

ADOBE® FLASH® MEDIA SERVER 3.5

Developer Guide

Last updated 3/2/2010

© 2009 Adobe Systems Incorporated. All rights reserved.

Adobe® Flash® Media Server 3.5 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, Adobe Premiere, ActionScript, Acrobat Connect, Creative Suite, Dreamweaver, Flash, Flash Lite, Flex, Flex Builder, and XMP 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. Mac is a trademark of Apple Inc., registered in the United States and other countries. All other trademarks are the property of their respective owners.

This work is licensed under the Creative Commons Attribution Non-Commercial 3.0 License. To view a copy of this license, visit

<http://creativecommons.org/licenses/by-nc/3.0/us/>

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

Speech compression and decompression technology licensed from Nellymoser, Inc. (www.nellymoser.com)

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

About server editions	1
Overview of Flash Media Server	1
Set up a development environment	3
Create a Hello World application	3
Creating an application	6
Test an application	7
Deploy an application	9

Chapter 2: Streaming services

About streaming services	12
Using the live service	12
Using the vod service	14
Creating clients for streaming services	17
Streaming services API	18

Chapter 3: Developing media applications

About media applications	20
Connecting to the server	20
Managing connections	25
Streaming recorded media files	28
Handling errors	33
Working with playlists	35
Dynamic streaming	38
Detecting bandwidth	46
Detecting stream length	50

Chapter 4: Working with live video

Capturing live video	53
Adding DVR features to live video	55
Adding metadata to a live stream	61
Publishing from server to server	67

Chapter 5: Developing interactive applications

About interactive applications	70
Shared objects	70
Allow or deny access to assets	74
Authenticate clients	75
Authenticate users	79

Index	82
--------------------	----

Chapter 1: Getting started

About server editions

Adobe® Flash® Media Server offers a combination of streaming media and 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.

For more information about which features each edition supports, see the *Technical Overview* at www.adobe.com/go/learn_fms_techov_en.

Overview of Flash Media Server

Client-server architecture

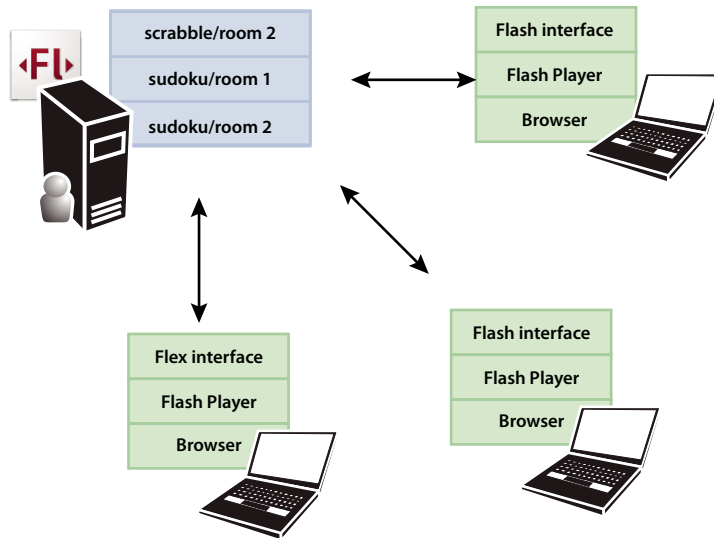
Flash Media Server is a hub. Applications connect to the hub using Real-Time Messaging Protocol. The server can send data to and receive data from many connected users. A user can capture live video or audio using a camera and microphone attached to a computer running Adobe® Flash® Player or Adobe® AIR™ 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, Adobe 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 and permissions, 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 written in Server-Side ActionScript (which is like 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, Adobe 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 and later support ActionScript 3.0, ActionScript 2.0, and ActionScript 1.0.

Video or audio files Many media applications stream recorded audio or video from the server to clients. Flash Media Server supports playback of a variety of stream formats, including Flash Video (FLV), MPEG-3 (MP3), and MPEG-4 (MP4 and F4V).

Camera or microphone You can use Adobe Flash Media Live Encoder to stream live video or audio to the server. You can also create your own client that captures live audio and video. In both cases, you need a camera and a microphone to capture the video and audio.

Server-Side ActionScript Most applications use Server-Side ActionScript code written in a file with the suffix `.asc`, called an *ActionScript Communication File*. The file is named either `main.asc`, or `myApplication.asc`. 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.

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 to use has Flash or Flex installed on the same computer as the server.

Install the server

- ❖ Run the installer to install an edition of Flash Media Server.

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 3.5 > Start Adobe Flash Media Server 3.5.
- 2 From the Start menu, select All Programs > Adobe > Flash Media Server 3.5 > 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

- 1 Open Control Panel > Administrative Tools > Services. In the Services window, make sure that both Flash Media Administration Server and Flash Media Server are started.
- 2 To troubleshoot start-up issues, check the logs in the `RootInstall/logs` folder. The `master.xx.log` file and the `core.xx.log` file show failures. The `edge.xx.log` file shows on which ports the is listening.

Install Flash

- ❖ Download and install Flash at www.adobe.com/go/flash.

Install Flex

- ❖ Download and install the Adobe Flex SDK or Adobe Flex™ Builder™ at www.adobe.com/go/flex.

Create a Hello World application

Overview

Note: The following sections apply to Flash Media Interactive Server and Flash Media Development Server. You cannot write server-side code for Flash Media Streaming Server.

This sample shows simple communication from the client to the server and back again. When a user clicks a button, the client connects to the server. The client calls a server-side function that returns a string. When the server replies, the client displays the string sent from the server.

The sample files are in the `RootInstall\documentation\samples\HelloWorld` folder.

Create the user interface

- 1 Start Flash and select Create New > Flash File (ActionScript 3.0).
- 2 In the Document Class field, enter **HelloWorld**. If you see an ActionScript Class Warning message about a missing definition—click OK. You will add the class file in the next section.
- 3 Choose Windows > Components. Click User Interface and double-click Button to add it to the Stage. On the Properties tab, enter the instance name **connectBtn**.
- 4 Add a Label component above the button, and give it the instance name **textLbl**.
- 5 Save the file as HelloWorld fla.

You can save the client files to any location.

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 box 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 Inside the HelloWorld class declaration, declare variables for the connection and the server responder:

```
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, 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 to the same folder as the HelloWorld.fla file.

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 Save the file as HelloWorld.asc in the *RootInstall/applications/HelloWorld* folder. (Create the “HelloWorld” folder when you save the file.)

Compile and run the application

- 1 Verify that the server is running.
- 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 Flash Output window.

Creating an application

Writing client-side code

A client has code written in ActionScript that connects to the server, handles events, and does other work. With Flash, 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.

For information about learning ActionScript and working with video, see the following resources:

- “Working with Video” in *Programming ActionScript 3.0* at www.adobe.com/go/learn_fms_video_en.
- “ActionScript 2.0 Migration” appendix in the *ActionScript 3.0 Language and Components Reference*.
- *ActionScript 3.0 Language and Components Reference* at www.adobe.com/go/learn_fms_as3lr_en.
- Flash Help Resource Center at www.adobe.com/go/learn_fms_flashhrc_en
- Flex Help Resource Center at www.adobe.com/go/learn_fms_flexhrc_en

More Help topics

“[Copy client-side files to a web server](#)” on page 11

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.

Place the server-side code in a file named `main.asc` or `yourApplicationName.asc`, where `yourApplicationName` is a folder in the `RootInstall/applications` folder. For example, to create 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.

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/appName`

`RootInstall/applications/appName/scripts`

By default, the applications folder is in the root installation folder (`C:\Program Files\Adobe\Flash Media Server 3.5\applications`, on Windows). To configure the location of the applications folder, edit the `fms.ini` or the `Vhost.xml` configuration file. In the `fms.ini` file, edit the following parameter: `VHOST.APPSDIR = C:\Program Files\Adobe\Flash Media Server 3.5\applications`. In the `Vhost.xml` file, edit the `AppsDir` element.

More Help topics

“[Copy server-side script files to the server](#)” on page 10

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. Use a JavaScript editor, such as the Script window in Flash or Adobe® Dreamweaver®, that encodes files to the UTF-8 standard. Use built-in JavaScript methods, such as `Date.toLocaleString()`, to convert 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 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 OS)`, 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:

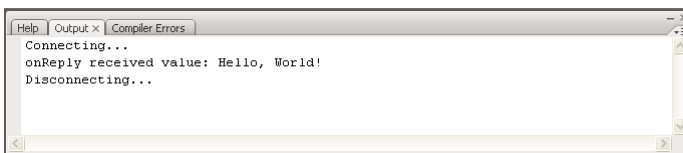
```
// This is the CORRECT way to create double-byte method names
obj["Any_hi_byte_name"] = function() {}

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

Test an application

Test and debug a 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 Output window (this example is from the [“Create a Hello World application”](#) on page 3):



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

Test and debug a server-side script

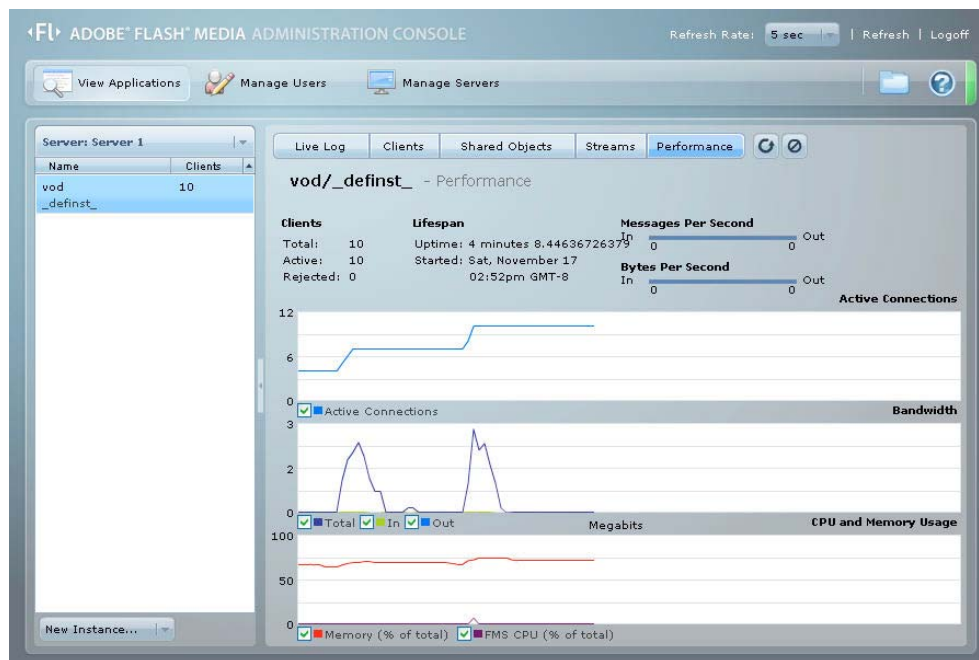
To test a server-side script, use `trace()` statements to monitor each processing point. View the output of `trace()` statements in the Live Log in the Administration Console.

To open the Administration Console, choose Start > All Programs > Adobe > Flash Media Server 3.5 > Flash Media Administration Console.

When a client connects to an application on the server, the 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 do not take effect until the application is restarted. Use the Administration Console to restart the application, 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.



The Administration Console checking the performance of an application while it is running

View the output of a server-side script

The output of the `trace()` statements in a `main.asc` file is sent to the following log file:

`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 on. You can view a log file with any text editor.

While an application is running, you can view the Live Log in the Administration Console. In the Administration Console opens, click View Applications, then Live Log.

Debug with the Administration Console

To playback streams and inspect data about shared objects, an application must make a special debug connection to the Administration Console.

The availability and number of debugging sessions is set in the `AllowDebugDefault` and `MaxPendingDebugConnections` elements of the `Application.xml` configuration file. By default, debugging is not allowed. To override the debug setting in the `Application.xml` file, add the following code to an application's server-side code:

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

Note: Set `allowDebug` to `false` before deploying the application.

To start a debugging session:

- 1 Open the Administration Console.
- 2 Choose View Applications.
- 3 Select the application to debug from the list or choose New Instance and create a new instance of an application.
- 4 Press the Streams button to see the list of playing streams, if any.
- 5 Click on one of the streams.
- 6 Press the Play Stream button.
- 7 A pop-up window will open and the stream will play.
- 8 Press the Shared Objects button to see the application's shared objects, if any.
- 9 Select a shared object.
- 10 Press the Close Debug button to end the debug session.

Deploy an application

Register an application with the server

To connect to an application, the server must know that the application exists. This process is called registering the application with the server. To register an application with the server, create a folder for the application in the applications folder. For example, create the following folder to register an application called "myApplication":

```
RootInstall/applications/myApplication
```

The client-side code that connects to the application looks like the following:

```
myNetConnection.connect("rtmp://fms.examples.com/myApplication");
```

To create instances of an application, create subfolders. For example, the following folder creates an instance of myApplication called "room1":

```
RootInstall/applications/myApplication/room1.
```

The client-side code that connects to the application instance looks like the following:

```
myNetConnection.connect("rtmp://fms.examples.com/myApplication/room1");
```

Every application must have a folder in the applications folder. Usually, the application folder contains the server-side script and any media assets, but the folder can also be empty. It can also contain an application-specific `Application.xml` file.

By default, the applications folder is in the root installation folder (C:\Program Files\Adobe\Flash Media Server 3.5\applications, on Windows). To configure the location of the applications folder, edit the fms.ini or the Vhost.xml configuration file. In the fms.ini file, edit the following parameter: `VHOST.APPSDIR = C:\Program Files\Adobe\Flash Media Server 3.5\applications`. In the Vhost.xml file, edit the `AppDir` element.

Copy server-side script files to the server

Copy server-side script files for an application to the folder you registered on the server. For example, for an application called “videoPlayer”, copy the main.asc file to `RootInstall/applications/videoPlayer`. You can also place server-side scripts in “scripts” subfolder. For example, you can use either of these locations:

`RootInstall/applications/appName`

`RootInstall/applications/appName/scripts`

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

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
<code>-archive archive</code>	Specifies the name of the archive file, which has a <code>.far</code> extension.
<code>-files file1 [file2 ... fileN]</code>	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 media files to the server

Copy video and audio files to the `streams/_definst_` folder in the application folder:

`RootInstall/applications/someApplication/streams/_definst_`

If an application connects to an instance of the application, for example, `nc.connect("rtmp://fms.example.com/someApplication/someInstance")`, place the streams in the following folder:

RootInstall/applications/someApplication/streams/someInstance/

There are several ways to configure the server to look for media files stored in other locations. See the following links for more information.

More Help topics

“[Using the vod service](#)” on page 14

Copy client-side files to a web server

You can deploy SWF files and HTML files on any web server. The SWF file contains the `NetConnection.connect()` call that connects to the application on Flash Media Server. The HTML file is a container for the SWF file. For more information, see the Flash documentation at www.adobe.com/go/documentation.

Deploy a Flash Lite SWF file or Adobe AIR content the same way you would deploy any Flash Lite or AIR application. For more information, see the Flash Lite and Adobe AIR documentation at www.adobe.com/go/documentation.

By default, Flash Media Server 3.5 and later install with Apache HTTP Server. If you installed and enabled the web server, you can deploy SWF files and HTML files from the same computer on which Flash Media Server is installed. You can also deploy JPG, GIF, and many other file types. For a complete list of the file types Apache serves by default, see the *RootInstall/Apache2.2/conf/httpd.conf* file.

By default, Apache is configured with the following aliases:

Alias	Path
/	<i>RootInstall/webroot</i>
/cgi-bin/	<i>RootInstall/Apache2.2/cgi-bin</i>
/local-cgi-bin/	<i>RootInstall/Apache2.2/local-cgi-bin</i>

To serve files over HTTP, place the files in the *RootInstall/webroot* folder or an appropriate subfolder. To serve CGI programs, place the files in the *RootInstall/Apache2.2/cgi-bin* folder.

You can also add subfolders to these folders. Subfolders cannot use the following names from the *fmshttpd.conf* file: `icons`, `error`, `SWFs`, `vod`, `cgi-bin`, or `local-cgi-bin`. In addition, subfolder names cannot use the following reserved words: `open`, `send`, `idle`, `fcs`, `fms`.

By default, Apache is configured to serve all HTTP content from the *RootInstall/webroot* folder. If you want to provision the server to serve files from application-specific directories, edit the *httpd.conf* Apache configuration file.

Chapter 2: Streaming services

About streaming services

All editions of Flash Media Server version 3 and later provide two streaming services, live and vod. The live service streams live video. The vod service streams recorded video on demand. The live and vod services are signed 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.

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. You cannot modify the server-side code in the streaming services. However, the services have a custom client-side API that lets you call server-side methods.

Both services can be duplicated and renamed to create multiple publishing points. The server can support an unlimited number of service instances.

Flash Media Server 3.5 optionally installs with Apache web server. If you install Apache, you can serve clients over HTTP. You can also choose to serve media over RTMP and HTTP.

Using the live service

About the live service

The live service is a publishing point on Flash Media Server. The following live video sources can publish to the live service:

- [Flash Media Live Encoder](#), version 2.0 or later

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

- Server-side scripts running on Flash Media Interactive Server and Flash Media Development Server (see [“Publishing from server to server”](#) on page 67)
- A custom-built Flash Player or AIR application

You can use any of the following clients to play video streamed from the live service:

- Flash Media Server sample video player (*RootInstall/samples/videoPlayer*)
- Flash FLVPlayback component (*fl.video.FLVPlayback*)
- Flex VideoDisplay component (*mx.controls.VideoDisplay*)
- Any custom-built application that runs in Flash Player, AIR, or Flash Lite 3

On Flash Media Streaming Server, the live service cannot record streams. Also, the live service does not support DVR features. Flash Media Interactive Server and Flash Media Development Server support unsigned (user-created) applications. If you're using one of these server editions, you can modify the live service source code to create your own applications that include DVR functionality. You can also modify the live service to allow live stream recording.

Test the live service

Use Flash Media Live Encoder to test the live service

- 1 Connect a camera to the computer.
- 2 Open Flash Media Live Encoder and click Start.

By default, Flash Media Live Encoder publishes a stream named `livestream` to `rtmp://localhost/live`. Use these settings to connect to the live service if Flash Media Live Encoder is installed on the same computer as the server. If Flash Media Live Encoder is installed on a different computer, replace `localhost` with the Flash Media Server IP address.
- 3 Double-click the `RootInstall/samples/videoPlayer/videoplayer.html` file to open the sample video player in a browser.
- 4 In the sample video player, do one of the following:
 - In the list videos, click “livestream”.
 - Enter `rtmp://localhost/live/livestream`, check the LIVE checkbox, and click PLAY STREAM.

Use a custom Flash Player application to test the live service

- 1 Connect a camera to the computer.
- 2 Do one of the following:
 - Choose Start > All Programs > Adobe > Flash Media Server 3.5 > Flash Media Server Start Screen. Click Interactive to open the live video sample.
 - If you installed Apache web server with Flash Media Server, browse to `http://localhost/` and click Interactive to open the live video sample.
 - Click Publish.

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

Modify the live service

Note: You cannot modify the live service on Flash Media Streaming Server

- 1 Create a folder in the `RootInstall/applications` folder or use the default `RootInstall/live` folder.
- 2 Do one of the following:
 - If you’re using the default `applications/live` folder, move the `main.far` file to a safe location. Copy the `RootInstall/samples/applications/live` `main.asc` file to the `applications/live` folder.
 - If you created a folder in step 1, copy the `main.asc`, `Application.xml`, `allowedHTMLdomains.txt`, and `allowedSWFdomains.txt` files from the `RootInstall/samples/applications/live` folder to the new folder.
- 3 Open the `RootInstall/conf/fms.ini` file in a text editor and edit the `LIVE_DIR` parameter to point to the folder you created in step 1.

To make recordings of live streams available over HTTP or RTMP, set the `LIVE_DIR` parameter to `RootInstall\webroot\live_recorded`. Recordings in this folder are available from the following addresses:

File format	Address
F4V	<pre>http://serverName/live_recorded/fileName.f4v rtmp://serverName/live/live_recorded/mp4:fileName rtmp://serverName/live/live_recorded/mp4:fileName.f4v</pre>
FLV	<pre>http://serverName/live_recorded/fileName.flv rtmp://serverName/live/live_recorded/fileName rtmp://serverName/live/live_recorded/fileName.flv</pre>

The `Application.xml` file in the folder you created in step 1 uses the `LIVE_DIR` parameter in a `<Streams>` tag. The `<Streams>` tag specifies a virtual directory mapping for streams.

- Restart the server.

Disable live services

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

Using the vod service

Streaming media from the vod service

The vod (video on demand) service lets you stream recorded media to clients without building an application or configuring the server. Simply copy recorded media files to the server and clients can stream them. By default, copy media files to the following locations:

- To stream files over RTMP only, copy files to `RootInstall/applications/vod/media`.
- To stream files over RTMP and allow progressive download over HTTP, copy files to `RootInstall/webroot/vod`.

Note: To serve files over HTTP, install Apache.

You can use any of the following clients to play video streamed from the vod service:

- Flash Media Server sample video player (`RootInstall/samples/videoPlayer`)
- Flash FLVPlayback component (`fl.video.FLVPlayback`)
- Flex VideoDisplay component (`mx.controls.VideoDisplay`)
- Any custom-built application that runs in Flash Player, AIR, or Flash Lite 3

Test the vod service

- Double-click the `RootInstall/samples/videoPlayer/videoplayer.html` file to open the sample video player in a browser.
- Click any video in the `SOURCE: /applications/vod/media` list or in the `SOURCE: /webroot/vod` list.
- To stream your own videos to the video player, do one of the following:
 - To stream media over RTMP only, place files in the `RootInstall/applications/vod/media` folder.

- If you installed Apache and want to serve media over RTMP or HTTP, place video files in the *RootInstall/webroot/vod* folder.

4 Enter the address of the video in the STREAM URL text box, check VOD, and click PLAY STREAM.

The following table lists the possible addresses based on file format and protocol:

File format	Address
F4V	http://serverName/vod/fileName.f4v rtmp://serverName/vod/mp4:fileName rtmp://serverName/vod/mp4:fileName.f4v
FLV	http://serverName/vod/fileName.flv rtmp://serverName/vod/fileName rtmp://serverName/vod/fileName.flv

Note: To use localhost as the serverName for HTTP, append the port number 8134, for example, *http://localhost:8134/vod/video.f4v*. The server uses port 8134 internally for HTTP.

5 Choose Start > All Programs > Adobe > Flash Media Server 3.5 > Flash Media Administration Console to open the Administration Console. Log in to the server to see the vod service running.

Note: When you play a video over HTTP, the client does not connect to the vod application. Instead, Apache serves the video to the client.

Vod service configuration parameters

Two parameters in the *RootInstall/conf/fms.ini* file determine the locations of the media folders for the vod application:

```
VOD_COMMON_DIR = C:\Program Files\Adobe\Flash Media Server 3.5\webroot\vod
VOD_DIR = C:\Program Files\Adobe\Flash Media Server 3.5\applications\vod\media
```

The *Application.xml* file in the *RootInstall/applications/vod* folder uses these parameters in <Streams> tags to specify virtual directory mappings for streams. The server looks for media in the order the <Streams> tags are listed in the *Application.xml* file.

If you installed Apache, you can stream media over RTMP and HTTP. Place media files in the folder specified in the *VOD_COMMON_DIR* parameter. To stream media over RTMP only, place media files in the folder specified in the *VOD_DIR* parameter. Media files are available at the following addresses:

File format	Address
F4V	http://serverName/vod/fileName.f4v rtmp://serverName/vod/mp4:fileName rtmp://serverName/vod/mp4:fileName.f4v
FLV	http://serverName/vod/fileName.flv rtmp://serverName/vod/fileName rtmp://serverName/vod/fileName.flv

Duplicate the vod service

The server supports an unlimited number of instances of 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 at *RootInstall/applications/vod2*.

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

- 2 Clients can connect to the vod service at the URL `rtmp://flashmediaserver/vod2`.
- 3 Open the *fms.ini* file (located in *RootInstall/conf*) and do the following:
 - Add a parameter to set the content path for the new service, for example: `VOD2_DIR = C:\Program Files\Adobe\Flash Media Server 3.5\applications\vod2\media`.
 - If you installed Apache and want the media files to be available over HTTP, add a new `VOD2_COMMON_DIR` parameter: `VOD2_COMMON_DIR = C:\Program Files\Adobe\Flash Media Server 3.5\webroot\vod2`.
- 4 Open the *Application.xml* file in the *RootInstall/applications/vod2* folder and do the following:
 - Edit the virtual directory to the following: `<Streams>/;${VOD2_DIR}</Streams>`.
 - Edit the virtual directory to the following: `<Streams>/;${VOD2_COMMON_DIR}</Streams>`.
- 5 Place recorded media files into the following locations:
 - Place files that stream only over RTMP in the `C:\Program Files\Adobe\Flash Media Server 3.5\applications\vod\media` folder.
 - Place files that stream over RTMP or HTTP in the `C:\Program Files\Adobe\Flash Media Server 3.5\webroot\vod2`.

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.

Modify the vod service

Note: You cannot modify the vod service on Flash Media Streaming Server

- 1 Create a folder in the *RootInstall/applications* folder or use the default *RootInstall/vod* folder.
- 2 Do one of the following:
 - If you're using the default *applications/vod* folder, move the *main.far* file to a safe location. Copy the *RootInstall/samples/applications/vod main.asc* file to the *applications/vod* folder.
 - If you created a folder in step 1, copy the *main.asc*, *Application.xml*, *allowedHTMLdomains.txt*, and *allowedSWFdomains.txt* files from the *RootInstall/samples/applications/vod* folder to the new folder.
- 3 Open the *RootInstall/conf/fms.ini* file in a text editor and edit the `VOD_DIR` parameter and the `VOD_COMMON_DIR` parameter to point to the correct locations.
- 4 Restart the server.

Disable vod services

- ❖ Move any vod service folders out of the applications folder.

Creating clients for streaming services

Using the sample video player

The Flash Media Server Start Screen includes an embedded video player. The video player lets you see video streaming from the server immediately after installation. Use the code provided on the Start Screen to embed the video player in your own application.

You can examine the video player at *RootInstall/samples/videoplayer* and modify it as desired.

Using the Flash FLVPlayback component

You can use the FLVPlayback component, available since Flash 8, as a client for the vod and live services. Set the `source` parameter (or `contentPath` parameter in ActionScript 2.0) to the full URL of the stream. To connect to the live service, set the `isLive` parameter to `true`. You can set parameters in the Component inspector or in ActionScript.

Set the `source` (or `contentPath`) parameter to the URL of the stream. Do not include the `/media` folder in the path. For example, the following is the path to a sample file: `rtmp://localhost/vod/sample.flv`.

Using the Flex VideoDisplay component

You can use the Flex VideoDisplay control to play an FLV or F4V file in a Flex application. The component supports progressive download over HTTP, streaming from Flash Media Server, and streaming from a Camera object. Set the `source` parameter to the full URL of the stream. Do not include the `/media` folder in the path. For example, the following is the path to a sample file: `rtmp://localhost/vod/sample`. To stream live video, set the `live` parameter to `true`.

Connecting to a streaming service

Like all Flash Media Server applications, streaming services expect the `NetConnection.connect()` URI to be in the following format:

```
rtmp://hostName/serviceName/[formatType:] [instanceName/]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 For mp3 files, `mp3:`. For MP4/F4V files, `mp4:`. Not required for FLV files.

fileOrStreamName Either a filename (for example, `my_video.mp4`) or a path (for example, `subdir/subdir2/my_video.mp4`), for example,

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

For MPEG-4-based files, if the file on the server uses a filename extension (`.mp4`, `.f4v`, and so on), specify it. If the stream is live and the publisher specified a filename extension, specify it.

Unsupported features

Clients for the vod and live services can use any Flash Player features except remote shared objects (`SharedObject.getRemote()`).

You cannot edit the server-side code for streaming services. However, the services do have a custom API that lets you access information from the server. Call the `NetConnection.call()` method from client-side code and pass it the name of the API you want to call. For more information, see “[Streaming services API](#)” on page 18.

Allow connections from specific domains

By default, clients can connect to the live and vod services from any domain. You can limit the domains from which clients can connect.

- ❖ 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.

Access raw audio and video data in the live service

Note: Flash Media Streaming Server does not support this feature.

Beginning with Flash Media Server 3 and Flash Player 9.0.115.0, you can access raw audio and video data in live streams. Use this data to create snapshots in your applications. To access the data, call the ActionScript 3.0 `BitmapData.draw()` and `SoundMixer.computeSpectrum()` methods. For more information, see *ActionScript 3.0 Language and Components Reference* at www.adobe.com/go/learn_fms_asdoc3_en.

By default, Flash Media Server prevents you from accessing streams. To allow stream access, do the following:

- 1 Move the `main.far` file from `RootInstall/applications/live` to `RootInstall/samples/applications/live`.
You cannot edit FAR files, so you must replace `main.far` with `main.asc`.
- 2 Copy the `main.asc` file from `RootInstall/samples/applications/live` to `RootInstall/applications/live`.
- 3 Open the `main.asc` file in a text editor.
- 4 Uncomment the following code to allow all clients to access all streams:

```
//p_client.audioSampleAccess = "/";  
//p_client.videoSampleAccess = "/";
```

- 5 Save the `main.asc` file.

Streaming services API

getStreamLength()

```
getStreamLength(streamObj)
```

Returns the length of a stream, in seconds. Call this method from a client-side script and specify a response object to receive the returned value.

Availability

Flash Media Server 3, vod streaming service

Parameters

`streamObj` A Stream object.

Returns

A number.

Example

The following client-side code gets the length of the `sample_video` stream and returns the value to `returnObj`:

```
nc.call("getStreamLength", returnObj, "sample_video");
```

getPageUrl()

```
getPageUrl()
```

Returns the URL of the web page in which the client SWF file is embedded. If the SWF file isn't embedded in a web page, the value is the location of the SWF file. The following code shows the two examples:

```
// trace.swf file is embedded in trace.html.  
getPageUrl returns: http://www.example.com/trace.html
```

```
// trace.swf is not embedded in an HTML file.  
getPageUrl returns: http://www.example.com/trace.swf
```

The value must be an HTTP address. For security reasons, local file address (for example, `file:///C:/Flash Media Server applications/example.html`) are not displayed.

Availability

Flash Media Server 3, vod streaming service, live streaming service

Example

The following example calls the `getPageUrl()` method on the server:

```
nc.call("getPageUrl", returnObj);
```

getReferrer()

```
getReferrer()
```

Returns the URL of the SWF file or the server where the connection originated.

Availability

Flash Media Server 3, vod streaming service, live streaming service

Example

The following code calls the `getReferrer()` method on the server:

```
myNetConnection.call("getReferrer", returnObj);
```

Chapter 3: Developing media applications

About 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.

Live video applications capture, encode, and stream live media to the server. The server streams the media to 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.

Connecting to the server

About the NetConnection class

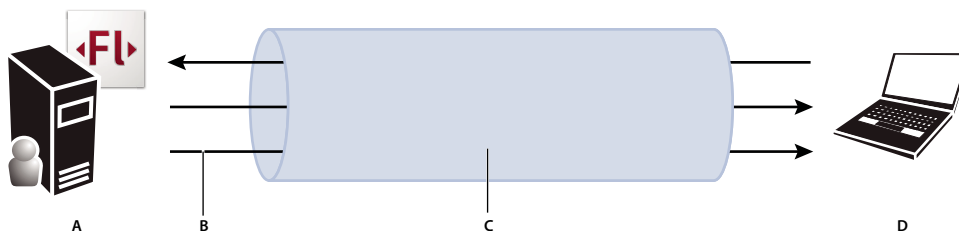
Before a client can play audio and video from Flash Media Server, it must connect to the server. The connection request is accepted or rejected by an application instance on the server, and connection messages are sent back to the client. Once the application accepts the connection request, a connection 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, Adobe AIR, or Flash Lite 3 client

About 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: http:	The protocol used to connect to Adobe Flash Media Server, which is the Adobe Real-Time Messaging Protocol. Possible values are rtmp, rtmpe, rtmpts, rtmpt, rtmpte, and http. 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. If the protocol is http, the default port is 80.
/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: chatroom/room1, chatroom/room2, and so on. If you do not specify an instance name, the client connects to the default application instance, named <code>_definst_</code> .

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.

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.

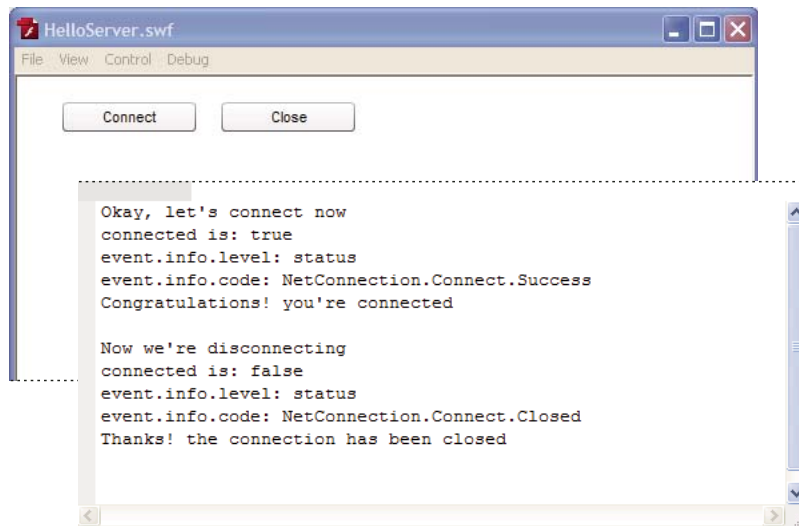
Mapping in Vhost.xml	URI in NetStream call	Location of published stream
<VirtualDirectory><Streams> tag		
low;e:\fmsstreams	"myStream"	c:\...\RootInstall\applications\yourApp\streams_definst_\myStream.flv
low;e:\fmsstreams	"low/myStream"	e:\fmsstreams\myStream.flv
low;\mynetworkDrive\share\fmsstreams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

More Help topics

[“Using the vod service”](#) on page 14

Hello Server application

You can find the HelloServer application in the *RootInstall/documentation/samples/HelloServer* folder. 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");
```

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 CS4 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 CS4 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 action. 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 can have server-side code in a `main.asc` or `applicationName.asc` file that manages clients trying to connect.

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.

More Help topics

[“Writing server-side code”](#) on page 6

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. 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 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
```

Streaming recorded media files

Playing media files on a client

To play a stream, pass a URI to the client-side `NetStream.play()` method 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()`. The `play()` method takes four parameters, with the following syntax:

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

The parameters are as follows:

- `name`. A string that contains the name of a recorded file (the stream name). Use the correct syntax for the stream name depending on the file format of the stream.

File format	Syntax	Example
FLV	Specify the stream name without a file extension.	<code>NetStream.play("myflvstream");</code>
MP3 or ID3	Specify the stream name with the prefix <code>mp3:</code> or <code>id3:</code> , respectively, and without a file extension.	<code>NetStream.play("mp3:mymp3stream");</code> <code>NetStream.play("id3:myid3data");</code>
MPEG-4 based files, such as F4V or MP4	Specify the stream name with the prefix <code>mp4:</code> . The prefix indicates to the server that the file is in MPEG-4 Part 12 container format. If the file on the server uses a filename extension, you must specify it.	<code>NetStream.play("mp4:myvideo.mp4");</code> <code>NetStream.play("mp4:myvideo.f4v");</code>

- `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 `play()` calls from a playlist.

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

Naming streams

Stream names cannot contain any of the following characters: \ / : * ? " < > |.

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 streams

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.

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:

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

More Help topics

“[Example: Add metadata to live video](#)” on page 62

Using XMP metadata

You can deliver Adobe Extensible Metadata Platform (XMP) metadata embedded video streaming through Flash Media Server to Flash Player. Flash Media Server supports XMP metadata embedded in FLV and MP4/F4V formats. Flash Media Server 3.5 supports one XMP metadata packet per MP4/F4V file.

With XMP metadata, you have a communication system that provides critical media information from media creation to the point where media is viewed. XMP information you add during the production process can add to the interactive experience of the media. In addition, speech-to-text metadata embedded within files and encoded from Adobe encoding tools such as Flash Media Live Encoder can be delivered. AMF0 and AMF3 connections are supported. XMP metadata can be internal information about the file or information for end users.

For example, you could create a trailer in Adobe® Premiere® and transfer the metadata to the FLV file. When users view the file, they can use Flash Player 10 search to look for metadata and jump to a specific location in the file. When `NetStream` plays the content, an `onXMPData` message with the single field `data` is sent as a callback. The `data` field contains the entire XMP message from the media file.

For detailed information about XMP, see www.adobe.com/go/learn_fms_xmp_en.

Example: Media player

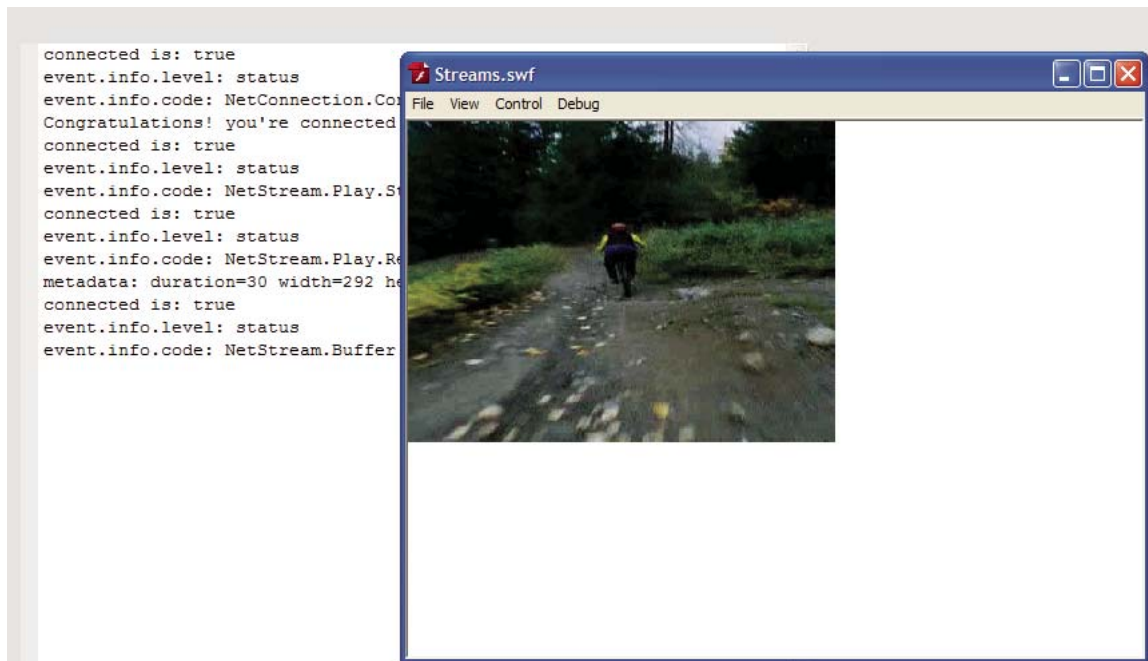
This tutorial uses ActionScript 3.0 to add a Video object to the Stage to display video. For more information about working with video, see the “Working with Video” chapter in *Programming ActionScript 3.0* at www.adobe.com/go/learn_fms_video_en.

Note: This example uses the *MediaPlayer* sample, *MediaPlayer.as*, from the *RootInstall/documentation/samples* folder.

Run the example in Flash

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

- 1 To register the application with the server, create a *RootInstall/applications/mediaplayer* folder.
- 2 Copy the *RootInstall/documentation/samples/MediaPlayer/streams* folder to the *RootInstall/applications/mediaplayer* folder so you have the following:
RootInstall/applications/mediaplayer/streams/_definst_/bikes.flv
- 3 In Flash, open the *MediaPlayer.fla* file from the *RootInstall/documentation/samples/MediaPlayer* folder.
- 4 Select Control > Test Movie. The video plays without sound and the Output window displays messages.



The Output window and the video in test-movie mode

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 example in Flex

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

The video runs in an application window. Click the Flex Builder window to see the output messages.

Write the main client class

- 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, `MediaPlayer`, and declare the variables you'll need within it:

```
public class MediaPlayer extends Sprite  
{  
    var nc:NetConnection;  
    var ns:NetStream;  
    var video:Video;  
    ...  
}
```

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

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

- 4 Create a `netStatusHandler` function that handles both `NetConnection` and `NetStream` events:

```

private function netStatusHandler(event:NetStatusEvent):void{
    trace("event.info.level: " + event.info.level + "\n", "event.info.code: " +
event.info.code);
    switch (event.info.code){
        case "NetConnection.Connect.Success":
            // Call doPlaylist() or doVideo() here.
            doPlaylist(nc);
            break;
        case "NetConnection.Connect.Failed":
            // Handle this case here.
            break;
        case "NetConnection.Connect.Rejected":
            // Handle this case here.
            break;
        case "NetStream.Play.Stop":
            // Handle this case here.
            break;
        case "NetStream.Play.StreamNotFound":
            // Handle this case here.
            break;
        case "NetStream.Publish.BadName":
            trace("The stream name is already used");
            // Handle this case here.
            break;
    }
}

```

Note: To see the full list of event codes that are available, see `NetStatusEvent.info` in the *ActionScript 3.0 Language and Components Reference*.

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

```

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

```

Notice that you set the `client` property to an instance of the `CustomClient` class. `CustomClient` is a separate class that defines some special event handlers.

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

```

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

```

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

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

```

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

```

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:

```
ns.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. Inform customers of errors that occur during playback:

- The network connection fails during playback.
- The buffer empties before playback is complete.

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.

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;
        ...
    }
}
```

Working with 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.

Adobe Evangelist Jens Loeffler has written an [Adobe DevNet tutorial](#) that uses client-side and server-side playlists.

Create a client-side playlist

This playlist uses the 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 *MediaPlayer* sample, *MediaPlayer.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

Swap streams in a playlist

Swapping streams means to exchange one stream for another. While stream switching occurs at keyframes, swapping streams occurs at the stream boundary. Swapping streams is, therefore, useful with playlists. One use case is with playlists that contain content with advertising segments. After collecting statistics about usage patterns, you can swap one advertisement for another.

Use the `NetStream.play2()` method with the transition mode `SWAP` to swap streams in a playlist. The `NetStream.play2()` method takes a `NetStreamPlayOptions` object as a parameter. In the `NetStreamPlayOptions` object, specify the old stream, the stream to switch to, and the kind of transition to use—in this case, `NetStreamPlayTransitions.SWAP`.

Note: The `NetStream.play2()` method is supported in Flash Media Server 3.5 and later and Flash Player 10 and later.

For example, suppose a playlist is set to play Stream A, Stream B, and Stream C, in that order.

```
ns.play("streamA", 0, -1, true);
ns.play("streamB", 0, -1, false);
ns.play("streamC", 0, -1, false);
...
```

While Stream A plays, and before the server begins sending Stream C, you determine that you want to play Stream Z instead of Stream C. To perform this transition, use code like the following sample:

```
var param:NetStreamPlayOptions = new NetStreamPlayOptions();
param.oldStreamName = "streamC";
param.streamName = "streamZ";
param.transition = NetStreamPlayTransitions.SWAP
ns.play2(param);
```

The `SWAP` transition differs from the `SWITCH` transition. The call to swap streams must occur before the server delivers the old stream (in this example, `streamC`). If `streamC` is already in play, the server does not swap the content and sends a `NetStream.Play.Failed` event. If the server has not yet delivered `streamC`, the server swaps the content. The result is that `streamA`, `streamB`, and `streamZ` play.

When the server swaps to a stream with different content, the client application resets the buffer. The server swaps the stream at the start of the new stream, ensuring an uninterrupted experience.

Create a server-side playlist

A server-side playlist is a list of media played in sequence over a Server-Side ActionScript Stream object. A playlist can contain both live and recorded media.

A server-side playlist plays over a live stream (the Stream object). You can record the stream as an FLV or F4V file as it plays. If a playlist contains only FLV files, you can record it as an FLV file or as an F4V file. Otherwise, record it as an F4V file.

The following code creates a server-side playlist of two live streams and one recorded stream:

```
// Start the playlist when the application loads.
// This is a live playlist, it is not recorded.
application.onAppStart = function(){
    this.myStream = Stream.get("serverplaylist");
    // Play a live stream for 30 seconds.
    this.myStream.play("liveStream1", -1, 30);
    // Play a recorded stream in full after liveStream1 plays.
    this.myStream.play("mp4:recordedStream1.f4v", 0, -1, false);
    // Play another live stream for 30 seconds after recordedStream1 plays.
    this.myStream.play("liveStream2", -1, 30, false)
}
```

To play the playlist, call `NetStream.play("serverplaylist")` on the client. To play the playlist smoothly, set the client-side `NetStream.bufferTime` property to at least 1 second. (The default value is 0.1 seconds.)

To add media to a playlist, call `Stream.play()` and pass `false` for the `reset` parameter. When you pass `false`, the media doesn't start playing until the media currently playing has stopped.

The following server-side code plays two recorded media files consecutively and records them to the file "playlist.f4v". Each media file plays in its entirety. To play the playlist, call `NetStream.play("mp4:playlist.f4v")` on the client.

```
application.onAppStart = function(){
    this.clientStream = Stream.get("mp4:playlist.f4v");
    this.clientStream.record();
    this.clientStream.play("mp4:british.mp4", 0, -1);
    this.clientStream.play("mp4:shadows.mp4", 0, -1, false);
};
```



The following example was provided by Jens Loeffler on his blog flashstreamworks.com.

Use a server-side playlist to export a highlight clips during a DVR-enabled live event. Build the playlist (for example, an opening clip, then a selected part of the live event, then a section of an archived clip), and record it as an .f4v file. To play the highlight reel, just point your streaming player to the exported .f4v file.

```
application.myStream = Stream.get("mp4:highlights.f4v");
if (application.myStream){
    application.myStream.record();
    application.myStream.play("mp4:titles.f4v", 0, 15);
    application.myStream.play("livesmith", -1, 30, false);
    application.myStream.play("mp4:smitharchive.mp4", 0, 30, false);
    application.myStream.play("mp4:closing.f4v", 0, 15, false);
}
};
```

To play this example, the client calls `NetStream.play("mp4:highlights.f4v")`.

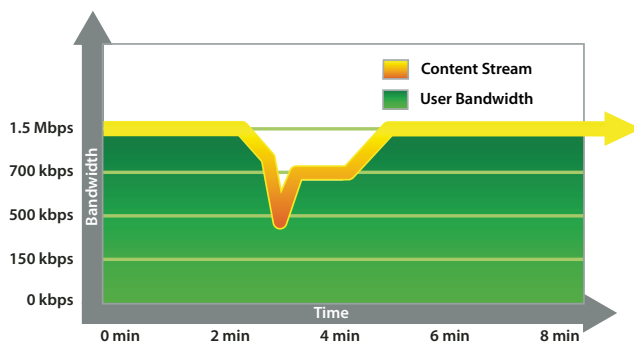
To play a server-side playlist, set the client-side `NetStream.bufferTime` property to at least 1 second. (The default value is 0.1 seconds.)

Dynamic streaming

About dynamic streaming

Note: The dynamic streaming API discussed is supported in Flash Media Server 3.5 and later and Flash Player 10 and later.

Flash Media Server can receive commands to switch between versions of a content stream that are encoded at different bit rates. This feature lets your media application adapt to changing network conditions. It also lets your application adapt to clients with different capabilities, such as mobile devices with lower processing power and smaller screens. For example, suppose the server is streaming high-definition video to a client application but encounters poor network conditions. The server can switch to a standard-definition stream at a lower bit rate. If network conditions improve, the server can switch back to HD video. The transitions occur seamlessly in the client. Although network conditions have changed, the video streaming to the client is uninterrupted.



The server delivers streams that match bandwidth changes to maintain QoS

For optimal user experience, dynamic streaming requires the following:

- The different versions or pieces of content are synchronized: the video timelines must match.
- Audio or other data in each content stream is synchronized with the video data in that stream.

The server implements a transition between two pieces of recorded content in three possible ways, depending on the type of content that is being streamed:

- Video-only streams. Transitions occur at the nearest keyframe in the target timeline.
- Video and audio streams. Transitions occur at the audio sample that immediately precedes the nearest keyframe in the timeline of the target stream. The audio timelines of the initial and target streams must match, or an audio artifact results.
- Audio-only streams. Transitions occur at the nearest possible sample.

Implementing transitions between live video content is slightly more complicated. The streams must have timestamps that are synchronized closely enough, within 3-5 milliseconds, so that the server can select accurate transition points.

Using ActionScript to switch streams

Stream transitions occur on the server, but the command to switch streams and the determination to do so comes from the client application. The application developer includes logic to monitor download and playback statistics and to switch from the old stream to the new stream when appropriate. Developers can use the ActionScript 3.0 `NetStream.info` property and `NetStreamInfo` class to monitor download and playback statistics. The `NetStream.play2()` method and the associated `NetStreamPlayOptions` class offer the ability to change to different streams in mid-play. (Similar APIs are also available in ActionScript 2.0.)

The application developer must also ensure that the client application has a playback buffer that is large enough to absorb any delay caused by the transition. Two factors that can cause a delay are the keyframe interval of the live stream and if the two streams being switched are not synchronized. For example, a 2-second buffer cannot accommodate a 3-second transition delay. Ensure that, after initial playback, the client buffer (the value of `NetStream.bufferTime`) is set to at least the default value of 10 seconds or more.

Adobe developed a new class called `DynamicStream` that extends the `NetStream` class. The `DynamicStream` class contains event listeners that monitor bandwidth, buffer usage, and dropped frames. The class switches streams based on that information. You can use the `DynamicStream` class to implement dynamic streaming, or you can use the `DynamicStream` class as a reference to write your own dynamic streaming algorithms. If you're migrating legacy code, it's a good idea to use the `DynamicStream` class.

Download the ActionScript class files and documentation for these classes from www.adobe.com/go/fms_dynamicstreaming.

Note: These classes are not part of the ActionScript 3.0 library. These are custom classes developed by Adobe for Flash Media Server users.

Determining when to use dynamic streaming

Adobe recommends that you use dynamic streaming for content that meets some or all of the following criteria:

- Video with long duration
- Video with large file size
- HD video
- Video with larger dimensions, such as full screen video
- Content distributed to users who are more susceptible to bandwidth issues, such as home users, rather than corporate users

Encoding recommendations

To provide users with the best experience, when you encode the content, follow the recommendations in this [DevNet article](#). It discusses best practices for encoding multi-bit rate content streams. Here is a summary of the recommendations:

- Ensure that the video timelines of the streams are related and compatible.
- For a truly seamless experience, use the same codecs and audio bitrates in all streams. If you don't, you may hear small audio pops when the streams switch.
- While not required, it is helpful if the keyframe interval (keyframe frequency) and frame rate (fps) are consistent across the different versions of content. A shorter keyframe interval lets the server switch streams more quickly, which means that the client can have a smaller playback buffer.
- Use the same audio sample rate, as much as possible. For low-end encodings, you can save bandwidth by using mono audio, rather than stereo.

The following table shows various bit rates that you could use to encode a single piece of content:

Bit rate
150 kbps
300 kbps

Bit rate
500 kbps
700 kbps
1.5 Mbps (full web HD)

Determining when to switch streams

You can consider many factors when determining when to switch streams, such as the buffer length, number of bytes downloaded, and number of frames dropped. The `DynamicStream` and `DynamicStreamItem` classes, which you can download from adobe.com, are built with these factors in mind and contain the logic required for a dynamic streaming application.

If you prefer to develop your own application logic, it may be helpful to use the following strategy for streaming video:

- 1 For initial playback, select the lowest bit rate that is appropriate for the screen or device. For example, if the video plays in the web browser on a standard computer, an appropriate stream for initial playback is 300 kbps at 320 x 240.
- 2 To start playback quickly, select a small buffer length.
- 3 Once playback begins, increase the buffer length to a larger value, such as 30-60 seconds.
- 4 Begin monitoring the client bandwidth (`NetStream.info.maxBytesPerSecond`) and buffer size (`NetStream.bufferLength`) as it fills.

When current bandwidth is sufficient, the buffer fills quickly and stays steady. If the bandwidth begins to drop, the buffer starts to empty.

- 5 If the client bandwidth exceeds the requirements of the stream and the buffer is filling or is full, you can switch to higher-resolution content.

Verify that client bandwidth is sufficient before switching. In addition to client bandwidth and buffer length, you can check additional statistics, such as the number of dropped frames (`NetStream.info.droppedFrames`).

- 6 After each transition to higher-resolution content, continue to monitor the buffer every 5 seconds, using a timer. If the buffer begins to empty, switch to lower-resolution content and monitor the buffer more frequently, such as after every 2 seconds.
- 7 Continue to upgrade while bandwidth is plentiful and the buffer is filling or full. For the best user experience, be conservative. Upgrade only when the reported bandwidth exceeds stream requirements by a solid margin.

More Help topics

[Using the `DynamicStream` classes](#)

[Dynamic streaming for advanced developers](#)

[Dynamic streaming under the hood](#)

Check client bandwidth

Monitor the client bandwidth to help determine when switching streams is desirable. When client bandwidth is good, the client application can request the server to switch to a higher video bit rate. When client bandwidth is low, the client application can request the server to switch to a lower bit rate.

To measure bandwidth, use the `NetStream.info` property. A call to `NetStream.info` returns a `NetStreamInfo` object with properties that reflect the rate of incoming audio, video, and data bytes of the stream. With information about the incoming data rate, you can deduce the quality of the bandwidth.

Specifically, use the `*byteCount` and `*bytesPerSecond` properties in the `NetStreamInfo` class (or, in ActionScript 2.0, the object returned by `NetStream.getInfo()`). For details on these properties, see the ActionScript Language References.

One way to measure the client bandwidth is to measure the `NetStreamInfo.byteCount` property over a period of time to get the value of bytes per second and when a `NetStream.Buffer.Full` status event is received. This value approximates the maximum bandwidth available. Then compare the available bandwidth to the available bit rates and implement transitions as needed.

Note: The `byteCount` property does not return the same value as `sc-stream-bytes` in the server Access log. The `byteCount` property is a Quality of Service designed to provide data that can help you decide when to switch streams. Do not use the `bytesCount` property for billing.

Check for dropped frames

In addition to monitoring the buffer, check for dropped frames. Switch to a stream with a lower bit rate if too many frames are being dropped. Use the `NetStreamInfo.droppedFrames` property. This read-only property is a number and returns the number of video frames dropped in the current `NetStream` playback session.

One strategy to determine the rate of dropped frames is as follows: using a timer, calculate the difference between the current value of dropped frames and a previous value. Store that difference in a variable, `droppedFPS`. Monitor the current number of incoming frames per second in another variable, `currentFPS`. If `droppedFPS` exceeds 20% of the value of `currentFPS`, switch to a lower bit rate.

Switch streams

To request a transition between streams with the same content encoded at different bit rates, the client application uses the `NetStream.play2()` method. This method takes as a parameter a `NetStreamPlayOptions` object, which indicates how the server switches streams.

Note: The `NetStream.play2()` method extends the `NetStream.play()` method.

The `NetStreamPlayOptions` object contains the following properties:

Property	Description
<code>oldStreamName</code>	The name of the stream currently being played (the old stream).
<code>streamName</code>	The name of the new stream to play; the stream to switch to.
<code>start</code>	The start time of the new stream to play. For most dynamic streaming purposes, the default value of -2 is appropriate. It tells the application to play the live stream specified in <code>streamName</code> . If a live stream of that name is not found, Flash Player plays the recorded stream specified in <code>streamName</code> . If a live or a recorded stream is not found, Flash Player opens a live stream named <code>streamName</code> , even though no one is publishing on it. When someone does begin publishing on that stream, the application begins playing it.
<code>len</code>	The duration (length) of the playback. For most dynamic streaming purposes, the default value of -1 is appropriate. This value means that the application plays a live stream until it is no longer available or plays a recorded stream until it ends.

Property	Description
transition	The transition mode. Possible values are constants in the <code>NetStreamPlayTransition</code> class. The one most applicable to switching between the same content at different bit rates is <code>SWITCH</code> . For information on other modes, see the <code>NetStreamPlayTransition</code> class in the ActionScript 3.0 Language Reference.

The following code example uses the `SWITCH` option to tell the server to switch to a higher bit rate stream. The example does not pass a value for `oldStreamName`, which tells the server to switch to the new stream at the next logical keyframe. This technique provides the smoothest viewing experience. (When using playlists, pass a value for `oldStreamName`; see “[Swap streams in a playlist](#)” on page 36.) In most dynamic stream-switching cases with a recorded video+audio stream, you can keep the default values of `start` and `len`, as in the example.

When the client requests a transition, the server sends a `NetStatusEvent.NET_STATUS` event with the code `NetStream.Play.Transition`. (In ActionScript 2.0, it sends an `onStatus` event with the same code.) The server sends this event to the client almost immediately, which indicates that the operation has succeeded. When the first frames of the new stream render, the server sends an `onPlayStatus` message with the code `NetStream.Play.TransitionComplete`. This event lets the client know exactly when the new stream has started to render.

If a client seeks after Flash Player sends a `NetStream.Play.Transition` message, the streams switch successfully, but Flash Player does not send a `NetStream.Play.TransitionComplete` message. The player doesn't send the message because after the seek it enters a new state and cannot send status events about the old state. The player behaves the same way for other callback methods such as `onMetaData()`.

The following example handles function a stream transition:

```
var stream:NetStream = new NetStream(connection);
stream.addEventListener(NetStatusEvent.NET_STATUS, netStatusHandler);
stream.client = new CustomClient();
var video:Video = new Video();
video.attachNetStream(stream);
stream.play("streamA_150kbps");
addChild(video);
...
//Set up the transition to 700 kpbs content
var param:NetStreamPlayOptions = new NetStreamPlayOptions();
param.streamName = streamA_700kbps;
param.transition = NetStreamPlayTransitions.SWITCH;
stream.play2(param);

//Handler function for the Transition event
class CustomClient{
    public function onPlayStatus(info:Object):void {
        trace("switch time: time=" + info.time + " name=" + info.name + " type=" + info.type);
    }
}
```

The `NetStream.Play.Transition` status event contains a `reason` field. Use this field to get additional information about the state of a transition request. The `reason` field usually contains the code `"NetStream.Transition.Success"`, meaning the transition request succeeded and was processed normally. When switching between live streams, it is possible that the server could not find a synchronization point between the two streams. In this case, the server forces a transition to occur at an arbitrary frame and the `reason` field contains the code `"NetStream.Transition.Forced"`. This situation can occur under the following circumstances:

- The two streams being switched don't have the same timeline, and therefore the server cannot select a time to perform the switch.
- The keyframe interval of the new stream is longer than the client's playback buffer, which is the maximum amount of time the server will wait for a transition. Since the server did not see a keyframe, it cannot select a frame for the switch.
- The live queue delay for the live streams is longer than the client's playback buffer, which creates a delay similar to a long keyframe interval.

Handling metadata during stream transitions

Flash Media Server sends a `NetStream.Play.TransitionComplete` status event when the bits of a stream transition render to the client. When switching to a new stream, the `onMetadata` message for the new stream is sent immediately following the `NetStream.Play.TransitionComplete` status event. Listen for the `TransitionComplete` event before capturing the metadata. If the stream is live, all data keyframes associated with the stream are transmitted.

Setting the client buffer

Buffering manages fluctuations in bandwidth while a video is playing.

To create a good experience for users, set the buffer to a small value initially. A smaller value lets the stream begin playing relatively quickly. Once playback begins, increase the buffer to a larger value. A larger value ensures that the stream plays more smoothly regardless of noise or interruptions on the network.

To create the best experience for users, monitor the progress of a video and manage buffering as the video downloads. Consider setting different buffer sizes for different users, to ensure the best playback experience. One choice is to measure client bandwidth and set an initial buffer size based on it.

Monitor the buffer size in the client to determine when to switch streams. When client bandwidth is good, the amount of data in the buffer grows or the buffer is full. The client can request the server to switch to a higher video bit rate. When client bandwidth is low, the amount of data in the buffer shrinks or the buffer empties. The client can request the server to switch to a lower bit rate.

Call `NetStream.info()` to get a `NetStreamInfoObject` with properties that reflect the current statistics of the stream. The properties that deal with the buffer are the `BufferLength` and `BufferByteLength` properties. For details on these properties, see the *ActionScript 3.0 Language and Components Reference*.

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 has 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 and later no longer clear the buffer when a stream is paused. This feature allows viewers to resume playback without experiencing any hesitation. You can also use `NetStream.pause()` in code to buffer data. You could buffer data while viewers are watching a commercial, for example, and then unpauses the stream when the main video starts. For more information, see the `NetStream.pause()` entry in the *ActionScript Language Reference*.*

Set buffer time

To change the buffer time, in seconds, set the `NetStream.bufferTime` property:

```
ns.bufferTime(10);
```

The right size for the buffer varies depending on user bandwidth, and the following values are only suggestions. 5-10 seconds is a good initial buffer size for fast connections; 10 seconds is a good initial buffer size for slow connections. After playback starts, 30-60 seconds is a good buffer size.

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.

- 1 In the constructor function of your main client 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");
```

- 2 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 = 30;  
            trace("Expanded buffer to 30");  
            break;  
        case "NetStream.Buffer.Empty":  
            ns.bufferTime = 8;  
            trace("Reduced buffer to 8");  
            break;  
    }  
}
```

- 3 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:NetStream = 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 = 8;
    trace("Set an initial buffer time of 8 seconds");

    addChild(video);
}

```

4 Create 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()`.

Recognizing transitions in log files

You can track stream events through access logs. Use the logs to differentiate a single stream play with transitions from multiple plays of a stream or different streams. When a transition for a single content stream occurs, the server tracks the status of the stream as a transition. The server logs a stop event for the original stream and a play event for the new stream. Normal stop and play events (that is, a stop or play without a transition) have a status code of 200. Stream transitions have a status code of 210. The access logs provide the following additional information:

Field	Description
x-sid	The ID of a stream. This ID is unique for the client session but not across sessions
x-trans-sname	The name of the stream that the server is transitioning from (the original stream)
x-trans-sname-query	The query stream portion of the stream name for the stream that the server is transitioning from
x-trans-file-ext	The filename extension portion of the stream name for the stream that the server is transitioning from

If you use a log processor, ensure that it looks at both the status codes and the x-sid values. Look at both values to recognize that a transition occurred on a single logical stream. As with normal streams, stream transitions occur in play/stop pairs. Your log processor can track stream transitions by recognizing stop events that have a 210 status code followed by play events with a 210 status code on the same stream. By looking at the status code, the log processor can also differentiate a stream transition from a play or stop event without a transition.

Detecting bandwidth

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 in 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 is required by native bandwidth detection. It takes an argument, `... rest`. The function must return a value, even if the value is 0, to indicate to the server that the client has received the data. You can call `onBWCheck()` multiple times.

The server calls the `onBWDone()` function when it finishes measuring the bandwidth. It takes four arguments. The first argument it returns is the bandwidth measured in Kbps. The second and third arguments are not used. The fourth argument is the latency in milliseconds.

This class is a client to your main ActionScript 3.0 class.

Write the main 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 or Flex Builder. 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.

- 1 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  
    }  
}
```

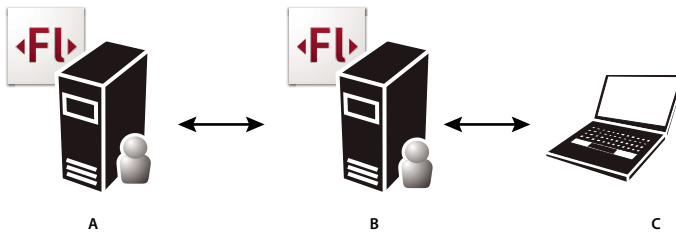
- 2 Create a `NetConnection` object and connect to the server:

```
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, especially if you want to reuse existing code.

If you use edge servers, native bandwidth detection is performed at the outermost edge server to reduce the load on the origin servers. However, script-based 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 server and the edge server can affect the bandwidth measurement.
A. Origin server B. Edge server C. Client

To use script-based bandwidth detection, use the main.asc file Adobe provides for bandwidth detection. You can find the main.asc file with the vod application. You might also need to make some changes in your ActionScript 3.0 client (see the BandwidthServer.as (ActionScript 3.0) sample). Keep in mind that this bandwidth detection is not as reliable as connecting to the vod application or using native bandwidth detection.

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](#)” on page 46). 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 bandwidth from server-side code

The following Server-Side ActionScript code initiates bandwidth detection from the server:

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

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

Detecting stream length

About detecting stream length

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

Pass the stream name to the `Stream.length()` method. You can pass a virtual stream name or a stream name in a URI relative to the application instance.

For example, the following code gets the length of a stream located in the `streams/_definst_` folder of an application:

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

Get the length of a stream

This example uses Server-Side ActionScript to get 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”](#) on page 9.

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 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");
    }
}
}
```

Chapter 4: Working with live video

Adobe Flash Media Server can broadcast live audio and video content to Flash Player, AIR, and Flash Lite clients. To capture and encode live content and stream it to Flash Media Server, you can use Adobe Flash Media Live Encoder or build a custom Flash Player or AIR application.

You can capture live events in real time and stream them to large audiences or create social media applications that include live audio and video. For example, Adobe® Acrobat® Connect™ Pro is a web conferencing application that uses Flash Media Server to capture and broadcast live audio and video.

Capturing live video

Using Flash Media Live Encoder to capture video

Flash Media Live Encoder is a free application that captures live video, encodes it, and streams it to Flash Media Server. By default, Flash Media Live Encoder is configured to stream video to the live service at `rtmp://localhost/live`.

Flash Media Server 3.5 installs with a sample video player that can play streams from the live service. The video player is installed to the folder `RootInstall\samples\`. If you installed the Apache web server, you can access the video player from the Start Screen at `http://localhost`.

You can also use the FLVPlayback component in Flash as a video player. To learn how, see *Beginner's guide to streaming live video with Flash Media Server 3* at www.adobe.com/go/learn_fms_livestream_en.

For more information about Flash Media Live Encoder, see www.adobe.com/go/fme.

More Help topics

“[Using the live service](#)” on page 12

Example: Custom video capture application

The following steps build an application in ActionScript 3.0 that:

- Captures and encodes video.
- Displays the video as it's captured.
- Streams video from the client to Flash Media Server.
- Streams video from Flash Media Server back to the client.
- Displays the video streamed from the server.

Note: To test this code, create a `RootInstall/applications/publishlive` folder on the server. Open the `RootInstall/documentation/samples/publishlive/PublishLive.swf` file to connect to the application.

1. In a new `.as` file, create a `NetConnection` object. To connect to the server, pass the URI of the application to the `NetConnection.connect()` method.

```

var nc:NetConnection;
var ns:NetStream;
var nsPlayer:NetStream;
var vid:Video;
var vidPlayer:Video;
var cam:Camera;
var mic:Microphone;

nc = new NetConnection();
nc.addEventListener(NetStatusEvent.NET_STATUS, onNetStatus);
nc.connect("rtmp://localhost/publishlive");

```

2. Define a function to handle `NetStatusEvent` events. If the client makes a successful connect to the server, the code calls functions that run the application.

```

function onNetStatus(event:NetStatusEvent):void{
    trace(event.info.code);
    if(event.info.code == "NetConnection.Connect.Success"){
        publishCamera();
        displayPublishingVideo();
        displayPlaybackVideo();
    }
}

```

3. Publish the video being captured by the camera and the audio being captured by the microphone. First, get references to the camera and microphone data. Create a `NetStream` object on the `NetConnection` you made to the server. Then call `NetStream.attachCamera()` to attach the captured video to the `NetStream` object. Call `NetStream.attachAudio()` to attach the live audio. Finally, call `NetStream.publish("streamName", "live")` to send the audio and video to the server.

```

function publishCamera() {
    cam = Camera.getCamera();
    mic = Microphone.getMicrophone();
    ns = new NetStream(nc);
    ns.attachCamera(cam);
    ns.attachAudio(mic);
    ns.publish("myCamera", "live");
}

```

4. Display the video the client is streaming to the server. Create a `Video` object. The `Video` object is a display object. Call `Video.attachCamera(cam)` to attach the camera video feed to the video object. Call `addChild(vid)` to add the `Video` object to the display list so that it appears in Flash Player.

```

function displayPublishingVideo():void {
    vid = new Video();
    vid.x = 10;
    vid.y = 10;
    vid.attachCamera(cam);
    addChild(vid);
}

```

5. Display the video the server is streaming to the client. A client that creates a stream to send to a server is *publishing*, while a client playing a stream is *subscribing*. The client in this example publishes a stream and subscribes to a stream. The client must create two streams, an outgoing (publishing) stream and an incoming (subscribing) stream. In this example, the outgoing stream is `ns` and the incoming stream is `nsPlayer`. You can use the same `NetConnection` object for both streams.

To play the audio and video as it returns from the server, call `NetStream.play()` and pass it the name of the stream you published. To display the incoming video, call `Video.attachNetStream()`:

```
function displayPlaybackVideo():void{
    nsPlayer = new NetStream(nc);
    nsPlayer.play("myCamera");
    vidPlayer = new Video();
    vidPlayer.x = cam.width + 20;
    vidPlayer.y = 10;
    vidPlayer.attachNetStream(nsPlayer);
    addChild(vidPlayer);
}
```

Use `ActionScript` to allow users to control playback, display video fullscreen, use cuepoints, and use metadata. For more information, see the “Working with video” chapter in *Programming ActionScript 3.0* at www.adobe.com/go/learn_fms_video_en.

This example was built in Flash. To see an example built in Flex, download the FMS Feature Explorer AIR application at www.adobe.com/go/fms_featureexplorer. Follow the instructions to download a ZIP file of the sample applications and extract them to the applications folder on your server. Once the applications are registered with the server, browse in the FMS Feature Explorer to the AVControl > Publish Live Video application. You can view the source code in a viewer built in to the application.

Adding DVR features to live video

About DVR support

Note: Flash Media Streaming Server does not support DVR functionality.

A DVR (digital video recorder) lets viewers pause live video and resumes playback from the paused location. Viewers can also rewind a live event, play the recorded section, and seek to the live section again. Flash Media Server 3.5 adds support for DVR features. Just write a few lines of code and a video player can act like a DVR. Examples of DVR applications are instant replay and “catch-up” services.

Note: Adobe is updating the FLVPlayback component to support DVR features. Check the Flash Media Server Help and Support Center at www.adobe.com/go/learn_fms_docs_en to see if the component is available.

Publishing video for DVR playback

When you play a stream with DVR features, you do not play a live stream, you play a recorded stream. When you’re viewing the stream “live”, you’re really viewing the recorded stream just after it was recorded.

To publish a stream for a DVR video player from a client, use the following code in your SWF file or AIR application:

```
NetStream.publish("myvideo", "record")
NetStream.publish("myvideo", "append")
```

To publish a stream for a DVR video player from the server, use the following server-side code:

```
Stream.record("record")
```

The `Stream.record()` method has two new parameters, `maxDuration` and `maxSize`, that let you specify the maximum length and file size of a stream. The following code publishes a stream with a maximum recording length of 10 mins (600 seconds) and an unlimited file size:

```
Stream.record("record", 600, -1)
```

To subscribe to a stream published for a DVR video player, use the following code:

```
NetStream.play("myvideo", 0, -1)
```

The previous code allows viewers joining an event late to view from the beginning.

To return to the beginning of a stream at any time, call the following:

```
NetStream.seek(0)
```

To start recording in the middle of an event, call the Server-Side ActionScript `Stream.record()` method. Calling this method lets you start and stop recording at any time.

To create a button that seeks to the latest available part of the recording (which is considered “live”), seek to a large number, for example:

```
NetStream.seek(1000000)
```

Using Flash Media Live Encoder to capture video for DVR playback

You can use Flash Media Live Encoder 3 to capture video for DVR playback. Earlier versions of Flash Media Live Encoder do not support recording to the server. For more information, see <http://www.adobe.com/go/fme>.

Example: Custom capture, publish, and DVR playback

This example is a client application that does the following:

- Captures and encodes video.
- Displays the video as it’s captured.
- Streams video from the client to Flash Media Server.
- Streams video from Flash Media Server back to the client.
- Displays the video streamed from the server in a player that lets you rewind and pause live video.

Note: To test this code, create a `RootInstall/applications/dvr` folder on the server. Open the `RootInstall/documentation/samples/dvr/DVR.swf` file to connect to the application.

- 1 On Flash Media Server, create a `RootInstall/applications/dvr` folder.
- 2 In Flash, create an ActionScript file and save it as `DVR.as`.
- 3 Copy and paste the following code into the Script window:

```
package {
    import flash.display.MovieClip;
    import flash.net.NetConnection;
    import flash.events.*;
    import flash.net.NetStream;
    import flash.media.Video;
    import flash.media.Camera;
    import flash.media.Microphone;
    import fl.controls.Button;
    public class DVR extends MovieClip
    {
        private var nc:NetConnection;
        private var ns:NetStream;
        private var nsPlayer:NetStream;
        private var vid:Video;
        private var vidPlayer:Video;
        private var cam:Camera;
        private var mic:Microphone;
        private var pauseBtn:Button;
        private var rewindBtn:Button;
        private var playBtn:Button;
        private var dvrFlag:Boolean;

        public function DVR()
        {
            nc = new NetConnection();
            nc.addEventListener(NetStatusEvent.NET_STATUS, onNetStatus);
            nc.connect("rtmp://localhost/dvr");
            setupButtons();
            dvrFlag = true;
        }
        private function onNetStatus(event:NetStatusEvent):void{
            trace(event.info.code);
            switch(event.info.code) {
                case "NetConnection.Connect.Success":
                    publishCamera();
                    displayPublishingVideo();
                    displayPlaybackVideo();
                    break;
                case "NetStream.Play.Start":
                    trace("dvrFlag " + dvrFlag);
                    if(dvrFlag){
                        nsPlayer.seek(1000000);
                        dvrFlag = false;
                    }
                    break;
            }
        }

        private function onAsyncError(event:AsyncErrorEvent):void{
            trace(event.text);
        }

        private function onClick(event:MouseEvent):void {
            switch(event.currentTarget) {
                case rewindBtn:
                    nsPlayer.seek(nsPlayer.time - 5);
            }
        }
    }
}
```

```
        break;
    case pauseBtn:
        nsPlayer.pause();
        break;
    case playBtn:
        nsPlayer.resume();
        break;
    }
}

private function publishCamera() {
    cam = Camera.getCamera();
    mic = Microphone.getMicrophone();
    ns = new NetStream(nc);
    ns.client = new CustomClient();
    ns.addEventListener(NetStatusEvent.NET_STATUS, onNetStatus);
    ns.addEventListener(AsyncErrorEvent.ASYNC_ERROR, onAsyncError);
    ns.attachCamera(cam);
    ns.attachAudio(mic);
    ns.publish("myvideo", "record");
}

private function displayPublishingVideo():void {
    vid = new Video(cam.width, cam.height);
    vid.x = 10;
    vid.y = 10;
    vid.attachCamera(cam);
    addChild(vid);
}

private function displayPlaybackVideo():void{
    nsPlayer = new NetStream(nc);
    nsPlayer.client = new CustomClient();
    nsPlayer.addEventListener(NetStatusEvent.NET_STATUS, onNetStatus);
    nsPlayer.addEventListener(AsyncErrorEvent.ASYNC_ERROR, onAsyncError);
    nsPlayer.play("myvideo", 0, -1);
    vidPlayer = new Video(cam.width, cam.height);
    vidPlayer.x = cam.width + 20;
    vidPlayer.y = 10;
    vidPlayer.attachNetStream(nsPlayer);
    addChild(vidPlayer);
}

private function setupButtons():void {
    rewindBtn = new Button();
    pauseBtn = new Button();
    playBtn = new Button();

    rewindBtn.width = 50;
    pauseBtn.width = 50;
    playBtn.width = 50;
    rewindBtn.move(180,150);
```

```

        pauseBtn.move(235,150);
        playBtn.move(290,150);
        rewindBtn.label = "Rewind";
        pauseBtn.label = "Pause";
        playBtn.label = "Play";

        rewindBtn.addEventListener(MouseEvent.CLICK, onClick);
        pauseBtn.addEventListener(MouseEvent.CLICK, onClick);
        playBtn.addEventListener(MouseEvent.CLICK, onClick);
        addChild(rewindBtn);
        addChild(pauseBtn);
        addChild(playBtn);
    }
}
}
class CustomClient {
    public function onMetaData(info:Object):void {
        trace("metadata: duration=" + info.duration);
    }
}
}

```

- 4 Save the DVR.as file.
- 5 Choose File > New > Flash File (ActionScript 3.0) and click OK.
- 6 Save the file as DVR.fla in the same folder as the DVR.as file.
- 7 Open the Components Panel, drag a Button to the Stage, and delete it.
This action adds the button to the Library. The button is added to the application at runtime.
- 8 Choose File > Publish Settings. Click the Flash tab. Click Script Settings and enter DVR as the Document class. Click the checkmark to validate the path.
- 9 Save the file and choose Control > Test Movie to run the application.

Limiting the size and duration of recordings

You can limit the maximum size and duration of recordings using parameters in the Application.xml configuration file, Server-Side ActionScript, and the Authorization plug-in. Set these values to prevent using excessive disk space. The following are the Application.xml file parameters:

XML element	Description
Application/StreamManager/Recording/MaxDuration	The maximum duration of a recording, in seconds. The default value is -1, which enforces no maximum duration.
Application/StreamManager/Recording/MaxDurationCap	The maximum duration of a recording cap, in seconds. The default value is -1, which enforces no cap on maximum duration. The server-side <code>Stream.record()</code> method cannot override this value. The Authorization plug-in can override this value.
Application/StreamManager/Recording/MaxSize	The maximum size of a recording, in kilobytes. The default value is -1, which enforces no maximum size.
Application/StreamManager/Recording/MaxSizeCap	The maximum size of a recording cap, in kilobytes. The default value is -1, which enforces no cap on maximum size. The server-side <code>Stream.record()</code> method cannot override this value. The Authorization plug-in can override this value.

In addition to setting maximum values, in the `Application.xml` file you can set maximum cap values. Server-side scripts cannot override these caps. CDNs can use these caps to set a limit that clients cannot override.

Note: *The Authorization plug-in can override any values set in the `Application.xml` file.*

To set values in Server-Side ActionScript, call `Stream.record()` and pass values for the `maxDuration` and `maxSize` parameters. The following code limits the recording to 5 minutes and sets an unlimited maximum file size (up to the value of `MaxSizeCap`):

```
s.record("record", 300, -1);
```

The server truncates recordings greater than `MaxCapSize` and `MaxCapDuration`.

Scaling DVR applications

To build large-scale applications, use the server-side `NetConnection` class to chain multiple servers together. In this scenario, a client can request a stream that does not reside on the server to which it is connected. Use the server-side `ProxyStream` class to create a look-up mechanism for finding streams in the server chain.

You can set values in the `Vhost.xml` configuration file to configure the disk cache that holds the streams:

XML element	Attribute	Description
<code>VirtualHost/Proxy/CacheDir</code>	<code>enabled</code>	Determines whether the disk cache is enabled.
	<code>useAppDir</code>	Specifies whether to separate the cache subdirectories by application.
<code>VirtualHost/Proxy/CacheDir/Path</code>		The root directory of the disk cache.
<code>VirtualHost/Proxy/CacheDir/MaxSize</code>		The maximum size of the disk in gigabytes. The default value is 32. The value 0 disables the disk cache. The value -1 specifies no maximum value.
<code>VirtualHost/Proxy/RequestTimeout</code>		The maximum amount of time to wait for a response to a request (for metadata, content, and so on) from an upstream server., in seconds. The default value is 2 seconds.

If a server has multiple virtual hosts, point each virtual host to its own cache directory.

If the server runs out of disk space on an intermediate or edge server while writing to the `CacheDir`, it logs the following warning message in the `core.xx.log` for each segment that fails to write to disk: I/O Failed on cached stream file C:\Program Files\Adobe\Flash Media Server 3.5\cache\streams\00\proxyapp\10.192.16.125\C\Program Files\Adobe\Flash Media Server 3.5_361\applications\primaryapp\streams_definst_sample1_1500kbps.f4v\0000000000000000 during write: 28 No space left on device.

Logging

Streams played in a DVR video player are played as recorded streams. These streams log the same events in the log files as every recorded stream.

More Help topics

[“Dynamic streaming”](#) on page 38

Adding metadata to a live stream

About metadata

Flash Media Server can add data messages to the beginning of a live video stream. This metadata can contain information about the video, such as title, copyright information, duration of the video, or creation date. Metadata can also contain advertising information or current statistics like sports scores. Any client connecting to the server, even if they connect late, 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.

Note: Because DVR applications use recorded streams, you do not need to use data keyframes to push metadata to the client. In DVR applications (and in all recorded video applications) the `onMetaData()` method is called at the beginning of the stream and during events such as seek and pause.

Working with metadata

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

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

The `onMetaData` parameter specifies a function 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.

To add metadata to a live stream in a client-side script, use the following code:

```
var metaData:Object = new Object();
metaData.title = "myStream";
metaData.width = 400;
metaData.height = 200;
ns.send("@setDataFrame", "onMetaData", metaData);
```

To clear metadata from a live stream in a client-side script, use the following code:

```
ns.send("@clearDataFrame", "onMetaData");
```

To add metadata to a live stream in a server-side script, use the following code:

```
s = new Stream(nc);
s.onStatus = function(info){
    if (info.code == "NetStream.Publish.Start"){
        metaData = new Object();
        metaData.title = "myStream";
        metaData.width = 400;
        metaData.height = 200;
        this.send("@setDataFrame", "onMetaData", metaData);
    }
};
s.publish("myStream");
```

To clear metadata from a live stream in a server-side script, use the following code:

```
s.send("@clearDataFrame", "onMetaData");
```

To retrieve metadata from a live stream in a client-side script, use the following code:

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

Example: Add metadata to live video

This example is a client application that does the following:

- Captures and encodes video.
- Displays the video as it's captured.
- Streams video from the client to Flash Media Server.
- Sends metadata to the server that the server sends to clients when they play the live stream.
- Streams video from Flash Media Server back to the client when you click a button.
- Displays the video streamed from the server.
- Displays the metadata sent from the server in a TextArea component.

Note: To test this code, create a *RootInstall/applications/publishlive* folder on the server. Open the *RootInstall/documentation/samples/metadata/Metadata.swf* file to connect to the application.

- 1 On Flash Media Server, create a *RootInstall/applications/publishlive* folder.
- 2 In Flash, create an ActionScript file and save it as *Metadata.as*.
- 3 Copy and paste the following code into the Script window:

```
package {  
    import flash.display.MovieClip;  
    import flash.net.NetConnection;  
    import flash.events.NetStatusEvent;  
    import flash.events.MouseEvent;  
    import flash.events.AsyncErrorEvent;  
    import flash.net.NetStream;  
    import flash.media.Video;  
    import flash.media.Camera;  
    import flash.media.Microphone;  
    import fl.controls.Button;  
    import fl.controls.Label;  
    import fl.controls.TextArea;  
    public class Metadata extends MovieClip {  
        private var nc:NetConnection;  
        private var ns:NetStream;  
        private var nsPlayer:NetStream;  
        private var vid:Video;  
        private var vidPlayer:Video;  
        private var cam:Camera;  
        private var mic:Microphone;  
        private var clearBtn:Button;  
        private var startPlaybackBtn:Button;  
        private var outgoingLbl:Label;  
        private var incomingLbl:Label;  
        private var myMetadata:Object;
```

```

private var outputWindow:TextArea;

public function Metadata(){
    setupUI();
    nc = new NetConnection();
    nc.addEventListener(NetStatusEvent.NET_STATUS, onNetStatus);
    nc.connect("rtmp://localhost/publishlive");
}

/*
 * Clear the MetaData associated with the stream
 */
private function clearHandler(event:MouseEvent):void {
    if (ns){
        trace("Clearing MetaData");
        ns.send("@clearDataFrame", "onMetaData");
    }
}

private function startHandler(event:MouseEvent):void {
    displayPlaybackVideo();
}

private function onNetStatus(event:NetStatusEvent):void {
    trace(event.target + ": " + event.info.code);
    switch (event.info.code)
    {
        case "NetConnection.Connect.Success":
            publishCamera();
            displayPublishingVideo();
            break;
        case "NetStream.Publish.Start":
            sendMetadata();
            break;
    }
}

private function asyncErrorHandler(event:AsyncErrorEvent):void {
    trace(event.text);
}

private function sendMetadata():void {
    trace("sendMetaData() called")
    myMetadata = new Object();
    myMetadata.customProp = "Welcome to the Live feed of YOUR LIFE, already in progress.";
    ns.send("@setDataFrame", "onMetaData", myMetadata);
}

private function publishCamera():void {
    cam = Camera.getCamera();
    mic = Microphone.getMicrophone();
    ns = new NetStream(nc);
    ns.client = this;
    ns.addEventListener(NetStatusEvent.NET_STATUS, onNetStatus);
    ns.addEventListener(AsyncErrorEvent.ASYNC_ERROR, asyncErrorHandler);
    ns.attachCamera(cam);
    ns.attachAudio(mic);
    ns.publish("myCamera", "live");
}

```

```
private function displayPublishingVideo():void {
    vid = new Video(cam.width, cam.height);
    vid.x = 10;
    vid.y = 10;
    vid.attachCamera(cam);
    addChild(vid);
}

private function displayPlaybackVideo():void {
    nsPlayer = new NetStream(nc);
    nsPlayer.client = this;
    nsPlayer.addEventListener(NetStatusEvent.NET_STATUS, onNetStatus);
    nsPlayer.addEventListener(AsyncErrorEvent.ASYNC_ERROR, asyncErrorHandler);
    nsPlayer.play("myCamera");
    vidPlayer = new Video(cam.width, cam.height);
    vidPlayer.x = cam.width + 100;
    vidPlayer.y = 10;
    vidPlayer.attachNetStream(nsPlayer);
    addChild(vidPlayer);
}

private function setupUI():void {
    outputWindow = new TextArea();
    outputWindow.move(250, 175);
    outputWindow.width = 200;
    outputWindow.height = 50;

    outgoingLbl = new Label();
    incomingLbl = new Label();
    outgoingLbl.width = 150;
    incomingLbl.width = 150;
    outgoingLbl.text = "Publishing Stream";
    incomingLbl.text = "Playback Stream";
    outgoingLbl.move(30, 150);
    incomingLbl.move(300, 150);

    startPlaybackBtn = new Button();
    startPlaybackBtn.width = 150;
    startPlaybackBtn.move(250, 345);
    startPlaybackBtn.label = "View Live Event";
}
```

```

startPlaybackBtn.addEventListener(MouseEvent.CLICK, startHandler);

clearBtn = new Button();
clearBtn.width = 100;
clearBtn.move(135,345);
clearBtn.label = "Clear Metadata";
clearBtn.addEventListener(MouseEvent.CLICK, clearHandler);

addChild(clearBtn);
addChild(outgoingLbl);
addChild(incomingLbl);
addChild(startPlaybackBtn);
addChild(outputWindow);
}

public function onMetaData(info:Object):void {
    outputWindow.appendText(info.customProp);
}
}
}

```

- 4 Save the file.
- 5 Choose File > New > Flash File (ActionScript 3.0) and click OK.
- 6 Save the file as Metadata.fla in the same folder as the Metadata.as file.
- 7 Open the Components Panel, drag a Button and a TextArea component to the Stage, and delete them.
This action adds the components to the Library. The components are added to the application at runtime.
- 8 Choose File > Publish Settings. Click the Flash tab. Click Script Settings and enter Metadata as the Document class. Click the checkmark to validate the path.
- 9 Save the file and choose Control > Test Movie to run the application.

Metadata properties for live streams

Flash Media Live Encoder sets the following metadata properties and values. 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.
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.

Metadata property name	Data type	Description
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 10 HE-AAC 11 Speex
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 7 H.264
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, 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 10 HE-AAC 11 Speex
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 7 H.264

Metadata property name	Data type	Description
canSeekToEnd	Boolean	Whether the last video frame is a keyframe (<code>true</code> if yes, <code>false</code> 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 and record the stream by adding "record" to the `NetStream.publish()` call, the server attempts to merge your metadata properties with the standard metadata properties. If there is a conflict between the two, the server uses the standard metadata properties. For example, suppose you add these metadata properties:

```
duration=5
x=200
y=300
```

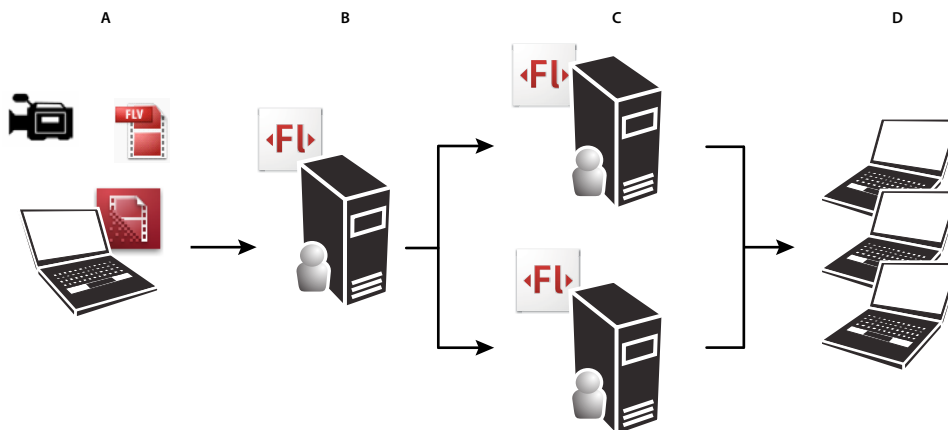
When the server starts to record the video, it begins to write its own metadata properties to the file, including `duration`. If the recording is 20 seconds long, the server adds `duration=20` to the metadata, overwriting the value you specified. However, `x=200` and `y=300` are still saved as metadata, 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.

When a stream is recorded, the recording starts at a keyframe. Recordings stop immediately on keyframes or I-frames.

Publishing 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 as follows:

- 1 A client connects to an application on Server 1 in New York City and calls `NetStream.publish()` to publish a live stream. This client can be a custom Flash Player or AIR application or Flash Media Live Encoder.

- 2 The server-side script on Server 1 receives an `application.onPublish()` event with the name of the published stream.
- 3 The `application.onPublish()` handler creates a `NetStream` object and calls `NetStream.publish()` to republish the live stream to Server 2 (Chicago) and Server 3 (Los Angeles).
- 4 Subscribers connecting to Server 2 and Server 3 receive the same live stream.
- 5 The applications receive `application.onUnpublish()` events when the clients stops publishing.

Example: Multipoint publishing

In this example, the client captures, encodes, and publishes the stream to the server. You can also use Flash Media Live Encoder for the same purpose.

Note: To test this code, create a `RootInstall/applications/livestreams` folder on the server. Open the Administration Console and create an instance of the `livestreams` application. Click `Live Logs` to see the server-side `trace()` statements as the application runs. Open the `RootInstall/documentation/samples/livestreams/LiveStreams.swf` file to connect to the application.

- 1 In a client-side script, call the `NetStream.publish()` method to publish a live stream:

```
ns.publish("localnews", "live");
```

Note: To use Flash Media Live Encoder as a publishing client, enter the FMS URL

`rtmp://localhost/livestreams` and the Stream `localnews`.

- 2 In the server-side `main.asc` file, define an `application.onPublish()` event handler. This handler accepts the stream name that was published from the client, connects to the remote server, and republishes the stream to the remote server. (In this example, the remote server is another instance of the same application).

```
// 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 livestreams/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" );
    }
}
```

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

- 3 In the `main.asc` file, handle events that occur on the `NetStream` object you used to publish from your server to the remote server:

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

The server-side `NetStream.publish()` method triggers a `NetStatus` event with a `NetStream.Publish.Start` code, just like the client-side `NetStream.publish()` method.

4 Define what happens when the client stops publishing:

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

Chapter 5: Developing interactive applications

About interactive applications

In addition to streaming video applications, Adobe Flash Media Interactive Server and Adobe Flash Media Development Server can host interactive 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.

The majority of the topics in this section are 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](#)” on page 72, 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.

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](#)” on page 72.

- 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:

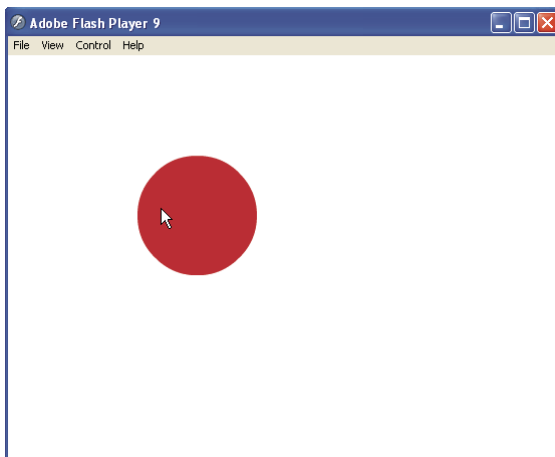
```
sharedBall.x = so.data.x;
```

This is usually done in a `sync` event handler, as shown in “[SharedBall example](#)” on page 72.

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.*



The SharedBall application running in Flash Player

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 Flash, 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 Professional, 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.MOUSE_DOWN, pickup);
    sharedBall.addEventListener(MouseEvent.MOUSE_UP, place);
    sharedBall.addEventListener(MouseEvent.MOUSE_MOVE, 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 or 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. These 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.

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 in Flash Player 9 Update 3 and later.

Allow or 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 Flash Remoting, see 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 74
- Access plug-in 77
- Administration Console 3, 8, 34
- Adobe Flash Media Development Server 1
- Adobe Flash Media Interactive Server 1
- Adobe Flash Media Streaming Server 1
- Adobe Real-Time Messaging Protocol (RTMP) 1
- allowedHTMLdomains.txt file 18
- allowedSWFdomains.txt file 18
- Application class
 - application.onConnect event 26
 - application.onDisconnect event 26
- applications
 - client-side code 6
 - connection URI 20
 - debugging 7
 - double-byte 7
 - Hello World sample 3
 - HelloServer sample 22
 - interactive 70
 - live service 12
 - media 2, 20
 - registering with the server 9
 - server-side code 6
 - using video player in 17
 - vod service 14
- audio data, accessing 18
- authentication
 - clients 75
 - domains 77
 - Flash Player version 77
 - using a token 80
 - using external resource 79

B

- bandwidth 40, 46, 48
- buffering 44

C

- checkBandwidth function 46
- Client class
 - ip property 75
 - readAccess property 74
 - writeAccess property 74

clients

- authenticating 75
- creating 17
- client-side code
 - about 6
 - debugging 7
- client-side playlists 35
- components 17
- connections 17, 25, 34

D

- debugging
 - Administration Console sessions 9
 - client-side code 7
 - server-side code 8
- domains 18, 77
- double-byte applications 7
- dropped frames 41
- dynamic streaming 38, 39

E

- error handling 33
- events
 - netStatus 34
 - onMetaData 33
 - onPlayStatus 33

F

- Flash Archive utility (FAR) 10
- FLVCheck utility 33

H

- HelloServer application 22
- HelloWorld application 3

I

- installing the server 3

L

- live service 12
- live video
 - metadata 61
 - publishing 54
 - subscribing 54
- log file 8, 45

M

- media player 30
- metadata
 - about 65
 - adding to live streams 29, 61
 - in stream transitions 43
 - properties 66
 - XMP 18, 29
- multipoint publishing 67

N

- NetConnection class
 - about 20
 - client property 47
 - status codes 26, 34
- netStatus events 34
- NetStream class
 - pause method 43
 - play method 28

O

- onBWCheck function 46
- onBWDone function 46
- onMetaData event handler 33
- onPlayStatus event handler 33
- onXMPData 29

P

- playlists 35, 36
- progressive download 30
- publishing
 - live video 54
 - multipoint 67

R

- RTMP 1

S

- Server-Side ActionScript
 - about 6
 - handling connections 25
 - packaging 10
 - trace statement 8
- shared objects 17, 70, 74
- starting the server 3

- Stream class 50
- stream not found error 34
- streaming services 12, 17, 18
- streams
 - buffering 43
 - detecting length 50
 - dynamic 38
 - pausing 43
 - playing 28
 - publishing 54
 - subscribing to 54
 - switching 41
- SWF files, verifying 77

U

- URIs 20
- UTF-8 encoding 7

V

- video data, accessing 18
- video player 17
- video snapshots 29
- vod service 12, 14

X

- XMP metadata 29