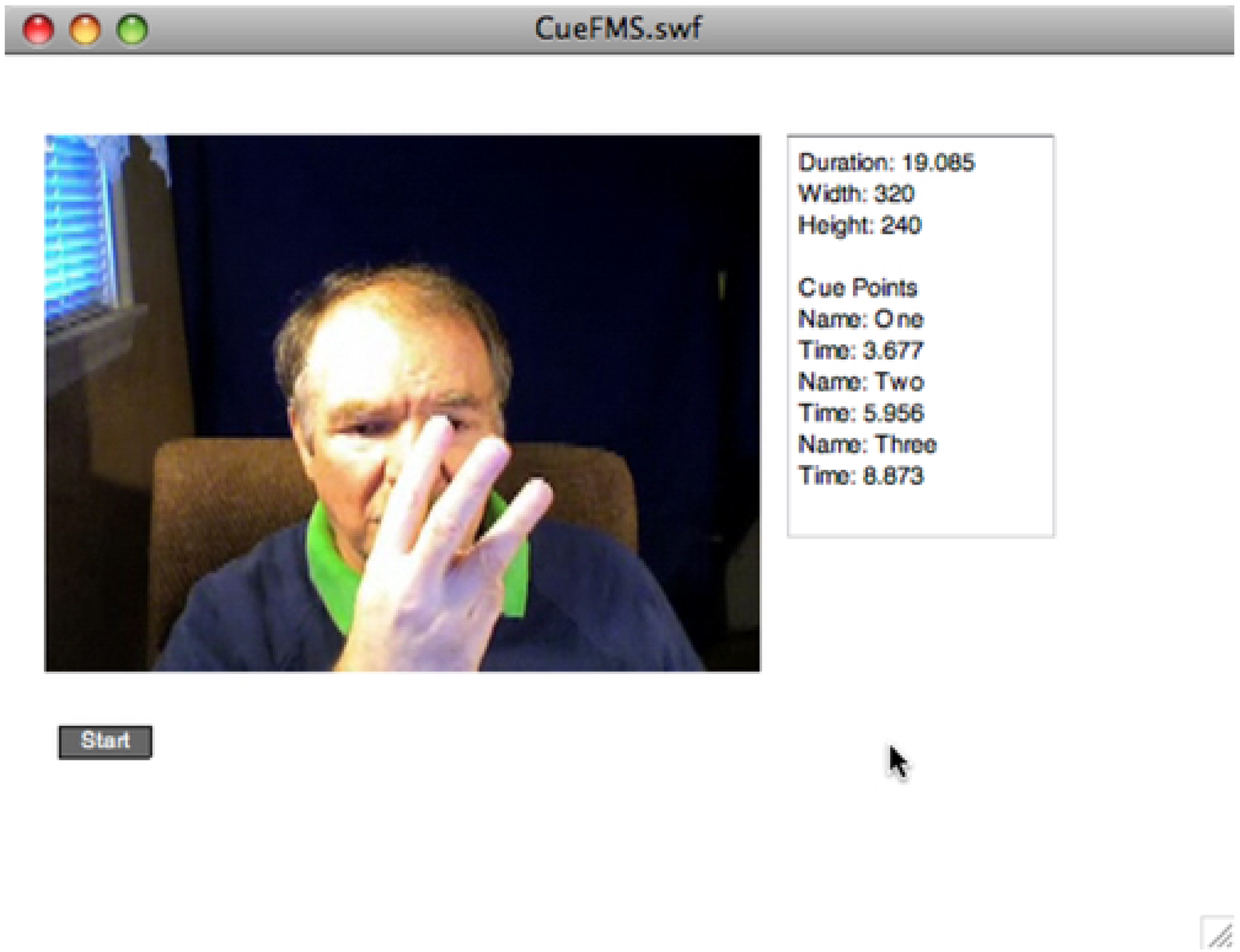
```
package
{
  import flash.media.Video;
  import flash.net.NetConnection;
  import flash.net.NetStream;
  import flash.display.Sprite;
  import flash.events.MouseEvent;
  import flash.events.NetStatusEvent;
  import fl.controls.TextArea;

  public class CueFMS extends Sprite
  {
```

```
private var nc:NetConnection;
private var ns:NetStream;
private var vid:Video;
private var metaCue:Object;
private var btn:Btn;
private var rtmpNow:String;
private var good:Boolean;
private var metaDur:Number;
private var metaWidth:Number;
private var metaHeight:Number;
private var textArea:TextArea;

public function CueFMS ()
{
     nc=new NetConnection ;
     rtmpNow="rtmp://192.168.0.11/cuepoint/cueball";

     nc.connect (rtmpNow);
     nc.addEventListener (NetStatusEvent.NET_STATUS,checkConnect);
}
private function checkConnect (e:NetStatusEvent)
{
     good=e.info.code == "NetConnection.Connect.Success";
     if (good)
     {
          ns=new NetStream(nc);
          vid=new Video;
          vid.width=320;
          vid.height=240;
          vid.x=18,vid.y=36;
          addChild (vid);

          textArea=new TextArea;
          textArea.x=350, textArea.y=36;
          textArea.width=120, textArea.height=180;
          addChild(textArea);

          btn=new Btn();
          btn.x=24,btn.y=300;
          addChild (btn);
          btn.addEventListener (MouseEvent.CLICK,startClass);

          //Capture metadata and cuedata from FLV file
          metaCue=new Object();
          ns.client=metaCue;
          metaCue.onMetaData=getMeta;
          metaCue.onCuePoint=getAcue;
          metaCue.onPlayStatus=clearVideo;
     }
}

private function startClass (e:MouseEvent)
{
     ns.play ("CuePoint");
     vid.attachNetStream (ns);
}

private function getMeta (mdata:Object):void
{
```

```
            //All metadata can be captured here.
            metaDur=mdata.duration;
            metaWidth=mdata.width;
            metaHeight=mdata.height;
            textArea.text="Duration: "+metaDur;
            textArea.appendText("\nWidth: "+metaWidth);
            textArea.appendText("\nHeight: "+metaHeight);
            textArea.appendText("\n\nCue Points");
        }

        private function getAcue (cue:Object)
        {
            textArea.appendText("\nName: "+cue.name+"\nTime: "+cue.time);
        }

        private function clearVideo (end:Object)
        {
            vid.clear ();
        }
    }
}
```

Figure 10-1 shows the output in the middle of the video. Using hand signals, I placed a cue point where each number appeared, naming the cue points after the number. In Figure 10-1, the name of the current cue point is "Three"; the view shows three fingers. Simultaneously with the cue point name, the time of the cue point is also available, and it too appears. Figure 10-1 shows that when then cue point "Three" appears, the three fingers emerge 8.873 seconds into the video.

Figure 10-1. Cue Point and metadata appear in a TextArea component



Besides the cue points, the metadata appear. However, the metadata is available when the FLV file begins to play, and shows that the video is 19.085 seconds long and has a native width of 320 and height of 240. (You can set the video to a different height and width, but the native dimensions generally provide a better video.)

Example 10-1 includes a third event, `NetStream.onPlayStatus`. Because the event only dispatches when the stream has completely finished, it's useful for clearing the screen or initiating some other end-of-play action. In this case, the only thing behind the video is a green rectangle the size of video. More importantly, you can use the event to start a new video or another kind of action related to the application.

10.2. Server-Side LoadVars Class

If you have used the LoadVars class in any of your ActionScript 2.0 client-side applications, you know how useful and powerful it is for loading and sending variables to middleware such as PHP and ColdFusion. Likewise, `LoadVars` instances can load variables from text files using the same name and value pairs for sending and loading data to and from ActionScript 2.0. Accessing data from a URL in client-side applications can now be done on the server side. The LoadVars class represents a double anomaly in server-side ActionScript. At its base, server-side ActionScript is actually ActionScript 1.0, but LoadVars is an ActionScript 2.0 class. In ActionScript 3.0, the Loader class has replaced the LoadVars class.

10.2.1. FMS 3.0 Server-Side LoadVars Class Properties

Often, server- and client-side classes have major differences in their properties, methods, or event handlers. All of the properties, methods, and events are identical to the client-side counterpart in LoadVars ActionScript 2.0. Table 10-1 summarizes all of the classes' features.

Table 10-1. LoadVars methods, properties, and event handlers

Method	Use/Function
addRequestHeader()	Places/changes *http* request header sent with POST actions. For example: SOAPAction ContentType.
decode()	Converts URL-encoded query string to LoadVars properties.
getBytesLoaded()	Gets current bytes loaded from load() or sendAndLoad method() use.
getBytesTotal()	Gets total bytes loaded from load()or sendAndLoad method() use.
load()	Get variables from called URL parsing name/value pairs into current LoadVars instance.
send()	Sends variables and their values in a LoadVars instance to target URL.
sendAndLoad()	Uploads and downloads variables to target URL and LoadVars instance.
toString()	Provides string with all variables in LoadVars instance, in MIME encoding.
Properties	Characteristics
contentType	Provides the MIME type sent with a call using send() or sendAndLoad( ).
loaded	Boolean indicating when load() or sendAndLoad() action is complete
Event Handler	Action to Activate Handler
onData	Data download has completed (or an error has occurred while downloading data).
onHTTPStatus	FMS receives http status code.
onLoad	Either load() or sendAndLoad() operation is complete.

If you know how to use `LoadVars()` from using standard ActionScript 2.0, you shouldn't have any problem using server-side `LoadVars()`. However, be aware that ActionScript 3.0 uses the Loader class, which is very different in the way it handles loading data.

10.2.2. Passing Name-Value Pairs

The key to using LoadVars( ), is understanding the name-value pair structure in UTF-8 formatted code. Essentially, the name-value pair is a text-based format understood by Flash when passed using LoadVars( ). For example, an email variable with an email value might look like this:

```
cusEmail=sandlight@comcast.net
```

No quotation marks surround string variables and no spaces are used between the variable name and its value, demarcated by the equal sign (=). Variables are separated by ampersands (&) with no spaces between the previous value and the new variable name. For example:

```
business=Sandlight&cusEmail=sandlight@comcast.net
```

delineates two variables and their associated values. Broken down, they would appear as:

Variable Name	Variable Value
business	Sandlight
cusEmail	sandlight@comcast.net

Given that format, when name-value pairs are brought into a LoadVars( ) instance, the variables are properties and the values of the variables are assigned to the similarly named property. The following code shows what the LoadVars( ) instance would look like using the name-value pairs in the example:

```
var myData_lv = new LoadVars( );
myData_lv.business;
myData_lv.cusEmail;
```

To load data from a URL, the load() method extracts the name/value pairs and places the values in the instance properties. The following line loads the text file:

```
myData_lv.load("http://www.myplace.com/business/busMail.txt");
```

The trace output goes into the FMS application console; you will need to open it to see the output. Because this code is on the server, you cannot output it directly to a text field or component. You can either use a call to the client side to have a client-side script output the data to some readable output on the page, or simply use the output to trigger a server-side function useable by server-side code. If you're using your own computer for both server and development platform, you can use localhost and put your text file in the folder with the server-side script. However, you would be better served if you placed the text file on another server. The security for such placement would be very high because finding the location of the URL through the FMS server code would be extremely difficult.

10.3. Minimalist Example Using Server-Side **LoadVars( )**

To give an example of how you might use the server-side `LoadVars( )`, the following application launches a function that uses `LoadVars()` to open a remotely stored set of values for setting the bandwidth. It uses a `Client` function with the `LoadVars()`operation to open a text file with the name-value pairs for setting bandwidth limits that might be used with a phone modem, a DSL line, and a cable connection. These values will be saved in a file named bw.txt and placed on your Web server (make sure you specify the URL in the server-side code). Like the rest of the application, the code is minimal:

```
modem=7000&dsl=9375&cable=12800
```

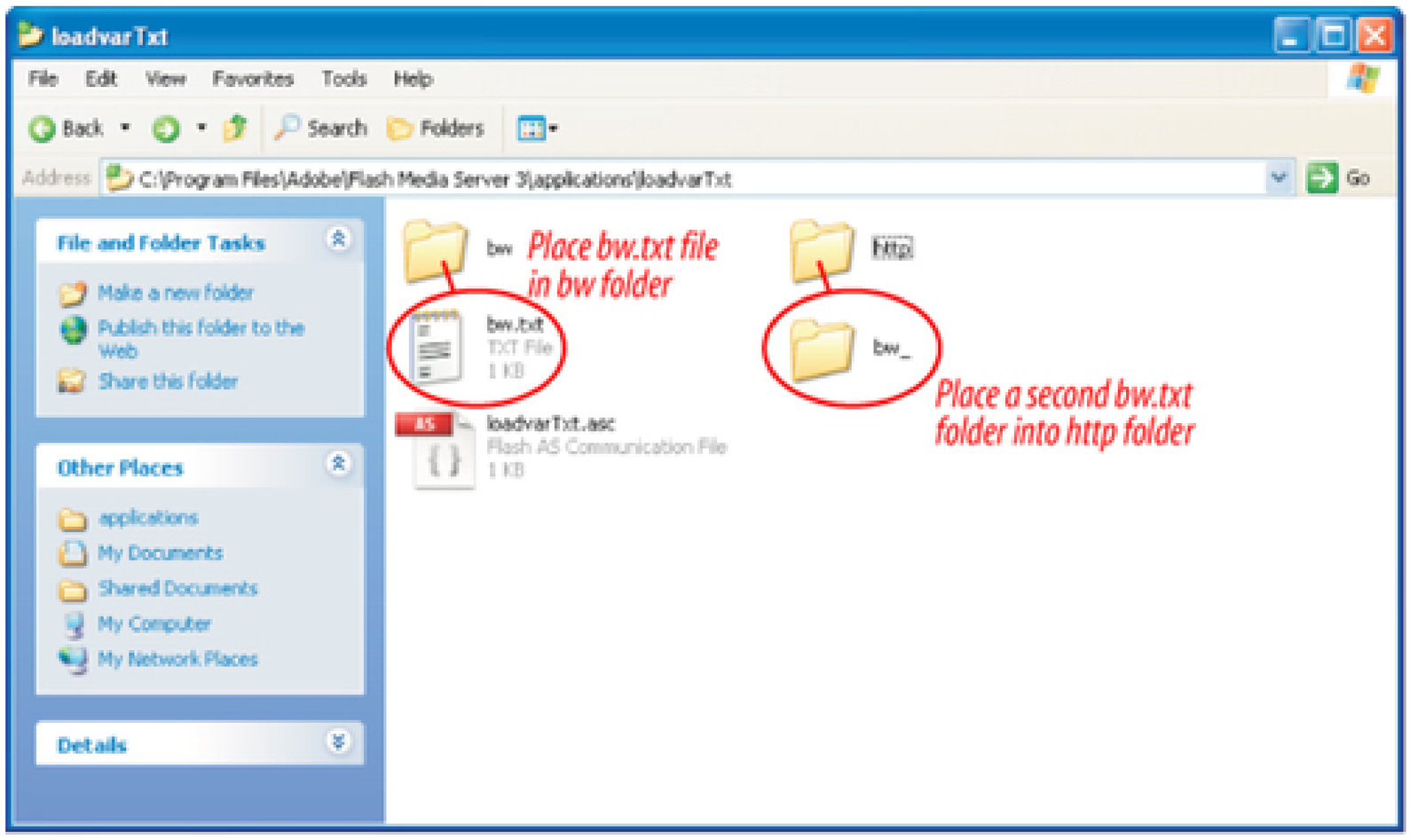
Breaking it down, you can see the following name-value pairs: The trick is to assign the appropriate value when one of the variable names is passed from the client-side code. The `bw.txt` file will be placed in a folder named `bw` on the server.

10.3.1. Server Side **LoadVars**

Setting up the server-side folders using `LoadVars` in a server-side script are a bit unusual. Follow these steps and refer to Figure 10-2 to set everything correctly:

1. In the server side applications folder, create a folder named loadvarTxt.
2. In the LoadVarTxt folder, place the server side file, *loadvarTxt.asc*.
3. Create two more folders, one named *bw* and the other *http*, and place them both in the *loadvarTxt* folder.
4. In the *bw* folder, place a text file with the following contents:
5. modem=7000&dsl=9375&cable=12800
6. Save the file as bw.txt.
7. Open the *http* folder and place an empty folder named *bw*in it. This is a second folder with the name *bw*.

Figure 10-2. Server-side folder arrangement for loading text file



To begin, the Example 10-2 shows the key function `client.loadit`, used to load the data and make the appropriate bandwidth setting. Because this is a server-side script, be sure to place it in a folder named *loadvarTxt*, and save it with an .asc file extension.

Example 10-2. loadvarTxt.asc

Code View:

```
//Server Side LoadVars
application.onAppStart = function() {
  trace(this.name+" has been re-loaded");
};
application.onConnect = function(client) {
  this.acceptConnection(client);
  client.loadit = function(bwset) {
        this.setter = bwset;
        var loadBW = new LoadVars();
        loadBW.onLoad = function(success) {
             if (success) {
                  trace(this.contentType);
                  switch (client.setter) {
                  case 'modem' :
                       c2s = s2c=Number(this.modem);
                       trace(c2s);
                       client.setBandwidthLimit(s2c, c2s);
                       break;
                  case 'dsl' :
                       c2s = s2c=Number(this.dsl);
                       trace(c2s);
                       client.setBandwidthLimit(s2c, c2s);
                       break;
                  case 'cable' :
                       c2s = s2c=Number(this.cable);
                       trace(c2s);
                       client.setBandwidthLimit(s2c, c2s);
```

```
                        break;
                    }
                } else {
                    trace("problem");
                }
            };
            loadBW.load("http://localhost/bw/bw.txt");
    };
};
```

In looking at the code, there's nothing fancy about it. The bandwidth setting is synchronous-both client-to-server and server-to-client settings use the same value. The trace statements display both the bandwidth setting values and the MIME types to the FMS3 console, as shown in Figure 10-3.

10.3.2. Client Side Bandwidth Selector

On the client-side, the code is equally simple. All it requires is some way of calling the server-side `client.loadit()` function and passing a parameter that will tell it which stored name to use in getting a value. Other than a List component with the instance name `list` and a little connection light class, `ConnectLight.as`, nothing else is required. First, create the connection light as shown in Example 10-3. It needs to be done first because the client-side script that calls the server-side script uses it.

Example 10-3. ConnectLight.as

```
package
{
  import flash.display.Shape;
  import flash.display.Sprite;

  public class ConnectLight extends Sprite
  {
        private var conLite:Shape;

        public function ConnectLight(clr:uint)
        {
            conLite=new Shape();
            conLite.graphics.beginFill(clr);
            conLite.graphics.lineStyle(.5,0x000000);
            conLite.graphics.drawCircle(0,0,5);
            addChild(conLite);
        }
  }
}
```

The ConnectLight.as file needs to be stored in the same folder as Example 10-4, LoadVarTxt.as, the class used to fire the server-side function that sets the bandwidth limits. In setting the bandwidth, the server-side script uses data stored in a text file for a permanent reference. If the bandwidth limits change, all you need to do is change the contents of the text file storing the bandwidth information.

The main role of the client-side script is to make it simple to select a bandwidth level; to do that, the Example 10-4 script uses a List component. The user can choose from modem, DSL, or cable speeds.

Example 10-4. LoadVarTxt.as

Code View:

```
package
{
  import fl.controls.List;
  import fl.controls.Label;
  import flash.net.NetConnection;
  import fl.data.DataProvider;
  import flash.events.NetStatusEvent;
  import flash.events.Event;
  import flash.display.Sprite;

  public class LoadBW extends Sprite
  {
        private var rtmpNow:String;
        private var nc:NetConnection;
        private var list:List;
        private var dp:DataProvider;
        private var listLabel:Label;
        private var bw:String;
        private var good:Boolean;
        private var connectBtn:ConnectBtn;
        private var connectVal:uint;
        private var checker:Array;

        public function LoadBW ()
        {
             dp=new DataProvider();
             dp.addItem ({label:"Bandwidth"});
             dp.addItem ({label:"modem"});
             dp.addItem ({label:"dsl"});
             dp.addItem ({label:"cable"});

             list=new List();
             list.move (250,120);
             list.addEventListener (Event.CHANGE, setBW);
             list.labelField="label";
             list.dataProvider=dp;
             list.selectedIndex=0;
             addChild (list);

             rtmpNow = "rtmp://192.168.0.11/loadvarTxt/bw";
             nc = new NetConnection ();
             nc.connect (rtmpNow);
             nc.addEventListener (NetStatusEvent.NET_STATUS,checkConnect);
        }

        private function checkConnect (e:NetStatusEvent)
        {
             good=e.info.code=="NetConnection.Connect.Success";
             connectVal=uint(good);
             checker=[0xaa0000,0x00aa00];
             connectBtn=new ConnectBtn(checker[connectVal]);
             connectBtn.x=240,connectBtn.y=130;
             addChild (connectBtn);
        }
```

```
        private function setBW (e:Event):void
        {
             bw=list.selectedItem.label;
             nc.call ("loadit",null,bw);
        }
  }
}
```

Whenever the user selects an item in the List component, the change fires a function that calls the server-side `loadit` function and passes a parameter, `setBW`. The `setBW` parameter is simply a variable with the name of the label selected in the List component. Once the server side receives the setBW parameter, the switch statement compares the parameter with the variables in the `LoadVars()` instance `loadBW`. In this case, the variables in the text file become properties of `loadBW`. So, when the text file data is loaded into the `loadBW`, it becomes:

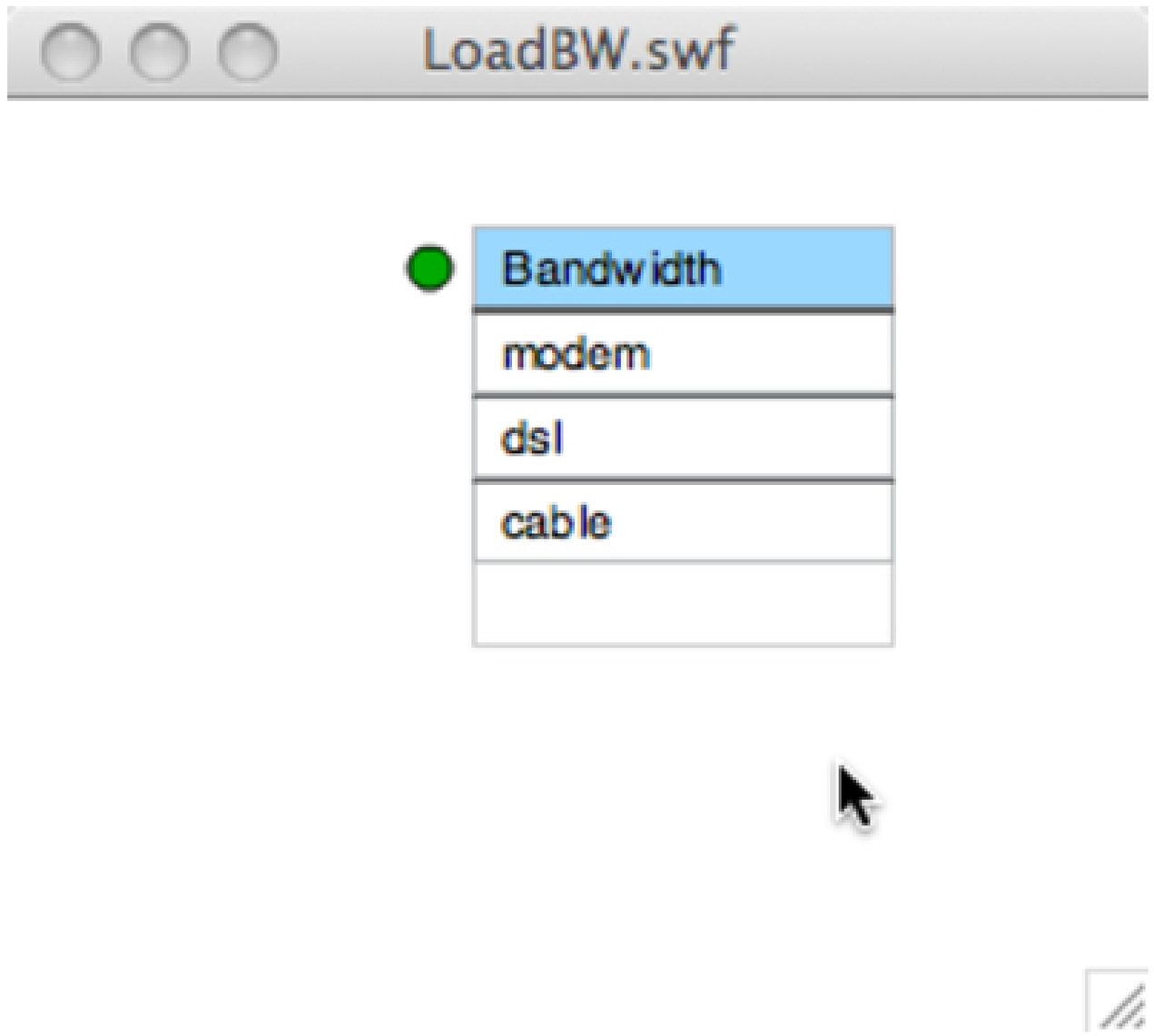
```
loadBW.modem;
loadBW.dsl;
loadBW.cable;
```

Example 10-4 shows that a good part of the script is just setting up the List component, getting a data provider and values to make a simple user interface. What it really comes down to are the lines:

```
bw=list.selectedItem.label;
nc.call ("loadit",null,bw);
```

The first line gets the value for the `bw` parameter and the second line calls the server-side script. The server-side script does the real work. When everything is set up, run the client-side application. Figure 10-3 shows how it will appear if all is working well.

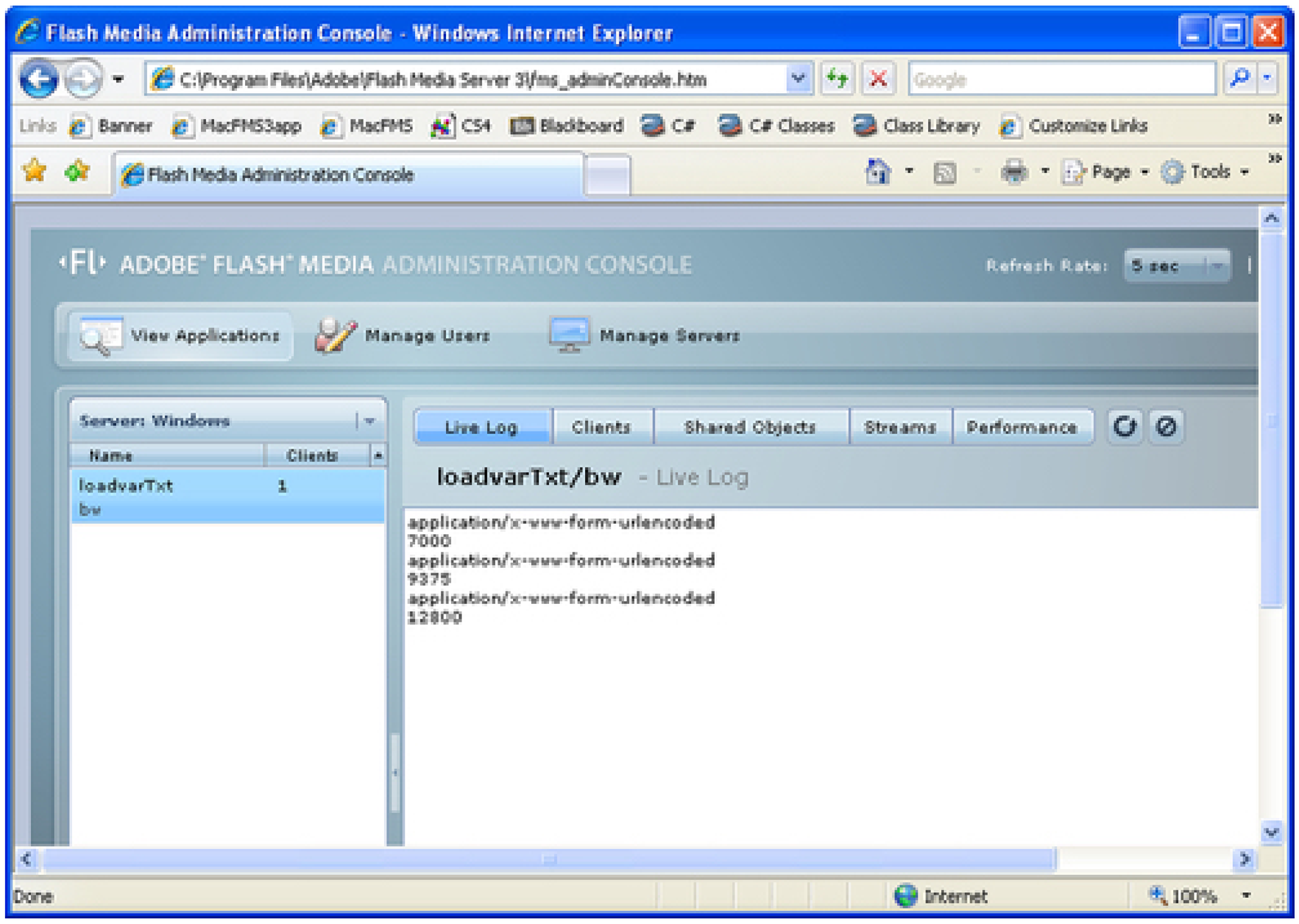
Figure 10-3. User interface and connection light



A red connection light rather than a green one means that the connection failed. Check that your server-side folders are correctly placed and that your RTMP value is correct.

Each of the properties' values is that assigned in the name-value pair provided by the text file (bw.txt). Because the value for a bandwidth setting must be a number, all of the values are numeric. Clicking a setting (modem, DSL, cable) in the List UI displays both the values and the MIME information in the Administration console, as shown in Figure 10-4.

Figure 10-4. Traces show MIME types and values loaded into bandwidth limit settings



The use of server-side `LoadVars()` in this manner may seem trivial, but it illustrates how data can be transferred from a recorded text file directly to the server without the use of a client. You can change the values in the text files easily without having to change either the server- or client-side code, and you can do it on virtually any Web host. In fact, you could change the text file without having to restart or refresh the application at all.

10.4. Server-Side XML Class

In many ways, the server-side XML class is like the LoadVars class in that it works much like the client-side version of the same class. However, instead of using text files, it works with XML files and a very different set of methods and properties. The differences between client- and server-side XML classes are significant. To show the similarities and differences between the XML and LoadVars classes, this section uses a similar application that illustrates how to work with the XML class-it sets the bandwidth limits. To get started, you'll look at an overview of the class's methods, properties, and event handlers.

10.4.1. Client-Side and Server-Side XML Class

Compared with the XML class in ActionScript 3.0, the server-side XML class is relatively small. For example, ActionScript 3.0 has an XML class with seven properties and 43 methods, while the FMS 3.0 server side XML class has 19 properties and only 17 methods. ActionScript 3.0 employs the ECMAScript standards for XML (ECMA-357 edition 2) and the FMS 3.0 XML class does not. Table 10-2 shows the methods, properties, and event handlers associated with the class used with FMS 3.0 server-side ActionScript:

Table 10-2. Methods, properties, and event handlers used with the server-side XML class

Method	Use/Function
addRequestHeader()	Places/changes *http* request header for POST operations.
appendChild()	Adds node to end of specified object's child list.
cloneNode()	Duplicates the specified node; clones all children (optional).
createElement	Creates new element.
createTextNode()	Creates new text node.
getBytesLoaded()	Gets current bytes loaded from load() or sendAndLoad method()use.
getBytesTotal()	Gets total bytes loaded from load() or sendAndLoad method() use.
getNamespaceforPrefix()	Get namespace URI associated with node's specified prefix.
getPrefixForNamespace()	Gets prefix associated with specified namespace URI for the node.
hasChildNodes()	Boolean return; true if node has child nodes, otherwise false.
insertBefore()	Inserts node in front of existing node.
load()	Loads XML document into XML object tree.
parseXML( )	Parses an XML document into the specified XML object tree.
removeNode()	Removes the specified node from its parent.
send()	Sends XML object to URL.
sendAndLoad()	Sends XML object and loads server response into another XML object.
toString()	Converts node and any children to XML text.
Property	Characteristics

Method	Use/Function
attributes	Object with all attributes of XML instance
childNodes	Array of XML's children; read-only.
contentType	Provides the MIME type sent with a call using send() or sendAndLoad( ).
docTypeDecl	Sets/returns information about declared DOCTYPE.
firstChild	Reference to first child; read-only.
ignoreWhite	Boolean setting for discarding all white spaces in text nodes; default to false.
lastChild	Last child in list of specified node; read-only.
loaded	Boolean indicating when load() or sendAndLoad() action is complete.
localName	Local name portion of node name; read-only.
namespaceURI	For node with a prefix, the namespace URI is value of XMLns for prefix-namespace URI.
nextSibling	Next sibling in parent node's child list; read-only.
nodeName	Name of the XML object's node.
nodeType	XML element or node; read-only.
nodeValue	If text node, returns the text contents.
parentNode	Reference to parent node; read-only.
prefix	Prefix portion of node name; read-only.
previousSibling	Previous sibling in parent node's child list; read-only.
status	Numeric code indicating success or failure of document parsing operation.
XMLDecl	Information about document's XML declaration.
Event Handler	Action to Activate Handler
onData	XML text download completed (or download error encountered).
onHTTPStatus	FMS receives HTTP status code.
onLoad	Either load() or sendAndLoad() operation is complete; Boolean.

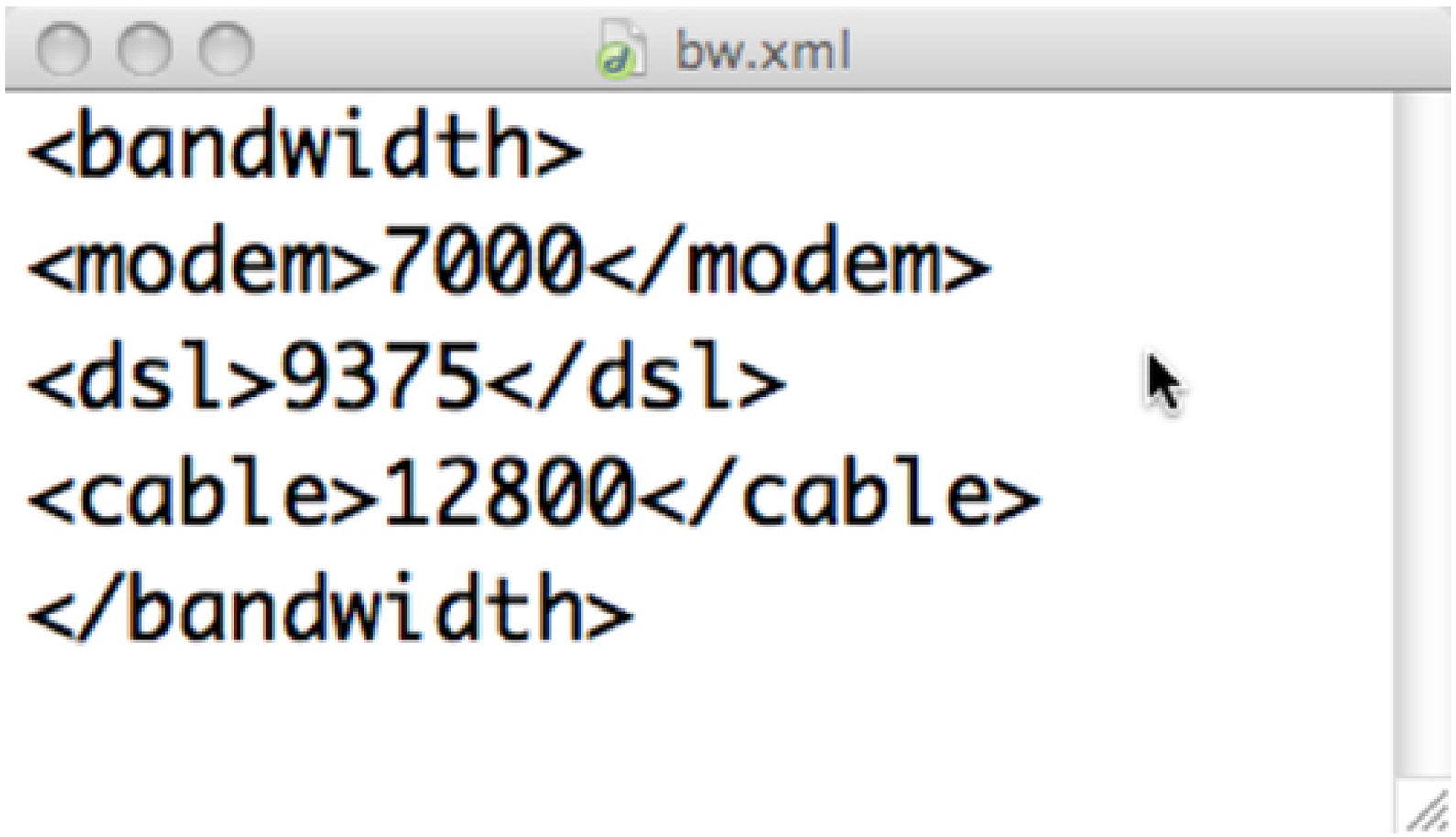
If you're not familiar with XML, none of the properties or methods will be of much use to you. Explaining XML is beyond the scope of this book. If you're not familiar with XML, you need to learn the basics before continuing. One place to start is the online tutorial at htttp://www.w3schools.com/XML/. To learn more about Standard ECMA-367 used as a base for ActionScript 3.0 XML, check out http://www.ecma-international.org/publications/standards/Ecma-357.htm. Several books on XML are available as well.

10.4.2. Placing Data in an XML File

Using a simple XML file, the same data and labels placed in the text files in name-value pairs, are put into nodes. The parent node is <bandwidth> and the three child nodes are <modem>, <dsl>, and <cable>. The actual values are placed in the child nodes and extracted using the XML class. Figure 10-5 shows the XML file in

a text editor and saved as *bw.xml*.

Figure 10-5. XML File



Using the XML class methods, the goal is to load the XML file and select the value of the correct node and place that value in a bandwidth limit setting.

10.4.3. Loading Into and Extracting Data from the XML File

Using the XML class, you can load the contents of the XML file and access data in the file. The example *bw.xml* file from the previous section shows how each part of the file can be placed into a class instance.

Like other class instances, create an XML instance using the format:

```
myXML= new XML( );
```

Once you have created an XML instance, load the XML file into the instance using the format:

```
myXML.load("http://myURL.domain.com/myfiles/my.xml");
```

If you were working with your development system, your reference would be:

```
myXML.load("http://localhost/myfiles/my.xml");
```

The XML file is wherever you place it relative to the loading reference; it does not have to be on the server but can be placed wherever you want. This is a nice way to hide your XML code from uninvited snoopers.

Once your XML data is loaded into your XML instance, you can extract it by different property references. The `childNodes` property effectively contains the entire set of nodes in the XML file. Think of it as an array of the XML object instance's contents. For example, using the *bw.xml* file, you could display the entire contents with the following function:

```
client.loadXML = function(client)
{
  var myXML = new XML( );
  myXML.ignoreWhite = true;
  myXML.onLoad = function(success)
  {
        if(success)
        {
             trace(this.childNodes);
        }
  }
  myXML.load("http://localhost/test.xml");
}
```

The output shows the entire file in the FMS3 Administrative console.

Code View:

```
<bandwidth><modem>7000</modem><dsl>9375</dsl><cable>12800</cable></bandwidth>
```

Different strategies are available to extract the contents of the XML file using the XML class properties. Given the simplicity of the sample XML file you're using, all you have to do is examine the nodes and node names of the contents of the first child, `<bandwidth>`. Like the `childNodes` property, the `firstChild` is an array with all of its child nodes. The following code snippet shows how to find a match between a search word and the name of the child node.

```
client.loadXML = function(search)
{
var myXML = new XML( );
myXML.ignoreWhite = true;
myXML.onLoad = function(success) {
if (success)
{
  var setmyXML = new XML( );
  setmyXML = this.firstChild;
  for (x=0; x<setmyXML.childNodes.length; x++)
  {
        if (setmyXML.childNodes[x].nodeName.toString()== search)
        {
             found=setmyXML.childNodes[x].childNodes[0].toString( );
```

In the server-side function that initiates the routine, a value (`search`) is passed for use in a match routine. Once the XML file has been loaded by one XML instance, another XML instance, `<bandwidth>`, is created to contain the first child. To find the match value, a loop examines the second XML instances, `setmyXML`, which is made up of the first child of the first XML instance. Given the shallow nature of the XML file, all that needs to be found is the node name of the child nodes. When one of those matches the search term, it's placed in a variable named found, converted to a string. The following outline shows this process:

- First XML instance is entire XML file that has been loaded.
- Second XML instance is first child node of first XML instance.
- Second XML instance's child nodes are in an array.
- Loop through the array to find which array element's node name matches search term.

By working with the different properties, methods, and events of the XML class, you'll find far more uses for this new server-side class. If you're not too familiar with XML, you can learn more about both XML and using the server-side XML class by experimenting with both. The next section provides a minimalist example using these constructs.

10.4.4. Minimalist Example Using Server-Side XML Class

As promised, this example does the same thing as the LoadVars example in this chapter. But rather than use text data, it uses an XML data source and the XML class. You'll use the simple XML file introduced earlier in this chapter, shown again here for reference:

```
<bandwidth>
<modem>7000</modem>
<dsl>9375</dsl>
<cable>12800</cable>
</bandwidth>
```

Save it as bw.xml in a folder named bw in the same folder where you stored your *loadXML.asc* file. It is based on a function called from the client-side script, client.loadXML. Two XML instances are used to get the needed data. First, `bwXML` loads the entire XML file. Once it's loaded, a second XML instance, `setbwXML` is loaded with the first child of `loadXML`. A specific value, `bwset`, is passed in the function and a loop searches for the child node with the name in `bwset` in the `setbwXML` instance. When that value is found, it is first transformed into a string from the XML structure, and then using the `Number` function, turns the string into a number. If you search the FMS3 Server-Side ActionScript (SSAS) dictionary, you won't find the Number class. That's because SSAS is actually JavaScript with the added SSAS classes and functions. Example 10-5 shows how the data is loaded from an XML file and placed into values:

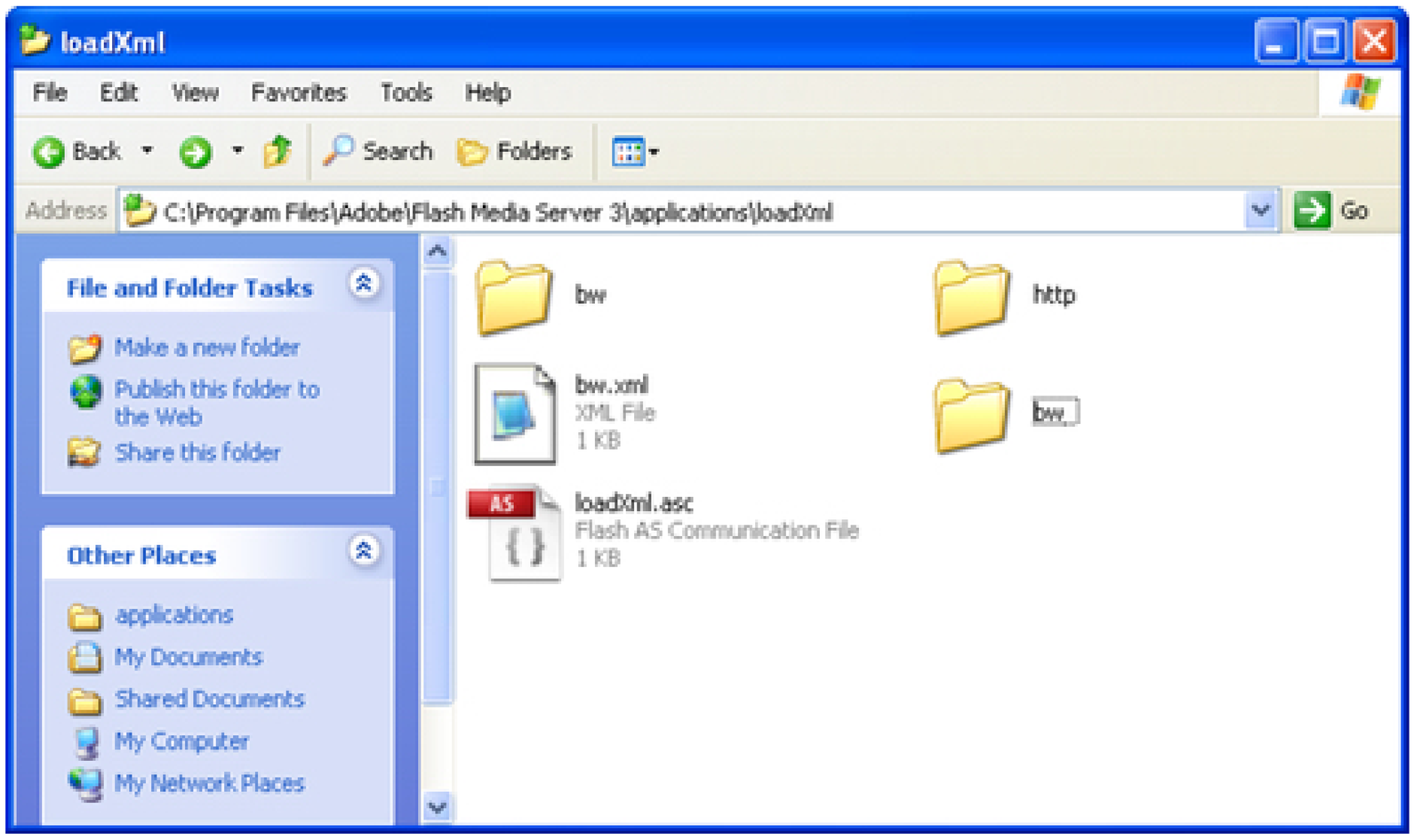
Example 10-5. loadXML.asc

Code View:

```
//XML Example #1
application.onAppStart = function()
{
  trace(this.name+" has been re-loaded");
};
application.onConnect = function(client)
{
  this.acceptConnection(client);
  client.loadXML = function(bwset) {
        var bwXML = new XML();
        bwXML.ignoreWhite = true;
        bwXML.onLoad = function(success)
        {
             if (success) {
                  var setbwXML = new XML();
                  setbwXML = this.firstChild;
                  for (x=0; x<setbwXML.childNodes.length; x++)
                  {
                       if (setbwXML.childNodes[x].nodeName.toString() == bwset)
                       {
                       s2c = c2s=setbwXML.childNodes[x].childNodes[0].toString();
                            s2c = Number(s2c);
                            c2s = Number(c2s);
                            trace(s2c);
                            client.setBandwidthLimit(s2c, c2s);
                       }
                  }
             }
             else
             {
                  trace("problem");
             }
        };
        bwXML.load("http://localhost/bw/bw.xml");
  };
};
```

Save the file as loadXML.asc or main.asc, your preference. Place it in a folder named loaderXML on the server-side applications folder.

Figure 10-6. Server side file arrangement using xml data source



Then, using a client-side script that's almost identical to the one you used for the loadVars example, create a user interface to send the message to the server-side script that will fire the different functions.

Example 10-6. LoadXMLbwt.as

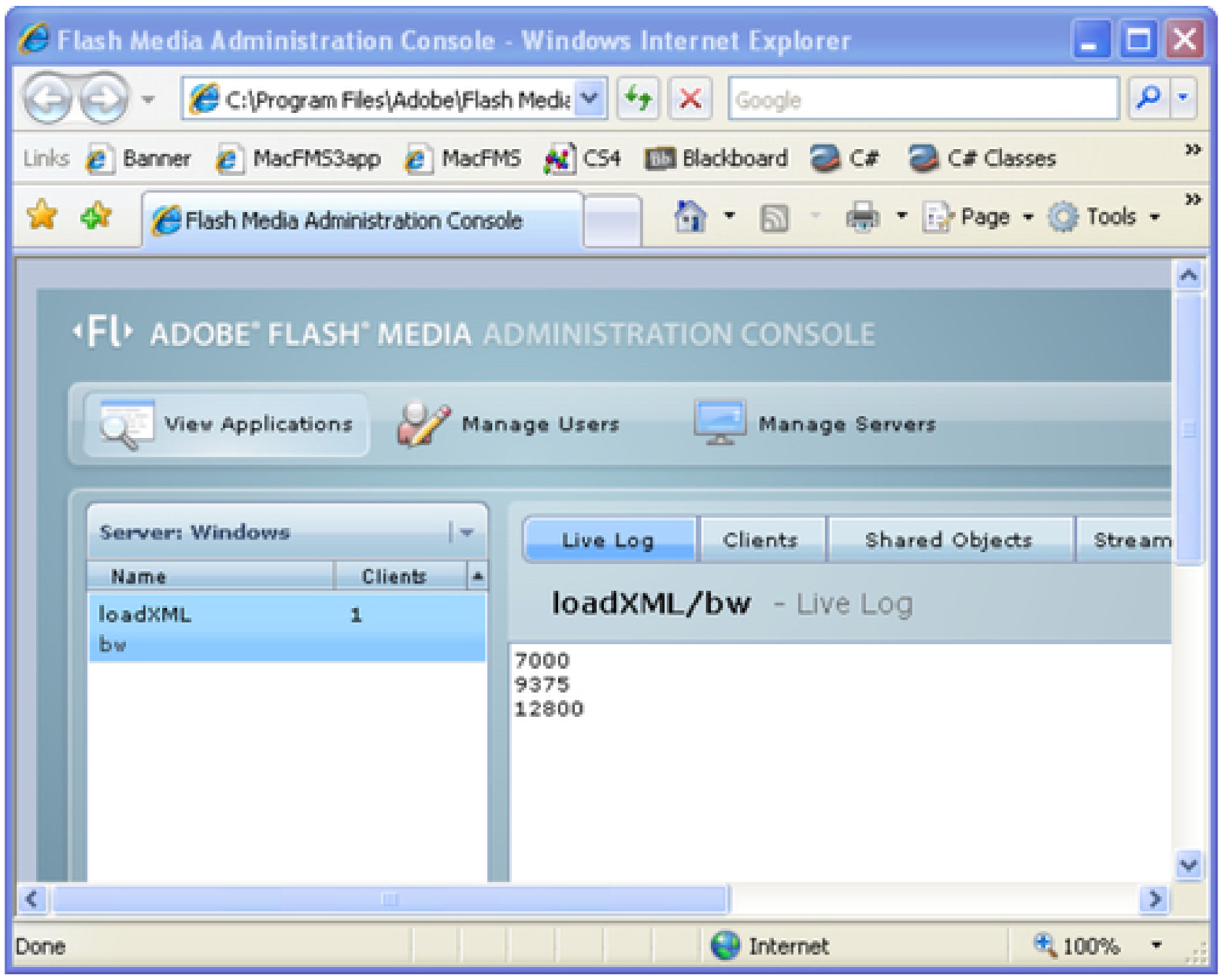
Code View:

```
package
{
  import fl.controls.List;
  import fl.controls.Label;
  import flash.net.NetConnection;
  import fl.data.DataProvider;
  import flash.events.NetStatusEvent;
  import flash.events.Event;
  import flash.display.Sprite;

  public class LoadXMLbw extends Sprite
  {
        private var rtmpNow:String;
        private var nc:NetConnection;
        private var list:List;
        private var dp:DataProvider;
        private var listLabel:Label;
        private var bw:String;
        private var good:Boolean;
        private var connectLight:ConnectLight;
        private var connectVal:uint;
        private var checker:Array;

        public function LoadXMLbw ()
        {
             dp=new DataProvider();
             dp.addItem ({label:"Bandwidth"});
             dp.addItem ({label:"modem"});
             dp.addItem ({label:"dsl"});
```

```
            dp.addItem ({label:"cable"});

            list=new List();
            list.move (250,120);
            list.addEventListener (Event.CHANGE, setBW);
            list.labelField="label";
            list.dataProvider=dp;
            list.selectedIndex=0;
            addChild (list);

            rtmpNow = "rtmp://192.168.0.11/loadXML/bw";
            nc = new NetConnection ();
            nc.connect (rtmpNow);
            nc.addEventListener (NetStatusEvent.NET_STATUS,checkConnect);
        }

        private function checkConnect (e:NetStatusEvent)
        {
            good=e.info.code=="NetConnection.Connect.Success";
            connectVal=uint(good);
            checker=[0xaa0000,0x00aa00];
            connectLight=new ConnectLight(checker[connectVal]);
            connectLight.x=240,connectLight.y=130;
            addChild (connectLight);
        }

        private function setBW (e:Event):void
        {
            bw=list.selectedItem.label;
            nc.call ("loadXML",null,bw);
        }
    }
}
```

When you test it, your Administration console simply shows the actual values that are passed, as shown in Figure 10-7.

Figure 10-7. Bandwidth settings from XML file



All you see are the different settings for the selected bandwidth settings, and that's fine since all the application does is change bandwidth settings. However, to understand a little more about the XML class properties, this next script traces out more features and uses another property.

NOTE

If the application does not work with your hosting service, check the protocols it uses. Everything should be fine if you test it on a local host or using a LAN or on a remote server, but some hosting services have certain protocols that disallow certain kinds of access.

Before entering the script, you need to take a quick look at the `XML.attributes` property. Essentially, the attributes property is an object containing all of the attributes of an XML instance. Using a key index, you can both assign and retrieve an `XML.attributes` associative array. The key index is a name you used for the attribute. In this next example, the argument passed from the client-side script, `bwset`, is first placed into a `Client.property` named `setter`. Then the `client.setter` property is assigned to the first child attributes named `myBW`. This is unnecessarily convoluted, but the purpose is to show how values can be assigned to different elements that make up XML properties. To work with this, just add the code in Example 10-7, saved as *loadXML.asc* or *main.asc* to the original *.asc* script.

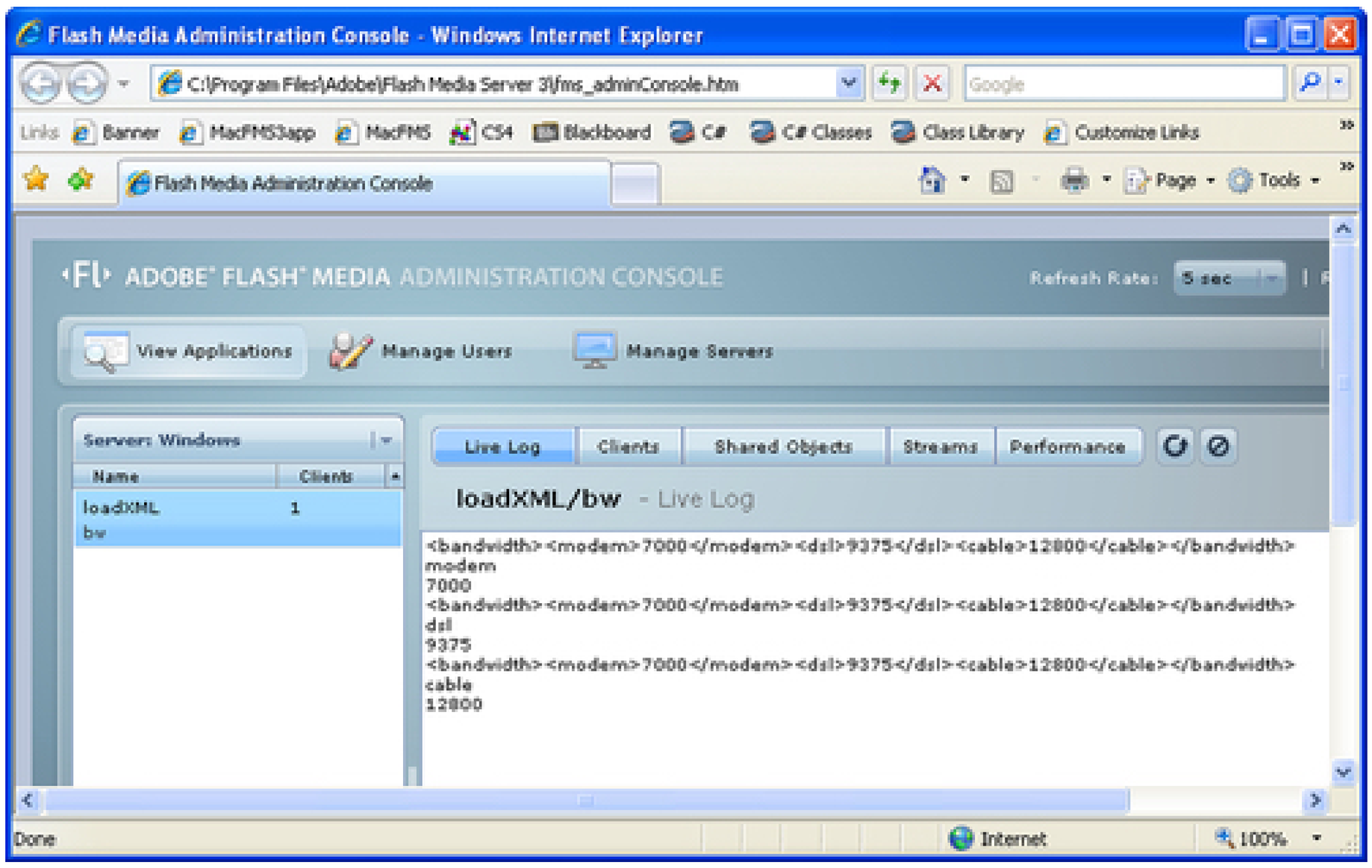
Example 10-7. LoadXML.as

Code View:

```
//XML Example 2
application.onAppStart = function()
{
  trace(this.name+" has been re-loaded");
};
application.onConnect = function(client)
{
  this.acceptConnection(client);
  client.loadXML = function(bwset)
  {
        this.setter = bwset;
        var bwXML = new XML();
        bwXML.ignoreWhite = true;
        bwXML.onLoad = function(success)
        {
             if (success)
             {
                  var setbwXML = new XML();
                  trace(this.childNodes);
                  setbwXML = this.firstChild;
                  setbwXML.firstChild.attributes.myBW = client.setter;
                  var showAtt = setbwXML.firstChild.attributes.myBW;
                  trace(showAtt);
                  for (x=0; x<setbwXML.childNodes.length; x++)
                  {
                       if (setbwXML.childNodes[x].nodeName.toString() == bwset)
                       {
                            s2c = c2s=setbwXML.childNodes[x].childNodes[0].
toString();
                            s2c = Number(s2c);
                            c2s = Number(c2s);
                            trace(s2c);
                            client.setBandwidthLimit(s2c, c2s);
                       }
                  }
             }
             else
             {
                  trace("problem");
             }
        };
        bwXML.load("http://localhost/bw/bw.xml");
  };
};
```

Figure 10-8 shows the additional information passed on to the FMS3 console. As you begin to work with the XML class, this should give you a better idea of what information is stored in the files.

Figure 10-8. Expanded information shown from the XML object



The server-side XML class is an ideal way to access information passed from a data source such as a database. By accessing the XML file, the XML class becomes a simple way of using a wide variety of information stored in a database with the coupled advantage of security.

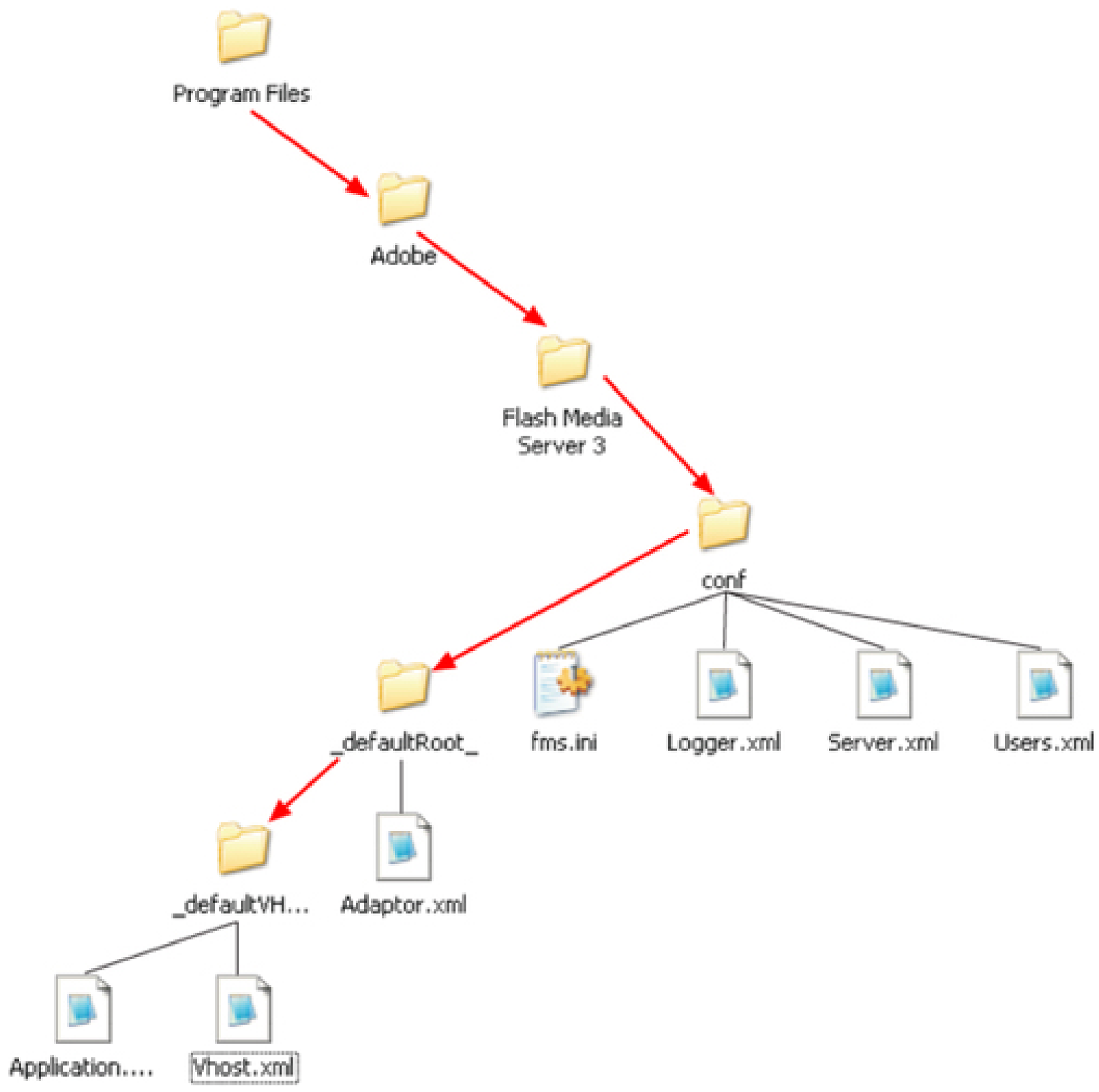
10.5. Using the Configuration Files

FMS3 contains a number of XML configuration files that you can use to customize it for special needs and applications. This section provides an overview of these files and how they might be changed for a limited range of applications. However, if you want to change these files for more advanced uses, especially on your actual server (instead of locally on your development PC used as a server), read Chapter 3, *Configuring the server* of the PDF file, *Adobe Flash Media Server: Configuration and Administration Guide*, which is in the documentation that comes with FMS3 (*flashmediaserver_config_admin.pdf*). The examples and explanations here are very specific for purposes of illustration and may not apply to what you need (or want!) for your system.

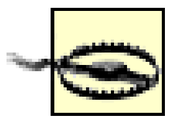
10.5.1. Mapping the Config Files

Because the files are tucked away in the root of FMS3, a map will help you locate them on a typical Windows setup. Linux has the same file beginning at the Linux FMS3 root within the parallel folders. Figure 10-9 provides a map and outline of the files.

Figure 10-9. Map of the configuration files



In the mapping, the full name of the defaultVHost folder and the Application.xml file were cut off. Otherwise, everything is where you should find them in the respective folders.



If you want to experiment with the different files in the *conf* directory, make a backup of the folder and all of its subfolders and files, and put it in a safe place. Then if you run into trouble you can get the backup and use it to return the configuration to the original state.

10.5.2. Changing XML and ini Files

Changing the values in any of the XML files or the FMS.ini file is simply a matter of opening them using a text editor such as Microsoft Notepad, making the changes, and saving the file using the same name. This section looks at the different files and provides examples of what you can change. Keep in mind, though, that this is just an introductory glimpse of what can be changed. You'll need to back up all of the originals and work only in a local development environment until you've gone over the Chapter 6*: Administering the server* in *Adobe Flash Media Server: Configuration and Administration Guide* carefully. To be safe, copy the originals to a separate drive, disk, or CD. If you change any of the files in the conf file or restore them to the originals, you must restart your FMS3 server for them to take effect.

10.5.2.1. FMS.ini

The first file to look at is the FMS.ini file. It provides an overview of the parameters you can change in the different XML files. The file is well commented-the pound signs (#) indicate a comment line. For example, the following code segment shows the extensive comments, indicating the configurable parameters in the *Server.xml* file. Shown is the example and the actual host port for logging into the FMS console following the commented code showing the correct protocol.

```
###############################################################
# This section contains configurable parameters in Server.xml #
###############################################################
# IP address and port Flash Media Admin Server should listen on
# For example:
# SERVER.ADMINSERVER_HOSTPORT = :1111
#
SERVER.ADMINSERVER_HOSTPORT = :1111
```

Among other uses, this information lets you look up your administrator username or password, if you ever forget it, in the *FMS.ini* file. All of the information is written to the file when you initially install FMS3, but if you change it by writing into the FMS.ini and *Server.xml* files, it will accept the new parameters once the server is restarted.

If a parameter is set in the FMS.ini file, it is in one of the XML files (and not always in the ones in the comment lines) and has a special format. For example, the Server.xml file contains the following:

```
<HostPort>${SERVER.ADMINSERVER_HOSTPORT}</HostPort>
```

It points to `SERVER.ADMINSERVER_HOSTPORT` in the FMS.ini file. This value is typically 1111, but changing the value in the FMS.ini file, not the XML file, changes the value. Updating is a lot easier because rather than poking

through several XML files, you can make changes in the FMS.ini file, and these changes will be passed to the XML file with the appropriate tag.

10.5.2.2. Server.xml

The *Server.xml* file has 81 different tags used in configuring the server, by far the most of any of the XML files. You need to become familiar with each of these tags before making any changes. One tag that you may have to work with is `<HostPort>`, to configure it so that it works with your firewall configuration. However, because the host port value is in the FMS.ini file, you must make any changes to the FMS.ini file instead of directly to the `<HostPort>` tag in the Server.xml file.

10.5.2.3. Users.xml

If you want to increase your security, *Users.xml* is a good place to start. You will find both `<Allow>` and `<Deny>` tags in both the `<User>` and `<HTTPCommands>` containers. If you want to restrict access points for the administrator to Flash Media Admin Service, for example, you would set up the `<Allow>` tags in the `<User>` container to a specific list of domains. For example:

```
<Allow>www.sandlight.com, 12.34.5.678 </Allow>
```

allows access only from either the URL http://www.sandlight.com or the IP address 12.34.5.678. If anyone from another domain attempts to access the Flash Media Admin Service, they'll be denied. The `<Deny>` tags do the opposite. Any address in the `<Deny>` container stops only those listed from access.

10.5.2.4. Logger.xml

The FMS3 folder contains a logs folder, inside of which are several different types of logs. For example, access.00.log shows the most recent start of your server and different actions taken-such as connections, disconnections, plays, and stops. If you're using the Developer version of FMS3, a new log appears every time you restart your computer or restart FMS3. Several other log files in the same folder provide other information. A folder within the logs folder, *_defaultVHost_*, contains logs for the applications you have run on the server. For example, I have an application named buf, and within the *buf* folder is a log, *application.00.log*, showing dates and times of use, IP addresses of users, and other information that was passed about logging on and off the application.

You can configure the server to create individual log files for each vhost by changing the Scope tag in the *Server.xml* file. The main logging file, though, is the *Logger.xml* file. It contains different parameters for the nodes as in the other XML files used for configuring the server. For example, the default `<history>` node value in *Logger.xml* is 5. Basically, that means that after log 05 (for example, admin.05.log), the next server start will knock off the last log (05) as the other log numbers get shifted upward, starting with log 00. If you want to have more than six logs of the different files (00–05), just increase the `<history>` node value. This can be handy during certain types of development cycles when you want to keep track of the logged values over a period greater than the default. Several other nodes in the file let you customize how your logs keep data and its format. For example, the `<Time>` node can be changed from the default `local` to `UTC` if you're working with others across time zones, especially ones on opposite sides of the 0 GMT, such as Paris and New York, or International Date Line, such as Los Angeles and Tokyo. You can also specify the delimiter used to separate your log data, which can be useful if you want to integrate with third-party or other custom programs to interpret and present your log data. You can also specify what events get logged, and where the log files are stored, among other settings.

10.5.2.5. Adaptor.xml

The adaptor file specifies several different tags for individual network adaptors. One of the primary user-set values is the host port, typically 1935. Within the `<HostPortList>` container is the `<HostPort>` node. Like a lot

of other values in the different XML files, the actual value is held in the FMS.ini file and passed to the XML file in the format `${FILENAME.NODENAME}`. So the actual value is in the *FMS.ini* file, and any changes of that port number need to be placed in the FMS.ini file. You can also employ the Adaptor.xml file to selectively allow or reject clients using the `<Allow>` and `<Deny>` tags, specifying comma delimited lists of URLs or IP addresses from the originating client. If you need to implement an SSL certificates for each adaptor, you'd add those settings here as well.

10.5.2.6. Vhost.xml

The Vhost.xml file houses the virtual host parameters for FMS. This is an extremely useful and a bit more complex file than the others. The primary importance of the Vhost.xml file is for creating additional virtual hosts using a single server. The addition of virtual hosts lets you use the same FMS server with several different developers working on different applications. For example, a university may want to have different virtual hosts for students so that they can develop their own applications and have access to separate application roots. The process of setting up additional virtual hosts is clearly beyond the scope of this book and cannot be done with the Developer's version of FMS3, but you will find some useful tutorials online. For example, Stefan Richter's tutorial at http://www.flashcomguru.com/tutorials/configureFCS.cfm provides a step-by-step instructions for adding vhosts to Flash Communication Server (not FMS3, but the setups are almost the same).

One change you can make to your *Vhost.xml* file that may be instructive, and in some applications extremely useful, is to change the applications root. The `<AppsDir>` node is the path to the source of your server-side files. The default address is in the FMS.ini file as VHOST.ASSPDIR with an assigned address. The default address is C:\Program Files\Adobe\Flash Media Server 3\applications.

By changing the address, you can store your server-side applications wherever you want. This is especially important for multiple virtual hosts, but sometimes it's simply easier for administrative purposes to have your server-side materials, including all of your FLV files for streaming, in some place other than the default location. However, if you're using multiple virtual hosts, you must remove (or simply comment out using a pound sign) the `VHOST.APPSDIR` line in the FMS.ini file. Then, in the Vhost.xml file, place the address in the `<AppsDir>` node.

10.5.2.7. Application.xml

The final XML file in the configuration folders is used for the settings for the different applications you create. The default settings will differ depending on your license configuration, FMIS, FMSS, or the Developer version. One of the important node attributes you will find in the Application.xml file is override. The override attribute specifies whether a setting can be overridden by a different setting in the application. Earlier versions of Flash Media Server had bandwidth limitations, but none of the three current versions, FMIS3, FMSS3, or Developers FMS3, has limitations. Still, you may want to cap bandwidth for your physical server's capabilities or for unit charges of bandwidth. So you could change the bandwidth override ability in the *Application.xml* from,

```
<Bandwidth override=""no"">
```

to

```
<Bandwidth override=""yes"">
```

By assigning `override=""yes""` to a node, you allow the application to set a value that exceeds the limit set by the application. So, for example, for the following setting in your *Application.xml* file:

```
<ServerToClient>12500000</ServerToClient>
<ClientToServer>12500000</ClientToServer >
```

and your bandwidth override is set to "yes," your application can have the following line in a server-side script that will set the bandwidth limits beyond the value set in the nodes:

```
client.setBandwidthLimit(15000000, 22500000);
```

Of course, in cases where preserving and limiting the bandwidth for applications is lower, you can reset the values in the *Application.xml* file so that no bandwidth overrides are possible. Remember, these settings apply to all of the applications running on the server or vhost. If you want to apply settings to a specific application, you can do so by editing a version of *Application.xml* and placing it in the directory of the application (for example, applications/myApp), the same location as your *main.asc* file.

10.6. Doing More with Flash Media Server 3

This book is designed as a starting point for work with FMS3. While you've seen several applications using FMS3, they are meant to be a doorway to an unlimited set of possibilities and not a finite set of limits. In the years that I've been using Flash Communication Server, FMS2, and FMS3, I've never ceased to be amazed by the incredible and practical applications developed with it. Audio/visual communication over the Web is becoming a fact of life and certainly holds a central place in FMS3 applications. But communication with text and graphics using shared objects is equally important and coming to play a central role in Web-based applications, where different media needs to be shared and dynamically updated over the Web. For me, the best source of new ideas and applications for FMS3 has been my clients. Because they have very practical goals to achieve, they see FMS3 as a means to accomplish exactly what they want to get done. Furthermore, my clients represent a wealth of creativity. I always look forward to starting new projects using FMS3 to address new challenges, and usually end up with even more new insights into what is possible with this incredible development environment.