This excerpt is protected by copyright law. It is your responsibility to obtain permissions necessary for any proposed use of this material. Please direct your inquiries to permissions@oreilly.com.
Adobe Developer Library, a copublishing partnership between O’Reilly Media Inc., and Adobe Systems, Inc., is the authoritative resource for developers using Adobe technologies. These comprehensive resources offer learning solutions to help developers create cutting-edge interactive web applications that can reach virtually anyone on any platform.

With top-quality books and innovative online resources covering the latest tools for rich-Internet application development, the Adobe Developer Library delivers expert training straight from the source. Topics include ActionScript, Adobe Flex®, Adobe Flash®, and Adobe Acrobat®.

Get the latest news about books, online resources, and more at http://adobedeveloperlibrary.com.
# Table of Contents

1. **Getting Started with Flash Media Server 3** ................................. 1
   - The New Flavors for Flash Media Server 3 ................................. 1
   - What Is a Media Server? ......................................................... 2
   - Installing FMS3 ........................................................................ 3
   - Organizing Your Development Environment .............................. 12
   - Testing FMS3 Connections ......................................................... 16
   - Using the FMS3 Administration Console .................................. 18
   - Using This Book ....................................................................... 19

2. **Recording and Playing Back Streaming Audio and Video** ................. 23
   - Streaming and Broadcasting ...................................................... 23
   - Minimalist Project .................................................................... 24
   - Combined Record and Playback Application .............................. 40

3. **Setting Your Camera and Microphone** ........................................... 45
   - Camera and Microphone Methods for Setting Parameters ........... 45
   - Minimalist Project .................................................................... 48
   - Dynamically Testing Your Camera and Microphone Settings ....... 55
   - Key Considerations .................................................................. 61
   - Adjusting Camera and Audio with Flash Media Encoder ............ 62

4. **Nonpersistent Client-Side Remote Shared Objects** ............................ 67
   - Sharing Data on Multiple Connections ...................................... 67
   - Instantiating Remote Shared Objects ........................................ 68
   - Minimalist Project for Shared Movie Clip .................................. 73
   - Minimalist Project for Shared Text ............................................ 77
   - Minimalist Project for Shared Function ...................................... 80
   - An Upgraded Text Chat .............................................................. 85

5. **Two-Way Audio-Video Communications** ......................................... 91
   - Face-to-Face Communication .................................................... 91
   - The NetStream Bundle .............................................................. 91
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>The NetStream Class and Live Streams</td>
<td>92</td>
</tr>
<tr>
<td>The World’s Easiest Two-Way A/V Chat Application</td>
<td>94</td>
</tr>
<tr>
<td>A Better Two-Way Chat Application</td>
<td>98</td>
</tr>
<tr>
<td>Four-Way Conference Application</td>
<td>107</td>
</tr>
<tr>
<td>Moving On to More Server-Side Applications</td>
<td>114</td>
</tr>
<tr>
<td><strong>6. Broadcasting and Server-Side Bandwidth Control</strong></td>
<td>115</td>
</tr>
<tr>
<td>Casting Many Streams</td>
<td>115</td>
</tr>
<tr>
<td>Switching Cameras</td>
<td>115</td>
</tr>
<tr>
<td>The Minimum Studio</td>
<td>118</td>
</tr>
<tr>
<td>Introduction to the Server Side</td>
<td>126</td>
</tr>
<tr>
<td>Dynamic Camera, Microphone, and Bandwidth Controls</td>
<td>135</td>
</tr>
<tr>
<td>Bandwidth Checker</td>
<td>144</td>
</tr>
<tr>
<td>Conclusion</td>
<td>148</td>
</tr>
<tr>
<td><strong>7. Working With Server-Side Files: The File Class</strong></td>
<td>149</td>
</tr>
<tr>
<td>Recording Data</td>
<td>149</td>
</tr>
<tr>
<td>The File Class</td>
<td>150</td>
</tr>
<tr>
<td>Client-Side Formatting</td>
<td>160</td>
</tr>
<tr>
<td>Server-Side Formatting</td>
<td>162</td>
</tr>
<tr>
<td>Beggar’s Database</td>
<td>166</td>
</tr>
<tr>
<td><strong>8. Server-Side Shared Objects</strong></td>
<td>173</td>
</tr>
<tr>
<td>What Are Server-Side Shared Objects?</td>
<td>173</td>
</tr>
<tr>
<td>Working with Server-Side Shared Objects</td>
<td>174</td>
</tr>
<tr>
<td>Removing Users</td>
<td>179</td>
</tr>
<tr>
<td>Persistent Server-Side Shared Object</td>
<td>186</td>
</tr>
<tr>
<td>The Idea Factory</td>
<td>192</td>
</tr>
<tr>
<td><strong>9. Server-Side Streams</strong></td>
<td>201</td>
</tr>
<tr>
<td>Stream Management</td>
<td>201</td>
</tr>
<tr>
<td>Anatomy of <code>Stream.play()</code></td>
<td>201</td>
</tr>
<tr>
<td>Playing MP3 Files</td>
<td>203</td>
</tr>
<tr>
<td>Changing Streams</td>
<td>217</td>
</tr>
<tr>
<td>Server-Side NetStream Class</td>
<td>230</td>
</tr>
<tr>
<td><strong>10. Bringing in Data and Working with Configuration Files</strong></td>
<td>239</td>
</tr>
<tr>
<td>Cue Points, Metadata, and Stream Completion</td>
<td>239</td>
</tr>
<tr>
<td>Server-Side LoadVars Class</td>
<td>243</td>
</tr>
<tr>
<td>Minimalist Example Using Server-Side <code>LoadVars()</code></td>
<td>245</td>
</tr>
<tr>
<td>Server-Side XML Class</td>
<td>251</td>
</tr>
<tr>
<td>Using the Configuration Files</td>
<td>261</td>
</tr>
<tr>
<td>Doing More with Flash Media Server 3</td>
<td>266</td>
</tr>
</tbody>
</table>
CHAPTER 5

Two-Way Audio-Video Communications

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.

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 × 2 = 4; 4 – 2 = 2) while four connections generate twelve (4 × 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.

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:

```javascript
//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:

```javascript
//Stream In
nsIn=new NetStream(nc);
nstream.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.

**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](image)

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

**Classes**

- NetConnection

94 | Chapter 5: Two-Way Audio-Video Communications
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

```actionscript
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.
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 "Testing FMS3 Connections" in Chapter 1.

**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. How-
ever, 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.

**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.

**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:

```javascript
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
- onDisconnect

**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 event of 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.

**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.

```javascript
application.onConnect=function(currentClient)
```

*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:

```javascript
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.

```javascript
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.

### 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.

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

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

*Example 5-2. deux.asc*
//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.)
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.

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:

```javascript
responder = new Responder (streamNow);
nc.call ("streamSelect",responder);
```

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,

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

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

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.
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:

```ActionScript
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.

**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*

```actionscript
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.

![Two-way chat with one module](image)

**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.

**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.
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 be essential. To begin, look at the size of the videos sent across the Internet. In the Deux application, the camera is set to 240 × 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 × 180 matrix of bits, the matrix is 80 × 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 × 100 = 8,000.
b. 8,000 × 12 = 96,000 (where 12 is the number of streams).
c. 96,000 × 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.

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:

```java
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:

```java
//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.

**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 × 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](image)

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.

```java
Example 5-4. Quad.as
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);
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.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.

“Four-Way Conference Application” | 113
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.

**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.