

ADOBE® FLASH® MEDIA SERVER DEVELOPER GUIDE



© 2007 Adobe Systems Incorporated. All rights reserved.

Adobe® Flash® Media Server Developer Guide

If this guide is distributed with software that includes an end user agreement, this guide, as well as the software described in it, is furnished under license and may be used or copied only in accordance with the terms of such license. Except as permitted by any such license, no part of this guide may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, recording, or otherwise, without the prior written permission of Adobe Systems Incorporated. Please note that the content in this guide is protected under copyright law even if it is not distributed with software that includes an end user license agreement.

The content of this guide is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies that may appear in the informational content contained in this guide.

Please remember that existing artwork or images that you may want to include in your project may be protected under copyright law. The unauthorized incorporation of such material into your new work could be a violation of the rights of the copyright owner. Please be sure to obtain any permission required from the copyright owner.

Any references to company names in sample templates are for demonstration purposes only and are not intended to refer to any actual organization.

Adobe, the Adobe logo, Adobe AIR, ActionScript, Creative Suite, Dreamweaver, Flash, Flex, and Flex Builder are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries.

Java is a trademark or registered trademark of Sun Microsystems, Inc. in the United States and other countries. Microsoft and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. All other trademarks are the property of their respective owners.

**Sorenson
Spark.** Sorenson™ Spark™ video compression and decompression technology licensed from Sorenson Media, Inc.

Licensee shall not use the MP3 compressed audio within the Software for real time broadcasting (terrestrial, satellite, cable or other media), or broadcasting via Internet or other networks, such as but not limited to intranets, etc., or in pay-audio or audio on demand applications to any non-PC device (i.e., mobile phones or set-top boxes). Licensee acknowledges that use of the Software for non-PC devices, as described herein, may require the payment of licensing royalties or other amounts to third parties who may hold intellectual property rights related to the MP3 technology and that Adobe has not paid any royalties or other amounts on account of third party intellectual property rights for such use. If Licensee requires an MP3 decoder for such non-PC use, Licensee is responsible for obtaining the necessary MP3 technology license.

Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA.

Notice to U.S. Government End Users. The Software and Documentation are "Commercial Items," as that term is defined at 48 C.F.R. §2.101, consisting of "Commercial Computer Software" and "Commercial Computer Software Documentation," as such terms are used in 48 C.F.R. §12.212 or 48 C.F.R. §227.7202, as applicable. Consistent with 48 C.F.R. §12.212 or 48 C.F.R. §§227.7202-1 through 227.7202-4, as applicable, the Commercial Computer Software and Commercial Computer Software Documentation are being licensed to U.S. Government end users (a) only as Commercial Items and (b) with only those rights as are granted to all other end users pursuant to the terms and conditions herein. Unpublished-rights reserved under the copyright laws of the United States. Adobe Systems Incorporated, 345 Park Avenue, San Jose, CA 95110-2704, USA. For U.S. Government End Users, Adobe agrees to comply with all applicable equal opportunity laws including, if appropriate, the provisions of Executive Order 11246, as amended, Section 402 of the Vietnam Era Veterans Readjustment Assistance Act of 1974 (38 USC 4212), and Section 503 of the Rehabilitation Act of 1973, as amended, and the regulations at 41 CFR Parts 60-1 through 60-60, 60-250, and 60-741. The affirmative action clause and regulations contained in the preceding sentence shall be incorporated by reference.

Contents

Chapter 1: Getting started

Overview	1
Set up a development environment	3
Hello World application	4
Create an application	6
Test an application	8
Deploy an application	10

Chapter 2: Streaming services

Using the live service	12
Using the vod service	13
Creating clients for streaming services	14

Chapter 3: Developing media applications

Connecting to the server	16
Managing connections	21
Recorded streams	23
Handling errors	28
Playlists	30
Multiple bit rate switching	31
Detecting bandwidth	32
Detecting stream length	36
Buffering streams dynamically	39

Chapter 4: Developing live video applications

Capturing and streaming live audio and video	41
Record live video	43
Add metadata to a live stream	44
Publish from server to server	48

Chapter 5: Developing social media applications

Shared objects	51
Allow and deny access to assets	55
Authenticate clients	56
Authenticate users	60

Chapter 1: Getting started

Adobe® Flash® Media Server offers a combination of streaming media and social interactivity for building rich media applications. Flash Media Server offers instant start, live video streams, and variable streaming rates based on the user's bandwidth.

There are three editions of Flash Media Server:

Flash Media Interactive Server The full-featured edition of the server.

Flash Media Development Server A development version of Flash Media Interactive Server. Supports all the same features but limits the number of connections.

Flash Media Streaming Server Supports the live and vod streaming services only. This server edition does not support server-side scripting or stream recording.

Note: It's a good idea to read Adobe Flash Media Server Technical Overview before using this guide.

Overview

Client-server architecture

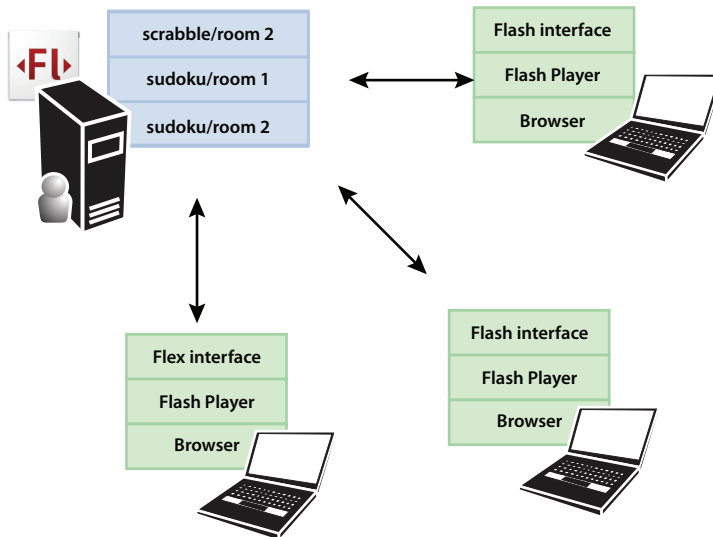
Flash Media Server is a hub. Applications connect to the hub using Real-Time Messaging Protocol (RTMP and its variants, for more information, see [The application URI](#)), and the server can send data to and receive data from many connected users. A user can capture live video or audio using a webcam or microphone attached to a computer running Adobe Flash Player and publish it to a server that streams it to thousands of users worldwide. Users worldwide can participate in an online game, with all moves synchronized for all users.

Users connect to the server through a network connection. A connection is similar to a large pipe and can carry many streams of data. Each stream travels in one direction and transports content between one client and the server. Each server can handle many connections concurrently, with the number determined by your server capacity.

An application that runs on Flash Media Server has a client-server architecture. The client application is developed in Adobe Flash or Adobe Flex and runs in Flash Player, AIR, or Flash Lite 3. It can capture and display audio and video and handle user interaction. The server application runs on the server. It manages client connections, writes to the server's file system, and performs other tasks.

The client must initiate the connection to the server. Once connected, the client can communicate with the server and with other clients. More specifically, the client connects to an *instance* of the application running on the server. An example of an application instance is an online game with different rooms for various groups of users. In that case, each room is an instance.

Many instances of an application can run at the same time. Each application instance has its own unique name and provides unique resources to clients. Multiple clients can connect to the same application instance or to different instances.



Several clients connecting to multiple applications (sudoku and scrabble) and application instances (room 2, room 1, and room 2) running on Flash Media Server

Parts of a media application

The client application is written in ActionScript™ and compiles to a SWF file. The server application is code written in Server-Side ActionScript (which is similar to ActionScript 1.0, but runs on the server, rather than on the client). A media application usually has recorded or live audio and video that it streams from server to client, client to server, or server to server.

A typical Flash Media Server application has these parts:

Client user interface The client displays a user interface, such as controls to start, stop, or pause a video. The user interface can run in Flash Player, AIR, or Flash Lite 3 and can be developed with Adobe Flash or Adobe Flex.

Client-side ActionScript The client contains ActionScript code that handles user interaction and connects to the server. Flash Media Server 3 supports ActionScript 3.0. Client applications developed in ActionScript 2.0 or ActionScript 1.0 for an earlier version of Flash Media Server are compatible with Flash Media Server 3.

Video or audio Many media applications stream recorded audio or video from the server to clients, or from a client to the server and then to other clients. Pre-recorded files may be in Flash Video (FLV), MP3, or MP4 format. Video files recorded by the server are always in FLV format, with the suffix *.flv*. See [Stream formats](#).

Camera or microphone A client can stream live video or audio to the server using the [Adobe Flash Media Encoder](#) or your own custom Flash application that supports live streaming. The client captures audio and video using its own microphone and camera.

Server-Side ActionScript Many applications include Server-Side ActionScript code packaged in a file with the suffix *.asc*, formally called an *ActionScript Communication File*. The file is named either *main.asc*, or *myApplication.asc* (see [Writing server-side code](#)). The server-side code handles the work the server does, such as streaming audio and video and defining what happens when users connect and disconnect. See the *Server-Side ActionScript Language Reference*.

Stream formats

Flash Media Server supports playback of a variety of stream formats, including Flash Video (FLV), MPEG-3 (MP3), and MPEG-4 (MP4). For more information, see *Adobe Flash Media Server Technical Overview*.

Set up a development environment

Install the server

You can use the free developer edition of the server for developing and testing applications. The easiest development environment has Flash or Flex installed on the same computer as the server.

Install the server

- ❖ Install Flash Media Development Server.

See *Adobe Flash Media Server Installation Guide* if you need detailed instructions.

Start the server

When you install the server, you can set it to start automatically when you boot your computer. If the server is not already started, you can start it manually.

- 1 From the Start menu, select All Programs > Adobe > Flash Media Server 3 > Start Adobe Flash Media Server 3.
- 2 From the Start menu, select All Programs > Adobe > Flash Media Server 3 > Start Flash Media Administration Server 3.

Note: You need the Administration Server if you want to open the Administration Console (for example, to view `server trace()` messages or connection counts).

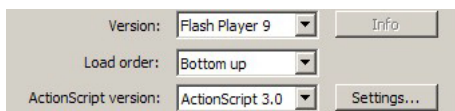
Verify that the server is running

- ❖ Open Control Panel > Administrative Tools > Services. In the Services window, make sure that both Flash Media Administration Server and Flash Media Server are started.

Install Flash

To build Flash interfaces that use ActionScript 3.0, you need Flash CS3, as well as Flash Player 9.

- 1 Download and install Adobe Flash CS3 Professional.
- 2 Download and install Flash Player 9.
- 3 Start Flash CS3 and Select File > New or File > Open to open a file.
- 4 Select File > Publish Settings.
- 5 On the Formats tab, make sure both Flash and HTML are selected.
- 6 On the Flash tab, for Version, select Flash Player 9, for ActionScript Version, select ActionScript 3.0.



Install Flex

To build Flex interfaces, you need Adobe Flex Builder or Adobe Flex SDK, as well as Flash Player 9.

- 1 Download and install the Adobe Flex 2 SDK or Adobe Flex Builder 2.
- 2 Download and install Flash Player 9.
- 3 In Flex Builder, make sure Project > Build Automatically is selected.

Hello World application

Overview

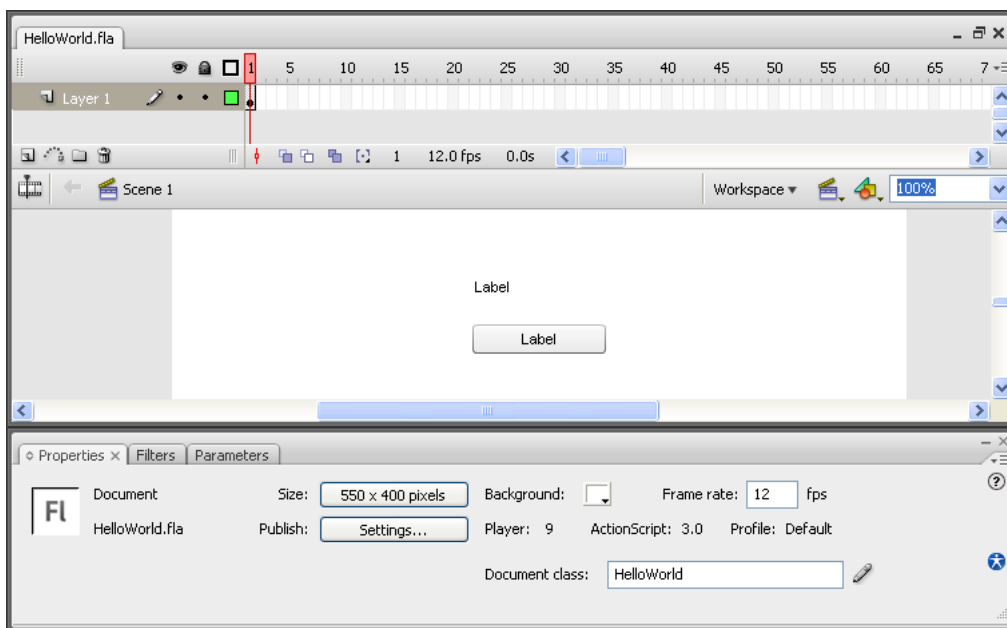
Note: The following sections apply to Flash Media Interactive Server and Flash Media Development Server.

This example uses Flash CS3 to show how to connect a Flash file to a server-side script and how to request information from the server. In this example, the Flash interface has one button (Connect) and one label (initially blank). When a user clicks the Connect button, the client connects to the server; then the client calls a server-side function to return a string value. When the server replies, the client's *responder* function displays the returned string in the label. The client continues by changing the button's label to Disconnect. When the Disconnect button is clicked, the client closes the connection and clears the label.

The example files are in the HelloWorld directory.

Create the user interface

- 1 Start Flash CS3 and select Create New > Flash File (ActionScript 3.0).
- 2 In the Document Class field, enter HelloWorld. You may see an ActionScript Class Warning message about a missing definition—click OK, as you will be adding the class file in the next section.
- 3 Choose Windows > Components and select User Interface > Button. On the Properties tab, name the Button `connectBtn`.
- 4 Add a Label component, move it up above the button, and name it `textLbl1`.
- 5 Save the file as HelloWorld.fla.



Write the client-side script

This script provides two button actions, either connecting to or disconnecting from the server. When connecting, the script calls the server with a string (“World”), which triggers a response that displays the returned string (“Hello, World!”).

- 1 Choose File > New > ActionScript File. Check that the Target field has HelloWorld.fla.

- 2 Declare the package and import the required Flash classes:

```
package {
    import flash.display.MovieClip;
    import flash.net.Responder;
    import flash.net.NetConnection;
    import flash.events.MouseEvent;
    public class HelloWorld extends MovieClip {
    }
}
```

- 3 Declare variables for the connection and the server responder (see the [ActionScript 3.0 Language and Components Reference](#)):

```
private var nc:NetConnection;
private var myResponder:Responder = new Responder(onReply);
```

- 4 Define the class constructor. Set the label and button display values, and add an event listener to the button:

```
public function HelloWorld() {
    textLbl.text = "";
    connectBtn.label = "Connect";
    connectBtn.addEventListener(MouseEvent.CLICK, connectHandler);
}
```

- 5 Define the event listener actions, which depend on the button's current label:

```
public function connectHandler(event:MouseEvent):void {
    if (connectBtn.label == "Connect") {
        trace("Connecting...");
        nc = new NetConnection();
        // Connect to the server.
        nc.connect("rtmp://localhost/HelloWorld");
        // Call the server's client function serverHelloMsg, in HelloWorld.asc.
        nc.call("serverHelloMsg", myResponder, "World");
        connectBtn.label = "Disconnect";
    } else {
        trace("Disconnecting...");
        // Close the connection.
        nc.close();
        connectBtn.label = "Connect";
        textLbl.text = "";
    }
}
```

- 6 Define the responder function (see the [ActionScript 3.0 Language and Components Reference](#)), which sets the label's display value:

```
private function onReply(result:Object):void {
    trace("onReply received value: " + result);
    textLbl.text = String(result);
}
```

- 7 Save the file as HelloWorld.as.

Write the server-side script

1 Choose File > New > ActionScript Communications File.

2 Define the server-side function and the connection logic:

```
application.onConnect = function( client ) {  
    client.serverHelloMsg = function( helloStr ) {  
        return "Hello, " + helloStr + "!";  
    }  
    application.acceptConnection( client );  
}
```

3 Create a HelloWorld folder in the *RootInstall/applications* folder.

4 Save the file as HelloWorld.asc in the *RootInstall/applications/HelloWorld* folder.

Compile and run the application

1 Verify that the server is running (see [Start the server](#)).

2 Select the HelloWorld.fla file tab.

3 Choose Control > Test Movie.

4 Click the Connect button.

“Hello, World!” is displayed, and the button label changes to Disconnect.

5 Click the Disconnect button.

The output of the `trace()` statements is displayed in the Output window in Flash CS3.

Create an application

Write the client-side code

A client has code written in ActionScript that connects to the server, handles events, and does other work. With Flash CS3, you can use ActionScript 3.0, 2.0, or 1.0, but ActionScript 3.0 offers many new features. With Flex, you must use ActionScript 3.0.

ActionScript 3.0 is significantly different from ActionScript 2.0. This guide assumes you are writing ActionScript 3.0 classes in external *.as* files, with a package name that corresponds to a directory structure in your development environment.

Create an ActionScript 3.0 class in Flash

1 If the ActionScript file is in the same directory as the corresponding FLA file, no package name is needed:

```
package {  
}
```

2 If you saved the file in a subdirectory below the FLA file, the package name must match the directory path to the *.as* file, for example:

```
package com.examples {  
}
```

3 Add import statements and a class declaration:

```
package {
```

```
import flash.display.MovieClip;

public class MyClass extends MovieClip {
}
}
```

The class name should match the filename without the `.as` extension. Your class might import or extend other ActionScript 3.0 classes, such as `MovieClip`.

You are now ready to start writing ActionScript code for Flash Media Interactive Server.

Create an ActionScript 3.0 class in Flex

- 1 Start Adobe Flex Builder.
- 2 Create a new project. Choose File > New > ActionScript Project and follow the wizard.

If you created an ActionScript project, an ActionScript 3.0 file opens, with package and class declarations already filled in:

```
package {
    import flash.display.Sprite;

    public class Test extends Sprite
    {
        public function TestAs()
        {
        }
    }
}
```

- 3 (Optional) If you created a Flex project, choose File > New > ActionScript Class.

Writing server-side code

In general, applications require server-side code written in Server-Side ActionScript if they need to do any of the following:

Authenticate clients By user name and password, or by credentials stored in an application server or database.

Implement connection logic By taking some action when a client connects or disconnects.

Update clients By calling remote methods on clients or updating shared objects that affect all connected clients.

Handle streams By allowing you to play, record, and manage streams sent to and from the server.

Connect to other servers By calling a web service or creating a network socket to an application server or database.

Extend the server Using the access, authorization, or file adaptors.

Place the server-side code in a file named `main.asc` or `yourApplicationName.asc`, where `yourApplicationName` is a registered application name with Flash Media Interactive Server. To register an application with the server, create a folder in the `RootInstall/applications` folder with the application name. For example, to register an application called `skatingClips`, create the folder `RootInstall/applications/skatingClips`. The server-side code would be in a file called `main.asc` or `skatingClips.asc` in the `skatingClips` folder.

To configure the location of the applications directory, edit the `fms.ini` or the `Vhost.xml` configuration file; see *Adobe Flash Media Server Configuration and Administration Guide*.

The server-side code goes at the top level of the application directory, or in its `scripts` subdirectory. For example, you can use either of these locations:

RootInstall/applications/sudoku

RootInstall/applications/sudoku/scripts

Client and application objects

Server-side scripts have access to two special objects, the `client` object and the `application` object. When a client connects to an application on Flash Media Server, the server creates an instance of the server-side `Client` class to represent the client. An application can have thousands of clients connected. In your server-side code, you can use the `client` object to send and receive messages to individual clients.

Each application also has a single `application` object, which is an instance of the server-side `Application` class. The `application` object represents the application instance. You can use it to accept clients, disconnect them, shut down the application, and so on.

Writing double-byte applications

If you use Server-Side ActionScript to develop an application that uses double-byte text (such as an Asian language character set), place your server-side code in a `main.asc` file that is UTF-8 encoded. This means you'll need a JavaScript editor, such as the Script window in Flash or Dreamweaver, that encodes files to the UTF-8 standard. Then, you can use built-in JavaScript methods, such as `Date.toLocaleString()`, which converts the string to the locale encoding for that system.

Some simple text editors might not encode files to the UTF-8 standard. However, some editors, such as Microsoft Notepad for Windows XP and Windows 2000, provide a Save As option to encode files in the UTF-8 standard.

Set UTF-8 encoding in Dreamweaver

- 1 Check the document encoding setting by selecting **Modify > Page Properties**, then **Document Encoding**. Choose **Unicode (UTF-8)**.
- 2 Change the inline input setting by selecting **Edit > Preferences (Windows)** or **Dreamweaver > Preferences (Mac)**, and then click **General**. Select **Enable Double-Byte Online Input** to enable double-byte text.

Use double-byte characters as method names

- ❖ Assign method names using the object array operator, not the dot operator:

```
// This is the CORRECT way to create double-byte method names
obj["Any_hi_byte_name"] = function() {}

// This is the INCORRECT way to create double-byte method names.
obj.Any_hi_byte_name = function() {}
```

Test an application

Test and debug a server-side script

To test a server-side script, use `trace()` statements to monitor each processing point.

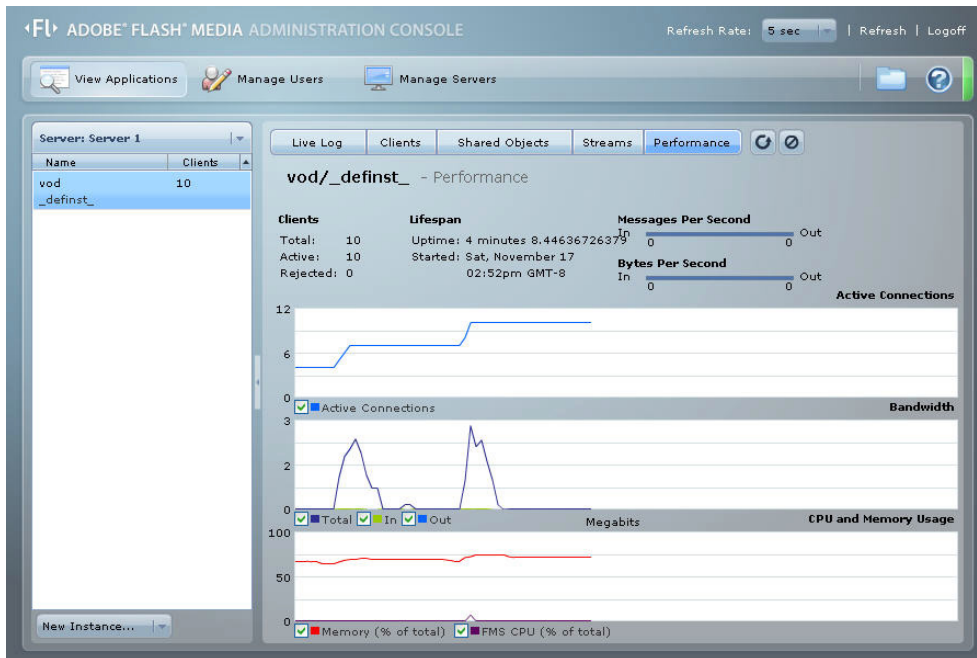
Use the Administration Console to start, stop, reload, and view applications, with **Start > All Programs > Adobe Flash Media Server 3 > Flash Media Administration Console**.

When your client connects to the server, that application is loaded and can be seen in the Administration Console. To load an application directly from the Administration Console, select from the **New Instance** list of available application names. You can also stop an application or reload it—in either case, all clients are disconnected.

Note: When you edit and save an `.asc` file, the changes will not take effect until that application is restarted. If the application is already running, use the Administration Console to close it, then connect to the application again.

For each application instance, you can observe its live log, clients, shared objects, if any, streams in use, and performance statistics.

This is an example of checking the performance of an application while it is running.



View the output of a server-side script

The output of the `trace()` statements used by a `main.asc` file are sent to a log file, typically:

`RootInstall/logs/_defaultVHost_/yourApplicationName/yourInstanceName/application.xx.log`

Where `yourInstanceName` is `_definst_` by default, and `xx` is the instance number, 00 for the most recent log file, 01 for the previous instance, and so forth.

You can view a log file with any text editor.

While an application is running, you can view its live log file from the Administration Console:

- 1 From the Windows desktop, click Start > All Programs > Adobe Flash Media Server 3 > Flash Media Administration Console.
- 2 When the Administration Console opens, click View Applications, then Live Log.

Debug with the Administration Console

The availability and number of debugging sessions is set in `Application.xml` with the `AllowDebugDefault` and `MaxPendingDebugConnections` parameters. By default, debugging is disallowed. For details, see *Adobe Flash Media Server Configuration and Administration Guide*.

You can override the debug setting in `Application.xml` by adding the following line to an application's server-side code:

```
application.allowDebug = true;
```

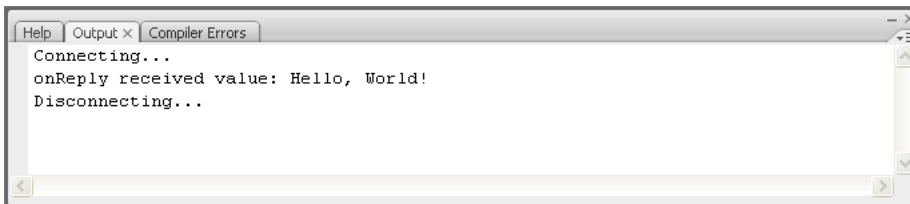
Note: If used, set this value to false before deploying the application.

To start a debugging session:

- 1 Start Flash Media Server and Flash Media Administration Server.
- 2 Start the Administration Console.
- 3 Log in to the Administration Server with the username and password set during installation.
- 4 Start your application on Flash Media Server.
- 5 Select the application to debug from the list in the Administration Console.
- 6 Press the Streams button to see the list of playing streams, if any.
- 7 Click on one of the streams.
- 8 Press the Play Stream button.
- 9 A pop-up window will open and the stream will play.
- 10 Press the Shared Objects button to see the application's shared objects, if any.
- 11 Select a shared object.
- 12 Press the Close Debug button to end the debug session.

Test and debug a client-side script

To help test a client-side script, use `trace()` statements to monitor each processing point. The output is shown in the Flash CS3 Window > Output window (this example is from [Hello World application](#)):



To debug a client-side script, use the Debug menu in Flash CS3 to set breakpoints, step into functions, and so forth. You can inspect the state of the script with Windows > Debug Panels.

Deploy an application

Copy server-side files and assets to Flash Media Server

Applications that connect to Flash Media Server must be registered with the server so the server knows who they are when they try to connect. To register an application with the server, create a folder for the application in the main applications folder, which is located in the root installation folder by default (Flash Media Server 3/applications). To create instances of an application, create subfolders within that application's folder (for example, Flash Media Server 3/applications/exampleApplication/instance1). Copy the server-side script files and assets (such as streams) to the destination server's applications directory:

RootInstall/applications/YourApplicationName

RootInstall/applications/YourApplicationName/scripts

RootInstall/applications/YourApplicationName/YourInstanceName/

RootInstall/applications/YourApplicationName/streams/YourInstanceName/

By default, Flash Media Server searches the above directories when an application is requested.

Note: To replace a running application, copy the new files over, then use the Administration Console to restart the application.

For example, to run the Streams sample in the *RootInstall/documentation/samples* directory, you first need to copy the Streams folder to the applications directory, as follows: *RootInstall/applications/Streams*. To run the StreamLength sample, copy the StreamLength folder to the application directory, as follows: *RootInstall/applications/StreamLength*. Ensure the folder contains the main.asc file and the streams subdirectory. (The AS and FLA are source files and do not need to reside in this directory, and the SWF file can be run from anywhere.)

Packaging server-side files

Flash Media Server includes a command-line archive compiler utility, *far.exe*, which lets you package server-side scripts into a FAR file, which is an archive file like a ZIP file, to simplify deployment. You can also use the archive compiler utility to compile server-side script files to bytecode (with the file extension *.ase*) to speed the time required to load an application instance.

A large application can contain multiple server-side script files stored in different locations. Some files are located in the application directory and others are scattered in the script library paths that are defined in the server configuration file. To simplify deployment of your media application, you can package your server-side JS, ASC, and ASE files in a self-contained Flash Media Server archive file (a FAR file).

The FAR file is a package that includes the main script file (which is either *main.js*, *main.asc*, *main.ase*, *applicationName.js*, *applicationName.asc*, or *applicationName.ase*) and any other script files that are referred to in the main script.

The syntax for running the archive compiler utility to create a script package is as follows:

```
c:\> far -package -archive <archive> -files <file1> [<file2> ... <fileN>]
```

The following table describes the command-line options available for *far -package*.

Option	Description
-archive <i>archive</i>	Specifies the name of the archive file, which has a <i>.far</i> extension.
-files <i>file1</i> [<i>file2</i> ... <i>fileN</i>]	Specifies the list of files to be included in the archive file. At least one file is required.

Note: If the main script refers to scripts in a subdirectory, the hierarchy must be maintained in the archive file. To maintain this hierarchy, Adobe recommends that you run the FAR utility in the same directory where the main script is located.

Copy client-side files to a web server

Copy SWF files to a web server.

Chapter 2: Streaming services

All editions of Flash Media Server 3 provide two streaming services, live (live video) and vod (video on demand). These services are implemented as server-side components of Flash Media Server applications. Sample clients for both services are installed with the server. You can modify the sample clients for production use or create your own.

The live and vod services are signed (approved) by Adobe. Flash Media Streaming Server only supports signed services—it cannot run other applications. Flash Media Interactive Server and Flash Media Development Server support signed services as well as any other applications you create.

Both services can be duplicated and renamed as needed to create multiple publishing points and to provision customers. There is no limit to the number of service instances a server can support.

Using the live service

About the live service

The live service is a publishing point on Flash Media Server. You can use Flash Media Encoder to capture, encode, and stream live video to the live service and play the video with the sample client or with the FLVPlayback component. You can also build your own application to capture video and your own client application to play the video.

The following live video sources can publish to the live service:

- [Flash Media Encoder 2.0](#)

Note: Only Flash Media Interactive Server and Flash Media Development Server support Flash Media Encoder Authentication Add-in.

- Flash Media Interactive Server and Flash Media Development Server (see [Publish from server to server](#))
- A custom-built Flash Player application that records audio and video

Test the live service

1 Connect a camera to the computer.

2 Open Flash Media Encoder and click Start.

By default, Flash Media Encoder publishes a stream to the default live publishing point on the same computer, `rtmp://localhost/live`, and publishes a stream named `livestream`.

3 Double-click the `RootInstall/samples/applications/live/livetest.html` file to open the client application in a browser and see the live stream.

To use the FLVPlayback component as the client, set the `contentPath` parameter to the URL of the publishing point and the stream name (`rtmp://localhost/live/livestream`, by default) and set the `isLive` parameter to `true`.

Modify the live service

1 Duplicate the *RootInstall/applications/live* folder in the applications folder and give it a new name, for example, *live2*. In this case, the new live service is located here: *RootInstall/applications/live2*.

You can create as many instances of the live service as you need.

2 Open the *fms.ini* file (located in *RootInstall/conf*) and add a new entry to set the content path for the new service, for example, `LIVE2_DIR = C:\Program Files\Adobe\Flash Media Server 3\applications\live2`.

3 Open the *Application.xml* file in *RootInstall/applications/live2* and change the virtual directory entry to `<Streams>/;${LIVE2_DIR}</Streams>`.

4 Restart Flash Media Server.

5 Clients can connect to the publishing point at the URL `rtmp://flashmediaserver/live2/streamname`.

If the client is the FLVPlayback component, set the `contentPath` parameter to the URL of the publishing point plus the stream name and set the `isLive` parameter to `true`.

Disable live services

❖ Move any live services folders out of the applications folder.

Using the vod service

About the vod service

The vod (video on demand) service lets you stream recorded media without building an application or configuring the server. You can use the Flash CS3 and Flash 8 FLVPlayback components as clients. Copy MP4, FLV, and MP3 files into the vod application's media folder to stream the media to clients.

Test the vod service

1 Do one of the following:

- Double-click the *RootInstall/samples/applications/vod/vodtest.swf* file to open a client in the stand-alone Flash Player.
- Double-click the *RootInstall/samples/applications/vod/vodtest.html* file to open a client in Flash Player in a web browser.

2 Click Go.

3 (Optional) To play other versions of the sample file, select the file from the pop-up menu and click Go.

Modify the vod service

1 Duplicate the *RootInstall/applications/vod* folder in the applications folder and give it a new name, for example, *vod2*. In this case, the new vod service is located here: *RootInstall/applications/vod2*.

You can create as many instances of the vod service as you need.

2 Open the *fms.ini* file (located in *RootInstall/conf*) and add a new entry to set the content path for the new service, for example, `VOD2_DIR = C:\Program Files\Adobe\Flash Media Server 3\applications\vod2\media`.

3 Open the `Application.xml` file in the `RootInstall/applications/vod2` folder and add `VOD2_DIR` to the virtual directory entry list: `<Streams>/;${VOD2_DIR}</Streams>`.

4 Restart Adobe Flash Media Server.

5 Place recorded media files into the folder you specified in the `fms.ini` file (in this example, `C:\Program Files\Adobe\Flash Media Server 3\applications\vod2\media`).

The media files are now accessible from the URL `rtmp://flashmediaserver/vod2/filename`.

Note: You do not have to specify the media subdirectory in the URL; the media directory is specified in the path you set in the `fms.ini` file.

Disable vod services

- ❖ Move any vod service folders you want to disable out of the applications folder.

Creating clients for streaming services

Creating client applications

Start with the provided sample client code (`RootInstall/samples/applications/live` and `RootInstall/samples/applications/vod`) and modify it as desired.

Clients for the vod and live services can use any Flash Player features except the following:

- Recording live streams (`NetStream.publish("streamName", "record")`).
- Remote shared objects (`SharedObject.getRemote()`).

Using the FLVPlayback component

You can use the Flash 8 FLVPlayback component and the Flash CS3 FLVPlayback component as clients for the vod and live services. Set the `contentPath` parameter to the URL of the stream and, if you're connecting to the live service, set the `isLive` parameter to `true`.

Connecting to a streaming service

The streaming services expect the incoming URI to be in the following format:

```
rtmp://hostName/serviceName/instanceName/[formatType:]fileOrStreamName
```

`hostName` The Flash Media Server domain name.

`serviceName` Either `live` or `vod`.

`instanceName` If the client is connecting to the default instance, you can either omit the instance name or use `_definst_`. If the client is connecting to an instance you have created, such as `room1`, use that name.

`formatType` One of the supported file formats, `flv:`, `mp3:` or `mp4:`. The default format if unspecified is `flv:`.

`fileOrStreamName` Either a file name (for example, `my_video.mp4`) or a pathname (for example, `subdir/subdir2/my_video.mp4`). If the file is an FLV or MP3 file, you do not need to specify the file format. If the file is an MP4 file, you must specify the file format, for example,

```
rtmp://www.examplemediaserver.com/vod/ClassicFilms/mp4:AnOldMovie.mp4.
```

Allow connections from specific domains

By default, clients can connect to the live and vod services from any domain. To limit the domains from which clients can connect, edit text files in the service's applications folder.

- ❖ Navigate to the *RootInstall/applications/live* or *RootInstall/applications/vod* folder and do one of the following:
 - To add a domain for SWF clients, edit the `allowedSWFdomains.txt` file.
 - To add a domain for HTML clients, edit the `allowedHTMLdomains.txt` file.

The TXT files contain detailed information about adding domains.

Chapter 3: Developing media applications

Video applications for Adobe® Flash® Media Interactive Server can be *video on demand* or *live video* applications. Video on demand applications stream recorded video from the server, such as television shows, commercials, or user-created video stored on the server. An organization may have a large archive of videos or be producing new videos regularly. The videos can be short clips (0–30 seconds), long clips (30 seconds to 5 minutes), or very long clips (5 minutes to hours long).

Live video applications stream live video from the server to users, or from one user to the server and then on to other users. Live video is typically used for live events, such as corporate meetings, education, sports events, and concerts, or delivered continually, for example, by a television or radio station. You can use Adobe Flash Media Encoder, available from Adobe.com, to encode and stream live video.

Connecting to the server

The NetConnection class

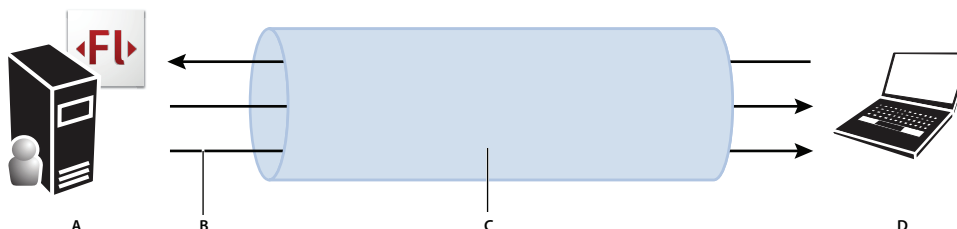
Before a client can play audio and video from Flash Media Server, it must connect to the server. The connection request is accepted or rejected by an application instance on the server, and connection messages are sent back to the client. Once the application accepts the connection request, a connection is available to both the client and the server.

The NetConnection class connects a client to an application instance on the server. In the simplest case, you can connect by creating an instance of NetConnection and then calling the `connect()` method with the URI to an application instance:

```
var nc:NetConnection = new NetConnection();
nc.connect("rtmp://localhost/HelloServer");
```

Streams handle the flow of audio, video, and data over a network connection. A NetConnection object is like a pipe that streams audio, video, and data from client to server, or from server to client. Once you create the NetConnection object, you can attach one or more streams to it.

A stream can carry more than one type of content (audio, video, and data). However, a stream flows in only one direction, from server to client or client to server.



Many streams can use one NetConnection object between client and server.

A. Flash Media Server B. Single stream of data C. NetConnection D. Flash Player, AIR, or Flash Lite 3 client

The application URI

The URI to the application can be absolute or relative and has the following syntax (items in brackets are optional):

```
protocol: [//host] [:port] /appname/ [instanceName]
```

The parts of the URI are described in the following table.

Part	Example	Description
protocol:	rtmp:	The protocol used to connect to Adobe Flash Media Server, which is the Adobe Real-Time Messaging Protocol. Possible values are rtmp, rtmpe, rtmpe, rtmpt, and rtmpte. For more information, see the <i>Technical Overview</i> .
//host	//www.example.com //localhost	The host name of a local or remote computer. To connect to a server on the same host computer as the client, use //localhost or omit the //host identifier.
:port	:1935	The port number to connect to on Adobe Flash Media Server. If the protocol is rtmp, the default port is 1935 and you don't need to specify the port number.
/appname/	/sudoku/	The name of a subdirectory in <i>RootInstall/applications</i> , where your application files reside. You can specify another location for your applications directory in the <i>fms.ini</i> configuration file (at <i>RootInstall/conf/fms.ini</i>).
instanceName	room1	An instance of the application to which the client connects. For example, a chat room application can have many chat rooms: <i>chatroom/room1</i> , <i>chatroom/room2</i> , and so on. If you do not specify an instance name, the client connects to the default application instance, named <i>_definst_</i> .

The only parts of the URI that are required are the protocol and the application name, as in the following:

```
rtmp://www.example.com/sudoku/
```

In the following example, the client is on the same computer as the server, which is common while you are developing and testing applications:

```
rtmp:/sudoku/room1
```

Mapping URIs to local and network drives

Flash Media Server simplifies the mapping of URIs to local and network drives by using *virtual directories*. Virtual directories let you publish and store media files in different, predetermined locations, which can help you organize your media files. Configure virtual directories in the *VirtualDirectory/Streams* tag of the *Vhost.xml* file. For more information, see [Mapping virtual directories to physical directories](#) in the *Configuration and Administration Guide*.

One way you can use directory mapping is to separate storage of different kinds of resources. For example, your application could allow users to view either high-bandwidth video or low-bandwidth video, and you might want to store high-bandwidth and low-bandwidth video in separate folders. You can create a mapping wherein all streams that start with *low* are stored in a specific directory, *C:\low_bandwidth*, and all streams that start with *high* are stored in a different directory:

```
<VirtualDirectory>
  <Streams>low;c:\low_bandwidth</Streams>
  <Streams>high;c:\high_bandwidth</Streams>
</VirtualDirectory>
```

When the client wants to access low-bandwidth video, the client calls `ns.play("low/sample")`. This call tells the server to look for the `sample.flv` file in the `c:\low_bandwidth` folder.

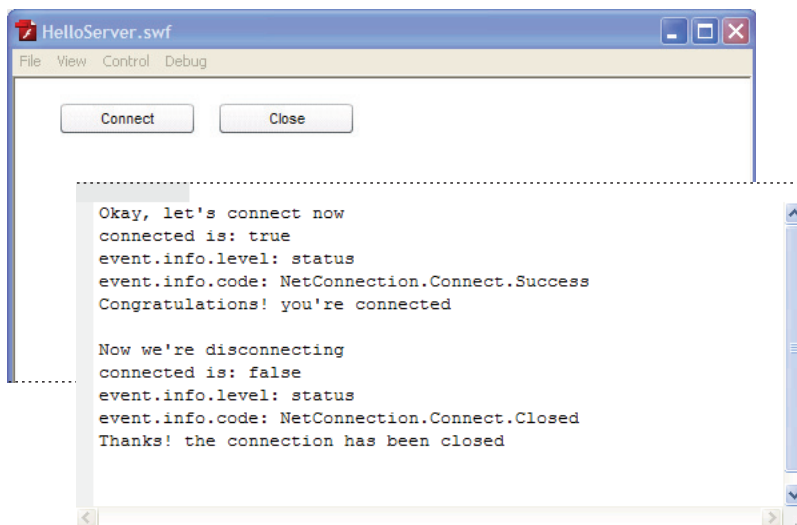
Similarly, a call to `ns.play("high/sample")` tells the server to look for the `sample.flv` file in the `c:\high_bandwidth` folder.

The following table shows three examples of different virtual directory configurations, including mapping to a local drive and a network drive, and how the configurations determine the directory to which a recorded stream is published. In the first case, because the URI specified ("myStream") does not match the virtual directory name that is specified ("low"), the server publishes the stream to the default streams directory.

Mapping in Vhost.xml <VirtualDirectory><Streams> tag	URI in NetStream call	Location of published stream
low:e:\fmsstreams	"myStream"	c:\...\ <i>RootInstall</i> \applications\yourApp\streams_definst_\myStream.flv
low:e:\fmsstreams	"low/myStream"	e:\fmsstreams\myStream.flv
low;\mynetworkDrive\share\fmsstreams	"low/myStream"	\\mynetworkDrive\share\fmsstreams\myStream.flv

Hello Server application

You can find the HelloServer application in the documentation/samples/HelloServer directory in the root install directory. This simple Flash application displays two buttons that enable you to connect to the server and close the connection. The output window displays messages about the connection status.



Run the application

The easiest way to run the sample is to install it on the same computer as the server.

- 1 Copy the HelloServer folder from the documentation/samples directory in the Flash Media Server root install directory to a location on your client computer.
- 2 Register the application by creating a folder in the server's applications folder:
RootInstall/applications/HelloServer
- 3 (Optional) To run the sample on a server installed on a different computer, open HelloServer.as and edit this line to add the URL to your server:

```
nc.connect("rtmp://localhost/HelloServer");
```

See ["Connecting to the server"](#) for details on how to construct a URL.

Design the Flash user interface

The sample is already built and included in the samples folder. However, these instructions show you how to recreate it, so that you can build it on your own and add to it.

- 1 In Adobe Flash CS3 Professional, choose File > New > Flash File (ActionScript 3.0), and click OK.
- 2 Choose Window > Components to open the Components panel.
- 3 Click the Button component and drag it to the Stage.
- 4 In the Properties Inspector, click the Properties tab. Select MovieClip as the instance behavior, and enter the instance name **connectBtn**.
- 5 Click the Parameters tab, then Label. Enter **Connect** as the button label.
- 6 Drag a second button component to the Stage.
- 7 Give the second button the instance name **closeBtn** and the label **Close**.
- 8 Save the FLA file, naming it HelloServer.fla.

Write the client-side code

You can find the complete ActionScript sample in HelloServer.as in the documentation/samples/HelloServer directory in the Flash Media Server root install directory. While you develop ActionScript 3.0 code, refer to the [ActionScript 3.0 Language and Components Reference](#).

- 1 In Adobe Flash CS3 Professional, choose File > New > ActionScript File, and click OK.
- 2 Save the ActionScript file with a name that begins with a capital letter and has the extension *.as*, for example, HelloServer.as.
- 3 Return to the FLA file. Choose File > Publish Settings. Click the Flash tab, then Settings.
- 4 In the Document Class box, enter HelloServer. Click the green check mark to make sure the class file can be located.
- 5 Click OK, then OK again.
- 6 In the ActionScript file, enter a package declaration. If you saved the file to the same directory as the FLA file, do not use a package name, for example:

```
package {  
}
```

However, if you saved the file to a subdirectory below the FLA file, the package name must match the directory path to your ActionScript file, for example:

```
package samples {  
}
```

- 7 Within the package, import the ActionScript classes you need:

```
import flash.display.MovieClip;  
import flash.net.NetConnection;  
import flash.events.NetStatusEvent;  
import flash.events.MouseEvent;
```

- 8 After the `import` statements, create a class declaration. Within the class, define a variable of type `NetConnection`:

```
public class HelloServer extends MovieClip {  
    private var nc:NetConnection;  
}
```

Be sure the class extends `MovieClip`.

- 9 Write the class constructor, registering an event listener on each button:

```
public function HelloServer() {
    // register listeners for mouse clicks on the two buttons
    connectBtn.addEventListener(MouseEvent.CLICK, connectHandler);
    closeBtn.addEventListener(MouseEvent.CLICK, closeHandler);
}
```

Use `addEventListener()` to call an event handler named `connectHandler()` when a click `MouseEvent` occurs on the Connect button. Likewise, call `closeHandler()` when a click `MouseEvent` occurs on the Close button.

- 10 Write the `connectHandler()` function to connect to the server when a user clicks the Connect button:

```
public function connectHandler(event:MouseEvent):void {
    trace("Okay, let's connect now");
    nc = new NetConnection();
    nc.addEventListener(NetStatusEvent.NET_STATUS, netStatusHandler);
    nc.connect("rtmp://localhost/HelloServer");
}
```

In `connectHandler()`, add an event listener to listen for a `netStatus` event returned by the `NetConnection` object. Then, connect to the application instance on the server by calling `NetConnection.connect()` with the correct URI. This URI connects to an application instance named *HelloServer*, where the server runs on the same computer as the client.

- 11 Write the `closeHandler()` function to define what happens when a user clicks the Close button:

```
public function closeHandler(event:MouseEvent):void {
    trace("Now we're disconnecting");
    nc.close();
}
```

It's a best practice to explicitly call `close()` to close the connection to the server.

- 12 Write the `netStatusHandler()` function to handle `netStatus` objects returned by the `NetConnection` object:

```
public function netStatusHandler(event:NetStatusEvent):void {
    trace("connected is: " + nc.connected);
    trace("event.info.level: " + event.info.level);
    trace("event.info.code: " + event.info.code);
    switch (event.info.code)
    {
        case "NetConnection.Connect.Success":
            trace("Congratulations! you're connected" + "\n");
            break;
        case "NetConnection.Connect.Rejected":
            trace("Oops! the connection was rejected" + "\n");
            break;
        case "NetConnection.Connect.Closed":
            trace("Thanks! the connection has been closed" + "\n");
            break;
    }
}
```

A `netStatus` object contains an `info` object, which in turn contains a `level` and a `code` that describes the connection status.

Understand the connection messages

When you run the sample and click the Connect button, you see messages like this, as long as the connection is successful:

```
Okay, let's connect now
connected is: true
event.info.level: status
event.info.code: NetConnection.Connect.Success
Congratulations! you're connected
```

The line `connected is: true` shows the value of the `NetConnection.connected` property, meaning whether Flash Player is connected to the server over RTMP. The next two lines describe the `netStatus` event the `NetConnection` object sends to report its connection status:

```
event.info.level: status
event.info.code: NetConnection.Connect.Success
```

The `level` property can have two values, `status` or `error`. The `code` property describes the status of the connection. You can check for various `code` values in your `netStatusHandler` function and take actions. Always check for a successful connection before you create streams or do other work in your application.

Likewise, when you click the Close button, you see the following:

```
Now we're disconnecting
connected is: false
event.info.level: status
event.info.code: NetConnection.Connect.Closed
Thanks! the connection has been closed
```

Managing connections

Connection status codes

Once the connection between client and server is made, it can break for various reasons. The network might go down, the server might stop, or the connection might be closed from the server or the client. Any change in the connection status creates a `netStatus` event, which has both a `code` and a `level` property describing the change. This is one `code` and `level` combination:

Code	Level	Meaning
<code>NetConnection.Connect.Success</code>	<code>status</code>	A connection has been established successfully.

See `NetStatus.info` in the [ActionScript 3.0 Language and Components Reference](#) for a complete list of all code and level values that can be returned in a `netStatus` event.

When the event is returned, you can access the connection code and level with `event.info.code` and `event.info.level`. You can also check the `NetConnection.connected` property (which has a value of `true` or `false`) to see if the connection still exists. If the connection can't be made or becomes unavailable, you need to take some action from the application client.

Managing connections in server-side code

An application might also have server-side code in a `main.asc` or `applicationName.asc` file that manages clients trying to connect (see [Writing server-side code](#) for an introduction).

The server-side code has access to `client` objects, which represent individual clients on the server side, and a single `application` object, which enables you to manage the application instance. In the server code, you use Server-Side ActionScript and the server-side information objects (see the [Server-Side ActionScript Language Reference](#)).

In the server-side code, the application can accept or reject connections from clients, shut down the application, and perform other tasks to manage the connection. When a client connects, the application receives an `application.onConnect` event. Likewise, when the client disconnects, the application receives an `application.onDisconnect` event.

To manage the connection from the server, start with `application.onConnect()` and `application.onDisconnect()` in Server-Side ActionScript.

Managing connections sample application

This example shows how to manage connections from both the application client and the server-side code.

Write the client code

In the client code, you need to check for specific connection codes and handle them. Create live streams or play recorded streams only when the client receives `NetConnection.Connect.Success`. When the client receives `NetConnection.Connect.AppShutDown`, all streams from server to client or client to server are shut down. In that case, close the connection to the server.

Note: See the *SimpleConnectManage sample*, *SimpleConnectManage.as*, written in ActionScript 3.0.

- 1 Create a `NetConnection` object and call the `connect()` method to connect to the server.
- 2 Write a `netStatus` event handler. In it, check for specific connection codes, and take an action for each:

```
public function netStatusHandler(event:NetStatusEvent):void
{
    trace("connected is: " + nc.connected);
    trace("event.info.level: " + event.info.level);
    trace("event.info.code: " + event.info.code);
    switch (event.info.code)
    {
        case "NetConnection.Connect.Success":
            trace("Congratulations! you're connected");
            // create live streams
            // play recorded streams
            break;
        case "NetConnection.Connect.Rejected":
            trace("Oops! the connection was rejected");
            // try to connect again
            break;
        case "NetConnection.Connect.Failed":
            trace("The server may be down or unreachable");
            // display a message for the user
            break;
        case "NetConnection.Connect.AppShutDown":
            trace("The application is shutting down");
            // this method disconnects all stream objects
            nc.close();
            break;
        case "NetConnection.Connect.Closed":
            trace("The connection was closed successfully - goodbye");
            // display a reconnect button
            break;
    }
}
```

Run the code

Note: These instructions apply to any ActionScript 3.0 example without a Flash user interface in this guide. The ActionScript 3.0 examples are provided for your convenience.

- 1 Check the client-side code to see which application it connects to:

```
nc.connect("rtmp://localhost/HelloServer");
```

- 2 Register the application on the server by creating an application instance directory for it in the applications directory, for example:

```
RootInstall/applications/HelloServer
```

- 3 (Optional) Or, to use an application you already have registered with the server, change the URI used in the call to `connect()`:

```
nc.connect("rtmp://localhost/MyApplication");
```

- 4 In Adobe Flex Builder or Eclipse with the Flex Builder plug-in, create an ActionScript project named SimpleConnectManage (choose File > New > ActionScript Project, and follow the wizard).
- 5 Add the SimpleConnectManage sample files to the project.
- 6 Choose Run > Debug. In the Debug window, enter SimpleConnectManage for Project and SimpleConnectManage.as for Application file. Click Debug.
- 7 Close the empty application window that opens, and return to Flex Builder or Eclipse. Check the messages in the Console window.

If the connection is successful, you should see output like this:

```
connected is: true
event.info.level: status
event.info.code: NetConnection.Connect.Success
Congratulations! you're connected
[SWF] C:\samples\SimpleConnectManage\bin\SimpleConnectManage-debug.swf - 2,377 bytes
after decompression
```

Recorded streams

Playing recorded streams

One of the most popular uses of Adobe Flash Media Server is to stream recorded audio and video files that are stored on the server to many clients.

To play a recorded stream, pass a URI to `NetStream.play()` to locate the recorded file, as in the following:

```
ns.play("bikes");
```

This line specifies the recorded stream named `bikes.flv` within the application to which you are connected with `NetConnection.connect()`. Briefly, the `play()` method takes four parameters, with this syntax:

```
public function play( name:Object [,start:Number [,len:Number [,reset:Object] ] ] ):void
```

name	The name of a recorded file.
------	------------------------------

start	The time from the start of the video at which to start play, in seconds.
len	The duration of the playback, in seconds.
reset	Whether to clear any previous <code>play()</code> calls from a playlist.

These parameters are described in detail in `NetStream.play()` in the [ActionScript 3.0 Language and Components Reference](#).

Capturing video snapshots

This feature enables you to get a thumbnail snapshot of a given video, including sound, for display purposes.

Flash Player clients are permitted to access data from streams in the directories specified by the `Client.audioSampleAccess` and `Client.videoSampleAccess` properties. See the [ActionScript 3.0 Language and Components Reference](#).

To access data, call `BitmapData.draw()` and `SoundMixer.computeSpectrum()` on the client—see “Accessing raw sound data” in *Programming ActionScript 3.0*.

Handling metadata in the stream

A recorded media file often has metadata encoded in it by the server or a tool. The Flash Video Exporter utility (version 1.1 or later) is a tool that embeds a video’s duration, frame rate, and other information into the video file itself. Other video encoders embed different sets of metadata, or you can explicitly add your own metadata (see “[Add metadata to a live stream](#)”).

The `NetStream` object that plays the stream on the client dispatches an `onMetaData` event when the stream encounters the metadata. To read the metadata, you must handle the event and extract the `info` object that contains the metadata. For example, if a file is encoded with Flash Video Exporter, the `info` object contains these properties:

duration	The duration of the video.
width	The width of the video display.
height	The height of the video display.
framerate	The frame rate at which the video was encoded.

See “[Add metadata to a live stream](#)” for a list of property names Adobe suggests for users adding metadata to live video streaming from client to server.

Video player example

If you have built a Flash video player interface for progressive download video, you may have used the `FLVPlayback` component to design the video player interface. This tutorial uses a different technique: adding the `Video` object to the `Stage` using `ActionScript 3.0`.

Note: For this tutorial, use the *Streams sample*, `Streams.as`, from the `root_installation_folder_documentation/samples` folder.

To use the `FLVPlayback` component with Flash, see:

- [Adobe Creative Suite 3 Video Workshop](#), “Creating a Video Application with Components”
- [The HTML tutorial](#), “Creating a Video Application with Components”

Run the sample in Flash

The easiest way to run the sample is to install it on the same computer as your development server.

- 1 Place the Streams.as file in a sample directory for client applications.
- 2 Register the application by creating a directory for it in your server installation:

```
RootInstall/applications/Streams
```

- 3 Copy the contents of the Streams sample folder (including Streams.fla) to the Streams application directory. You should then have a directory named streams/_definst_ that contains the video file, bikes.flv:

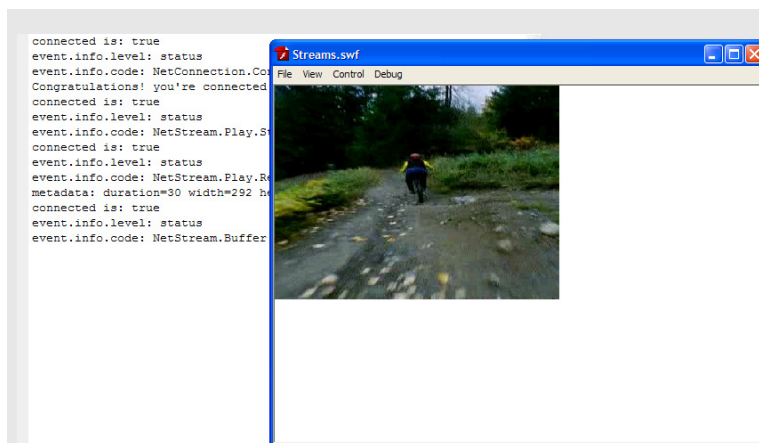
```
RootInstall/applications/Streams/streams/_definst_/bikes.flv
```

- 4 (Optional) To run the sample on a server installed on a different computer, open Streams.as and edit this line to add the URI to the application instance on your server:

```
nc.connect("rtmp://localhost/Streams");
```

See [“Connecting to the server”](#) for details on how to construct the URL.

- 5 In Flash CS3, open the copy of Streams.fla that is in the Flash Media Server applications directory.
- 6 Select Control > Test Movie. The video plays (with no sound) and the output window displays:



You can watch the output as the stream plays and the connection status changes. The call to `NetStream.play()` triggers the call to `onMetaData`, which displays metadata in the console window, like this:

```
metadata: duration=30 width=292 height=292 framerate=30
```

Run the sample in Flex

- 1 Open Streams.as in Flex Builder or Eclipse with the Flex Builder plug-in.
- 2 Choose Run > Debug. For Project, choose Streams. For Application file, choose Streams.as.
- 3 Click Debug.

An application window opens in which the video runs. Click the Flex Builder window to see the output messages. The full output looks like this:

```
connected is: true
event.info.level: status
event.info.code: NetConnection.Connect.Success
Congratulations! you're connected
```

```

connected is: true
event.info.level: status
event.info.code: NetStream.Play.Reset
connected is: true
event.info.level: status
event.info.code: NetStream.Play.Start
metadata: duration=30 width=292 height=292 framerate=30
[SWF] C:\samples\Streams\bin\Streams-debug.swf - 3,387 bytes after decompression
connected is: true
event.info.level: status
event.info.code: NetStream.Buffer.Full
connected is: true
event.info.level: status
event.info.code: NetStream.Play.Stop
The stream has finished playing
connected is: true
event.info.level: status
event.info.code: NetStream.Buffer.Flush
handling playstatus here
connected is: true
event.info.level: status
event.info.code: NetStream.Buffer.Empty

```

Write the main client class

Note: See the Streams sample, *Streams.as*, in *ActionScript 3.0*.

- 1 Create an ActionScript 3.0 class. Import `NetConnection`, `NetStream`, and any other classes you need:

```

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

```

- 2 Create a new class, `Streams`, and declare the variables you'll need within it:

```

public class Streams extends Sprite
{
    var nc:NetConnection;
    var stream:NetStream;
    var playStream:NetStream;
    var video:Video;
    ...
}

```

- 3 Define the `Streams` class constructor: create a `NetConnection` object and add an event listener to it, and connect to the server:

```

public function Streams()
{
    nc = new NetConnection();
    nc.addEventListener(NetStatusEvent.NET_STATUS, netStatusHandler);
    nc.connect("rtmp://localhost/Streams");
}

```

- 4 Create your `netStatusHandler` function (note that it handles both `NetConnection` and `NetStream` events):

```

private function netStatusHandler(event:NetStatusEvent):void
{

```

```

trace("connected is: " + nc.connected );
trace("event.info.level: " + event.info.level);
trace("event.info.code: " + event.info.code);

switch (event.info.code)
{
    case "NetConnection.Connect.Success":
        trace("Congratulations! you're connected");
        connectStream(nc);
        // createPlayList(nc);
        // instead you can also call createPlayList() here
        break;

    case "NetConnection.Connect.Failed":
    case "NetConnection.Connect.Rejected":
        trace ("Oops! the connection was rejected");
        break;

    case "NetStream.Play.Stop":
        trace("The stream has finished playing");
        break;

    case "NetStream.Play.StreamNotFound":
        trace("The server could not find the stream you specified");
        break;

    case "NetStream.Publish.BadName":
        trace("The stream name is already used");
        break;
}
}

```

(To see the full list of event codes that are available, check `NetStatusEvent.info` in the [ActionScript 3.0 Language and Components Reference](#).)

5 Create a `NetStream` object and register a `netStatus` event listener:

```

private function connectStream(nc:NetConnection):void {
    stream = new NetStream(nc);
    stream.addEventListener(NetStatusEvent.NET_STATUS, netStatusHandler);
    stream.client = new CustomClient();
    ...
}

```

Notice that you set the `client` property to an instance of the `CustomClient` class. `CustomClient` is a separate class you need to write that defines some special event handlers (see “[Write the client event handler class](#)”).

6 Create a `Video` object and attach the stream to it:

```

video = new Video();
video.attachNetStream(stream);

```

Here we create the `Video` object using ActionScript 3.0. You can also create it by dragging the `Video` symbol to the Stage in Flash.

In ActionScript 3.0, use `video.attachNetStream()`—not `video.attachVideo()` as in ActionScript 2.0—to attach the stream to the `Video` object.

7 Call `NetStream.play()` to play the stream and `addChild()` to add it to the Stage:

```

...
stream.play("bikes", 0);
addChild(video);
}

```

You don't need to call `addChild()` if you dragged a Video symbol to the Stage using Flash.

The URI of the stream you pass to `NetStream.play()` is relative to the URI of the application you pass to `NetConnection.connect()`.

Write the client event handler class

You also need to write the `CustomClient` class, which contains the `onMetaData` and `onPlayStatus` event handlers. You must handle these events when you call `NetStream.play()`, but you cannot use the `addEventListener()` method to register the event handlers.

- 1 In your main client class, attach the new class to the `NetStream.client` property:

```
stream.client = new CustomClient();
```

- 2 Create the new client class:

```
class CustomClient {  
}
```

- 3 Write a function named `onMetaData()` to handle the `onMetaData` event:

```
public function onMetaData(info:Object):void {  
    trace("metadata: duration=" + info.duration + " width=" + info.width +  
          " height=" + info.height + " framerate=" + info.framerate);  
}
```

- 4 Write a function named `onPlayStatus()` to handle the `onPlayStatus` event:

```
public function onPlayStatus(info:Object):void {  
    trace("handling playstatus here");  
}
```

Checking video files before playing

Use the `FLVCheck` tool to check a recorded video file for errors before playing it. Errors in the video file might prevent it from playing correctly. For more information, see *Adobe Flash Media Server Configuration and Administration Guide*.

Handling errors

About error handling

As you build video applications, it is important to learn the art of managing connections and streams. In a networked environment, a connection attempt might fail for any of these reasons:

- Any section of the network between client and server might be down.
- The URI to which the client attempts to connect is incorrect.
- The application instance does not exist on the server.
- The server is down or busy.
- The maximum number of clients or maximum bandwidth threshold may have been exceeded.

If a connection is established successfully, you can then create a `NetStream` object and stream video. However, the stream might encounter problems. You might need to monitor the current frame rate, watch for buffer empty messages, downsample video and seek to the point of failure, or handle a stream that is not found.

To be resilient, your application needs to listen for and handle `netStatus` events that affect connections and streams. As you test and run your application, you can also use the Administration Console to troubleshoot various connection and stream events.

Handle a failed connection

If a connection cannot be made, handle the `netStatus` event *before* you create a `NetStream` object or any other objects. You may need to retry connecting to the server's URI, ask the user to reenter a user name or password, or take some other action.

The event codes to watch for and sample actions to take are as follows:

Event	Action
<code>NetConnection.Connect.Failed</code>	Display a message for the user that the server is down.
<code>NetConnection.Connect.Rejected</code>	Try to connect again.
<code>NetConnection.Connect.AppShutDown</code>	Disconnect all stream objects and close the connection.

Note: Use the `SimpleConnectManage` sample, `SimpleConnectManage.as`, written in ActionScript 3.0.

Write client code to handle netStatus events

❖ Create a `NetConnection` object and connect to the server. Then, write a `netStatus` event handler in which you detect each event and handle it appropriately for your application, for example:

```
public function netStatusHandler(event:NetStatusEvent):void
{
    trace("connected is: " + nc.connected );
    trace("event.info.level: " + event.info.level);
    trace("event.info.code: " + event.info.code);
    switch (event.info.code)
    {
        ...
        case "NetConnection.Connect.Rejected":
            trace ("Oops! the connection was rejected");
            // try to connect again
            break;
        case "NetConnection.Connect.Failed":
            trace("The server may be down or unreachable");
            break;
        case "NetConnection.Connect.AppShutDown":
            trace("The application is shutting down");
            // this method disconnects all stream objects
            nc.close();
            break;
        ...
    }
}
```

Handle a stream not found

If a stream your application attempts to play is not found, a `netStatus` event is triggered with a code of `NetStream.Play.StreamNotFound`. Your `netStatus` event handler should detect this code and take some action, such as displaying a message for the user or playing a standard stream in a default location.

Note: Use the `Streams` sample, `Streams.as`, written in ActionScript 3.0.

Write the client code

- ❖ In your `netStatus` event handler, check for the `StreamNotFound` code and take some action:

```
private function onNetStatus(event:NetStatusEvent):void {
    switch (event.info.code) {
        case "NetStream.Play.StreamNotFound":
            trace("The server could not find the stream you specified");
            ns.play( "public/welcome");
            break;
        ...
    }
}
```

Playlists

About playlists

A *playlist* is a list of streams to play in a sequence. The server handles the list of streams as a continuous stream and provides buffering, so that the viewer experiences no interruption when the stream changes.

You can define a playlist in your client-side code. To do so, call `NetStream.play()` and specify stream names as parameters to the `NetStream.play()` method. The `play()` method is described in detail in the [ActionScript 3.0 Language and Components Reference](#) and the *ActionScript 2.0 Client Language Reference Addendum*.

Because you call `play()` from a `NetStream` object, and a `NetStream` object is associated with a `NetConnection` object, all streams in a playlist originate from the same server.

Create a client-side playlist

This playlist uses names of streams that are stored on the server. To change the playlist, you need to change the code in your application client.

Note: Use the *Streams* sample, *Streams.as*, written in *ActionScript 3.0*.

- 1 Create a `NetConnection` object, connect to the server, and add a `netStatus` event handler.
- 2 Create a `NetStream` object and listen for `netStatus` events:

```
private function createPlayList(nc:NetConnection):void {
    stream = new NetStream(nc);
    stream.addEventListener(NetStatusEvent.NET_STATUS, netStatusHandler);
    stream.client = new CustomClient();
    ...
}
```

- 3 Attach the `NetStream` object to a `Video` object:

```
video = new Video();
video.attachNetStream(stream);
```

- 4 Define a series of `play()` methods on the `NetStream` object:

```
stream.play( "advertisement", 0, 30 );
stream.play( "myvideo", 10, -1, false );
stream.play( "bikes", 0, -1, false );
stream.play( "parade", 30, 120, false);
addChild(video);
}
```

- 5 Listen for `NetStream` event codes in your `netStatus` event handler:

```
private function netStatusHandler(event:NetStatusEvent):void
{
    ...
    case "NetStream.Play.Stop":
        trace("The stream has finished playing");
        break;
    case "NetStream.Play.StreamNotFound":
        trace("The server could not find the stream");
        break;
}
```

This playlist plays these streams:

- A recorded stream named *advertisement.flv*, from the beginning, for 30 seconds
- The recorded stream *myvideo.flv*, starting 10 seconds in, until it ends
- The recorded stream *bikes.flv*, from start to end
- The recorded stream *parade.flv*, starting 30 seconds in and continuing for 2 minutes

Multiple bit rate switching

About multiple bit rate switching

Adobe Flash Media Server can encode and deliver On2 V6 and Sorenson Spark encoded video. Flash Player 8 and 9 support both codecs, while Flash Player 7 and earlier versions support only the Sorenson Spark codec.

You can create virtual directories on the server to store copies of video streams in each format. This lets your application deliver the highest quality content to clients based on their Flash Player version.

Consider an example: A user wants to play a stream and has Flash Player 8 installed, which can play On2 video. The client application requests the *HappyStream.flv* file. After contacting the server, Flash Player 8 determines the value of the `Client.virtualKey` property. The `virtualKey` property maps to the `c:\streams\on2` directory, instead of the default `c:\streams` directory, so the server plays the *HappyStream.flv* stream encoded with the On2 codec.

Deliver streams based on Flash Player version

- ❖ Edit the `VirtualKeys` and `VirtualDirectory` elements in the `Vhost.xml` file as follows:

```
<VirtualKeys>
  <Key from="WIN 7,0,19,0" to="WIN 9,0,115,0">A</Key>
  <Key from="WIN 6,0,0,0" to="WIN 7,0,18,0">B</Key>
  <Key from="MAC 6,0,0,0" to="MAC 7,0,55,0">B</Key>
</VirtualKeys>
<VirtualDirectory>
  <Streams key="A">foo;c:\streams\on2</Streams>
  <Streams key="B">foo;c:\streams\sorenson</Streams>
  <Streams key="">foo;c:\streams</Streams>
</VirtualDirectory>
```

For more information about editing the `Key` and `Streams` elements, see *Adobe Flash Media Server Configuration and Administration Guide*.

Detecting bandwidth

About detecting bandwidth

Matching a data stream to the client's bandwidth capacity is perhaps the most important factor in ensuring a good playback experience. Once you have detected a user's bandwidth, you can:

- Choose a video encoded at a bit rate appropriate for the user's bandwidth speed.
- Play a video and set the buffer size based on the detected bandwidth.

In Adobe Flash Media Server 3, bandwidth detection is built in to the server. The new bandwidth detection, called *native bandwidth detection*, provides better performance and scalability. To use native bandwidth detection, make sure bandwidth detection is enabled, and write client code that calls functions built in to Adobe Flash Media Server.

With native bandwidth detection, you can use any version of ActionScript on the client (examples in ActionScript 2.0 and 3.0 are provided below). You do not need to add or change server-side code.

If you prefer to use your existing server-side bandwidth detection code from a previous release of Flash Media Server, you can disable native bandwidth detection by configuring `BandwidthDetection` in `Application.xml`.

Initiate bandwidth detection from the server

The following Server-Side ActionScript code initiates bandwidth detection from the server. In an edge-origin deployment, bandwidth is detected at the origin server, not at the edge server:

```
application.onConnect = function (clientObj){
    this.acceptConnection(clientObj);
    clientObj.checkBandwidth();
}
```

When initiating bandwidth detection from the server, do not call `checkBandwidth` from the client.

ActionScript 3.0 native bandwidth detection

The client should initiate bandwidth detection after successfully connecting to the server. To start bandwidth detection, call `NetConnection.call()`, passing it the special command `checkBandwidth`. No server-side code is needed.

Note: Use the Bandwidth sample, `Bandwidth.as`, written in ActionScript 3.0.

Edit Application.xml

- ❖ Make sure bandwidth detection is enabled in the `Application.xml` file for your application:

```
<BandwidthDetection enabled="true">
```

Bandwidth detection is enabled by default. You can use an `Application.xml` file specific to your application or one that applies to a virtual host (see *Adobe Flash Media Server Configuration and Administration Guide* for details).

Write the client event handler class

- ❖ Create an ActionScript 3.0 class that handles events and calls bandwidth detection on the server. It must implement the `onBWCheck` and `onBWDone` functions:

```
class Client {
    public function onBWCheck(... rest):Number {
        return 0;
    }
}
```

```

public function onBWDone(... rest):void {
    var p_bw:Number;
    if (rest.length > 0) p_bw = rest[0];
        // your application should do something here
        // when the bandwidth check is complete
        trace("bandwidth = " + p_bw + " Kbps.");
    }
}

```

The `onBWCheck` function must return a value, even if the value is 0. The `onBWDone` function should contain the application logic. This class will be a client to your main ActionScript 3.0 class.

Write the main client class

- 1 Create your main ActionScript 3.0 class, giving it a package and class name of your choice:

```

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

    public class Bandwidth extends Sprite
    {
    }
}

```

You can create the main and client classes in the same file.

- 2 In the constructor of the main class, create a `NetConnection` object, set the `NetConnection.client` property to an instance of the client class, and connect to the server:

```

private var nc:NetConnection;

public function Bandwidth()
{
    nc = new NetConnection();
    nc.addEventListener(NetStatusEvent.NET_STATUS, netStatusHandler);
    nc.client = new Client();
    nc.connect("rtmp://localhost/FlashVideoApp");
}

```

- 3 In the `netStatus` event handler, call `NetConnection.call()` if the connection is successful, passing `checkBandwidth` as the command to execute and `null` for the response object:

```

public function netStatusHandler(event:NetStatusEvent):void
{
    switch (event.info.code)
    {
        case "NetConnection.Connect.Success":
            // calls bandwidth detection code built in to the server
            // no server-side code required
            trace("The connection was made successfully");
            nc.call("checkBandwidth", null);
            break;
        case "NetConnection.Connect.Rejected":
            trace("sorry, the connection was rejected");
            break;
        case "NetConnection.Connect.Failed":
            trace("Failed to connect to server.");
            break;
    }
}

```

Note: The `checkBandwidth()` method belongs to the `Client` class on the server.

Run the sample

- ❖ Test the main class from Flash CS3 or Flex Builder 2. You will see output like this showing you the client's bandwidth:

```
[SWF] C:\samples\Bandwidth\bin\Bandwidth-debug.swf - 2,137 bytes after decompression
The connection was made successfully
bandwidth = 7287
```

In this example, the `Client` class simply displays the bandwidth value. In your client, you should take some action, such as choosing a specific recorded video to stream to the client based on the client's bandwidth.

ActionScript 2.0 native bandwidth detection

You can also use native bandwidth detection from ActionScript 2.0. Just as in ActionScript 3.0, you define functions named `onBWCheck()` and `onBWDone()`, and you make a call to `NetConnection.call()`, passing it the function name `checkBandwidth`.

Note: Use the `BandwidthAS2` sample, `BandwidthAS2.as`, written in ActionScript 2.0.

Edit Application.xml

- ❖ Make sure bandwidth detection is enabled in the `Application.xml` file for your application:

```
<BandwidthDetection enabled="true">
```

Bandwidth detection is enabled by default. You can use an `Application.xml` file specific to your application or one that applies to a virtual host (see *Adobe Flash Media Server Configuration Guide* for details).

Write the client code

- 1 Define an event handler named `onBWCheck()` that receives data the server sends:

```
NetConnection.prototype.onBWCheck = function(data) {
    return 0;
}
```

This handler *must* return a value, but it can be any value, even 0. The return value lets the server know the data was received.

- 2 Define an event handler named `onBWDone()` that accepts one parameter, the measured bandwidth in kilobytes per second (Kbps):

```
NetConnection.prototype.onBWDone = function(bw) {
    trace("bw = " + bw + " Kbps");
}
```

When the server completes its bandwidth detection, it calls `onBWDone()` and returns the bandwidth figure.

- 3 Define an `onStatus` handler that calls `checkBandwidth` on the server if the connection is successful:

```
NetConnection.prototype.onStatus = function(info) {
    if (info.code == "NetConnection.Connect.Success") {
        this.call("checkBandwidth"); // tell server to start bandwidth detection
    }
}
```

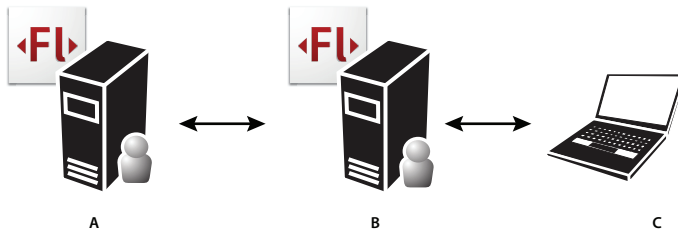
- 4 Create a `NetConnection` object and connect to the server:

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

Script-based bandwidth detection

You can disable native bandwidth detection in the server and use bandwidth detection in a server-side script, such as a `main.asc` file, especially if you want to reuse existing code.

If you use edge servers, server-side native bandwidth detection is performed at the outermost edge server to reduce the load on the origin servers. However, if you have a network of edge and origin servers, server-side bandwidth detection determines the bandwidth from the origin server to the client, not from the edge server to the client. If latency exists between the origin server and the edge server, it might affect the bandwidth calculation.



*Latency between the Origin and Edge server can affect the bandwidth measurement.
A. Origin server B. Edge server C. Client*

To use server-side bandwidth detection, use the specialized `main.asc` file Adobe provides for bandwidth detection. You can find the `main.asc` file in the `\ComponentsAS2\FLVPlayback` folder in the `Samples.zip` file provided with Flash CS3, which you can download at www.adobe.com/go/learn_fl_samples. You might also need to make some changes in your ActionScript 3.0 client (see the `BandwidthServer.as` (ActionScript 3.0) sample).

Edit Application.xml

- ❖ Disable native bandwidth detection in the `Application.xml` file for your application:

```
<BandwidthDetection enabled="false">
```

Bandwidth detection is enabled by default.

Write the client event handler class

- 1 Write your client code as if you are using native bandwidth detection (see “[ActionScript 3.0 native bandwidth detection](#)”). Create at least two classes, a main class that connects to the server and an event handler class.
- 2 In the event handler class, define the `onBWCheck` and `onBWDone` functions, as shown in `BandwidthServer.as`:

```
class Client {
    public function onBWCheck(... rest):Number {
        return 0;
    }
    public function onBWDone(... rest):void {
        var p_bw:Number;
        if (rest.length > 0)
            p_bw = rest[0];
        trace("bandwidth = " + p_bw);
    }
}
```

Make sure `onBWCheck()` returns a value, and `onBWDone()` contains your application logic.

Detecting stream length

About detecting stream length

The server-side `Stream` class allows you to detect the length of a recorded stream. The `Stream` class has a static method, `length()`, that returns the length in seconds of an audio or video stream. The length is measured by Adobe Flash Media Server and differs from the duration that `onMetaData` returns, which is set by a user or a tool.

With `Stream.length()`, you must specify the name of a stream. It can be an actual stream name, in a URI relative to the application instance used in `NetConnection.connect()`, or a virtual stream name.

To use an actual stream name, for an application located in `RootInstall/applications/dailyNews/` with the stream in the `dailyNews/streams/_definst_` subdirectory, call `Stream.length()` like this:

```
length = Stream.length( "parade" );      // for an FLV file
length = Stream.length( "mp3:parade.mp3" );    // for an MP3 file
length = Stream.length( "mp4:parade.mp4" );    // for an MP4 file
```

Virtual stream names

Stream names are not always unique on a server, so you can also use virtual stream names. With virtual stream names, multiple recorded streams can have the same name within different physical directories. You create a virtual stream name by configuring the values of `<VirtualDirectory>` and `<VirtualKeys>` in the `Vhost.xml` file.

The virtual stream name looks like an ordinary stream name, but can map to various locations on the server. With a virtual stream name, call `Stream.length()` just as with an actual stream name:

```
// this is a virtual stream name
length = Stream.length( "videos/parade" );
```

Edit Vhost.xml

A virtual stream name uses both a virtual key to identify the client and a virtual directory mapping. You only need to create a virtual stream mapping if the client uses a virtual stream name in its URI.

- 1 Locate `Vhost.xml` in your server installation.

By default, it is located at `RootInstall/conf/_defaultRoot/_defaultVHost_`.

- 2 Add a virtual key to the `Streams` element, for example:

```
<Streams key="9">
```

A virtual key is a mapping that chooses a `Streams` element to use. If the client attempting to play the stream has a key matching this virtual key, the server uses this virtual mapping.

- 3 Add a mapping between a virtual directory and a physical directory, for example:

```
<Streams key="9">videos;c:\data</Streams>
```

This maps all clients with a virtual key of 9 that are requesting streams whose URIs begin with `videos` to the physical directory `c:\data`. The stream name `videos/parade` maps to the physical file `c:/data/parade.flv`.

Get the length of a stream

This example shows how to have the server detect the length of a stream.

Note: Use the `StreamLength` sample, `main.asc` (Server-side ActionScript) and `StreamLength.as` (ActionScript 3.0). To run the sample, see the general instructions in [Deploy an application](#).

Write the server-side code

A client might need to retrieve the length of a stream stored on the server, for example, if a Flash CS3 presentation displays the length of a video to let the user decide whether to play it.

To do this, define a method in server-side code that calls `Stream.length()`, and then have the client call it using `NetConnection.call()`.

❖ In `main.asc`, define a function on the `client` object that calls `Stream.length()`. Do this within the `onConnect` handler:

```
application.onConnect = function( client ) {
    client.getStreamLength = function( streamName ) {
        trace("length is " + Stream.length( streamName ));
        return Stream.length( streamName );
    }
    application.acceptConnection( client );
}
```

Write the main client class

From the main client class, you call `getStreamLength()` in the server-side code. You need to create a `Responder` object to hold the response:

```
var responder:Responder = new Responder(onResult);
```

This line specifies that the `onResult()` function will handle the result. You also need to write `onResult()`, as shown in the following steps.

1 In your client code, create a package, import classes, and define variables as usual:

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

    import flash.net.NetStream;
    import flash.net.Responder;
    import flash.media.Video;
    ...
}
```

2 Create a new class, `StreamLength`:

```
public class StreamLength extends Sprite
{
    var nc:NetConnection;
    var stream:NetStream;
    var video:Video;
    var responder:Responder;
}
...
}
```

3 In the constructor for the `StreamLength` class, call `NetConnection.connect()` to connect to the server:

```
public function StreamLength()
{
    nc = new NetConnection();
    nc.addEventListener(NetStatusEvent.NET_STATUS, netStatusHandler);
    nc.connect("rtmp://localhost/StreamLength");
}
```

4 Add a `netStatus` event handler to handle a successful connection, rejected connection, and failed connection:

```
private function netStatusHandler(event:NetStatusEvent):void
```

```

{
    trace("connected is: " + nc.connected );
    trace("event.info.level: " + event.info.level);
    trace("event.info.code: " + event.info.code);

    switch (event.info.code)
    {
        case "NetConnection.Connect.Success":
            trace("Congratulations! you're connected");
            connectStream(nc);
            break;

        case "NetConnection.Connect.Rejected":
        case "NetConnection.Connect.Failed":
            trace ("Oops! the connection was rejected");
            break;
    }
}

```

- 5** Write a function to play the stream when a successful connection is made. In it, create a `Responder` object that handles its response in a function named `onResult()`. Then call `NetConnection.call()`, specifying `getStreamLength` as the function to call on the server, the `Responder` object, and the name of the stream:

```

// play a recorded stream on the server
private function connectStream(nc:NetConnection):void {
    stream = new NetStream(nc);
    stream.addEventListener(NetStatusEvent.NET_STATUS, netStatusHandler);
    stream.client = new CustomClient();

    responder = new Responder(onResult);
    nc.call("getStreamLength", responder, "bikes" );
}

```

- 6** Write the `onResult()` function to handle the stream length returned by `getStreamLength()` on the server:

```

private function onResult(result:Object):void {
    trace("The stream length is " + result + " seconds");
    output.text = "The stream length is " + result + " seconds";
}

```

Write the client event handler class

- ❖ As usual with playing a stream, write a separate class to handle the `onMetaData` and `onPlayStatus` events:

```

class CustomClient {
    public function onMetaData(info:Object):void {
        trace("metadata: duration=" + info.duration + " width=" + info.width +
            " height=" + info.height + " framerate=" + info.framerate);
    }
    public function onPlayStatus(info:Object):void {
        trace("handling playstatus here");
    }
}

```

Buffering streams dynamically

About buffering streams

Buffering a video stream helps ensure that the video plays smoothly, without interruption. Buffering manages fluctuations in bandwidth while a video is playing. These fluctuations can occur due to changes in a network connection or server load, or to additional work being done on the client computer.

To create the best experience for users, monitor the progress of a video and manage buffering as the video downloads. You may need to set different buffer sizes for different users, to ensure the best playback experience. One choice is to detect a user's bandwidth (see “[Detecting bandwidth](#)”) and set an initial buffer size based on it.

While the stream is playing, you can also detect and handle `netStatus` events. For example, when the buffer is full, the `netStatus` event returns an `info.code` value of `NetStream.Buffer.Full`. When the buffer is empty, another event fires with a `code` value of `NetStream.Buffer.Empty`. When the data is finished streaming, the `NetStream.Buffer.Flush` event is dispatched. You can listen for these events and set the buffer size smaller when empty and larger when full.

***Note:** Flash Player 9 Update 3 no longer clears the buffer when a stream is paused. This allows viewers to resume playback without experiencing any hesitation. Developers can also use `NetStream.pause()` in code to buffer data while viewers are watching a commercial, for example, and then unpause the stream when the main video starts. For more information, see the `NetStream.pause()` entry in the *Adobe Flash Media Server ActionScript 2.0 Language Reference* or in the *ActionScript 3.0 Language and Components Reference*.*

Handle buffer events

This example shows how to detect buffer events and adjust the buffer time dynamically, as events occur. Highlights of the code are shown here; to see the complete sample, see the `Buffer.as` sample file. To run the sample, see the general instructions in [Deploy an application](#).

To change the buffer time, use `NetStream.setBufferTime()` to set a value in seconds, for example:

```
ns.setBufferTime(2);
```

Two seconds is a good buffer size for fast connections; 10 seconds is a good buffer size for slow connections.

Write the main client class

- 1 Create an ActionScript 3.0 class.
- 2 In the constructor function of the class, create a `NetConnection` object and connect to the server (see `Buffer.as` in the `documentation/samples/Buffer` directory in the Flash Media Server root install directory):

```
nc = new NetConnection();  
nc.addEventListener(NetStatusEvent.NET_STATUS, netStatusHandler);  
nc.connect("rtmp://localhost/Buffer");
```

- 3 Write a `netStatus` event handler, checking for success, failure, and full buffer and empty buffer events and changing buffer size accordingly:

```
private function netStatusHandler(event:NetStatusEvent):void {  
    switch (event.info.code) {  
        case "NetConnection.Connect.Success":  
            trace("The connection was successful");  
            connectStream(nc);  
            break;  
        case "NetConnection.Connect.Failed":  
            trace("The connection failed");
```

```

        break;
    case "NetConnection.Connect.Rejected":
        trace("The connection was rejected");
        break;
    case "NetStream.Buffer.Full":
        ns.bufferTime = 10;
        trace("Expanded buffer to 10");
        break;
    case "NetStream.Buffer.Empty":
        ns.bufferTime = 2;
        trace("Reduced buffer to 2");
        break;
    }
}

```

- 4 Write a custom method to play a stream. In the method, set an initial buffer time, for example, 2 seconds:

```

private function connectStream(nc:NetConnection):void {
    ns = new NetStream(nc);
    ns.addEventListener(NetStatusEvent.NET_STATUS, netStatusHandler);
    ns.client = new CustomClient();

    video = new Video();
    video.attachNetStream(ns);

    ns.play( "bikes", 0 );
    ns.bufferTime = 2;
    trace("Set an initial buffer time of 2");

    addChild(video);
}

```

Write the client event handler class

- ❖ As usual when you stream video, write a separate client class that implements the `onMetaData()` and `onPlayStatus()` event handlers:

```

public function onMetaData(info:Object):void {
    trace("Metadata: duration=" + info.duration + " width=" + info.width + " height="
+ info.height + " framerate=" + info.framerate);
}

public function onPlayStatus(info:Object):void {
    switch (info.code) {
        case "NetStream.Play.Complete":
            trace("The stream has completed");
            break;
    }
}

```

These event handlers are needed when you call `NetStream.play()`.

Chapter 4: Developing live video applications

Adobe Flash Media Server clients can capture live audio and video from a microphone or camera and share the live content with other clients. This feature allows you to capture live events in real time and stream them to a large audience or create live audio and video conferences. You can use Flash Media Encoder to capture and stream live video to Flash Media Server. However, because Flash Media Encoder is only supported on Windows, you may want to create your own client that captures and streams live video.

Note: The majority of this chapter is only applicable to Flash Media Interactive Server and Flash Media Development Server, as Flash Media Streaming Server does not provide server-side programming.

Capturing and streaming live audio and video

About publishing streams and subscribing to streams

To send live video to Flash Media Server from Flash Player, you must attach the video to a `NetStream` object using `NetStream.attachVideo()`. Likewise, to attach live audio, use `NetStream.attachAudio()`.

To publish a live stream or subscribe to a recorded stream, use the client-side `NetStream` class. If a recorded video is already available on the server and you want the client to receive it, create a `NetStream` object in the client code and call `NetStream.play()`. To publish live video to the server, call `NetStream.publish()`.

The `publish()` method allows you to specify the URI to which you want to publish the stream. The URI is relative to the application folder on the server. For example, if a client is connected to `nc.connect("rtmp://www.example.com/TopNews/today")`, and it uses a relative URI to specify the stream name, for example, `stream.publish("local", "record")`, the video is recorded in a file named `local.flv` and published to `RootInstall/applications/TopNews/streams/today/local.flv`.

Clients connecting to the stream as subscribers would connect to the `TopNews/today` application instance and then specify `local` in the `NetStream.play()` method.

A client that creates a stream to send to a server is *publishing*, while a client that creates a stream to receive the content is *subscribing*. When the same client both publishes and subscribes, it must create two streams, an out stream and an in stream. The content sent over the stream can be recorded and streamed later from the server, or captured live by camera or microphone and streamed immediately.

LiveStreams example

To stream live audio and video from the client, use the `Camera`, `Microphone`, and `ActivityEvent` classes, along with `NetStream` and `NetConnection`, as usual.

Note: Use the LiveStreams sample: `LiveStreams.as`, and `main.asc`. (Highlights of the code are shown here.) For the user interface, use the `LiveStreams.fla` file, included with the sample. To run the sample, see the general instructions in [Deploy an application](#).

Write the client-side code

1 Create an ActionScript class. In your class, create a `NetConnection` object, add a `netStatus` event handler, and connect to the server.

2 Add the `ActivityEvent` handler:

```
private function activityHandler(event:ActivityEvent):void {
    trace("activityHandler: " + event);
    trace("activating: " + event.activating);
}
```

3 Write a function to publish the live stream. First, create `NetStream`, `Camera`, and `Microphone` objects:

```
private function publishLiveStream():void {
    ns = new NetStream(nc);
    ns.addEventListener(NetStatusEvent.NET_STATUS, netStatusHandler);
    ns.client = new CustomClient();

    camera = Camera.getCamera();
    mic = Microphone.getMicrophone();
    ...
}
```

4 Add `ActivityEvent` listeners to the `Camera` and `Microphone` objects. Then, attach the camera and microphone data to the `Video` object and the `NetStream` object, respectively, and call `NetStream.publish()`:

```
if (camera != null) {

    camera.addEventListener(ActivityEvent.ACTIVITY, activityHandler);

    video = new Video();
    video.attachCamera(camera);

    ns.attachCamera(camera);
}

if (mic != null) {

    mic.addEventListener(ActivityEvent.ACTIVITY, activityHandler);

    ns.attachAudio(mic);
}

if (camera != null || mic != null) {
    // start publishing
    // triggers NetStream.Publish.Start
    ns.publish("localNews", "live");
} else {
    trace("Please check your camera and microphone");
}
```

Write the server-side code

Note: The server-side code in the `LiveStreams` example demonstrates the multipoint publishing feature which lets you republish streams to additional servers for broadcasting to a wider audience.

1 Write the `application.onConnect()` and `application.onDisconnect()` functions, as usual (see `main.asc` in the `documentation/samples/LiveStreams` directory in the Flash Media Server root install directory).

These handle connecting the client to the application, and then disconnecting the client.

2 Write the `application.onPublish()` function, which is called when the client publishes a stream:

```

// called when the client publishes
application.onPublish = function(client, myStream) {

    trace(myStream.name + " is publishing into application " + application.name);

    // This is an example of using the multi-point publish feature to republish
    // streams to another application instance on the local server.
    if (application.name == "LiveStreams/_definst_"){

        trace("Republishing the stream into anotherInstance");

        nc = new NetConnection();
        nc.connect( "rtmp://localhost/LiveStreams/anotherInstance" );

        ns = new NetStream(nc);

        // called when the server NetStream object has a status
        ns.onStatus = function(info) {
            trace("Stream Status: " + info.code)
            if (info.code == "NetStream.Publish.Start") {
                trace("The stream is now publishing");
            }
        }

        ns.setBufferTime(2);
        ns.attach(myStream);
        ns.publish( myStream.name, "live" );
    }
}

```

In this case, `onPublish()` uses the multi-point publish feature to republish streams to another application instance on the local server. Notice that you use a `NetStream` object on the server.

- 3** In the `application.onPublish()` function, write an event handler for the `NetStream.onStatus` event:

```

ns.onStatus = function(info) {
    trace("Stream Status: " + info.code)
    if (info.code == "NetStream.Publish.Start") {
        trace("The stream is now publishing");
    }
}

```

The `NetStream` object has an `onStatus` event in Server-Side ActionScript, which is the equivalent of the `netStatus` event in ActionScript 3.0.

- 4** Write the `application.onUnpublish()` function, which is called when the client stops publishing:

```

application.onUnpublish = function( client, myStream ) {
    trace(myStream.name + " is unpublishing" );
}

```

Record live video

When clients send live video to the server, you can give them the option of recording the video and storing it on the server. Applications that use this feature include video blogging and other social media applications that build user communities.

Your client application should first create a `NetStream` object and attach `Camera` and `Microphone` objects to it.

To let a user record the live video, use the `NetStream.publish()` method with the name of a video file and pass "record" for the `howToPublish` parameter. When you pass "record", the file is saved on the server in a subdirectory within the directory that contains the application. Alternately, you can pass "append" for the `howToPublish` parameter to record live video and add it to the end of an existing file.

Note: All video is recorded in the FLV file format.

❖ Follow the instructions in [Capturing and streaming live audio and video](#), but use the `record` parameter with the `NetStream.publish()` method:

```
ns.publish("public/myVacation", "record");
```

Add metadata to a live stream

About metadata

Using Flash Media Server, you can add a data message to the beginning of a live video stream. Useful data might include the duration of the video, the date it was created, the creator's name, and so on. Any client connecting to the server receives the metadata when the live video is played.

The metadata you add is in the form of *data keyframes*. Each data keyframe can contain multiple data properties, such as a title, height, and width.

Call the `NetStream.send()` method in a client-side script to add metadata to a live stream. The following is the `NetStream.send()` syntax for adding a data keyframe:

```
NetStream.send(@setDataFrame, onMetaData [, metadata ])
```

The `onMetaData` parameter is handler that handles the metadata, when it's received. You can create multiple data keyframes and each data keyframe must use a unique handler (for example, `onMetaData1`, `onMetaData2`, and so on).

The `metadata` parameter is an Object (or any subclass of Object) that contains the metadata to set in the stream. Each metadata item is a property with a name and a value set in the `metadata` object. You can use any name, but Adobe recommends that you use common names, so that the metadata you set can be easily read.

Call the `NetStream.send(@clearDataframe, onMetaData)` method in a client-side script to clear metadata from a live stream.

Note: To send metadata from a server-side script, use the *Server-Side ActionScript* `Stream.send()` method.

Metadata properties for live streams

The following metadata properties and values are set by Flash Media Encoder, you do not need to add this metadata to live streams:

Metadata property name	Data type	Description
lastkeyframetimestamp	Number	The timestamp of the last video keyframe recorded.
width	Number	The width of the video, in pixels.
height	Number	The height of the video, in pixels.
videodatarate	Number	The video bit rate.

Metadata property name	Data type	Description
audiodatarate	Number	The audio bit rate.
framerate	Number	The frames per second at which the video was recorded.
creationdate	String	The creation date of the file.
createdby	String	The creator of the file.
audiocodecid	Number	The audio codec ID used in the file. Values are: 0 Uncompressed 1 ADPCM 2 MP3 5 Nellymoser 8 kHz Mono 6 Nellymoser
videocodecid	Number	The video codec ID used in the file. Values are: 2 Sorenson H.263 3 Screen video 4 On2 VP6 5 On2 VP6 with transparency
audiodelay	Number	The delay introduced by the audio codec, in seconds.

Metadata properties for recorded live streams

If you record the file as you stream it (by using a "record" parameter with `NetStream.publish()`), Flash Media Server adds the standard metadata listed in the following table. You do not need to add this metadata to the live stream.

Metadata property name	Data type	Description
duration	Number	The length of the file, in seconds.
audiocodecid	Number	The audio codec ID used in the file. Values are: 0 Uncompressed 1 ADPCM 2 MP3 5 Nellymoser 8kHz Mono 6 Nellymoser
videocodecid	Number	The video codec ID used in the file. Values are: 2 Sorenson H.263 3 Screen video 4 On2 VP6 5 On2 VP6 with transparency
canSeekToEnd	Boolean	Whether the last video frame is a keyframe (true if yes, false if no).
creationdate	String	The date the file was created.
createdby	String	The name of the file creator.

If you add your own metadata to a live stream, but then record the stream by adding "record" to the `NetStream.publish()` call, Flash Media Server attempts to merge the metadata properties you specify with the standard metadata properties it adds to the stream. In case of a conflict between the two, the server uses its standard metadata properties.

For example, suppose you add these metadata properties:

```
duration=5
```

```
x=200
y=300
```

When the server starts to record the video, it begins to write its own metadata properties to the file, including `duration`. If the recording is 20 seconds long, the server adds `duration=20` to the metadata, overwriting the value you specified. However, `x=200` and `y=300` are still saved as metadata, because they create no conflict. The other properties the server sets, such as `audiocodecid`, `videocodecid`, `creationdate`, and so on, are also saved in the file.

Add metadata to live video

Note: See the updated LiveStreams sample, `LiveStreams.as`, at www.adobe.com/learn_fms_docs_en under the Developer Guide.

Write the main client class

- 1 Create an ActionScript class.
- 2 In the constructor, create a `NetConnection` object, register a `netStatus` event handler, and connect to the server.
- 3 Write a method to publish a live stream, using the `Camera` and `Microphone` classes and `NetStream.publish()`:

```
private function publishLiveStream():void {
    var ns:NetStream = new NetStream(nc);
    ns.addEventListener(NetStatusEvent.NET_STATUS, netStatusHandler);
    var camera:Camera = Camera.getCamera();
    var mic:Microphone = Microphone.getMicrophone();
    if (camera != null && mic != null) {
        camera.addEventListener(ActivityEvent.ACTIVITY, activityHandler);
        mic.addEventListener(ActivityEvent.ACTIVITY, activityHandler);
        ns.attachCamera( camera );
        addChild(video);
        ns.attachAudio( mic );
        // start publishing
        // triggers NetStream.Publish.Start
        ns.publish( "local", "record" );
    } else {
        trace("Please check your camera and microphone");
    }
}
```

The call to `NetStream.publish()` triggers a `netStatus` event with the code `NetStream.Publish.Start`. You can choose to broadcast the live stream or record it.

- 4 Add an `ActivityEvent` handler:

```
private function activityHandler(event:ActivityEvent):void {
    trace("activityHandler: " + event);
    trace("activating: " + event.activating);
}
```

- 5 In your `netStatus` event handler, watch for `NetStream.Publish.Start`. When it occurs, create a metadata object and set it in the live stream:

```
private function netStatusHandler(event:NetStatusEvent):void
{
    case "NetStream.Publish.Start":
        trace("Adding metadata to the stream");
        // when publishing starts, send the metadata
        var metaData:Object = new Object();
        metaData.title = "liveDJconcert";
}
```

```

        metaData.width = 400;
        metaData.height = 200;
        stream.send("@setDataFrame", "onMetaData", metaData);
        break;
        ...
    }

```

Write the client event handler class

You also need to write a client event handler class that handles the `onMetaData` event.

- ❖ Write the `onMetaData` handler, checking for width, height, or other metadata commonly used:

```

class CustomClient {
    public function onMetaData(info:Object):void {
        trace("width: " + info.width);
        trace("height: " + info.height);
    }
}

```

Clear metadata from live video

Note: See the updated *LiveStreams* sample, *LiveStreams.as*, at www.adobe.com/learn_fms_docs_en under the *Developer Guide*.

Write the main client class

- 1 In your `netStatus` event handler, watch for the event that has the code value `NetStream.Publish.Start`:

```

private function netStatusHandler(event:NetStatusEvent):void
{
    switch (event.info.code)
    {
        case "NetStream.Publish.Start":
            clearMetaData(ns);
    }
}

```

- 2 When `NetStream.Publish.Start` occurs, clear metadata from the stream by calling `NetStream.send()` with `@clearDataFrame`:

```

private function clearMetaData(ns:NetStream):void {
    this.ns = ns;
    ns.send("@clearDataFrame", "onMetaData");
    ns.attachCamera( camera );
    // clear the metadata
    ns.publish( "localNews" );
}

```

Write the client event handler class

You also need to write a client event handler class that handles the `onMetaData` event.

- ❖ Write the `onMetaData` handler, checking for width, height, or other metadata commonly used:

```

class CustomClient {
    public function onMetaData(info:Object):void {
        trace("width: " + info.width);
        trace("height: " + info.height);
    }
}

```

Retrieve metadata from live video

To display the metadata set in the live stream, you need to handle the `onMetaData` event. Handling the metadata set in a live stream is the same as extracting the metadata set by the server or a tool in a recorded file.

Note: See the updated LiveStreams sample, *LiveStreams.as*, at www.adobe.com/learn_fms_docs_en under the Developer Guide.

Write the client event handler class

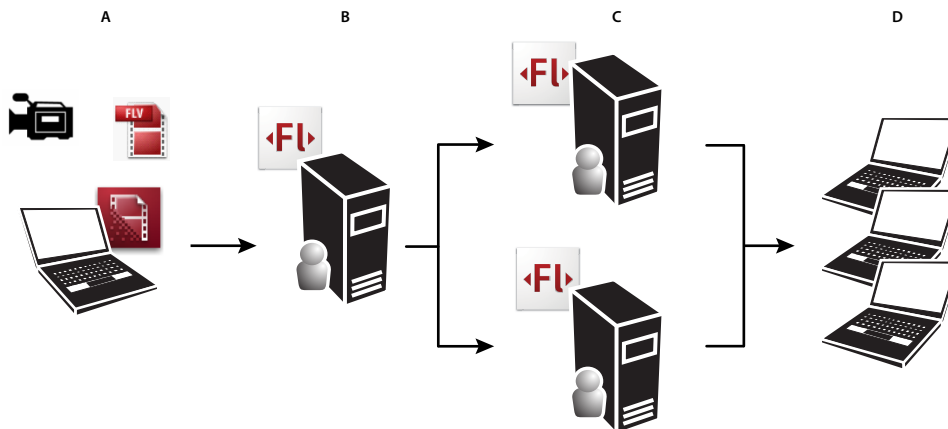
❖ Create a custom client class outside your main client class. In the custom client class, write an `onMetaData` function that takes an `info` object. Get the properties of the `info` object that were set as data keyframes:

```
class CustomClient {
    public function onMetaData(info:Object):void {
        trace("width: " + info.width);
        trace("height: " + info.height);
    }
}
```

Publish from server to server

About multipoint publishing

Multipoint publishing allows clients to publish to servers with only one client-to-server connection. This feature enables you to build large-scale live broadcasting applications, even with servers or subscribers in different geographic locations.



Using multipoint publishing to publish content from server to server, even across geographic boundaries
A. Live Video B. Server 1 (New York City) C. Server 2 (Chicago) and Server 3 (Los Angeles) D. Users

The application flow that corresponds to this illustration works like this:

- 1 A client connects to an application on Server 1 in New York City and starts publishing a live stream.
- 2 The application on Server 1 receives an `application.onPublish` event, which it handles in its `main.asc` file.
- 3 The application creates two `NetStream` objects in its `onPublish` event handler to stream to Server 2 (Chicago) and Server 3 (Los Angeles).

- 4 Server 1 rebroadcasts the live stream to Server 2 and Server 3 with a `NetStream.publish()` method.
- 5 Subscribers connecting to Server 2 and Server 3 receive the same live stream.
- 6 The application receives an `application.onUnpublish` event when the client stops publishing.

To use multipoint publishing, you need to write server-side code in a `main.asc` file.

Multipoint publishing

To use multipoint publishing, you need to write both client and server-side code.

Note: See the *LiveStreams* sample, *LiveStreams.as* (in ActionScript 3.0) and *main.asc* (in Server-side ActionScript), at www.adobe.com/learn/fms_docs_en under the Developer Guide.

Note:

Write the client-side code

- 1 In your client, publish a stream using `NetStream.publish()` and a stream name:

```
private function publishLiveStream():void {
    var ns:NetStream = new NetStream(nc);
    ns.addEventListener(NetStatusEvent.NET_STATUS, netStatusHandler);
    camera = Camera.getCamera();
    mic = Microphone.getMicrophone();
    if (camera != null && mic != null) {
        camera.addEventListener(ActivityEvent.ACTIVITY, activityHandler);
        mic.addEventListener(ActivityEvent.ACTIVITY, activityHandler);
        ns.attachCamera( camera );
        ns.attachAudio( mic );
        // start publishing
        // triggers NetStream.Publish.Start
        ns.publish( "localNews", "record" );
    } else {
        trace("Please check your camera and microphone");
    }
}
```

Write the server-side code

- 1 In your `main.asc` file on the server, define an `application.onPublish()` event handler that accepts the stream name and connects to the remote server:

```
// called when the client publishes
application.onPublish = function( client, myStream ) {
    trace(myStream.name + " is publishing" );
    nc = new NetConnection();
    nc.onStatus = function (info) {
        if (info.code == "NetConnection.Connect.Success") {
            ns = new NetStream(nc);
            ns.setBufferTime(2);
            ns.attach(myStream);
            ns.publish( myStream.name, "live" );
        }
    }
    nc.connect( "rtmp://xyz.com/myApp" );
}
}
```

Calling `NetStream.publish()` publishes the stream from your server to the remote server.

- 2 Handle events that occur on the NetStream object you used to publish from your server to the remote server:

```
ns.onStatus = function(info) {  
    if (info.code) == "NetStream.Publish.Start") {  
        trace("The stream is now publishing");  
        trace("Buffer time is : " + this.bufferTime);  
    }  
}
```

Just as with publishing live streams from a user's computer, the `NetStream.publish()` method triggers a `netStatus` event with a `NetStream.Publish.Start` code.

- 3 Define what happens when the client stops publishing:

```
application.onUnpublish = function( client, myStream ) {  
    trace(myStream.name + " is unpublishing" );  
}
```

Chapter 5: Developing social media applications

In addition to streaming video applications, Adobe Flash Media Interactive Server and Adobe Flash Media Development Server can host social media and other real-time communication applications. Users can capture live audio and video, upload them to the server, and share them with others. These server editions also provide access to remote shared objects that synchronize data between many users, and so is ideal for developing online games.

You can use Server-Side ActionScript to connect to other systems, including Java 2 Enterprise servers, web services, and Microsoft .NET servers. This connectivity allows applications to take advantage of services such as database authentication, real-time updates from web services, and e-mail.

In addition to these advanced techniques, social media applications can take advantage of the video development techniques described in [Developing media applications](#).

***Note:** The majority of this chapter is only applicable to Flash Media Interactive Server and Flash Media Development Server, as Adobe Flash Media Streaming Server does not support server-side programming.*

Shared objects

About shared objects

Use shared objects to synchronize users and store data. Shared objects can do anything from holding the position of pieces on a game board to broadcasting chat text messages. Shared objects let you keep track of what users are doing in real time.

With Flash Media Interactive Server or Flash Media Development Server, you can create and use *remote shared objects*, which share data between multiple client applications. When one user makes a change that updates the shared object on the server, the shared object sends the change to all other users. The remote shared object acts as a hub to synchronize many users. In the section [SharedBall example](#), when any user moves the ball, all users see it move.

***Note:** Flash Media Streaming Server does not support remote shared objects.*

All editions of the server support *local shared objects*, which are similar to browser cookies. Local shared objects are stored on the client computer and don't require a server.

Shared objects, whether local or remote, can also be *temporary* or *persistent*:

- A temporary shared object is created by a server-side script or by a client connecting to the shared object. When the last client disconnects and the server-side script is no longer using the shared object, it is deleted.
- Persistent shared objects retain data after all clients disconnect and even after the application instance stops running. Persistent shared objects are available on the server for the next time the application instance starts. They maintain state between application sessions. Persistent objects are stored in files on the server or client. See “[Setting the location of recorded streams and shared objects](#)” on page 32.

Persistent local shared objects To create persistent local shared objects, call the client-side `SharedObject.getLocal()` method. Persistent local shared objects have the extension `.sol`. You can specify a storage directory for the object by passing a value for the `localPath` parameter of the `SharedObject.getLocal()` command. By specifying a partial path for the location of a locally persistent remote shared object, you can let several applications from the same domain access the same shared objects.

Remotely persistent shared objects To create remote shared objects that are persistent on the server, pass a value of `true` for the `persistence` parameter in the client-side `SharedObject.getRemote()` method or in the server-side `SharedObject.get()` method. These shared objects are named with the extension `.fso` and are stored on the server in a subdirectory of the application that created the shared object. Flash Media Server creates these directories automatically; you don't have to create a directory for each instance name.

Remotely and locally persistent shared objects You create remote shared objects that are persistent on the client and the server by passing a local path for the `persistence` parameter in your client-side `SharedObject.getRemote()` command. The locally persistent shared object is named with the extension `.sor` and is stored on the client in the specified path. The remotely persistent `.fso` file is stored on the server in a subdirectory of the application that created the shared object.

Remote shared objects

Before you create a remote shared object, create a `NetConnection` object and connect to the server. Once you have the connection, use the methods in the `SharedObject` class to create and update the remote shared object. The general sequence of steps for using a remote shared object is outlined below:

- 1 Create a `NetConnection` object and connect to the server:

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

This is the simplest way to connect to the server. In a real application, you would add event listeners on the `NetConnection` object and define event handler methods. For more information, see [SharedBall example](#).

- 2 Create the remote shared object. When the connection is successful, call `SharedObject.getRemote()` to create a remote shared object on the server:

```
so = SharedObject.getRemote("ballPosition", nc.uri, false);
```

The first parameter is the name of the remote shared object. The second is the URI of the application you are connecting to and must be identical to the URI used in the `NetConnection.connect()` method. The easiest way to specify it is with the `nc.uri` property. The third parameter specifies whether the remote shared object is persistent. In this case, `false` is used to make the shared object temporary.

- 3 Connect to the remote shared object. Once the shared object is created, connect the client to the shared object using the `NetConnection` object you just created:

```
so.connect(nc);
```

You also need to add an event listener for `sync` events dispatched by the shared object:

```
so.addEventListener(SyncEvent.SYNC, syncHandler);
```

- 4 Synchronize the remote shared object with clients. Synchronizing the remote shared object requires two steps. First, when an individual client makes a change or sets a data value, you need to update the remote shared object. Next, update all other clients from the remote shared object.

- a To update the remote shared object when a client makes a change, use `setProperty()`:

```
so.setProperty("x", sharedBall.x);
```

You must use `setProperty()` to update values in the shared object. The remote shared object has a `data` property that contains attributes and values. However, in ActionScript 3.0, you cannot write values directly to it, as in:

```
so.data.x = sharedBall.x; // you can't do this
```

- b** When the shared object is updated, it dispatches a `sync` event. Synchronize the change to the remaining clients by reading the value of the shared object's `data` property:

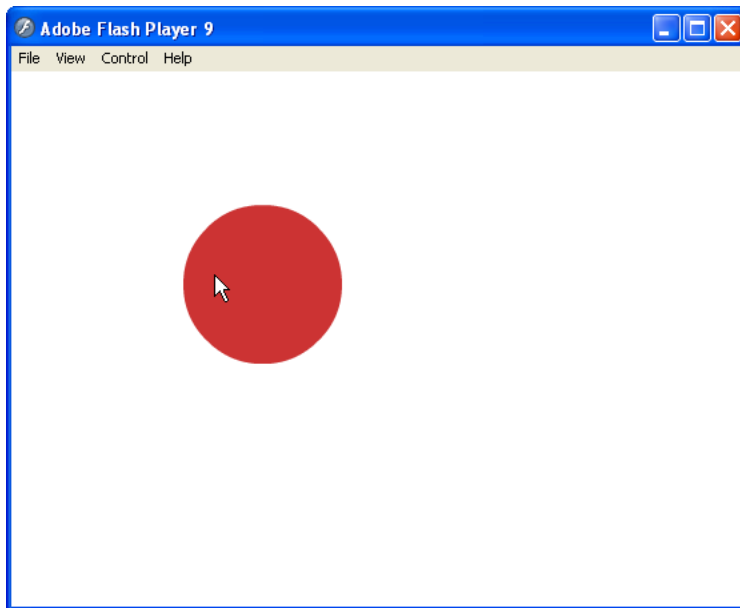
```
sharedBall.x = so.data.x;
```

This is usually done in a `sync` event handler, as shown in [SharedBall example](#).

SharedBall example

The SharedBall sample creates a temporary remote shared object. It's similar to a multiplayer game. When one user moves the ball, it moves for all other users.

Note: Use the SharedBall sample files (*SharedBall.fla*, *SharedBall.as*, and *SharedBall.swf*) in the *documentation/samples/SharedBall* directory in the Flash Media Server root install directory.



Run the application

- 1 Register the application with your server by creating an application directory named SharedBall:
`RootInstall/applications/SharedBall`
- 2 Open the SharedBall sample files from the *documentation/samples/SharedBall* directory in the Flash Media Server root install directory.
- 3 Open *SharedBall.swf* in a web browser.
- 4 Open a second instance of *SharedBall.swf* in a second browser window.
- 5 Move the ball in one window, and watch it move in the other.

Design the user interface

- 1 In Adobe Flash CS3 Professional, choose File > New > Flash File (ActionScript 3.0), and click OK.
- 2 From the toolbox, select the Rectangle tool. Drag to the lower-right corner, then select the Oval tool.
- 3 Draw a circle on the Stage. Give it any fill color you like.
- 4 Double-click the circle, and choose Modify > Convert to Symbol.
- 5 In the Convert to Symbol dialog box, name the symbol **ball**, check that Movie Clip is selected, and click OK.
- 6 Select the ball symbol on the Stage, and in the Property Inspector (Window > Properties), give it the instance name **sharedBall**.
- 7 Save the file as SharedBall.fla.

Write the client-side code

Be sure to look at the SharedBall.as sample file. These steps present only highlights.

- 1 In Flash CS3, create a new ActionScript file.
- 2 Create the class, extending MovieClip:

```
public class SharedBall extends MovieClip { ... }
```

The class must extend MovieClip, because the sharedBall symbol in the FLA file is a Movie Clip symbol.

- 3 Create the constructor, in which you add event listeners and connect to the server:

```
public function SharedBall()
{
    nc = new NetConnection();
    addEventListeners();
    nc.connect("rtmp://localhost/SharedBall");
}
```

- 4 Add event listeners for netStatus, mouseDown, mouseUp, and mouseMove events:

```
private function addEventListeners() {
    nc.addEventListener(NetStatusEvent.NET_STATUS, netStatusHandler);
    // sharedBall is defined in the FLA file
    sharedBall.addEventListener(MouseEvent.CLICK, pickup);
    sharedBall.addEventListener(MouseEvent.CLICK, place);
    sharedBall.addEventListener(MouseEvent.CLICK, moveIt);
}
```

- 5 In your netStatus handler, create a remote shared object when a connection is successful. (You'll also want to create error handlers for rejected and failed connections, shown in the sample AS file.) Connect to the shared object and add a sync event listener:

```
switch (event.info.code)
{
    case "NetConnection.Connect.Success":
        trace("Congratulations! you're connected");
        so = SharedObject.getRemote("ballPosition", nc.uri, false);
        so.connect(nc);
        so.addEventListener(SyncEvent.SYNC, syncHandler);
        break;
    ...
}
```

- 6 As a user moves the mouse, use setProperty() to set the changing ball location in the remote shared object:

```
function moveIt( event:MouseEvent ):void {
```

```

    if( so != null )
    {
        so.setProperty("x", sharedBall.x);
        so.setProperty("y", sharedBall.y);
    }
}

```

When the remote shared object is updated, it dispatches a `sync` event.

- 7 Write a `sync` event handler that updates all clients with the new ball position:

```

private function syncHandler(event:SyncEvent):void {
    sharedBall.x = so.data.x;
    sharedBall.y = so.data.y;
}

```

You can read the value of `so.data`, even though you can't write to it.

Broadcast messages to many users

A remote shared object allows either a client or server to send a message using `SharedObject.send()` to all clients connected to the shared object. The `send()` method can be used for text chat applications, for example, where all users subscribed to your shared object receive your message.

When you use `SharedObject.send()`, you, as broadcaster, also receive a copy of the message.

- 1 Write a method that `SharedObject.send()` will call:

```

private function doSomething(msg:String):void {
    trace("Here's the message: " + msg);
}

```

- 2 Call `send()` to broadcast the message:

```

so = SharedObject.getRemote("position", nc.uri, false);
so.connect(nc);
so.send("doSomething", msg);

```

Allow and deny access to assets

About access control

When users access the server, by default, they have full access to all streams and shared objects. However, you can use Server-Side ActionScript to create a dynamic access control list (ACL) for shared objects and streams. You can control who has access to create, read, or update shared objects or streams.

When a client connects to the server, the server-side script (`main.asc` or `yourApplicationName.asc`) is passed a `Client` object. Each `Client` object has `readAccess` and `writeAccess` properties. You can use these properties to control access for each connection.

Implement dynamic access control

The `Client.readAccess` and `Client.writeAccess` properties take string values. The values can contain multiple strings separated by semicolons, like this:

```

client.readAccess = "appStreams;/appSO/";
client.writeAccess = "appStreams/public;/appSO/public/";

```

By default, `readAccess` and `writeAccess` are set to `/`, which means the client can access every stream and shared object on the server.

Allow access to streams

- ❖ In `main.asc`, add an `onConnect()` function that specifies a directory name on the server in your `main.asc` file:

```
application.onConnect = function(client, name) {
    // give this new client the same name as passed in
    client.name = name;

    // give write access
    client.writeAccess = "appStreams/public/";

    // accept the new client's connection
    application.acceptConnection(client);
}
```

This `main.asc` file grants access to all URIs that start with `appStreams/public`.

Deny access to streams

- ❖ In `main.asc`, add an `onConnect()` function that specifies a null value for `client.writeAccess`:

```
application.onConnect = function(client, name) {
    ...
    // deny write access to the server
    client.writeAccess = "";
}
```

Define access to shared objects

- ❖ In `main.asc`, add an `onConnect()` function that specifies shared object names, using the same URI naming conventions:

```
application.onConnect = function(client, name) {
    ...
    client.writeAccess = "appSO/public/";
}
```

This gives the client write access to all shared objects whose URIs begin with `appSO/public/`.

Authenticate clients

Use properties of the Client object

When a client connects to an application, the server creates a `Client` object that contains information about the client and passes it to the `application.onConnect()` handler in Server-Side ActionScript. You can write server-side code to access the properties of the `Client` object and use the values to verify the validity of the connecting client:

```
application.onConnect = function( pClient ) {
    for (var i in pClient) {
        trace( "key: " + i + ", value: " + pClient[i] );
    }
}
```

Check the client's IP address

- ❖ In `main.asc`, check the value of `client.ip` and, if needed, reject the client's connection to the application:

```
if (client.ip.indexOf("60.120") !=0) {
```

```

        application.rejectConnection(client, {"Access Denied" });
    }

```

Check an originating URL

❖ In `main.asc`, check the value of `client.referrer` against a list of URLs that should be denied access. Make sure that SWF files that are connecting to your application are coming from a location you expect. If you find a match, reject the client's connection:

```

referrerList = {};
referrerList["http://www.example.com"] = true;
referrerList["http://www.abc.com"] = true;

if (!referrerList[client.referrer]) {
    application.rejectConnection(client, {"Access Denied" });
}

```

Use a unique key

1 In client-side ActionScript, create a unique key, as in the following code, which concatenates the local computer time with a random number:

```

var keyDate = String(new Date().getTime());
var keyNum = String(Math.random());
var uniqueKey = keyDate + keyNum;

```

2 Send the key to the server in the connection request:

```

nc.connect("rtmp://www.example.com/someApplication", uniqueKey);

```

3 The following code in the `main.asc` file looks for the unique key in the connection request. If the key is missing, or has already been used, the connection is rejected. This way, if a connection is replayed by an imposter, the replay attempt fails.

```

clientKeyList = new Object(); // holds the list of clients by key

application.onConnect = function( pClient, uniqueKey ) {
    if ( uniqueKey != undefined ) { // require a unique key with connection request
        if ( clientKeyList[uniqueKey] == undefined ) { // first time -- allow connection
            pClient.uniqueKey = uniqueKey;
            clientKeyList[uniqueKey] = pClient;
            this.acceptConnection(pClient);
        } else {
            trace( "Connection rejected" );
            this.rejectConnection(pClient);
        }
    }
}

application.onDisconnect = function( pClient ) {
    delete clientKeyList[pClient.uniqueKey];
}

```

Use an Access plug-in

An Access plug-in intercepts incoming requests before passing them on to Flash Media Interactive Server. You can program an Access plug-in to use any form of authentication. For more information, see *Adobe Flash Media interactive Server Plug-in Developer Guide*.

Use Flash Player version

You can protect your content from clients that aren't running in Flash Player, based on the user agent string received from the connection. The user agent string identifies the platform and Flash Player version, for example:

```
WIN 8,0,0,0  
MAC 9,0,45,0
```

There are two ways to access these strings:

Virtual keys Configure the server to remap the stream based on the Flash Player client. For more information, see [Multiple bit rate switching](#) and `VirtualKeys`.

Client.agent Challenge the connection using Server-Side ActionScript:

```
application.onConnect = function( pClient ) {  
    var platform      = pClient.agent.split(" ");  
    var versionMajor  = platform[1].split(",")[0];  
    var versionMinor  = platform[1].split(",")[1];  
    var versionBuild  = platform[1].split(",")[2];  
}  
  
// output example  
// Client.agent: WIN 9,0,45,0  
// platform[0]:  "WIN"  
// versionMajor:  9  
// versionMinor:  0  
// versionBuild: 45
```

Verify connecting SWF files

You can configure the server to verify the authenticity of client SWF files before allowing them to connect to an application. Verifying SWF files prevents someone from creating their own SWF files that attempt to stream your resources. SWF verification is supported by Flash Player 9 Update 3 and above. For more information, see the *Configuration and Administration Guide*.

Allow and deny connections from specific domains

If you know the domains from which the legitimate clients will be connecting, you can whitelist those domains. Conversely, you can blacklist known bad domains.

You can enter a static list of the domain names in the `Adaptor.xml` file. For more information, see *Adobe Flash Media Server Configuration and Administration Guide*.

You can also maintain these lists in your own server-side code and files. In the following example, a file named *bannedIPList.txt* contains a list of excluded IP addresses, which can be edited on the fly:

```
// bannedIPList.txt file contents:  
// 192.168.0.1  
// 128.493.33.0  
  
function getBannedIPList() {  
    var bannedIPFile = new File ("bannedIPList.txt") ;  
    bannedIPFile.open ("text", "read");  
  
    application.bannedIPList = bannedIPFile.readAll();  
  
    bannedIPFile.close();  
    delete bannedIPFile;  
}
```

```

application.onConnect = function(pClient) {
    var isIPOK = true;
    getBannedIPList();
    for (var index=0; index<this.bannedIPList.length; index++) {
        var currentIP = this.bannedIPList[index];
        if (pClient.ip == currentIP) {
            isIPOK = false;
            trace("ip was rejected");
            break;
        }
    }

    if (isIPOK) {
        this.acceptConnection(pClient);
    } else {
        this.rejectConnection(pClient);
    }
}

```

In addition, you can create server-side code to check if requests are coming in too quickly from a particular domain:

```

application.VERIFY_TIMEOUT_VALUE = 2000;

Client.prototype.verifyTimeOut = function() {
    trace(">>>> Closing Connection")
    clearInterval(this.$verifyTimeOut);
    application.disconnect(this);
}

function VerifyClientHandler(pClient) {
    this.onResult = function (pClientRet) {
        // if the client returns the correct key, then clear timer
        if (pClientRet.key == pClient.verifyKey.key) {
            trace("Connection Passed");
            clearInterval(pClient.$verifyTimeOut);
        }
    }
}

application.onConnect = function(pClient) {
    this.acceptConnection(pClient);

    // create a random key and package within an Object
    pClient.verifyKey = ({key: Math.random()});

    // send the key to the client
    pClient.call("verifyClient",
        new VerifyClientHandler(pClient),
        pClient.verifyKey);

    // set a wait timer
    pClient.$verifyTimeOut = setInterval(pClient,
        $verifyTimeOut,
        this.VERIFY_TIMEOUT_VALUE,
        pClient);
}

application.onDisconnect = function(pClient) {
    clearInterval(pClient.$verifyTimeOut);
}

```

Authenticate users

Authenticate using an external resource

For a limited audience, it is feasible to request credentials (login and password) and challenge them using an external resource, such as a database, LDAP server, or other access-granting service.

1. The SWF supplies the user credentials in the connection request.

The client provides a token or username/password using client-side ActionScript:

```
var sUsername = "someUsername";
var sPassword = "somePassword";

nc.connect("rtmp://server/secure1/", sUsername, sPassword);
```

2. Flash Media Server validates the credentials against a third-party system.

You can use the following classes to make calls from Server-Side ActionScript to external sources: `WebService`, `LoadVars`, `XML` classes, `NetServices` (connects to a Flash Remoting gateway). For more information about these classes, see the *Server-Side ActionScript Language Reference*. For more information about Flash Remoting, see http://www.adobe.com/go/learn_fms_flashremoting_en.

```
load("NetServices.asc"); // for Flash remoting
load("WebServices.asc"); // for SOAP web services

pendingConnections = new Object();

application.onConnect = function( pClient, pUsername, pPassword ) {

    // create a unique ID for the client
    pClient.FMSid = application.FMSid++;

    // place the client into a pending array
    pendingConnections[FMSid] = pClient;

    if (pUsername!= undefined && pPassword !=undefined) {
        // issue the external call (3 examples below)
        loadVars.send("http://xyz.com/auth.cfm");

        webService.authenticate(FMSid, pUsername, pPassword);

        netService.authenticate(FMSid, pUsername, pPassword);
    }

    // the result handler (sample only, you will have to customize this)
    // this command will return a true/false and the FMS client id
    Authenticate.onResult = { }
```

3. Flash Media Server accepts or rejects the connection.

If the credentials are valid, Flash Media Server accepts the connection:

```
loadVars.onData = function ( FMSid, pData ) {
    if (pData) {
        application.acceptConnection( pendingConnections[FMSid] );
        delete pendingConnections[FMSid];
    } else {
        application.rejectConnection ( pendingConnections[FMSid] );
    }
}
```

```
        delete pendingConnections[FMSid];  
    }  
}
```

Authenticate using a token

This technique is an alternative to a username/password style of authentication, where the token can be granted based on a property of the client.

The control flow is as follows:

- 1** The client SWF requests an authentication token from a third party.
- 2** The third party returns the token to the client.
- 3** The client sends the token with its connection request.
- 4** Flash Media Server verifies the token with the third party system.
- 5** Flash Media Server accepts the connection.

Index

A

- access control 55
- ActivityEvent class 41
- Administration Console 8
- Adobe Flash CS3 Professional 3
- Adobe Flash Media Development Server 1
- Adobe Flash Media Interactive Server 1
- Adobe Flash Media Server
 - installing 3
 - starting 3
- Adobe Flash Media Streaming Server 1
- Adobe Flash Player 3
- Adobe Flex 3
- Adobe Real-Time Messaging Protocol (RTMP) 1
- allowedHTMLdomains.txt file 15
- allowedSWFdomains.txt file 15
- Application class 8
 - application.onConnect event 22
 - application.onDisconnect event 22
 - application.onPublish event 41
 - application.onUnpublish event 41
- applications
 - client-side code 6
 - connection URI 16
 - debugging 10
 - double-byte 8
 - HelloServer sample 18
 - live service 12
 - registering with the server 10
 - server-side code 7
 - vod service 13
- authentication
 - clients 56
 - domains 58
 - Flash Player version 58

B

- bandwidth, detecting 32, 35
- buffering 39

C

- Camera class 41
- checkBandwidth function 32
- Client class 8
 - ip property 56
 - readAccess property 55
 - writeAccess property 55
- clients, authenticating 56
- client-side code
 - about 6
 - debugging 10

D

- debugging
 - Administration Console sessions 9
 - client-side code 10
 - server-side code 8
- domains
 - allowing connections from 15, 58
- double-byte applications 8

E

- events
 - netStatus 29
 - onMetaData 28
 - onPlayStatus 28

F

- Flash Archive utility (FAR) 11
- FLVCheck utility 28

H

- HelloServer application 18
- HelloWorld application 4

L

- live service 12
- live video
 - metadata 44, 48
 - publishing 41
 - recording 43
 - streaming 41
 - subscribing 41
- log file 9

M

- metadata
 - about 44
 - adding to live streams 44
 - properties 45
 - retrieving 48
- Microphone class 41
- multipoint publishing 48

N

- NetConnection class
 - about 16
 - client property 33
 - status codes 22, 29
- netStatus events 29
- NetStream class
 - pause method 39
 - play method 23

O

- onBWCheck function 32
- onBWDone function 32
- onMetaData event handler 28
- onPlayStatus event handler 28

P

- publishing
 - live video 41
 - multipoint 48

R

- RTMP 1

S

- Server-Side ActionScript
 - about 7
 - handling connections 21
 - packaging 11
 - trace statement 9
- shared objects
 - about 51
 - broadcasting messages 55
 - local 51
 - persistent 51

- remote 14, 51
- temporary 51
- SharedBall example 53
- Stream class 36
- streaming services 12
- streams
 - buffering 39
 - formats 2
 - pausing 39
 - playing 23
 - virtual naming 36
- SWF files
 - publishing 11
 - verifying authenticity of 58

U

- URI, format for connecting to server 16
- UTF-8 encoding 8

V

- Vhost.xml file 36
- vod service 12, 13