



Document Library

Die InterLake GmbH bietet Ihren Kunden in der Document Library Zugriff auf interessante Informationen. Wir stellen Ihnen Inhalte unserer Partner und aus frei zugänglichen Online Quellen zur Verfügung, die mit unserer Tätigkeit und der IT- und Medienbranche zu tun haben. Bitte beachten Sie, dass die Inhalte dem Copyright der jeweiligen Herausgeber unterliegen und diese Inhalte nicht ohne Referenz auf das jeweilige Copyright weitergegeben werden dürfen.

Weitere Informationen zu InterLake und weitere Dokumente finden Sie unter

www.interlake.net

Die InterLake Document Library steht den Nutzern der InterLake.Network zur Verfügung:



InterLake engagiert sich ehrenamtlich beim Münchener IT- und Medienverband FIWM, dem unser Geschäftsführer Sven Slazenger als Vorstandsmitglied angehört, sowie in der Macromedia ColdFusion User Group Central Europe, die unter unserer Leitung seit 1998 ein deutschsprachiges Forum für über 700 Macromedia Entwickler in Deutschland, Österreich und der Schweiz bietet.

Weitere Informationen zu diesen beiden Initiativen erhalten Sie bei Sven Slazenger (slazenger@interlake.net)

InterLake offers its customers and business partners access to an extensive document library. We are offering information from our partners and have also compiled freely available papers from the internet that we consider of value to you. Please be advised that the copyright belongs to the issuer of the information and that you may not distribute this content without referring to these copyrights.

Additional information about InterLake and more documents can be accessed free of charge at

www.interlake.net

The InterLake Document Library is available through the websites of the InterLake.Network:

InterLake is also an active member of the non-profit Munich IT- and Media Association FIWM. With InterLake CEO Sven Slazenger we provide one of the board members of the FIWM. We are also the founders (1998) of the Macromedia ColdFusion User Group Central Europe, the German language forum for Macromedia Developers from Austria, Germany and Switzerland and one of the largest Macromedia Communities worldwide.

For further information please contact Sven Slazenger (slazenger@interlake.net)



Home / DevNet / Flash Communication Server Development Center /

Flash Communication Server Article



Kristopher Schultz
Senior Interactive
Developer
[Mills/James Productions](#)

Taking Control of Connections

Once you've created applications with Macromedia Flash Communication Server MX, you may decide to have more control over all the connection requests your application is likely to receive. In this article, I take a look at two common ways of controlling connections: connection limits and password protections.

Limiting Connections

If you're running more than one application from Flash Communication Server, sooner or later you'll have to limit the number of connections each application uses, especially if your applications become popular. After all, you don't want your multiuser water balloon game to become inaccessible because your virtual dating application is hogging all of your server's connections. My recipe for limiting connections works with any communications application.

Let's start by looking at what really happens when a Macromedia Flash file connects to Flash Communication Server. Unless you specify otherwise, Flash Communication Server always accepts a connection request made by a client until it has no more connections left to offer. If you want to change this behavior, you need to provide the server with some rules to use when deciding whether it should allow or reject a connection. Writing a little bit of server-side ActionScript accomplishes this task.

Suppose you want no more than five users to connect to your application at one time. Create a new text file in your preferred text editor and save it as **main.asc**. The main.asc file allows you to add new capabilities to your communications application. Place the following code inside this file:

```
application.onConnect = function ( pClient ) {  
    if ( application.clients.length >= 5 ) {  
        application.rejectConnection(pClient);  
    } else {  
        application.acceptConnection(pClient);  
    }  
}
```

Here's what the code really does, line by line:

```
application.onConnect = function ( pClient ) {  
    ...  
}
```

`Application.onConnect` is an event that's triggered anytime a client has just made a temporary connection to the server. At this point the server must decide whether to allow the connection to continue or reject and close the connection. Building a function that attaches to this event is a convenient way for your server to follow some rules when making this decision.

```
if ( application.clients.length >= 5 ) {
```

Every application instance has an array called `clients`, which contains references to each client whose connection to the application has been accepted. Checking the `length` property of this array tells you how many clients are currently connected. By comparing this value to a number of your choosing, you can set a threshold above which some code will be executed. In this case, you are checking to see if there are five or more clients connected. Note that because the connection request that triggers the `onConnect` event hasn't been officially "approved" yet, it doesn't count as one of the clients in the `clients` array, even though it's taking up one of the server's available connections, at least temporarily.

```
application.rejectConnection(pClient);
```

This code is executed when the number of connections is at or above your limit of five. First it sends a status code to the client indicated by `pClient`, which looks like this:

```
"NetConnection.Connect.Rejected"
```

Then it immediately closes the temporary connection. Take note of the status code above, as you will refer to it later.

```
} else {  
    application.acceptConnection(pClient);  
}
```

This code is executed when the number of connections is below your limit of five. It simply allows the connection to continue and sends the following status code to the client:

```
"NetConnection.Connect.Success"
```

That's all the code you need for the `main.asc` file. Place this file inside your application's Flash Communication Server directory. The contents of `main.asc` are only referenced when the application first starts up, so if an instance of your application is already running on the server, restart it so your new script takes effect.

Now that you've given the server the power to decide when to allow a connection, let's give the client a way to gracefully react to the server's decision.

Inside your Macromedia Flash file write the following function to handle making a connection to the server. It's common practice to place functions like this on the first frame of your movie to ensure it'll be available when you need to call it:

```
function doConnect () {  
    vConn_nc = new NetConnection();  
    vConn_nc.connect("rtmp://yourflashcomserver.com/yourApp");  
    vConn_nc.onStatus = function ( pInfo ) {  
        if ( pInfo.code == "NetConnection.Connect.Success" ) {  
            gotoAndPlay("Start");  
        } else if ( pInfo.code == "NetConnection.Connect.Failed" ||  
            pInfo.code == "NetConnection.Connect.Rejected" )  
        {  
            gotoAndPlay("Access Denied");  
        }  
    }  
}
```

Here's what the code really does, line by line.

```
vConn_nc = new NetConnection();  
vConn_nc.connect("rtmp://yourflashcomserver.com/ourApp");
```

When the function is called, a new `NetConnection` object is created and an attempt is made to connect to your application.

```
vConn_nc.onStatus = function ( pInfo ) {
```

`NetConnection.onStatus` is an event that's triggered anytime the status of the connection changes. By building a function and attaching it to this event, your client file can react to these changes. Specifically, it can react to messages that result from the server's decision to accept or reject the connection. The information that the server provides is passed to the function as a special information object, which you've referred to using the parameter name `pInfo`.

```
if ( pInfo.code == "NetConnection.Connect.Success" ) {  
    gotoAndPlay("Start");
```

If the `code` property of the `pInfo` object indicates a successful connection, the client file jumps to a label on the main timeline called "Start" and the user can begin working with your application.

```
} Else if ( pInfo.code == "NetConnection.Connect.Failed" || pInfo.code ==  
"NetConnection.Connect.Rejected" ) {  
    gotoAndPlay("Access Denied");
```

Here you are actually responding to two different possible conditions. As I mentioned earlier, if the server reaches its overall connection limit, any attempt to connect will immediately fail and the client will see the "NetConnection.Connect.Failed" status code. If the server still has connections available but your application has reached its connection limit, as defined in your main.asc file, then the client will receive the "NetConnection.Connect.Rejected" status code. In either case, the client file jumps to a label on the main timeline called "Access Denied," where you could provide a message explaining that the maximum number of users has been reached and that the visitor should try again later.

That's all you need to do to limit the number of users who can connect to your application at any given time. That wasn't too complicated, was it? Now that you have that under your belt, let's explore another type of connection control: password protection.

Protecting with Passwords

In addition to controlling how many people can use your application at once, you might occasionally want to control *who* can use your application. One simple way to accomplish this is by asking users for a password before allowing them to connect. Let's look at how you can modify the connection control code you've already created so that it also checks for a password.

Start by changing the `doConnect()` function in your Macromedia Flash file. The modifications are shown in **bold**:

```
function doConnect ( pPassword ) {  
    vConn_nc = new NetConnection();  
    vConn_nc.connect("rtmp://yourflashcomserver.com/yourApp", pPassword);  
    vConn_nc.onStatus = function ( pInfo ) {  
        if ( pInfo.code == "NetConnection.Connect.Success" ) {  
            gotoAndPlay("Start");  
        } else if ( pInfo.code == "NetConnection.Connect.Failed" ||  
            pInfo.code == "NetConnection.Connect.Rejected" )
```

```

    {
        if ( pInfo.application.message == "wrong password" ) {
            gotoAndPlay("Wrong Password");
        } else {
            gotoAndPlay("No Connections");
        }
    }
}
}
}

```

Let's look at the modifications line by line:

```

function doConnect ( pPassword ) {
vConn_nc = new NetConnection();
vConn_nc.connect("rtmp://yourflashcomserver.com/yourApp", pPassword);

```

We modified this function so that you can now pass it a password string as a parameter. The parameter is now called `pPassword`. In your movie, you need to provide the user with a password input field and use the text of that field as your password string when triggering this function. The password is sent to the server by passing it as a parameter with your connection request.

```

vConn_nc.onStatus = function ( pInfo ) {
    if ( pInfo.code == "NetConnection.Connect.Success" ) {
        gotoAndPlay("Start");
    } else if ( pInfo.code == "NetConnection.Connect.Failed" ||
                pInfo.code == "NetConnection.Connect.Rejected" )
    {
        if ( pInfo.application.message == "wrong password" ) {
            gotoAndPlay("Wrong Password");
        } else {
            gotoAndPlay("No Connections");
        }
    }
}
}

```

You still check for a successful connection the same way as before. But this time when the client receives a "Failed" or "Rejected" status code, you have to determine whether the failure was due to a lack of available connections or because the user provided the wrong password.

Do this by checking for a custom message in the `application` parameter of the `pInfo` object sent from the server. If the message is "wrong password," send the main timeline to a label called "Wrong Password" to inform the user of their mistake. Otherwise, you can assume that the connection wasn't made because of a lack of available connections. In that case send the main timeline to a label called "No Connections."

But how exactly does the server send a custom message like "wrong password"? In your main.asc file, make the following modifications shown in **bold**:

```

application.onConnect = function ( pClient, pPassword ) {
    if ( application.clients.length > 5 ) {
        application.rejectConnection(pClient);
    } else if ( pPassword != "abc123" ) {
        var vError = new Object();
        vError.message = "wrong password";
        application.rejectConnection(pClient, vError);
    } else {

```

```
        application.acceptConnection(pClient);
    }
}
```

Let's see what these modifications accomplish:

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

At the beginning of your function definition you've added a parameter called `pPassword` so that you can use it to address the password string that the client file would be sending when it tries to connect.

```
If ( application.clients.length > 5 ) {
    application.rejectConnection(pClient);
```

As before, you still immediately reject the connection if the connection limit has been exceeded.

```
} Else if ( pPassword != "abc123" ) {
    var vError = new Object();
    vError.message = "wrong password";
    application.rejectConnection(pClient, vError);
```

This code determines where your password check occurs. If the user's password doesn't match the password you're looking for-in this case, "abc123"-you need to reject the connection. In addition to simply rejecting the connection, you want to provide a custom message so that the client knows the rejection resulted from an incorrect password. Do this by creating an object, called `vError`. Add a parameter to the object called `message` and assign it the value "wrong password." By adding the `vError` object as a second parameter to your `rejectConnection()` call, it will return to the client and be interpreted by the client code you added earlier.

```
} Else {
    acceptConnection(pClient);
}
```

Finally, if the connection attempt passes both the connection limit test and the password check, you allow the connection to continue.

That's it. With just a little bit of server-side and client-side code, you've now got two simple, yet powerful, ways to control connections to your applications.

About the author

Kristopher Schultz is currently a Senior Interactive Developer with [Mills/James Productions](#) in Columbus, Ohio, where he and his colleagues are creating award winning CD-ROM, kiosk, and web projects for a wide range of clients. Specializing in Director and Macromedia Flash development, Kris has been a devoted Macromedia user, teacher, and evangelist for nearly 8 years. Kris is also the president of the [Central Ohio Macromedia User Group](#) and encourages everyone to get involved with their local [Macromedia User Group](#) and other local design and development organizations. You can contact Kris at kristopher@writeme.com.



Company | [Site Map](#) | [Privacy](#) | [Contact Us](#) | [Accessibility](#) | [Report Piracy](#)

©1995-2003 Macromedia, Inc. [All rights reserved.](#)

Use of this website signifies your agreement to the [Terms of Use](#).

Powered by Google™

